Нижегородский государственный университет им. Н.И.Лобачевского

Межфакультетская магистратура по системному и прикладному программированию
для многоядерных компьютерных систем

Образовательный комплекс
"Введение в методы параллельного программирования"

Раздел "Параллельные методы умножения матриц"

Лабораторная работа 2: Параллельные алгоритмы матричного умножения

Разработчик: Е.А.Козинов

Нижний Новгород
2007

Содержание

Операция умножения матриц является одной из основных задач матричных вычислений. В данной лабораторной работе рассматриваются последовательный алгоритм матричного умножения, параллельный алгоритм, основанный на блочной схеме разделения данных, а также оптимизированный, с точки зрения использования кэш, вариант этого алгоритма.

Цель лабораторной работы

Целью данной лабораторной работы является разработка параллельной программы, которая выполняет умножение двух квадратных матриц. Выполнение лабораторной работы включает:

- Упражнение 1 – Определение задачи матричного умножения,
- Упражнение 2 – Реализация последовательного алгоритма матричного умножения,
- Упражнение 3 – Разработка параллельного алгоритма матричного умножения,
- Упражнение 4 - Реализация параллельного алгоритма умножения матриц.

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

При выполнении лабораторной работы предполагается знание раздела 4 "Параллельное программирование на основе OpenMP", раздела 6 "Принципы разработки параллельных методов" и раздела 8 "Параллельные алгоритмы матричного умножения" учебных материалов курса. Кроме того, предполагается, что выполнена лабораторная работа 1 "Параллельные алгоритмы матрично-векторного умножения".

Упражнение 1 – Определение задачи матричного умножения

Умножение матрицы *A* размера $m \times n$ и матрицы *B* размера $n \times l$ приводит к получению матрицы *C* размера $m \times l$, каждый элемент которой определяется в соответствии с выражением:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \le i < m, 0 \le j < l. \tag{2.1}$$

Как следует из (2.1) , каждый элемент результирующей матрицы *C* есть скалярное произведение соответствующих строки матрицы *A* и столбца матрицы *B* (рис. 2.1):

$$c_{ij} = \left(a_i, b_j^T\right), a_i = \left(a_{i0,} a_{i1}, ..., a_{in-1}\right), b_j^T = \left(b_{0j}, b_{1j}, ..., b_{n-1j}\right)^T. \tag{2.2}$$

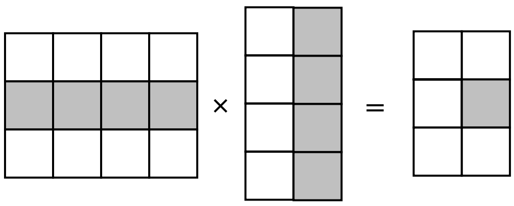

Рис. 2.1. Элемент результирующей матрицы *C* – это результат скалярного умножения соответствующих строки матрицы *A* и столбца матрицы *B*

Так, например, при умножении матрицы *A*, состоящей из 3 строк и 4 столбцов на матрицу *B* из 4 строк и 2 столбцов, получается матрица *C* из 3 строк и 2 столбцов:

$$\begin{pmatrix} 3 & 2 & 0 & -1 \\ 5 & -2 & 1 & 1 \\ 1 & 0 & -1 & -1 \end{pmatrix} \times \begin{pmatrix} 1 & -1 \\ 2 & 5 \\ -3 & 2 \\ 7 & 4 \end{pmatrix} = \begin{pmatrix} 0 & 3 \\ 5 & -9 \\ -3 & -7 \end{pmatrix}$$

Рис. 2.2. Пример умножения матриц

Тем самым, получение результирующей матрицы *C* предполагает повторение $m \times l$ однотипных операций по умножению строк матрицы *A* и столбцов матрицы *B*. Каждая такая операция включает умножение элементов строки и столбца матриц и последующее суммирование полученных произведений.

Псевдокод для представленного алгоритма умножения матрицы на вектор может выглядеть следующим образом (здесь и далее предполагается, что матрицы, участвующие в умножении, квадратные, то есть имеют размерность $Size \times Size$):

```
// Serial algorithm of matrix multiplication
for (i=0; i<Size; i++) {
  for (j=0; j<Size; j++) {
    MatrixC[i][j] = 0;
    for (k=0; k<Size; k++) {
      MatrixC[i][j] = MatrixC[i][j] + MatrixA[i][k]*MatrixB[k][j];
    }
  }
}
```

Упражнение 2 – Реализация последовательного алгоритма матричного умножения

При выполнении этого упражнения необходимо реализовать последовательный алгоритм матричного умножения. Начальный вариант будущей программы представлен в проекте *SerailMatrixMult*, который содержит часть исходного кода и в котором заданы необходимые параметры проекта. В ходе выполнения упражнения необходимо дополнить имеющийся вариант программы операциями ввода размера матриц, задание исходных данных, умножения матриц и вывода результатов.

Задание 1 – Открытие проекта SerialMatrixMult

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

После открытия проекта в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **SerialMM.cpp**, как это показано на рис. 2.3. После этих действий код, который предстоит в дальнейшем расширить, будет открыт в рабочей области Visual Studio.

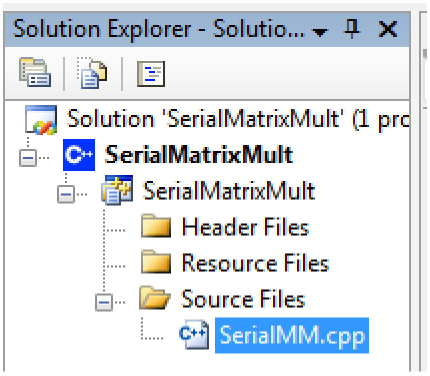

Рис. 2.3. Открытие файла SerialMM.cpp

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

Рассмотрим переменные, которые используются в основной функции (*main*) нашего приложения. Первые две из них (*pAMatrix* и *pBMatrix*) – это, соответственно, матрицы которые участвуют в матричном умножении в качестве аргументов. Третья переменная *pCMatrix* – матрица, которая должна быть получена в результате умножения. Переменная *Size* определяет размер матриц (предполагаем, что все матрицы квадратные, имеют размерность *Size*×*Size*).

```
  double* pAMatrix;  // The first argument of matrix multiplication
  double* pBMatrix;  // The second argument of matrix multiplication
  double* pCMatrix;  // The result matrix
  int Size;          // Size of matricies
```

Как и при разработке алгоритмов умножения матрицы на вектор, для хранения матриц используются одномерные массивы, в которых матрицы хранятся построчно. Таким образом, элемент, расположенный на пересечении *i*-ой строки и *j*-ого столбца матрицы, в одномерном массиве имеет индекс *i*Size+j*.

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

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

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

Сразу после запуска кода, в командной консоли появится сообщение: "`Serial matrix multiplication program`". Для того, чтобы завершить выполнение программы, нажмите любую клавишу.

Задание 2 – Ввод размеров матриц

Для задания исходных данных последовательного алгоритма матричного умножения реализуем функцию *ProcessInitialization*. Эта функция предназначена для определения размера матриц, выделения памяти для исходных матриц *pAMatrix* и *pBMatrix*, и матрицы-результата умножения *pCMatrix*, а также для задания значений элементов исходных матриц. Значит, функция должна иметь следующий интерфейс:

```
// Function for memory allocation and initialization of matrix elements
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, int &Size);
```

На первом этапе необходимо определить размер матриц (задать значение переменной *Size*). В тело функции *ProcessInitialization* добавьте выделенный фрагмент кода:

```
// Function for memory allocation and initialization of matrices’ elements
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, int &Size) {
  // Setting the size of matricies
  printf("\nEnter size of matricies: ");
  scanf("%d", &Size);
  printf("\nChosen matricies’ size = %d", Size);
}
```

Пользователю предоставляется возможность ввести размер матриц, который затем считывается из стандартного потока ввода *stdin* и сохраняется в целочисленной переменной *Size*. Далее печатается значение переменной *Size* (рис. 2.4).

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

```
void main() {
  double* pAMatrix;  // The first argument of matrix multiplication
  double* pBMatrix;  // The second argument of matrix multiplication
  double* pCMatrix;  // The result matrix
  int Size;          // Size of matricies
  double start, finish;
  double duration;

  printf ("Serial matrix multiplication program\n");
  ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);
  getch();
}
```

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

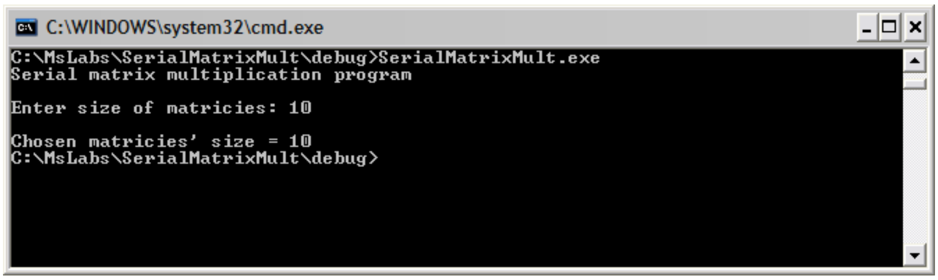

Рис. 2.4. Задание размера объектов

Как и лабораторной работе 1, выполним контроль правильности ввода. Организуем проверку размера матриц и, в случае ошибки (заданный размер является нулевым или отрицательным), продолжим запрашивать размер матриц до тех пор, пока не будет введено положительное число. Для реализации такого поведения поместим фрагмент кода, который производит ввод размера матриц, в цикл с постусловием:

```
  // Setting the size of matricies
  do {
    printf("\nEnter size of matricies: ");
    scanf("%d", &Size);
    printf("\nChosen matricies size = %d\n", Size);
    if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  } while (Size <= 0);
```

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

Задание 3 – Ввод данных

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

```
// Function for memory allocation and initialization of matrices' elements
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, int &Size) {
  // Setting the size of matricies
  do {
    <…>
  }
  while (Size <= 0);

  // Memory allocation
  pAMatrix = new double [Size*Size];
  pBMatrix = new double [Size*Size];
  pCMatrix = new double [Size*Size];
}
```

Далее необходимо задать значения всех элементов исходных объектов: матриц *pAMatrix*, *pBMatrix* и *pCMatrix*. Значения элементов результирующей матрицы до выполнения матричного умножения следует установить равными 0. Для задания значений элементов матриц *A* и *B* реализуем еще одну функцию *DummyDataInitialization*. Интерфейс и реализация этой функции представлены ниже:

```
// Function for simple initialization of matrix elements
void DummyDataInitialization(double* pAMatrix, double* pBMatrix, int Size){
  int i, j;  // Loop variables

  for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++) {
      pAMatrix[i*Size+j] = 1;
      pBMatrix[i*Size+j] = 1;
    }
  }
}
```

Как видно из представленного фрагмента кода, данная функция осуществляет задание элементов матриц простым образом: значения всех элементов матриц равны 1. То есть в случае, когда пользователь выбрал размер объектов, равный 4, будут определены следующие матрица и вектор:

$$pAMatrix = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix},\ pBMatrix = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

(задание данных при помощи датчика случайных чисел будет рассмотрено в задании 6).

Вызов функции *DummyDataInitialization* и процедуру заполнения результирующей матрицы нулями необходимо выполнить после выделения памяти внутри функции *ProcessInitialization*:

```
// Function for memory allocation and initialization of matrix elements
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, int &Size) {
  // Setting the size of matricies
  do {
    <…>
  }
  while (Size <= 0);

  // Memory allocation
  <…>

  // Initialization of matrix elements
  DummyDataInitialization(pAMatrix, pBMatrix, Size);
  for (int i=0; i<Size*Size; i++) {
    pCMatrix[i] = 0;
  }
}
```

Для контроля ввода данных воспользуемся функцией форматированного вывода объектов *PrintMatrix*, которая была разработана при выполнении лабораторной работы 1 и текст которой уже имеется в проекте (подробнее о функции *PrintMatrix* см. задание 3, упражнение 2 лабораторной работы 1). Добавим вызов этой функций для печати объектов *pAMatrix* и *pBMatrix* в основную функцию приложения:

```
  // Memory allocation and initialization of matrix elements
  ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

  // Matrix output
  printf ("Initial A Matrix \n");
  PrintMatrix(pAMatrix, Size, Size);
  printf("Initial B Matrix \n");
  PrintMatrix(pBMatrix, Size, Size);
```

Скомпилируйте и запустите приложение. Убедитесь в том, что ввод данных происходит по описанным правилам (рис. 2.5). Выполните несколько запусков приложения, задавайте различные размеры матриц.

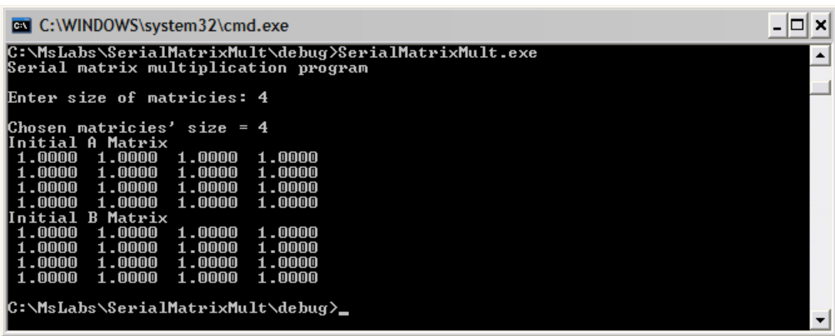

Рис. 2.5. Результат работы программы при завершении задания 3

Задание 4 – Завершение процесса вычислений

Перед выполнением матричного умножения сначала разработаем функцию для корректного завершения процесса вычислений. Для этого необходимо освободить память, выделенную динамически в процессе выполнения программы. Реализуем соответствующую функцию *ProcessTermination*. Память выделялась для хранения исходных матриц *pAMatrix* и *pBMatrix*, а также для хранения матрицы - результата умножения *pCMatrix*. Следовательно, эти объекты необходимо передать в функцию *ProcessTermination* в качестве аргументов:

```
// Function for computational process termination
void ProcessTermination (double* pAMatrix, double* pBMatrix,
  double* pCMatrix) {
  delete [] pAMatrix;
  delete [] pBMatrix;
  delete [] pCMatrix;
}
```

Вызов функции *ProcessTermination* необходимо выполнить перед завершением той части программы, которая выполняет умножение матриц:

```
  // Memory allocation and initialization of matrix elements
  ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

  // Matrix output
  printf ("Initial A Matrix \n");
  PrintMatrix(pAMatrix, Size, Size);
  printf("Initial B Matrix \n");
  PrintMatrix(pBMatrix, Size, Size);

  // Computational process termination
  ProcessTermination(pAMatrix, pBMatrix, pCMatrix);
```

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

Задание 5 – Реализация матричного умножения

Выполним теперь разработку основной вычислительной части программы. Для выполнения умножения матриц реализуем функцию *SerialResultCalculation*, которая принимает на вход исходные матрицы *pAMatrix* и *pBMatrix*, размер этих матриц *Size*, а также указатель на результирующую матрицу *pCMatrix*.

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

```
// Function for matrix multiplication
void SerialResultCalculation(double* pAMatrix, double* pBMatrix,
  double* pCMatrix, int Size) {
  int i, j, k;  // Loop variables
  for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++) {
      for (k=0; k<Size; k++) {
        pCMatrix[i*Size+j] += pAMatrix[i*Size+k]*pBMatrix[k*Size+j];
      }
    }
  }
}
```

Выполним вызов функции вычисления матричного произведения из основной программы. Для контроля правильности выполнения умножения распечатаем результирующую матрицу:

```
  // Memory allocation and initialization of matrix elements
  ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

  // Matrix output
  printf ("Initial A Matrix \n");
  PrintMatrix(pAMatrix, Size, Size);
  printf("Initial B Matrix \n");
```

```
  PrintMatrix(pBMatrix, Size, Size);

  // Matrix multiplication
  SerialResultCalculation(pAMatrix, pBMatrix, pCMatrix, Size);

  // Printing the result matrix
  printf ("\n Result Matrix: \n");
  PrintMatrix(pCMatrix, Size, Size);

  // Computational process termination
  ProcessTermination(pAMatrix, pBMatrix, pCMatrix);
```

Скомпилируйте и запустите приложение. Проанализируйте результат работы алгоритма умножения матриц. Если алгоритм реализован правильно, то в результате должна быть получена матрица, значения всех элементов которой равны порядку этой матрицы (рис. 2.6).

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \\ 3 & 3 & 3 \end{pmatrix}$$

Рис. 2.6. Результат матричного умножения

Проведите несколько вычислительных экспериментов, изменяя размеры объектов.

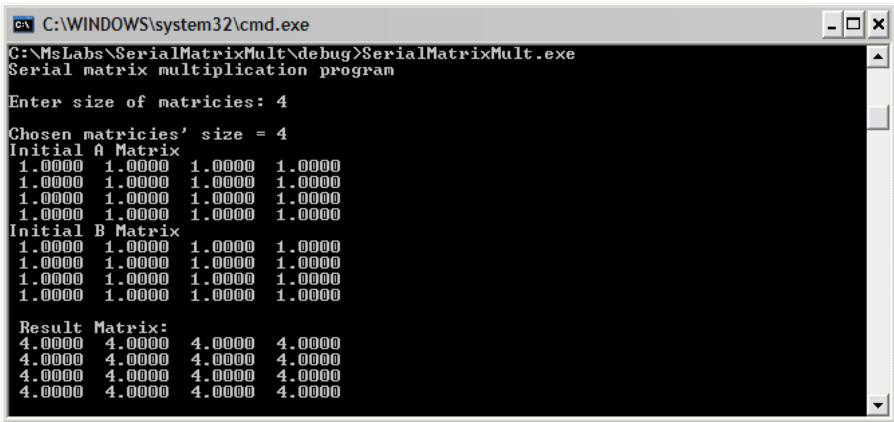

Рис. 2.7. Результат выполнения матрично-векторного умножения

Задание 6 – Проведение вычислительных экспериментов

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

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

Будем вызывать эту функцию вместо ранее разработанной функции *DummyDataInitialization*, которая генерировала такие данные, что можно было легко проверить правильность работы алгоритма:

```
// Function for memory allocation and initialization of matrix elements
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, int &Size) {
  // Setting the size of matricies
  do {
    <…>
  }
  while (Size <= 0);

  // Memory allocation
  <…>

  // Random initialization of matrix elements
  RandomDataInitialization(pAMatrix, pBMatrix, Size);
  for (int i=0; i<Size*Size; i++) {
    pCMatrix[i] = 0;
  }
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что данные генерируются случайным образом.

Для определения времени работы последовательного алгоритма добавьте в программу вызовы функции, которая позволяет узнать время работы программы или её части. Как и ранее, будем использовать функцию, описанную в лабораторной работе 1 "Параллельные алгоритмы матрично-векторного умножения" (см. лабораторная работа 1, упражнение 2, задание 6). Эта функция возвращает текущее значение времени в секундах, которое отсчитывается от фиксированного момента в прошлом:

```
double GetTime();
```

Добавим в программный код вычисление и вывод времени непосредственного выполнения умножения матрицы на вектор, для этого поставим замеры времени до и после вызова функции *SerialResultCalculation*:

```
  // Matrix multiplication
  Start = GetTime();
  SerialResultCalculation(pAMatrix, pBMatrix, pCMatrix, Size);
  Finish = GetTime();
  Duration = (Finish-Start);

  // Printing the result matrix
  printf ("\n Result Matrix: \n");
  PrintMatrix(pCMatrix, Size, Size);

  // Printing the time spent by matrix multiplication
  printf("\n Time of execution: %f\n", Duration);
```

Скомпилируйте и запустите приложение. Для проведения вычислительных экспериментов с большими объектами, отключите печать матриц (закомментируйте соответствующие строки кода). Проведите вычислительные эксперименты, результаты занесите в таблицу:

Таблица 2.1. Время выполнения последовательного алгоритма умножения матриц

Номер теста	Размер матрицы	Время работы (сек)
1	10	
2	100	
3	500	
4	1000	
5	1500	
6	2000	
7	2500	

8	3000	

Для построения теоретических оценок времени вычислений воспользуемся подходом, изложенным в Главе 8 учебных материалов курса. Время выполнения алгоритма складывается из времени, которое тратится непосредственно на вычисления, и времени, необходимого на чтение данных из оперативной памяти в кэш процессора.

Согласно алгоритму вычисления матричного произведения, изложенному в упражнении 1, получение результирующей матрицы предполагает повторение *Size×Size* однотипных операций по умножению строк матрицы *pAMatrix* и столбцов матрицы *pBMatrix*. Каждая такая операция включает умножение элементов строки и столбца (*Size* операций) и последующее суммирование полученных произведений (*Size-1* операций). Как результат, время необходимое непосредственно на матричное умножение можно определить при помощи соотношения:

$$T_{calc} = n^2 \cdot (2n-1) \cdot \tau \,, \tag{2.3}$$

где n есть размерность матрицы, τ - время выполнения одной базовой вычислительной операции.

Теперь оценим объем данных, которые необходимо прочитать из оперативной памяти в кэш вычислительного элемента в случае, когда размер матриц настолько велик, что они одновременно не могут быть помещены в кэш. Матрицы-аргументы(*pAMatrix, pBMatrix*) считываются в оперативную память несколько раз. Действительно, для вычисления одного элемента результирующей матрицы необходимо прочитать в кэш элементы одной строки матрицы *pAMatrix* и одного столбца матрицы *pBMatrix*. Поскольку матрицы хранятся в памяти построчно, а чтение в кэш происходит линейками по 64 байта (8 элементов типа double), то для вычисления одного элемента результирующей матрицы необходимо прочитать *Size+8Size* элементов. Всего необходимо вычислить $Size^2$ элементов результирующей матрицы, значит общий объем данных, необходимых для чтения из оперативной памяти в кэш, $Size^3+8Size^3$. Для построения оценок сверху можно предположить, что при вычислении каждого очередного элемента происходит повторное считывание строки и столбца исходных матриц в кэш, так как их совокупный объем может быть настолько велик, что при вычислении скалярного произведения и считывании в кэш «последних» элементов строки и столбца, происходит вытеснение «первых» элементов. Матрица *pCMatrix* считывается один раз, следовательно, количество загружаемых элементов составляет $Size^2$.

Таким образом, формула для оценки времени выполнения последовательного алгоритма умножения матриц может быть представлена следующим образом:

$$T_1 = n^2 \cdot (2n-1) \cdot \tau + \frac{8 \cdot \left(n^2 + n^3 + 8n^3\right)}{\beta} = n^2 \cdot (2n-1) \cdot \tau + \frac{8 \cdot \left(n^2 + 9n^3\right)}{\beta} \tag{2.4}$$

Метод оценивания параметров τ и β был подробно описан в лабораторной работе 1 "Параллельные алгоритмы матрично-векторного умножения". Выполните вычислительные для определения значений этих параметров.

Заполните таблицу сравнения реального времени выполнения программы матричного умножения со временем, которое может быть получено по формуле (2.4).

Таблица 2.2. Сравнение реального времени выполнения последовательного алгоритма умножения матриц со временем, вычисленным теоретически

Время выполнения одной операции τ (сек):			
Номер теста	Размер матрицы	Время работы (сек)	Теоретическое время (сек)
1	10		
2	100		
3	500		
4	1000		
5	1500		
6	2000		
7	2500		
8	3000		

Упражнение 3 – Разработка параллельного алгоритма матричного умножения

Для разработке параллельного алгоритма матричного умножения воспользуемся блочной схемой представления матриц. Выполним более подробное рассмотрение данного способа организации вычислений.

Определение подзадач

Блочная схема разбиения матриц подробно рассмотрена в подразделе 7.2 лекционного материала и в упражнении 3 лабораторной работы 1. При таком способе разделения данных исходные матрицы *A*, *B* и результирующая матрица *C* представляются в виде наборов блоков. Для более простого изложения следующего материала будем предполагать далее, что все матрицы являются квадратными размера $n\times n$, количество блоков по горизонтали и вертикали являются одинаковым и равным *q* (т.е. размер всех блоков равен $k\times k$, $k=n/q$). При таком представлении данных операция матричного умножения матриц *A* и *B* в блочном виде может быть представлена в виде:

$$\begin{pmatrix} A_{00} & A_{01} & \dots & A_{0q-1} \\ \dots & \dots & \dots & \dots \\ A_{q-10} & A_{q-11} & \dots & A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} & \dots & B_{0q-1} \\ \dots & \dots & \dots & \dots \\ B_{q-10} & B_{q-11} & \dots & B_{q-1q-1} \end{pmatrix} = \begin{pmatrix} C_{00} & C_{01} & \dots & C_{0q-1} \\ \dots & \dots & \dots & \dots \\ C_{q-10} & C_{q-11} & \dots & C_{q-1q-1} \end{pmatrix},$$

где каждый блок C_{ij} матрицы *C* определяется в соответствии с выражением

$$C_{ij} = \sum_{s=0}^{q-1} A_{is} B_{sj} \ . \tag{2.5}$$

При блочном разбиении данных для определения базовых подзадач естественным представляется взять за основу вычисления, выполняемые над матричными блоками. С учетом сказанного определим базовую подзадачу как процедуру вычисления всех элементов одного из блоков матрицы C.

Для выполнения всех необходимых вычислений базовым подзадачам должны быть доступны соответствующие наборы строк матрицы A и столбцов матрицы B.

Широко известны параллельные алгоритмы умножения матриц, основанные на блочном разделении данных, ориентированные на многопроцессорные вычислительные системы с распределенной памятью. К числу алгоритмов, реализующих описанный подход, относятся *алгоритм Фокса* (*Fox*) и *алгоритм Кэннона* (*Cannon*). Отличие этих алгоритмов состоит в последовательности передачи матричных блоков между процессорами вычислительной системы.

При выполнении параллельных алгоритмов на системах с общей памятью передачи сообщений отсутствуют и отличия в методах могут состоять только в порядке выполнения операций перемножения матричных блоков.

Выделение информационных зависимостей

Итак, за основу параллельных вычислений для матричного умножения при блочном разделении данных принят подход, при котором базовые подзадачи отвечают за вычисления отдельных блоков матрицы *C* и при этом в подзадачи на каждой итерации расчетов обрабатывают только по одному блоку исходных матриц A и B. Для нумерации подзадач будем использовать индексы размещаемых в подзадачах блоков матрицы *C*, т.е. подзадача *(i,j)* отвечает за вычисление блока C_{ij} – тем самым, набор подзадач образует квадратную решетку, соответствующую структуре блочного представления матрицы *C*.

Как уже отмечалось выше, для вычисления блока результирующей матрицы поток должен выполнить умножение горизонтальной полосы матрицы *A* на вертикальную полосу матрицы *B*. Организуем поблочное умножение полос. На каждой итерации *i* алгоритма *i*-ый блок полосы матрицы *A* умножается на *i*-ый блок полосы матрицы *B*, результат умножения блоков прибавляется к блоку результирующей матрицы (рис. 2.8). Количество итераций определяется размером блочной решетки.

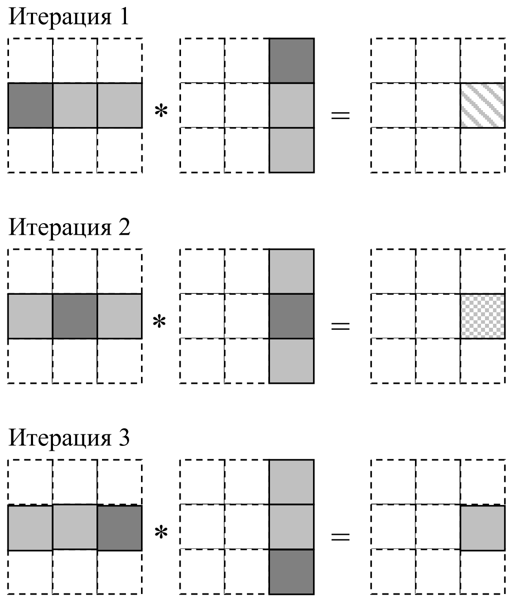

Рис. 2.8. Схема организации блочного умножения полос

Масштабирование и распределение подзадач по процессорам

В рассмотренной схеме параллельных вычислений количество блоков может варьироваться в зависимости от выбора размера блоков – эти размеры могут быть подобраны таким образом, чтобы общее количество базовых подзадач совпадало с числом вычислительных элементов p. Так, например, в наиболее простом случае, когда число вычислительных элементов представимо в виде $p=\delta^2$ (т.е. является полным квадратом) можно выбрать количество блоков в матрицах по вертикали и горизонтали равным δ (т.е. $q=\delta$). Такой способ определения количества блоков приводит к тому, что объем вычислений в каждой подзадаче является одинаковым и, тем самым, достигается полная балансировка вычислительной нагрузки между вычислительными элементами. В случае, когда число вычислительных элементов не является полным квадратом, число базовых подзадач $\pi=q\cdot q$ должно быть, по крайней мере, кратно числу вычислительных элементов.

Упражнение 4 – Реализация параллельного алгоритма умножения матриц

При выполнении этого упражнения будет выполнена реализация разработанного в упражнении 4 параллельного алгоритма блочного умножения матриц. При работе с этим упражнением Вы

- На практике познакомитесь со схемой организации матричных вычислений на основе блочного разделения данных,
- Получите опыт разработки более сложных параллельных программ,
- Познакомитесь с библиотекой функций OpenMP.

Задание 1 – Открытие проекта ParallelMatrixMult

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

После того, как Вы открыли проект, в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **ParallelMM.cpp**, как это показано на рисунке (2.9). После этих действий код, который вам предстоит модифицировать, будет открыт в рабочей области Visual Studio.

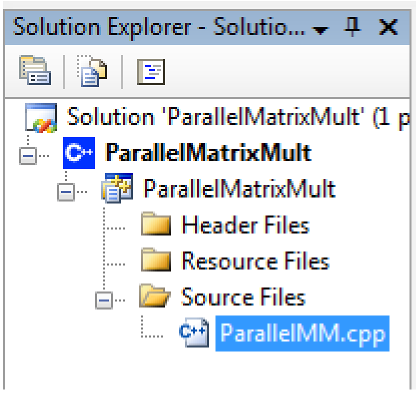

Рис. 2.9. Открытие файла **ParallelMM.cpp** с использованием Solution Explorer

В файле **ParallelMM.cpp** расположена главная функция (*main*) будущего параллельного приложения:

```
void main(int argc, char* argv[]) {
  double* pAMatrix;  // The first argument of matrix multiplication
  double* pBMatrix;  // The second argument of matrix multiplication
  double* pCMatrix;  // The result of matrix multiplication
  int Size;          // Sizes of matricies

  // Data initialization
  ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

  printf("Parallel matrix multiplication program\n");

  // Program termination
  ProcessTermination(pAMatrix, pBMatrix, pCMatrix, Size);
}
```

Также в файле **ParallelMM.cpp** расположены функции, перенесенные сюда из проекта, содержащего последовательный алгоритм умножения матриц: *DummyDataInitialization*, *RandomDataInitialization*, *SerialResultCalculation*, *PrintMatrix, ProcessInitialization, ProcessTermination* (подробно о назначении этих функций рассказывается в упражнении 2 данной лабораторной работы). Эти функции можно будет использовать и в параллельной программе.

Скомпилируйте и запустите приложение стандартными средствами Visual Studio. Убедитесь в том, что в командную консоль выводится приветствие: `"Parallel matrix multiplication program"`.

Задание 2 – Задание количества потоков

Для реализации алгоритма умножения матриц при блочном разбиении данных необходимо, чтобы количество параллельных потоков являлось полным квадратом. В разрабатываемом примере мы будем использовать 4 потока (определите переменную *ThreadsNum* и установите ее значение равным 4 – значение данной переменной должно переустанавливаться при изменении необходимого числа потоков). Определение числа потоков «вручную» до начала выполнения программы гарантирует корректную работу приложения на вычислительной системе с любым количеством доступных вычислительных элементов. Однако, если в вычислительной системе больше вычислительных элементов, чем определено параллельных потоков, эффективное использование ресурсов не может быть достигнуто.

Для определения количества параллельных потоков может быть использована функция *omp_set_num_threads* библиотеки OpenMP (функция должна быть вызвана в последовательном участке программы):

```
   int omp_set_num_threads (int NumThreads);
```

Добавьте вызов функции *omp_set_num_threads* в код функции *ParallelResultCalculation*. После этого объявите параллельную секцию, в которой разместите печать некоторого произвольного текста. Количество напечатанных строк должно совпадать с заданным количеством параллельных потоков.

```
void ParallelResultCalculation (double* pAMatrix, double* pBMatrix,
  double* pCMatrix, int Size) {
```

```
  int ThreadNum = 4;
  omp_set_num_threads(ThreadNum);
#pragma omp parallel
  {
    printf(“Hello!\n”);
  } // pragma omp parallel
}
```

Откомпилируйте и запустите программу. В нашем примере количество потоков было установлено равным четырем. В параллельной секции выполнялась печать строки “Hello!”. Так как потоков четыре, данное сообщение должно быть напечатано четыре раза. Результат работы программы представлен на рис. 2.10.

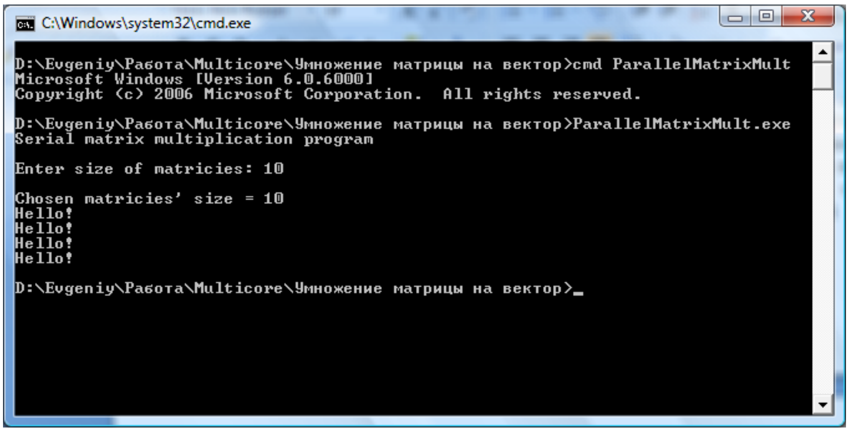

Рис. 2.10. Результат выполнения программы, в которой количество потоков установлено равным четырем

Задание 3 – Получение номеров потоков

Чтобы корректно определять блоки данных, над которыми поток должен выполнять вычисления, в программе необходимо иметь уникальный идентификатор потока. В качестве такого идентификатора может выступать номер потока. В библиотеке OpenMP существует функция для определения номера потока:

```
int omp_get_thread_num (void);
```

Определите переменную для хранения уникального номера потока *ThreadID*. При объявлении параллельной секции эта переменная должна быть определена как *private*, т. е. для этой переменной в каждом потоке должна быть создана локальная копия. Положения блоков матрицы и вектора, которые должны обрабатываться потоком, определяются в зависимости от значения переменной *ThreadID*.

Добавьте вызов функции, определяющей идентификатор потока, в код функции *ParallelResultCalculation*. Распечатайте идентификатор потока во всех потоках параллельной секции:

```
void ParallelResultCalculation (double* pAMatrix, double* pBMatrix,
double* pCMatrix, int Size) {
  int ThreadNum = 4;
  omp_set_num_threads(ThreadNum);
#pragma omp parallel
  {
    int ThreadID = omp_get_thread_num();
    printf(“Thread ID: %d\n”, ThreadID);
  } // pragma omp parallel
}
```

Добавьте вызов функции ParallelResultCalculation в основную функцию программы:

```
void main(int argc, char* argv[]) {
  …
  // Data initialization
  ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);
```

```
  ParallelResultCalculation(pAMatrix, pBMatrix, pCMatrix, Size);

  // Program termination
  ProcessTermination(pAMatrix, pBMatrix, pCMatrix, Size);
}
```

Откомпилируйте и запустите полученное приложение. Так как при запуске программы использовались четыре потока, то результат работы должен иметь вид, представленный на рис. 2.11 (однако следует помнить, что в зависимости от условий выполнения, порядок вывода сообщений параллельно выполняемых потоков может быть различным):

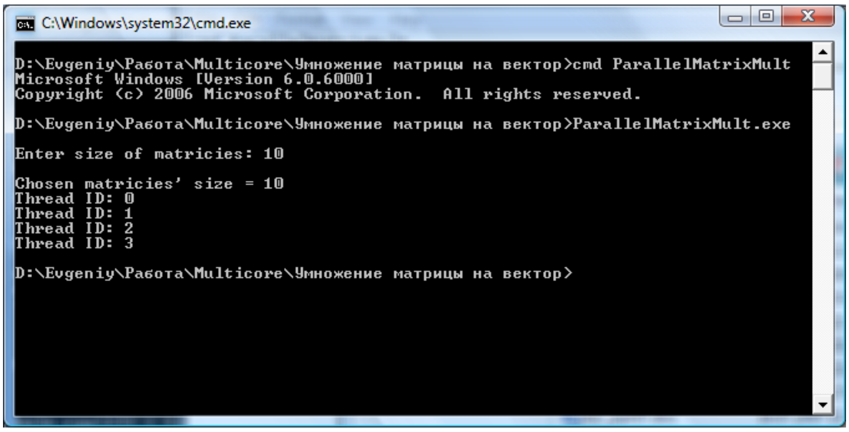

Рис. 2.11. Результат выполнения программы, использующей 4 потока

Задание 4 – Распределение блоков матриц по потокам

Как следует из описания параллельного алгоритма, число потоков должно являться полным квадратом для того, чтобы потоки можно было представить в виде двумерной квадратной решетки. Определим размер «решетки потоков» *GridSize* и размер матричного блока *BlockSize*.

Для запоминания координат положения потока в решетке потоков определим переменные *RowIndex* и *ColIndex*. Номер строки потоков, в котором расположен данный поток, есть результат целочисленного деления идентификатора потока на размер решетки потоков. Номер столбца, в котором расположен поток, есть остаток от деления идентификатора потока на размер решетки потоков.

Измените код функции *ParallelResultCalculation* таким образом, чтобы каждый поток параллельной секции печатал на экране не только свой идентификатор, но и положение в решетке потоков:

```
void ParallelResultCalculation (double* pAMatrix, double* pBMatrix,
double* pCMatrix, int Size) {
  int ThreadNum = 4;
  int GridSize = int (sqrt((double)ThreadNum));
  int BlockSize = Size/GridSize;

  printf("GridSize : (%d, %d)\n", GridSize , GridSize);
  printf("BlockSize: %d \n", BlockSize);

  omp_set_num_threads(ThreadNum);
#pragma omp parallel
  {
    int ThreadID = omp_get_thread_num();
    int RowIndex = ThreadID/GridSize;
    int ColIndex = ThreadID%GridSize;
    printf ("ThreadID (%d), position in grid: (%d, %d)\n", ThreadID,
      RowIndex, ColIndex);
  } // pragma omp parallel
}
```

Откомпилируйте и запустите приложение. Результат работы должен выглядеть так, как показано на рис.2.12:

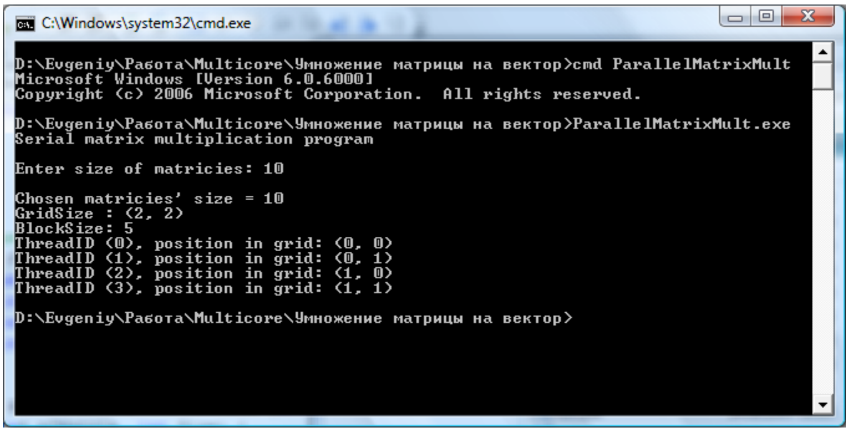

Рис. 2.12. Результат выполнения программы, вычисляющей размеры решетки потоков и блоков, а также положение потоков в решетке потоков

Задание 5 – Реализация блочного умножения матриц

Для выполнения операции матричного умножения каждый поток вычисляет элементы блока результирующей матрицы. Для этого поток последовательно перемножает блоки матриц *pAMatrix* и *pBMatrix,* результат умножения прибавляется к блоку матрицы *pCMatrix*. При этом умножение блоков происходит по правилам блочного умножения – см. (2.5) и рис. 2.8. Каждый поток должен выполнить *GridSize* итераций алгоритма. Каждая такая итерация есть умножение матричных блоков, итерации выполняются внешним циклом *for* по переменной *iter*.

Реализация параллельного алгоритма блочного умножения матриц может быть представлен следующим образом:

```
void ParallelResultCalculation (double* pAMatrix, double* pBMatrix,
  double* pCMatrix, int Size) {
  int ThreadNum = 4;
  int GridSize = int (sqrt((double)ThreadNum));
  int BlockSize = Size/GridSize;
  omp_set_num_threads(ThreadNum);
#pragma omp parallel
  {
    int ThreadID = omp_get_thread_num();
    int RowIndex = ThreadID/GridSize;
    int ColIndex = ThreadID%GridSize;
    for (int iter=0; iter<GridSize; iter++) {
      for (int i=RowIndex*BlockSize; i<(RowIndex+1)*BlockSize; i++)
        for (int j=ColIndex*BlockSize; j<(ColIndex+1)*BlockSize; j++)
          for (int k=iter*BlockSize; k<(iter+1)*BlockSize; k++)
            pCMatrix[i*Size+j] += pAMatrix[i*Size+k] * pBMatrix[k*Size+j];
    }
  } // pragma omp parallel
}
```

Задание 6 – Проверка правильности работы программы

Проверьте правильность выполнения алгоритма. Для этого разработайте функцию *TestResult*, которая сравнит результаты последовательного и параллельного алгоритмов. Для выполнения последовательного алгоритма можно использовать функцию *SerialResultCalculation*, разработанную в упражнении 2. Результат работы этой функции сохраните в матрице *pSerialResult*, а затем поэлементно сравните эту матрицу с матрицей *pCMatrix*, полученной при помощи написанного параллельного алгоритма. Следует отметить, что получение каждого элемента результирующей матрицы требует выполнения последовательности умножений и сложений вещественных чисел и порядок выполнения этих действий, в общем случае, может повлиять на наличие и величину машинной погрешности вычислений. Тем самым, элементы двух полученных матриц могут различаться на величину точности машинных вычислений и их проверка на полное совпадение не может быть использована для проверки правильности работы разработанной программы. Введем допустимую величину расхождения результатов последовательного и параллельного алгоритма *Accuracy* и будем считать матрицы равными, если их соответствующие элементы отличаются не более чем на эту указанную величину.

Функция *TestResult* должна иметь доступ к исходным матрицам *pAMatrix* и *pBMatrix* и результирующей *pCMatrix*. Один из возможных вариантов функции тестирования результатов представлен ниже:

```
void TestResult(double* pAMatrix, double* pBMatrix, double* pCMatrix,
  int Size) {
  double* pSerialResult;   // Result matrix of serial multiplication
  double Accuracy = 1.e-6; // Comparison accuracy
  int equal = 0;           // =1, if the matrices are not equal
  int i;  // Loop variable

  pSerialResult = new double [Size*Size];
  for (i=0; i<Size*Size; i++) {
    pSerialResult[i] = 0;
  }
  SerialResultCalculation(pAMatrix, pBMatrix, pSerialResult, Size);
  for (i=0; i<Size*Size; i++) {
    if (fabs(pSerialResult[i]-pCMatrix[i]) >= Accuracy)
      equal = 1;
  }
  if (equal == 1)
    printf("The results of serial and parallel algorithms "
      "are NOT identical. Check your code.");
   else
     printf("The results of serial and parallel algorithms "
      "are identical.");
  delete [] pSerialResult;
}
```

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

Закомментируйте вызовы функций, использующих отладочную печать, которые ранее использовались для контроля правильности выполнения этапов параллельного приложения. Вместо функции *DummyDataInitialization*, которая генерирует матрицы простого вида, вызовите функцию *RandomDataInitialization*, которая генерирует исходные матрицы при помощи датчика случайных чисел. Скомпилируйте и запустите приложение. Выполните несколько запусков программы задавая различные объемы исходных данных. Убедитесь в том, что приложение работает правильно.

Задание 7 – Проведение вычислительных экспериментов

Определите время выполнения параллельного алгоритма. Для этого добавьте в программный код замеры времени. Ниже представлена одна из возможных реализаций функции *main*:

```
  double Start, Finish;
  double Duration;
  …
  ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

  Start = GetTime();
  ParallelResultCalculation(pMatrix, pVector, pResult, Size);
  Finish = GetTime();
  Duration = Finish-Start;
  TestResult(pMatrix, pVector, pResult, Size);
  printf("Time of parallel execution = %f\n", Duration);
  …
```

Очевидно, что таким образом будет распечатано время, которое было затрачено на выполнение параллельного алгоритма.

Скомпилируйте и запустите приложение. Заполните следующую таблицу результатов вычислительных экспериментов:

Таблица 2.3. Таблица сравнения времени работы последовательного и параллельного алгоритмов

Номер эксперимента	Размер матриц	Последовательный алгоритм	Параллельный алгоритм	
			Время	Ускорение
1	10			
2	100			
3	500			
4	1000			
5	1500			
6	2000			
7	2500			
8	3000			

Для анализа эффективности параллельного алгоритма умножения матриц, основанного на разделении данных на блоки, воспользуйтесь подходом, изложенным в главе 8 учебных материалов курса. Блочное разбиение полос вносит изменение лишь в порядок выполнения вычислений, общий же объем вычислительных операций не меняется. Поскольку вычисления разделены между p потоками, которые выполняются параллельно на нескольких вычислительных элементах, то время, затраченное непосредственно на вычисления, можно оценить при помощи соотношения:

$$T_{calc} = \frac{n^2 \cdot (2n-1)}{p} \cdot \tau \,.$$

Оценим объем данных, которые необходимо прочитать из оперативной памяти в кэш процессора. На каждой итерации алгоритма каждый поток выполняет умножение двух матричных блоков размером *(Size/q)×(Size/q)* элементов. Для оценки объема данных, который при выполнении операции перемножения блоков матрицы должен быть прочитан из оперативной памяти в кэш, можно воспользоваться соотношением, полученным при анализе теоретической сложности последовательного алгоритма; необходимо только внести поправку на размер матричного блока.

$$T^1_{mem} = \frac{8 \cdot \left((n/q)^2 + 9(n/q)^3 \right)}{\beta}$$

Поскольку для вычисления блока результирующей матрицы каждый поток выполняет q итераций, то для определения времени, которое каждый поток тратит на чтение необходимых данных в кэш, необходимо умножить величину T_{mem}^{1} на число итераций q. Обращение нескольких потоков в память происходит строго последовательно. Общее число потоков – q^2. В результате анализе алгоритма можно заметить, что блок, который считывается в кэш одним из потоков, одновременно используется и другими (q-1) потоками. Так, например, при использовании решетки потоков 2×2, при выполнении первой итерации алгоритма блок A_{11} используется одновременно нулевым и первым потоком, блок A_{21} – одновременно вторым и третьим потоком, блок B_{11} – нулевым и вторым, а блок B_{12} – первым и третьим потоками. Следовательно, время, необходимое на чтение данных из оперативной памяти в кэш составляет:

$$T_{mem} = \frac{8 \cdot \left((n/q)^2 + 9(n/q)^3 \right)}{\beta} \cdot q \cdot q = \frac{8 \cdot \left(n^2 + 9 \cdot n^3/q \right)}{\beta} \,. \qquad (2.6)$$

Суммируя полученные выражения можно получить, что общее время выполнения параллельного алгоритма умножения матриц, основанного на блочном разделении данных, определяется соотношением:

$$T_p = \frac{n^2 \cdot (2n-1)}{p} \cdot \tau + \frac{8 \cdot \left(n^2 + 9 \cdot n^3/q \right)}{\beta} \qquad (2.7)$$

Вычислите теоретическое время выполнения параллельного алгоритма в соответствии с выражением (2.7). Результаты занесите в таблицу:

Таблица 2.4. Сравнение реального времени выполнения параллельного алгоритма умножения матриц с теоретическими оценками времени

Размер матриц	Эксперимент	Время вычислений (T_{calc})	Время доступа к памяти (T_{mem})	Общее время (T_p)
10				

100				
500				
1000				
1500				
2000				
2500				
3000				

Задание 8 – Оптимизация блочного алгоритма для эффективного использования кэш-памяти

Как видно из результатов анализа эффективности рассмотренного блочного параллельного алгоритма, значительная доля времени умножения матриц тратится на многократное чтение элементов матриц *pAMatrix* и *pBMatrix* из оперативной памяти в кэш. Эта ситуация имеет место, если блоки матриц *pAMatrix* и *pBMatrix* не могут быть помещены в кэш полностью. Тем самым, организация работы с кэш может быть улучшена.

Для повышения эффективности использования кэш-памяти количество разбиений матриц должно быть таким, чтобы в кэш одновременно могли быть помещены три матричных блока – блоки матриц *pAMatrix*, *pBMatrix* и *pCMatrix*. Если блоки матриц могут быть помещены в кэш полностью, то при вычислении результата умножения матричных блоков не происходит многократного чтения элементов блока в кэш, и, следовательно, затраты на загрузку данных из оперативной памяти существенно сокращаются.

Пусть V_{cache} – объем кэш в байтах. Тогда количество элементов типа double (числа с двойной точностью), которые могут быть помещены в кэш, составляет $V_{cache}/8$ (для хранения одного элемента типа double используется 8 байт). Таким образом, максимальный размер квадратного $k \times k$ матричного блока составляет:

$$k_{\max} = \left\lfloor \sqrt{V_{cache}/(3 \cdot 8)} \right\rfloor$$

Следовательно, минимально необходимое число разбиений *GridSize* при разделении матриц размером $n \times n$ составляет:

$$GridSize = \lceil m / k_{\max} \rceil.$$

Для упрощения программной реализации алгоритма количество блоков по горизонтали и вертикали *GridSize* должно быть таким, чтобы размер матриц *n* мог быть поделен на *GridSize* без остатка.

Реализуйте последовательный алгоритм умножения матриц, основанный на блочном разбиении матриц и ориентированный на максимально эффективное использование кэш-памяти. Ниже представлена одна из возможных реализаций описанного выше алгоритма:

```
// Serial block matrix mutiplication (cache-optimizied version)
void SerialBlockResultCalculation (double* pAMatrix, double* pBMatrix,
  double* pCMatrix, int Size) {
  int BlockSize = 250;
  int GridSize = int (Size/double(BlockSize));
  for (int n=0; n<GridSize; n++)
    for (int m=0; m<GridSize; m++)
      for (int iter=0; iter<GridSize; iter++)
        for (int i=n*BlockSize; i<(n+1)*BlockSize; i++)
          for (int j=m*BlockSize; j<(m+1)*BlockSize; j++)
            for (int k=iter*BlockSize; k<(iter+1)*BlockSize; k++)
              pCMatrix[i*Size+j] +=
                pAMatrix[i*Size+k] * pBMatrix[k*Size+j];
}
```

Следует отметить, что при использовании различных вычислительных систем с разным размером кэша, следует варьировать параметр *BlockSize* таким образом, чтобы три матричных блока могли быть одновременно помещены в кэш.

Подготовьте данный вариант последовательной программы. Выполните вычислительные эксперименты для разных размеров блоков. Определите по результатам экспериментов параметры программы, при которых достигается минимальное время вычислений. Заполните таблицу экспериментов вида (2.1).

Задание 9 – Реализация параллельного кэш-оптимизированного блочного алгоритма

Для распараллеливания представленного блочного алгоритма умножения матриц воспользуйтесь подходом, изложенным при рассмотрении блочного алгоритма. Пусть, как и ранее, поток отвечает за вычисление блока результирующей матрицы. Однако теперь, когда количество блоков определяется не количеством потоков, а объемом кэш-памяти, число блоков может существенно превосходить число доступных потоков. Поэтому каждый поток должен вычислять несколько матричных блоков. Распределите между потоками параллельной программы итерации второго цикла (цикла по переменной *m*). При таком распределении нагрузки на каждой итерации внешнего цикла поток последовательно выполняет поблочное умножение горизонтальной полосы матрицы *pAMatrix* на несколько вертикальных полос матрицы *pBMatrix*.

На рисунке представлена схема выполнения параллельного алгоритма умножения матриц, основанного на блочном разделении матриц и ориентированного на эффективное использование кэш-памяти. Здесь каждая матрица разделена на 4 матричных блока, параллельные фрагменты создаются два раза (по числу строк в блочной решетке), каждая из них отвечает за умножение текущей строки матрицы *A* на несколько столбцов матрицы *B* (в данном случае каждый поток умножает горизонтальную полосу матрицы *A* на одну вертикальную полосу матрицы *B*).

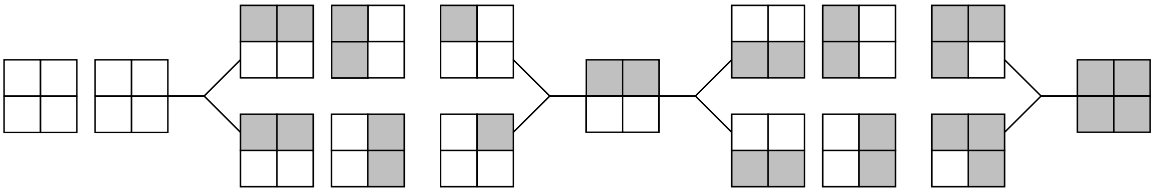

Рис. 2.13. Выполнение параллельного алгоритма умножения матриц, эффективно использующего кэш-память, в случае, когда каждая матрица разбита на 4 матричных блока и для выполнения алгоритма используется 2 потока

Программная реализация описанного подхода может быть представлена следующим образом:

```
void ParallelBlockResultCalculation (double* pAMatrix, double* pBMatrix,
  double* pCMatrix, int Size) {
  int BlockSize = 250;
  int GridSize = int (Size/double(BlockSize));
  for (int n=0; n<GridSize; n++)
#pragma omp parallel for
    for (int m=0; m<GridSize; m++)
      for (int iter=0; iter<GridSize; iter++)
        for (int i=n*BlockSize; i<(n+1)*BlockSize; i++)
          for (int j=m*BlockSize; j<(m+1)*BlockSize; j++)
            for (int k=iter*BlockSize; k<(iter+1)*BlockSize; k++)
              pCMatrix[i*Size+j] +=
                pAMatrix[i*Size+k] * pBMatrix[k*Size+j];
}
```

Задание 10 – Проведение вычислительных экспериментов для параллельного блочного кэш-оптимизированного алгоритма

Подготовьте данный вариант параллельной программы. Выполните вычислительные эксперименты для разных размеров блоков. Заполните таблицу результатов вычислительных экспериментов.

Таблица 2.5. Таблица сравнения времени работы последовательного и параллельного блочных алгоритмов умножения матриц, эффективно использующих кэш-память

Размер матриц	Базовый последовательный алгоритм (T_1)	Блочный последовательный алгоритм (T'_1)	Блочный параллельный алгоритм – размер блока _____					
			2 вычислительных элемента			4 вычислительных элемента		
			Время (T_2)	Ускорение		Время (T_4)	Ускорение	
				T'_1/T_2	T_1/T_2		T'_1/T_4	T_1/T_4
500								
1000								
1500								

2000								
2500								
3000								

При анализе эффективности параллельного алгоритма умножения матриц, основанного на разделении данных на блоки определенного размера с тем, чтобы максимально эффективно использовать кэш-память, можно использовать оценки, полученные при анализе предыдущего параллельного алгоритма. Действительно, объем вычислительных операций не изменяется. Поэтому вычислительная сложность блочного алгоритма составляет:

$$T_{calc} = \frac{n^2 \cdot (2n-1)}{p} \cdot \tau \,.$$

При умножении матриц с помощью рассмотренного алгоритма выполняется $GridSize^3$ (размер *GridSize* определяется как результат деления размера матриц n на размер матричного блока *BlockSize*) операций умножения матричных блоков, при этом сами блоки настолько малы, что на каждой итерации они могут быть одновременно помещены в кэш. Значит, время, необходимое на чтение данных из оперативной памяти в кэш может быть вычислено по формуле:

$$T_{mem} = \frac{8 \cdot \left(\frac{n}{BlockSize}\right)^3 \cdot 3BlockSize^2}{\beta} = \frac{24 \cdot n^3}{BlockSize \cdot \beta} \,.$$

Необходимо отметить, что при выполнении данного параллельного алгоритма параллельные секции создаются и закрываются *GridSize* раз, значит необходимо учесть время, затраченное на выполнение этих действий. Следовательно, общее время выполнения параллельного блочного алгоритма умножения матриц, ориентированного на эффективное использование кэш, может быть вычислено по формуле:

$$T_p = \frac{n^2 \cdot (2n-1)}{p} \cdot \tau + \frac{24 \cdot n^3}{BlockSize \cdot \beta} + \frac{n}{BlockSize} \cdot \delta \,. \tag{2.8}$$

Подсчитайте теоретические оценки времени выполнения матричного умножения по формуле (2.8), а также используя результаты предыдущих вычислительных экспериментов, заполните следующую таблицу:

Таблица 2.6. Сравнение реального времени выполнения параллельного блочного алгоритма, эффективно использующего кэш, с теоретическими оценками времени

Размер матриц	Эксперимент	Время вычислений (T_{calc})	Время доступа к памяти (T_{mem})	Общее время (T_p)
500				
1000				
1500				
2000				
2500				
3000				

Контрольные вопросы

- Насколько сильно отличаются время, затраченное на выполнение последовательного и параллельного алгоритма? Почему?
- Получилось ли ускорение при матрице размером 10 на 10? Почему?
- Насколько хорошо совпадают время, полученное теоретически, и реальное время выполнения алгоритма? В чем может состоять причина несовпадений?

Задания для самостоятельной работы

1. Для разработанных в лабораторной работе параллельных программ примените распараллеливание для циклов разного уровня вложенности. Проведите вычислительные эксперименты и оцените получаемое ускорение. Сравните полученные данные с результатами лабораторной работы.
2. Рассмотрите ленточную схему разделения матриц. Примените эту схему для разработки нескольких вариантов параллельного алгоритма умножения матриц и выполните их программную

реализацию. Проведите вычислительные эксперименты и оцените получаемое ускорение. Сравните полученные данные с результатами лабораторной работы.

3. Разработайте алгоритм матричного умножения на основе операции умножении матрицы на вектор и выполните его программную реализацию. Проведите вычислительные эксперименты и оцените получаемое ускорение. Сравните полученные данные с результатами лабораторной работы.

Приложение 1. Программный код последовательного приложения для матричного умножения

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

// Function that converts numbers form LongInt type to double type
double LiToDouble (LARGE_INTEGER x) {
  double result =
    ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
  return result;
}

// Function that gets the timestamp in seconds
double GetTime() {
  LARGE_INTEGER lpFrequency, lpPerfomanceCount;
  QueryPerformanceFrequency (&lpFrequency);
  QueryPerformanceCounter (&lpPerfomanceCount);
  return LiToDouble(lpPerfomanceCount)/LiToDouble(lpFrequency);
}

// Function for simple initialization of matrix elements
void DummyDataInitialization (double* pAMatrix,double* pBMatrix,int Size) {
  int i, j;  // Loop variables

  for (i=0; i<Size; i++)
    for (j=0; j<Size; j++) {
      pAMatrix[i*Size+j] = 1;
      pBMatrix[i*Size+j] = 1;
  }
}

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

// Function for memory allocation and initialization of matrix elements
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, int &Size) {
  // Setting the size of matricies
  do {
    printf("\nEnter size of matricies: ");
    scanf("%d", &Size);
    printf("\nChosen matricies' size = %d\n", Size);
    if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
```

```
  }
  while (Size <= 0);

  // Memory allocation
  pAMatrix = new double [Size*Size];
  pBMatrix = new double [Size*Size];
  pCMatrix = new double [Size*Size];

  // Initialization of matrix elements
  DummyDataInitialization(pAMatrix, pBMatrix, Size);
  for (int i=0; i<Size*Size; i++) {
    pCMatrix[i] = 0;
  }
}

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

// Function for matrix multiplication
void SerialResultCalculation(double* pAMatrix, double* pBMatrix,
  double* pCMatrix, int Size) {
  int i, j, k;  // Loop variables
  for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++)
      for (k=0; k<Size; k++)
        pCMatrix[i*Size+j] += pAMatrix[i*Size+k]*pBMatrix[k*Size+j];
  }
}

// Function for computational process termination
void ProcessTermination (double* pAMatrix, double* pBMatrix,
  double* pCMatrix) {
  delete [] pAMatrix;
  delete [] pBMatrix;
  delete [] pCMatrix;
}

void main() {
  double* pAMatrix;  // The first argument of matrix multiplication
  double* pBMatrix;  // The second argument of matrix multiplication
  double* pCMatrix;  // The result matrix
  int Size;            // Size of matricies
  double start, finish;
  double duration;

  printf("Serial matrix multiplication program\n");
  // Memory allocation and initialization of matrix elements
  ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

  // Matrix output
  printf ("Initial A Matrix \n");
  PrintMatrix(pAMatrix, Size, Size);
  printf("Initial B Matrix \n");
  PrintMatrix(pBMatrix, Size, Size);

  // Matrix multiplication
```

```
  start = GetTime();
  SerialResultCalculation(pAMatrix, pBMatrix, pCMatrix, Size);
  finish = GetTime();
  duration = (finish-start)/double(CLOCKS_PER_SEC);

  // Printing the result matrix
  printf ("\n Result Matrix: \n");
  PrintMatrix(pCMatrix, Size, Size);

  // Printing the time spent by matrix multiplication
  printf("\n Time of execution: %f\n", duration);

  // Computational process termination
  ProcessTermination(pAMatrix, pBMatrix, pCMatrix);
}
```

Приложение 2 – Программный код параллельного приложения для матричного умножения

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

// Function that converts numbers form LongInt type to double type
double LiToDouble (LARGE_INTEGER x) {
  double result =
    ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
  return result;
}

// Function that gets the timestamp in seconds
double GetTime() {
  LARGE_INTEGER lpFrequency, lpPerfomanceCount;
  QueryPerformanceFrequency (&lpFrequency);
  QueryPerformanceCounter (&lpPerfomanceCount);
  return LiToDouble(lpPerfomanceCount)/LiToDouble(lpFrequency);
}

// Function for simple initialization of matrix elements
void DummyDataInitialization (double* pAMatrix,double* pBMatrix,int Size) {
  int i, j;  // Loop variables

  for (i=0; i<Size; i++)
    for (j=0; j<Size; j++) {
      pAMatrix[i*Size+j] = 1;
      pBMatrix[i*Size+j] = 1;
  }
}

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

```
}

// Function for memory allocation and initialization of matrix elements
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, int &Size) {
  // Setting the size of matricies
  do {
    printf("\nEnter size of matricies: ");
    scanf("%d", &Size);
    printf("\nChosen matricies' size = %d\n", Size);
    if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  }
  while (Size <= 0);

  // Memory allocation
  pAMatrix = new double [Size*Size];
  pBMatrix = new double [Size*Size];
  pCMatrix = new double [Size*Size];

  // Initialization of matrix elements
  DummyDataInitialization(pAMatrix, pBMatrix, Size);
  for (int i=0; i<Size*Size; i++) {
    pCMatrix[i] = 0;
  }
}

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

// Function for matrix multiplication
void SerialResultCalculation(double* pAMatrix, double* pBMatrix,
  double* pCMatrix, int Size) {
  int i, j, k;  // Loop variables
  for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++)
      for (k=0; k<Size; k++)
        pCMatrix[i*Size+j] += pAMatrix[i*Size+k]*pBMatrix[k*Size+j];
  }
}

// Function for computational process termination
void ProcessTermination (double* pAMatrix, double* pBMatrix,
  double* pCMatrix) {
  delete [] pAMatrix;
  delete [] pBMatrix;
  delete [] pCMatrix;
}

void ParallelResultCalculation (double* pAMatrix, double* pBMatrix,
double* pCMatrix, int Size) {
  int ThreadNum = 4;
  int GridSize = int (sqrt((double)ThreadNum));
  int BlockSize = Size/GridSize;
  omp_set_num_threads(ThreadNum);
#pragma omp parallel
```

```
  {
    int ThreadID = omp_get_thread_num();
    int RowIndex = ThreadID/GridSize;
    int ColIndex = ThreadID%GridSize;
    for (int iter=0; iter<GridSize; iter++) {
      for (int i=RowIndex*BlockSize; i<(RowIndex+1)*BlockSize; i++)
        for (int j=ColIndex*BlockSize; j<(ColIndex+1)*BlockSize; j++)
          for (int k=iter*BlockSize; k<(iter+1)*BlockSize; k++)
            pCMatrix[i*Size+j] += pAMatrix[i*Size+k] * pBMatrix[k*Size+j];
    }
  } // pragma omp parallel
}

void TestResult(double* pAMatrix, double* pBMatrix, double* pCMatrix,
  int Size) {
  double* pSerialResult;   // Result matrix of serial multiplication
  double Accuracy = 1.e-6; // Comparison accuracy
  int equal = 0;           // =1, if the matrices are not equal
  int i;  // Loop variable

  pSerialResult = new double [Size*Size];
  for (i=0; i<Size*Size; i++) {
    pSerialResult[i] = 0;
  }
  SerialResultCalculation(pAMatrix, pBMatrix, pSerialResult, Size);
  for (i=0; i<Size*Size; i++) {
    if (fabs(pSerialResult[i]-pCMatrix[i]) >= Accuracy)
      equal = 1;
  }
  if (equal == 1)
    printf("The results of serial and parallel algorithms "
      "are NOT identical. Check your code.");
   else
     printf("The results of serial and parallel algorithms "
      "are identical.");
  delete [] pSerialResult;
}

// Serial block matrix mutiplication
void OptimalResultCalculation (double* pAMatrix, double* pBMatrix, double*
pCMatrix, int Size) {
int BlockSize = 250;
  int GridSize = int (Size/double(BlockSize));
for (int n=0; n<GridSize; n++)
    for (int m=0; m<GridSize; m++)
      for (int iter=0; iter<GridSize; iter++)
        for (int i=n*BlockSize; i<(n+1)*BlockSize; i++)
          for (int j=m*BlockSize; j<(m+1)*BlockSize; j++)
            for (int k=iter*BlockSize; k<(iter+1)*BlockSize; k++)
              pCMatrix[i*Size+j] +=
                pAMatrix[i*Size+k] * pBMatrix[k*Size+j];
}

void ParallelOptimalResultCalculation (double* pAMatrix, double* pBMatrix,
  double* pCMatrix, int Size) {
  int BlockSize = 250;
  int GridSize = int (Size/double(BlockSize));
  for (int n=0; n<GridSize; n++)
#pragma omp parallel for
    for (int m=0; m<GridSize; m++)
      for (int iter=0; iter<GridSize; iter++)
        for (int i=n*BlockSize; i<(n+1)*BlockSize; i++)
          for (int j=m*BlockSize; j<(m+1)*BlockSize; j++)
```

```
            for (int k=iter*BlockSize; k<(iter+1)*BlockSize; k++)
              pCMatrix[i*Size+j] +=
                pAMatrix[i*Size+k] * pBMatrix[k*Size+j];
}

void main() {
  double* pAMatrix;  // The first argument of matrix multiplication
  double* pBMatrix;  // The second argument of matrix multiplication
  double* pCMatrix;  // The result matrix
  int Size;            // Size of matricies
  double start, finish;
  double duration;

  printf("Serial matrix multiplication program\n");
  // Memory allocation and initialization of matrix elements
  ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

  // Matrix output
  printf ("Initial A Matrix \n");
  PrintMatrix(pAMatrix, Size, Size);
  printf("Initial B Matrix \n");
  PrintMatrix(pBMatrix, Size, Size);

  // Matrix multiplication
  start = GetTime();
  SerialResultCalculation(pAMatrix, pBMatrix, pCMatrix, Size);
  finish = GetTime();
  duration = (finish-start)/double(CLOCKS_PER_SEC);

  // Printing the result matrix
  printf ("\n Result Matrix: \n");
  PrintMatrix(pCMatrix, Size, Size);

  // Printing the time spent by matrix multiplication
  printf("\n Time of execution: %f\n", duration);

  start = GetTime();
  ParallelResultCalculation(pAMatrix, pBMatrix, pCMatrix, Size);
  finish = GetTime();
  duration = finish-start;
  TestResult(pAMatrix, pBMatrix, pCMatrix, Size);
  printf("Time of parallel execution = %f\n", duration);

  start = GetTime();
  OptimalResultCalculation(pAMatrix, pBMatrix, pCMatrix, Size);
  finish = GetTime();
  duration = finish-start;

  // Printing the time spent by matrix multiplication
  printf("\n Time of optimal execution: %f\n", duration);

  start = GetTime();
  ParallelOptimalResultCalculation(pAMatrix, pBMatrix, pCMatrix, Size);
  finish = GetTime();
  duration = finish-start;
  TestResult(pAMatrix, pBMatrix, pCMatrix, Size);
  printf("Time of optimal algorithm parallel execution = %f\n", duration);

  // Computational process termination
  ProcessTermination(pAMatrix, pBMatrix, pCMatrix);
}
```