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 матрицы на вектор"

Лабораторная работа 1: Параллельные алгоритмы матрично-векторного умножения

Разработчик: Е.А.Козинов

Нижний Новгород
2007

Содержание

Матрицы и матричные операции широко используются при математическом моделировании самых разнообразных процессов, явлений и систем. Матричные вычисления составляют основу многих научных и инженерных расчетов – среди областей приложений могут быть указаны вычислительная математика, физика, экономика и др.

Являясь вычислительно-трудоемкими, матричные вычисления представляют собой классическую область применения параллельных вычислений. С одной стороны, использование высокопроизводительных многопроцессорных систем позволяет существенно повысить сложность решаемых задач. С другой стороны, в силу своей достаточно простой формулировки матричные операции предоставляют прекрасную возможность для демонстрации многих приемов и методов параллельного программирования.

Цель лабораторной работы

Целью данной лабораторной работы является разработка параллельной программы матрично-векторного умножения при использовании в качестве аппаратной платформы многопроцессорной вычислительной системы с общей памятью.

- Упражнение 1 – Постановка задачи матрично-векторного умножения
- Упражнение 2 – Реализация последовательного алгоритма умножения матрицы на вектор
- Упражнение 3 – Разработка параллельного алгоритма умножения матрицы на вектор
- Упражнение 4 - Реализация параллельного алгоритма матрично-векторного умножения

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

При выполнении лабораторной работы предполагается, что имеющиеся в составе вычислительной системы процессоры обладают равной производительностью, являются равноправными при доступе к общей памяти и время доступа к памяти является одинаковым (при одновременном доступе нескольких процессоров к одному и тому же элементу памяти очередность и синхронизация доступа обеспечивается на аппаратном уровне). Многопроцессорные системы подобного типа обычно именуются *симметричными мультипроцессорами* (*symmetric multiprocessors, SMP*). Перечисленному выше набору предположений удовлетворяют также активно развиваемые в последнее время *многоядерные процессоры*, в которых каждое ядро представляет практически независимо функционирующее вычислительное устройство. Именно многоядерные процессоры (учитывая их новизну и массовый характер распространения) будут использоваться далее для проведения вычислительных экспериментов. Для общности излагаемого учебного материала, для упоминания одновременно и мультипроцессоров и многоядерных процессоров, при обозначении одного вычислительного устройства будет использоваться понятие *вычислительный элемент* (*ВЭ*).

При выполнении лабораторной работы предполагается знание раздела 4 "Параллельное программирование на основе OpenMP", раздела 6 "Принципы разработки параллельных методов" и раздела 7 "Параллельные методы умножения матрицы на вектор" учебных материалов курса.

Упражнение 1 – Постановка задачи умножения матрицы на вектор

В результате умножения матрицы A размерности $m \times n$ и вектора b, состоящего из n элементов, получается вектор c размера m, каждый i-ый элемент которого есть результат скалярного умножения i-й строки матрицы A (обозначим эту строчку a_i) и вектора b (см. рис. 1.1):

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j \,, 1 \le i \le m \,. \qquad (1.1)$$

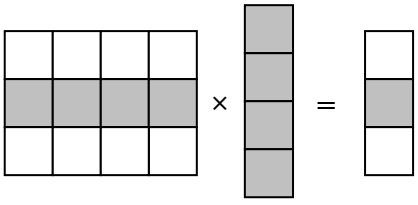

Рис. 1.1. Элемент результирующего вектора – это результат скалярного умножения строки матрицы на вектор

Так, например, при умножении матрицы, состоящей из 3 строк и 4 столбцов на вектор из 4 элементов, получается вектор размера 3:

$$\begin{pmatrix} 3 & 2 & 0 & -1 \\ 5 & -2 & 1 & 1 \\ 1 & 0 & -1 & -1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 8 \\ -6 \end{pmatrix}$$

Рис. 1.2. Умножение матрицы на вектор

Тем самым, получение результирующего вектора *c* предполагает повторение *m* однотипных операций по умножению строк матрицы *A* и вектора *b*. Каждая такая операция включает умножение элементов строки матрицы и вектора *b* и последующее суммирование полученных произведений.

Псевдокод для представленного алгоритма умножения матрицы на вектор может выглядеть следующим образом:

```
// Serial algorithm of matrix-vector multiplication
for (i = 0; i < m; i++){
  c[i] = 0;
  for (j = 0; j < n; j++){
    c[i] += A[i][j]*b[j]
  }
}
```

Упражнение 2 – Реализация последовательного алгоритма

При выполнении этого упражнения необходимо реализовать последовательный алгоритм матрично-векторного умножения. Начальный вариант будущей программы представлен в проекте *SerailMatrixVecorMult*, который содержит часть исходного кода и в котором заданы необходимые параметры проекта. В ходе выполнения упражнения необходимо дополнить имеющийся вариант программы операциями ввода размера объектов, инициализации матрицы и вектора, умножения матрицы на вектор и вывода результатов.

Задание 1 – Открытие проекта SerialMatrixVectorMult

Откройте проект **SerialMatrixVector**, 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\Serial Matrix Vector,**
- Дважды щелкните на файле **SerialMatrixVector.sln** или выбрав файл выполните команду **Open.**

После открытия проекта в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **SerialMV.cpp**, как это показано на рис. 1.3. После этих действий программный код, который предстоит в дальнейшем расширить, будет открыт в рабочей области Visual Studio.

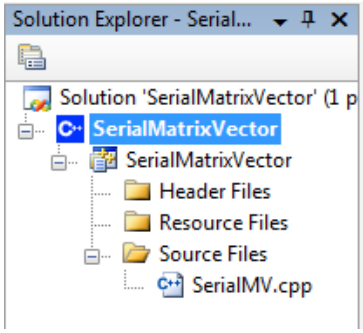

Рис. 1.3. Открытие файла SerialMV.cpp

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 (*pMatrix* и *pVector*) – это, соответственно, матрица и вектор, которые участвуют в матрично-векторном умножении в качестве аргументов. Третья переменная *pResult* – вектор, который

должен быть получен в результате матрично-векторного умножения. Переменная *Size* определяет размер матриц и векторов (предполагаем, что матрица *pMatrix* квадратная, имеет размерность *Size*×*Size*, и умножается на вектор из *Size* элементов). Далее объявлены переменные циклов.

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

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

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

```
  printf ("Serial matrix-vector 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-vector multiplication program`". Для того чтобы завершить выполнение программы, нажмите любую клавишу.

Задание 2 – Ввод размеров объектов

Для задания исходных данных последовательного алгоритма умножения матрицы на вектор реализуем функцию *ProcessInitialization*. Эта функция предназначена для определения размеров матрицы и вектора, выделения памяти для всех объектов, участвующих в умножении (исходных матрицы *pMatrix* и вектора *pVector*, и результата умножения *pResult*), а также для задания значений элементов исходных матрицы и вектора. Значит, функция должна иметь следующий интерфейс:

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

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

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

Пользователю предоставляется возможность ввести размер объектов (матрицы и вектора), который затем считывается из стандартного потока ввода *stdin* и сохраняется в целочисленной переменной *Size*. Далее печатается значение переменной *Size* (рис. 1.4).

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

```
void main() {
  double* pMatrix;  // Initial matrix
  double* pVector;  // Initial vector
  double* pResult;  // Result vector for matrix-vector multiplication
  int Size;         // Sizes of initial matrix and vector
  time_t start, finish;
  double duration;

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

```
}
```

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

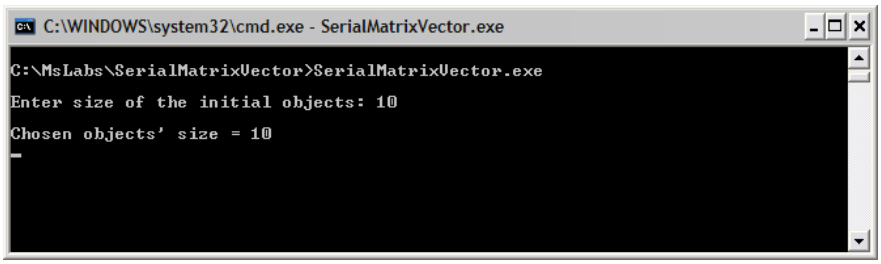

Рис. 1.4. Задание размера объектов

Теперь обратимся к вопросу контроля правильности ввода. Так, например, если в качестве размера объектов пользователь попытается ввести неположительное число, приложение должно либо завершить выполнение, либо продолжать запрашивать размер объектов до тех пор, пока не будет введено положительное число. Реализуем второй вариант поведения, для этого тот фрагмент кода, который производит ввод размера объектов, поместим в цикл с постусловием:

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

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 data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, int Size) {
  // Setting the size of initial matrix and vector
  do {
    <…>
  }
  while (Size <= 0);

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

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

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

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

```
}
```

Как видно из представленного фрагмента кода, данная функция осуществляет задание элементов матрицы и вектора достаточно простым образом: значение элемента матрицы совпадает с номером строки, в которой он расположен, а все элементы вектора равны 1. То есть в случае, когда пользователь выбрал размер объектов, равный 4, будут определены следующие матрица и вектор:

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

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

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

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int Size) {
  // Setting the size of initial matrix and vector
  do {
    <…>
  }
  while (Size <= 0);

  // Memory allocation
  <…>

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

Реализуем еще две функции, которые помогут контролировать ввод данных. Это функции форматированного вывода объектов: *PrintMatrix* и *PrintVector*. В качестве аргументов в функцию форматированной печати матрицы *PrintMatrix* передается указатель на одномерный массив, где эта матрица хранится построчно, а также размеры матрицы по вертикали (количество строк *RowCount*) и горизонтали (количество столбцов *ColCount*). Для форматированной печати вектора при помощи функции *PrintVector*, необходимо сообщить функции указатель на вектор, а также количество элементов в нем.

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

// Function for formatted vector output
void PrintVector (double* pVector, int Size) {
  int i;
  for (i=0; i<Size; i++)
    printf("%7.4f ", pVector[i]);
  printf("\n");
}
```

Добавим вызов этих функций в основную функцию приложения:

```
  // Memory allocation and data 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);
```

Скомпилируйте и запустите приложение. Убедитесь в том, что ввод данных происходит по описанным правилам (рис. 1.5). Выполните несколько запусков приложения, задавайте различные размеры объектов.

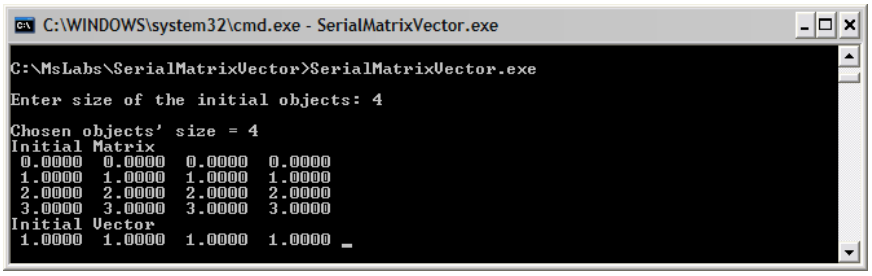

Рис. 1.5. Результат работы программы при завершении задания 3

Задание 4 – Завершение процесса вычислений

Перед выполнением матрично-векторного 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 матрицы *pMatrix* и вектора *pVector*, а также для хранения результата умножения *pResult*. Следовательно, эти объекты необходимо передать в функцию *ProcessTermination* в качестве аргументов:

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

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

```
  // Memory allocation and data inialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

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

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
```

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

Задание 5 – Реализация умножения матрицы на вектор

Выполним теперь разработку основной вычислительной части программы. Для выполнения умножения матрицы на вектор реализуем функцию *SerialResultCalculation*, которая принимает на вход исходные матрицу *pMatrix* и вектор *pVector*, размеры этих объектов *Size*, а также указатель на вектор в памяти, где должен быть сохранен результат *pResult*.

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

```
// Function for matrix-vector multiplication
void SerialResultCalculation(double* pMatrix, double* pVector, double*
pResult,
    int Size) {
  int i, j;  // Loop variables
  for (i=0; i<Size; i++) {
    pResult[i] = 0;
```

```
    for (j=0; j<Size; j++)
      pResult[i] += pMatrix[i*Size+j]*pVector[j];
  }
}
```

Следует отметить, что приведенный программный код может быть оптимизирован (вычисление индексов, использование кэша и т.п.), однако такая оптимизация не является целью данного учебного материала и приведет к усложнению программ.

Выполним вызов функции вычисления умножения матрицы на вектор из основной программы. Для контроля правильности выполнения умножения распечатаем результирующий вектор:

```
  // Memory allocation and data 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);

  // Matrix-vector multiplication
  SerialResultCalculation(pMatrix, pVector, pResult, Size);

  // Printing the result vector
  printf ("\n Result Vector: \n");
  PrintVector(pResult, Size);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
```

Скомпилируйте и запустите приложение. Проанализируйте результат работы алгоритма умножения матрицы на вектор. Если алгоритм реализован правильно, то результирующий вектор должен иметь следующую структуру: *i*-ый элемент результирующего вектора равен произведению размера вектора *Size* на номер элемента *i*. Так, если размер объектов *Size* равен 4, результирующий вектор *pResult* должен быть таким: *pResult* = (0, 4, 8, 12). Проведите несколько вычислительных экспериментов, изменяя размеры объектов.

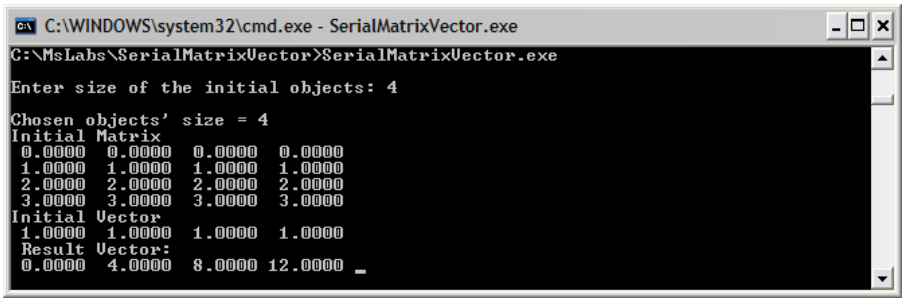

Рис. 1.6. Результат выполнения матрично-векторного умножения

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

```
      pMatrix[i*Size+j] = rand()/double(1000);
  }
}
```

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 data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int Size) {
  // Size of initial matrix and vector definition
  <…>

  // Memory allocation
  <…>

  // Random data initialization
  RandomDataInitialization(pMatrix, pVector, Size);
}
```

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

Для определения времени можно воспользоваться стандартной функцией языка C, которая позволяет узнать время работы программы или её части:

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

К сожалению, данный счетчик обладает малой чувствительностью, поэтому он не может быть использован для измерения времени работы программ с малым числом вычислительных операций. Реализуем функцию *GetTime()*, которая позволяет более точно замерить время выполнения участка кода с использованием функций библиотеки WinAPI:

```
// 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);
}
```

Функция *GetTime* возвращает текущее значение времени, которое отсчитывается от фиксированного момента в прошлом, в секундах. Следовательно, вызвав эту функцию два раза – до и после исследуемого фрагмента можно вычислить время его работы. Например, этот фрагмент вычислит время *duration* работы функции f().

```
double t1, t2;
t1 = GetTime();
f();
t2 = GetTime();
double duration = (t2-t1);
```

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

```
    // Matrix-vector multiplication
    Start = GetTime();
    ResultCalculation(pMatrix, pVector, pResult, Size);
    Finish = GetTime();
    Duration = (Finish-Start);

    // Printing the result vector
```

```
    printf ("\n Result Vector: \n");
    PrintVector(pResult, Size);
    // Printing the time spent by matrix-vector multiplication
    printf("\n Time of serial execution: %f", Duration);
```

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

Номер теста	Размер матрицы	Время работы (сек)
Тест № 1	10	
Тест № 2	100	
Тест № 3	1000	
Тест № 4	2000	
Тест № 5	3000	
Тест № 6	4000	
Тест № 7	5000	
Тест № 8	6000	
Тест № 9	7000	
Тест № 10	8000	
Тест № 11	9000	
Тест № 12	10 000	

Для построения теоретических оценок времени вычислений воспользуемся подходом, изложенным в Главе 7 учебных материалов курса. В результате проведенного анализа следует, что время выполнения вычислений для умножения матрицы на вектор может оценить при помощи следующего соотношения:

$$T_1 = n\,(2n-1)\cdot\tau + 8n^2/\beta\,, \qquad (1.2)$$

где n есть размерность матрицы, τ есть время выполнения одной вычислительной операции, β есть пропускная способность канала доступа к оперативной памяти,.

Процедура оценки значения параметров τ и β формулы (1.2) подробно изложена в главе 7.

Далее заполните таблицу сравнения реального времени выполнения со временем, которое может быть получено по формуле (1.2).

Время выполнения одной операции τ (сек):			
Номер теста	Размер матрицы	Время работы (сек)	Теоретическое время (сек)
Тест № 1	10		
Тест № 2	100		
Тест № 3	1000		
Тест № 4	2000		
Тест № 5	3000		
Тест № 6	4000		
Тест № 7	5000		
Тест № 8	6000		
Тест № 9	7000		
Тест № 10	8000		
Тест № 11	9000		
Тест № 12	10 000		

Упражнение 3 – Разработка параллельного алгоритма

Принципы распараллеливания

Разработка алгоритмов (а в особенности методов параллельных вычислений) для решения сложных научно-технических задач часто представляет собой значительную проблему. Здесь же мы будем полагать, что вычислительная схема решения нашей задачи умножения матрицы на вектор уже известна. Действия для определения эффективных способов организации параллельных вычислений могут состоять в следующем:

- Выполнить анализ имеющейся вычислительной схемы и осуществить ее разделение (*декомпозицию*) на части (*подзадачи*), которые могут быть реализованы в значительной степени независимо друг от друга,
- Выделить для сформированного набора подзадач *информационные взаимодействия*, которые должны осуществляться в ходе решения исходной поставленной задачи,
- Для *систем с общей памятью* выполнить распределение выделенных *подзадач* по *ВЭ*.

При самом общем рассмотрении понятно, что объем вычислений для каждого используемого потока должен быть примерно одинаков – это позволит обеспечить равномерную вычислительную загрузку (*балансировку*) *ВЭ*. Кроме того, также понятно, что распределение подзадач между процессорами (ядрами) должно быть выполнено таким образом, чтобы наличие информационных связей (*коммуникационных взаимодействий*) между подзадачами было минимальным.

Определение подзадач

Для многих методов матричных вычислений характерным является повторение одних и тех же вычислительных действий для разных элементов матриц. Данный момент свидетельствует о наличии *параллелизма по данным* при выполнении матричных расчетов и, как результат, распараллеливание матричных операций сводится в большинстве случаев к разделению обрабатываемых матриц между процессорами используемой вычислительной системы. Выбор способа разделения матриц приводит к определению конкретного метода параллельных вычислений; существование разных схем распределения данных порождает целый ряд *параллельных алгоритмов матричных вычислений*.

Дадим кратко общую характеристику распределения данных для матричных алгоритмов – более подробно данный материал содержится в разделе 7 учебного курса. Наиболее общие и широко используемые способы разделения матриц состоят в разбиении данных на *полосы* (по вертикали или горизонтали) или на прямоугольные фрагменты (*блоки*).

1. Ленточное разбиение матрицы. При *ленточном* (*block-striped*) разбиении каждому процессору (ядру) выделяется то или иное подмножество строк (*rowwise* или *горизонтальное разбиение*) или столбцов (*columnwise* или *вертикальное разбиение*) матрицы (рис. 1.7а и 1.7б). Разделение строк и столбцов на полосы в большинстве случаев происходит на *непрерывной* (*последовательной*) основе. При таком подходе для горизонтального разбиения по строкам, например, матрица A представляется в виде:

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

где $a_i = (a_{i1}, a_{i2}, ... a_{in})$, $0 \le i < m$, есть *i-я* строка матрицы A (предполагается, что количество строк m кратно числу процессоров (ядер) p, т.е. $m = k \cdot p$). Во всех алгоритмах матричного умножения и умножения матрицы на вектор, которые будут рассмотрены нами в этом и следующем разделах, используется разделение данных на непрерывной основе.

Другой возможный подход к формированию полос состоит в применении той или иной схемы *чередования* (*цикличности*) строк или столбцов. Как правило, для чередования используется число процессоров (ядер) p – в этом случае при горизонтальном разбиении матрица A принимает вид

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

Циклическая схема формирования полос может оказаться полезной для лучшей балансировки вычислительной нагрузки процессоров (ядер) (например, при решении системы линейных уравнений с использованием метода Гаусса – см. раздел 9 учебного курса).

2. Блочное разбиение матрицы. При *блочном* (*checkerboard block*) разделении матрица делится на прямоугольные наборы элементов – при этом, как правило, используется разделение на непрерывной основе. Пусть количество процессоров (ядер) составляет $p = s \cdot q$, количество строк матрицы является кратным s, а количество столбцов – кратным q, то есть $m = k \cdot s$ и $n = l \cdot q$. Представим исходную матрицу A в виде набора прямоугольных блоков следующим образом (рис. 1.7в):

$$A = \begin{pmatrix} A_{00} & A_{02} & \ldots A_{0q-1} \\ & \ldots & \\ A_{s-11} & A_{s-12} & \ldots A_{s-1q-1} \end{pmatrix},$$

где A_{ij} - блок матрицы, состоящий из элементов:

$$A_{ij} = \begin{pmatrix} a_{i_0 j_0} & a_{i_0 j_1} & \ldots a_{i_0 j_{l-1}} \\ & \ldots & \\ a_{i_{k-1} j_0} & a_{i_{k-1} j_1} & a_{i_{k-1} j_{l-1}} \end{pmatrix}, i_v = ik + v, 0 \le v < k, k = m/s, \; j_u = jl + u, 0 \le u \le l, l = n/q .$$

При таком подходе целесообразно, чтобы вычислительная система имела по крайней мере, логическую топологию процессорной решетки из *s* строк и *q* столбцов. В этом случае при разделении данных на непрерывной основе процессоры, соседние в структуре решетки, обрабатывают смежные блоки исходной матрицы. Следует отметить, однако, что и для блочной схемы может быть применено циклическое чередование строк и столбцов.

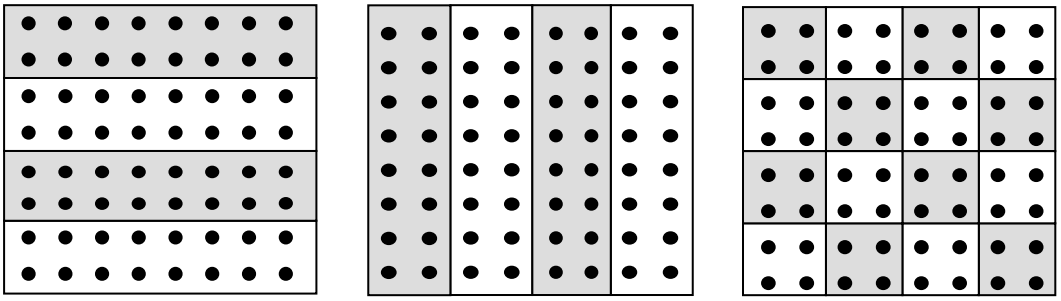

Рис. 1.7. Способы распределения элементов матрицы между процессорами вычислительной системы

Далее в лабораторной работе будет рассматриваться *алгоритм умножения матрицы на вектор, основанный на представлении матрицы непрерывными наборами (горизонтальными полосами) строк*. При таком способе разделения данных в качестве базовой подзадачи может быть выбрана *операция скалярного умножения одной строки матрицы на вектор*.

Выделение информационных зависимостей

В общем виде схема информационного взаимодействия подзадач в ходе выполняемых вычислений при ленточной схеме разделения данных показана на рис. 1.8.

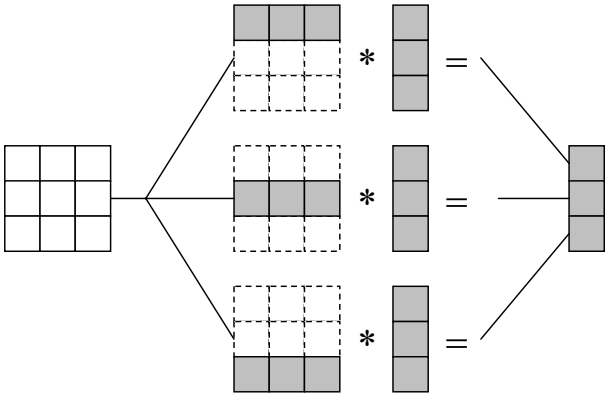

Рис. 1.8. Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на разделении матрицы по строкам

Масштабирование и распределение подзадач по вычислительным элементам

В процессе умножения плотной матрицы, разбитой на строки или столбцы, на вектор количество вычислительных операций для получения скалярного произведения одинаково для всех базовых подзадач. Поэтому в случае, когда число вычислительных элементов p меньше числа базовых подзадач m ($p<m$), возможно объединение базовых подзадач, для того чтобы каждый вычислительный элемент выполнял несколько таких задач, соответствующих непрерывной последовательности строк матрицы *pMatrix*. В этом случае по окончании вычислений каждая базовая подзадача определяет набор элементов результирующего вектора *pResult*.

Упражнение 4 – Реализация параллельного алгоритма

При выполнении этого упражнения Вам будет предложено разработать параллельный алгоритм умножения матрицы на вектор. При работе с этим упражнением Вы:

- Познакомитесь с основами OpenMP,
- Получите первый опыт разработки параллельных программ для вычислительных систем с общей памятью.

Задание 1 – Открытие проекта ParallelMatrixVectorMult

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

После того, как Вы открыли проект, в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **ParallelMV.cpp**, как это показано на рисунке 1.9. После этих действий код, который вам предстоит модифицировать, будет открыт в рабочей области Visual Studio.

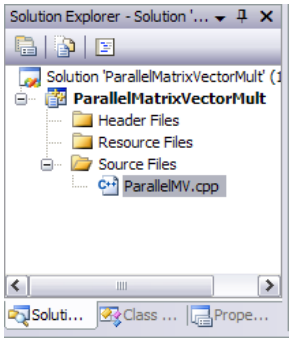

Рис. 1.9. Открытие файла ParallelMV.cpp с использованием Solution Explorer

В файле **ParallelMV.cpp** расположена главная функция (*main*) будущего параллельного приложения, которая содержит объявления необходимых переменных. Также в файле **ParallelMV.cpp** расположены функции, перенесенные сюда из проекта, содержащего последовательный алгоритм умножения матрицы на вектор: *DummyDataInitialization*, *RandomDataInitialization*, *SerialResultCalculation*, *PrintMatrix* и *PrintVector* (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-vector multiplication program`".

Задание 2 – Настройка проекта для использования OpenMP

Поддержка интерфейса OpenMP осуществляется компилятором. Для компиляции параллельного приложения, которое будет создано нами в рамках выполнения данного упражнения, необходимо использовать компилятор Intel C/C++ Compiler. При выполнении данного задания Вам предстоит настроить проект Visual Studio для использования компилятора Intel и возможностей OpenMP.

Прежде всего, проект Visual Studio необходимо конвертировать для использования компилятора Intel (по умолчанию в проектах Visual Studio используется компилятор Microsoft). Убедитесь, что на вашем компьютере установлен компилятор Intel. Далее щелкните правой кнопкой мыши на названии проекта в окне Solution Explorer и в появившемся контекстном меню выберете пункт **“Convert to use Intel(R) C++ Project System”**(см. рис. 1.10).

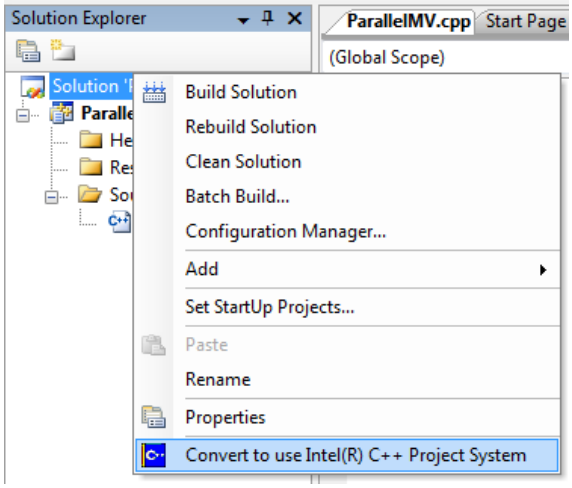

Рис. 1.10. Конвертация проекта использования компилятора Intel

В результате появиться предупреждение о том, что формат проекта изменится на формат, подходящий для компилятора. На это предупреждение надо ответить согласием, то есть нажать **“Yes”**(см. рис. 1.11).

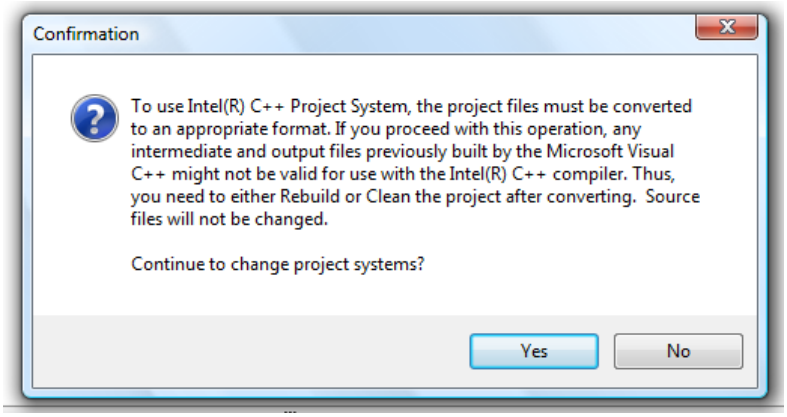

Рис. 1.11. Изменение формата проекта

Если все действия были проделаны правильно, то после их выполнения окно проекта Solution Explorer должно иметь вид, представленный на рисунке 1.12.

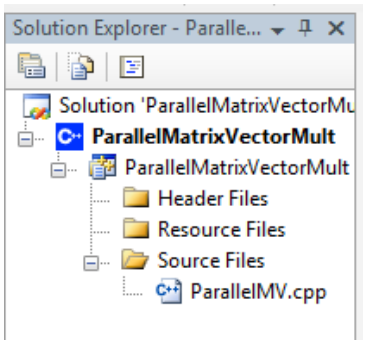

Рис. 1.12. Вид проекта преобразованного для использования Intel C++ Compiler

Далее необходимо указать, что при компиляции проекта будут использованы директивы OpenMP. Для этого нужно изменить одно из свойств проекта: откройте окно свойств проекта, щелкнув правой кнопкой мыши на названии проекта в окне Solution Explorer и выполнив команду **“Properties”**(см. рис. 1.13).

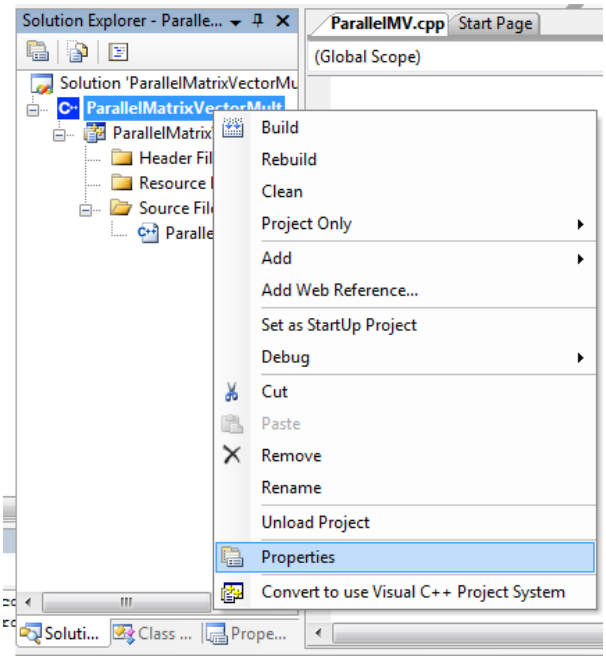

Рис. 1.13. Откройте окно свойств проекта

В результате у Вас должно открыться окно свойств проекта. В этом окне выберете вкладку "**Configuration Properties \ C/C++ \ Language**". В этой вкладке необходимо задать свойство "**OpenMP Support**" равным **"Generate Parallel Code (/openmp, equiv. to /Qopenmp)"**(см. рис. 1.14).

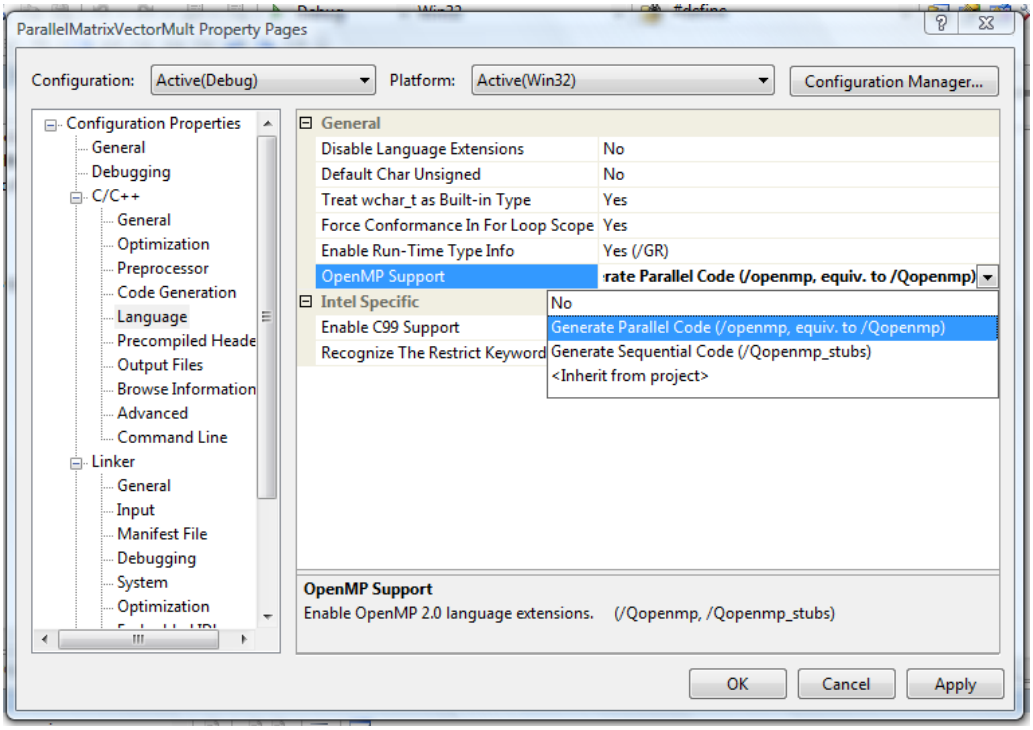

Рис. 1.14. Открытие файла ParallelMV.cpp с использованием Solution Explorer

Для завершения выполнения настроек проекта нажмите кнопку OK.

Задание 3 – Реализация параллельного алгоритма

Разработаем параллельную программу для умножения матрицы на вектор при ленточной схеме разделения данных по строкам. Напомним, что для организации параллельных вычислений предполагается использование многопроцессорных систем с общей памятью – тем самым, для разработки можно применить технологию OpenMP. В этом случае преобразовать разработанную ранее последовательную программу матрично-векторного умножения сравнительно просто – для этого

достаточно добавить директиву *parallel for* в функцию *SerialResultCalculation* (назовем новый вариант функции *ParallelResultCaclualtion*):

```
// Function for calculating matrix-vector multiplication
void ParallelResultCalculation (double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int i, j;  // Loop variables
#pragma omp paralell for private (j)
  for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++)
      pResult[i] += pMatrix[i*Size+j]*pVector[j];
  }
}
```

Данная функция производит умножение строк матрицы на вектор с использованием нескольких параллельных потоков. Каждый поток умножает горизонтальную полосу матрицы *pMatrix* на вектор *pVector* и вычисляет блок элементов результирующего вектора *pResult*.

Параллельные области в данной функции задаются директивой *parallel for*. Компилятор, поддерживающий технологию OpenMP, разделяет выполнение итераций цикла между несколькими потоками параллельной программы, количество которых обычно совпадает с числом вычислительных элементов (процессоров и/или ядер) в вычислительной системе. Параметр директивы *private* указывает необходимость создания отдельных копий для задаваемых переменных для каждого потока, данные копии могут использоваться в потоках независимо друг от друга.

Следует отметить, что матрица *pMatrix* и вектора *pVector* и *pResult* являются общими для всех потоков. Однако элементы матрицы *pMatrix* и вектора *pVector* используются только на чтение, а элементы вектора *pResult* между потоками разделяются без пересечения. Тем самым использование общих данных является корректным и организация параллельных вычислений выполнена правильно, без ошибок.

Задание 4 – Проверка правильности параллельной программы

Для проверки корректности работы параллельных программ разработайте функцию *TestResult*, которая сравнит результаты последовательного и параллельного алгоритмов. Для выполнения последовательного алгоритма можно использовать функцию *SerailResultCalculation*, разработанную в упражнении 2. Результат работы этой функции можно сохранить, например, в векторе *pSerialResult*, а затем поэлементно сравнить этот вектор с вектором *pResult*, полученным при помощи параллельного алгоритма. Далее представлен возможный вариант такой функции:

```
// Testing the result of parallel matrix-vector multiplication
void TestResult(double* pMatrix, double* pVector, double* pResult,
    int Size) {
  // Buffer for storing the result of serial matrix-vector multiplication
  double* pSerialResult;
  int equal = 0;  // Flag, that shows wheather the vectors are identical
  int i;          // Loop variable

  pSerialResult = new double [Size];
  SerialResultCalculation(pMatrix, pVector, pSerialResult, Size);
  for (i=0; i<Size; i++) {
    if (pResult[i] != pSerialResult[i])
      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 представленной функции является печать диагностического сообщения. Используя эту функцию, можно проверять результат работы параллельных алгоритмов независимо от того, насколько велики исходные объекты при любых значениях исходных данных. Так же функция не зависит от варианта используемого алгоритма и способа перемножения матрицы на вектор.

Вместо функции *DummyDataInitialization*, которая генерирует исходные объекты простого вида, вызовите функцию *RandomDataInitialization*, которая генерирует матрицу и вектор при помощи датчика случайных чисел. Скомпилируйте и запустите приложение. Задавайте различные объемы исходных данных. Убедитесь в том, что приложение работает корректно.

Задание 5 – Проведение вычислительных экспериментов

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

Определим время выполнения параллельного алгоритма. Для этого добавим в программный код замеры времени. Снова воспользуемся функцией *GetTime()* (эта функция подробно описана в упражнении 2). Добавьте выделенный программный код в функцию *main*:

```
ProcessInitialization(pMatrix, pVector, pResult, pProcRows, pProcResult,
  Size, RowNum);

double Start, Finish, Duration;

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:

Размер Матрицы	Последовательный алгоритм	Параллельный алгоритм			
		2 процессора (ядра)		4 процессора (ядра)	
		Время	Ускорение	Время	Ускорение
10					
100					
1000					
2000					
3000					
4000					
5000					
6000					
7000					
8000					
9000					
10 000					

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

Используем опять результаты Главы 7 учебных материалов курса. Время T_p выполнения параллельного алгоритма на компьютерной системе, имеющей p вычислительных элементов, может быть оценено при помощи следующего соотношения:

$$T_p = \frac{n \cdot (2n-1)}{p} \cdot \tau + \frac{8 \cdot n^2}{\beta}. \quad (1.3)$$

Вычислите максимальное теоретическое время выполнения параллельного алгоритма по формуле (1.3). Результаты занесите в таблицу:

Размер матрицы	2 процесса (ядра)		4 процесса (ядра)	
	Модель	Эксперимент	Модель	Эксперимент
10				
100				
1000				
2000				
3000				
4000				
5000				
6000				
7000				
8000				
9000				
10 000				

Контрольные вопросы

- Какие схемы разделения данных могут быть использованы при параллельном выполнении умножения матрицы на вектор?
- Какие возможности технологии OpenMP были использованы при разработке параллельной программы матрично-векторного умножения?
- Как обеспечивается правильность использования общих данных в разработанной параллельной программе?
- 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. Рассмотрите ленточную схему разделения матрицы по столбцам. Примените эту схему для разработки параллельного алгоритма умножения матрицы на вектор и выполните его программную реализацию. Проведите вычислительные эксперименты и оцените получаемое ускорение. Сравните полученные данные с результатами лабораторной работы.

2. Рассмотрите блочную схему разделения матрицы. Примените эту схему для разработки параллельного алгоритма умножения матрицы на вектор и выполните его программную реализацию. Проведите вычислительные эксперименты и оцените получаемое ускорение. Сравните полученные данные с результатами лабораторной работы.

3. Задание 2 может быть выполнено в усложненной постановке, при которой необходимо разработать два варианта параллельной программы с использованием и без применения механизма вложенных параллельных секций технологии OpenMP (см. Главу 7 учебных материалов курса). При выполнения такого усложненного задания следует сравнить сложность разработки разных вариантов программы и получаемую в результате эффективность параллельных вычислений.

Приложение 1. Программный код последовательного приложения для умножения матрицы на вектор

```
#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 definition of matrix and vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables

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

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

// Function for memory allocation and definition of object's elements
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int &Size) {
  // Size of initial matrix and vector definition
  do {
    printf("\nEnter size of the initial objects: ");
    scanf("%d", &Size);
    printf("\nChosen objects size = %d\n", Size);
      if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  }
  while (Size <= 0);
  // Memory allocation
  pMatrix = new double [Size*Size];
  pVector = new double [Size];
  pResult = new double [Size];
  // Definition of matrix and vector elements
  DummyDataInitialization(pMatrix, pVector, Size);
}

// 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 formatted vector output
void PrintVector (double* pVector, int Size) {
  int i;
  for (i=0; i<Size; i++)
    printf("%7.4f ", pVector[i]);
}

// Function for matrix-vector multiplication
void ResultCalculation(double* pMatrix, double* pVector, double* pResult,
  int Size) {
  int i, j;  // Loop variables
  for (i=0; i<Size; i++) {
    pResult[i] = 0;
      for (j=0; j<Size; j++)
        pResult[i] += pMatrix[i*Size+j]*pVector[j];
  }
}

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

void main() {
  double* pMatrix;  // The first argument - initial matrix
  double* pVector;  // The second argument - initial vector
  double* pResult;  // Result vector for matrix-vector multiplication
  int Size;           // Sizes of initial matrix and vector
  Double Start, Finish, Duration;

  printf("Serial matrix-vector multiplication program\n");
  // Memory allocation and definition of objects' elements
  ProcessInitialization(pMatrix, pVector, pResult, Size);

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

  // Matrix-vector multiplication
  Start = GetTime();
  ResultCalculation(pMatrix, pVector, pResult, Size);
  Finish = GetTime();
  Duration = Finish-Start;

  // Printing the result vector
  printf ("\n Result Vector: \n");
  PrintVector(pResult, Size);

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

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
}
```

Приложение 2 – Программный код параллельного приложения для умножения матрицы на вектор

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <windows.h>
#include <openmp.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 definition of matrix and vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables

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

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

// Function for memory allocation and definition of object's elements
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int &Size) {
  // Size of initial matrix and vector definition
  do {
    printf("\nEnter size of the initial objects: ");
    scanf("%d", &Size);
    printf("\nChosen objects size = %d\n", Size);
      if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  }
  while (Size <= 0);
  // Memory allocation
  pMatrix = new double [Size*Size];
  pVector = new double [Size];
  pResult = new double [Size];
  // Definition of matrix and vector elements
```

```
  DummyDataInitialization(pMatrix, pVector, Size);
}

// 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 formatted vector output
void PrintVector (double* pVector, int Size) {
  int i;
  for (i=0; i<Size; i++)
    printf("%7.4f ", pVector[i]);
}

// Function for serial matrix-vector multiplication
void SerialResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int i, j;  // Loop variables
  for (i=0; i<Size; i++) {
    pResult[i] = 0;
      for (j=0; j<Size; j++)
        pResult[i] += pMatrix[i*Size+j]*pVector[j];
  }
}

// Function for parallel matrix-vector multiplication
void ParallelResultCalculation (double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int i, j;  // Loop variables
#pragma omp paralell for private (j)
  for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++)
      pResult[i] += pMatrix[i*Size+j]*pVector[j];
  }
}

// Testing the result of parallel matrix-vector multiplication
void TestResult(double* pMatrix, double* pVector, double* pResult,
    int Size) {
  // Buffer for storing the result of serial matrix-vector multiplication
  double* pSerialResult;
  int equal = 0;  // Flag, that shows wheather the vectors are identical
  int i;          // Loop variable

  pSerialResult = new double [Size];
  SerialResultCalculation(pMatrix, pVector, pSerialResult, Size);
  for (i=0; i<Size; i++) {
    if (pResult[i] != pSerialResult[i])
      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;
```

```
}

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

void main() {
  double* pMatrix;  // The first argument - initial matrix
  double* pVector;  // The second argument - initial vector
  double* pResult;  // Result vector for matrix-vector multiplication
  int Size;           // Sizes of initial matrix and vector
  double Start, Finish, Duration;

  printf("Parallel matrix-vector multiplication program\n");
  // Memory allocation and definition of objects' elements
  ProcessInitialization(pMatrix, pVector, pResult, Size);

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

  // Matrix-vector multiplication
  Start = GetTime();
  ParallelResultCalculation(pMatrix, pVector, pResult, Size);
  Finish = GetTime();
  Duration = Finish-Start;
  TestResult(pMatrix, pVector, pResult, Size);

  // Printing the result vector
  printf ("\n Result Vector: \n");
  PrintVector(pResult, Size);

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

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
}
```