Министерство образования и науки РФ

Проект
«Создание системы подготовки высококвалифицированных кадров
в области суперкомпьютерных технологий и специализированного программного обеспечения»

К.В. Корняков, В.Д. Кустикова, И.Б. Мееров,
А.А. Сиднев, А.В. Сысоев, А.В. Шишков

ИНСТРУМЕНТЫ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ В СИСТЕМАХ С ОБЩЕЙ ПАМЯТЬЮ

Раздел 3. Распараллеливание циклов с использованием библиотеки Intel Threading Building Blocks на примере задачи матрично-векторного умножения

Лабораторная работа 4

Разработчик: А.А. Сиднев

Нижегородский государственный университет им. Н.И. Лобачевского
2010

7. Лабораторная работа. Распараллеливание циклов с использованием библиотеки Intel Threading Building Blocks на примере задачи матрично-векторного умножения

7.1. Введение

В настоящей лабораторной работе рассматривается один из инструментов параллельного программирования, предназначенный для распараллеливания решения задач в системах с общей памятью, – библиотека Intel Threading Building Blocks (TBB) [**Ошибка! Источник ссылки не найден.**]. Основная идея, заложенная в библиотеку, состоит в использовании стандартного высокоуровневого C++ для быстрой разработки кросс-платформенных, хорошо масштабируемых параллельных приложений (см. подробнее в [1]). Наряду с указанным выше, использование библиотеки TBB предоставляет механизмы абстрагирования от парадигм многопоточного программирования, позволяя сосредоточиться непосредственно на решении прикладной задачи, что является достаточно актуальным.

Изучение принципов функционирования и вопросов эффективного использования TBB проводится в данной лабораторной работе на примере задачи матрично-векторного умножения, ставшей одной из классических задач для формирования навыков параллельного программирования. При этом основной упор делается на ознакомление с распараллеливанием циклов.

В ходе выполнения работы предполагается, что слушатели имеют навыки разработки объектно-ориентированных программ на C++, а также владеют основами параллельного программирования в системах с общей памятью.

7.2. Методические рекомендации

7.2.1. Цели и задачи работы

Целью данной лабораторной работы является приобретение практических навыков распараллеливания циклов для систем с общей памятью при помощи библиотеки TBB.

Данная цель предполагает решение следующих задач:

- изучение способов инициализации и завершения работы библиотеки TBB (см. также [1]);
- освоение функциональности библиотеки TBB, связанной с распараллеливанием циклов (см. также [1]);
- рассмотрение учебной задачи, направленной на демонстрацию принципов распараллеливания циклов при помощи библиотеки TBB;
- самостоятельная разработка и отладка параллельных программ, реализующих распараллеливание циклов при помощи TBB.

7.2.2. Структура работы

Лабораторная работа состоит из введения, двух разделов, списка дополнительных заданий и списка литературы. Во введении обосновывается актуальность использования TBB в процессе разработки параллельных программ. В первом разделе приводятся методические рекомендации к лабораторной работе: формулируются цели и задачи, системные требования, рекомендации по проведению занятий. Во втором разделе изучается применение библиотеки TBB для распараллеливания циклов. Изучение проводится на примере задачи матрично-векторного умножения. В заключение приводятся задания для самостоятельной проработки, а также использованная и рекомендуемая литература.

Рассмотрение задачи матрично-векторного умножения проводится в соответствии с материалами учебного курса [8], разработанного в ННГУ (руководитель авторского коллектива д.т.н., проф. В.П. Гергель).

7.2.3. Системные требования

Приведем системные требования для Windows-систем. Аналогичную информацию для систем на базе Linux и Mac OS можно найти на официальном сайте TBB [**Ошибка! Источник ссылки не найден.**].

Аппаратное обеспечение

Минимальные требования

- Intel® Pentium® 4 процессор;
- 512 Мб ОЗУ;
- 300 Мб дискового пространства.

Рекомендуемые требования

- Intel Pentium 4 процессор с поддержкой технологии Hyper-Threading (HT Technology) или Intel® Xeon® процессор;
- 1 Гб ОЗУ.

Для изучения всех аспектов «реальных» параллельных вычислений желательно использование многоядерных процессоров компании Intel.

Программное обеспечение

- Microsoft Windows XP Professional, Microsoft Windows Server 2003 или Microsoft Windows Vista;
- Intel® C++ Compiler 9.0 for Windows или старше;
- Microsoft Visual C++ 7.1 или старше;
- Microsoft Internet Explorer 6.0 или старше;
- Adobe Reader 6.0 или старше.

7.2.4. Рекомендации по проведению занятий

При выполнении данной лабораторной работы рекомендуется придерживаться следующей последовательности изучения материала:

1. Изучить способы инициализации и завершения работы библиотеки TBB (см. также [1]).
2. Рассмотреть функциональность библиотеки TBB, связанную с распараллеливанием циклов (см. также [1]).

3. Разобрать один из вариантов решения учебной задачи – задачи умножения матрицы на вектор, демонстрирующей принципы распараллеливания циклов при помощи библиотеки TBB.
4. Перейти к выполнению дополнительных заданий, состоящих в самостоятельной разработке и отладке параллельных программ, реализующих распараллеливание циклов при помощи TBB.

7.3. Постановка задачи[1]

Имея несложную математическую постановку, задача матрично-векторного умножения является классической задачей для приобретения навыков разработки, отладки и оптимизации как последовательных, так и параллельных программ. Такая популярность задачи вызвана, прежде всего, богатыми возможностями демонстрации типовых подходов к организации процесса вычислений, а также эффективной реализации этих подходов средствами языка программирования. В данной лабораторной работе мы рассмотрим один из возможных подходов к решению задачи, его реализацию в виде последовательной программы на C++, а также на его примере познакомимся с принципами распараллеливания при помощи библиотеки TBB. При этом в основном будет затронута часть функциональности TBB, связанная с распараллеливанием циклов.

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

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

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 (n операций) и последующее суммирование полученных произведений (n – 1 операций). Общее количество необходимых скалярных операций есть величина $T_1 = m(2n - 1)$ [8].

7.4. Последовательный алгоритм

Спецификации последовательного алгоритма умножения матрицы на вектор могут быть представлены следующим образом:

Исходные данные: A[m][n] – матрица размерности $m \times n$.
b[n] – вектор, состоящий из n элементов.

---

[1] Часть данного материала, посвященная созданию последовательного приложения (задания 1–5 раздела 7.5), написана на основе раздела 7 курса [1], а также лабораторной работы «Умножение матрицы на вектор» с незначительными изменениями, преимущественно касающимися другого способа реализации замеров времени. Алгоритм параллельного умножения матрицы на вектор, рассматривающий матрицу как набор строк, изложен на основе раздела 7 курса [1]. Наличие подобных материалов в данной лабораторной работе прежде всего обосновано возможностью ее использования как отдельного методического материала.

Результат: c[n] – вектор из m элементов.

Алгоритм вычисления результата матрично-векторного умножения полностью описывается его формулой, поэтому сразу приведем код:

```
// Последовательный алгоритм умножения матрицы на вектор
for (i = 0; i < m; i++)
{
  c[i] = 0;
  for (j = 0; j < n; j++)
  {
    c[i] += A[i][j] * b[j];
  }
}
```

Матрично-векторное умножение – это последовательность вычисления скалярных произведений. Поскольку каждое вычисление скалярного произведения векторов длины n требует выполнения n операций умножения и n–1 операций сложения, его трудоемкость порядка $O(n)$. Для выполнения матрично-векторного умножения необходимо выполнить m операций вычисления скалярного произведения, таким образом, алгоритм имеет трудоемкость порядка $O(mn)$.

При дальнейшем изложении материала для снижения сложности и упрощения получаемых соотношений будем предполагать, что матрица A является квадратной, т.е. $m = n$.

Проведем пошагово реализацию последовательной программы умножения матрицы на вектор. Заметим, что вариант, который будет рассмотрен, разумеется, не является единственным. Будем также считать основной целью корректность работы полученной реализации, а не ее производительность на какой-либо конкретной архитектуре.

7.5. Реализация последовательного алгоритма

При выполнении этого упражнения необходимо реализовать последовательный алгоритм матрично-векторного умножения. Начальный вариант будущей программы представлен в проекте **`SerialMatrixVector`**, который содержит часть исходного кода и в котором заданы необходимые параметры проекта. В ходе данного 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 SerialMatrixVector

Откройте проект **SerialMatrixVector**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\TBBLabs\Serial MatrixVector**;
- 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. После этих действий программный код, который предстоит в дальнейшем расширить, будет открыт в редакторе кода **Microsoft Visual Studio 2005**.

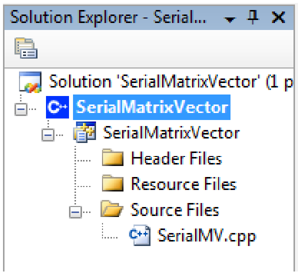

Рис. 1. Открытие файла **SerialMV.cpp**

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 в соответствии с соглашениями языка C. Таким образом, элемент, расположенный на пересечении *i*-й строки и *j*-го 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 **Microsoft Visual Studio 2005** 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 (исходной матрицы **pMatrix** и вектора **pVector**, и результата умножения **pResult**), а также для задания значений элементов исходной 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 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. 2).

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
  // Result vector for matrix-vector multiplication
  double* pResult;
  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.

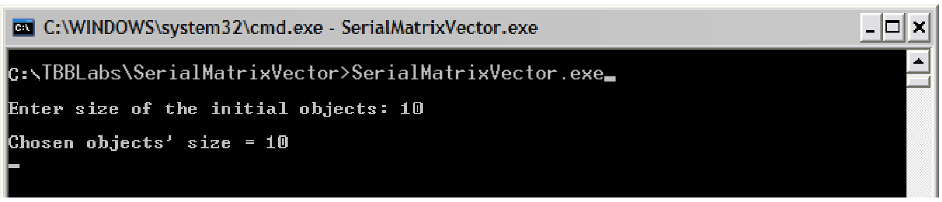

Рис. 2. Задание размера объектов

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 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 должна также выделять память 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 (формирование исходных данных при помощи датчика случайных чисел будет рассмотрено в задании 6):

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

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, – 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. 3). Выполните несколько запусков приложения, задавая различные размеры объектов.

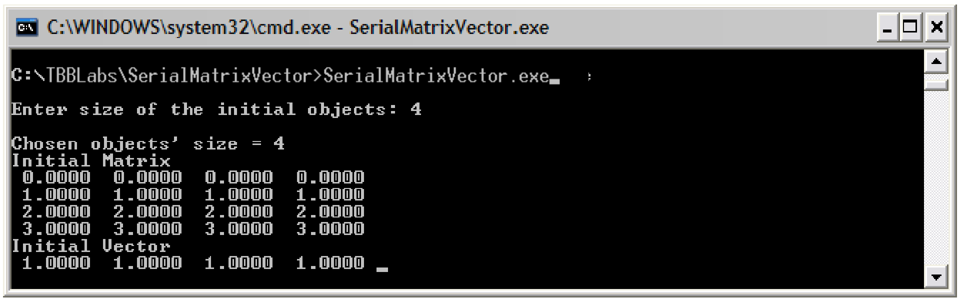

Рис. 3. Результат работы программы при завершении задания 3

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 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);

  // 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 ранее, 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 код может быть оптимизирован, 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*-й 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.

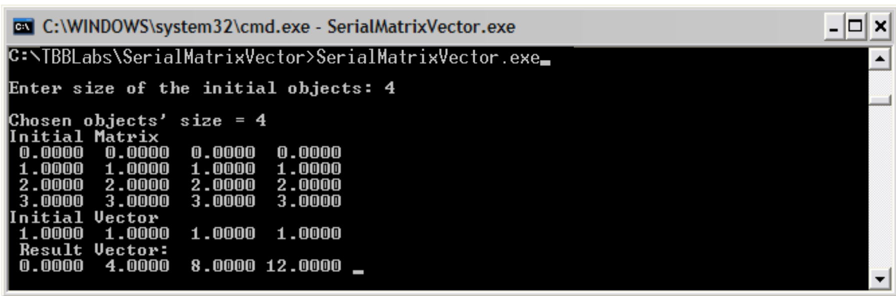

Рис. 4. Результат выполнения матрично-векторного умножения

Задание 6. Проведение вычислительных экспериментов

После реализации параллельной версии алгоритма нам потребуется оценивать ее ускорение. Для этого сейчас необходимо провести вычислительные эксперименты и замерить времена работы последовательной версии. Анализировать время выполнения последовательной реализации разумно 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);
  }
}
```

Заметим, что в отладочных целях следует инициализировать датчик случайных чисел при помощи функции **srand()** одним и тем же числом, что обеспечит нам одинаковые последовательности случайных чисел и позволит обнаружить ошибки, если они есть.

Будем вызывать эту функцию вместо ранее разработанной функции **DummyDataInitialization**, которая генерировала данные так, чтобы 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 С, которая позволяет узнать время работы программы или ее 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);
```

Отметим, что представленная реализация функции **`GetTime`** не претендует на оптимальность и может быть улучшена. Сделать это предоставляем читателям самостоятельно.

Библиотека TBB реализует собственный способ замера времени, сочетающий высокую разрешающую способность с простотой использования. Для этого предусмотрен специальный класс **`tick_count`**, содержащий статический метод **`tick_count::now()`**, возвращающий текущее время как объект класса **`tick_count`**. Как и раньше, для измерения времени необходимо вызвать метод **`now()`** в начале и в конце фрагмента программы с вычислительной частью, после чего вычесть из второго значения первое (операция вычитания для класса **`tick_count`** перегружена). Полученный результат можно перевести в секунды при помощи метода **`tick_count::seconds()`**.

Отметим, что в реализации библиотеки TBB для ОС Windows функция **`tick_count::now()`** вызывает использованный нами **`QueryPerformance Counter`**.

Чтобы воспользоваться описанным способом замера времени, необходимо подключить заголовочный файл tbb/tick_count.h.

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`**:

```
  tick_count Start, Finish;

  // Matrix-vector multiplication
  Start = tick_count::now();
  SerialResultCalculation(pMatrix, pVector, pResult, Size);
  Finish = tick_count::now();
  Duration = (Finish - Start).seconds();

  // 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). Убедитесь, что выбрана конфигурация **Release**. Проведите вычислительные эксперименты.

Задание 7. Оценка эффективности

Запустите программу несколько раз, задавая одни и те же размеры исходных данных. Нетрудно видеть, что время работы меняется от запуска к запуску. Эффект вызван особенностями работы программ под управлением многозадачной операционной системы и погрешностью механизма измерения. В связи с этим существуют несколько методик организации замеров времени. Мы в данной лабораторной работе рекомендуем выполнить приложение несколько раз и выбрать минимальное из полученных времен работы при одних и тех же размерах матрицы и вектора.

Проведите эксперименты, результаты занесите в таблицу 1.

Таблица 1

Время работы последовательного приложения

Номер теста	Параметр Size	Время работы последовательного приложения (сек)
1	250	
2	500	
3	1000	
4	2000	
5	4000	
6	8000	
7	10000	

7.6. Параллельный алгоритм[2]

7.6.1. Принципы распараллеливания

Разработка алгоритмов (а в особенности – методов параллельных вычислений) для решения сложных научно-технических задач часто представляет собой значительную проблему. В нашем случае будем полагать, что вычислительная схема решения задачи умножения матрицы на вектор уже известна. Действия для определения эффективного способа 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.

[2] По материалам лабораторной работы 1 к разделу 7 курса [8].

- Выполнить распределение выделенных *подзадач* по вычислительным элементам (процессорам, ядрам).

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

7.6.2. 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*.

Общая характеристика распределения данных для матричных алгоритмов содержится в разделе 7 учебного курса [8]. Наиболее общие и широко используемые способы разделения матриц состоят в разбиении данных на *полосы* (по вертикали или горизонтали) или на прямоугольные фрагменты (*блоки*).

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*.

7.6.3. Выделение информационных зависимостей

В общем виде схема информационного взаимодействия подзадач в ходе выполняемых вычислений при ленточной схеме разделения данных показана на рис. 5.

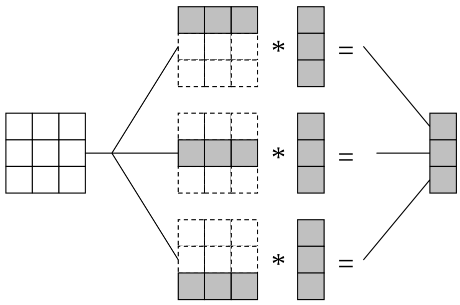

Рис. 5. Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на разделении матрицы по строкам

7.6.4. 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`**.

7.7. Реализация параллельного алгоритма

При выполнении этого упражнения необходимо реализовать параллельный алгоритм матрично-векторного умножения на основе имеющейся реализации последовательного алгоритма. При этом будут рассмотрены следующие возможности библиотеки TBB:

- инициализация библиотеки;
- настройка проекта для работы с библиотекой;
- использование алгоритма **`tbb::parallel_for`**.

Кроме перечисленного выше будет изучено влияние значений параметров библиотеки на время выполнения вычислений.

Задание 1. Открытие проекта ParallelMatrixVectorMult

Откройте проект **ParallelMatrixVectorMult**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005**;
- в меню **File** выполните команду **Open→Project/Solution**;
- в диалоговом окне **Open Project** выберите папку **C:\TBBLabs\ParallelMatrix Vector**;
- дважды щелкните на файле **ParallelMatrixVectorMult.sln** или подсветите его выполните команду **Open**.

После открытия проекта в окне **Solution Explorer** дважды щелкните на файле исходного кода **ParallelMV.cpp**, как это показано на рис. 6. После этих действий код, который вам предстоит модифицировать, будет открыт в редакторе кода **Microsoft Visual Studio 2005**.

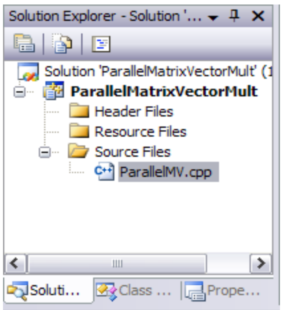

Рис. 6. Открытие файла ParallelMV.cpp

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`**. Эти функции можно будет использовать и в параллельной программе.

Скомпилируйте и запустите приложение стандартными средствами **Microsoft Visual Studio 2005**. Убедитесь в том, что в командную консоль выводится приветствие: "`Parallel matrix-vector multiplication program`".

Задание 2. Настройка проекта для использования TBB

Первое, что нам предстоит изменить, – это подсказать **Microsoft Visual Studio 2005** место расположения заголовочных файлов библиотеки **TBB**. Эта задача может быть решена двумя способами:

- Запустить файл **C:\Program Files\Intel\<TBB Directory>\<arch>\vc8\bin\ tbbvars.bat**, где <arch> принимает значения ia32 (Intel® IA-32 processors) или em64t (Intel® EM64T processors).
- В меню **Tools** выбрать пункт **Options**. В открывшемся окне выбрать пункт **Projects and Solutions\VC++ Directories**. В выпадающем списке **Show directories for** выбрать пункт **Include files**. Нажать левой кнопкой мыши на изображении папки и указать путь к папке **Include** библиотеки **TBB**: **C:\Program Files\Intel\<TBB Directory>\include**. Нажать **OK**.

Следующий шаг – указание пути к статической библиотеке **TBB**, с которой будет собираться наше приложение. Таких библиотек две: **tbb_debug.lib** и **tbb.lib**. Первая библиотека выполняет различные проверки во время выполнения приложения и полностью поддерживается профилировщиком **Intel Thread Profiler**, предназначена для компиляции и сборки отладочных версий программ. Вторая библиотека имеет гораздо более эффективную реализацию функций и методов и предназначена для компиляции и сборки финальных версий. Разумеется, имеет смысл подключать одну из двух библиотек в зависимости от ситуации (отладка, финальная сборка).

Для подключения библиотеки необходимо выполнить следующую последовательность действий:

- В меню **Tools** выбрать пункт **Options**. В открывшемся окне выбрать **Projects and Solutions\VC++ Directories**. В выпадающем списке **Show directories for** выбрать пункт **Library files**. Нажать левой кнопкой мыши на изображении папки и указать путь к папке **lib** библиотеки **TBB**: **C:\Program Files\Intel\<TBB Directory>\vc8\lib**. Нажать **OK**.
- В окне **Solution Explorer** нажать правой кнопкой мыши на названии проекта (**ParallelMatrixVectorMult**) и выбрать пункт **Properties**.
- Выбрать пункт **Linker\Input** и в поле **Additional Dependencies** ввести название библиотеки: **tbb_debug.lib** (для **Debug**-сборки) или **tbb.lib** (для **Release**-сборки).

Задание 3. Инициализация библиотеки TBB

Библиотека TBB является библиотекой классов, соответственно, вся (или почти вся) функциональность реализована посредством классов.

Началу работы с библиотекой предшествует обязательная процедура инициализации. Для этого необходимо создать экземпляр класса **`task_scheduler_init`**. Конструктор этого класса в зависимости от полученных параметров выполняет следующие действия:

- Указывает библиотеке TBB на необходимость автоматически определить число создаваемых потоков (**`task_scheduler_init::automatic`** является значением по умолчанию).
- Указывает библиотеке на необходимость создать нужное разработчику число потоков (параметр конструктора – число потоков).
- Указывает библиотеке на необходимость отложить инициализацию до момента, когда это будет необходимо разработчику (**`task_scheduler_init::deferred`**). В этом случае сама инициализация происходит только после вызова метода **`task_scheduler_init::initialize`** с параметром *n* – число потоков, которое необходимо разработчику.

При выполнении данной лабораторной работы мы рекомендуем придерживаться схемы, подразумевающей инициализацию планировщика потоков в начале программы с параметром по умолчанию (автоматическое определение числа потоков) и завершение работы библиотеки в конце программы. Достаточно часто предлагаемая схема обеспечивает наилучшую производительность.

Добавим в наш код инициализацию и завершение работы библиотеки. Для этого подключим заголовочный файл tbb/task_scheduler_init.h, выполним создание и уничтожение объекта – инициализатора библиотеки.

```
#include "tbb/task_scheduler_init.h"
<...>
void main()
{
  double* pMatrix;  // The first argument - initial matrix
  double* pVector;  // The second argument - initial vector
  // Result vector for matrix-vector multiplication
  double* pResult;
  int Size;         // Sizes of initial matrix and vector

  <...>

  task_scheduler_init init;

  <...>
  return 0;
}
```

Обратите внимание, что завершение работы библиотеки происходит при выходе из функции **`main`** в результате автоматического вызова деструктора локального объекта **`init`**.

Выполните компиляцию и сборку. Убедитесь, что сообщения об ошибках отсутствуют.

Задание 4. Распараллеливание цикла средствами библиотеки ТВВ

Разработаем параллельную программу для умножения матрицы на вектор при ленточной схеме разделения данных по строкам. Для организации параллельных вычислений предполагается использование многопроцессорных систем с общей памятью, что позволяет нам использовать ТВВ.

Параллельный алгоритм состоит в умножении строк матрицы на вектор с использованием нескольких потоков. Будем понимать далее под *итерацией* процедуру вычисления скалярного произведения строки матрицы на вектор. Тогда каждый поток выполняет несколько таких итераций, фактически, умножая горизонтальную полосу матрицы **pMatrix** на вектор **pVector** и вычисляя блок элементов результирующего вектора **pResult**. Заметим полное отсутствие зависимости по данным между разными итерациями, что создает предпосылки для хорошего распараллеливания.

Заметим также, что матрица **pMatrix** и векторы **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 не составляет больших проблем.

Итак, схема вычислений может быть представлена в виде цикла, итерацией которого является скалярное произведение. Библиотека ТВВ предоставляет возможность легко реализовать параллельную версию таких вычислений. Для этого в библиотеке имеется шаблонная функция **tbb::parallel_for**.

```
template<typename Range, typename Body>
void parallel_for(const Range& range, const Body& body);
```

Первый аргумент этой функции – **range** – итерационное пространство, к которому необходимо применить метод **body::operator()**. Второй аргумент функции – функтор.

Подробное описание того, что такое итерационное пространство, может быть найдено в разделе 6.5. Здесь же мы лишь напомним, что итерационное пространство в ТВВ определяет способ разделения общих данных между параллельно работающими потоками. Необходимое нам в этой задаче одномерное итерационное пространство (действительно, мы имеем **Size** итераций, состоящих в вычислении скалярного произведения) моделируется шаблонным классом **blocked_range**.

```
template<typename Value> class blocked_range;
```

Такое пространство задает диапазон значений в виде полуинтервала **[begin, end)**, с размером порции (количеством итераций), обрабатываемым одним потоком, равным **grainsize**.

В нашей задаче имеет смысл считать, что **begin = 0**, **end = Size**. Задав **grainsize = 1** (по умолчанию), мы реализуем максимально возможный параллелизм, так как библиотека сможет при наличии достаточного числа процессоров/ядер создать число потоков, равное **Size**. Отметим также, что в реальной ситуации (когда число процессоров/ядер много меньше **Size**) в отличие от OpenMP задание **grainsize = 1** не означает, что потоки будут обрабатывать итерации в каком бы то ни было заранее нам известном регулярном порядке. Решение о том, какая итерация каким потоком будет выполняться, принимает библиотека. Средств управления порядком выполнения в ТВВ нет. Более подробную информацию по данному вопросу см. в разделе 6.5.

Задание параметров выполняется при помощи следующего конструктора:

```
blocked_range(Value begin, Value end, size_t grainsize = 1);
```

О выборе оптимального значения параметра **grainsize** мы поговорим чуть позже, а сейчас перейдем к рассмотрению второго параметра функции **parallel_for** – функтора **body**.

В применении к **parallel_for** функтор определяет действие, которое должно быть выполнено потоком над пакетом итераций размером **grainsize**. Функтор для **parallel_for** должен удовлетворять следующим требованиям:

- в качестве полей класса иметь все общие для потоков данные;
- иметь конструктор-инициализатор;
- иметь конструктор копирования;
- иметь деструктор;
- иметь константный метод **operator()**, получающий константную ссылку на итерационное пространство и реализующий вычисления.

Функция **parallel_for** выполняет разделение итерационного пространства (**range**) на непересекающиеся подмножества размером **grainsize** и вызывает для каждого подмножества метод **body::operator()** в отдельном потоке. Заметим, что перед запуском вычислений **parallel_for** создает копию функтора для каждого потока, что требует корректно реализованного конструктора копирования. После завершения работы **parallel_for** уничтожает созданные объекты (что требует наличия деструктора).

Приступим к реализации функтора для решаемой задачи. Напомним, что функтор реализует итерации, то есть в рамках выбранной схемы вычислений подсчитывает скалярные произведения векторов.

```
class MatrixVectorMultiplicator
{
  const double *pMatrix,
               *pvector,
               *pResult;
  const int Size;

public:
  void operator()( const blocked_range<int>& r ) const
  {
    int begin = r.begin();
    int end   = r.end();

    for (int i = begin; i != end; i++)
    {
      pResult[i] = 0;
      for (int j = 0; j < Size; j++)
        pResult[i] += pMatrix[i * Size + j] * pVector[j];
    }
  }

  MatrixVectorMultiplicator(double *pm, double *pv, double *pr,
    int sz): pMatrix(pm), pVector(pv), pResult(pr), Size(sz)
  {}
};
```

Функтор содержит поля, в которых хранятся значения переменных, участвующих в вычислениях в методе **operator()**. Инициализация этих полей осуществляется с помощью конструктора. Конструктор копирования и деструктор, создаваемые компилятором по умолчанию, в данном случае корректны, поэтому их реализация в явном виде не приводится. Значения начальной и конечной итераций (**r.begin()**, **r.end()**) считаются вне тела цикла. Это сделано для того, чтобы эти функции не вызывались на каждой итерации, т. к. это может привести к замедлению работы приложения.

Реализуем функцию параллельного вычисления умножения матрицы на вектор при помощи функции **parallel_for** из библиотеки TBB:

```
void ParallelResultCalculation (double* pMatrix, double* pVector,
  double* pResult, int Size, int grainsize)
{
  parallel_for(blocked_range<int>(0, Size, grainsize),
    MatrixVectorMultiplicator(pMatrix, pVector, pResult, Size));
}
```

Параметр **grainsize** у разработанной функции предусмотрен для будущих экспериментов с размером порции итераций.

Добавим вызов функции параллельного умножения матрицы на вектор в функцию **main** (не забудем подключить еще 2 заголовочных файла). В качестве значения **grainsize** укажем размер вектора.

```
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"

<...>

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

  <...>
  task_scheduler_init init;
  <...>

  ParallelResultCalculation(pMatrix, pVector, pResult,Size, Size);
  PrintVector(pResult, Size);

  <...>

  return 0;
}
```

Задание 5. Проведение вычислительных экспериментов

Выполните сборку и запуск приложения. Убедитесь в отсутствии ошибок и правильной работе приложения. Убедитесь, что время работы параллельного приложения осталось примерно таким же, как и у последовательного.

Задание 6. Выбор значения параметра grainsize

Параметр **`grainsize`** задает число итераций из итерационного пространства, которое составит размер порции вычислений для передачи отдельному потоку. Итерационное пространство распределяется между всеми потоками в зависимости от значения параметра **`grainsize`**. Принцип распределения описан в разделе 6.5 в пункте «Планирование вычислений».

Выбор значения параметра **`grainsize`** сильно влияет на производительность параллельной программы. При этом невозможно указать какое-то одно «замечательное» значение, которое подойдет для всех задач. На практике **`grainsize`** подбирается экспериментально.

Руководствуясь описанным в пункте «Планирование вычислений» раздела 6.5 алгоритмом, реализуем программный код для подбора значения параметра **`grainsize`**. Начальное значение **`grainsize`** возьмем равным **`Size`**. Далее, в цикле, будем уменьшать это значение в 2 раза, пока оно не станет равным 1. В зависимости от входных данных (**`Size`**) желаемое значение **`grainsize`** будет меняться.

```
int main()
{
  double *pMatrix = NULL,
         *pVector = NULL,
         *pResult = NULL;

  int Size = 0;

  task_scheduler_init init(1);

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

  ProcessInitialization(pMatrix, pVector, pResult, Size);

  //Computing experiments
  const int repeatCount = 64;
  tick_count startTime;
  double duration;

  for (int grainsize = Size; grainsize >= 1; grainsize /= 2)
  {
    double minDuration = DBL_MAX;

    // Matrix-vector multiplication. Parallel version based on TBB
    // ...

    printf("Grain size is %d, execution time is %f s.\n",
            grainsize, minDuration);
  }

  ProcessTermination(pMatrix, pVector, pResult);
  getch();

  return 0;
}
```

Выполните сборку и запуск приложения. Определите, как меняется время работы при уменьшении **grainsize** (рис. 7). Определите желаемое значение **grainsize**.

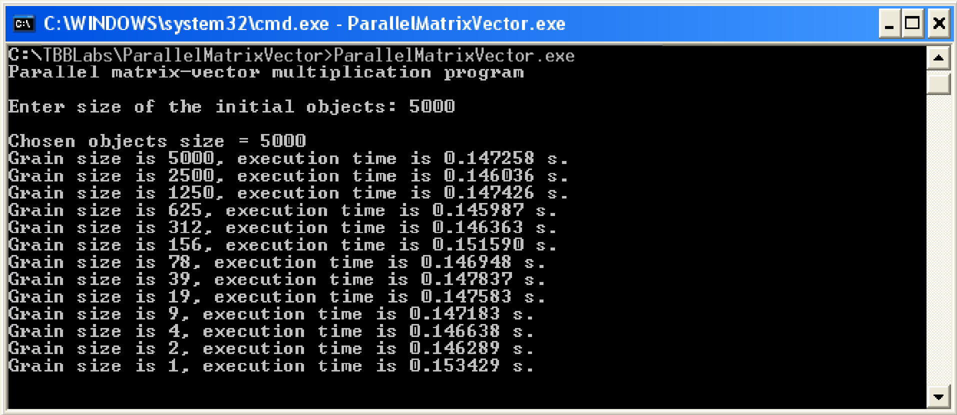

Рис. 7. Результаты эксперимента по подбору значения **grainsize**

Руководствуясь приведенным выше алгоритмом по подбору значения **grainsize** и результатами вычислительного эксперимента, представленными на рис. 7, можно сделать вывод, что даже минимальное значение **grainsize = 1** будет подходящим для задачи умножения квадратной матрицы на вектор.

Задание 7. Оценка эффективности

Выполните сборку приложения в конфигурации **Release**. Запустите программу несколько раз, задавая одни и те же размеры исходных данных. Как и ранее, мы рекомендуем выполнить приложение несколько раз и выбрать минимальное из полученных времен работы при одних и тех же размерах матрицы и вектора.

Заполните недостающие столбцы таблицы 2.

Таблица 2

Время работы параллельного приложения

Номер теста	Параметр Size	Время работы последовательного приложения (сек)	Время работы параллельного приложения (сек)	Выбранное значение **grainsize**
1	250			
2	500			
3	1000			
4	2000			
5	4000			
6	8000			
7	10000			

На основе значений, записанных в таблицу 2, посчитайте ускорение вычислений и заполните таблицу 3.

Таблица 3

Ускорение вычислений

Номер теста	Параметр Size	Ускорение параллельного алгоритма
1	250	
2	500	
3	1000	
4	2000	
5	4000	
6	8000	
7	10000	

Проведите вычислительные эксперименты для задачи с фиксированными значениями исходных данных (4000×4000) с различными значениями **`grainsize`** и заполните два первых столбца таблицы 4. Выберите желаемое значение **`grainsize`** и заполните последнее поле в таблице 4.

Таблица 4

Подбор значения параметра `grainsize`

Номер теста	Значение **`grainsize`**	Время работы последовательного приложения (сек)	Время работы параллельного приложения (сек)	Выбранное значение **`grainsize`**
1	4000			
2	1000			
3	250			
4	64			
5	16			
6	4			

7.8. Контрольные вопросы

5. Какие факторы влияют на время работы приложения?
6. Как собрать приложение, использующее библиотеку TBB?
7. Как инициализировать библиотеку TBB?
8. Как завершить работу библиотеки TBB?
9. Как инициализировать библиотеку TBB с определенным числом программных потоков?
10. Каким образом организовано распараллеливание цикла **`for`** в библиотеке TBB?
11. Что такое функтор?
12. Что такое итерационное пространство?
13. Каков смысл параметра **`grainsize`**?
14. Как работает конструкция **`parallel_for`** в библиотеке TBB?
15. Какие методы функтора необходимо реализовать при использовании алгоритма **`parallel_for`**?
16. Как выбрать значение параметра **`grainsize`**?

7.9. Задания для самостоятельной работы

1. Измените созданную реализацию на случай неквадратной матрицы.
2. Разработайте программу умножения квадратных матриц.
3. Разработайте программу умножения прямоугольных матриц.

7.10. Литература

Использованные источники

1. Гергель В.П., Лабутина А.А. Умножение матрицы на вектор // Материалы образовательного комплекса «Параллельное программирование на OpenMP». Нижний Новгород, 2007.
2. Сиднев А.А., Сысоев А.В., Мееров И.Б. Библиотека Intel Threading Building Blocks – краткое описание // Материалы образовательного комплекса «Технологии разработки параллельных программ». Нижний Новгород, 2007.
3. Intel® Threading Building Blocks. Reference Manual. Version 1.6. – Intel Corporation, 2007.
4. Intel® Threading Building Blocks. Tutorial. Version 1.6. – Intel Corporation, 2007.

Рекомендуемая литература

5. Andrews G.R. Foundations of Multithreaded, Parallel, and Distributed Programming. Reading, MA: Addison-Wesley, 2000 (русский перевод: Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. М.: Издательский дом «Вильямс», 2003).
6. Quinn M.J. Parallel Programming in C with MPI and OpenMP. New York, NY: McGraw-Hill, 2004.
7. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002.
8. Гергель В.П. Теория и практика параллельных вычислений. М.: Интернет-Университет Информационных технологий; БИНОМ. Лаборатория знаний, 2007.
9. Немнюгин С.А, Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. СПб.: БХВ-Петербург, 2002.

Дополнительная литература

10. Березин И.С., Жидков И.П. Методы вычислений. М.: Наука, 1966.
11. Майерс С. Эффективное использование C++. 35 новых способов улучшить стиль программирования. СПб.: Питер, 2006.
12. Майерс С. Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ. М.: ДМК Пресс, 2006.
13. Павловская Т.А. C/C++. Программирование на языке высокого уровня. СПб.: Питер, 2003.

14. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows: Пер. с англ. 4-е изд. СПб.: Питер; М.: Издательско-торговый дом «Русская редакция», 2001.

Информационные ресурсы сети Интернет

15. Страница библиотеки TBB на сайте корпорации Intel – [http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm].
16. Сайт сообщества пользователей TBB – [http://threadingbuildingblocks.org].
17. Сайт Лаборатории параллельных информационных технологий НИВЦ МГУ – [http://www.parallel.ru].
18. Официальный сайт OpenMP – [www.openmp.org].

7.11. Приложения

7.11.1. Программный код последовательного умножения матрицы на вектор

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <windows.h>
#include "tbb/tick_count.h"

using namespace tbb;

// 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 Duration;
  tick_count Start, Finish;

  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();
  Start = tick_count::now();
  ResultCalculation(pMatrix, pVector, pResult, Size);
  //Finish = GetTime();
  Finish = tick_count::now();
  //Duration = Finish - Start;
  Duration = (Finish - Start).seconds();

  // 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);
}
```

7.11.2. Программный код параллельного матрично-векторного умножения

```
#include <iomanip>
#include <iostream>
#include <limits>
#include <time.h>

#include "tbb/tick_count.h"
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"

using namespace tbb;
using namespace std;

class DotProductCalc
{
  double *pMatrix,
         *pVector,
         *pResult;

  int Size;

public:
  void operator()(const blocked_range<int>& r) const
  {
    int begin = r.begin();
    int end   = r.end();

    for (int i = begin; i != end; i++)
    {
      pResult[i] = 0;
      for (int j = 0; j < Size; j++)
        pResult[i] += pMatrix[i * Size + j] * pVector[j];
    }
  }

  DotProductCalc(double *pm, double *pv, double *pr, int sz):
     pMatrix(pm), pVector(pv), pResult(pr), Size(sz)
  {}
};

// 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 parallel matrix-vector multiplication
void ParallelResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size, int grainsize)
{
  parallel_for(blocked_range<int>(0, Size, grainsize),
    DotProductCalc(pMatrix, pVector, pResult, Size));
}

// 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 Duration;
  tick_count Start, Finish;
  task_scheduler_init init;

  printf("Prallel 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();
  Start = tick_count::now();
  ParallelResultCalculation(pMatrix, pVector, pResult, Size, 1);
  //Finish = GetTime();
  Finish = tick_count::now();
```

```
  //Duration = Finish - Start;
  Duration = (Finish - Start).seconds();

  // 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);
}
```

Курс разработан и модернизирован в лаборатории «Информационные технологии» (ITLab) факультета Вычислительной математики и кибернетики Нижегородского государственного университета им. Н.И. Лобачевского с использованием материалов, подготовленных в рамках Приоритетного национального проекта «Образование», в рамках программы развития ННГУ как Национального исследовательского университета, а также при поддержке компании Интел.