Нижегородский государственный университет им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Инструменты параллельного программирования для систем с общей памятью

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

Мееров И.Б., Сысоев А.В., Сиднев А.А.
Кафедра математического обеспечения ЭВМ

Содержание

Введение…

- В работе рассматривается один из инструментов параллельного программирования, предназначенный для распараллеливания решения задач в системах с общей памятью, – библиотека **Intel Threading Building Blocks (TBB)**.
- **Основная идея TBB**: использование C++ для быстрой разработки *кросс-платформенных*, *хорошо масштабируемых параллельных приложений.*
- TBB предоставляет **механизмы абстрагирования** от парадигм многопоточного программирования, позволяя сосредоточиться непосредственно на решении *прикладной задачи*.

Введение

- Изучение принципов функционирования и вопросов эффективного использования ТВВ 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.

Рекомендации по выполнению работы…

- Целью данной лабораторной работы является приобретение практических навыков распараллеливания циклов для систем с общей памятью при помощи библиотеки TBB.
- Системные требования (для Windows-систем):
- **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.

Рекомендации по выполнению работы…

- Целью данной лабораторной работы является приобретение практических навыков распараллеливания циклов для систем с общей памятью при помощи библиотеки TBB.
- Системные требования (для Windows-систем):
- **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.

Структура работы

- Изучение способов инициализации и завершения работы библиотеки TBB.
- Рассмотрение функциональности библиотеки TBB, связанной с распараллеливанием циклов.
- Разбор одного из вариантов решения учебной задачи – умножения матрицы на вектор, демонстрирующей принципы распараллеливания циклов при помощи библиотеки TBB.
- Выполнение дополнительных заданий, состоящих в самостоятельной разработке и отладке параллельных программ, реализующих распараллеливание циклов при помощи TBB.

Задача матрично-векторного умножения

Постановка задачи

- Рассмотрим матрицу *A* размерности *m* x *n* и вектор *b*, состоящий из *n* элементов.
- Решение задачи – 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*.

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

- Общее количество необходимых скалярных операций есть величина $\boldsymbol{T_1 = m \cdot (2n - 1)}$.

Последовательный алгоритм…

- Исходные данные:
  - A[m][n] – матрица размерности *m x n*.
  - b[n] – вектор, состоящий из *n* элементов.
- Результат:
  - c[n] – вектор из *m* элементов.

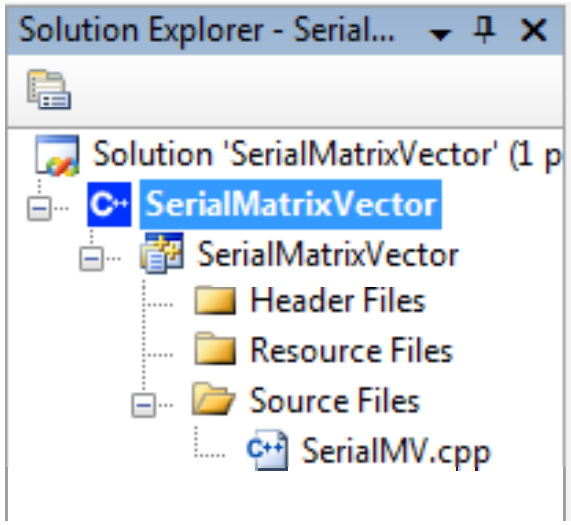
```
// Последовательный алгоритм умножения матрицы на вектор
for (i = 0; i < m; i++)
{
  c[i] = 0;
  for (j = 0; j < n; j++)
  {
    c[i] += A[i][j] * b[j];
  }
}
```

Последовательный алгоритм…

- Некоторые замечания:
  - Далее 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.

Реализация последовательного алгоритма…

- **Задание 1 – Открытие проекта SerialMatrixVector**
- Откройте проект **SerialMatrixVector**



- 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**.

Реализация последовательного алгоритма…

- Изучите представленный код и назначение основных переменных.

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

- Обратите внимание, что для хранения матрицы **pMatrix** используется одномерный массив, в котором матрица хранится построчно в соответствии с соглашениями языка С. Таким образом, элемент, расположенный на пересечении *i*-ой строки и *j*-ого столбца матрицы, в одномерном массиве имеет индекс *i*Size+j*.
- Выполните команду **Rebuild Solution** в меню **Build,** запустите приложение (**F5** или **Debug/Start debugging**).

Реализация последовательного алгоритма…

- **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.

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

Реализация последовательного алгоритма…

- 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);
}
```

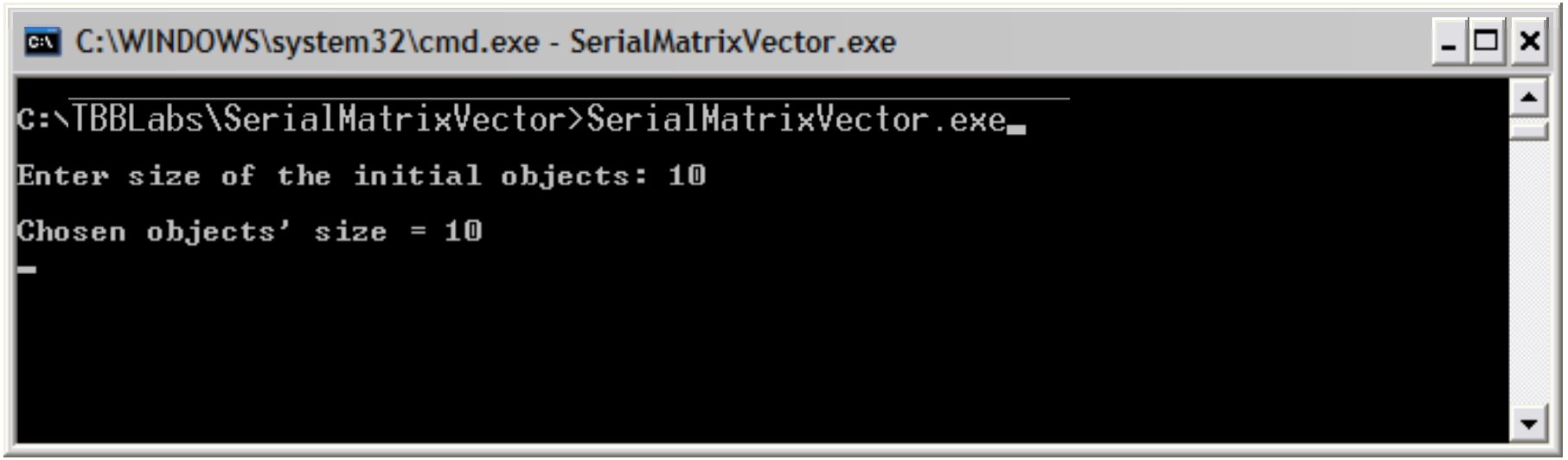
Реализация последовательного алгоритма…

- 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
  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();
}
```

Реализация последовательного алгоритма…

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



Реализация последовательного алгоритма…

- Теперь обратимся к вопросу контроля правильности ввода: 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);
```

- Скомпилируйте и запустите приложение, проверьте работоспособность.

Реализация последовательного алгоритма…

- **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];
}
```

Реализация последовательного алгоритма…

- Далее необходимо задать значения всех элементов матрицы **pMatrix** и вектора **pVector**. Для выполнения этих действий реализуем еще одну функцию **DummyDataInitialization**:

```
// Function for simple initialization
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;
  }
}
```

Реализация последовательного алгоритма…

- 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);
}
```

Реализация последовательного алгоритма…

- Реализуем еще две функции, которые помогут контролировать ввод данных, – функции форматированного вывода объектов: **PrintMatrix** и **PrintVector**:

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

Реализация последовательного алгоритма…

- 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 корректно. Выполните несколько запусков приложения, задавая различные размеры объектов.

Реализация последовательного алгоритма…

- **Задание 4 – Завершение процесса вычислений**
- Перед выполнением матрично-векторного умножения сначала разработаем функцию для корректного завершения процесса вычислений.

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

Реализация последовательного алгоритма…

- 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.

Реализация последовательного алгоритма…

- **Задание 5 – Реализация умножения матрицы на вектор**
- Для выполнения умножения матрицы на вектор реализуем функцию **SerialResultCalculation**.

```
// 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];
  }
}
```

Реализация последовательного алгоритма…

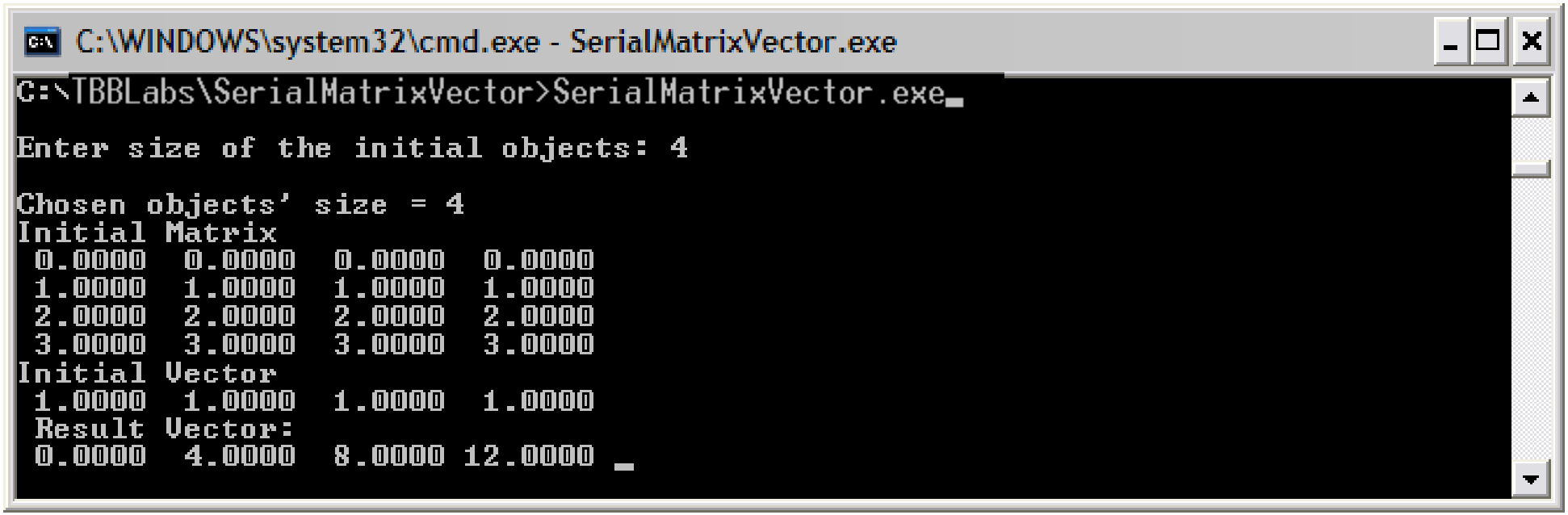
- Выполним вызов функции вычисления умножения матрицы на вектор из основной программы. Для контроля правильности выполнения умножения распечатаем результирующий вектор:

```
// 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);
```

Реализация последовательного алгоритма…

- Скомпилируйте и запустите приложение. Проанализируйте результат работы алгоритма умножения матрицы на вектор.



Реализация последовательного алгоритма…

- **Задание 6 – Проведение вычислительных экспериментов**
- После реализации параллельной версии алгоритма нам потребуется оценивать ее ускорение.
- Для этого 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).

Реализация последовательного алгоритма…

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

- 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.

Реализация последовательного алгоритма…

- 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);
}
```

Реализация последовательного алгоритма…

- Реализуем функцию **GetTime()**, которая позволяет достаточно 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);
}
```

Реализация последовательного алгоритма…

- 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);
```

Реализация последовательного алгоритма…

- Библиотека TBB реализует собственный способ замера времени, сочетающий высокую разрешаемую способность с простотой использования. Для этого предусмотрен специальный класс **tick_count**, содержащий содержит 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.
- 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 **QueryPerformanceCounter**.

Реализация последовательного алгоритма…

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

```
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);
```

Реализация последовательного алгоритма…

- 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.

Реализация последовательного алгоритма…

- **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.

Реализация последовательного алгоритма…

- Проведите эксперименты, результаты занесите в таблицу 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	

Параллельный алгоритм. Принципы распараллеливания…

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

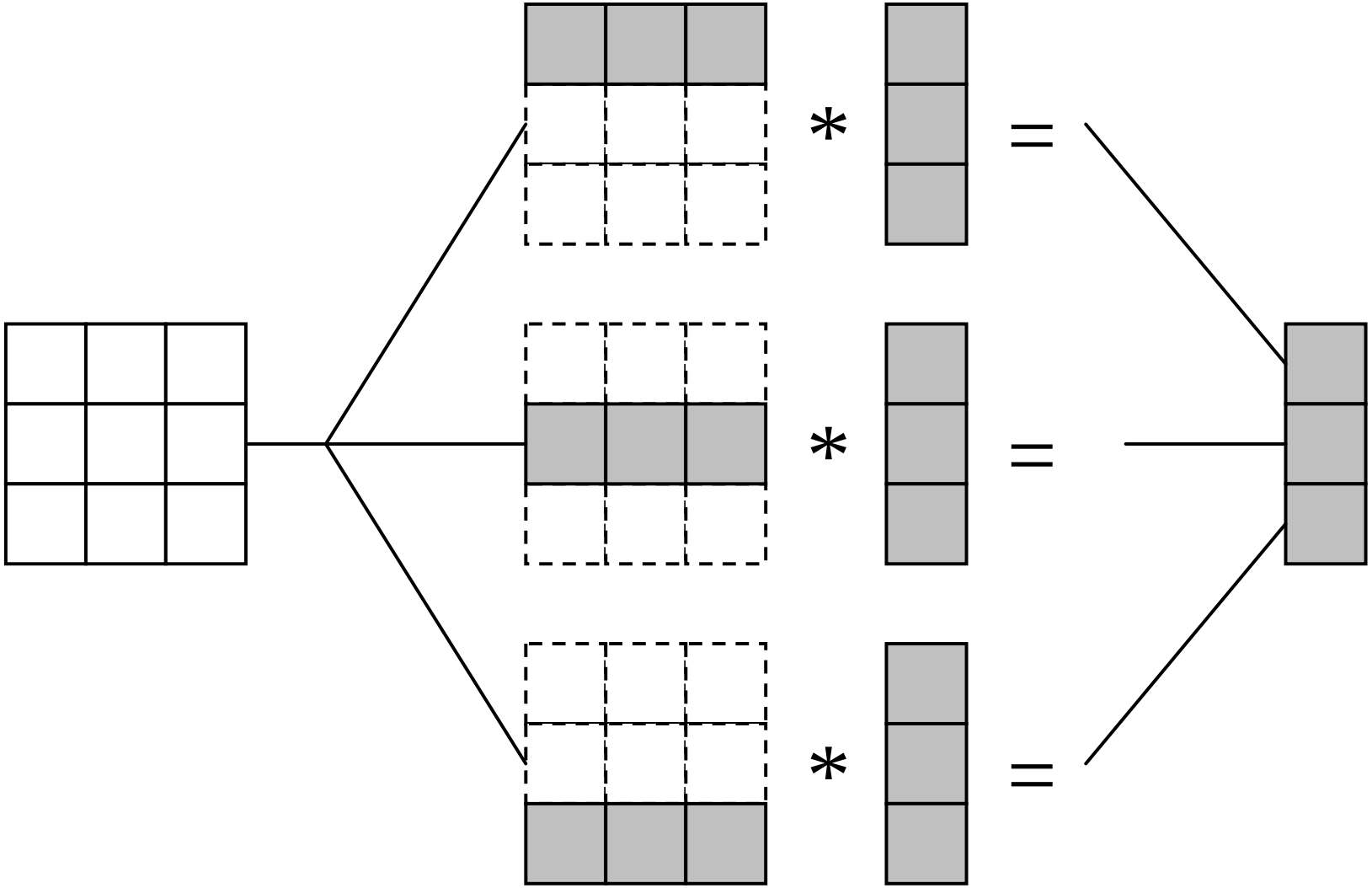
Параллельный алгоритм. Принципы распараллеливания

- Желательно обеспечить примерно одинаковый объем вычислений для каждого используемого потока – это позволит обеспечить равномерную вычислительную загрузку (*балансировку*).
- Распределение подзадач между процессорами (ядрами) желательно выполнено таким образом, чтобы число информационных связей (*коммуникационных взаимодействий*) между подзадачами было минимальным.

Параллельный алгоритм. Определение подзадач

- 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 будет рассматриваться *алгоритм умножения матрицы на вектор, основанный на представлении матрицы непрерывными наборами* (*горизонтальными полосами*) *строк*. Базовая подзадача – *операция скалярного умножения одной строки матрицы на вектор*.

Параллельный алгоритм. Выделение информационных зависимостей



Параллельный алгоритм…

- **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.
  - В случае, когда число вычислительных элементов *p* меньше числа базовых подзадач *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**.

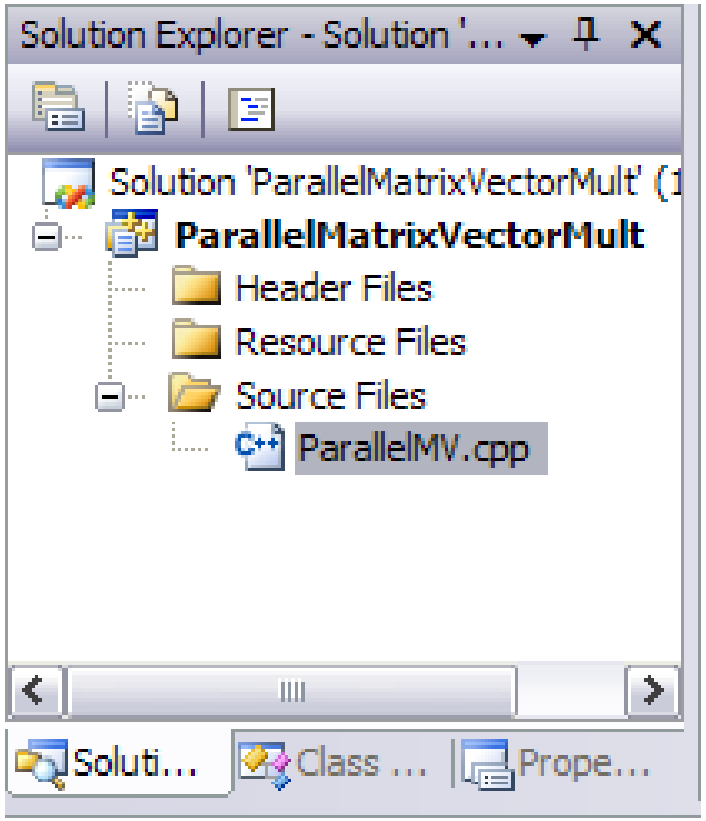
Реализация параллельного алгоритма…

- 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.

Реализация параллельного алгоритма…

- **Задание 1 – Открытие проекта ParallelMatrixVectorMult:**

Откройте проект **ParallelMatrixVectorMult**:



Реализация параллельного алгоритма…

- В файле **ParallelMV.cpp** расположена функция **main** будущего параллельного приложения.
- 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".

Реализация параллельного алгоритма…

- **Задание 2 – Настройка проекта для использования TBB**:

Установка путей к заголовочным файлам (2 способа):

- 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**.

Реализация параллельного алгоритма…

- **Задание 2 – Настройка проекта для использования TBB:**

Установка путей к библиотекам TBB:

- 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).

Реализация параллельного алгоритма…

- **Задание 3 – Инициализация библиотеки TBB**:
- Началу работы с библиотекой предшествует обязательная процедура инициализации. Для этого необходимо создать экземпляр класса **task_scheduler_init**.

Реализация параллельного алгоритма…

- **Задание 3 – Инициализация библиотеки TBB**:
- Конструктор класса **Task_scheduler_init** 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 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.

Реализация параллельного алгоритма…

- **Рекомендация**:
- 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.

Реализация параллельного алгоритма…

- **Инициализация библиотеки**:

```
#include "tbb/task_scheduler_init.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;
  <...>
  return 0;
}
```

Реализация параллельного алгоритма…

- **Задание 4 – Распараллеливание цикла средствами библиотеки TBB**:
  - 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 TBB.
  - Параллельный алгоритм состоит в умножении строк матрицы на вектор с использованием нескольких потоков.

Реализация параллельного алгоритма…

- **Задание 4 – Распараллеливание цикла средствами библиотеки TBB**:
  - 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.

Реализация параллельного алгоритма…

- **Задание 4 – Распараллеливание цикла средствами библиотеки TBB**:
  - Заметим также, 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 не составляет больших проблем.

Реализация параллельного алгоритма…

- **Задание 4 – Распараллеливание цикла средствами библиотеки TBB**:

Реализация параллельного алгоритма…

- **Задание 4 – Распараллеливание цикла средствами библиотеки TBB**:
- Схема вычислений может быть представлена в виде цикла, итерацией которого является скалярное произведение.
- Библиотека TBB предоставляет возможность легко реализовать параллельную версию таких вычислений. Для этого в библиотеки имеется шаблонная функция **tbb::parallel_for**.
- template<typename Range, typename Body>

  void parallel_for(const Range& range, const Body& body);

Реализация параллельного алгоритма…

- **Задание 4 – Распараллеливание цикла средствами библиотеки TBB**:
- 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 (класс специального вида, основная функциональность которого сосредоточена в методе **operator()**).
- *Подробное описание того, что такое итерационное пространство, может быть найдено в документе «TBB – краткое описание» настоящего курса.*

Реализация параллельного алгоритма…

- **Задание 4 – Распараллеливание цикла средствами библиотеки TBB**:
  - 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**.

Реализация параллельного алгоритма…

- **Задание 4 – Распараллеливание цикла средствами библиотеки TBB**:
  - 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**.
  - 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 в процессе самого выполнения.

Реализация параллельного алгоритма…

- **Задание 4 – Распараллеливание цикла средствами библиотеки TBB**:
  - Задание параметров выполняется при помощи следующего конструктора:

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

О выборе значения grainsize мы поговорим позже…

Реализация параллельного алгоритма…

- **Задание 4 – Распараллеливание цикла средствами библиотеки TBB**:
- 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.

Реализация параллельного алгоритма…

- **Задание 4 – Распараллеливание цикла средствами библиотеки TBB**:
  - Функция **parallel_for** выполняет разделение итерационного пространства (**range**) на не пересекающиеся подмножества размером **grainsize** и вызывает для каждого подмножества метод **body::operator()** в отдельном потоке.
  - 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).

Реализация параллельного алгоритма…

- **Задание 4 – Распараллеливание цикла средствами библиотеки TBB, реализация функтора**:

```
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)  {}
};
```

Реализация параллельного алгоритма…

- **Задание 4 – Распараллеливание цикла средствами библиотеки TBB**:
  - 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));
}
```

Реализация параллельного алгоритма…

- **Задание 4 – Распараллеливание цикла средствами библиотеки TBB**:
  - 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.

Реализация параллельного алгоритма…

- **Задание 4 – Распараллеливание цикла средствами библиотеки TBB (итог)**:

```
#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;
}
```

Реализация параллельного алгоритма…

- **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.

Реализация параллельного алгоритма…

- **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**.

Реализация параллельного алгоритма…

- **Задание 6 – Выбор значения параметра grainsize**:
- Принцип распределения итерационного пространства:
  - Если **grainsize** равно общему числу итераций, то все итерации будут выполнены на одном потоке.
  - Если **grainsize** = <общее число итераций> / < число вычислительных устройств >, то каждый поток (скорее всего) выполнит одинаковое число итераций, равное **grainsize**.
  - Если **grainsize** меньше, чем <общее число итераций> / <число число вычислительных устройств >, планировщик потоков распределит итерации между потоками по специальному алгоритму. Алгоритм описан в документе «TBB – краткое описание».

Реализация параллельного алгоритма…

- **Задание 6 – Выбор значения параметра grainsize**:
- 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.
- С одной стороны, малое значение **grainsize** способствует масштабируемости приложения.
- С другой стороны, работа планировщика потоков занимает определенное время, поэтому, чем меньше значение **grainsize**, тем больше времени потребуется планировщику на распределение заданий.

Реализация параллельного алгоритма…

- **Задание 6 – Выбор значения параметра grainsize**:
- *Выводы*:
- значение параметра **grainsize** не должно быть слишком маленьким, т.к. это может негативно отразиться на времени работы приложения (большие накладные расходы на работу планировщика);
- значение параметра **grainsize** не должно быть слишком большим, т.к. это может негативно отразиться на масштабируемости приложения.
- На практике приходится искать «золотую середину».

Реализация параллельного алгоритма…

- **Задание 6 – Выбор значения параметра grainsize**:
- *Возможный алгоритм*:
  - Установите значение **grainsize** достаточно большим, например равным размеру итерационного пространства в случае использования класса **blocked_range**.
  - Запустите приложение в один поток, замерьте время его выполнения.
  - Установите значение **grainsize** в 2 раза меньше, запустите приложение по-прежнему в один поток и оцените замедление по отношению к шагу 2. Если приложение замедлилось на 5-10%, это хороший результат. Продолжайте уменьшение **grainsize** до тех пор, пока замедление не превысит 5-10%.

Реализация параллельного алгоритма…

- **Задание 6 – Выбор значения параметра grainsize**:
  - Реализуйте описанный алгоритм.
  - Выполните сборку.
  - Проведите пробные запуски.

Реализация параллельного алгоритма…

- **Задание 7 – Оценка эффективности**:
- Выполните сборку приложения в конфигурации **Release**.
- Запустите программу несколько раз, задавая одни и те же размеры исходных данных.
- Мы рекомендуем выполнить приложение несколько раз и выбрать минимальное из полученных времен работы при одних и тех же размерах матрицы и вектора.
- Зафиксируйте результаты экспериментов, рассчитайте ускорение вычислений.

Реализация параллельного алгоритма…

- **Задание 7 – Оценка эффективности**:
- *Таблица 2*:

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			

Реализация параллельного алгоритма…

- **Задание 7 – Оценка эффективности**:
- *Таблица 3*:

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

Реализация параллельного алгоритма…

- **Задание 7 – Оценка эффективности**:
- Проведите вычислительные эксперименты для задачи с фиксированными значениями исходных данных (4000x4000) с различными значениями **grainsize** и заполните первые два столбца таблицы 4.
- Выберите желаемое значение **grainsize** и заполните последнее поле в таблице 4.

Реализация параллельного алгоритма…

- **Задание 7 – Оценка эффективности**:

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			

Краткий обзор работы

- Данная лабораторная работа посвящена изучению распараллеливания циклов на общей памяти с использованием библиотеки Intel Threading Building Blocks.
- Изучение материала выполнено на примере популярной задачи для демонстрации возможностей параллельного программирования – задаче матрично-векторного умножения.
- Рассмотрена классическая постановка задачи, приведен последовательный алгоритм решения, разобрана его последовательная реализация.
- Приведен один из возможных алгоритмов распараллеливания, а также его реализация при помощи TBB.

Контрольные вопросы…

- Какие факторы влияют на время работы приложения?
- Как собрать приложение, использующее библиотеку TBB?
- Как инициализировать библиотеку TBB?
- Как завершить работу библиотеки TBB?
- Как инициализировать библиотеку TBB с определенным числом программных потоков?

Контрольные вопросы

- Каким образом организовано распараллеливание цикла **for** в библиотеке TBB?
- Что такое функтор?
- Что такое итерационное пространство?
- Каков смысл параметра **grainsize**?
- Как работает конструкция **parallel_for** в библиотеке TBB?
- Какие методы функтора необходимо реализовать при использовании алгоритма **parallel_for**?
- Как выбрать значение параметра **grainsize**?

Задания для самостоятельной работы

- Измените созданную реализацию на случай неквадратной матрицы.
- Разработайте программу умножения квадратных матриц.
- Разработайте программу умножения прямоугольных матриц.

Использованные источники информации

- Гергель В.П., Лабутина А.А. Умножение матрицы на вектор // Материалы образовательного комплекса «Параллельное программирование на OpenMP». Нижний Новгород, 2007.
- 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.
- Intel® Threading Building Blocks. Reference Manual. Version 1.6. – Intel Corporation, 2007.
- Intel® Threading Building Blocks. Tutorial. Version 1.6. – Intel Corporation, 2007.

Рекомендуемая литература

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

Дополнительная литература

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

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

Авторский коллектив

- Мееров Иосиф Борисович,
  к.т.н., доцент кафедры Математического обеспечения ЭВМ факультета ВМК ННГУ.
- Сысоев Александр Владимирович,
  ассистент кафедры Математического обеспечения ЭВМ факультета ВМК ННГУ.
- Сиднев Алексей Александрович,
  ассистент кафедры Математического обеспечения ЭВМ факультета ВМК ННГУ.