Нижегородский государственный университет им. Н.И.Лобачевского

Межфакультетская магистратура по системному и прикладному программированию для многоядерных компьютерных систем

Учебный курс "Введение в методы параллельного программирования"

Раздел "Параллельные методы умножения матрицы на вектор"

Разработчик: А.А.Лабутина

Нижний Новгород
2007

Содержание

Удалено: 20

Удалено: 21

Удалено: 24

7. Параллельные методы умножения матрицы на вектор

7.1. Введение

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

С учетом значимости эффективного выполнения матричных расчетов многие стандартные библиотеки программ содержат процедуры для различных матричных операций. Объем программного обеспечения для обработки матриц постоянно увеличивается – разрабатываются новые экономные структуры хранения для матриц специального типа (треугольных, ленточных, разреженных и т.п.), создаются различные высокоэффективные машинно-зависимые реализации алгоритмов, проводятся теоретические исследования для поиска более быстрых методов матричных вычислений.

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

В данном разделе обсуждаются методы параллельных вычислений для операции матрично-векторного умножения, в следующем разделе (раздел 8) рассматривается операция перемножения матриц. Важный вид матричных вычислений – решение систем линейных уравнений – представлен в разделе 9. Общий для всех перечисленных задач вопрос разделения обрабатываемых матриц между параллельно работающими потоками рассматривается в подразделе 7.2.

При изложении следующего материала будем полагать, что рассматриваемые матрицы являются *плотными* (*dense*), в которых число нулевых элементов является незначительным по сравнению с общим количеством элементов матриц.

7.2. Принципы распараллеливания

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 потоками. Выбор способа разделения матриц приводит к определению конкретного метода параллельных вычислений; существование разных схем распределения данных порождает целый ряд *параллельных алгоритмов матричных вычислений*.

Наиболее общие и широко используемые способы разделения матриц состоят в разбиении данных на *полосы* (по вертикали или горизонтали) или на прямоугольные фрагменты (*блоки*).

1. Ленточное разбиение матрицы. При *ленточном* (*block-striped*) разбиении каждому потоку выделяется то или иное подмножество строк (*rowwise* или *горизонтальное разбиение*) или столбцов (*columnwise* или *вертикальное разбиение*) матрицы (рис. 7.1). Разделение строк и столбцов на полосы в большинстве случаев происходит на *непрерывной* (*последовательной*) основе. При таком подходе для горизонтального разбиения по строкам, например, матрица A представляется в виде (см. рис. 7.1)

$$A = (A_0, A_1, ..., A_{p-1})^T,\ A_i = (a_{i_0}, a_{i_1}, ..., a_{i_{k-1}}),\ i_j = ik + j,\ 0 \le j < k,\ k = m/p, \qquad (7.1)$$

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

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

$$A = (A_0, A_1, ..., A_{p-1})^T,\ A_i = (a_{i_0}, a_{i_1}, ..., a_{i_{k-1}}),\ i_j = i + jp,\ 0 \le j < k,\ k = m/p. \qquad (7.2)$$

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

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

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

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

$$A_{ij} = \begin{pmatrix} a_{i_0 j_0} & a_{i_0 j_1} & ...a_{i_0 j_{l-1}} \\ & ... & \\ a_{i_{k-1} j_0} & a_{i_{k-1} j_1} & a_{i_{k-1} j_{l-1}} \end{pmatrix}, i_v = ik + v, 0 \le v < k, k = m/s, \; j_u = jl + u, 0 \le u \le l, l = n/q. \quad (7.3)$$

При таком подходе часто оказывается полезным, чтобы топология вычислительной системы имела (по крайней мере, на логическом уровне) вид решетки из *s* строк и *q* столбцов. В этом случае при разделении данных на непрерывной основе вычисления в большинстве случаев могут быть организованы таким образом, чтобы вычислительные элементы, соседние в структуре решетки, обрабатывали смежные блоки исходной матрицы. Следует отметить, однако, что и для блочной схемы может быть применено циклическое чередование строк и столбцов.

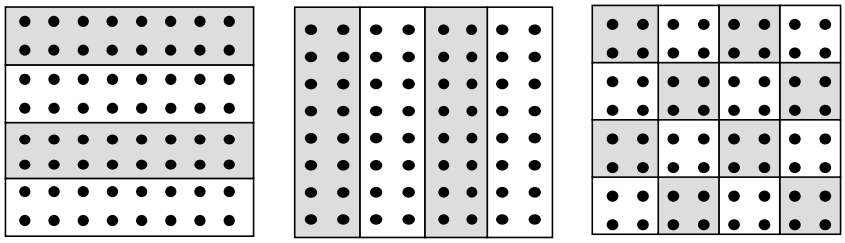


Рис. 7.1. Способы распределения элементов матрицы между потоками

В данном разделе рассматриваются три параллельных алгоритма для умножения квадратной матрицы на вектор. Каждый подход основан на разном типе распределения исходных данных (элементов матрицы и вектора) между потоками. Для каждого рассматриваемого алгоритма проводится теоретическая и экспериментальная оценка эффективности получаемых параллельных вычислений для определения наилучшего способа разделения данных.

7.3. Постановка задачи

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

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j, 1 \le i \le m. \quad (7.4)$$

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

$$T_1 = m \cdot (2n - 1)$$

7.4. Последовательный алгоритм

Последовательный алгоритм умножения матрицы на вектор может быть представлен следующим образом.

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

```
// Алгоритм 7.1
// 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]
  }
}
```

Алгоритм 7.1. Последовательный алгоритм умножения матрицы на вектор

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. $m=n$.

Представим возможный вариант последовательной программы умножения матрицы на вектор.

1. Главная функция программы. Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 7.1
// Последовательное умножение матрицы на вектор
void main(int argc, char* argv[]) {
  double* pMatrix;  // Initial matrix
  double* pVector;  // Initial vector
  double* pResult;  // Result vector for matrix-vector multiplication
  int Size;         // Sizes of initial matrix and vector

  // Data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

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

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

2. Функция ProcessInitialization. Эта функция определяет размер и элементы для матрицы A и вектора b. Значения для матрицы A и вектора b определяются в функции *RandomDataInitialization.*

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

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

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

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

Реализация функции *RandomDataInitialization* предлагается для самостоятельного выполнения. Исходные данные могут быть введены с клавиатуры, прочитаны из файла или сгенерированы при помощи датчика случайных чисел.

3. Функция SerialResultCaculation. Данная функция производит умножение матрицы на вектор.

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

Разработку функции *ProcessTermination* также предлагается выполнить самостоятельно.

Рассмотрим возможные способы организации параллельных вычислений для задачи умножении матрицы на вектор на многопроцессорных вычислительных системах с общей памятью. При изложении материала будем предполагать, что имеющиеся в составе системы процессоры обладают равной производительностью, являются равноправными при доступе к общей памяти и время доступа к памяти является одинаковым (при одновременном доступе нескольких процессоров к одному и тому же элементу памяти очередность и синхронизация доступа обеспечивается на аппаратном уровне). Многопроцессорные системы подобного типа обычно именуются *симметричными мультипроцессорами* (*symmetric multiprocessors, SMP*). Перечисленному выше набору предположений удовлетворяют также активно развиваемые в последнее время *многоядерные процессоры*, в которых каждое ядро представляет практически независимо функционирующее вычислительное устройство. Именно многоядерные процессоры (учитывая их новизну и массовый характер распространения) будут использоваться далее для проведения вычислительных экспериментов. Для общности излагаемого учебного материала для упоминания одновременно и мультипроцессоров и многоядерных процессоров для обозначения одного вычислительного устройства будет использоваться понятие *вычислительного элемента* (*ВЭ*).

7.5. Умножение матрицы на вектор при разделении данных по строкам

Рассмотрим в качестве первого примера организации параллельных матричных вычислений 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. После завершения вычислений каждая базовая подзадача определяет один из элементов вектора результата *c*.

7.5.1. Выделение информационных зависимостей

В общем виде схема информационного взаимодействия подзадач в ходе выполняемых вычислений показана на рис. 7.2.

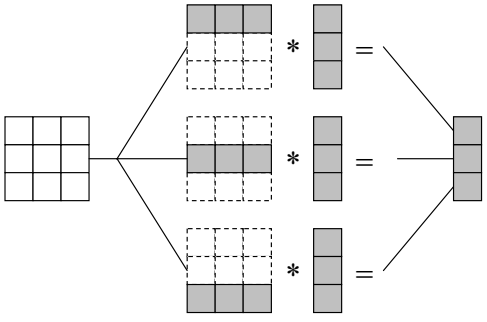


Рис. 7.2. Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на разделении матрицы по строкам

7.5.2. Масштабирование и распределение подзадач по вычислительным элементам

В процессе умножения плотной матрицы 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, следует объединить базовые подзадачи таким образом, чтобы каждый ВЭ выполнял несколько таких задач, соответствующих непрерывной последовательности строк матрицы A. В этом случае по окончании вычислений каждая базовая подзадача определяет набор элементов (*блок*) результирующего вектора c.

7.5.3. Анализ эффективности

Алгоритм умножения матрицы на вектор, основанный на ленточном горизонтальном разбиении матрицы, обладает хорошей «локализацией вычислений», то есть каждый поток параллельной программы использует только «свои» данные, и ему не требуются данные, которые в данный момент обрабатывает другой поток, нет обмена данными между потоками, не возникает необходимости синхронизации. Это означает, что в данной задаче практически нет накладных расходов на организацию параллелизма (за исключением расходов на организацию и закрытие параллельной секции), и можно ожидать линейного ускорения. Однако, как будет видно из представленных результатов, ускорение, которое демонстрирует параллельный алгоритм умножения матрицы на вектор, основанный на ленточном горизонтальном разделении данных, далеко от линейного. В чем же причина такого поведения параллельной программы?

Задача умножения матрицы на вектор обладает достаточно невысокой вычислительной сложностью – трудоемкость алгоритма имеет порядок $O(n^2)$. Такой же порядок - $O(n^2)$ – имеет и объем данных, обрабатываемый алгоритмом умножения. Время решения задачи одним потоком складывается из времени, когда процессор непосредственно выполняет вычисления, и времени, которое тратится на чтение необходимых для вычислений данных из оперативной памяти в кэш. При этом время, необходимое на чтение данных, может быть сопоставимо или в несколько раз превосходить время счета.

Проведем простой эксперимент: измерим время выполнения последовательной программы, которая суммирует все элементы матрицы, и сравним это время со временем выполнения умножения матрицы того же размера на вектор. Результаты вычислительных экспериментов приведены в таблице 7.1 и представлены на рис. 7.3.

Таблица 7.1. Сравнение времени выполнения алгоритма умножения матрицы на вектор и алгоритма суммирования всех элементов матрицы.

Размер матрицы	Время выполнения умножения матрицы на вектор	Время выполнения суммирования элементов матрицы
1000	0,0031	0,0025
2000	0,0107	0,0086
3000	0,0232	0,018
4000	0,0408	0,032
5000	0,0632	0,0501
6000	0,0908	0,0707
7000	0,1232	0,0966
8000	0,1611	0,1262
9000	0,2036	0,1596
10000	0,2508	0,1968

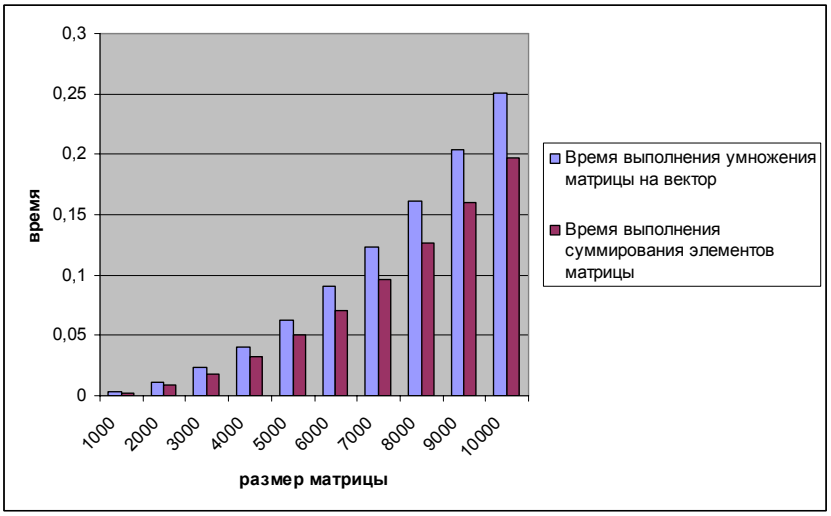


Рис. 7.3. Время выполнения умножения матрицы на вектор и суммирования всех элементов матрицы

При суммировании элементов матрицы на каждой итерации цикла выполняется простая операция сложения двух чисел. При умножении матрицы на вектор на каждой итерации цикла выполняются две операции: более сложная операция умножения и операция сложения. Но, несмотря на большую вычислительную сложность, время работы алгоритма умножения матрицы на вектор превосходит время выполнения сложения элементов матрицы всего на 20 процентов. Этот эксперимент практически подтверждает наше предположение о том, что основная часть времени тратится на выборку необходимых данных из оперативной памяти в кэш процессора.

Процесс выполнения последовательного алгоритма умножения матрицы на вектор можно представить в виде диаграммы (рис. 7.4).

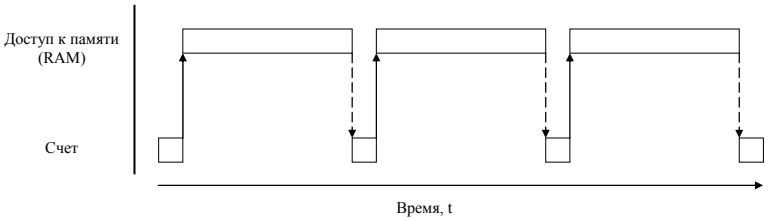


Рис. 7.4. Диаграмма состояний процесса выполнения последовательного алгоритма умножения матрицы на вектор

Следует отметить, что схема, представленная на рис. 7.4, является определенным упрощением процесса вычислений, реально выполняемых в компьютере, поскольку в современных процессорах на аппаратном уровне часто реализуются те или иные алгоритмы предсказания потребности данных для обработки и часть данных может перемещаться в кэш заранее (обеспечивая тем самым совмещение обработки и подкачки данных). Однако для предварительного анализа (тем более для быстро выполняемых операций обработки) использование такой упрощенной схемы рис. 7.4. является вполне допустимой. Дополнительная информация по архитектуре современных процессоров может быть получена, например, в [].

Примечание [LA1]: Добавить

Как уже отмечалось выше, время выполнения последовательного алгоритма складывается из времени вычислений и времени доступа к памяти:

$$T_1 = T_{calc} + T_{mem} \qquad (7.5)$$

Время вычислений – произведение количества выполненных операций N на время выполнения одной операции τ. Для алгоритма умножения матрицы на вектор количество вычислительных операций N определяется выражением:

$$N = n \cdot (2n - 1),$$

где n – размерность матрицы.

Время доступа к памяти – результат деления объема данных M (в данном случае $M=8\cdot n^2$), которые необходимо закачать в кэш процессора, на скорость доступа (пропускная способность канала доступа) к оперативной памяти β:

$$T_1 = N \cdot \tau + M / \beta \qquad (7.6)$$

Для того, чтобы оценить время одной операции τ, измерим время выполнения последовательного алгоритма умножения матрицы на вектор при малых объемах данных, таких, чтобы матрица и вектор полностью поместились в кэш вычислительного элемента. Чтобы исключить необходимость выборки данных из оперативной памяти, перед началом вычислений заполним матрицу и вектор случайными числами – выполнение этого действия гарантирует закачку данных в кэш. Далее при решении задачи все время будет тратиться непосредственно на вычисления, так как нет необходимости загружать данные из оперативной памяти. Поделив полученное время на количество выполненных операций, получим время выполнения одной операции. Для вычислительной системы, которая использовалась для проведения экспериментов, было получено значение τ, равное 0,586 нс.

Теперь, зная время выполнения одной вычислительной операции, необходимо определить скорость доступа к оперативной памяти. Для этого вычтем из времени выполнения задачи умножения матрицы на вектор время, необходимое на выполнение вычислений. Оставшееся время – время, затраченное на выборку данных из оперативной памяти в кэш процессора. Поделив это время на объем загруженных данных, получим эффективную скорость доступа к оперативной памяти (понятно, что данные расчеты должны выполняться для экспериментов, при которых размер матрицы значительно превышает размер кэша). По результатам выполненных экспериментов для используемой вычислительной системы получим значение β, равное 5,5 Гб/с. В таблице 7.2 и на рис. 7.5 представлено сравнение реального времени выполнения последовательного алгоритма умножения матрицы на вектор и времени, вычисленного по формуле 7.6.

Таблица 7.2. Сравнение экспериментального и теоретического времени выполнения последовательного алгоритма умножения матрицы на вектор

Размер матриц	Эксперимент	Модель
1000	0,0031	0,0026
2000	0,0107	0,0105
3000	0,0232	0,0236
4000	0,0408	0,0420
5000	0,0632	0,0657
6000	0,0908	0,0946
7000	0,1232	0,1287
8000	0,1611	0,1681
9000	0,2036	0,2127
10000	0,2508	0,2626

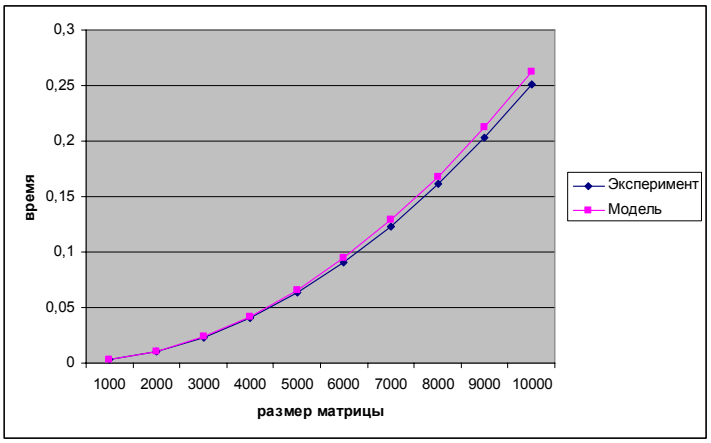


Рис. 7.5. График зависимости экспериментального и теоретического времени выполнения последовательного алгоритма от объема исходных данных (ленточное разбиение матрицы по строкам)

В многоядерных процессорах Intel архитектуры Core 2 Duo ядра процессоров разделяют общий канал доступа к памяти. То есть, несмотря на то, что вычисления могут выполняться ядрами параллельно, доступ к памяти осуществляется строго последовательно. Представим, как будет выглядеть диаграмма состояний потоков, выполняющих параллельный алгоритм умножения матрицы на вектор, при условии, что эти потоки запущены на ядрах одного двухъядерного процессора:

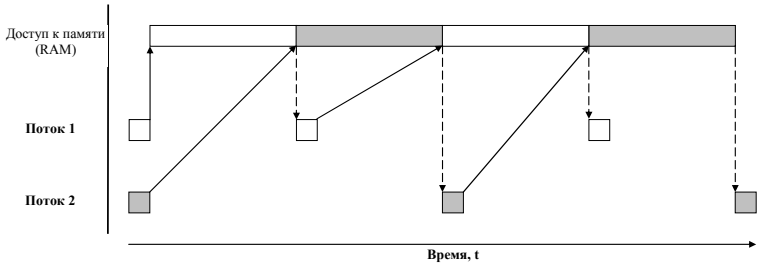


Рис. 7.6. Диаграмма состояний потоков при выполнении параллельного алгоритма умножения матрицы на вектор

Следовательно, время T_p выполнения параллельного алгоритма на компьютерной системе, имеющей p вычислительных элементов, с использованием p потоков и при описанной выше схеме доступа к памяти может быть оценено при помощи следующего соотношения:

$$\frac{8 \cdot n^2}{\beta} \le T_p \le \frac{n \cdot (2n-1)}{p} \cdot \tau + \frac{8 \cdot n^2}{\beta}. \qquad (7.7)$$

7.5.4. Программная реализация

Для того, чтобы разработать параллельную программу, реализующую описанный подход, при помощи технологии OpenMP, необходимо внести минимальные изменения в функцию умножения матрицы на вектор. Достаточно добавить одну директиву *parallel for* в функции *SerialResultCaclualtion* (назовем новый вариант функции *ParallelResultCaclualtion*):

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

Данная функция производит умножение строк матрицы на вектор с использованием нескольких параллельных потоков. Каждый поток выполняет вычисления над несколькими соседними строками матрицы и, таким образом, получает блок результирующего вектора *c*.

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

В данной функции между потоками параллельной программы разделяются итерации внешнего цикла алгоритма умножения матрицы на вектор. В результате, каждый поток выполняет непрерывную последовательность итераций цикла по переменной *i*, и, таким образом, умножает горизонтальную полосу матрицы *A* на вектор *b* и вычисляет блок элементов результирующего вектора *c*.

7.5.5. Результаты вычислительных экспериментов

Рассмотрим результаты вычислительных экспериментов, выполненных для оценки эффективности параллельного алгоритма умножения матрицы на вектор. Эксперименты проводились на двухъядерном вычислительном узле на базе процессора Intel Core 2 6300, 1.87 ГГц, 2 Гб RAM под управлением

операционной системы Microsoft Windows XP Professional. Разработка программ проводилась в среде Microsoft Visual Studio 2003, для компиляции использовался Intel C++ Compiler 9.0 for Windows.

Результаты вычислительных экспериментов приведены в таблице 7.3. Времена выполнения алгоритмов указаны в секундах.

Таблица 7.3. Результаты вычислительных экспериментов для параллельного алгоритма умножения матрицы на вектор при ленточной схеме разделении данных по строкам

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
1000	0,0031	0,0024	1,2475
2000	0,0107	0,0075	1,4197
3000	0,0232	0,0165	1,4044
4000	0,0408	0,0288	1,4149
5000	0,0632	0,0445	1,4221
6000	0,0908	0,0635	1,4300
7000	0,1232	0,0867	1,4209
8000	0,1611	0,1127	1,4297
9000	0,2036	0,1416	1,4374
10000	0,2508	0,1753	1,4311

Рис. 7.7. Зависимость ускорения от количества исходных данных при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на ленточном горизонтальном разбиении матрицы

В таблице 7.4 и на рис. 7.8 представлены результаты сравнения времени выполнения параллельного алгоритма умножения матрицы на вектор с использованием двух потоков со временем, полученным при помощи модели (7.7).

Таблица 7.4. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор, основанного на ленточном горизонтальном разбиении матрицы, с использованием двух потоков

Размер матриц	Эксперимент	Время счета (модель)	Время доступа к памяти (модель)	Общее время (модель)
1000	0,0031	5,86E-04	0,0015	0,0020
2000	0,0107	2,34E-03	0,0058	0,0082
3000	0,0232	5,27E-03	0,0131	0,0184
4000	0,0408	9,37E-03	0,0233	0,0327
5000	0,0632	1,46E-02	0,0364	0,0510
6000	0,0908	2,11E-02	0,0524	0,0735
7000	0,1232	2,87E-02	0,0713	0,0999
8000	0,1611	3,75E-02	0,0931	0,1306

9000	0,2036	4,75E-02	0,1178	0,1653
10000	0,2508	5,86E-02	0,1455	0,2041

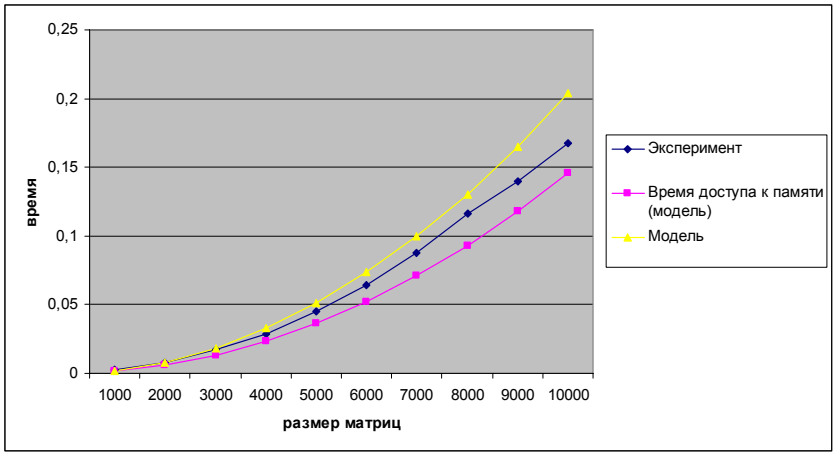


Рис. 7.8. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма от объема исходных данных при использовании двух потоков (ленточное разбиение матрицы по строкам)

7.6. Умножение матрицы на вектор при разделении данных по столбцам

Рассмотрим теперь другой подход к параллельному умножению матрицы на вектор, основанный на разделении исходной матрицы на непрерывные наборы (вертикальные полосы) столбцов.

7.6.1. Определение подзадач и выделение информационных зависимостей

При таком способе разделения данных в качестве базовой подзадачи может быть выбрана операция умножения столбца матрицы A на один из элементов вектора b. Для организации вычислений в этом случае каждая базовая подзадача i, $0 \le i < n$, должна иметь доступ к i-ому столбцу матрицы A.

Каждая базовая задача i выполняет умножение своего столбца матрицы A на элемент b_i, в итоге в каждой подзадаче получается вектор $c'(i)$ промежуточных результатов. Для получения элементов результирующего вектора c необходимо просуммировать вектора $c'(i)$, которые были получены каждой подзадачей.

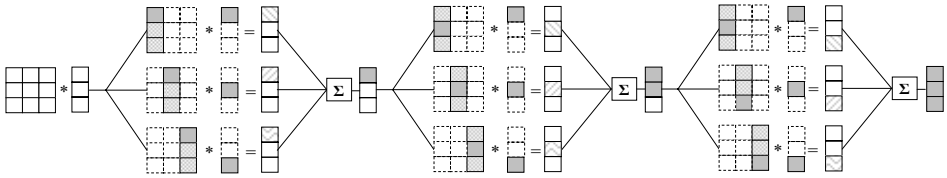


Рис. 7.9. Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор с использованием разбиения матрицы по столбцам

Для того, чтобы реализовать такой подход, необходимо распараллелить внутренний цикл алгоритма умножения матрицы на вектор. В результате данного подхода параллельные области будут создаваться для вычисления каждого отдельного элемента результирующего вектора. Программа будет представляться в виде набора последовательных (*однопотоковых*) и параллельных (*многопотоковых*) участков. Подобный принцип организации параллелизма получил название «*вилочного*» (*fork-join*) или *пульсирующего параллелизма*. При выполнении многопотокового участка каждый поток выполняет умножение одного элемента своего столбца исходной матрицы на один элемент вектора. Следующий за этим однопотоковый участок производит суммирование полученных результатов для того, чтобы вычислить элемент результирующего вектора (см. рис. 7.9). Таким образом, программа будет состоять из чередования n многопотоковых и n однопотоковых участков.

7.6.2. Масштабирование и распределение подзадач по вычислительным элементам

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и равным объемом передаваемых данных. В случае, когда количество столбцов матрицы превышает число процессоров, базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько соседних столбцов (в этом случае исходная матрица A разбивается на ряд вертикальных полос). При соблюдении равенства размера полос такой способ агрегации вычислений обеспечивает равномерность распределения вычислительной нагрузки по процессорам, составляющим многопроцессорную вычислительную систему.

7.6.3. Анализ эффективности

При анализе эффективности данного параллельного алгоритма будем полагаться на результаты, полученные в подразделе 7.5.4. Очевидно, что ни время выполнения одной вычислительной операции, ни эффективная скорость доступа к оперативной памяти не зависят от того, каким образом распределяются вычисления между потоками параллельной программы.

Прежде всего, следует заметить, что при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на вертикальном разделении матрицы, выполняется больше арифметических операций. Действительно, на каждой итерации внешнего цикла после вычисления каждым потоком своей частичной суммы необходимо выполнить редукцию полученных результатов. Сложность выполнения операции редукции – $log_2 p$. После выполнения редукции данных, необходимо запомнить результат в очередной элемент результирующего вектора. Таким образом, время вычислений для данного алгоритма определяется формулой:

$$T_{calc} = \left(\frac{n \cdot (2n-1)}{p} + n \cdot \log_2 p + n \right) \cdot \tau .$$

Поскольку объем данных, необходимых для чтения из оперативной памяти в кэш не изменяется при переходе от алгоритма, основанного на горизонтальном разделении данных, к алгоритму, основанному на вертикальном разделении данных, то и время чтения данных в кэш остается таким же, что и при выполнении первого параллельного алгоритма.

На каждой итерации внешнего цикла алгоритма умножения матрицы на вектор организуется параллельная секция для вычисления каждого элемента результирующего вектора. Для организации и синхронизированного закрытия параллельных секций, а также для выполнения редукции вызываются специализированные функции библиотеки OpenMP. С одной стороны, для выполнения каждой такой функции необходимо определенное время. С другой стороны, в процессе выполнения эти функции используют данные, которые должны быть загружены в кэш процессора. Так как объем кэша ограничен, это ведет к вытеснению и последующей повторной загрузке элементов матрицы и вектора. На выполнение таких подкачек данных тоже тратится некоторое время. Обозначим накладные расходы на организацию параллельности на каждой итерации через δ. Тогда суммарные накладные расходы составляют:

$$T_{overhead} = n \cdot \delta$$

Таким образом, время выполнения параллельного алгоритма умножения матрицы на вектор, основанного на вертикальном разделении матрицы, может быть вычислено по формуле:

$$T_p = \left(\frac{n \cdot (2n-1)}{p} + n \cdot \log_2 p + n \right) \cdot \tau + \frac{8 \cdot n^2}{\beta} + n \cdot \delta \qquad (7.8)$$

7.6.4. Программная реализация

При реализации данного подхода к параллельному умножению матрицы на вектор каждый очередной многопотоковый участок параллельного алгоритма служит для вычисления одного элемента результирующего вектора. Очевидный способ реализовать подобный подход может состоять в следующем:

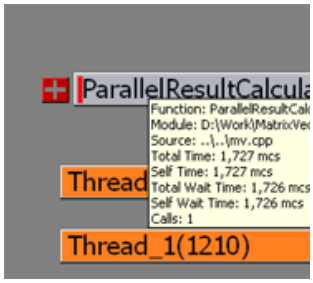
```
// Function for calculating matrix-vector multiplication
void ParallelResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int i, j;  // Loop variables
  for (i=0; i<Size; i++) {
#pragma omp paralell for
    for (j=0; j<Size; j++) // !!! Внимание – см. приводимые далее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

Удалено: Как видно из представленного программного кода, н

Отформатировано: ниже на 6 пт

Примечание [ГВП2]: Необходимо дать ссылку

Удалено: Для анализа программного кода и подтверждения выдвинутых предположений воспользуемся специализированной программной системой, предназначенной для профилирования и оптимизации программ, Intel VTune Performance Analyzer (см., например, []). Данная программная система позволяет находить в программе критический (вычислительно-трудоемкий) код, измерять различные характеристики эффективности кода и наблюдать за процессом выполнения программы в динамике. ¶
Проанализируем при помощи инструмента Call Graph (*Граф вызова функций*) профилировщика Intel VTune Performance Analyzer приложение, выполняющее параллельный алгоритм умножения матрицы на вектор, основанный на разделении матрицы на горизонтальные полосы, и сконцентрируем свое внимание на анализе выполнения функции *ParallelResultCalculation*. На рис. 7.10 представлен результат анализа. Видно, что собственное (без учета времени выполнения вызываемых функций) время и полное время выполнения функции совпадают. Это означает, что, несмотря на то, что функция *ParallelResultCalculation* вызывает другие функции (в данном случае – функции библиотеки времени исполнения OpenMP), практически все время тратится непосредственно на вычисления.¶



¶
<#>Общее и собственное время выполнения функции *ParallelResultCalculation* в случае разделения матрицы на горизонтальные полосы¶
Теперь перейдем к анализу параллельного алгоритма, основанного на разделении матрицы по столбцам. В данном случае собственное время выполнения функции *ParallelResultCalculation* существенно меньше полного времени ее выполнения. Большая доля времени тратится на выполнение вложенных функций, которыми являются функции, необходимые для организации и закрытия параллельных секций, а также для выполнения редукции (рис. 7. ... [1]

```
}
```

Выполним проверку правильности выполнения полученной программы – разработаем для этого функцию *TestResult*. Для контроля результатов выполним умножение исходной матрицы на вектор при помощи последовательного алгоритма, а затем сравним поэлементно вектора, полученные в результате выполнения последовательного и параллельного алгоритмов (следует отметить, что сравнение равенства вещественных чисел следует выполнять с некоторой явно указываемой точностью):

```
// Function for testing parallel matrix-vector multiplication
void TestResult(double* pMatrix, double* pVector, double* pResult, int Size) {
  double* pSerialResult = new double [Size];
  double Accuracy =1.0e-6;
  int equal = 0;

  // Serial matrix-vector multiplication
  SerialResultCalculation(pMatrix, pVector, pSerialResult, Size);

  // Testing multiplication results
  for (int i=0; i<Size; i++)
    if (fabs(pResult[i]-pSerialResult[i])>Accuracy) equal++;
  if (equal)
    printf("The result is NOT correct \n");
  else
    printf ("The result is correct \n");

  delete [] pSerialResult;
}
```

Результатом работы этой функции является печать диагностического сообщения. Данная функция позволяет контролировать правильность выполнения параллельного алгоритма в автоматическом режиме, независимо от сложности и объема исходных данных.

Результаты экспериментов показывают, что разработанная функция параллельного умножения матрицы на вектор, реализующая разделение данных по столбцам, не всегда дает верный результат. Причина неправильной работы кроется в неправильном использовании переменных, являющихся общими для нескольких потоков. В разработанном варианте программы такими общими переменными являются элементы вектора *pResult*. В ходе вычислений несколько разных потоков могут, например, попытаться одновременно записать новые значения в одну и ту же общую переменную, что приведет к получению ошибочных результатов. Для исключения взаимовлияния потоки должны использовать общие данные с соблюдением правил взаимоисключения, которые бы обеспечивали использование общих переменных в каждый момент времени только одним потоком (а потоки, пытающие получить доступ к «занятым» общим данным должны блокироваться).

Реализация взаимоисключения доступа может состоять в использовании предусмотренных в OpenMP специальных переменных синхронизации - *замков*:

```
// Function for calculating matrix-vector multiplication
void ParallelResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int i, j;  // Loop variables
  omp_lock_t MyLock;
  omp_init_lock(&MyLock);

  for (i=0; i<Size; i++) {
#pragma omp paralell for
    for (j=0; j<Size; j++) {
      omp_set_lock(&MyLock);
      pResult[i] += pMatrix[i*Size+j]*pVector[j];
      omp_unset_lock(&MyLock);
    }
  }
  omp destroy lock(&MyLock);
}
```

Функция *omp_set_lock* выполняется для потока только в том случае, если другие потоки не выполнили вызов этой функции для той же самой переменной; в противном случае поток блокируется (переменная-замок пропускает только один поток). Продолжение блокированных потоков осуществляется только после

выполнения для замка функции *omp_unset_lock* (при наличии нескольких блокированных потоков после снятия замка выполнение разрешается только для одного потока и замок снова «закрывается»).

После внесения этих изменений в код функции, выполняющей умножение матрицы на вектор, результирующий вектор совпадает с вектором, полученным при помощи последовательного алгоритма. Но реализация алгоритма регулирования доступа к элементу вектора *pResult[i]* привела к тому, что потоки многопотокового участка могут выполняться только последовательно, так как один поток не может получить право на изменение разделяемой переменной до тех пор, пока такое изменение выполняет другой поток.

Для одновременного решения проблемы разделения и защиты общих переменных в OpenMP реализован стандартный механизм эффективного выполнения операции редукции при распараллеливании циклов. Данный механизм состоит в использовании директивы *reduction*. Если при распараллеливании циклов указывается директива *reduction*, компилятор автоматически создает локальные копии переменной редуцирования для каждого потока параллельной программы, а после выполнения цикла собирает значения локальных копий в разделяемую переменную с использованием операции, указанной как аргумент директивы редукции.

Представим новый вариант параллельной программы умножения матрицы на вектор с использованием алгоритма разбиения матрицы по столбцам. Ниже приведем только код основной функции, выполняющей параллельный алгоритм умножения матрицы на вектор.

```
// Function for calculating matrix-vector multiplication
void ParallelResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int i, j;  // Loop variables
  double IterGlobalSum = 0;

  for (i=0; i<Size; i++) {
    IterGlobalSum = 0;
#pragma omp parallel for reduction(+:IterGlobalSum)
    for (j=0; j<Size; j++)
      IterGlobalSum += pMatrix[i*Size+j]*pVector[j];
    pResult[i] = IterGlobalSum;
  }
}
```

7.6.5. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма умножения матрицы на вектор при разбиении данных по столбцам проводились при условиях, указанных в п. 7.5.5.

Для анализа программного кода и подтверждения выдвинутых предположений воспользуемся специализированной программной системой, предназначенной для профилирования и оптимизации программ, Intel VTune Performance Analyzer (см., например, []). Данная программная система позволяет находить в программе критический (вычислительно-трудоемкий) код, измерять различные характеристики эффективности кода и наблюдать за процессом выполнения программы в динамике.

Примечание [ГВПЗ]: Необходимо дать ссылку

Проанализируем при помощи инструмента Call Graph (*Граф вызова функций*) профилировщика Intel VTune Performance Analyzer приложение, выполняющее параллельный алгоритм умножения матрицы на вектор, основанный на разделении матрицы на горизонтальные полосы, и сконцентрируем свое внимание на анализе выполнения функции *ParallelResultCalculation*. На рис. 7.10 представлен результат анализа. Видно, что собственное (без учета времени выполнения вызываемых функций) время и полное время выполнения функции совпадают. Это означает, что, несмотря на то, что функция *ParallelResultCalculation* вызывает другие функции (в данном случае – функции библиотеки времени исполнения OpenMP), практически все время тратится непосредственно на вычисления.

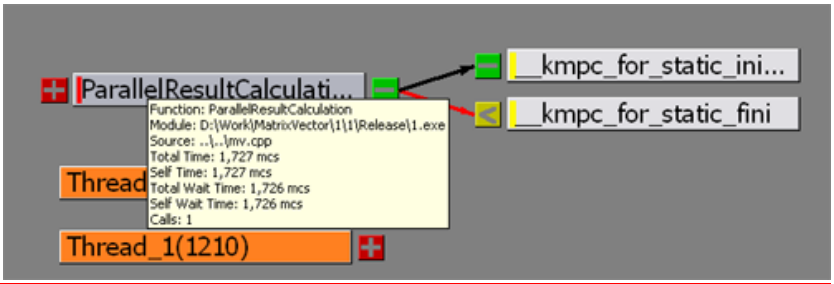


Рис. 7.10. Общее и собственное время выполнения функции *ParallelResultCalculation* в случае разделения матрицы на горизонтальные полосы

Формат: Список

Теперь перейдем к анализу параллельного алгоритма, основанного на разделении матрицы по столбцам. В данном случае собственное время выполнения функции *ParallelResultCalculation* существенно меньше полного времени ее выполнения. Большая доля времени тратится на выполнение вложенных функций, которыми являются функции, необходимые для организации и закрытия параллельных секций, а также для выполнения редукции (рис. 7.11).

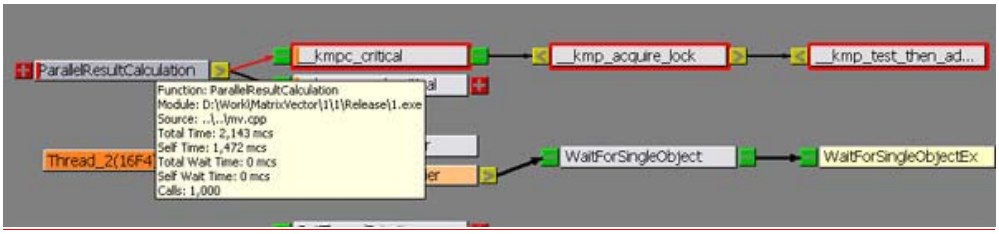


Рис. 7.11. Общее и собственное время выполнения функции *ParallelResultCalculation* в случае разделения матрицы на вертикальные полосы

Формат: Список

Далее воспользуемся инструментом Sampling[2)] профилировщика Intel VTune Performance Analyzer, обеспечивающим автоматизированный процесс регистрации различных событий, возникающих в процессоре во время исполнения профилируемой программы. Создадим проект, в котором вызовем последовательно функцию, выполняющую параллельный алгоритм умножения матрицы на вектор, основанный на горизонтальном разделении матрицы (*ParallelResultCalculation1*), и функцию, выполняющую параллельный алгоритм, основанный на вертикальном разделении матрицы (*ParallelResultCalculation2*). Измерим число возникновений события выделения новой кэш-линии первого уровня (в Intel VTune Performance Analyzer данное событие называется L1 Cache Line Allocation). Результаты профилирования приложения представлены на рис. 7.12. Эксперименты проводились для матрицы размером 1000×1000 элементов.

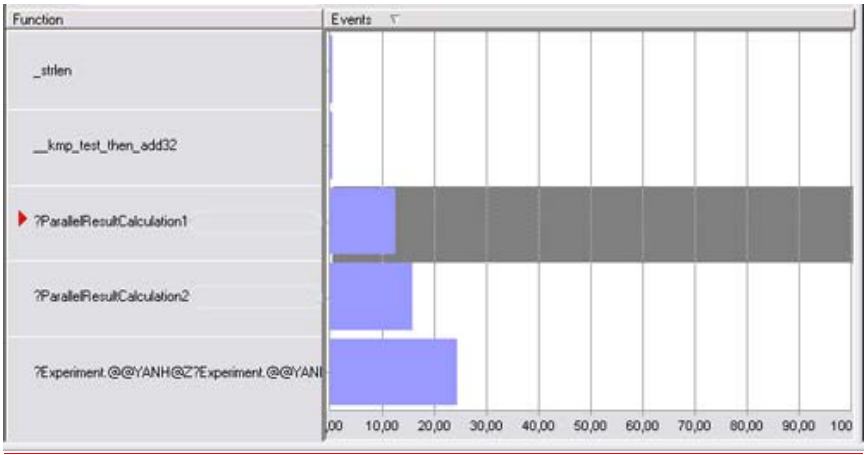


Рис. 7.12. Соотношение количества событий выделения кэш-линий первого уровня для двух параллельных алгоритмов умножения матрицы на вектор.

Формат: Список

При выполнении параллельного алгоритма умножения матрицы на вектор, основанного на вертикальном разделении, выделяется больше (в среднем на 25%) линий кэша первого уровня.

Далее, оценим накладные расходы на организацию параллельности на каждой итерации алгоритма, основанного на вертикальном разделении данных. Для этого подготовим еще один вариант параллельного алгоритма умножения матрицы на вектор при разделении матрицы по столбцам. Образуем в этом варианте программы потоки, которые самостоятельно определяют (используя идентификатор потока) блоки элементов матрицы и вектора для обработки. Далее потоки параллельно выполняют умножение своих

[2)] В литературе практически не встречается общепринятого именования инструмента Sampling на русском языке и для его обозначения часто используют транслитерацию вида *Сэмплирование*.

столбцов матрицы на элементы вектора, результат умножения сохраняется в общем массиве *AllResults*. Количество элементов в этом массиве равно *Size*×*ThreadNum*, где *ThreadNum* – общее число потоков. Потоки осуществляют запись частичных результатов в непересекающиеся сегменты массива *AllResults*: нулевой поток осуществляет запись в элементы с *0* по *Size-1*, первый поток – в элементы с *Size* по *2·Size-1*, и т.д. После выполнения умножения элементы вектора *AllResults* необходимо просуммировать для получения элементов результирующего вектора.

```
void ParallelResultCalculation(double* pMatrix, double* pVector,
  double* pResult, double* AllResults, int Size) {
  int ThreadNum;
#pragma omp parallel shared (ThreadNum)
  {
    ThreadNum = omp_get_num_threads();
    int ThreadID = omp_get_thread_num();
    int BlockSize = Size/ThreadNum;
    double IterResult;
    for (int i=0; i<Size; i++) {
      IterResult = 0;
      for (int j=0; j<BlockSize; j++)
        IterResult += pMatrix[i*Size+j+ThreadID*BlockSize] *
          pVector[j+ThreadID*BlockSize];
      AllResults[Size*ThreadID+i] = IterResult;
    }
  }
  for (int i=0; i<Size; i++)
    for (int j=0; j<ThreadNum; j++)
      pResult[i] += AllResults[j*Size+i];
}
```

Как видно из представленного программного кода, при выполнении данного алгоритма параллельная секция создается только один раз, и, следовательно, время, необходимое для выполнения функций библиотеки OpenMP, отвечающих за организацию и закрытие параллельных секций, пренебрежимо мало. Для того, чтобы оценить накладные расходы δ на организацию параллельности на каждой итерации алгоритма, необходимо из времени выполнения исходного алгоритма умножения матрицы на вектор, основанного на вертикальном разделении данных, вычесть время выполнения вновь разработанного параллельного алгоритма и поделить полученную разницу на количество параллельных секций. Эксперименты показывают, что величина δ равна 10 микросекунд ($1 \cdot 10^{-5}$ с).

Результаты вычислительных экспериментов приведены в таблице 7.5.

Таблица 7.5. Результаты вычислительных экспериментов по исследованию параллельного алгоритма умножения матрицы на вектор, основанного на разбиении матрицы по столбцам

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
1000	0,0031	0,0136	0,2249
2000	0,0107	0,0298	0,3582
3000	0,0232	0,0484	0,4784
4000	0,0408	0,0704	0,5795
5000	0,0632	0,0959	0,6598
6000	0,0908	0,1255	0,7237
7000	0,1232	0,1574	0,7825
8000	0,1611	0,1939	0,8305
9000	0,2036	0,2326	0,8753
10000	0,2508	0,2764	0,9074

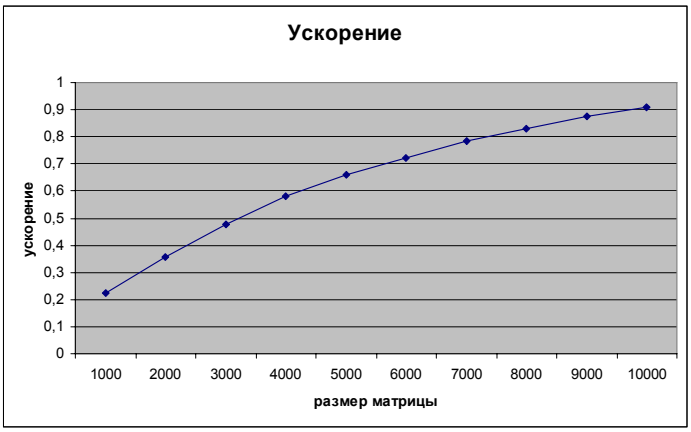


Рис. 7.13. Зависимость ускорения размера матрицы при выполнении параллельного алгоритма умножения матрицы на вектор (ленточное разбиение матрицы по столбцам)

В таблице 7.6 и на рис. 7.14 представлены результаты сравнения времени выполнения параллельного алгоритма умножения матрицы на вектор с использованием двух потоков со временем, полученным при помощи модели (7.8).

Таблица 7.6. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор, основанного на ленточном вертикальном разбиении матрицы, с использованием двух потоков

Размер матриц	Эксперимент	Модель
1000	0,0136	0,0120
2000	0,0298	0,0282
3000	0,0484	0,0484
4000	0,0704	0,0727
5000	0,0959	0,1010
6000	0,1255	0,1335
7000	0,1574	0,1700
8000	0,1939	0,2106
9000	0,2326	0,2553
10000	0,2764	0,3041

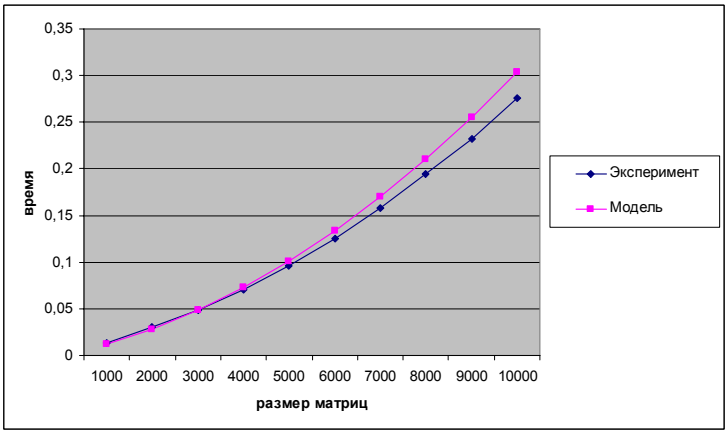


Рис. 7.14. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма от объема исходных данных при использовании двух потоков (ленточное разбиение матрицы по столбцам)

7.7. Умножение матрицы на вектор при блочном разделении данных

Рассмотрим теперь параллельный алгоритм умножения матрицы на вектор, который основан на ином способе разделения данных – на разбиении матрицы на прямоугольные фрагменты (*блоки*).

7.7.1. Определение подзадач

Блочная схема разбиения матриц подробно рассмотрена в подразделе 7.2. При таком способе разделения данных исходная матрица A представляется в виде набора прямоугольных блоков:

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

где A_{ij}, $0 \le i < s$, $0 \le j < q$, есть блок матрицы:

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

(здесь, как и ранее, предполагается, что $p = s \cdot q$, количество строк матрицы является кратным s, а количество столбцов – кратным q, то есть $m = k \cdot s$ и $n = l \cdot q$).

При использовании блочного представления матрицы A базовые подзадачи целесообразно определить на основе вычислений, выполняемых над матричными блоками. Для нумерации подзадач могут использоваться индексы располагаемых в подзадачах блоков матрицы A, т.е. подзадача *(i,j)* производит вычисления над блоком A_{ij}. Помимо блока матрицы A каждая подзадача должна иметь доступ к блоку вектора b. При этом для блоков одной и той же подзадачи должны соблюдаться определенные правила соответствия – операция умножения блока матрицы A_{ij} может быть выполнена только, если блок вектора $b'(i,j)$ имеет вид

$b'(i,j) = (b'_0(i,j), \ldots, b'_{l-1}(i,j))$, где $b'_u(i,j) = b_{j_u}$, $j_u = jl + u, 0 \le u < l, l = n/q$.

7.7.2. Выделение информационных зависимостей

Рассмотрим общую схему параллельных вычислений для операции умножения матрицы на вектор при блочном разделении исходных данных. После перемножения блоков матрицы A и вектора b каждая подзадача *(i,j)* будет содержать вектор частичных результатов $c'(i,j)$, определяемый в соответствии с выражениями:

$$c'_v(i,j) = \sum_{u=0}^{l-1} a_{i_v j_u} b_{j_u}, i_v = ik + v, 0 \le v < k, k = m/s, \; j_u = jl + u, 0 \le u \le l, l = n/q.$$

Как можно видеть из приведенных выражений, каждый поток вычисляет блок вектора частичных результатов исходной задачи умножения матрицы на вектор. Полный результат – вектор c – можно определить суммированием элементов блока вектора частичных результатов с одинаковыми индексами по всем подзадачам, относящимся к одним и тем же строкам решетки потоков, т.е.

$$c_v(i) = \sum_{j=0}^{q-1} c'_v(i,j), 0 \le i < s, \; 0 \le v < k, k = m/s.$$

Общая схема выполняемых вычислений для умножения матрицы на вектор при блочном разделении данных показана на рис. 7.15.

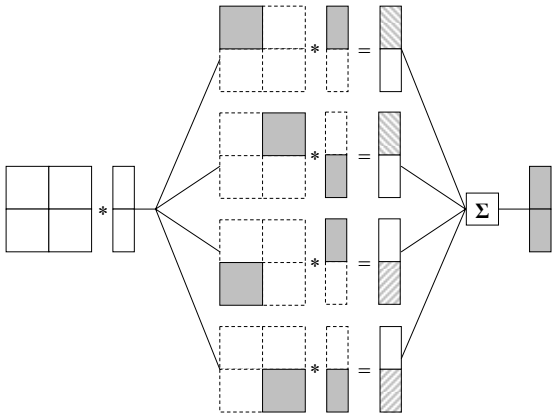


Рис. 7.15. Общая схема параллельного алгоритма умножения матрицы на вектор при блочном разделении данных

При дальнейшем изложении материала для снижения сложности и упрощения получаемых соотношений будем полагать, что число блоков в матрице A совпадает по горизонтали и по вертикали, т.е. $s=q$. Для эффективного выполнения параллельного алгоритма умножения матрицы на вектор, основанного на блочном разделении данных, целесообразно выделить число параллельных потоков совпадающим с количеством блоков матрицы A, т.е. такое количество потоков, которое является полным квадратом q^2. Данное требование может привести к тому, что число вычислительных элементов и количество потоков может различаться – для обозначения числа потоков в дальнейшем будем использовать переменную π, т.е. $\pi = q^2$. Дополнительно можно отметить заранее, что для эффективного выполнения вычислений количество потоков π должно быть, по крайней мере, кратным числу вычислительных элементов p.

На рис. 7.15 приведена схема информационного взаимодействия подзадач в случае, когда для выполнения алгоритма создано 4 потока.

Рассмотрев представленную схему параллельных вычислений, можно сделать вывод о том, что информационная зависимость базовых подзадач проявляется только на этапе суммирования результатов перемножения блоков матрицы A и блоков вектора b.

7.7.3. Масштабирование и распределение подзадач по вычислительным элементам

При сделанных предположениях общее количество базовых подзадач совпадает с числом выделенных потоков. Так, например, если определить число потоков $\pi=q\cdot q$, то размер блоков матрицы A определяется соотношениями:

$k=m/q,\ l=n/q,$

где k и l есть количество строк и столбцов в блоках матрицы A. Такой способ определения размера блоков приводит к тому, что объем вычислений в каждой подзадаче является равным и, тем самым, достигается полная балансировка вычислительной нагрузки между потоками.

Формат: Список

7.7.4. Анализ эффективности

1. Первый вариант. Пусть каждый поток параллельной программы определяет блок матрицы и вектора, которые он должен обработать, после этого умножение блоков выполняется потоками параллельно. Далее элементы результирующего вектора вычисляются при помощи редукции полученных частичных результатов (см. рис. 7.15).

При анализе эффективности данного параллельного алгоритма умножения матрицы на вектор, основанного на блочном разделении матрицы, обратим внимание на дополнительные вычислительные операции: после того, как каждый поток выполнил умножение блока матрицы на блок вектора, необходимо сложить вектора частичных результатов для получения результирующего вектора. Если для выполнения параллельного алгоритма использовалось π потоков, то, с учетом использованной при реализации алгоритма схемы редукции данных, количество дополнительных операций равно $n\cdot\pi$. Таким образом, время вычислений можно определить по формуле:

$$T_{calc} = \left(\frac{n \cdot (2n-1)}{p} + n \cdot \pi \right) \cdot \tau .$$

Для оценки полного времени выполнения параллельного алгоритма можно использовать все положения, использованные при выводе необходимых аналитических соотношений для параллельного алгоритма при ленточном горизонтальном разделении данных (см. выражение (7.3)). Как результат, в данном случае оценка сложности параллельных вычислений может быть представлена в следующем виде:

$$\frac{8\cdot n^2}{\beta}\le T_p\le\left(\frac{n\cdot(2n-1)}{p}+n\cdot\pi\right)\cdot\tau+\frac{8\cdot n^2}{\beta} \qquad (7.9)$$

2. Второй вариант. Стандарт OpenMP предусматривает возможность организации *вложенных параллельных секций*. Это значит, что внутри одной параллельной секции можно объявить вторую параллельную секцию, которая разделит каждый из потоков внешней параллельной секции на несколько вложенных потоков.

Продемонстрируем описанный подход в случае, когда при объявлении каждой новой параллельной секции поток выполнения разделяется на два. При перемножении матрицы на вектор для внешнего цикла объявим внешнюю параллельную секцию. Потоки этой секции разделяют между собой строки матрицы – каждый поток в своих вычислениях использует горизонтальную полосу матрицы. Далее, используя механизм вложенного параллелизма, объявим внутреннюю параллельную секцию: потоки этой параллельной секции делят между собой столбцы матрицы. Таким образом будет реализовано блочное разделение данных. Схема выполнения данного параллельного алгоритма представлена на рис. 7.16.

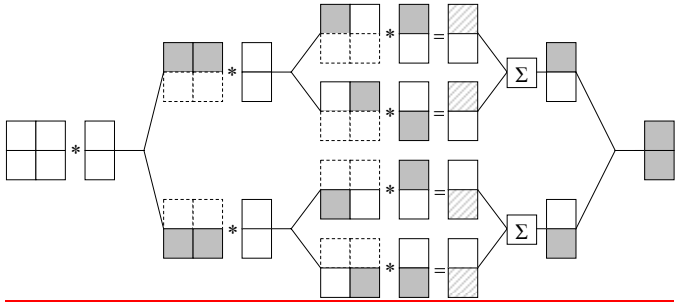


Рис. 7.16. Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор с использованием разбиения матрицы на блоки и вложенного параллелизма

Отформатировано: Подпись к рисунку, Справа: -0,01 см

При анализе данной реализации параллельного алгоритма умножения матрицы на вектор, можно выделить дополнительные вычислительные операции: после вычисления частичного результата каждым параллельным потоком «внутренней» параллельной секции необходимо выполнить редукцию и записать полученный результат в соответствующий элемент результирующего вектора. Напомним, что для выполнения задачи используется π параллельных потоков, $\pi=q\cdot q$, и их число может отличаться от количества имеющихся вычислительных элементов (процессоров или ядер). С учетом данного обстоятельства можно определить, что время выполнения вычислений ограничено сверху величиной:

$$T_{calc}=\left(\frac{n\cdot(2n-1)}{p}+n\cdot q\right)\cdot\tau .$$

В случае же, когда количества вычислительных элементов совпадает с числом потоков, т.е. $p=\pi$, оценка времени вычислений может быть получена более точно:

$$T_{calc}=\left(\frac{n\cdot(2n-1)}{p}+\frac{n}{q}\cdot\log_2 q+\frac{n}{q}\right)\cdot\tau$$

При выполнении вычислений, «внутренние» параллельные секции создаются и закрываются много раз, кроме того, дополнительное время тратится на синхронизацию и выполнение операции редукции.

Отформатировано: Обычный

Накладные расходы на организацию и закрытие параллельных секций были измерены при анализе параллельного алгоритма, основанного на вертикальном разделении данных (см. подраздел 7.6). В данном случае каждый поток создает параллельные секции n/q. Раз. Таким образом, время выполнения параллельного алгоритма может быть вычислено по формуле:

$$T_p=\left(\frac{n\cdot(2n-1)}{p}+\frac{n}{q}\cdot\log_2 q+\frac{n}{q}\right)\cdot\tau+\frac{8\cdot n^2}{\beta}+\frac{n}{q}\cdot\delta \qquad (7.10)$$

7.7.5. Программная реализация

Формат: Список

1. Первый вариант. Как уже отмечалось выше, для эффективного выполнения алгоритма умножения матрицы на вектор необходимо, чтобы количество параллельных потоков являлось полным квадратом. В приведенном ниже примере используется 4 потока (значение переменной *GridThreadsNum* устанавливается равным 4 – значение данной переменной должно переустанавливаться при изменении необходимого числа потоков). Определение числа потоков «вручную» до начала выполнения программы гарантирует корректную работу приложения на вычислительной системе с любым количеством доступных вычислительных элементов. Однако, если в вычислительной системе больше вычислительных элементов, чем определено параллельных потоков, эффективное использование ресурсов не может быть достигнуто.

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

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

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

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

В представленном варианте алгоритма номер потока сохраняется в переменной *ThreadID*, которая при объявлении параллельной секции определена как *private*, то есть для этой переменной в каждом потоке создается локальная копия. Положения блоков матрицы и вектора, которые должны обрабатываться потоком, определяются в зависимости от значения переменной *ThreadID*.

Для накопления результатов умножения блоков матрицы и вектора в каждом потоке используется локальная область памяти *pThreadResult*. Для того, чтобы получить элемент результирующего вектора, необходимо просуммировать соответствующие элементы векторов *pThreadResult* со всех потоков. Для обеспечения контроля доступа к разделяемому ресурсу *pResult* со всех потоков параллельной программы используется *механизм критических секций*. Для объявления критической секции служит директива:

```
#pragma omp critical
  structured block
```

Блок операций, следующий за объявлением критической секции, в каждый момент времени может выполняться только одним потоком. Подобная организация программы гарантирует единственность доступа потоков для изменения разделяемых данных внутри критической секции.

Представим возможный вариант параллельной программы умножения матрицы на вектор с использованием алгоритма разбиения матрицы по блокам. Ниже приведем только код основной функции, выполняющей параллельный алгоритм умножения матрицы на вектор.

```
// Function for matrix-vector multiplication
void ResultCalculation(double* pMatrix, double* pVector, double* pResult,
  int Size) {
  int ThreadID;
  int GridThreadsNum = 4;
  int GridSize = int (sqrt(double(GridThreadsNum)));
  int BlockSize = Size/GridSize; // we assume that the matrix size is
                                 // divisible by the grid size
  omp_set_num_threads(GridThreadsNum);

#pragma omp parallel private (ThreadID)
  {
    ThreadID = omp get thread num();
    double * pThreadResult = new double [Size];
    for (int i=0; i<Size; i++) {
      pThreadResult[i] = 0;
    }
    int i start = (int(ThreadID/GridSize))*BlockSize;
    int j start = (ThreadID%GridSize)*BlockSize;
    for (int i=0; i< BlockSize; i++) {
      for (int j=0; j < BlockSize; j++)
        pThreadResult[i+i start] +=
            pMatrix[(i+i start)*Size+(j+j start)] * pVector[j+i start];
    }
#pragma omp critical
```

```
    for (int i=0; i<Size; i++) {
      pResult[i] += pThreadResult[i];
    }
    delete [] pThreadResult;
  } // pragma omp parallel
}
```

Следует отметить, что для запоминания результатов потоков в *pThreadResult* и их объединения в массиве *pResult* используется несколько "упрощенная" схема реализации с целью получения более простого варианта программного кода.

2. Второй вариант. Использование механизма OpenMP для организации вложенного параллелизма позволяет значительно проще реализовать параллельный алгоритм умножения матрицы на вектор, основанный на блочном разделении матриц. Однако следует отметить, что на данный момент не все компиляторы, реализующие стандарт OpenMP, поддерживают вложенный параллелизм. Для компиляции представленного ниже кода использовался компилятор Intel C++ Compiler 9.0 for Windows, поддерживающий вложенный параллелизм.

Для того, чтобы включить поддержку вложенного параллелизма, необходимо вызвать функцию *omp_set_nested* библиотеки OpenMP (функция должна быть вызвана из последовательного участка программы):

```
    void omp_set_nested(int nested);
```

Для разделения матрицы на полосы, необходимо разделить внешний цикл алгоритма умножения матрицы на вектор при помощи директивы *omp parallel for*. Для дальнейшего разделения полос матрицы на блоки – разделить внутренний цикл между потоками при помощи той же директивы. Поскольку для вычисления каждого элемента результирующего вектора создается параллельная секция, внутри которой каждый поток вычисляет частичную сумму для этого элемента, необходимо применить механизм редукции, подробно описанный в подразделе 7.6.

```
// Function for matrix-vector multiplication
void ResultCalculation(double* pMatrix, double* pVector, double* pResult,
  int Size) {
  int NestedThreadsNum = 2;
  omp set num threads(NestedThreadsNum);
  omp set nested(true);
#pragma omp parallel for
  for (int i=0; i<Size; i++) {
    double ThreadResult = 0;
#pragma omp parallel for reduction(+:ThreadResult)
    for (int j=0; j<Size; j++)
      ThreadResult += pMatrix[i*Size+j]*pVector[j];
    pResult[i] = ThreadResult;
  }
}
```

Отметим, что в приведенной программе для задания количество потоков, создаваемых на каждом уровне вложенности параллельных областей, используется переменная *NestedThreadsNum* (в данном варианте программы ее значение устанавливается равным 2 – данное значение должно переустанавливаться при изменении необходимого числа потоков).

7.7.6. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма проводились при тех же условиях, что и ранее выполненные расчеты (см. п. 7.5.5).

1. Первый вариант реализации. Результаты экспериментов приведены в таблице 7.7. и на рис. 7.18.

Таблица 7.3. Результаты вычислительных экспериментов по исследованию параллельного алгоритма умножения матрицы на вектор при блочном разделении данных

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
1000	0,0031	0,0027	1,1220
2000	0,0107	0,0078	1,3640
3000	0,0232	0,0165	1,4035
4000	0,0408	0,0292	1,3948

Отформатировано: интервал После: 0 пт

Удалено: При реализации первого варианта блочного параллельного алгоритма умножения матрицы на вектор необходимо «вручную» определить блоки данных и блоки вектора, над которыми каждый поток выполняет вычисления. В то же время, стандарт OpenMP предусматривает возможность организации *вложенных параллельных секций*. Это значит, что внутри одной параллельной секции можно объявить вторую параллельную секцию, которая разделит каждый из потоков внешней параллельной секции на несколько вложенных потоков. ¶
Продемонстрируем описанный подход в случае, когда при объявлении каждой новой параллельной секции поток выполнения разделяется на два. При перемножении матрицы на вектор для внешнего цикла объявим внешнюю параллельную секцию. Потоки этой секции разделяют между собой строки матрицы – каждый поток в своих вычислениях использует горизонтальную полосу матрицы. Далее, используя механизм вложенного параллелизма, объявим внутреннюю параллельную секцию: потоки этой параллельной секции делят между собой столбцы полосы матрицы. Таким образом будет реализовано блочное разделение данных. Схема выполнения данного параллельного алгоритма представлена на рис. 7.16.¶ ... [2]

Удалено: ¶
<#>Анализ эффективности¶
1. Первый вариант реализации. При анализе эффективности первой реализации параллельного алгоритма умножения матрицы на вектор, основанного на блочном разделении матрицы, обратим внимание на дополнительные вычислительные операции: после того, как каждый поток выполнил умножение блока матрицы на блок вектора, необходимо сложить вектора частичных результатов для получения результирующего вектора. Если для выполнения параллельного алгоритма использовалось π потоков, то, с учетом использованной при реализации алгоритма схемы редукции данных, количество дополнительных операций равно $n \cdot \pi$. Таким образом, время вычислений можно определить по формуле:¶

$$T_{calc} = \left(\frac{n \cdot (2n-1)}{p} + n \cdot \pi \right) \cdot \tau \,.$$ ¶ ... [3]

5000	0,0632	0,0454	1,3938
6000	0,0908	0,0646	1,4051
7000	0,1232	0,0877	1,4047
8000	0,1611	0,1141	1,4115
9000	0,2036	0,1446	1,4078
10000	0,2508	0,1787	1,4032

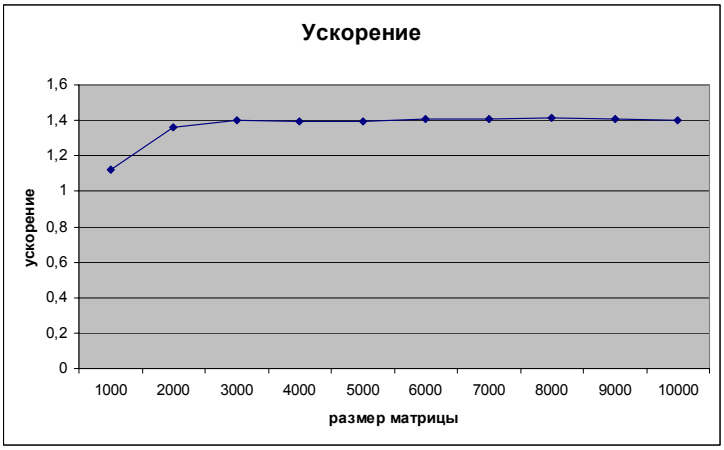


Рис. 7.17. Зависимость ускорения от размера матриц при выполнении параллельного алгоритма умножения матрицы на вектор (блочное разбиение матрицы) для разных размеров матриц

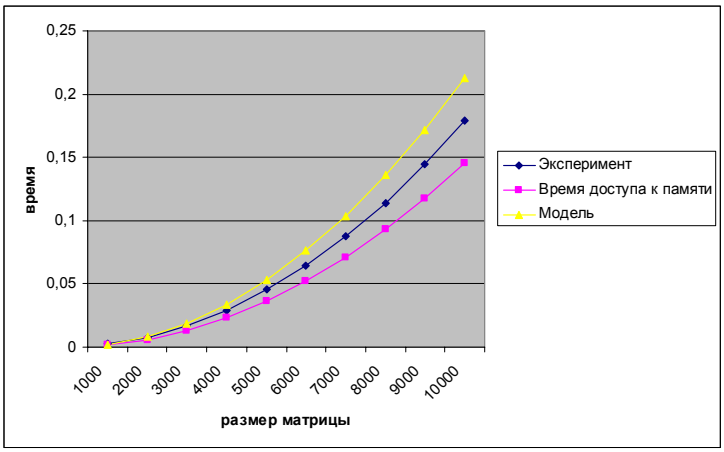
Формат: Список

Инструмент Call Graph профилировщика Intel VTune Perfomance Analyzer показывает, что затраты на организацию параллелизма не превосходят 5%.

В таблице 7.8 и на рис. 7.19 представлены результаты сравнения времени выполнения параллельного алгоритма умножения матрицы на вектор с использованием четырех потоков со временем, полученным при помощи модели (7.9).

Таблица 7.8. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор, основанного на блочном разбиении матрицы, с использованием четырех потоков

Размер матриц	Эксперимент	Время счета (модель)	Время доступа к памяти (модель)	Общее время (модель)
1000	0,0027	0,0006	0,0015	0,0021
2000	0,0078	0,0023	0,0058	0,0085
3000	0,0165	0,0053	0,0131	0,0191
4000	0,0292	0,0094	0,0233	0,0340
5000	0,0454	0,0147	0,0364	0,0531
6000	0,0646	0,0211	0,0524	0,0764
7000	0,0877	0,0287	0,0713	0,1040
8000	0,1141	0,0375	0,0931	0,1358
9000	0,1446	0,0475	0,1178	0,1719
10000	0,1787	0,0586	0,1455	0,2122



Формат: Список

Рис. 7.18. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма от объема исходных данных при использовании четырех потоков (блочное разбиение матрицы)

2. Второй вариант реализации. Результаты экспериментов приведены в таблице 7.9. и на рис. 7.20.

Таблица 7.9. Результаты вычислительных экспериментов по исследованию параллельного алгоритма умножения матрицы на вектор при блочном разделении данных и использовании вложенного параллелизма

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
1000	0,0031	0,0077	0,3961
2000	0,0107	0,0161	0,6617
3000	0,0232	0,0257	0,9020
4000	0,0408	0,0374	1,0890
5000	0,0632	0,0539	1,1737
6000	0,0908	0,0742	1,2234
7000	0,1232	0,0968	1,2729
8000	0,1611	0,1260	1,2785
9000	0,2036	0,1528	1,3321
10000	0,2508	0,1937	1,2947

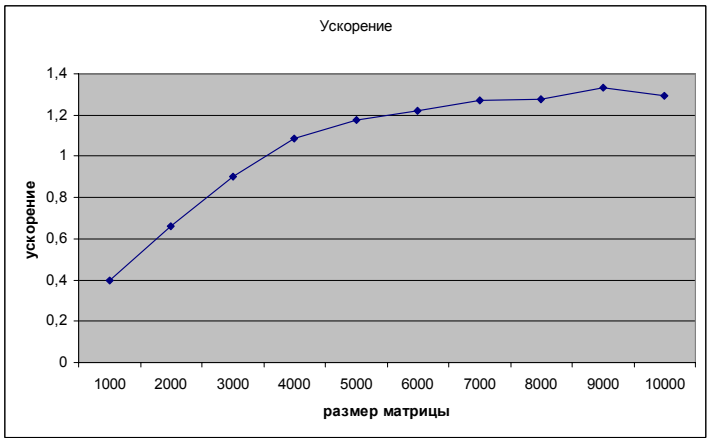


Рис. 7.19. Зависимость ускорения от размера матриц при выполнении параллельного алгоритма умножения матрицы на вектор (блочное разбиение матрицы) с использованием вложенного параллелизма для разных размеров матриц

Формат: Список

Как видно из представленных таблиц и графиков, при небольших размерах объектов и незначительных объемах вычислений, алгоритм, основанный на автоматическом распределении вычислительной нагрузки, существенно проигрывает по производительности алгоритму, основанному на «ручном» разделении данных. Это объясняется тем, что в этом случае больший вес имеют накладные расходы на организацию параллелизма. Однако с ростом вычислительной нагрузки, доля времени, затраченного на организацию параллелизма, становится меньше, производительность алгоритма повышается. При максимальных размерах матрицы и вектора он практически не уступает алгоритму, реализованному на основе «ручного» разделения.

Для оценки доли накладных расходов на обеспечение параллелизма снова воспользуемся профилировщиком Intel VTune Performance Analyzer (рис. 7.20).

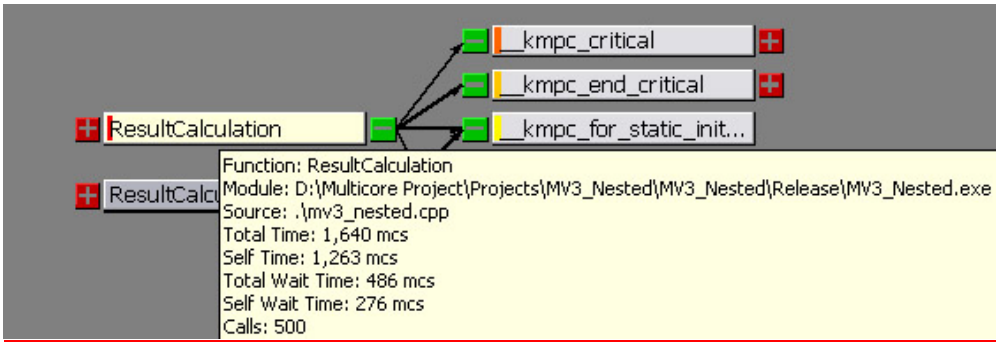


Рис. 7.20. Общее и собственное время выполнения функции *ParallelResultCalculation* при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на блочном разделении данных, реализованного с помощью вложенного параллелизма

Отформатировано: Подпись к рисунку

В таблице 7.10 и на рис. 7.21 представлены результаты сравнения времени выполнения параллельного алгоритма умножения матрицы на вектор с использованием четырех потоков со временем, полученным при помощи модели (7.10).

Таблица 7.10. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор, основанного на блочном разбиении матрицы, с использованием четырех потоков (вложенный параллелизм)

Размер матриц	Эксперимент	Модель
1000	0,0031	0,0070
2000	0,0107	0,0182
3000	0,0232	0,0334
4000	0,0408	0,0526
5000	0,0632	0,0760
6000	0,0908	0,1035
7000	0,1232	0,1350
8000	0,1611	0,1706
9000	0,2036	0,2103
10000	0,2508	0,2541

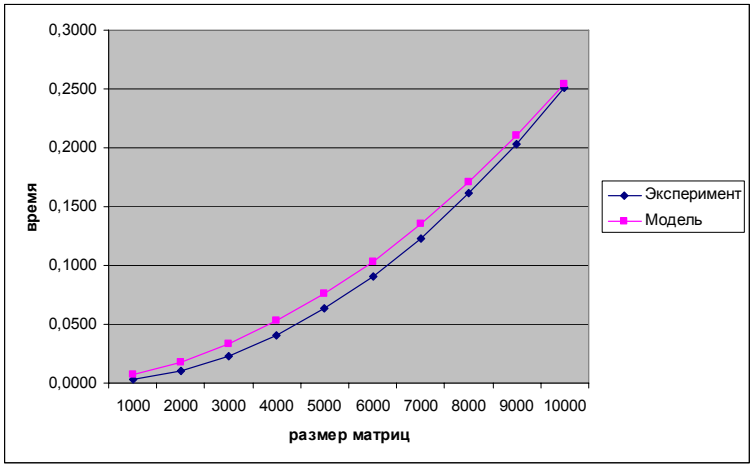


Рис. 7.21. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма от объема исходных данных при использовании четырех потоков (блочное разбиение матрицы, вложенный параллелизм)

Отформатировано: Подпись к рисунку

Удалено: ¶

7.8. Краткий обзор раздела

В данном разделе на примере задачи умножения матрицы на вектор рассматриваются возможные схемы разделения матриц между потоками параллельной программы, предназначенной для выполнения на многопроцессорной вычислительной системе с общей памятью и/или на вычислительной системе с многоядерными процессорами. Эти схемы разделения данных являются общими и могут быть использованы для организации параллельных вычислений при выполнении любых матричных операций. В числе излагаемых схем способы разбиения матриц на *полосы* (по вертикали или горизонтали) или на прямоугольные наборы элементов (*блоки*).

Далее в разделе с использованием рассмотренных способов разделения матриц подробно излагаются три возможных варианта параллельного выполнения операции умножения матрицы на вектор. Первый алгоритм основан на разделении матрицы между потоками по строкам, второй – на разделении матрицы по столбцам, а третий – на блочном разделении данных. Каждый алгоритм представлен в соответствии с общей схемой, описанной в разделе 6, - вначале определяются базовые подзадачи, затем выделяются информационные зависимости подзадач, далее обсуждается масштабирование и распределение подзадач между вычислительными элементами. В завершение для каждого алгоритма проводится анализ эффективности получаемых параллельных вычислений и приводятся результаты вычислительных экспериментов. Для всех рассматриваемых параллельных алгоритмов умножения матрицы на вектор приводятся возможные варианты программной реализации.

Полученные показатели ускорения и эффективности показывают, что все используемые способы разделения данных приводят к равномерной балансировке вычислительной нагрузки, и отличия возникают только в трудоемкости выполняемых информационных взаимодействий между потоками. В этом отношении интересным представляется проследить, как выбор способа разделения данных влияет на характер организации параллельных участков программы, выделить основные различия в использовании потоками общих ресурсов и необходимых операций по синхронизации доступа к ним.

На рис. 7.22 на общем графике представлены показатели ускорения, полученные в результате выполнения вычислительных экспериментов для всех рассмотренных алгоритмов. Как можно заметить, некоторое преимущество по ускорению имеет параллельный алгоритм умножения матрицы на вектор при ленточном разделении по строкам.

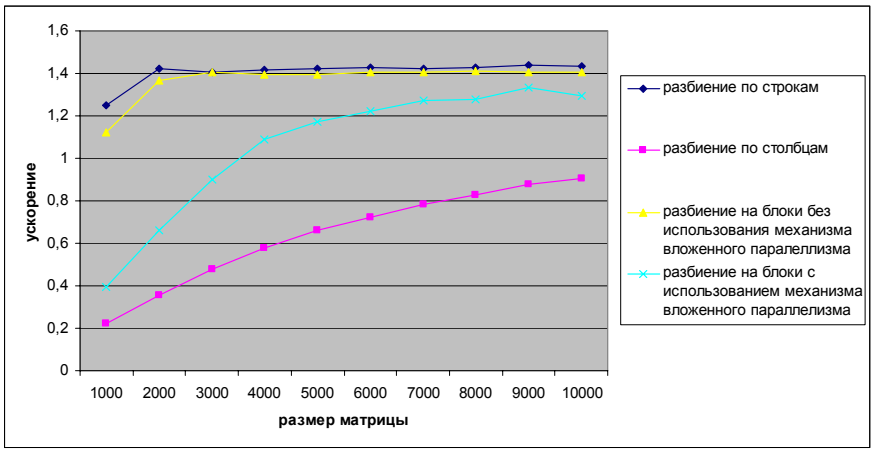


Рис. 7.22. Показатели ускорения рассмотренных параллельных алгоритмов умножения по результатам вычислительных экспериментов

7.9. Обзор литературы

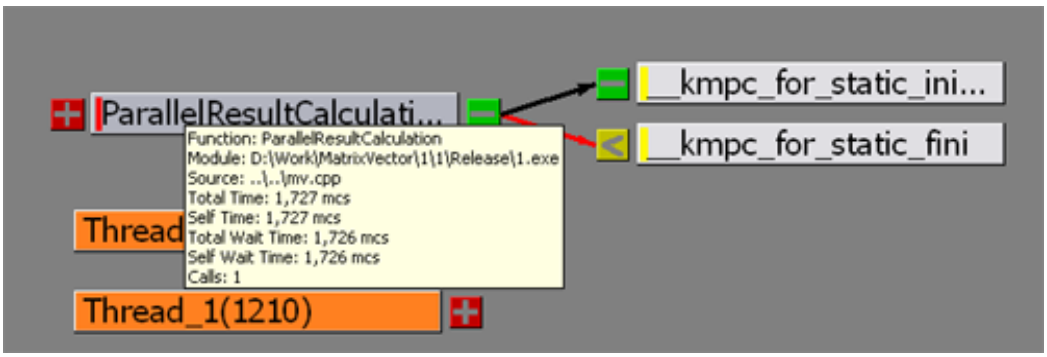
Задача умножения матрицы на вектор часто используется как демонстрационный пример параллельного программирования и, как результат, широко рассматривается в литературе. В качестве дополнительного учебного материала могут быть рекомендованы работы Воеводин В.В. и Воеводин Вл.В. (2002), Kumar (1994) и Quinn (2003). Широкое обсуждение вопросов параллельного выполнения матричных вычислений выполнено в работе Dongarra, et al. (1999).

7.10. Контрольные вопросы

1. Назовите основные способы распределения элементов матрицы между потоками.
2. В чем состоит постановка задачи умножения матрицы на вектор?
3. Какова вычислительная сложность последовательного алгоритма умножения матрицы на вектор?
4. Какие подходы могут быть предложены для разработки параллельных алгоритмов умножения матрицы на вектор?
5. Представьте общие схемы рассмотренных параллельных алгоритмов умножения матрицы на вектор.
6. Проведите анализ и получите показатели эффективности для одного из рассмотренных алгоритмов.
7. Какой из представленных алгоритмов умножения матрицы на вектор обладает лучшими показателями ускорения и эффективности?
8. Может ли использование циклической схемы разделения данных повлиять на время работы каждого из представленных алгоритмов?
9. Какие информационные взаимодействия выполняются для алгоритмов при ленточной схеме разделения данных? В чем различие необходимых операций по организации параллельных участков программы и синхронизации доступа к общим ресурсам при разделении матрицы по строкам и столбцам?
10. Какие информационные взаимодействия выполняются для блочного алгоритма умножения матрицы на вектор?
11. Какие средства технологии OpenMP и функции соответствующей библиотеки оказались необходимыми при программной реализации алгоритмов?

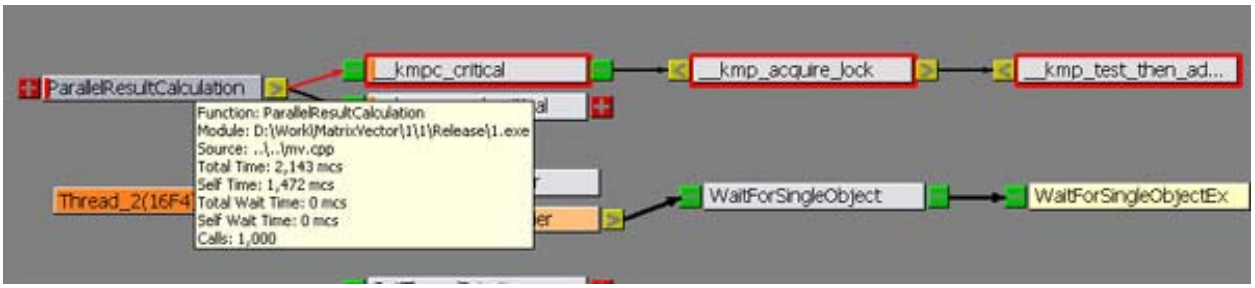
Для анализа программного кода и подтверждения выдвинутых предположений воспользуемся специализированной программной системой, предназначенной для профилирования и оптимизации программ, Intel VTune Performance Analyzer (см., например, [][ГВП1]). Данная программная система позволяет находить в программе критический (вычислительно-трудоемкий) код, измерять различные характеристики эффективности кода и наблюдать за процессом выполнения программы в динамике.

Проанализируем при помощи инструмента Call Graph (*Граф вызова функций*) профилировщика Intel VTune Performance Analyzer приложение, выполняющее параллельный алгоритм умножения матрицы на вектор, основанный на разделении матрицы на горизонтальные полосы, и сконцентрируем свое внимание на анализе выполнения функции *ParallelResultCalculation*. На рис. 7.10 представлен результат анализа. Видно, что собственное (без учета времени выполнения вызываемых функций) время и полное время выполнения функции совпадают. Это означает, что, несмотря на то, что функция *ParallelResultCalculation* вызывает другие функции (в данном случае – функции библиотеки времени исполнения OpenMP), практически все время тратится непосредственно на вычисления.



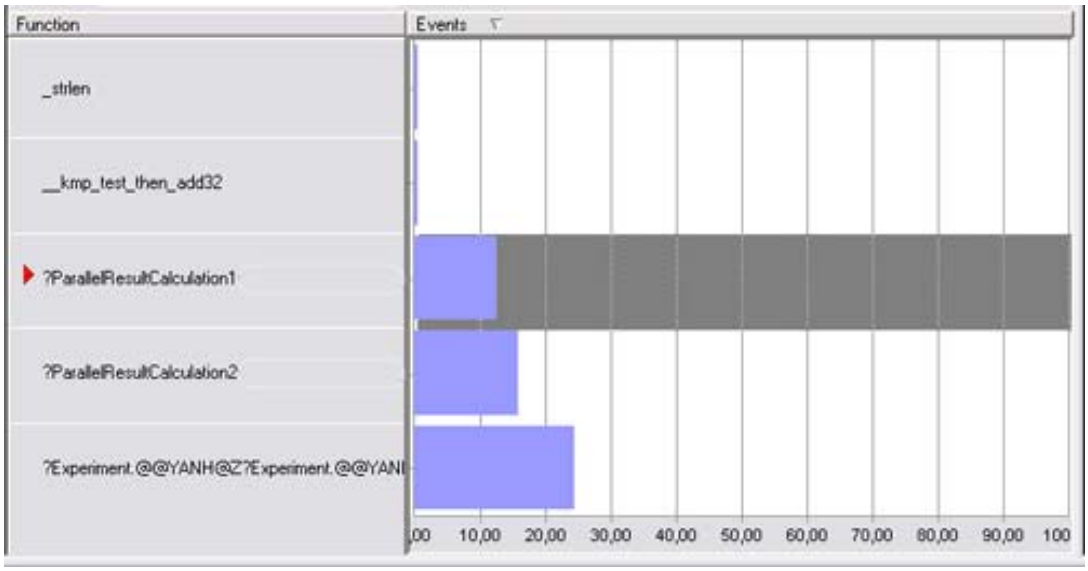
Общее и собственное время выполнения функции *ParallelResultCalculation* в случае разделения матрицы на горизонтальные полосы

Теперь перейдем к анализу параллельного алгоритма, основанного на разделении матрицы по столбцам. В данном случае собственное время выполнения функции *ParallelResultCalculation* существенно меньше полного времени ее выполнения. Большая доля времени тратится на выполнение вложенных функций, которыми являются функции, необходимые для организации и закрытия параллельных секций, а также для выполнения редукции (рис. 7.11).



Общее и собственное время выполнения функции *ParallelResultCalculation* в случае разделения матрицы на вертикальные полосы

Далее воспользуемся инструментом Sampling[1)] профилировщика Intel VTune Performance Analyzer, обеспечивающим автоматизированный процесс регистрации различных событий, возникающих в процессоре во время исполнения профилируемой программы. Создадим проект, в котором вызовем последовательно функцию, выполняющую параллельный алгоритм умножения матрицы на вектор, основанный на горизонтальном разделении матрицы (*ParallelResultCalculation1*), и функцию, выполняющую параллельный алгоритм, основанный на вертикальном разделении матрицы (*ParallelResultCalculation2*). Измерим число возникновений события выделения новой кэш-линии первого уровня (в Intel VTune Performance Analyzer данное событие называется L1 Cache Line Allocation). Результаты профилирования приложения представлены на рис. 7.12. Эксперименты проводились для матрицы размером 1000×1000 элементов.



Соотношение количества событий выделения кэш-линий первого уровня для двух параллельных алгоритмов умножения матрицы на вектор.

При выполнении параллельного алгоритма умножения матрицы на вектор, основанного на вертикальном разделении, выделяется больше (в среднем на 25%) линий кэша первого уровня.

Далее, оценим накладные расходы на организацию параллельности на каждой итерации алгоритма, основанного на вертикальном разделении данных. Для этого подготовим еще один вариант параллельного алгоритма умножения матрицы на вектор при разделении матрицы по столбцам. Образуем в этом варианте программы потоки, которые самостоятельно определяют (используя идентификатор потока) блоки элементов матрицы и вектора для обработки. Далее потоки параллельно выполняют умножение своих столбцов матрицы на элементы вектора, результат умножения сохраняется в общем массиве *AllResults*. Количество элементов в этом массиве равно *Size×ThreadNum*, где *ThreadNum* – общее число потоков. Потоки осуществляют запись частичных результатов в непересекающиеся сегменты массива *AllResults*: нулевой поток осуществляет запись в элементы с *0* по *Size-1*, первый поток – в элементы с *Size* по *2·Size-1*, и т.д. После выполнения умножения элементы вектора *AllResults* необходимо просуммировать для получения элементов результирующего вектора.

```
void ParallelResultCalculation(double* pMatrix, double* pVector,
  double* pResult, double* AllResults, int Size) {
  int ThreadNum;
#pragma omp parallel shared (ThreadNum)
  {
    ThreadNum = omp_get_num_threads();
    int ThreadID = omp_get_thread_num();
    int BlockSize = Size/ThreadNum;
    double IterResult;
    for (int i=0; i<Size; i++) {
      IterResult = 0;
      for (int j=0; j<BlockSize; j++)
        IterResult += pMatrix[i*Size+j+ThreadID*BlockSize] *
          pVector[j+ThreadID*BlockSize];
      AllResults[Size*ThreadID+i] = IterResult;
    }
  }
  for (int i=0; i<Size; i++)
    for (int j=0; j<ThreadNum; j++)
      pResult[i] += AllResults[j*Size+i];
}
```

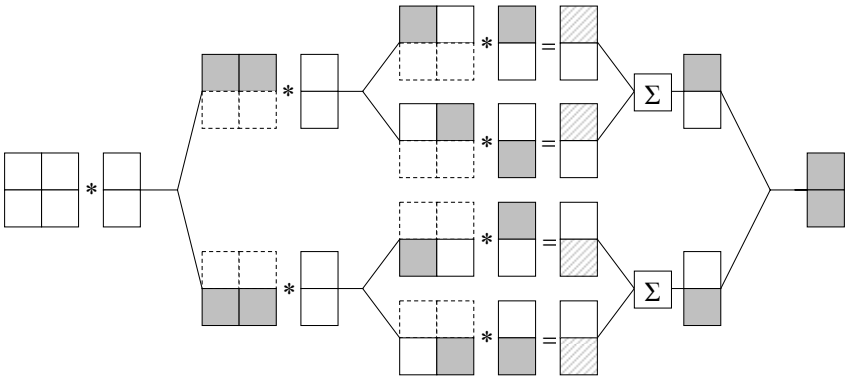
Как видно из представленного программного кода, при выполнении данного алгоритма параллельная секция создается только один раз, и, следовательно, время, необходимое для выполнения функций библиотеки OpenMP, отвечающих за организацию и закрытие параллельных секций, пренебрежимо мало. Для того,

алгоритма и поделить полученную разницу на количество параллельных секций. Эксперименты показывают, что величина δ равна 10 микросекунд ($1 \cdot 10^{-5}$ с).

Стр. 23: [2] Удалено **labutina.a** **06.09.2007 12:12:00**

При реализации первого варианта блочного параллельного алгоритма умножения матрицы на вектор необходимо «вручную» определить блоки данных и блоки вектора, над которыми каждый поток выполняет вычисления. В то же время, стандарт OpenMP предусматривает возможность организации *вложенных параллельных секций*. Это значит, что внутри одной параллельной секции можно объявить вторую параллельную секцию, которая разделит каждый из потоков внешней параллельной секции на несколько вложенных потоков.

Продемонстрируем описанный подход в случае, когда при объявлении каждой новой параллельной секции поток выполнения разделяется на два. При перемножении матрицы на вектор для внешнего цикла объявим внешнюю параллельную секцию. Потоки этой секции разделяют между собой строки матрицы – каждый поток в своих вычислениях использует горизонтальную полосу матрицы. Далее, используя механизм вложенного параллелизма, объявим внутреннюю параллельную секцию: потоки этой параллельной секции делят между собой столбцы полосы матрицы. Таким образом будет реализовано блочное разделение данных. Схема выполнения данного параллельного алгоритма представлена на рис. 7.16.



Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор с использованием разбиения матрицы на блоки и вложенного параллелизма

Использование механизма OpenMP для организации вложенного параллелизма позволяет значительно проще реализовать параллельный алгоритм умножения матрицы на вектор, основанный на блочном разделении матриц. Однако следует отметить, что на данный момент не все компиляторы, реализующие стандарт OpenMP, поддерживают вложенный параллелизм. Для компиляции представленного ниже кода использовался компилятор Intel C++ Compiler 9.0 for Windows, поддерживающий вложенный параллелизм.

Стр. 23: [3] Удалено **labutina.a** **06.09.2007 12:04:00**

Анализ эффективности

1. Первый вариант реализации. При анализе эффективности первой реализации параллельного алгоритма умножения матрицы на вектор, основанного на блочном разделении матрицы, обратим внимание на дополнительные вычислительные операции: после того, как каждый поток выполнил умножение блока матрицы на блок вектора, необходимо сложить вектора частичных результатов для получения результирующего вектора. Если для выполнения параллельного алгоритма использовалось π потоков, то, с учетом использованной при реализации алгоритма схемы редукции данных, количество дополнительных операций равно $n \cdot \pi$. Таким образом, время вычислений можно определить по формуле:

$$T_{calc} = \left(\frac{n \cdot (2n-1)}{p} + n \cdot \pi \right) \cdot \tau .$$

Для оценки полного времени выполнения параллельного алгоритма можно использовать все положения, использованные при выводе необходимых аналитических соотношений для параллельного алгоритма при ленточном горизонтальном разделении данных (см. выражение (7.3)). Как результат, в данном случае оценка сложности параллельных вычислений может быть представлена в следующем виде:

Инструмент Call Graph профилировщика Intel VTune Perfomance Analyzer показывает, что затраты на организацию параллелизма не превосходят 5%.

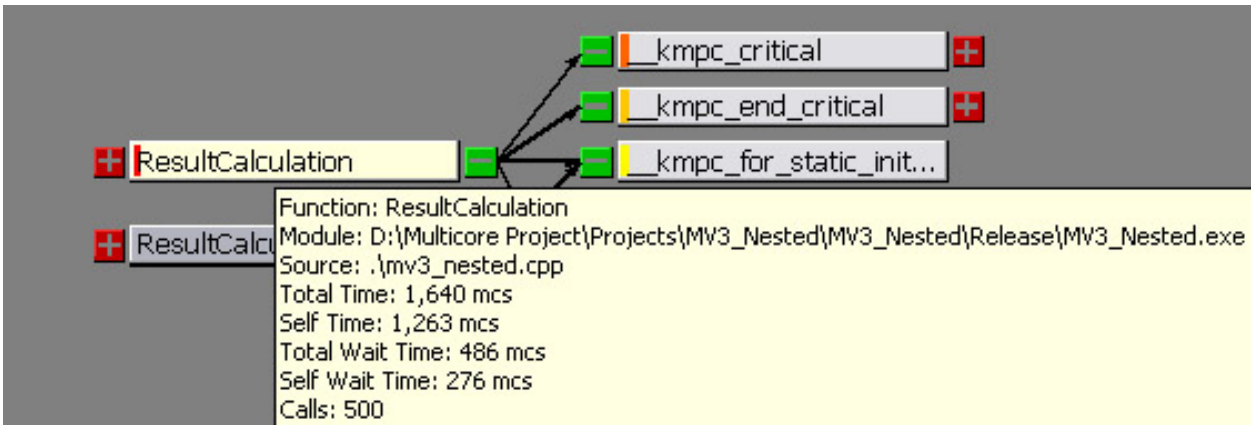
2. Второй вариант реализации. Выполнив анализ второй вариант программной реализации параллельного алгоритма умножения матрицы на вектор, можно выделить дополнительные вычислительные операции: после вычисления частичного результата каждым параллельным потоком «внутренней» параллельной секции необходимо выполнить редукцию и записать полученный результат в соответствующий элемент результирующего вектора. Напомним, что для выполнения задачи используется π параллельных потоков, $\pi=q\cdot q$, и их число может отличаться от количества имеющихся вычислительных элементов (процессоров или ядер). С учетом данного обстоятельства можно определить, что время выполнения вычислений ограничено сверху величиной:

$$T_{calc}=\left(\frac{n\cdot(2n-1)}{p}+n\cdot q\right)\cdot\tau\,.$$

В случае же, когда количества вычислительных элементов совпадает с числом потоков, т.е. $p=\pi$, оценка времени вычислений может быть получена более точно:

$$T_{calc}=\left(\frac{n\cdot(2n-1)}{p}+\frac{n}{q}\cdot\log_2 q+\frac{n}{q}\right)\cdot\tau$$

При выполнении вычислений, «внутренние» параллельные секции создаются и закрываются много раз, кроме того, дополнительное время тратится на синхронизацию и выполнение операции редукции. Для оценки доли накладных расходов на обеспечение параллелизма снова воспользуемся профилировщиком Intel VTune Performance Analyzer (рис. 7.17).



Общее и собственное время выполнения функции *ParallelResultCalculation* при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на блочном разделении данных, реализованного с помощью вложенного параллелизма

Накладные расходы на организацию и закрытие параллельных секций были измерены при анализе параллельного алгоритма, основанного на вертикальном разделении данных (см. подраздел 7.6). В данном случае каждый поток создает параллельные секции *n/q*. Раз. Таким образом, время выполнения параллельного алгоритма может быть вычислено по формуле:

$$T_p=\left(\frac{n\cdot(2n-1)}{p}+\frac{n}{q}\cdot\log_2 q+\frac{n}{q}\right)\cdot\tau+\frac{8\cdot n^2}{\beta}+\frac{n}{q}\cdot\delta \qquad (7.10)$$