7. Умножение 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) излагается более общий случай 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 процессорами 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 процессорами используемой вычислительной системы. Выбор способа разделения матриц приводит к определению конкретного метода параллельных вычислений; существование разных схем распределения данных порождает целый ряд *параллельных алгоритмов матричных вычислений*.

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

1. Ленточное разбиение матрицы. При *ленточном* (*block-striped*) разбиении каждому процессору 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)

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

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 процессоров p, т.е. $m = k \cdot p$). Во всех алгоритмах матричного умножения и умножения матрицы на вектор, которые будут рассмотрены нами в этом и следующем разделах, используется разделение данных на непрерывной основе.

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

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

Циклическая схема формирования полос может оказаться полезной для лучшей балансировки вычислительной нагрузки процессоров (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 процессоров 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:

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

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

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

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

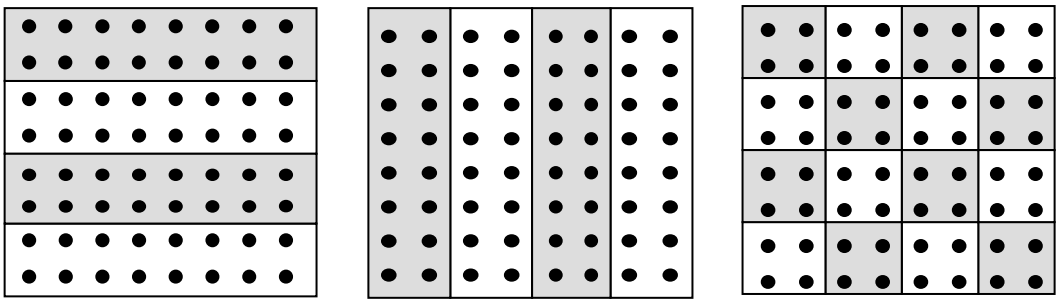

Рис. 7.1. Способы распределения элементов матрицы между процессорами вычислительной системы

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 процессорами. Разделение данных меняет схему взаимодействия процессоров, поэтому каждый из представленных методов существенным образом отличается от двух остальных.

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.

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

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 $O(mn)$.

7.5. Разделение данных

При выполнении параллельных алгоритмов умножения матрицы на вектор, кроме матрицы A необходимо разделить еще вектор b и вектор результата c. Элементы векторов можно *продублировать*,

то есть скопировать все элементы вектора на все процессоры, составляющие многопроцессорную вычислительную систему, или *разделить* между процессорами. При *блочном* разбиении вектора из n элементов каждый процессор обрабатывает непрерывную последовательность из k элементов вектора (мы предполагаем, что размерность вектора n нацело делится на число процессоров, т.е. $n = k \cdot p$).

Поясним, почему дублирование векторов b и c между процессорами является допустимым решением (далее для простоты изложения будем полагать, что $m=n$). Вектора b и c состоят из n элементов, т. е. содержат столько же данных, сколько и одна строка или один столбец матрицы. Если процессор хранит строку или столбец матрицы и одиночные элементы векторов b и c, то общее число сохраняемых элементов имеет порядок $O(n)$. Если процессор хранит строку (столбец) матрицы и все элементы векторов b и c, то общее число сохраняемых элементов также порядка $O(n)$. Таким образом, при дублировании и при разделении векторов требования к объему памяти из одного класса сложности.

7.6. 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.

7.6.1. Выделение информационных зависимостей

Для выполнения базовой подзадачи скалярного произведения процессор должен содержать соответствующую строку матрицы A и копию вектора b. После завершения вычислений каждая базовая подзадача определяет один из элементов вектора результата c.

Для объединения результатов расчета и получения полного вектора c на каждом из процессоров вычислительной системы необходимо выполнить операцию обобщенного сбора данных (см. раздел 6), в которой каждый процессор передает свой вычисленный элемент вектора c всем остальным процессорам. Этот шаг можно выполнить, например, с использованием функции *MPI_Allgather* из библиотеки MPI.

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

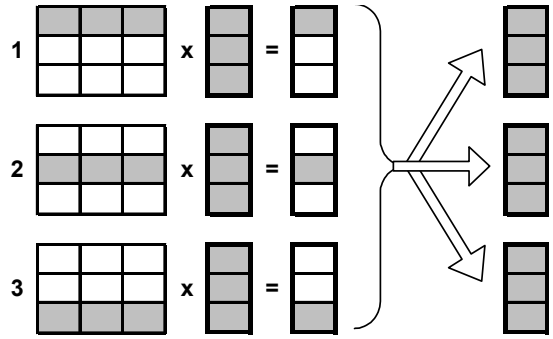

Рис. 7.2. Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на разделении матрицы по строкам

7.6.2. Масштабирование и распределение подзадач по процессорам

В процессе умножения плотной матрицы на вектор количество вычислительных операций для получения скалярного произведения одинаково для всех базовых подзадач. Поэтому в случае, когда число процессоров p меньше числа базовых подзадач m, мы можем объединить базовые подзадачи таким образом, чтобы каждый процессор выполнял несколько таких задач, соответствующих непрерывной последовательности строк матрицы A. В этом случае по окончании вычислений каждая базовая подзадача определяет набор элементов результирующего вектора c.

Распределение подзадач между процессорами вычислительной системы может быть выполнено произвольным образом.

7.6.3. Анализ эффективности

Для анализа эффективности параллельных вычислений здесь и далее будут строиться два типа оценок. В первой из них трудоемкость алгоритмов оценивается в количестве вычислительных операций, необходимых для решения поставленной задачи, без учета затрат времени на передачу данных между процессорами, а длительность всех вычислительных операций считается одинаковой. Кроме того,

константы в получаемых соотношениях, как правило, не указываются - для первого типа оценок важен прежде всего порядок сложности алгоритма, а не точное выражение времени выполнения вычислений. Как результат, в большинстве случаев подобные оценки получаются достаточно простыми и могут быть использованы для начального анализа эффективности разрабатываемых алгоритмов и методов.

Второй тип оценок направлен на формирование как можно более точных соотношений для предсказания времени выполнения алгоритмов. Получение таких оценок проводится, как правило, при помощи уточнения выражений, полученных на первом этапе. Для этого в имеющиеся соотношения вводятся параметры, задающие длительность выполнения операций, строятся оценки трудоемкости коммуникационных операций, указываются все необходимые константы. Точность получаемых выражений проверяется при помощи проведения вычислительных экспериментов, по результатам которых времена выполненных расчетов сравниваются с теоретически предсказанными оценками длительностей вычислений. Как результат, оценки подобного типа имеют, как правило, более сложный вид, но позволяют более точно оценивать эффективность разрабатываемых методов параллельных вычислений.

Рассмотрим трудоемкость алгоритма умножения матрицы на вектор. В случае, если матрица A квадратная ($m=n$), последовательный алгоритм умножения матрицы на вектор имеет сложность $T_1=n^2$. В случае параллельных вычислений каждый процессор производит умножение только части (полосы) матрицы A на вектор b, размер этих полос равен n/p строк. При вычислении скалярного произведения одной строки матрицы и вектора необходимо произвести n операций умножения и $(n-1)$ операций сложения. Следовательно, вычислительная трудоемкость параллельного алгоритма определяется выражением:

$$T_p=n^2/p. \tag{7.5}$$

С учетом этой оценки показатели ускорения и эффективности параллельного алгоритма имеют вид:

$$S_p = \frac{n^2}{n^2/p} = p\,, \quad E_p = \frac{n^2}{p\cdot(n^2/p)} = 1\,. \tag{7.6}$$

Построенные выше оценки времени вычислений выражены в количестве операций и, кроме того, определены без учета затрат на выполнение операций передачи данных. Используем ранее высказанные предположения о том, что выполняемые операции умножения и сложения имеют одинаковую длительность τ. Кроме того, будем предполагать также, что вычислительная система является однородной, т.е. все процессоры, составляющие эту систему, обладают одинаковой производительностью. С учетом введенных предположений время выполнения параллельного алгоритма, связанное непосредственно с вычислениями, составляет

$$T_p(calc) = \lceil n/p \rceil \cdot (2n-1)\cdot \tau$$

(здесь и далее операция $\lceil\ \rceil$ есть округление до целого в большую сторону).

Оценка трудоемкости операции обобщенного сбора данных уже выполнялась в разделе 6 (см. п.6.3.4). Как уже отмечалась ранее, данная операция может быть выполнена за $\lceil \log_2 p \rceil$ итераций[1.)] На первой итерации взаимодействующие пары процессоров обмениваются сообщениями объемом $w\lceil n/p \rceil$ (w есть размер одного элемента вектора c в байтах), на второй итерации этот объем увеличивается вдвое и оказывается равным $2w\lceil n/p \rceil$ и т.д. Как результат, длительность выполнения операции сбора данных при использовании модели Хокни может быть определена при помощи следующего выражения

$$T_p(comm) = \sum_{i=1}^{\lceil \log_2 p \rceil} (\alpha + 2^{i-1} w \lceil n/p \rceil / \beta) = \alpha \lceil \log_2 p \rceil + w \lceil n/p \rceil (2^{\lceil \log_2 p \rceil} - 1)/\beta\,, \tag{7.7}$$

где α – латентность сети передачи данных, β – пропускная способность сети. Таким образом, общее время выполнения параллельного алгоритма составляет

$$T_p = (n/p)\cdot(2n-1)\cdot\tau + \alpha\cdot\log_2 p + w(n/p)(p-1)/\beta \quad . \tag{7.8}$$

(для упрощения выражения в (7.8) предполагалось, что значения n/p и $\log_2 p$ являются целыми).

7.6.4. Программная реализация

Представим возможный вариант параллельной программы умножения матрицы на вектор с использованием алгоритма разбиения матрицы по строкам. При этом реализация отдельных модулей не приводится, если их отсутствие не оказывает влияние на понимании общей схемы параллельных вычислений.

[1)] Будем предполагать, что топология вычислительной системы допускает возможность такого эффективного способа выполнения операция обобщенного сбора данных (это возможно, в частности, если структура сети передачи данных имеет вид гиперкуба или полного графа.

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

```
// Программа 7.1
// Умножение матрицы на вектор – ленточное горизонтальное разбиение
// (исходный и результирующий вектора дублируются между процессорами)
void main(int argc, char* argv[]) {
  double* pMatrix;  // The first argument - initial matrix
  double* pVector;  // The second argument - initial vector
  double* pResult;  // Result vector for matrix-vector multiplication
  int Size;         // Sizes of initial matrix and vector
  double* pProcRows;
  double* pProcResult;
  int RowNum;
  double Start, Finish, Duration;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

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

  DataDistribution(pMatrix, pProcRows, pVector, Size, RowNum);

  ParallelResultCalculation(pProcRows, pVector, pProcResult, Size, RowNum);

  ResultReplication(pProcResult, pResult, Size, RowNum);

  ProcessTermination(pMatrix, pVector, pResult, pProcRows, pProcResult);

  MPI_Finalize();
}
```

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, double* &pProcRows, double* &pProcResult,
  int &Size, int &RowNum) {
  int RestRows; // Number of rows, that haven't been distributed yet
  int i;            // Loop variable

  if (ProcRank == 0) {
    do {
      printf("\nEnter size of the initial objects: ");
      scanf("%d", &Size);
      if (Size < ProcNum) {
        printf("Size of the objects must be greater than
                number of processes! \n ");
      }
    }
    while (Size < ProcNum);
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  RestRows = Size;
  for (i=0; i<ProcRank; i++)
    RestRows = RestRows-RestRows/(ProcNum-i);
  RowNum = RestRows/(ProcNum-ProcRank);

  pVector = new double [Size];
  pResult = new double [Size];
  pProcRows = new double [RowNum*Size];
```

```
  pProcResult = new double [RowNum];

  if (ProcRank == 0) {
    pMatrix = new double [Size*Size];
    RandomDataInitialization(pMatrix, pVector, Size);
  }
}
```

3. Функция DataDistribution. Осуществляет рассылку вектора *b* и распределение строк исходной матрицы *A* по процессам вычислительной системы. Следует отметить, что когда количество строк матрицы *n* не является кратным числу процессоров *p*, объем пересылаемых данных для процессов может оказаться разным[2)], и для передачи сообщений необходимо использовать функцию *MPI_Scatterv* библиотеки MPI.

```
// Data distribution among the processes
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
                      int Size, int RowNum) {
  int *pSendNum; // the number of elements sent to the process
  int *pSendInd; // the index of the first data element sent to the process
  int RestRows=Size; // Number of rows, that haven't been distributed yet

  MPI_Bcast(pVector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Alloc memory for temporary objects
  pSendInd = new int [ProcNum];
  pSendNum = new int [ProcNum];

  // Define the disposition of the matrix rows for current process
  RowNum = (Size/ProcNum);
  pSendNum[0] = RowNum*Size;
  pSendInd[0] = 0;
  for (int i=1; i<ProcNum; i++) {
    RestRows -= RowNum;
    RowNum = RestRows/(ProcNum-i);
    pSendNum[i] = RowNum*Size;
    pSendInd[i] = pSendInd[i-1]+pSendNum[i-1];
  }

  // Scatter the rows
  MPI_Scatterv(pMatrix , pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Free the memory
  delete [] pSendNum;
  delete [] pSendInd;
}
```

Следует отметить, что такое разделение действий генерации исходных данных и их рассылки между процессами может быть не оправданным в реальных параллельных вычислениях при большом объеме данных. Широко используемый подход в таких случаях состоит в организации передачи процессам сразу же после того, как данные процессов будут сгенерированы. Снижение затрат памяти для хранения данных может быть достигнуто также и за счет организации генерации данных в последнем процессе (при таком подходе память для пересылаемых данных и для данных процесса может быть одной и той же).

4. Функция ParallelResultCaculation. Данная функция производит умножение на вектор тех строк матрицы, которые распределены на данный процесс и, таким образом, получает блок результирующего вектора *b*.

```
// Function for calculating partial matrix-vector multiplication
void ParallelResultCalculation(double* pProcRows, double* pVector, double*
pProcResult, int Size, int RowNum) {
  int i, j;  // Loop variables
  for (i=0; i<RowNum; i++) {
    pProcResult[i] = 0;
```

[2)] В рассматриваемой программе для снижения сложности программного кода размер полос для всех процессов, кроме последнего, устанавливается равным.

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

5. Функция ResultReplication. Объединяет блоки результирующего вектора *b*, полученные на разных процессах, и копирует вектор результата на все процессы вычислительной системы.

```
// Function for gathering the result vector
void ResultReplication(double* pProcResult, double* pResult, int Size,
    int RowNum) {
  int i;             // Loop variable
  int *pReceiveNum;  // Number of elements, that current process sends
  int *pReceiveInd;  /* Index of the first element from current process
                        in result vector */
  int RestRows=Size; // Number of rows, that haven't been distributed yet

  //Alloc memory for temporary objects
  pReceiveNum = new int [ProcNum];
  pReceiveInd = new int [ProcNum];

  //Define the disposition of the result vector block of current processor
  pReceiveInd[0] = 0;
  pReceiveNum[0] = Size/ProcNum;
  for (i=1; i<ProcNum; i++) {
    RestRows -= pReceiveNum[i-1];
    pReceiveNum[i] = RestRows/(ProcNum-i);
    pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1];
  }
  //Gather the whole result vector on every processor
  MPI_Allgatherv(pProcResult, pReceiveNum[ProcRank], MPI_DOUBLE, pResult,
    pReceiveNum, pReceiveInd, MPI_DOUBLE, MPI_COMM_WORLD);

  //Free the memory
  delete [] pReceiveNum;
  delete [] pReceiveInd;
}
```

7.6.5. Результаты вычислительных экспериментов

Рассмотрим результаты вычислительных экспериментов, выполненных для оценки эффективности рассмотренного параллельного алгоритма умножения матрицы на вектор. Кроме того, используем полученные результаты для сравнения теоретических оценок и экспериментальных показателей времени вычислений и проверим тем самым точность полученных аналитических соотношений. Эксперименты проводились на вычислительном кластере на базе процессоров Intel XEON 4 EM64T, 3000 Мгц и сети Gigabit Ethernet под управлением операционной системы Microsoft Windows Server 2003 Standart x64 Edition (см. п. 1.2.3).

Определение параметров теоретических зависимостей (величин τ, w, α, β) осуществлялось следующим образом. Для оценки длительности τ базовой скалярной операции проводилось решение задачи умножения матрицы на вектор при помощи последовательного алгоритма и полученное таким образом время вычислений делилось на общее количество выполненных операций – в результате подобных экспериментов для величины τ было получено значение 1.93 нсек. Эксперименты, выполненные для определения параметров сети передачи данных, показали значения латентности α и пропускной способности β соответственно 47 мкс и 53.29 Мбайт/с. Все вычисления производились над числовыми значениями типа double, т. е. величина *w* равна 8 байт.

Результаты вычислительных экспериментов приведены в таблице 7.1. Эксперименты проводились с использованием двух, четырех и восьми процессоров. Времена выполнения алгоритмов указаны в секундах.

Таблица 7.1. Результаты вычислительных экспериментов для параллельного алгоритма умножения матрицы на вектор при ленточной схеме разделении данных по строкам

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
		Время	Ускорение	Время	Ускорение	Время	Ускорение

1000	0,0041	0,0021	1,8798	0,0017	2,4089	0,0175	0,2333
2000	0,016	0,0084	1,8843	0,0047	3,3388	0,0032	4,9443
3000	0,031	0,0185	1,6700	0,0097	3,1778	0,0059	5,1952
4000	0,062	0,0381	1,6263	0,0188	3,2838	0,0244	2,5329
5000	0,11	0,0574	1,9156	0,0314	3,4993	0,0150	7,3216

Сравнение экспериментального времени T_p^* выполнения параллельного алгоритма и теоретического времени T_p, вычисленного в соответствии с выражением (7.8), представлено в таблице 7.2 и в графическом виде на рис. 7.3 и 7.4.

Таблица 7.2. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор, основанного на разбиении матрицы по строкам

Размер объектов	2 процессора		4 процессора		8 процессоров	
	T_p	T_p^*	T_p	T_p^*	T_p	T_p^*
1000	0,0069	0,0021	0,0108	0,0017	0,0152	0,0175
2000	0,0132	0,0084	0,0140	0,0047	0,0169	0,0032
3000	0,0235	0,0185	0,0193	0,0097	0,0196	0,0059
4000	0,0379	0,0381	0,0265	0,0188	0,0233	0,0244
5000	0,0565	0,0574	0,0359	0,0314	0,0280	0,0150

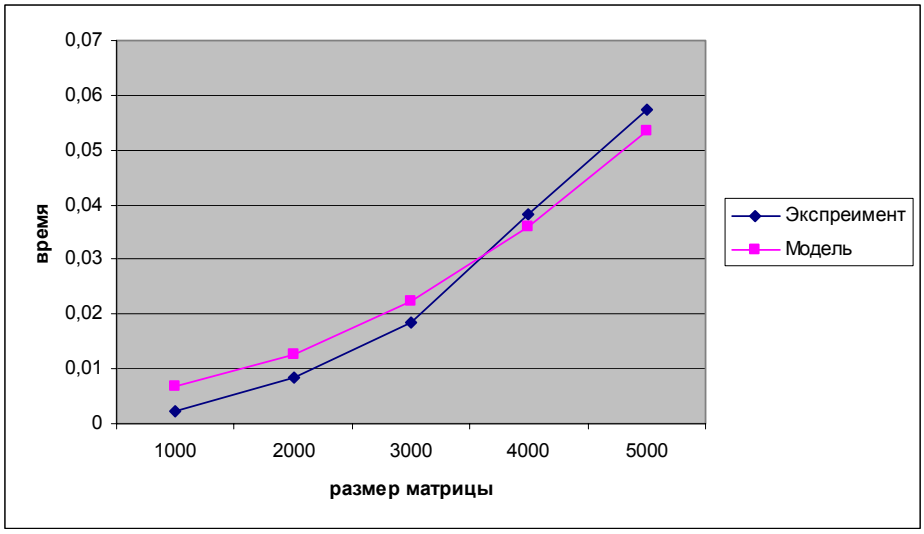

Рис. 7.3. График зависимости экспериментального T_p^* и теоретического T_p времени выполнения параллельного алгоритма на двух процессорах от объема исходных данных (ленточное разбиение матрицы по строкам)

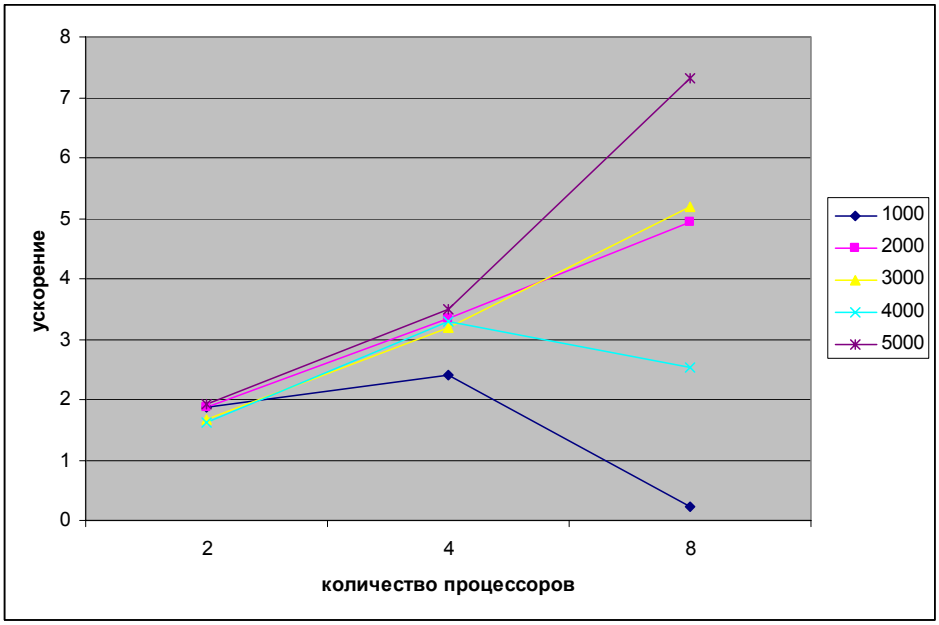

Рис. 7.4. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма умножения матрицы на вектор (ленточное разбиение по строкам) для разных размеров матриц

7.7. 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.

7.7.1. 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 содержать i-й столбец матрицы A и i-е элементы b_i и c_i векторов b и c.

Параллельный алгоритм умножения матрицы на вектор начинается с того, что 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. Далее для получения элементов результирующего вектора c подзадачи должны обменяться своими промежуточными данными между собой (элемент j, $0 \le j < n$, частичного результата $c'(i)$ подзадачи i, $0 \le i < n$, должен быть передан подзадаче j). Данная обобщенная передача данных (*all-to-all communication* или *total exchange*) является наиболее общей коммуникационной процедурой и может быть реализована при помощи функции *MPI_Alltoall* библиотеки *MPI*. После выполнения передачи данных каждая базовая подзадача i, $0 \le i < n$, будет содержать n частичных значений $c'_i(j)$, $0 \le j < n$,, после сложения которых и определяется элемент c_i вектора результата c (см. рис. 7.5).

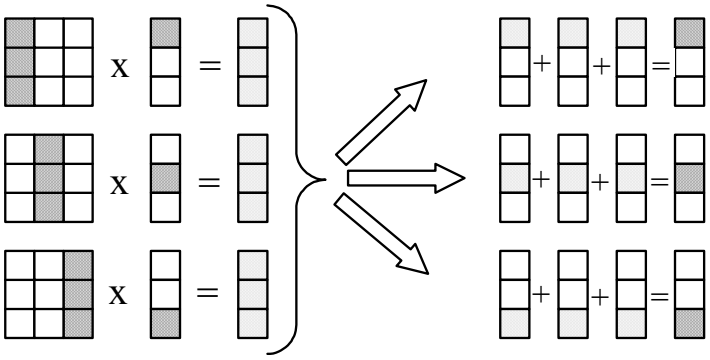

Рис. 7.5. Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор с использованием разбиения матрицы по столбцам

7.7.2. Масштабирование и распределение подзадач по процессорам

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.

Как и в предыдущем алгоритме, распределение подзадач между процессорами вычислительной системы может быть выполнено произвольным образом.

7.7.3. Анализ эффективности

Пусть, как и ранее, матрица A является квадратной, то есть $m=n$. На первом этапе вычислений каждый процессор умножает принадлежащие ему столбцы матрицы A на элементы вектора b , после умножения полученные значения суммируются для каждой строки матрицы A в отдельности

$$c'_s(i) = \sum_{j=j_0}^{j_{l-1}} a_{sj} b_j,\ 0 \le s < n, \tag{7.9}$$

(j_0 и j_{l-1} есть начальный и конечный индексы столбцов базовой подзадачи i, $0 \le i < n$). Поскольку размеры полосы матрицы A и блока вектора b равны n/p, то трудоемкость таких вычислений может оцениваться как $T' = n^2 / p$ операций. После обмена данными между подзадачами на втором этапе вычислений каждый процессор суммирует полученные значения для своего блока результирующего вектора c. Количество суммируемых значений для каждого элемента c_i вектора c совпадает с числом процессоров p, размер блока вектора c на одном процессоре равен n/p и, тем самым, число выполняемых операций для второго этапа оказывается равным $T'' = n$. С учетом полученных соотношений показатели ускорения и эффективности параллельного алгоритма могут быть выражены следующим образом:

$$S_p = \frac{n^2}{n^2/p} = p\ ,\quad E_p = \frac{n^2}{p \cdot (n^2/p)} = 1\ . \tag{7.10}$$

Теперь рассмотрим более точные соотношения для оценки времени выполнения параллельного алгоритма. С учетом ранее проведенных рассуждений время выполнения вычислительных операций алгоритма может быть оценено при помощи выражения

$$T_p(calc) = [n \cdot (2 \cdot \lceil n/p \rceil - 1) + n] \cdot \tau\ . \tag{7.11}$$

(здесь, как и ранее, τ есть время выполнения одной элементарной скалярной операции).

Для выполнения операции обобщенной передачи данных рассмотрим два возможных способа реализации (см. также раздел 3). Первый способ обеспечивается алгоритмом, согласно которому каждый процессор последовательно передает свои данные всем остальным процессорам вычислительном системы. Предположим, что процессоры могут одновременно отправлять и принимать сообщения и между любой парой процессоров имеется прямая линия связи, тогда оценка трудоемкости (время исполнения) такого алгоритма обобщенной передачи данных может быть определена как

$$T_p^1(comm) = (p-1)(\alpha + w\lceil n/p \rceil / \beta). \quad (7.12)$$

(напомним, что α – латентность сети передачи данных, β – пропускная способность сети, *w* – размер элемента данных в байтах)

Второй способ выполнения операции обмена данными рассмотрен в разделе 3, когда топология вычислительной сети может быть представлена в виде гиперкуба. Как было показано, выполнение такого алгоритма может быть осуществлено за $\lceil \log_2 p \rceil$ шагов, на каждом из которых каждый процессор передает и получает сообщение из *n/2* элементов. Как результат, время операции передачи данных при таком подходе составляет величину:

$$T_p^2(comm) = \lceil \log_2 p \rceil (\alpha \cdot + w(n/2)/\beta). \quad (7.13)$$

С учетом полученных соотношений общее время выполнения параллельного алгоритма умножения матрицы на вектор при разбиении данных по столбцам выражается следующим соотношениями:

- Для первого способа выполнения операции передачи данных

$$T_p^1 = [n \cdot (2 \cdot \lceil n/p \rceil - 1) + n] \cdot \tau + (p-1)(\alpha + w\lceil n/p \rceil / \beta). \quad (7.14)$$

- Для второго способа выполнения операции передачи данных

$$T_p^2 = [n \cdot (2 \cdot \lceil n/p \rceil - 1) + n] \cdot \tau + \lceil \log_2 p \rceil (\alpha \cdot + w(n/2)/\beta). \quad (7.15)$$

7.7.4. 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. 7.6.5. Результаты вычислительных экспериментов приведены в таблице 7.3.

Таблица 7.3. Результаты вычислительных экспериментов по исследованию параллельного алгоритма умножения матрицы на вектор, основанного на разбиении матрицы по столбцам

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
		Время	Ускорение	Время	Ускорение	Время	Ускорение
1000	0,0041	0,0022	1,8352	0,0132	0,3100	0,0008	4,9409
2000	0,016	0,0085	1,8799	0,0046	3,4246	0,0029	5,4682
3000	0,031	0,019	1,6315	0,0095	3,2413	0,0055	5,5456
4000	0,062	0,0331	1,8679	0,0168	3,6714	0,0090	6,8599
5000	0,11	0,0518	2,1228	0,0265	4,1361	0,0136	8,0580

Сравнение экспериментального времени T_p^* выполнения эксперимента и времени T_p, вычисленного по соотношениям (7.14), (7.15), представлено в таблице 7.4 и на рис. 7.6 и 7.7. Теоретическое время T_p^1 вычисляется согласно (7.14), а теоретическое время T_p^2 – в соответствии с (7.15).

Таблица 7.4. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор, основанного на разбиении матрицы по столбцам.

Размер матрицы	2 процессора			4 процессора			8 процессоров		
	T_p^1	T_p^2	T_p^*	T_p^1	T_p^2	T_p^*	T_p^1	T_p^2	T_p^*
1000	0,0021	0,0021	0,0022	0,0014	0,0013	0,0013	0,0015	0,0011	0,0008
2000	0,0080	0,0080	0,0085	0,0044	0,0044	0,0046	0,0031	0,0027	0,0029
3000	0,0177	0,0177	0,019	0,0094	0,0094	0,0095	0,0056	0,0054	0,0055
4000	0,0313	0,0313	0,0331	0,0162	0,0163	0,0168	0,0091	0,0090	0,0090
5000	0,0487	0,0487	0,0518	0,0251	0,0251	0,0265	0,0136	0,0135	0,0136

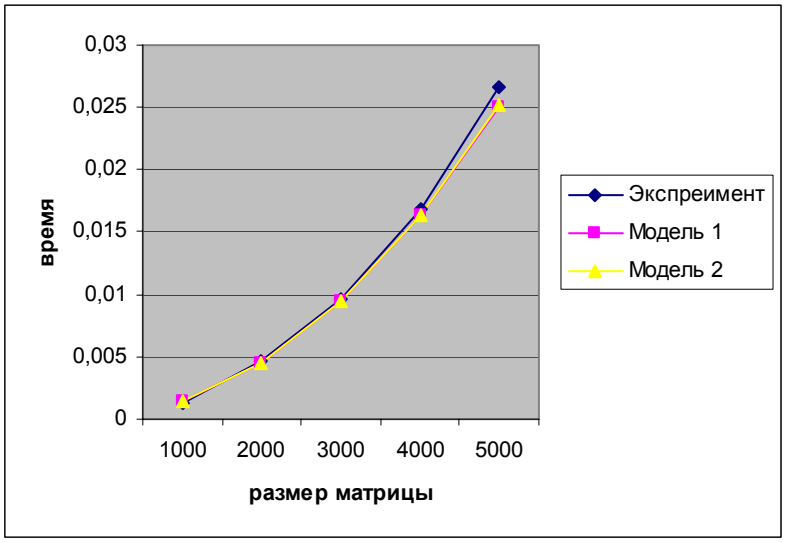

Рис. 7.6. График зависимости теоретического и экспериментального времени выполнения параллельного алгоритма на четырех процессорах от объема исходных данных (ленточное разбиение матрицы по столбцам)

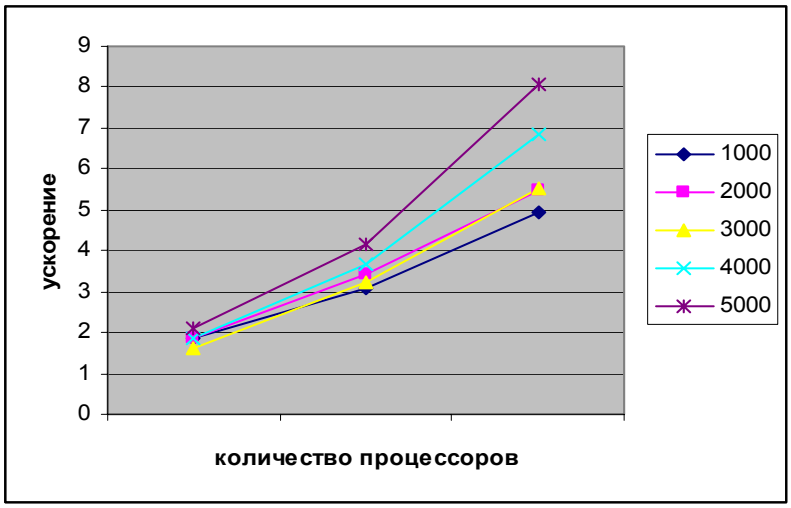

Рис. 7.7. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма умножения матрицы на вектор (ленточное разбиение матрицы по столбцам) для разных размеров матриц

7.8. 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*).

7.8.1. 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:

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

(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)* содержит блок A_{ij}. Помимо блока матрицы A каждая подзадача должна содержать также и блок вектора b. При этом для блоков одной и той 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 $b'_u(i,j) = b_{j_u}$, $j_u = jl + u, 0 \le u < l, l = n/q$.

7.8.2. 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

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

Поэлементное суммирование векторов частичных результатов для каждой горизонтальной полосы (данная процедура часто именуется как *операция редукции* – см. раздел 3) блоков матрицы A позволяет получить результирующий вектор c

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

Для размещения вектора c применим ту же схему, что и для исходного вектора b: организуем вычисления таким образом, чтобы при завершении расчетов вектор c располагался поблочно в каждой из вертикальных полос блоков матрицы A (тем самым, каждый блок вектора c должен быть продублирован по каждой горизонтальной полосе). Выполнение всех необходимых действий для этого – суммирование частичных результатов и дублирование блоков результирующего вектора - может быть обеспечено при помощи функции *MPI_Allreduce* библиотеки *MPI*.

Общая схема выполняемых вычислений для умножения матрицы на вектор при блочном разделении данных показана на рис. 7.8.

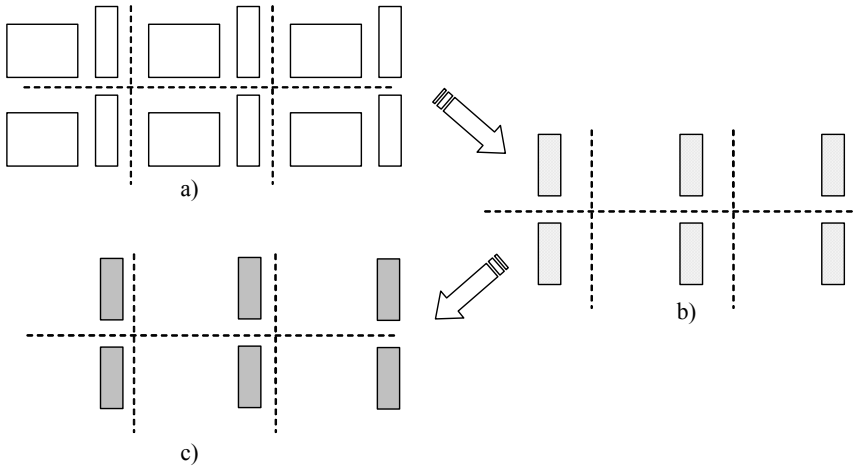

Рис. 7.8. Общая схема параллельного алгоритма умножения матрицы на вектор при блочном разделении данных: a) исходное распределение результатов, b) распределение векторов частичных результатов, c) распределение блоков результирующего вектора c

Рассмотрев представленную схему параллельных вычислений, можно сделать вывод, что информационная зависимость базовых подзадач проявляется только на этапе суммирования результатов перемножения блоков матрицы *A* и блоков вектора *b*. Выполнение таких расчетов может быть выполнено по обычной каскадной схеме (см. раздел 6) и, как результат, характер имеющихся информационных связей для подзадач одной и той же горизонтальной полосы блоков соответствует структуре двоичного дерева.

7.8.3. Масштабирование и распределение подзадач по процессорам

Размер блоков матрицы *A* может быть подобран таким образом, чтобы общее количество базовых подзадач совпадало с числом процессоров *p*. Так, например, если определить размер блочной решетки как $p=s \cdot q$, то

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

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 процессорами.

Возможность выбора остается при определении размеров блочной структуры матрицы *A*. Большое количество блоков по горизонтали приводит к возрастанию числа итераций в операции редукции результатов блочного умножения, а увеличение размера блочной решетки по вертикали повышает объем передаваемых данных между процессорами. Простое, частое используемое решение, состоит в использовании одинакового количества блоков по вертикали и горизонтали, т.е.

$$s = q = \sqrt{p}\ .$$

Следует отметить, что блочная схема разделения данных является обобщением всех рассмотренных в данном разделе подходов. Действительно, при $q=1$ блоки сводятся к горизонтальным полосам матрицы *A*, при $s=1$ исходные данные разбиваются на вертикальные полосы.

При решении вопроса распределения подзадач между процессорами должна учитываться возможность эффективного выполнения операции редукции. Возможный вариант подходящего способа распределения состоит в выделении для подзадач одной и той же горизонтальной полосы блоков множества процессоров, структура сети передачи данных между которыми имеет вид гиперкуба или полного графа.

7.8.4. Анализ эффективности

Выполним анализ эффективности параллельного алгоритма умножения матрицы на вектор при обычных уже предположениях, что матрица *A* является квадратной, т.е. $m=n$. Будем предполагать также, что процессоры, составляющие многопроцессорную вычислительную систему, образуют прямоугольную решетку $p=s \times q$ (*s* – количество строк в процессорной решетке, *q* – количество столбцов).

Общий анализ эффективности приводит к идеальным показателям параллельного алгоритма:

$$S_p = \frac{n^2}{n^2/p} = p\ ,\quad E_p = \frac{n^2}{p \cdot (n^2/p)} = 1. \tag{7.15}$$

Для уточнения полученных соотношений оценим более точно количество вычислительных операций алгоритма и учтем затраты на выполнение операций передачи данных между процессорами.

Общее время умножения блоков матрицы *A* и вектора *b* может быть определено как

$$T_p(calc) = \lceil n/s \rceil \cdot (2 \cdot \lceil n/q \rceil - 1) \cdot \tau\ . \tag{7.16}$$

Операция редукции данных может быть выполнена с использованием каскадной схемы и включает, тем самым, $log_2 q$ итераций передачи сообщений размера $w\lceil n/s \rceil$. Как результат, оценка коммуникационных затрат параллельного алгоритма при использовании модели Хокни может быть определена при помощи следующего выражения

$$T_p(comm) = (\alpha + w \lceil n/s \rceil / \beta) \lceil \log_2 q \rceil. \tag{7.17}$$

Таким образом, общее время выполнения параллельного алгоритма умножения матрицы на вектор при блочном разделении данных составляет

$$T_p = \lceil n/s \rceil \cdot (2 \cdot \lceil n/q \rceil - 1) \cdot \tau + (\alpha + w \lceil n/s \rceil / \beta) \lceil \log_2 q \rceil. \tag{7.18}$$

7.8.5. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма проводились при тех же условиях, что и ранее выполненные расчеты (см. п. 7.6.5). Результаты экспериментов приведены в таблице 7.5. Вычисления проводились с использованием двух, четырех и шести процессоров.

Таблица 7.5. Результаты вычислительных экспериментов по исследованию параллельного алгоритма умножения матрицы на вектор при блочном разделении данных

Размер матриц	Последовательный алгоритм	Параллельный алгоритм			
		4 процессора		9 процессоров	
		Время	Ускорение	Время	Ускорение
1000	0,0041	0,0028	1,4260	0,0011	3,7998
2000	0,016	0,0099	1,6127	0,0042	3,7514
3000	0,031	0,0214	1,4441	0,0095	3,2614
4000	0,062	0,0381	1,6254	0,0175	3,5420
5000	0,11	0,0583	1,8860	0,0263	4,1755

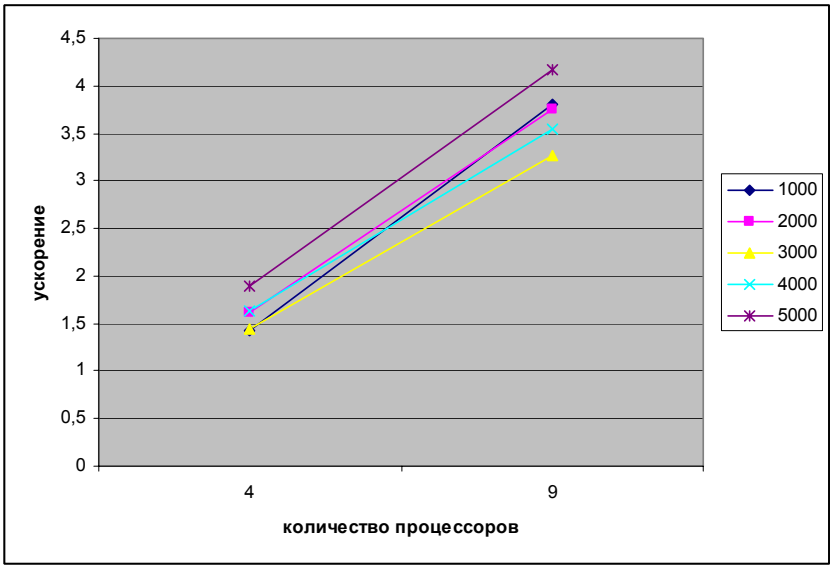

Рис. 7.9. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма умножения матрицы на вектор (блочное разбиение матрицы) для разных размеров матриц

Сравнение экспериментального времени T_p^* выполнения эксперимента и теоретического времени T_p, вычисленного в соответствии с выражением (7.18), представлено в таблице 7.6 и на рис. 7.10.

Таблица 7.6. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор при блочном разделении данных

Размер матриц	4 процессора		9 процессоров	
	T_p	T_p^*	T_p	T_p^*
1000	0,0025	0,0028	0,0012	0,0010
2000	0,0095	0,0099	0,0043	0,0042
3000	0,0212	0,0214	0,0095	0,0095
4000	0,0376	0,0381	0,0168	0,0175
5000	0,0586	0,0583	0,0262	0,0263

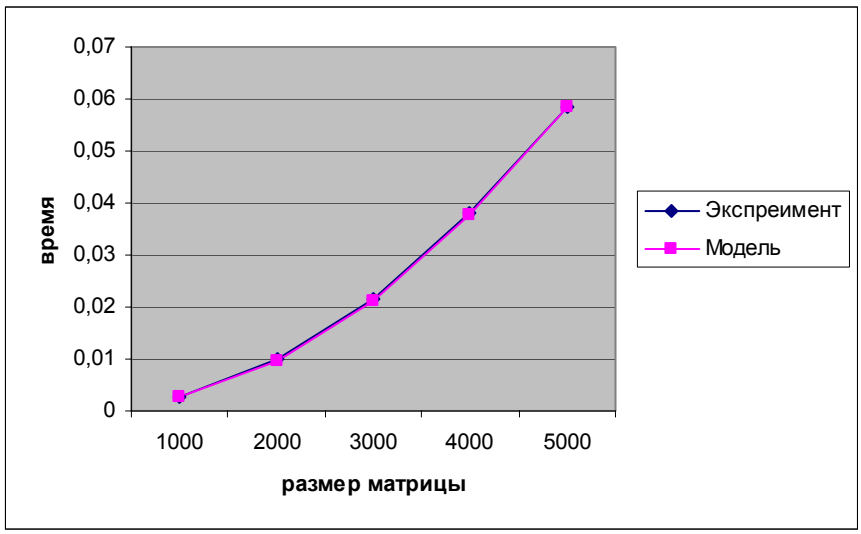

Рис. 7.10. График зависимости экспериментального и теоретического времени проведения эксперимента на четырех процессорах от объема исходных данных (блочное разбиение матрицы)

7.9. Краткий обзор раздела

В данном разделе на примере задачи умножения матрицы на вектор рассматриваются возможные схемы разделения матриц между процессорами многопроцессорной вычислительной системы, которые могут быть использованы для организации параллельных вычислений при выполнении 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 процессорами 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 процессорами. В завершение для каждого алгоритма проводится анализ эффективности получаемых параллельных вычислений и приводятся результаты вычислительных экспериментов. Для алгоритма умножения матрицы на вектор при ленточном разделении данных по строкам приводится возможный вариант программной реализации.

Полученные показатели эффективности показывают, что все используемые способы разделения данных приводят к равномерной балансировке вычислительной нагрузки, и отличия возникают только в трудоемкости выполняемых информационных взаимодействий между процессорами. В этом отношении интересным представляется проследить, как выбор способа разделения данных влияет на характер необходимых операций передачи данных и выделить основные различия в коммуникационных действиях разных алгоритмов. Кроме того, важным является определение целесообразной структуры линий связи между процессорами для эффективного выполнения соответствующего параллельного алгоритма. Так, например, алгоритмы, основанные на ленточном разделении данных, ориентированы на топологию сети в виде гиперкуба или полного графа. Для реализации алгоритма, основанного на блочном разделении данных, необходимо наличие топологии решетки.

На рис. 7.11 на общем графике представлены показатели ускорения, полученные в результате выполнения вычислительных экспериментов для всех рассмотренных алгоритмов. Как можно заметить, при использовании шести процессоров некоторое преимущество по ускорению имеет параллельный алгоритм умножения матрицы на вектор при ленточном разделении по строкам. Выполненные расчеты показывают также, что при большем количестве процессоров более эффективным становится блочный алгоритм умножения.

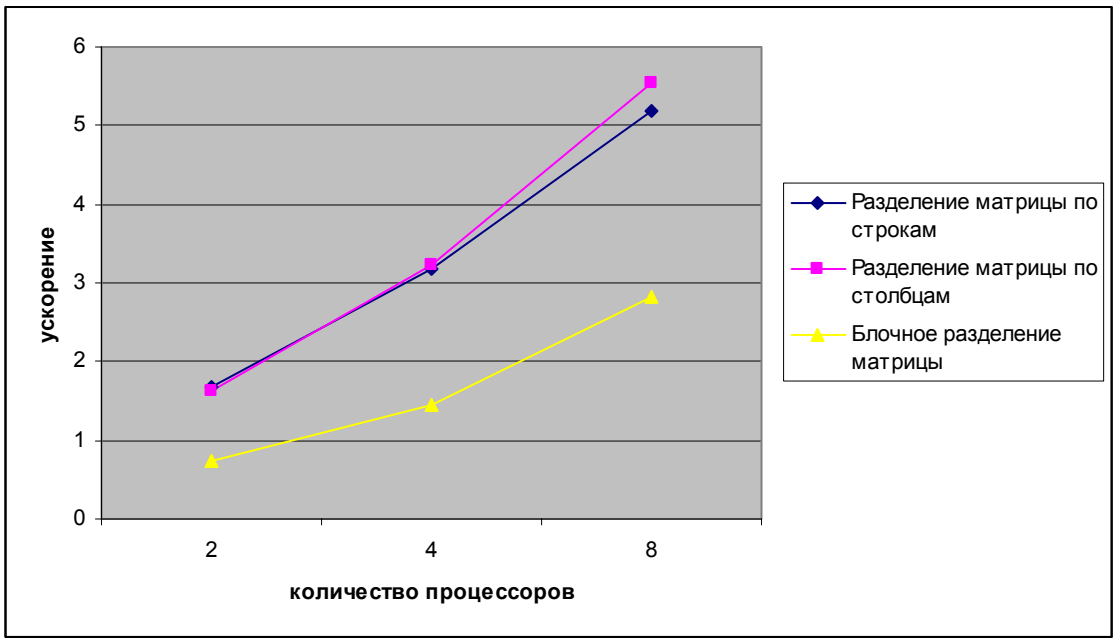

Рис. 7.11. Показатели ускорения рассмотренных параллельных алгоритмов умножения по результатам вычислительных экспериментов с матрицами размера 2000×2000 и векторами из 2000 элементов

7.10. 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).

При рассмотрении вопросов программной реализации параллельных методов может быть рекомендована работа Blackford, et al. (1997). В данной работе рассматривается хорошо известная и широко используемая в практике параллельных вычислений программная библиотека численных методов ScaLAPACK.

7.11. Контрольные вопросы

1. Назовите основные способы распределения элементов матрицы между процессорами вычислительной системы.
2. В чем состоит постановка задачи умножения матрицы на вектор?
3. Какова вычислительная сложность последовательного алгоритма умножения матрицы на вектор?
4. Почему при разработке параллельных алгоритмов умножения матрицы на вектор допустимо дублировать вектор-операнд на все процессоры?
5. Какие подходы могут быть предложены для разработки параллельных алгоритмов умножения матрицы на вектор?
6. Представьте общие схемы рассмотренных параллельных алгоритмов умножения матрицы на вектор.
7. Проведите анализ и получите показатели эффективности для одного из рассмотренных алгоритмов.
8. Какой из представленных алгоритмов умножения матрицы на вектор обладает лучшими показателями ускорения и эффективности?
9. Может ли использование циклической схемы разделения данных повлиять на время работы каждого из представленных алгоритмов?
10. Какие информационные взаимодействия выполняются для алгоритмов при ленточной схеме разделения данных? В чем различие необходимых операций передачи данным при разделении матрицы по строкам и столбцам?
11. Какие информационные взаимодействия выполняются для блочного алгоритма умножения матрицы на вектор?

12. Какая топология коммуникационной сети является целесообразной для каждого из рассмотренных алгоритмов?
13. Дайте общую характеристику программной реализации алгоритма умножения матрицы на вектор при разделении данных по строкам. В чем могут состоять различия в программной реализации других рассмотренных алгоритмов?
14. Какие функции библиотеки MPI оказались необходимыми при программной реализации алгоритмов?

7.12. Задачи и упражнения

1. Выполните реализацию параллельного алгоритма, основанного на ленточном разбиении матрицы на вертикальные полосы. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты реальных экспериментов с ранее подготовленными теоретическими оценками.

2. Выполните реализацию параллельного алгоритма, основанного на разбиении матрицы на блоки. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты реальных экспериментов с ранее подготовленными теоретическими оценками.

Литература

Dongarra, J.J., Duff, L.S., Sorensen, D.C., Vorst, H.A.V. (1999). Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools). Soc for Industrial & Applied Math/

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J. J., Hammarling, S., Henry, G., Petitet, A., Stanley, D. Walker, R.C. Whaley, K. (1997). Scalapack Users' Guide (Software, Environments, Tools). Soc for Industrial & Applied Math.

Foster, I. (1995). Designing and Building Parallel Programs: Concepts and Tools for Software Engineering. Reading, MA: Addison-Wesley.