ГЛАВА 8
ПАРАЛЛЕЛЬНЫЕ МЕТОДЫ РЕШЕНИЯ СИСТЕМ ЛИНЕЙНЫХ УРАВНЕНИЙ

Системы линейных уравнений возникают при решении ряда прикладных задач, описываемых дифференциальными, интегральными или системами нелинейных (трансцендентных) уравнений. Они могут появляться также в задачах математического программирования, статистической обработки данных, аппроксимации функций, при дискретизации краевых дифференциальных задач методом конечных разностей или методом конечных элементов и др.

Матрицы коэффициентов систем линейных уравнений могут иметь различную структуру и свойства. Матрицы решаемых систем могут быть плотными и их порядок может достигать несколько тысяч строк и столбцов. При решении многих задач могут появляться системы, обладающие симметричными положительно определёнными ленточными матрицами с порядком в десятки тысяч и шириной ленты в несколько тысяч элементов. И, наконец, при рассмотрении большого ряда задач могут возникать системы линейных уравнений с разрежёнными матрицами с порядком в миллионы строк и столбцов.

8.1. Постановка задачи

Линейное уравнение с n неизвестными x_0, x_1, …, x_{n-1} может быть определено при помощи выражения

$$a_0 x_0 + a_1 x_1 + \ldots + a_{n-1} x_{n-1} = b \qquad (8.1)$$

где величины a_0, a_1, …, a_{n-1} и b представляют собой постоянные значения.

Множество n линейных уравнений

$$\begin{aligned} a_{0,0}x_0 \quad &+a_{0,1}x_1 \quad &+\ldots+a_{0,n-1}x_{n-1} \quad &= b_0 \\ a_{1,0}x_0 \quad &+a_{1,1}x_1 \quad &+\ldots+a_{1,n-1}x_{n-1} \quad &= b_1 \\ &\ldots & & \\ a_{n-1,0}x_0 \quad &+a_{n-1,1}x_1 \quad &+\ldots+a_{n-1,n-1}x_{n-1} \quad &= b_{n-1} \end{aligned} \qquad (8.2)$$

называется *системой линейных уравнений* или *линейной системой*. В более кратком (*матричном*) виде система может представлена как

$$Ax = b,$$

где $A = (a_{i,j})$ есть вещественная матрица размера $n \times n$, а векторы b и x состоят из n элементов.

Под *задачей решения системы линейных уравнений* для заданных матрицы A и вектора b обычно понимается нахождение значения вектора неизвестных x, при котором выполняются все уравнения системы.

8.2. Метод Гаусса

Метод Гаусса является широко известным *прямым* алгоритмом решения систем линейных уравнений, для которых матрицы коэффициентов являются *плотными*. Если система линейных уравнений является *невырожденной*, то метод Гаусса гарантирует нахождение решения с погрешностью, определяемой точностью машинных вычислений. Основная идея метода состоит в приведении матрицы A посредством эквивалентных преобразований (не меняющих решение системы (8.2)) к треугольному виду, после чего значения искомых неизвестных могут быть получены непосредственно в явном виде.

В разделе дается общая характеристика метода Гаусса, достаточная для начального понимания алгоритма и позволяющая рассмотреть возможные способы параллельных вычислений при решении систем линейных уравнений. Более полное изложение алгоритма со строгим обсуждением вопросов точности получаемых решений может быть получено в работах [2–4,44,68] и др.

8.2.1. Общая схема метода

Метод Гаусса основывается на возможности выполнения преобразований линейных уравнений, которые не меняют при этом решение рассматриваемой системы (такие преобразования носят наименование *эквивалентных*). К числу таких преобразований относятся:

- умножение любого из уравнений на ненулевую константу,
- перестановка уравнений,
- прибавление к уравнению любого другого уравнения системы.

Метод Гаусса включает последовательное выполнение двух этапов. На первом этапе – *прямой ход* метода Гаусса – исходная система линейный уравнений при помощи последовательного исключения неизвестных приводится к верхнему треугольному виду

$$U x = c ,$$

где матрица коэффициентов получаемой системы имеет вид

$$U = \begin{pmatrix} u_{0,0} & u_{0,1} & \dots & u_{0,n-1} \\ 0 & u_{1,1} & \dots & u_{1,n-1} \\ & & \dots & \\ 0 & 0 & \dots & u_{n-1,n-1} \end{pmatrix}. \qquad (8.3)$$

На *обратном ходе* метода Гаусса (второй этап алгоритма) осуществляется определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной x_{n-1}, после этого из предпоследнего уравнения становится возможным определение переменной x_{n-2} и т. д.

8.2.2. Прямой ход метода Гаусса

Прямой ход метода Гаусса состоит в последовательном исключении неизвестных в уравнениях решаемой системы линейных уравнений. На итерации i, $0 \le i < n-1$, метода производится исключение неизвестной i для всех уравнений с номерами k, больших i (т. е. $i < k \le n-1$). Для этого из этих уравнений осуществляется вычитание строки i, умноженной на константу (a_{ki}/a_{ii}) с тем, чтобы результирующий коэффициент при неизвестной x_i в строках оказался нулевым – все необходимые вычисления могут быть определены при помощи соотношений:

$$\begin{aligned} a'_{kj} &= a_{kj} - (a_{ki} / a_{ii}) \cdot a_{ij}, \\ b'_k &= b_k - (a_{ki} / a_{ii}) \cdot b_i, \end{aligned} \quad i \le j \le n-1, i < k \le n-1, 0 \le i < n-1$$

(следует отметить, что аналогичные вычисления выполняются и над вектором b).

Поясним выполнение прямого хода метода Гаусса на примере системы линейных уравнений вида:

$$\begin{aligned} x_0 &+3x_1 &+2x_2 &= 1 \\ 2x_0 &+7x_1 &+5x_2 &= 18 \\ x_0 &+4x_1 &+6x_2 &= 26 \end{aligned}.$$

На первой итерации производится исключение неизвестной x_0 из второй и третьей строки. Для этого из этих строк нужно вычесть первую строку, умноженную соответственно на 2 и 1. После этих преобразований система уравнений принимает вид:

$$\begin{aligned} x_0 \quad +3x_1 \quad +2x_2 \quad &= \quad 1 \\ x_1 \quad + \; x_2 \quad &= \quad 16. \\ x_1 \quad +4x_2 \quad &= \quad 25 \end{aligned}$$

В результате остается выполнить последнюю итерацию и исключить неизвестную x_1 из третьего уравнения. Для этого необходимо вычесть вторую строку и в окончательной форме система имеет следующий вид:

$$\begin{aligned} x_0 \quad +3x_1 \quad +2x_2 \quad &= \quad 1 \\ x_1 \quad + x_2 \quad &= \quad 16. \\ 3x_2 \quad &= \quad 9 \end{aligned}$$

На рис. 8.1 представлена общая схема состояния данных на i-й итерации прямого хода алгоритма Гаусса. Все коэффициенты при неизвестных, расположенные ниже главной диагонали и левее столбца i, уже являются нулевыми. На i-й итерации прямого хода метода Гаусса осуществляется обнуление коэффициентов столбца i, расположенных ниже главной диагонали, путем вычитания строки i, умноженной на нужную ненулевую константу. После проведения (n–1) подобных итераций матрица, определяющая систему линейных уравнений, становится приведенной к верхнему треугольному виду.

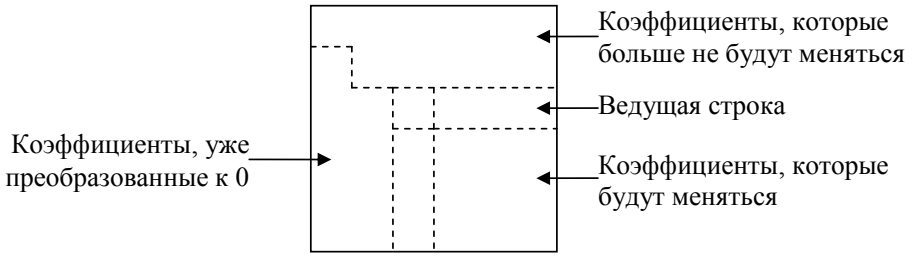

Рис. 8.1. Итерация прямого хода алгоритма Гаусса

При выполнении прямого хода метода Гаусса строка, которая используется для исключения неизвестных, носит наименование *ведущей*, а диагональный элемент ведущей строки называется *ведущим элементом*. Как можно заметить, выполнение вычислений является возможным только в случае, если ведущий элемент имеет ненулевое значение. Более того, если ведущий элемент a_{ii} имеет малое значение, то деление и умножение строк на этот элемент может приводить к накоплению вычислительной погрешности и вычислительной неустойчивости алгоритма.

Возможный способ избежать подобной проблемы может состоять в следующем: при выполнении каждой очередной итерации прямого хода метода Гаусса следует определить коэффициент с максимальным значением по абсолютной величине в столбце, соответствующем исключаемой неизвестной, т. е.

$$y = \max_{i \le k \le n-1} |a_{ki}|,$$

и выбрать в качестве ведущей строку, в которой этот коэффициент располагается (данная схема выбора ведущего значения носит наименование *метода главных элементов*).

Вычислительная сложность прямого хода алгоритма Гаусса с выбором ведущей строки имеет порядок $O(n^3)$.

8.2.3. Обратный ход метода Гаусса

После приведения матрицы коэффициентов к верхнему треугольному виду становится возможным определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной x_{n-1}, после этого из предпоследнего уравнения становится возможным определение переменной x_{n-2} и т. д. В общем виде, выполняемые вычисления при обратном ходе метода Гаусса могут быть представлены при помощи соотношений:

$$\begin{aligned} &x_{n-1} = b_{n-1} / a_{n-1,n-1}, \\ &x_i = (b_i - \sum_{j=i+1}^{n-1} a_{ij} x_j) / a_{ii}, \;\; i = n-2, n-3, ..., 0 \end{aligned} \tag{8.4}$$

Поясним, как и ранее, выполнение обратного хода метода Гаусса на примере рассмотренной в п. 8.2.2 системы линейных уравнений

$$\begin{aligned} x_0 \quad +3x_1 \quad +2x_2 \quad &= \quad 1 \\ x_1 \quad + x_2 \quad &= \quad 16. \\ 3x_2 \quad &= \quad 9 \end{aligned}$$

Из последнего уравнения системы можно определить, что неизвестная x_2 имеет значение 3. В результате становится возможным разрешение второго уравнения и определение значение неизвестной $x_1 = 13$, т. е.

$$\begin{aligned} x_0 \quad +3x_1 \quad +2x_2 \quad &= \quad 1 \\ x_1 \quad &= \quad 13. \\ x_2 \quad &= \quad 3 \end{aligned}$$

На последней итерации обратного хода метода Гаусса определяется значение неизвестной x_0, равное –44.

С учетом последующего параллельного выполнения можно отметить, что учет получаемых значений неизвестных может выполняться сразу во всех уравнениях системы (и эти действия могут выполняться в уравнениях одновременно и независимо друг от друга). Так, в рассматриваемом примере после определения значения неизвестной x_2 система уравнений может быть приведена к виду

$$\begin{aligned} x_0 + 3x_1 &= -5 \\ x_1 &= 13 \\ x_2 &= 3 \end{aligned}$$

Вычислительная сложность обратного хода алгоритма Гаусса составляет $O(n^2)$.

8.2.4. Программная реализация

Рассмотрим возможный вариант реализации последовательного алгоритма Гаусса для решения систем линейных уравнений. При этом реализация отдельных модулей не приводится, если их отсутствие не оказывает влияние на понимании общей схемы вычислений.

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

```
// Программа 8.1
// Алгоритм Гаусса решения систем линейных уравнений
// Номера ведущих строк по номеру итераций
int* pPivotPos;
// Номера итераций, на которых строки выбирались
// в качестве ведущих
int* pPivotIter;

void main() {
  double* pMatrix;  // Матрица системы уравнений
  double* pVector;  // Вектор правых частей
  double* pResult;  // Вектор неизвестных
  int Size;         // Количество уравнений

  // Выделение памяти и инициализация данных
  ProcessInitialization(pMatrix, pVector,
    pResult, Size);
```

```
  // Выполнение алгоритма Гаусса
  SerialResultCalculation(pMatrix, pVector,
    pResult, Size);

  // Завершение вычислений
  ProcessTermination(pMatrix, pVector, pResult);
}
```

Следует пояснить использование дополнительных массивов. Элементы массива *pPivotPos* определяют номера строк матрицы, выбираемых в качестве ведущих, по итерациям прямого хода метода Гаусса. Именно в этом порядке должны выполняться затем итерации обратного хода для определения значений неизвестных системы линейных уравнений.

Элементы массива *pPivotIter* определяют номера итераций прямого хода метода Гаусса, на которых строки использовались в качестве ведущих (т. е. строка i выбиралась ведущей на итерации $pPivotIter[i]$). Начальное значение элементов массива устанавливается равным –1 и тем самым значение элемента массива $pPivotIter[i]$, равное –1, является признаком того, что строка i процесса все еще подлежит обработке. Кроме того, важно отметить, что запоминаемые в элементах массива *pPivotIter* номера итераций дополнительно означают и номера неизвестных, для определения которых будут использованы соответствующие строки уравнения.

Функция *ProcessInitialization* определяет исходные данные решаемой задачи (число неизвестных), выделяет память для хранения данных, осуществляет ввод матрицы коэффициентов системы линейных уравнений и вектора правых частей (или формирует эти данные при помощи датчика случайных чисел).

Функция *ProcessTermination* выполняет необходимый вывод результатов решения задачи и освобождает всю ранее выделенную память для хранения данных.

Реализация всех перечисленных функций может быть выполнена по аналогии с ранее рассмотренными примерами и предоставляется читателю в качестве самостоятельного упражнения.

2. Функция SerialResultCalculation. Функция реализует логику работы алгоритма Гаусса, последовательно вызывает функции, выполняющие прямой и обратный ходы метода.

```
// Функция для выполнения метода Гаусса
void SerialResultCalculation (double* pMatrix,
  double* pVector, double* pResult, int Size) {
  // Прямой ход метода Гаусса
  SerialGaussianElimination (pMatrix, pVector, Size);
```

```
  // Обратный ход метода Гаусса
  SerialBackSubstitution (pMatrix, pVector,
    pResult, Size);
}
```

3. Функция SerialGaussianElimination. Функция выполняет прямой ход алгоритма Гаусса.

```
// Функция для выполнения прямого хода метода Гаусса
void SerialGaussianElimination (double* pMatrix,
  double* pVector,int Size) {
  int Iter;       // Номер итерации
  int PivotRow;   // Номер ведущей строки
  for (Iter=0; Iter<Size; Iter++) {
    // Определение ведущей строки
    PivotRow = FindPivotRow(pMatrix, Size, Iter);
    pPivotPos[Iter] = PivotRow;
    pPivotIter[PivotRow] = Iter;
    SerialColumnElimination(pMatrix, pVector,
      Size, Iter, PivotRow);
  }
}
```

4. Функция FindPivotRow. Эта функция определяет строку линейной системы, которую следует использовать в качестве ведущей на данной итерации алгоритма. В качестве ведущей строки выбирается строка, которая ранее не использовалась в качестве ведущей (т. е. для которой элемент массива $pSerialPivotIter$ равен –1) с максимальным по абсолютному значению элементом, расположенным в столбце $Iter$.

```
// Функция для нахождения ведущей строки
int FindPivotRow (double* pMatrix, int Size, int Iter)
{
  int PivotRow = -1;    // Номер ведущей строки
  double MaxValue =  0; // Значение ведущего элемента
  int i;

  // Определение строки с max элементом
  for (i=0; i<Size; i++) {
    if ((pSerialPivotIter[i] == -1) &&
       (fabs(pMatrix[i*Size+Iter]) > MaxValue)) {
      PivotRow = i;
      MaxValue = fabs(pMatrix[i*Size+Iter]);
    }
  }
```

```
  return PivotRow;
}
```

5. Функция SerialColumnElimination. Функция проводит вычитание ведущей строки из строк процесса, которые еще не использовались в качестве ведущих (т. е. для которых элементы массива *pSerialPivotIter* равны –1).

```
// Функция исключения текущей неизвестной
void SerialColumnElimination (double* pMatrix,
  double* pVector, int Size, int Iter, int Pivot) {
  double PivotValue, PivotFactor;
  PivotValue = pMatrix[Pivot*Size+Iter];
  for (int i=0; i<Size; i++) {
    if (pSerialPivotIter[i] == -1) {
      PivotFactor = pMatrix[i*Size+Iter] / PivotValue;
      for (int j=Iter; j<Size; j++) {
        pMatrix[i*Size + j] -= PivotFactor *
          pMatrix[Pivot*Size+j];
      }
      pVector[i] -= PivotFactor * pVector[Pivot];
    }
  }
}
```

6. Функция SerialBackSubstitution. Функция реализует обратный ход метода Гаусса.

```
// Функция для выполнения обратного хода метода Гаусса
void SerialBackSubstitution (double* pMatrix,
  double* pVector, double* pResult, int Size) {
  int RowIndex, Row;
  for (int i=Size-1; i>=0; i--) {
    RowIndex = pPivotPos[i];
    pResult[i] =
      pVector[RowIndex]/pMatrix[RowIndex*Size+i];
    pMatrix[RowIndex*Size +i] = 1;
    for (int j=0; j<i; j++) {
      Row = pPivotPos[j];
      pVector[Row] -= pMatrix[Row*Size+i]*pResult[i];
      pMatrix[Row*Size+i] = 0;
    }
  }
}
```

8.2.5. Анализ эффективности

При анализе эффективности последовательного алгоритма умножения матриц снова используем подход, примененный в п. 6.5.4.

Итак, время выполнения алгоритма складывается из времени, которое тратится непосредственно на вычисления, и времени, необходимого на чтение данных из оперативной памяти в кэш процессора.

Оценим количество вычислительных операций, выполняемых на каждой итерации прямого и обратного ходов алгоритма Гаусса.

На каждой итерации прямого хода метода Гаусса необходимо осуществить выбор ведущей строки среди строк, подлежащих обработке. На i-й итерации прямого хода число строк, подлежащих обработке, равно $(n–i)$, где n – порядок матрицы линейной системы. После выбора ведущей строки производится вычитание этой строки, умноженной на константу, из всех строк, подлежащих обработке. Каждая строка содержит $(n–i)$ ненулевых элементов, число строк – $(n–i)$. Таким образом, количество вычислительных операций, необходимых для вычитания ведущей строки из остальных строк составляет 2 $(n–i)\cdot(n–i)$. Для приведения матрицы к верхнетреугольному виду необходимо выполнить $(n–1)$ итерацию прямого хода метода Гаусса. Следовательно, время выполнения вычислений составляет:

$$T_{calc}^{1} = \sum_{i=0}^{n-2} \left[\ n-i \ + 2 \cdot \ n-i \ ^{2} \right] \cdot \tau .$$

На каждой итерации обратного хода метода Гаусса во всех строках линейной системы, подлежащих обработке, выполняется корректировка элемента вектора правых частей. На i-й итерации обратного хода число строк, подлежащих обработке, равно $(n–i)$. Для вычисления значений всех неизвестных необходимо выполнить $(n–1)$ итерацию обратного хода. Следовательно, количество вычислительных операций может быть получено по формуле:

$$T_{calc}^{2} = \sum_{i=0}^{n-2} 2 \cdot \ n-i \ \cdot \tau .$$

Итак, время, которое тратится непосредственно на вычисления при выполнении последовательного алгоритма Гаусса:

$$T_{calc} = \sum_{i=0}^{n-2} \left[\ n-i \ + 2 \cdot \ n-i \ ^{2} + 2 \ n-i \ \right] \cdot \tau = \frac{4n^3 + 15n^2 + 11n - 30}{6} \tau . \quad (8.5)$$

Теперь необходимо оценить объем данных, которые нужно прочитать из оперативной памяти в кэш вычислительного элемента в случае, когда

размер линейной системы настолько велик, что матрица и векторы, описывающие эту систему, одновременно не могут быть помещены в кэш.

На каждой итерации прямого хода метода Гаусса для выбора ведущей строки необходимо просмотреть элементы, расположенные в i-м столбце у всех строк, подлежащих обработке, и выбрать среди них максимальный по абсолютному значению элемент. Соответственно, все эти элементы необходимо прочитать из оперативной памяти. Количество строк, подлежащих обработке на i-й итерации прямого хода, составляет $(n - i)$. Напомним также, что считывание данных происходит не по одному элементу, а кэш-строками по 64 байта. Следовательно, из оперативной памяти считывается $64\cdot(n - i)$ байт. После выбора ведущей строки для выполнения вычитания необходимо прочитать из оперативной памяти ненулевые элементы строк, подлежащих обработке, и соответствующие элементы вектора правых частей. Количество строк, подлежащих обработке, составляет $(n - i)$, в каждой строке содержится $(n - i)$ ненулевых элементов. Следовательно, объем загружаемых данных $64\cdot(n - i)\,(n - i) + 64\cdot(n - i)$.

Таким образом, время, которое тратится на считывание необходимых данных из оперативной памяти в кэш при выполнении прямого хода метода Гаусса, составляет:

$$T_{mem}^{1} = \frac{64\cdot\sum_{i=0}^{n-2}\left\lceil \frac{n-i}{\beta} + \left(\frac{n-i}{\beta}\right)^{2} + \frac{n-i}{\beta}\right\rceil}{\beta} = \frac{64\cdot\sum_{i=0}^{n-2}\left\lceil \left(\frac{n-i}{\beta}\right)^{2} + 2\cdot\frac{n-i}{\beta}\right\rceil}{\beta}.$$

На каждой i-й итерации обратного хода метода Гаусса производится обнуление элементов, расположенных в i-м столбце и корректировка элементов вектора правых частей в строках линейной системы, подлежащих обработке. Число таких строк равно $(n - i)$. Для выполнения вычислений необходимо прочитать из памяти элементы матрицы и вектора, подлежащие изменению. Подлежащие обработке элементы матрицы расположены в памяти не последовательно, следовательно, считывание из оперативной памяти в кэш будет происходить линейками по 64 байта. Таким образом, время, необходимое на чтение необходимых данных при выполнении обратного хода, может быть определено по формуле:

$$T_{mem}^{2} = \frac{64\cdot\sum_{i=0}^{n-2}\left\lceil \frac{n-i}{\beta} + \frac{n-i}{\beta}\right\rceil}{\beta} = \frac{64\cdot 2\cdot\sum_{i=0}^{n-2}\frac{n-i}{\beta}}{\beta}.$$

Общее время, необходимое на считывание необходимых данных из оперативной памяти в кэш при выполнении последовательного алгоритма Гаусса, составляет:

$$T_{mem} = \frac{64 \cdot \sum_{i=0}^{n-2} \left[(n-i)^2 + 2 \cdot (n-i) \right]}{\beta} + \frac{64 \cdot 2 \cdot \sum_{i=0}^{n-2} (n-i)}{\beta} = \frac{64 \cdot (2n^3 + 15n^2 + 13n - 30)}{6\beta}. \quad (8.6)$$

Следовательно, общее время выполнения последовательного алгоритма Гаусса составляет:

$$T_1 = \frac{4n^3 + 15n^2 + 11n - 30}{6}\tau + \frac{64 \cdot (2n^3 + 15n^2 + 13n - 30)}{6\beta}. \quad (8.7)$$

Если, как и в предыдущих разделах, учесть латентность памяти, то модель приобретет следующий вид:

$$T_1 = \frac{4n^3 + 15n^2 + 11n - 30}{6}\tau + \frac{2n^3 + 15n^2 + 13n - 30}{6}\left(\alpha + \frac{64}{\beta}\right). \quad (8.8)$$

Полученная модель является моделью на худший случай. Для более точной оценки необходимо учесть частоту кэш промахов γ (см. п. 6.5.4):

$$T_1 = \frac{4n^3 + 15n^2 + 11n - 30}{6}\tau + \gamma\frac{2n^3 + 15n^2 + 13n - 30}{6}\left(\alpha + \frac{64}{\beta}\right). \quad (8.9)$$

8.2.6. Результаты вычислительных экспериментов

Эксперименты проводились на двухпроцессорном вычислительном узле на базе четырехъядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Для снижения сложности построения теоретических оценок времени выполнения метода Гаусса, при компиляции и построении программ для проведения вычислительных экспериментов функция оптимизации кода компилятором была отключена (результаты оценки влияния компиляторной оптимизации на эффективность программного кода приведены в п. 7.2.4).

Чтобы оценить время τ одной операции, измерим время выполнения последовательного алгоритма Гаусса при малых объемах данных, таких, чтобы матрица линейной системы, вектор правых частей и вектор-результат полностью поместились в кэш вычислительного элемента (процессора или его ядра). Чтобы исключить необходимость выборки данных из оперативной памяти, перед началом вычислений заполним матрицу линейной системы и вектор правых частей случайными числами (для обеспечения гарантированной разрешимости системы уравнений исходная матрица задается в виде нижней треугольной матрицы), а вектор-результат –

нулями. Выполнение этих действий гарантирует предварительное перемещение данных в кэш. Далее при решении задачи все время будет тратиться непосредственно на вычисления, т. к. 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 5,820 нс.

Оценка величины пропускной способности β канала доступа к оперативной памяти проводилась в п. 6.5.4 и определена для используемого вычислительного узла как 12,44 Гб/с, а латентность памяти α = 8,31 нс.

В табл. 8.1 и на рис. 8.2 представлены результаты сравнения времени выполнения T_1 последовательного алгоритма Гаусса со временем T_1^*, полученным при помощи модели (8.9). Частота кэш=промахов, измеренная с помощью системы VPS, для одного потока была оценена как 0,0133.

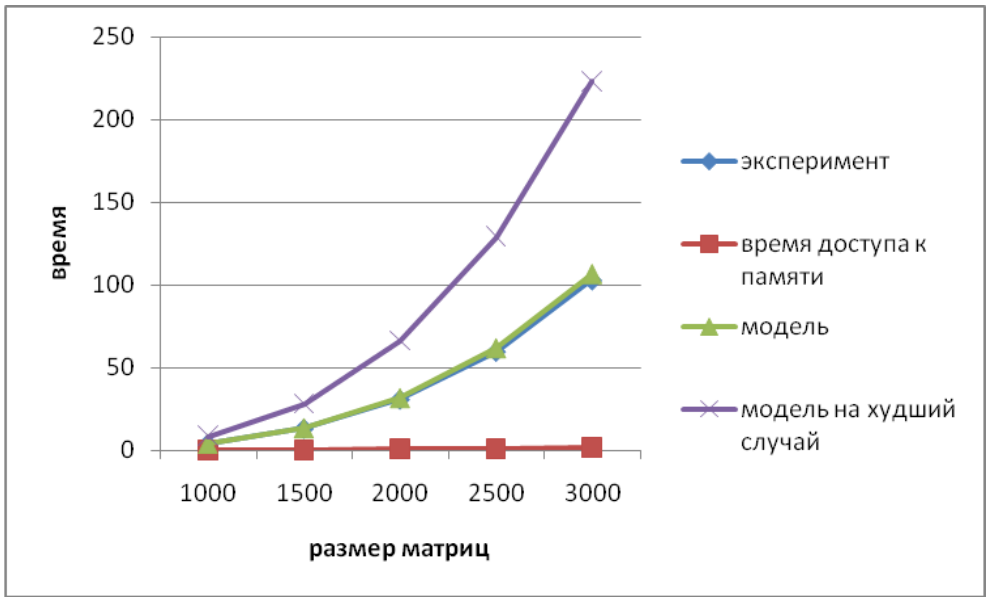

Рис. 8.2. График зависимости экспериментального и теоретического времен выполнения последовательного алгоритма от объема исходных данных

Таблица 8.1.

Сравнение экспериментального и теоретического времени выполнения последовательного алгоритма Гаусса

Размер матриц	T_1	T_1^* (*calc*) (модель)	Модель 8.8 – оценка сверху		Модель 8.9 – уточненная оценка	
			T_1^* (*mem*)	T_1^*	T_1^* (*mem*)	T_1^*
1000	3,7488	3,8946	4,3999	8,2945	0,0585	3,9531
1500	12,9136	13,1278	14,8128	27,9405	0,1970	13,3248
2000	30,5710	31,0982	35,0681	66,1663	0,4664	31,5646
2500	59,6408	60,7160	68,4411	129,1570	0,9103	61,6262
3000	103,0359	104,8910	118,2072	223,0982	1,5722	106,4631

8.3. Параллельный вариант метода Гаусса

При внимательном рассмотрении метода Гаусса можно заметить, что все вычисления сводятся к однотипным вычислительным операциям над строками матрицы коэффициентов системы линейных уравнений. Как результат, в основу параллельной реализации алгоритма Гаусса может быть положен принцип распараллеливания по данным. В качестве базовой подзадачи можно принять тогда все вычисления, связанные с обработкой одной строки матрицы A и соответствующего элемента вектора b.

8.3.1. Выделение информационных зависимостей

Рассмотрим общую схему параллельных вычислений и возникающие при этом информационные зависимости между базовыми подзадачами.

Для выполнения **прямого хода** метода Гаусса необходимо осуществить $(n - 1)$ итерацию по исключению неизвестных для преобразования матрицы коэффициентов A к верхнему треугольному виду.

Выполнение итерации i, $0 \le i < n-1$, прямого хода метода Гаусса включает ряд последовательных действий. Прежде всего, в самом начале итерации необходимо выбрать ведущую строку, которая при использовании метода главных элементов определяется поиском строки с наибольшим по абсолютной величине значением среди элементов столбца i, соответствующего исключаемой переменной x_i. Зная ведущую строку, подзадачи выполняют вычитание строк, обеспечивая тем самым исключение соответствующей неизвестной x_i..

При выполнении **обратного хода** метода Гаусса подзадачи выполняют необходимые вычисления для нахождения значения неизвестных. Как только какая-либо подзадача i, $0 \le i < n - 1$, определяет значение своей переменной x_i, это значение должно быть использовано всеми подзадача-

ми с номерами k, $k < i$, для выполнения корректировки значений элементов вектора b.

8.3.2. Масштабирование и распределение подзадач

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью. Вместе с этим следует учитывать, что по мере выполнения вычислений часть подзадач будет завершать свои вычисления (например, после того, как строка подзадачи использовалась в качестве ведущей при прямом ходе метода Гаусса). Решение этой проблемы может состоять, например, в укрупнении подзадач – тем более, что обычно размер матрицы, описывающей систему линейных уравнений, оказывается большим, чем число доступных вычислительных элементов (т. е., $p < n$). При этом для объединения нескольких строк в рамках единой подзадачи целесообразно использовать не последовательную, а циклическую схему разделения матрицы на горизонтальные полосы (см. раздел 6.2).

8.3.3. Программная реализация

Проведя анализ последовательного варианта алгоритма Гаусса, можно заключить, что распараллеливание возможно для следующих вычислительных процедур:

- поиск ведущей строки (функция *FindPivotRow*),
- вычитание ведущей строки из всех строк, подлежащих обработке (функция *EliminateColumns*),
- выполнение обратного хода метода Гаусса.

1. Распараллеливание процедуры поиска ведущей строки. На каждой i-й итерации прямого хода метода Гаусса необходимо просмотреть строки, подлежащие обработке, и выбрать среди них строку, у которой в i-м столбце расположен максимальный по абсолютному значению элемент. Каждый поток параллельной программы отвечает за вычисления, производимые над множеством строк матрицы линейной системы и соответствующими элементами векторов. Можно изменить алгоритм таким образом, чтобы сначала все потоки параллельно определили «локальную» ведущую строку в рамках своей полосы строк матрицы, а затем осуществить выбор «глобальной» ведущей строки среди «локальных» ведущих строк потоков.

Для этого объявим новую структуру данных для хранения «локальной» ведущей строки *TThreadPivotRow*. Полями этой структуры являются номер «локальной» ведущей строки в матрице линейной системы и значение, расположенное в столбце с исключаемой неизвестной.

Чтобы выбор ведущей строки выполнялся потоками параллельно, объявим параллельный фрагмент при помощи директивы *parallel*. Внутри параллельного фрагмента объявим переменную типа *TThreadPivotRow* для хранения «локальной» ведущей строки. Поскольку данная переменная объявлена внутри параллельного фрагмента, локальные копии этой переменной будут созданы во всех потоках параллельной секции. Для распределения итераций основного цикла выбора ведущей строки между потоками воспользуемся директивной *for*.

После того, как все потоки определили «локальную» ведущую строку, до завершения параллельного фрагмента необходимо выполнить редукцию полученных значений и выбрать «глобальную» ведущую строку. Для этого необходимо просмотреть выбранные «локальные» ведущие строки и выбрать среди них строку с максимальным значением поля *MaxValue*. Выбор глобальной ведущей строки можно обеспечить при помощи механизма *критических секций* (директива *critical*). Код, расположенный внутри критической секции, в каждый момент времени выполняется только одним потоком (любые потоки, пытающиеся получить доступ к уже выполняемой критической секции, блокируются). После того, как выполнение критической секции завершается, к ее выполнению может приступить другой поток (в первую очередь из числа блокированных при доступе к данной критической секции). Подобная схема выполнения обычно именуется *взаимоисключением* – более подробно понятие критической секции рассмотрено в разделе 4 учебных материалов.

С учетом высказанных предложений параллельный вариант функции *FindPivotRow* может быть представлен следующим образом.

```
// Программа 9.2
// Параллельный алгоритм Гаусса
typedef struct {
  int PivotRow;
  double MaxValue;
} TThreadPivotRow;

// Функция для нахождения ведущей строки
int ParallelFindPivotRow(double* pMatrix, int Size,
  int Iter) {
  int PivotRow = -1;    // Номер ведущей строки
  double MaxValue =  0; // Значение ведущего элемента
  int i;

  // Определение строки с max элементом
#pragma omp parallel
```

```
  {
    TThreadPivotRow ThreadPivotRow;
    ThreadPivotRow.MaxValue = 0;
    ThreadPivotRow.PivotRow = -1;
#pragma omp for
    for (i=0; i<Size; i++) {
      if ((pSerialPivotIter[i] == -1) &&
        (fabs(pMatrix[i*Size+Iter]) >
          ThreadPivotRow.MaxValue)) {
        ThreadPivotRow.PivotRow = i;
        ThreadPivotRow.MaxValue                    =
fabs(pMatrix[i*Size+Iter]);
      }
    }
#pragma omp critical
    {
      if (ThreadPivotRow.MaxValue > MaxValue){
        MaxValue = ThreadPivotRow.MaxValue;
        PivotRow = ThreadPivotRow.PivotRow;
      }
    } // pragma omp critical
  } // pragma omp parallel
  return PivotRow;
}
```

Следует понимать, что введение синхронизации при организации критической секции уменьшит возможность достижения максимально возможного ускорения параллельных вычислений. Для снижения нежелательных эффектов синхронизации можно, например, при помощи запоминания результатов потоков (параметров «потоковых» ведущих строк) в отдельном массиве, в котором далее выполнить обычный последовательный поиск (реализация данного варианта поиска ведущей строки может быть использована в качестве темы самостоятельного упражнения).

2. Распараллеливание процедуры исключения очередной неизвестной. После выбора «глобальной» ведущей строки необходимо обнулить элементы столбца с исключаемой неизвестной во всех строках, подлежащих обработке. Реализуем параллельный вариант этого алгоритма в функции *ParallelEliminateColumns*.

За обработку одной строки матрицы линейной системы отвечает одна итерация внешнего цикла по переменной *i*. Поскольку преобразования строк осуществляются независимо, то можно распределить их между параллельными потоками и выполнить параллельно. Для этого воспользуем-

ся директивой *parallel for* для распределения итераций внешнего цикла между потоками. При этом переменная *PivotFactor*, которая используется для хранения множителя, на который умножается текущая строка, должна быть локализована (параметр *private* директивы *parallel for*):

```
// Функция исключения текущей неизвестной
void ParallelColumnElimination (double* pMatrix,
  double* pVector, int Size,
  int Iter, int Pivot) {
  double PivotValue, PivotFactor;
  PivotValue = pMatrix[Pivot*Size+Iter];
#pragma omp parallel for private (PivotFactor)
  for (int i=0; i<Size; i++) {
    if (pSerialPivotIter[i] == -1) {
      PivotFactor = pMatrix[i*Size+Iter] / PivotValue;
      for (int j=Iter; j<Size; j++) {
        pMatrix[i*Size + j] -= PivotFactor *
          pMatrix[Pivot*Size+j];
      }
      pVector[i] -= PivotFactor * pVector[Pivot];
    }
  }
}
```

Следует отметить, что итерации цикла по умолчанию распределяются между потоками поровну. Например, если для выполнения цикла, содержащего 100 итераций, используется два потока, то первый поток отвечает за выполнение итераций с номерами от 0 до 49, а второй – за выполнение итераций с номерами от 50 до 99. В случае, если итерации цикла имеют разный объем вычислений, такой способ распределения итераций может приводить к неравномерной вычислительной нагрузке потоков.

При выполнении прямого хода метода Гаусса после выбора ведущей строки вычисления производятся только над строками, подлежащими обработке (строка с номером i подлежит обработке, если она ранее не выбиралась в качестве ведущей – признаком такой ситуации является значение элемента массива *pSerialPivotIter[i]*, равное –1). Расположение уже отработанных строк может быть достаточно произвольным и определяется порядком выбора ведущих строк в соответствии с методом главных элементов. Как результат, итерации цикла могут обладать разной вычислительной сложностью – итерация цикла может быть «нагруженной», если она соответствует еще неотработанной строке и необходимо выполнить вычитание строк, а может быть и вычислительно простой – при попадании на уже отработанную строку матрицы.

Для того, чтобы обеспечить равномерную загрузку вычислительных элементов, можно изменить принятое по умолчанию правило для распределения итераций между потоками при помощи параметра *schedule* директивы *parallel for*, выбрав, например, динамическую схему распределения итераций между потоками. В этом случае, итерации цикла распределяются между потоками поблочно; при завершении обработки очередного блока каждый поток получает для вычислений очередной блок итераций – более подробно правила управления распределением итераций цикла между потоками рассмотрены в разделе 5 учебных материалов.

Как можно видеть, подобная динамическая схема распределения итераций цикла между потоками может решить проблему балансировки вычислений. Возможный улучшенный вариант параллельного выполнения процедуры исключения очередной неизвестной для прямого хода методы Гаусса может выглядеть следующим образом:

```
// Функция исключения текущей неизвестной
void ParallelColumnElimination (double* pMatrix,
  double* pVector, int Pivot,
  int Iter, int Size) {
  double PivotValue, PivotFactor;
  PivotValue = pMatrix[Pivot*Size+Iter];
#pragma omp parallel for private(PivotFactor) /
  schedule(dynamic,1)
  for (int i=0; i<Size; i++) {
    …
  }
}
```

Динамическая схема (*dynamic*) организует подобие «очереди итераций»: каждый поток берет на выполнение текущую итерацию из очереди и как только итерация выполнена, происходит новое обращение к очереди за новой итерацией. Такой способ распределения вычислительной нагрузки гарантирует балансировку вычислительных элементов.

Используя разработанные функции *ParallelFindPivotRow* и *ParallelColumnElimination*, параллельный вариант прямого хода метода Гаусса может быть представлен в следующем виде:

```
// Функция для выполнения прямого хода метода Гаусса
void ParallelGaussianElimination(double* pMatrix,
  double* pVector,int Size) {
  int Iter;       // Номер итерации
  int PivotRow;   // Номер ведущей строки
  for (Iter=0; Iter<Size; Iter++) {
    // Определение ведущей строки
```

```
    PivotRow = ParallelFindPivotRow(pMatrix, Size,
      Iter);
    pPivotPos[Iter] = PivotRow;
    pPivotIter[PivotRow] = Iter;
    ParallelColumnElimination(pMatrix, pVector, Size,
      Iter, PivotRow);
  }
}
```

3. Распараллеливание обратного хода метода Гаусса. Итерации обратного хода метода Гаусса должны выполняться строго последовательно. Однако обработка строк линейной системы на каждой итерации обратного хода происходит независимо, и, следовательно, данные вычисления можно распределить между параллельными потоками согласно вычислительной схеме параллельного алгоритма.

При помощи директивы *parallel for* распределим между потоками параллельной программы итерации внутреннего цикла, при этом переменная *Row*, которая хранит номер текущей обрабатываемой строки, должна быть локализована. Получаемый в результате параллельный вариант функции для выполнения обратного хода метода Гаусса может выглядеть следующим образом:

```
// Функция для выполнения обратного хода метода Гаусса
void ParallelBackSubstitution (double* pMatrix,
  double* pVector, double* pResult, int Size) {
  int RowIndex, Row;
  for (int i=Size-1; i>=0; i--) {
    RowIndex = pSerialPivotPos[i];
    pResult[i] =
      pVector[RowIndex]/pMatrix[RowIndex*Size+i];
    pMatrix[RowIndex*Size+i] = 1;
#pragma omp parallel for private (Row)
    for (int j=0; j<i; j++) {
      Row = pSerialPivotPos[j];
      pVector[Row] -= pMatrix[Row*Size+i]*pResult[i];
      pMatrix[Row*Size+i] = 0;
    }
  }
}
```

8.3.4. Анализ эффективности

При разработке параллельного алгоритма все вычислительные операции, выполняемые алгоритмом Гаусса, были распределены между потоками параллельной программы. Однако для реализации параллельного метода главных элементов потребовалось ввести дополнительный блок кода, отвечающий за редукцию данных, полученных разными потоками.

Следовательно, время, необходимое для выполнения вычислений на этапе прямого хода, можно определить при помощи соотношения:

$$T_p(calc) = T_1(calc)/p + 3p\tau,$$

где второе слагаемое $3p\tau$ определяет затраты на выбор «глобальной» ведущей строки. Подставив выражение $T_1(calc)$, получим, что время выполнения вычислений для параллельного варианта метода описывается выражением:

$$T_p(calc) = [(4n^3+15n^2+11n-30)/6p]\tau + 3p\tau. \qquad (8.10)$$

Поскольку обращения к памяти параллельными потоками осуществляются строго последовательно, то время на чтение необходимых данных из оперативной памяти в кэш при выполнении параллельного метода Гаусса совпадает со временем, полученным при анализе последовательного алгоритма:

$$T_p(mem) = \frac{64\cdot\ 2n^3+15n^2+13n-30}{6\beta}. \qquad (8.11)$$

Если учесть латентность памяти, то получим:

$$T_p(mem) = \frac{2n^3+15n^2+13n-30}{6}\left(\alpha+\frac{64}{\beta}\right). \qquad (8.12)$$

Теперь необходимо оценить величину накладных расходов, обусловленных организацией и закрытием параллельных секций. Как отмечалось в гл. 6, время δ, необходимое на организацию и закрытие параллельной секции, составляет 0,25 мкс. Параллельная секция создается при каждом выборе ведущей строки, при выполнении вычитания ведущей строки из остальных строк линейной системы, подлежащих обработке, а также при выполнении каждой итерации обратного хода метода Гаусса. Таким образом, общее число параллельных секций составляет $3\cdot(n-1)$.

Сводя воедино все полученные оценки, можно заключить, что время выполнения параллельного метода Гаусса описывается соотношением:

$$T_p = \frac{4n^3+15n^2+11n-30}{6p}\tau + 3p + \\ + \frac{2n^3+15n^2+13n-30}{6}\left(\alpha+\frac{64}{\beta}\right) + 3(n-1)\delta\cdot \qquad (8.13)$$

Для уточнения модели следует учесть коэффициент кэш промахов γ:

$$T_p = \frac{4n^3+15n^2+11n-30}{6p}\tau + 3p\tau +$$

$$+\gamma\frac{2n^3+15n^2+13n-30}{6}\left(\alpha+\frac{64}{\beta}\right)+3(n-1)\delta. \quad (8.14)$$

8.3.5. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта метода Гаусса для решения систем линейных уравнений проводились при условиях, указанных в п. 6.5.5 и состоят в следующем.

Эксперименты проводились на двухпроцессорном вычислительном узле на базе четырехъядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows.Результаты вычислительных экспериментов приведены в табл. 8.2 и на рис. 8.3. Времена выполнения алгоритмов указаны в секундах.

В табл. 8.3, 8.4 и на рис. 8.4, 8.5 представлены результаты сравнения времени выполнения T_p параллельного алгоритма Гаусса с использованием двух и четырех потоков со временем T_p^*, полученным при помощи модели (8.14). Частота кэш-промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,045, а для четырех потоков эта величина была оценена как 0,077.

Таблица 8.2.

Результаты вычислительных экспериментов для параллельного алгоритма Гаусса

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		время	ускорение	время	ускорение
1000	3,7488	2,8875	1,2983	1,5627	2,3989
1500	12,9136	7,9223	1,6300	4,7501	2,7186
2000	30,5710	17,0476	1,7933	11,3656	2,6898
2500	59,6408	31,8271	1,8739	22,2635	2,6789
3000	103,0359	53,7655	1,9164	38,5342	2,6739

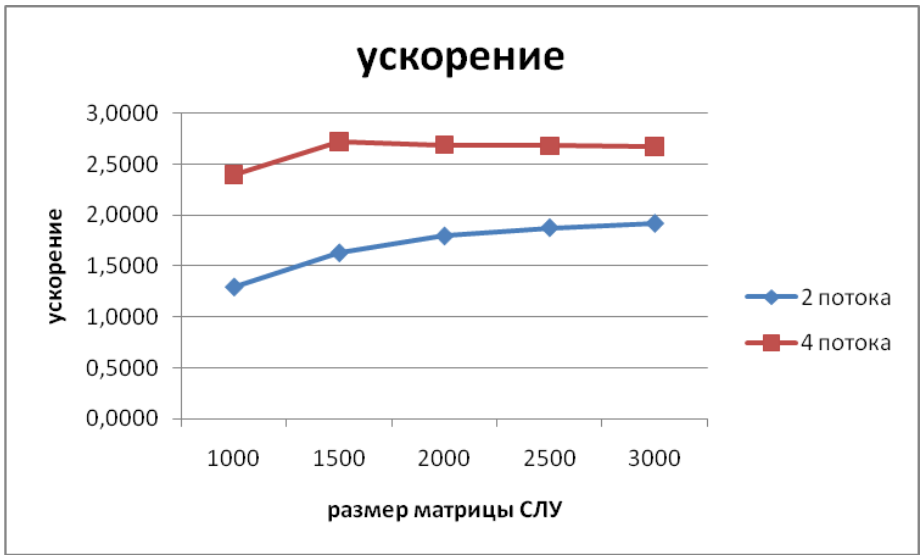

Рис. 8.3. Зависимость ускорения от количества исходных данных при выполнении параллельного алгоритма Гаусса

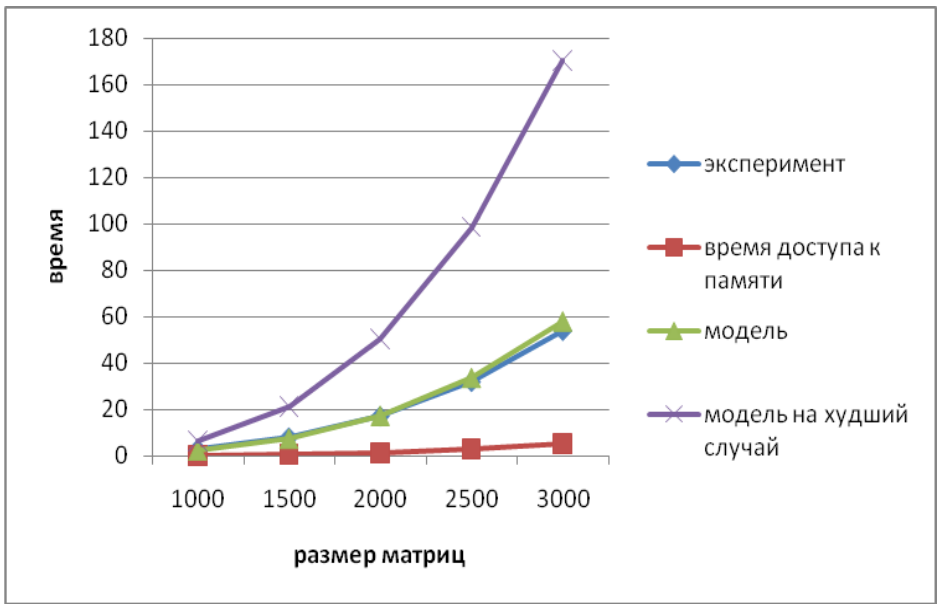

Рис. 8.4. График зависимости экспериментального и теоретического времен выполнения параллельного алгоритма Гаусса от объема исходных данных при использовании двух потоков

Таблица 8.3.

Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма Гаусса с использованием двух потоков

Размер матриц	T_p	T_p^* (*calc*) (модель)	Модель 8.13 – оценка сверху		Модель 8.14 – уточненная оценка	
			T_p^* (*mem*)	T_p^*	T_p^* (*mem*)	T_p^*
1000	2,8875	1,9480	4,3999	6,3479	0,1980	2,1460
1500	7,9223	6,5650	14,8128	21,3778	0,6666	7,2316
2000	17,0476	15,5506	35,0681	50,6187	1,5781	17,1287
2500	31,8271	30,3599	68,4411	98,8009	3,0798	33,4397
3000	53,7655	52,4477	118,2072	170,6549	5,3193	57,7671

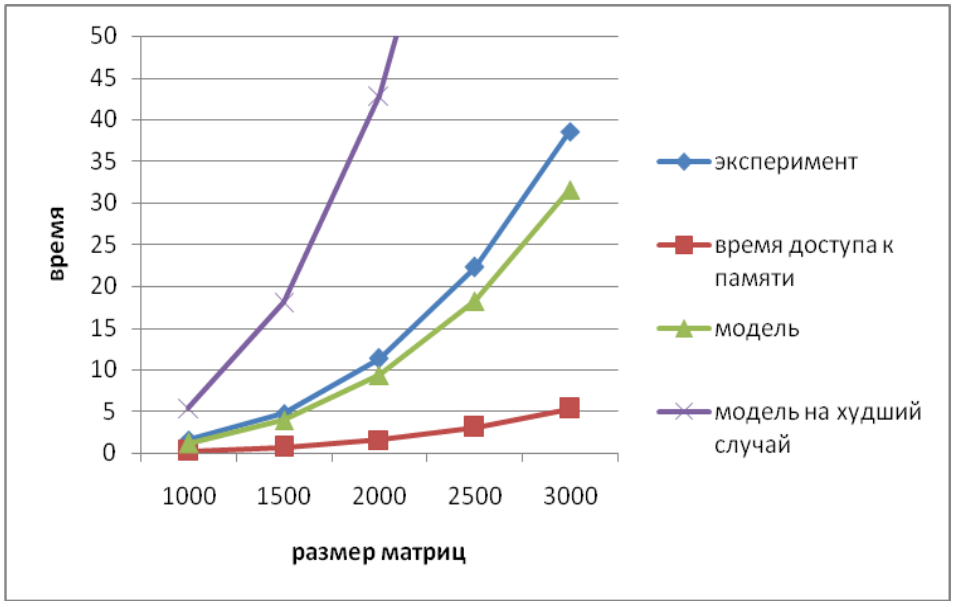

Рис. 8.5. График зависимости экспериментального и теоретического времен выполнения параллельного алгоритма Гаусса от объема исходных данных при использовании четырех потоков

Таблица 8.4.
Сравнение экспериментального и теоретического времен выполнения параллельного алгоритма Гаусса с использованием четырех потоков

Размер матриц	T_p	T_p^* (*calc*) (модель)	Модель 8.13 – оценка сверху		Модель 8.14 – уточненная оценка	
			T_p^* (*mem*)	T_p^*	T_p^* (*mem*)	T_p^*
1000	1,5627	0,9744	4,3999	5,3743	0,1980	1,1724
1500	4,7501	3,2831	14,8128	18,0958	0,6666	3,9496
2000	11,3656	7,7761	35,0681	42,8441	1,5781	9,3541
2500	22,2635	15,1809	68,4411	83,6219	3,0798	18,2607
3000	38,5342	26,2250	118,2072	144,4322	5,3193	31,5443

8.4. Метод сопряженных градиентов

Рассмотрим теперь совершенно иной подход к решению систем линейных уравнений, при котором к искомому точному решению $x*$ системы $Ax = b$ строится последовательность приближенных решений x^0, x^1,…, x^k,… При этом процесс вычислений организуется таким способом, что каждое очередное приближение дает оценку точного решения со все уменьшающейся погрешностью, и при продолжении расчетов оценка точного решения может быть получена с любой требуемой точностью. Подобные *итерационные методы* решения систем линейных уравнений широко используются в практике вычислений. К преимуществам итерационных методов можно отнести меньший объем (по сравнению, например, с методом Гаусса) необходимых вычислений для решения разреженных систем линейных уравнений, возможность более быстрого получения начальных приближений искомого решения, наличие эффективных способов организации параллельных вычислений. Дополнительная информация с описанием таких методов, рассмотрение вопросов сходимости и точности получаемых решений может быть получена, например, в [2–4,44].

Одним из наиболее известных итерационных алгоритмов является *метод сопряженных градиентов*, который может быть применен для решения симметричной положительно определенной системы линейных уравнений большой размерности.

Напомним, что матрица A является *симметричной*, если она совпадает со своей транспонированной матрицей, т. е. $A = A^T$. Матрица A назы-

вается *положительно определенной*, если для любого вектора x справедливо: $x^T Ax > 0$.

Известно (см., например, [2–4,44]), что после выполнения n итераций алгоритма (n есть порядок решаемой системы линейных уравнений), очередное приближение x^n совпадает с точным решением.

8.4.1. Последовательный алгоритм

Если матрица A симметричная и положительно определенная, то функция

$$q(x) = \frac{1}{2} x^T \cdot A \cdot x - x^T b + c \tag{8.15}$$

имеет единственный минимум, который достигается в точке $x*$, совпадающей с решением системы линейных уравнений (8.2). *Метод сопряженных градиентов* является одним из широкого класса итерационных алгоритмов, которые позволяют найти решение (8.2) путем минимизации функции $q(x)$.

Итерация метода сопряженных градиентов состоит в вычислении очередного приближения к точному решению в соответствии с правилом:

$$x^k = x^{k-1} + s^k d^k . \tag{8.16}$$

Тем самым, новое значение приближения x^k вычисляется с учетом приближения, построенного на предыдущем шаге x^{k-1}, скалярного шага s^k и вектора направления d^k.

Перед выполнением первой итерации вектора x^0 и d^0 полагаются равными нулю, а для вектора g^0 устанавливается значение равное $–b$. Далее каждая итерация для вычисления очередного значения приближения x^k включает выполнение четырех шагов:

Шаг 1: Вычисление градиента:

$$g^k = A \cdot x^{k-1} - b ; \tag{8.17}$$

Шаг 2: Вычисление вектора направления:

$$d^k = -g^k + \frac{(g^k)^T , g^k}{(g^{k-1})^T , g^{k-1}} d^{k-1}, \tag{8.18}$$

где (g^T,g) есть скалярное произведение векторов;

Шаг 3: Вычисление величины смещения по выбранному направлению:

$$s^k = \frac{d^k, g^k}{(d^k)^T \cdot A \cdot d^k}; \tag{8.19}$$

Шаг 4: Вычисление нового приближения:

$$x^k = x^{k-1} + s^k d^k. \tag{8.20}$$

Как можно заметить, данные выражения включают две операции умножения матрицы на вектор, четыре операции скалярного произведения и пять операций над векторами. Как результат, общее количество числа операций, выполняемых на одной итерации, составляет

$$t_1 = 4n^2 + 11n.$$

Как уже отмечалось ранее, для нахождения точного решения системы линейных уравнений с положительно определенной симметричной матрицей необходимо выполнить n итераций. Таким образом, для нахождения решения системы необходимо выполнить

$$T_1 = 4n^3 + 11n^2, \tag{8.21}$$

и, тем самым, сложность алгоритма имеет порядок $O(n^3)$.

Поясним выполнение метода сопряженных градиентов на примере решения системы линейных уравнений вида:

$$\begin{array}{rcl} 3x_0 - x_1 & = & 3 \\ -x_0 + 3x_1 & = & 7 \end{array}.$$

Эта система уравнений второго порядка обладает симметричной положительно определенной матрицей, для нахождения точного решения этой системы достаточно провести всего две итерации метода.

На первой итерации было получено значение градиента $g^1 = (-3, -7)$, значение вектора направления $d^1 = (3, 7)$, значение величины смещения $s^1 = 0.439$. Соответственно, очередное приближение к точному решению системы $x^1 = (1.318, 3.076)$.

На второй итерации было получено значение градиента $g^2 = (-2.121, 0.909)$, значение вектора направления $d^2 = (2.397, -0.266)$, а величина смещения – $s^2 = 0.284$. Очередное приближение $x^2 = (2, 3)$ совпадает с точным решением системы x^*.

На рис. 8.6 представлена последовательность приближений к точному решению, построенная методом сопряженных градиентов (в качестве начального приближения x^0 выбрана точка (0,0)).

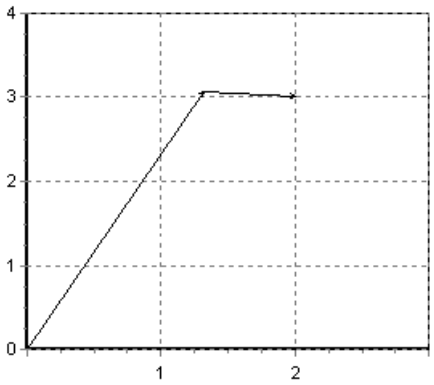

Рис. 8.6. Итерации метода сопряженных градиентов при решении системы второго порядка

8.4.2. Организация параллельных вычислений

При разработке параллельного варианта метода сопряженных градиентов для решения систем линейных уравнений в первую очередь следует учесть, что выполнение итераций метода осуществляется последовательно и тем самым наиболее целесообразный подход состоит в распараллеливании вычислений, реализуемых в ходе выполнения итераций.

Анализ соотношений (8.17)–(8.21) показывает, что основные вычисления, выполняемые в соответствии с методом, состоят в умножении матрицы A на вектоpы x и d, и, как результат, при организации параллельных вычислений может быть полностью использован материал, изложенный в разделе 6.

Дополнительные вычисления в (8.17)–(8.21), имеющие меньший порядок сложности, представляют собой различные операции обработки векторов (скалярное произведение, сложение и вычитание, умножение на скаляр). Организация таких вычислений, конечно же, должна быть согласована с выбранным параллельным способом выполнения операция умножения матрицы на вектор.

8.4.3. Программная реализация

Рассмотрим возможный вариант реализации параллельного варианта метода сопряженных градиентов для решения систем линейных уравнений с симметричной положительно-определенной матрицей. При этом реализация отдельных модулей не приводится, если их отсутствие не оказывает влияния на понимании общей схемы вычислений.

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

```
// Программа 9.3
// Метод сопряженных градиентов
void main() {
  double* pMatrix;  // Матрица системы уравнений
  double* pVector;  // Вектор правых частей
  double* pResult;  // Вектор неизвестных
  int Size;         // Количество уравнений

  printf ("Метод сопряженных градиентов\n");

  // Выделение памяти и инициализация данных
  ProcessInitialization(pMatrix, pVector,
    pResult, Size);

  // Выполнение метода сопряженных градиентов
  ParallelResultCalculation(pMatrix, pVector,
    pResult, Size);

  // Завершение вычислений
  ProcessTermination(pMatrix, pVector, pResult);
}
```

Функция *ProcessInitialization* определяет исходные данные решаемой задачи (число неизвестных), выделяет память для хранения данных, осуществляет ввод матрицы коэффициентов системы уравнений и вектора правых частей (или формирует эти данные при помощи датчика случайных чисел).

Функция *ProcessTermination* выполняет необходимый вывод результатов решения задачи и освобождает всю ранее выделенную память для хранения данных.

Реализация всех перечисленных функций может быть выполнена по аналогии с ранее рассмотренными примерами и предоставляется читателю в качестве самостоятельного упражнения.

2. Функция ParallelResultCalculation. Реализует логику работы параллельного варианта метода сопряженных градиентов.

```
// Функция для метода сопряженных градиентов
void ParallelResultCalculation(double* pMatrix,
  double* pVector, double* pResult, int Size) {
  double *CurrentApproximation,
         *PreviousApproximation;
  double *CurrentGradient,  *PreviousGradient;
  double *CurrentDirection, *PreviousDirection;
```

```
  double  *Denom;
  double Step;

  int Iter = 1, MaxIter = Size + 1;
  float Accuracy = 0.0001f;

  AllocateVectors(CurrentApproximation,
    PreviousApproximation, CurrentGradient,
    PreviousGradient, CurrentDirection,
    PreviousDirection, Denom, Size);

  for (int i=0; i<Size; i++) {
    PreviousApproximation[i] = 0;
    PreviousDirection[i] = 0;
    PreviousGradient[i] = -pVector[i];
  }

  do {
    if (Iter > 1) {
      SwapPointers(PreviousApproximation,
        CurrentApproximation);
      SwapPointers(PreviousGradient, CurrentGradient);
      SwapPointers(PreviousDirection,
        CurrentDirection);
    }
    // Вычисление градиента
#pragma omp parallel for
    for (int i=0; i<Size; i++) {
      CurrentGradient[i] = -pVector[i];
      for (int j=0; j<Size; j++)
        CurrentGradient[i] +=
          pMatrix[i*Size+j]*PreviousApproximation[j];
    }

    // Вычисление направления
    double IP1 = 0, IP2 = 0;
#pragma omp parallel for reduction(+:IP1,IP2)
    for (int i=0; i<Size; i++) {
      IP1 += CurrentGradient[i]*CurrentGradient[i];
      IP2 += PreviousGradient[i]*PreviousGradient[i];
    }
#pragma omp parallel for
    for (i=0; i<Size; i++) {
```

```
      CurrentDirection[i] = -CurrentGradient[i] +
        PreviousDirection[i]*IP1/IP2;
    }

    // Вычисление величины шага
    IP1 = 0;
    IP2 = 0;
#pragma omp parallel for reduction(+:IP1,IP2)
    for (int i=0; i<Size; i++) {
      Denom[i] = 0;
      for (int j=0; j<Size; j++)
        Denom[i] +=
          pMatrix[i*Size+j]*CurrentDirection[j];
      IP1 += CurrentDirection[i]*CurrentGradient[i];
      IP2 += CurrentDirection[i]*Denom[i];
    }
    Step = -IP1/IP2;

    // Вычисление нового приближения
#pragma omp parallel for
    for (i=0; i<Size; i++) {
      CurrentApproximation[i] =
        PreviousApproximation[i] +
        Step*CurrentDirection[i];
    }
    Iter++;
  } while
      ((Dest(PreviousApproximation,
        CurrentApproximation, Size) > Accuracy)
       && (Iter < MaxIter));

  for (int i=0; i<Size; i++)
    pResult[i] = CurrentApproximation[i];

  DeleteVectors(CurrentApproximation,
    PreviousApproximation, CurrentGradient,
    PreviousGradient, CurrentDirection,
    PreviousDirection, Denom);
}
```

8.4.4. Анализ эффективности

Выберем для дальнейшего анализа эффективности получаемых параллельных вычислений параллельный алгоритм матрично-векторного умно-

жения при ленточном горизонтальном разделении матрицы. При этом операции над векторами, обладающие меньшей вычислительной трудоемкостью, также будем выполнять в многопоточном режиме.

Вычислительная трудоемкость последовательного метода сопряженных градиентов была уже определена ранее в (8.21).

Определим время выполнения параллельной реализации метода сопряженных градиентов. Вычислительная сложность параллельных операций умножения матрицы на вектор при использовании схемы ленточного горизонтального разделения матрицы составляет (здесь и далее L – количество итераций, выполняемых методом):

$$T_p^1(calc) = L \cdot \frac{2n \cdot \ 2n-1}{p} \cdot \tau \text{ (см. раздел 6).}$$

Все остальные операции над векторами (скалярное произведение, сложение, умножение на константу), также выполняются в многопоточном режиме. Следовательно, общая вычислительная сложность параллельного варианта метода сопряженных градиентов является равной:

$$T_p(calc) = L \cdot \frac{4n^2 + 11n}{p} \cdot \tau . \qquad (8.22)$$

Уточним теперь приведенные выражения – оценим трудоемкость операции считывания необходимых данных из оперативной памяти в кэш вычислительных элементов. Пусть размер матрицы линейной системы настолько велик, что матрица и векторы, принимающие участие в вычислениях, не могут быть одновременно помещены в кэш. Отдельно рассмотрим каждый из четырех шагов, выполняемых на каждой итерации метода сопряженных градиентов.

В операции вычисления градиента принимают участие матрица системы, вектор правых частей, вектор предыдущего приближения и непосредственно сам вектор градиента. Следовательно, объем данных, прочитанных из оперативной памяти, составляет:

$$T_p^1(mem) = \frac{64 \ \ n^2 + 3n}{\beta} .$$

Для вычисления текущего вектора направления необходимо получить доступ к текущему и предыдущему векторам градиентов, к вектору направления, полученному на предыдущем шаге, а также к текущему вектору направления. Следовательно, время, необходимое на загрузку данных, составляет:

$$T_p^2(mem) = \frac{64 \cdot 4n}{\beta}.$$

Далее для вычисления величины смещения в выбранном направлении необходимо выполнить вычисления, в которых участвуют матрица линейной системы, а также вектор градиента и вектор направления. Затраты на загрузку необходимых данных:

$$T_p^3(mem) = \frac{64 \ n^2 + 2n}{\beta}.$$

При вычислении очередного приближения используются векторы предыдущего и текущего приближений, и вектор направления. Следовательно, время на чтение необходимых данных из оперативной памяти в кэш:

$$T_p^4(mem) = \frac{64 \cdot 3n}{\beta}.$$

Далее для определения достигнутой точности вычисляется расстояние между предыдущим и текущим приближениями. Для этого необходимо иметь доступ к этим векторам:

$$T_p^5(mem) = \frac{64 \cdot 2n}{\beta}.$$

Следовательно, время, которое тратится на загрузку необходимых данных при выполнении параллельного варианта метода сопряженных градиентов, составляет:

$$T_p(mem) = L \cdot \frac{64 \cdot \ 2n^2 + 14n}{\beta}. \qquad (8.23)$$

В модели необходимо также учесть латентность оперативной памяти (см п. 6.5.4):

$$T_p(mem) = L \cdot \ 2n^2 + 14n \ \cdot \left(\alpha + \frac{64}{\beta} \right). \qquad (8.24)$$

Суммируя полученные выражения, в итоге получаем, что общее время выполнения алгоритма в худшем случае составляет:

$$T_p = L \cdot \frac{4n^2 + 11n}{p} \cdot \tau + L \cdot \ 2n^2 + 14n \ \cdot \left(\alpha + \frac{64}{\beta} \right) \qquad (8.25)$$

Для построения точной модели необходимо также учесть коэффициент кэш промахов (см п. 6.5.4):

$$T_p = L \cdot \frac{4n^2 + 11n}{p} \cdot \tau + \gamma \cdot L \cdot \left(2n^2 + 14n\right) \cdot \left(\alpha + \frac{64}{\beta}\right) \qquad (8.26)$$

8.4.5. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта метода сопряженных градиентов для решения систем линейных уравнений с симметричной положительно-определенной матрицей проводились при условиях, указанных в п. 8.3.5. Результаты вычислительных экспериментов приведены в табл. 8.5. Времена выполнения алгоритмов указаны в секундах.

Таблица 8.5.

Результаты вычислительных экспериментов для параллельного метода сопряженных градиентов

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		время	ускорение	время	ускорение
1000	20,06335	11,24981	1,7834	5,631906	3,5624
1500	67,79017	38,71493	1,7510	19,57367	3,4633
2000	160,5922	92,15922	1,7426	46,41194	3,4601
2500	315,8357	180,7888	1,7470	91,32836	3,4582
3000	546,2592	312,0792	1,7504	157,7329	3,4632

В табл. 8.6, 8.7 и на рис. 8.8, 8.9 представлены результаты сравнения времени выполнения T_p параллельного метода сопряженных градиентов с использованием двух и четырех потоков со временем T_p^*, полученным при помощи модели (8.26) (длительность одной базовой вычислительной операции τ равна 5,057 нс). Количество проведенных итераций метода всегда равнялось количеству строк матрицы системы линейных уравнений. Частота кэш-промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,0085, а для четырех потоков эта величина была оценена как 0,0102.

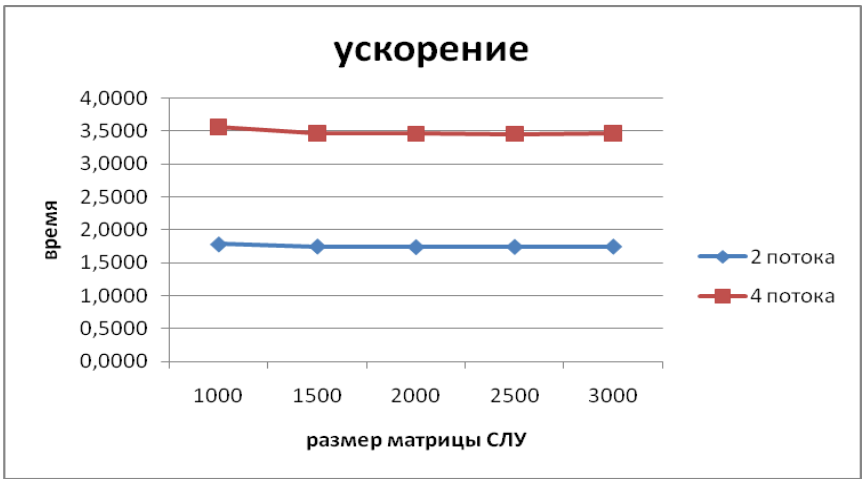

Рис. 8.7. Зависимость ускорения от количества исходных данных при выполнении параллельного метода сопряженных градиентов

Таблица 8.6.

Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма сопряженных градиентов с использованием двух потоков

Размер матриц	T_p	T_p^* (*calc*) (модель)	Модель 8.25 – оценка сверху		Модель 8.26 – уточненная оценка	
			T_p^* (*mem*)	T_p^*	T_p^* (*mem*)	T_p^*
1000	11,2498	10,1418	26,3862	36,5280	0,2243	10,3661
1500	38,7149	34,1973	88,8469	123,0443	0,7552	34,9525
2000	92,1592	81,0233	210,3556	291,3789	1,7880	82,8113
2500	180,7888	158,2051	410,5642	568,7693	3,4898	161,6949
3000	312,0792	273,3283	709,1248	982,4531	6,0276	279,3559

Таблица 8.7.

Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма сопряженных градиентов с использованием четырех потоков

Размер матриц	T_p	T_p^* (*calc*) (модель)	Модель 8.25 – оценка сверху		Модель 8.26 – уточненная оценка	
			T_p^* (*mem*)	T_p^*	T_p^* (*mem*)	T_p^*
1000	5,6319	5,0709	26,3862	31,4571	0,2691	5,3400
1500	19,5737	17,0987	88,8469	105,9456	0,9062	18,0049
2000	46,4119	40,5116	210,3556	250,8672	2,1456	42,6573
2500	91,3284	79,1025	410,5642	489,6667	4,1878	83,2903
3000	157,7329	136,6642	709,1248	845,7889	7,2331	143,8972

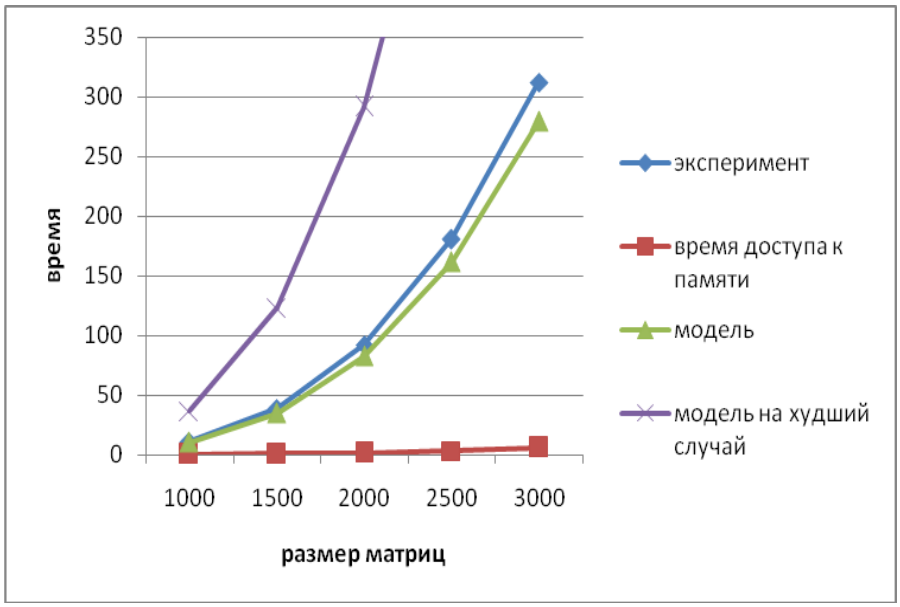

Рис. 8.8. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма сопряженных градиентов от объема исходных данных при использовании двух потоков

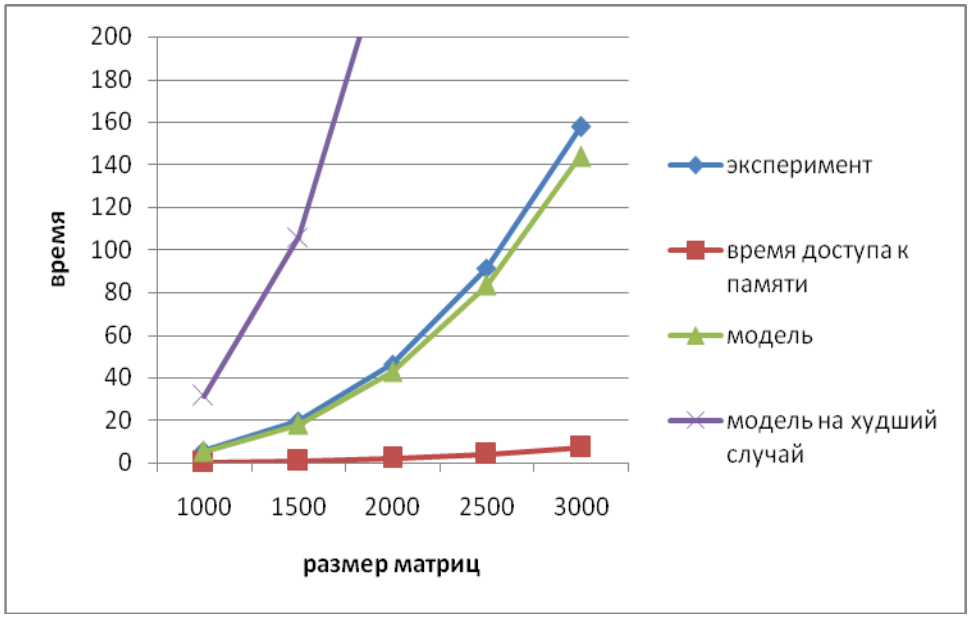

Рис. 8.9. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма сопряженных градиентов от объема исходных данных при использовании четырех потоков

8.5. Краткий обзор главы

Данная глава посвящена проблеме параллельных вычислений при решении систем линейных уравнений. Изложение материала проводится с использованием двух известных алгоритма – *метод Гаусса*, как пример прямого алгоритма решения задачи, и итерационный *метод сопряженных градиентов*.

Параллельный вариант метода Гаусса основывается на ленточном разделении матрицы между вычислительными элементами. Для определения параллельного варианта метода проведен полный цикл проектирования – определяются базовые подзадачи, выделяются информационные взаимодействия, обсуждаются вопросы масштабирования, выводятся оценки показателей эффективности, предлагается схема программной реализации и приводятся результаты вычислительных экспериментов. При программной реализации алгоритма для лучшей балансировки вычислительной нагрузки между потоками используется поддерживаемая в OpenMP динамическая схема распределения вычислений.

Важный момент при рассмотрении *параллельного варианта метода сопряженных градиентов* состоит в том, что здесь способ параллельных вычислений можно получить через параллельные алгоритмы выполняемых вычислений – операций умножения матрицы на вектор, скалярного произведения векторов, сложения и вычитания векторов. Выбранный для учебного изучения подход состоит в распараллеливании всех операций над матрицами и векторами. Этот подход позволил организовать параллельные вычисления с высокими показателями эффективности – см. рис. 8.10.

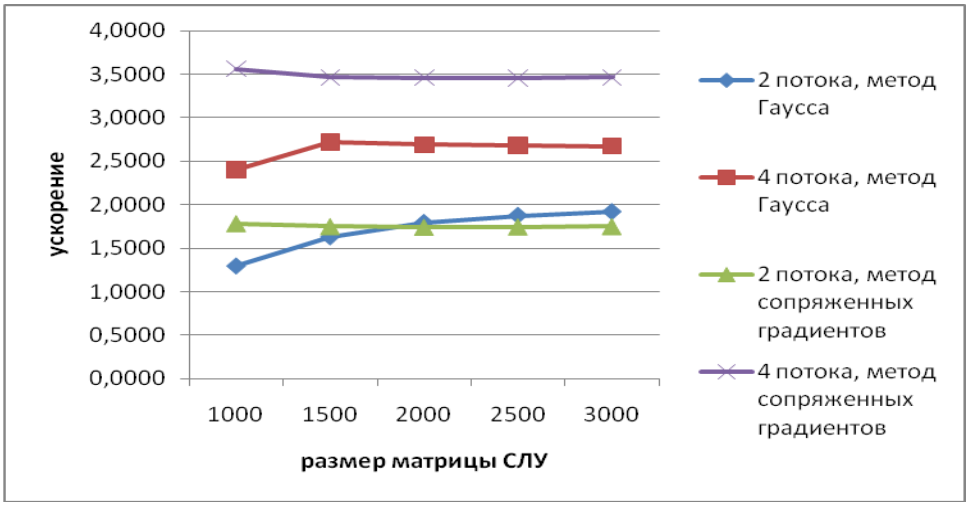

Рис. 8.10. Ускорение параллельных алгоритмов решения системы линейных уравнений в зависимости от размера матрицы

8.6. Обзор литературы

Проблема численного решения систем линейных уравнений широко рассматривается в литературе. Для освоения тематики могут быть рекомендованы работы [2–4,23,72,85]. Широкое обсуждение вопросов параллельных вычислений для решения данного типа задач выполнено в работе [44,50]. При рассмотрении вопросов программной реализации параллельных методов рекомендуется работа [45]. В данной работе рассматривается хорошо известная и широко используемая в практике параллельных вычислений программная библиотека численных методов ScaLAPACK.

8.7. Контрольные вопросы

1. Что представляет собой система линейных уравнений? Какие типы систем Вам известны? Какие методы могут быть использованы для решения систем разных типов?

2. В чем состоит задача решения системы линейных уравнений?

3. В чем идея параллельного обобщения метода Гаусса?

4. Какие показатели эффективности для параллельного варианта метода Гаусса?

6. В чем состоит схема программной реализации параллельного варианта метода Гаусса?

7. В чем идея параллельной реализации метода сопряженных градиентов?

8. Какой из алгоритмов обладает лучшими показателями ускорения?

8.8. Задачи и упражнения

1. Выполните анализ эффективности параллельных вычислений в отдельности для прямого и обратного этапов метода Гаусса. Оцените, на каком этапе происходит большее снижение показателей.

2. Выполните разработку параллельного варианта метода Гаусса при вертикальном разбиении матрицы по столбцам. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты выполненных экспериментов с ранее полученными теоретическими оценками.

3. Выполните разработку параллельных вариантов методов Якоби и Зейделя решения систем линейных уравнений (см. например, [2–3,72]). Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты выполненных экспериментов с ранее полученными теоретическими оценками.