Решение системы линейных уравнений блочным методом Холецкого

Бобровников М. Р.

1 Постановка задачи

Задача. *Найти решение системы линейных уравнений $Ax = b$. Где A — симметричная вещественнозначная матрицы размера $n \times n$, b — известный вектор размера n и x — неизвестный вектор. Искать решение будем с помощью разложения Холецкого матрицы $A = R^T DR$. Далее найдем такое y, что $R^T y = b$. Затем найдем x из условия $DRx = y$. Здесь R — верхнетреугольная матрица, D — диагональная матрица с 1 или -1 на диагонали.*

Теорема. *Пусть матрица A — самосопряженная и все ее угловые миноры отличны от нуля. Тогда существует матрица $R = (r_{ij}) \in RT(n)$ с вещественными положительными элементами на главной диагонали и диагональная матрица D с вещественными равными по модулю единице дигональными элементами такие, что $A = R^T DR$.*

Решение задачи. Применим точечный метод Холецкого для поиска матрицы R. Элементы d_{ii}, r_{ii} ,r_{ij} могут быть вычислены по следующим формулам:

$$d_{ii} = sgn(a_{ii} - \sum_{k=1}^{i-1} |r_{ki}|^2 d_{kk}),\ i = 1, ..., n, \tag{1}$$

$$r_{ii} = \sqrt{|a_{ii} - \sum_{k=1}^{i-1} |r_{ki}|^2 d_{kk}|},\ i = 1, ..., n,$$

$$r_{ij} = (a_{ij} - \sum_{k=1}^{i-1} r_{ki} d_{kk} r_{kj})/(r_{ii} d_{ii}),\ i < j,\ i, j = 1, ..., n,$$

$\square$

2 Оценка сложности в алгоритме построения верхнетреугольной матрицы в разложении Холецкого

Из формул (**??**) следует, что для вычисления элемента d_{ii}, $i = 1, ..., n$ требуется $i - 1$ мультипликативных и столько же аддитивных операций. Следовательно, вычисление всех элементов матрицы D требует $\sum_{i=1}^{n}(i - 1) = n(n - 1)/2 = O(n^2)$ мультипликативных и столько аддитивных операций.

При фиксированном $i = 1, ..., n$ вычисление элементов r_{ij} для всех $j = i + 1, ..n$ по формулам (**??**) требует $1 + \sum_{j=i+1}^{n}(i - 1) = (n - i)(i - 1) + 1$ мультипликативных и $\sum_{j=i+1}^{n}(i - 1) = (n - i)(i - 1)$ аддитивных операций. Следовательно, вычисление всех элементов матрицы R требует n операций извлечения корня, $\sum_{i=1}^{n}((n-i)(i-1)+(i-1)+1) = n^3/6+O(n^2)$ $(n \to \infty)$ мультипликативных и $\sum_{i=1}^{n}((n-i)(i-1)+(i-1)) = n^3/6+O(n^2)$ $(n \to \infty)$ аддитивных операций. Таким образом нахождение матрицы R требует $n^3/3 + O(n^2)$ $(n \to \infty)$ арифметических операций.

3 Описание блочного метода Холецкого

Разобьем матрицу A на блоки размера $m \times m$. Из формулы $A = R^T DR$ ясно, что формулы для нахождения блоков матрицы R имеют вид:

$$(R_{ii},\ D_i) = decomp(A_{ii} - \sum_{k=1}^{i-1} R_{ki}^T D_k R_{ki}),\ i = 1, ..., n, \tag{2}$$

$$R_{ij} = D_i^{-1}(R_{ii}^T)^{-1}(A_{ij} - \sum_{k=1}^{i-1} R_{ki}^T D_k R_{kj}),\ i < j,\ i, j = 1, ..., n,$$

Здесь $decomp(U)$ — функция, которая вычисляет разложение Холецкого симметричной матрицы U. Из формул (**??**) видно, что для вычислений не нужно хранить дополнительные матрицы, кроме как одну — для сохранения результата нахождения R_{ii}^{-1}. Результирующие матрицы $D_i,\ R_{ii},\ , R_{ij}$ можно хранить прямо на месте A и D.

3.1 Оценка сложности в алгоритме построения верхнетреугольной матрицы в блочном разложении Холецкого

Пусть $N = n/m$(т. е количество блоков в строке и столбце матрицы), а m нацело делит n. Сначала подсчитаем количество операций, требуемых для вычисления диагональных блоков R_{ii}. Пусть сложность вычисления произведения матриц $Mult(n) = 2n^3 - n^2$
Сложность разложения Холецкого $Chol(n) = n^3/3$
Тогда сложность вычисления блока R_{ii}: $(i-1)Mult(m) + Chol(m)$

Сложность вычисления всех диагональных блоков $R_{ii}, i \in 1, ..., N$:

$$S_1(n, m) = \sum_{i=1}^{N}((i-1)Mult(m) + Chol(m)) = n^2m - n^2/2 - 2/3nm^2 + nm/2$$

Найдем сложность вычисление всех блоков, стоящих над диагональю. Для вычисления $R_{ki}^T D_k R_{kj}$ требуется $Mult(m)$ операций. Для вычисления $A_{ij} - \sum_{k=1}^{i-1} R_{ki}^T D_k R_{kj}$ требуется $W(i, m) = \sum_{k=1}^{i-1}(Mult(m) + m^2)$ операций, т.к на каждом шаге нужно перемножать и вычитать матрицы. Умножение на треугольную матрицу требует $Y(n) = n^3$ операций. Вычисление R_{ij} : $R(i, m) = W(i, m) + Y(m) = 2im^3 - m^3 - m^2$. Тогда

$$S_1(n, m) = \sum_{i=1}^{N} \sum_{j=i+1}^{N} R(i, m) = \sum_{i=1}^{N} \sum_{j=i+1}^{N} (2im^3 - m^3 - m^2) = 1/6n(n-m)(2n-m-3)$$

Чтобы $(N-1)$ раз найти обратную к верхнетреугольной: $S_3(n, m) = (N-1)m^3/3 = (n-m)m^2/3$

Итак, нахождение всех блоков R_{ij} требует

$$S(n, m) = S_1 + S_2 + S_3 = n^3/3 - m^2n^2/6 + 2/3mn^2 - n^2 + m^3n/6 - m^2n/3 + mn - m^3/3$$

$S(n, n) = n^3/3$
$S(n, 1) = n^3/3 + O(n^2),\ (n \to \infty)$

3.2 Оптимизация хранения матриц для работы с кэшем процессора

Так как матрица A симметричная, то логично хранить не всю матрицу, а только верхнюю ее часть над главной диагональю и саму диагональ. Матрицу можно расположить в памяти так, что все блоки на диагонали будут иметь треугольный вид, а все остальные блоки будут либо квадратными размером $m \times m$ или $l \times l$, либо прямоугольными размером $m \times l$. Здесь l — размер последнего блока, равный $n - (n/m) * m$, здесь / обозначает деление нацело. Каждый блок линейно располагается в памяти. Таким образом все вычисления можно производить прямо на месте блока A_{ij} без копирования элементов блока.

4 Описание параллельного блочного метода Холецкого

Считаем, что p — количество потоков и потоки никак не мешают друг другу. В описанном выше линейном алгоритме мы находили блоки построчно. В i-ой строке сначала получаем блок на диагонали, затем находим обратную к нему обратную матрицу. Все следующие на этой строке блоки получаются из уже вычисленных блоков, находящихся только в предыдущих строках, и полученной обратной. Таким образом, все блоки вне диагонали можно искать в любом порядке в контексте текущей строки.
Предлагается следующий параллельный алгоритм, в котором каждый поток считает блоки только из своего столбца. Принадлежность столбца потоку определяется так: i-ый поток обрабатывает столбцы с номерами $i + m * p$, где $m \in \mathbb{Z}$

```
Для каждой строки i из {1,..,N}:
    Сохранение копии Temp = A_{ii}
    Синхронизация потоков
    Вычисление R_{ii}
    Вычисление R_{ii}^{-1}
    Если i = tid (mod p):
        Запись R_{ii} на место A_{ii}
    Для каждого столбца j из {i+1,..,N}:
        Если j = tid (mod p):
            Вычисление R_{ij}
            Запись R_{ij} на место A_{ij}
```

4.1 Описание параллельного решения системы $R^T y = b$

Представляем, что R^T на самом деле не транспонированная и лежит в память как R. Но работаем с ней как с транспонированной. Тогда получаем следующий алгоритм:

```
Для каждого i из {1,..,N}:
    Sum = 0
    Для каждой строки k из {1,.., i - 1}:
        Sum += R_{ki} * y[k]
    y[i] = (b[i] - Sum) / R_{ii}
```

Сделаем его параллельным. Пусть имеется p потоков. Каждый поток подсчитывает свой $y[i]$. Правило такое: если $i\%p == tid$, то поток с номером tid вычисляет $y[i]$. Также

```
void
compute_y_thread (mat a, vec y, vec b, int n, int m,
                  int tid, int p,
                  pthread_cond_t *cond,
```

```
                    pthread_mutex_t *mut,
                    int *elems_done)
{
  for (int i = 0 + tid; i < n; i += p)
    {
      double sum = 0;
      int k;
      for (k = 0; k <= i - p; k++)
        {
          sum += a[n * k + i] * y[k];
        }
      pthread_mutex_lock (mut);
      while (*elems_done < i)
        {
          pthread_cond_wait (cond, mut);
        }
      pthread_mutex_unlock (mut);
      for (; k < i; k++)
        {
          sum += a[n * k + i] * y[k];
        }
      y[i] = (b[i] - total_sum) / a[i * n + i];
      pthread_mutex_lock (mut);
      (*elems_done)++;
      pthread_cond_broadcast (cond);
      pthread_mutex_unlock (mut);
    }
}
```

Используется n точек синхронизации.

Итак, был получен параллельный алгоритм. Причем каждый поток использует только свою память, а синхронизация осуществляется за счет использования условной переменной, где условие это количество уже полученных элементов $y[i]$.

Сложность обычного алгоритма составляет

$$\sum_{i=1}^{n}\sum_{k=1}^{i} 2 = n * (n+1) = n^2 + n$$

А многопоточного, в силу того, что никакой достаточно большой участок кода не выполняется с заблокированным мьютексом

$$(n^2 + n)/p$$

Теперь получим алгоритм поиска вектора x.

```
void
compute_x_thread (mat a, vec x, vec y, vec d, int n, int m,
                  int tid, int p,
                  int *sh_thread_sum)
{
  for (int i = n - 1; i >= 0; i--)
```

```
  {
    sh_thread_sum[tid] = 0;
    double sum = 0;
    for (int k = i + 1 + tid; k < n; k += p)
      {
        sh_thread_sum[tid] += a[n * k + i] * x[k];
      }
    synchronize (p);
    if (tid == i % p)
      {
        for (int j = 0; j < p; j++)
          {
            sum += sh_thread_sum[j];
          }
        x[i] = d[i] * (y[i] - d[i] * sum) / a[n * i + i];
      }
    synchronize (p);
  }
}
```

Используется $2n$ точек синхронизации.

Здесь все потоки используют только свою память. Сложность обычного алгоритма составляет

$$\sum_{i=1}^{n}\sum_{k=1}^{i} 2 = n*(n+1) = n^2 + n$$

А многопоточного, в силу того, что никакой достаточно большой участок кода не выполняется с заблокированным мьютексом

$$(n^2 + n)/p$$

4.2 Оценка сложности в алгоритме построения верхнетреугольной матрицы в параллельном блочном разложении Холецкого

Из приведенного выше алгоритма видно, что все потоки вычисляют совпадающие диагональные блоки и обратные к ним, поэтому $S_0(n,m,p) = S_0(n,m)$ и $S_3(n,m,p) = S_3(n,m)$. Вычисление внедиагональных блоков происходит полностью параллельно: $S_1(n,m,p) = S_1(n,m)/p$ Итак,

$$S(n,m,p) = S_0(n,m) + S_3(n,m) + S_1(n,m)/p =$$

$$n^3/3p - m^2n^2/6p - mn^2/3p + mn^2 - n^2/2p - n^2/2 + m^3n/6p - m^2n/3 + mn/2p + mn/2 - m^3/3$$

Причем

$$S(n,m,1) = n^3/3 - m^2n^2/6 + 2/3mn^2 - n^2 + m^3n/6 - m^2n/3 + mn - m^3/3 = S(n,m)$$

5 Описание параллельного блочного метода Холецкого с использованием MPI

Отличие MPI версии от версии, которая использует потоки заключено в том, что из одного процесса невозможно иметь доступ ко всей матрице. Для этого будем использовать функции, которые предоставляет стандарт MPI.

5.1 Хранение матрицы в памяти

Для простоты n делится на m. Матрица представляется в виде $N = n/m$ блок-столбцов с треугольником, на гипотенузе которого расположены элементы диагонали исходной матрицы. Процесс с рангом rank считает "своими"и выделяет для них память блок-столбцы для которых выполнено $rank = I\%comm\_size$, где $I \in \{0, ..., N-1\}$

5.2 Описание алгоритма

Алгоритм разложения

```
  Для каждой строки I из {0,.., N-1}:
      Если I % commSize == rank:
      {
          memcpy (col, columns[I],
              (m * m * I + m * (m + 1) / 2) * sizeof (double))
      }
      MPI_Bcast (col, m * m * I + m * (m + 1) / 2, MPI_DOUBLE,
          I % commSize, MPI_COMM_WORLD)
      Triangle := col + m * m * I;
      R[I, I], D[I] := choletsky_decompose(Triangle)
      R[I, I]^(-1) := inverse(R[I, I])
      Если I % commSize == rank:
          A[I, I] = R[I, I]
      Для каждого столбца J из {i+1,.., N-1}:
          Если J % comm_size == rank:
              Для каждого блока K из {0,.., I-1}:
                  A[I, J] -= col[K]^T * D[K] * R[K, J]
              A[I, J] = (R[I, I]^-1)^T*D[I]*A[I, J]
```

Нахождение y

```
for (int I = 0; I < columns_n; I++)
{
  if (I % commSize == rank)
    {
      for (size_t j = 0; j < col_width; j++)
        {
          double *a = columns[I];
          double sum = 0;
          for (size_t i = 0; i < I * m; i++)
            {
              sum += a[col_width * i + j] * y[i];
            }
          a += I * m * col_width;
          for (size_t i = I * m; i < I * m + j; i++)
            {
              sum += a[get_elU (i - I * m, j, col_width)] * y[i];
            }
          y[I * m + j] =
              (b[I * m + j] - sum) / a[get_elU (j, j, col_width)];
        }
```

```
    }
  MPI_Bcast (y + I * m, col_width, MPI_DOUBLE, I % commSize,
             MPI_COMM_WORLD);
}
```

Нахождение x

```
for (size_t i = n - 1; i < n; i--)
{
  buf = gather_row (rows_p, i);
  if (rank == root)
    {
      double sum = 0;
      for (size_t j = i + 1; j < n; j++)
        {
          sum += buf[j] * x[j];
        }
      x[i] = d[i] * (y[i] - d[i] * sum) / buf[i];
    }
}
```

5.3 Объем пересылаемых данных

В разложении нужно N раз переслать данные общим размером равным размеру всей матрицы, то есть $n * (n + 1)/2$. При нахождении вектора y нужно N раз переслать данные общим размером n. При нахождении вектора x нужно n раз переслать данные общим размером $n * (n + 1)/2$.

В итоге количество пересылок равно $2N+n$, а общий объем пересылаемых данных $n^2+\frac{3}{2}n$.

5.4 Оценка количества операций

По сравнению с поточной версией количество операций не изменилось.