ALMATY TECHNOLOGICAL UNIVERSITY

Высокопроизводительные параллельные вычисления

Лекция №5
Тема: Параллельные методы умножения матрицы на вектор

Козбакова Айнур Холдасовна– PhD, профессор,
кафедра «Информационные системы»
Факультет «Инжиниринг и информационных технологии» 01.09.2025 г.

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 дают прекрасную возможность для демонстрации многих приемов и методов параллельного программирования.

В данном разделе обсуждаются методы параллельных вычислений для операции матрично–векторного умножения, в следующей главе будет рассмотрена операция перемножения матриц.

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.

Принципы распараллеливания

Для многих методов матричных вычислений характерным является повторение одних и тех же вычислительных действий для разных элементов матриц. Данный момент свидетельствует о наличии параллелизма по данным при выполнении матричных расчетов и, как результат, распараллеливание матричных операций сводится в большинстве случаев к разделению обрабатываемых матриц между потоками. Выбор способа разделения матриц приводит к определению конкретного метода параллельных вычислений; существование разных схем распределения данных порождает целый ряд параллельных алгоритмов матричных вычислений.

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

1. Ленточное разбиение матрицы. При ленточном (blockstriped) разбиении каждому потоку выделяется то или иное подмножество строк (rowwise или горизонтальное разбиение) или столбцов (columnwise или вертикальное разбиение) матрицы (3.1). 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 (3.1)

$$A = (A_0, A_1, \ldots, A_{p-1})^T, A_i = (a_{i_0}, a_{i_1}, \ldots, a_{i_{k-1}}), i_j = ik+j, 0 \leq j < k, k = m/p, \quad (3.1)$$

где ai=(ai1,ai2,...,ain), 0<=i<m, есть i –я строка матрицы A (предполагается, что количество строк m кратно числу процессоров p, т.е. m = kxp). Во всех алгоритмах матричного умножения и умножения матрицы на вектор, которые будут рассмотрены в этой и следующей лекциях, применяется разделение данных на непрерывной основе.

Другой возможный подход к формированию полос состоит в применении той или иной схемы чередования (цикличности) строк или столбцов. Как правило, для чередования

= + < =

используется число потоков p – в этом случае при горизонтальном разбиении матрица A принимает вид
$A = (A_0, A_1, \ldots, A_{p-1})^T, A_i = (a_{i_0}, a_{i_1}, \ldots, a_{i_{k-1}}), i_j = i + jp, 0 \le j < k, k = m/p,$

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

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

$$A = \begin{pmatrix} A_{00} & A_{02} & \ldots A_{0q-1} \\ & \ldots & \\ A_{s-11} & A_{s-12} & \ldots A_{s-1q-1} \end{pmatrix}, \tag{3.2}$$

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

$$A = \begin{pmatrix} a_{i_0 j_0} & a_{i_0 j_1} & \ldots a_{i_0 j_{l-1}} \\ & \ldots & \\ a_{i_{k-1} j_0} & a_{i_{k-1} j_1} & \ldots a_{i_{k-1} j_{l-1}} \end{pmatrix}, \begin{matrix} i_\nu = ik + \nu, 0 \le \nu < k, \ k = m/s, \\ j_u = jl + u, 0 \le u \le l, \ l = n/q. \end{matrix} \tag{3.3}$$

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

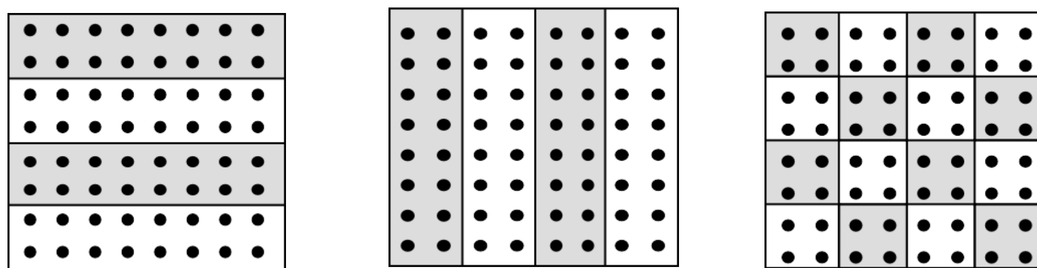

Рисунок 3.1 – Способы распределения элементов матрицымежду потоками

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

В результате умножения матрицы A размерности m*n и вектора b, состоящего из *n* элементов, получается вектор c размера m, каждый i –й элемент которого есть результат скалярного умножения i –й строки матрицы А (обозначим эту строчку a_i) и вектора b:

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j, 1 \le i \le m. \quad (3.4)$$

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

$$T_1 = m*(2n-1)$$

3.3 Последовательный алгоритм

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

```
// Программа 3.1
// Умножение матрицы на вектор
// (последовательный алгоритм)
for (i = 0; i < m; i++){
  c[i] = 0;
  for (j = 0; j < n; j++){
```

Матрично–векторное умножение – это последовательность вычисления скалярных произведений. Поскольку каждое вычисление скалярного произведения векторов длины n требует выполнения n операций умножения и n–1 операций сложения, его трудоемкость порядка $O(n)$. Для выполнения матрично–векторного умножения необходимо выполнить m операций вычисления скалярного произведения; таким образом, алгоритм имеет трудоемкость порядка $O(mn)$.

При дальнейшем изложении материала для снижения сложности и упрощения получаемых соотношений будем предполагать, что матрица A является квадратной, т. е. $m = n$.

Представим возможный вариант последовательной программы умножения матрицы на вектор.

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

// Программа 3.2

```
// Умножение матрицы на вектор – ленточное горизонтальное разбиение
// (исходный и результирующий векторы дублируются между процессами)
void main(int argc, char* argv[]) {
  double* pMatrix;  // Первый аргумент – исходная матрица
  double* pVector;  // Второй аргумент – исходный вектор
  double* pResult;  // Результат умножения матрицы на вектор
  int Size;         // Размеры исходных матрицы и вектора
  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);

  // Выделение памяти и инициализация исходных данных
  ProcessInitialization(pMatrix, pVector, pResult, pProcRows,
                        pProcResult, Size, RowNum);

  // Распределение исходных данных между процессами
  DataDistribution(pMatrix, pProcRows, pVector, Size, RowNum);

  // Параллельное выполнение умножения матрицы на вектор
  ParallelResultCalculation(pProcRows, pVector, pProcResult,
                            Size, RowNum);

  // Сбор результирующего вектора на всех процессах
```

```
        ResultReplication(pProcResult, pResult, Size, RowNum);

  // Завершение процесса вычислений
  ProcessTermination(pMatrix, pVector, pResult, pProcRows,
          pProcResult);

  MPI_Finalize();
}
```

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

RandomDataInitialization.

```
// Функция для выделения памяти и инициализации исходных данных
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, double* &pProcRows, double* &pProcResult,
  int &Size, int &RowNum) {
  int RestRows;     // Количество строк матрицы, которые еще
          // не распределены
  int i;

  if (ProcRank == 0) {
    do {
      printf("\nВведите размер матрицы: ");
      scanf("%d", &Size);
      if (Size < ProcNum) {
        printf("Размер матрицы должен превышать количество
процессов! \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 и для передачи сообщений необходимо использовать функцию MPI_Scatterv библиотеки *MPI*.

```
// Функция для распределения исходных данных между процессами
void DataDistribution(double* pMatrix, double* pProcRows,
  double* pVector, int Size, int RowNum) {
  int *pSendNum;          // Количество элементов, посылаемых процессу
  int *pSendInd;    // Индекс первого элемента данных,
```

```
                          // посылаемого процессу
int RestRows=Size;        // Количество строк матрицы, которые еще
                          // не распределены

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

// Выделение памяти для хранения временных объектов
pSendInd = new int [ProcNum];
pSendNum = new int [ProcNum];

// Определение положения строк матрицы, предназначенных
// каждому процессу
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];
}
// Рассылка строк матрицы
MPI_Scatterv(pMatrix, pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
  pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

// Освобождение памяти
delete [] pSendNum;
delete [] pSendInd;
```

```
}
```

Следует отметить, что такое разделение действий генерации исходных данных и их рассылки между процессами может быть неоправданным в реальных параллельных вычислениях при большом объеме данных. Широко используемый подход в таких случаях состоит в организации передачи сообщений 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 получается блок результирующего вектора c.

```
// Функция для вычисления части результирующего вектора

void ParallelResultCalculation(double* pProcRows, double* pVector,

  double* pProcResult, int Size, int RowNum) {

  int i, j;

  for (i=0; i<RowNum; i++) {

    pProcResult[i] = 0;

    for (j=0; j<Size; j++)

      pProcResult[i] += pProcRows[i*Size+j]*pVector[j];

  }

}
```

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

```
// Функция для сбора результирующего вектора на всех процессах

void ResultReplication(double* pProcResult, double* pResult,

int Size, int RowNum) {

  int *pReceiveNum;  // Количество элементов, посылаемых процессом

  int *pReceiveInd;  // Индекс элемента данных в результирующем

                  // векторе

  int RestRows=Size; // Количество строк матрицы, которые еще не
```

```
                // распределены
  int i;

  // Выделение памяти для временных объектов
  pReceiveNum = new int [ProcNum];
  pReceiveInd = new int [ProcNum];

  // Определение положения блоков результирующего вектора
  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];
  }
  // Сбор всего результирующего вектора на всех процессах
  MPI_Allgatherv(pProcResult, pReceiveNum[ProcRank],
    MPI_DOUBLE, pResult, pReceiveNum, pReceiveInd,
    MPI_DOUBLE, MPI_COMM_WORLD);

  // Освобождение памяти
  delete [] pReceiveNum;
  delete [] pReceiveInd;
}
```