УДК 004.43

Разработка параллельной предикатной программы решения системы линейных уравнений методом Гаусса-Жордано

Каблуков И.В. (Институт систем информатики СО РАН),

Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Описывается технология предикатного программирования для реализации параллельной программы решения системы линейных уравнений методом Гаусса-Жордано. Предикатная программа изначально параллельна и не требует распараллеливания. Описывается построение предикатной программы и ее оптимизирующая трансформация с получением эффективной параллельной императивной программы.

Ключевые слова: *параллельное программирование, трансформации программ, функциональное программирование.*

1. Введение

В настоящей работе представлена технология предикатного программирования для реализации параллельной программы решения системы линейных уравнений методом Гаусса-Жордано. В течение десяти лет на данной программе демонстрируется техника работы с массивами в магистерском курсе НГУ «Предикатное программирование». Стимулом публикации послужила работа [4] с описанием реализации асинхронно-потоковой программы алгоритма разложения Холецкого для вычисления квадратного корня матрицы. Схема этого алгоритма во многом похожа на схему Гаусса-Жордано.

Эффективность предикатных программ достигается применением *оптимизирующих трансформаций*. Они определяют отличную от классической оптимизацию среднего уровня с переводом предикатной программы в эффективную императивную программу.

В предикатном программировании нет распараллеливания. Предикатная программа изначально параллельна. В языке предикатного программирования P имеются параллельный оператор и конструкция векторного параллелизма «определение массива». Задача лишь в том, чтобы донести исходный параллелизм до конечной императивной программы.

Во втором разделе дается краткое описание языка P [3]. В третьем разделе подробно описывается построение предикатной программы решения системы линейных уравнений методом Гаусса-Жордано. Далее представлена последовательность оптимизирующих трансформаций с получением эффективной параллельной программы. Природа параллелизма и особенности реализации параллельных программ исследуются в разделе 5. В разделе 6 проводится сравнение с подходами потокового параллелизма. Заключение содержит замечания по реализации параллельных программ для разных платформ.

2. Предикатное программирование

Предикатная программа является предикатом (логической формулой) в форме вычислимого оператора. Полная предикатная программа состоит из набора рекурсивных *предикатных программ* на языке P [3] следующего вида:

```
<имя программы>(<описания аргументов>: <описания результатов>)
pre <предусловие>
post <постусловие>
{ <оператор> }
```

Необязательные конструкции предусловия и постусловия являются формулами на языке исчисления предикатов. Они используются для улучшения понимания программ и для дедуктивной верификации [6, 10, 19].

Ниже представлены основные конструкции языка P: оператор присваивания, блок (оператор суперпозиции), параллельный оператор, условный оператор, вызов программы и описание переменных, используемое для аргументов, результатов и локальных переменных.

```
<переменная> = <выражение>
{<оператор1>; <оператор2>}
<оператор1> || <оператор2>
if (<логическое выражение>) <оператор1> else <оператор2>
<имя программы>(<список аргументов>: <список результатов>)
<тип> <пробел> <список имен переменных>
```

Вызов вида H(x: y), где x – набор выражений, а y – набор переменных, эквивалентен оператору присваивания y = H(x). Здесь вызов программы H представлен в форме вызова функции H(x).

В языке P имеется развитая система типов: подтипы, структуры, множества, алгебраические типы, предикатные типы, массивы. Значением типа **array**(E, J) является *массив* с *элементами массива* типа E и *индексами* конечного типа J. Тип массива является предикатным типом, его значения (массивы) являются тотальными и однозначными предикатами. Массив всегда вычисляется полностью для всех его элементов. Для элемента

массива используется привычная форма записи A[j] вместо функциональной формы A(j); соответствующая предикатная форма есть A(j**:** z).

Операции с массивами определяются через *оператор каррирования* A(j**:** z) {B(x, j**:** z)} [12], где аргументы j и результаты z являются формальными параметрами, а оператор B(x, j**:** z) – телом определяемой предикатной программы A. Результатом исполнения оператора каррирования является новая предикатная программа A(j**:** z), получаемая фиксацией текущих значений переменных набора x в операторе B(x, j**:** z). В случае, когда A – массив, дополнительно происходит вычисление всех элементов массива A.

В языке P оператор каррирования для массива A записывается в виде оператора присваивания A = **for** (j) {B(x, j)}. Конструкция в правой части оператора – *определение массива*. Значением определения массива является новый массив, в котором для всякого допустимого индекса j соответствующий элемент массива равен значению выражения B(x, j). Вычисления элементов массива для разных индексов j могут быть реализованы параллельно. Это векторный параллелизм, который становится матричным в случае, когда j – пара индексов. Рассмотрим пример:

```
type Vec = array (int, 1..5);
Vec  A;
A =  for (j) { j*j };
```

Здесь описывается массив A с элементами типа **int** и индексами от 1 до 5. Результатом исполнения данного фрагмента является массив A со значениями элементов: 1, 4, 9, 16, 25.

Определение части массива представляется следующим оператором каррирования:

A(j**:** z) { **if** (m(x)≤j & j≤n(x)) B(x, j**:** z) **else** C(j**:** z) } .

В языке P данная конструкция записывается в виде оператора:

A = C **for** (j) {**case** m(x)..n(x)**:** B(x, j) } .

При исполнении реализуется замена части массива C в диапазоне индексов от m(x) до n(x). Модифицированное значение массива присваивается массиву A. При этом исходный массив C остается неизменным. В случае, когда m(x) и n(x) совпадают, вместо диапазона m(x)..n(x) указывается просто m(x). Если j – пара индексов, то диапазоны могут быть по обоим индексам; в этом случае они разделяются запятой.

В определении части массива может быть несколько **case**-частей. Например, оператор A = C **for** (j) { **case** m..n**:** B(x, j) **case** p..k**:** D(x, j) } является эквивалентом оператора A = (C **for** (j) { **case** m..n**:** B(x, j) }) { **case** p..k**:** D(x, j) } .

Введем понятие *гиперфункции* – программы с несколькими *ветвями* результатов [3, разд. 3.1, 3.3]. Гиперфункция A(x**:** y**:** z) имеет две ветви результатов y и z. Исполнение гиперфункции завершается одной из ветвей с вычислением результатов по этой ветви; результаты других ветвей не вычисляются. *Вызов гиперфункции* записывается в виде A(x**:** y #M1**:** z #M2). Здесь M1 и M2 – метки программы, содержащей вызов. Операторы перехода #M1 и #M2 встроены в ветви вызова. Исполнение вызова либо завершается первой ветвью с вычислением y и переходом на метку M1, либо второй ветвью с вычислением z и переходом на метку M2.

Эффективность предикатных программ достигается применением следующих оптимизирующих трансформаций [2], переводящих программу на императивное расширение языка **P**:

- замена хвостовой рекурсии циклом;
- подстановка тела программы на место ее вызова;
- склеивание переменных: замена всех вхождений одной переменной на другую переменную;
- кодирование алгебраических типов (списков и деревьев) с помощью массивов и указателей.

Эффективность программы также обеспечивается оптимизацией, реализуемой программистом, на уровне предикатной программы. Для приведения рекурсии к хвостовому виду применяется метод обобщения исходной задачи. Далее обычно открывается возможность проведения серии последующих улучшений алгоритма. Итоговая программа по эффективности не уступает написанной вручную и, как правило, короче [1, 6, 10, 11, 19].

3. Программа решения системы линейных уравнений методом Гаусса-Жордано

Рассматривается классический алгоритм решения системы линейных уравнений метода Гаусса-Жордано. Это метод исключения переменных с приведением к эквивалентной системе треугольного вида. Метод работает, если не появляется нулей на главной диагонали. Здесь представлен более общий алгоритм, который успешно «борется» с нулями и, в частности, распознает вырожденность системы уравнений.

Пусть имеется система линейных уравнений:

$$\mathsf{LinEq(n,\ a,\ b,\ x)} \ \cong \ \sum_{j=1}^{n} a_{ij}x_j = b_i, i = 1..n$$

Здесь a – квадратная матрица размерности n×n, b – вектор свободных членов, x – вектор неизвестных переменных, значения которых требуется вычислить.

Для построения спецификации задачи необходимо формализовать предикат LinEq. Определим типы вектора и матрицы:

```
nat n;
type I1n = 1..n;
type VEC = array (real, I1n);
type MATR = array (real, I1n, I1n);
```

Переменная n, являющаяся параметром задачи, определена здесь как глобальная, что позволяет опускать ее в программах, формулах и параметрических типах I1n(n), VEC(n) и MATR(n). Введем вспомогательную формулу подсчета суммы значений некоторой функции f, заданной параметром, для аргументов от 1 до k:

```
formula SUM(nat k, predicate(nat: real) f: real) = k=0? 0: SUM(k - 1, f) + f(k);
```

Наконец, формула LinEq определяет набор равенств, соответствующий системе линейных уравнений:

```
formula LinEq(MATR a, VEC b, x) =
        forall i=1..n. SUM( n, predicate(nat j: real) {a[i, j] * x[j]} ) = b[i];
```

Здесь, конструкция **predicate**(**nat** j**: real**) {a[i, j] * x[j]} литерально определяет функцию f(j) = a[i, j] * x[j].

Представленная ниже программа Lin является гиперфункцией. Определяется, вырождена ли матрица a. В случае невырожденной матрицы решается система уравнений – вычисляется вектор неизвестных x. При этом реализуется первая ветвь гиперфункции. Если обнаруживается, что матрица a вырождена, то завершение гиперфункции Lin реализуется по второй ветви.

```
Lin(MATR a, VEC b : VEC x : )
pre  n > 0
pre 1: det(a) != 0
post 1: LinEq(a, b, x)
{       TriangleMatr(a, b:  MATR c, VEC d :  #2);
        Solution(c, d:  VEC x ) #1
}
```

В гиперфункции Lin две ветви результатов, причем у второй ветви (после второго двоеточия в первой строке) результатов нет. Условие n > 0 есть общее предусловие программы. Предусловие первой ветви det(a) != 0 определяется как ненулевой детерминант

матрицы a. Постусловие по первой ветви – предикат LinEq(n, a, b, x). Вторая ветвь не имеет постусловия, так как у нее нет результатов.

Гиперфункция TriangleMatr реализует приведение исходной системы уравнений a*x = b к эквивалентной системе уравнений c*x = d с треугольной матрицей c. Если в процессе преобразования матрицы a обнаруживается ее вырожденность, то исполнение гиперфункции завершается ветвью 2. Оператор перехода #2 во второй ветви вызова TriangleMatr определяет завершение гиперфункции Lin второй ветвью.

Программа Solution реализует решение системы c*x = d для треугольной матрицы c. Оператор перехода #1 после вызова Solution определяет завершение программы Lin первой ветвью.

Определим спецификацию программы TriangleMatr. Условие того, что новая система уравнений c*x = d эквивалентна исходной невырожденной системе уравнений a*x = b, представлено формулой postLin.

```
formula postLin(MATR a, VEC b, MATR c, VEC d) =
    forall VEC x, y. LinEq(a, b, x) & LinEq(c, d, y) ⇒ x = y
```

Формула triangle определяет второе условие: матрица c является верхней треугольной, т.е. все элементы главной диагонали равны 1, а элементы ниже главной диагонали – нулевые.

```
formula triangle(MATR c) =
    (forall i=1..n. c[i, i] = 1) & forall i,j=1..n. (i > j ⇒ c[i, j] = 0)
```

Программа TriangleMatr представлена ниже.

```
TriangleMatr(MATR a, VEC b: MATR a', VEC b' :  );
pre  n > 0
pre 1: det(a) != 0
post 1: postLin(a, b, a', b') & triangle(a')
{       Jord(a, b, 1:  a', b' #1 :  #2) };
```

Здесь применяется метод обобщения исходной задачи TriangleMatr. Рассмотрим более общую задачу Jord, в которой имеется дополнительный аргумент k – номер текущей строки матрицы a. Формула triaCol вводит дополнительное предусловие в программе Jord: матрица a триангулирована для столбцов от 1 до k-1, т.е. элементы главной диагонали до k-1 равны единице, а элементы ниже этих диагональных элементов равны нулю, см. Рис.1.

```
formula triaCol(nat k, MATR a) =
    forall i=1..k-1. (a[i, i] = 1) & forall i = 1..n, j=1..k-1. (i > j ⇒ a[i, j] = 0);
```

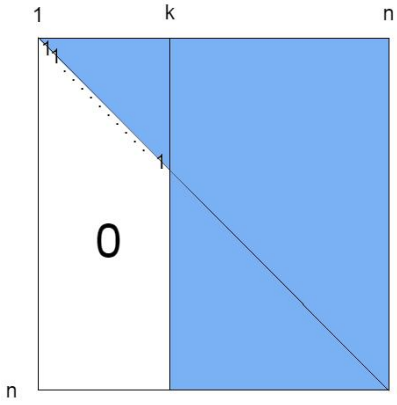

Рис.1

Задача TriangleMatr сводится к задаче Jord при k = 1.

Программа гиперфункции Jord приведена ниже.

```
Jord(MATR a, VEC b, nat k: MATR a', VEC b' : )
pre 1 ≤ k ≤ n & triaCol(k, a)
pre 1: det(a) != 0
post 1: postLin(a, b, a', b') & triangle(a')
{       diagEl(a, b, k: MATR c, VEC d : #2);
        norm(k, c, d: MATR e, VEC f);
        subtrLines(k, e, f: MATR g, VEC h);
        if (k = n) { a' = g || b' = h #1 }
        else   Jord(g, h, k+1:  a', b' #1 : #2)
};
```

Программа Jord реализует следующую последовательность действий. Вызов программы diagEl обеспечивает истинность условия a[k, k] ≠ 0, обменивая k-ю строку матрицы a с последующими строками. Если удается обеспечить a[k, k] ≠ 0, то вызов программы norm преобразует систему уравнений к эквивалентной системе, в которой a[k, k] = 1. Вызов subtrLines реализует вычитание k-й строки из последующих строк матрицы так, чтобы занулить k-й столбец. Если k<n, реализуется рекурсивный вызов программы Jord с параметром k+1.

Цель гиперфункции diagEl – обеспечить истинность условия a[k, k] ≠ 0. Если a[k, k]=0, делается попытка обменять k-ю строку матрицы a с одной из последующих строк. Если во всех строках нулевые элементы, то матрица a вырождена, и тогда программа завершается второй ветвью гиперфункции.

```
diagEl(MATR a, VEC b, nat k: MATR a', VEC b': )
pre  1 ≤ k ≤ n & triaCol(k, a)
pre 1: det(a) != 0
post 1: a'[k, k] != 0 & postLin(a, b, a', b')
{       if (a[k, k] != 0) { a' = a || b' = b #1}
        else perm(a, b, k, k+1: a', b' #1 : #2)
}
```

Гиперфункция perm является обобщением diagEl. Имеется параметр m, причем m>k. Дополнительное предусловие: элементы k–го столбца в строках от k до m-1 – нулевые. Программа perm обменивает k–ю строку с одной из строк от m до n, если такое возможно.

```
perm(MATR a, VEC b, nat k, m: MATR a', VEC b' : )
pre  1 ≤ k ≤ n & k<m ≤ n+1 & triaCol(k, a) & forall i = k..m-1. a[i, k] = 0
pre 1: det(a) != 0
post 1: a'[k, k] != 0 & postLin(a, b, a', b')
{       if (m>n) #2
        else if (a[m, k] != 0) {
                b' = b for (I1n j) {case k: b[m] case m: b[k]} ||
                a' = a for (I1n i, j) {case (k, k..n): a[m, j] case (m, k..n): a[k, j]}
                #1
        } else  perm(a, b, k, m + 1: a', b' #1 : #2)
}
```

Программа norm приводит диагональный элемент a[k, k] к единице путем деления k–ой строки на этот диагональный элемент:

```
norm(nat k, MATR a, VEC b: MATR a', VEC b')
pre  1 ≤ k ≤ n & a[k, k] != 0 &  triaCol(k, a)
post a'[k, k] = 1 & postLin(a, b, a', b')
{       b' = b for (I1n j) { case k: b[k] / a[k, k] } ||
        a' = a for (I1n i, j) { case (k, k): 1 case (k, k+1..n): a[k, j] / a[k, k]}
}
```

Программа subtrLines приводит к нулю все элементы в k–ом столбце ниже главной диагонали путем вычета k–ой строки из всех нижележащих строк с подходящими множителями:

```
subtrLines(nat k, MATR a, VEC b: MATR a', VEC b')
pre  1 ≤ k ≤ n & a[k, k] = 1 &  triaCol(k, a)
post triaCol(k+1, a) & postLin(a, b, a', b')
{ b' = b for (I1n i) {case k+1..n: b[i] - b[k] × a[i, k]} ||
  a' = a for (I1n i, j) {case (k+1..n, k): 0 case (k+1..n, k+1..n): a[i, j] - a[k, j] × a[i, k]}
}
```

Фрагмент **case** (k+1..n, k): 0 специально выделен отдельно из общего случая.

Программа Solution находит решение системы уравнений a*x = b для верхнетреугольной матрицы a.

```
Solution(MATR a, VEC b: VEC x )
pre  n > 0 & triangle(a)
post LinEq(a, b, x)
{       uniMat(a, b, n: x ) }
```

Решение реализуется последовательным приведением к нулю всех элементов матрицы a выше главной диагонали. Применяется метод обобщения исходной задачи Solution, которая сводится к более общей задаче uniMat. Имеется параметр k – номер текущего столбца матрицы a. Формула nulColl вводит дополнительное предусловие в программе uniMat: в матрице a все элементы выше главной диагонали с индексами от k+1 до n равны нулю.

```
formula nulColl(nat k, MATR a) = forall j=k+1..n, i=1..j-1. a[i, j] = 0;
```

Программа uniMat последовательно обнуляет столбцы матрицы выше главной диагонали, начиная с последнего столбца.

```
uniMat(MATR a, VEC b, nat k: VEC x )
pre  n > 0 & triangle(a) & nulCol(k, a)
post LinEq(a, b, x)
{       if (k = 1) x = b
        else { subtrCol(a, b, k : MATR c, VEC d);
                uniMat(c, d, k-1: x )
        }
}
```

Для очередного k–го столбца программа subtrCol обнуляет все элементы выше главной диагонали.

```
subtrCol(MATR a, VEC b, nat k: MATR a', VEC b')
pre  n > 0 & triangle(a) & nulCol(k, a)
post nulCol(k-1, a) & postLin(a, b, a', b')
{       b' = b for (I1n i) {case 1..k-1: b[i] - b[k] × a[i, k]} ||
        a' = a for (I1n i, j) {case (1..k-1, k): 0 }
}
```

4. Оптимизирующие трансформации

Эффективная императивная программа получается из предикатной программы применением системы оптимизирующих трансформаций. Определим набор трансформаций для программы решения системы линейных уравнений Lin.

На первом этапе применяются трансформации склеивания переменных, реализуемых применением команд вида x ← y, z, задающих замену всех вхождений переменных y и z на

переменную x. Корректность склеивания обеспечивается на основе потокового анализа программы [2].

На втором этапе проводится замена хвостовой рекурсии циклом **for**. Также реализуются упрощения. Результатом проведения двух этапов трансформации является следующая программа:

Склеивания: a ← c; b ← d, x;

```
Lin(MATR a, VEC b:  b : )
{       TriangleMatr(a, b:  a, b :  #2);
        Solution(a, b:  b ) #1
}
```

Склеивания: a ← a'; b ← b';

```
TriangleMatr(MATR a, VEC b:  a, b :  );
{       Jord(a, b, 1:  a, b #1:  #2) }
```

Склеивания: a ← c, e, g, a'; b ← d, f, h, b';

```
Jord(MATR a, VEC b, nat k:  a, b : )
{       for(;;) {
                diagEl(a, b, k: a, b : #2);
                norm(k, a, b: a, b);
                subtrLines(k, a, b: a, b);
                if (k = n) #1
                else  k = k + 1
        }
}
```

Склеивания: a ← a'; b ← b';

```
diagEl(MATR a, VEC b, nat k:  a, b : )
{       if (a[k, k] != 0) #1
        else perm(a, b, k, k+1: a, b #1 | #2)
}
```

Склеивания: a ← a'; b ← b';

```
perm(MATR a, VEC b, nat k, m:  a,  b : )
{       for(;;) {
            if (m>n) #2
            else if (a[m, k] != 0) {
               b = b for (I1n j) {case k: b[m] case m: b[k]} ||
               a = a for (I1n i, j) {case (k, k..n): a[m, j] case (m, k..n): a[k, j]}
               #1
            } else  m = m + 1
        }
}
```

Склеивания: a ← a'; b ← b';

```
norm(nat k, MATR a, VEC b:  a,  b)
{       b = b for (I1n j) {case k: b[k] / a[k, k] } ||
        a = a for (I1n i, j) { case (k, k): 1 case (k, k+1..n): a[k, j] / a[k, k]}
}
```

Склеивания: a ← a'; b ← b';

```
subtrLines(nat k, MATR a, VEC b:  a,  b)
{ b = b for (I1n i) {case k+1..n: b[i] - b[k] × a[i, k]} ||
  a = a for (I1n i, j) { case (k+1..n, k): 0 case (k+1..n, k+1..n): a[i, j] - a[k, j] × a[i, k]}
}
```

Склеивание: b ← x;

```
Solution(MATR a, VEC b:   b )
{       uniMat(a, b, n:  b ) }
```

Склеивания: a ← c; b ← d, x;

```
uniMat(MATR a, VEC b, nat k:   b )
{       for(;;) {
                if (k = 1) return;
                subtrCol(a, b, k : a, b);
                k = k - 1
        }
}
```

Склеивания: a ← a'; b ← b';

```
subtrCol(MATR a, VEC b, nat k:  a,  b)
{       b = b for (I1n i) {case 1..k-1: b[i] - b[k] × a[i, k]} ||
        a = a for (I1n i, j) {case (1..k-1, k): 0 }
}
```

На третьем этапе, после устранения рекурсии, проводится открытая подстановка тел программ на места вызовов программ. Подстановка программ реализуется в следующей последовательности:

perm → diagEl;
subtrLines, norm, diagEl → Jord → TriangleMatr → Lin;
subtrCol → uniMat → Solution → Lin;

Результатом указанной серии подстановок является следующая программа.

```
Lin(MATR a, VEC b:  b : )
{       for(nat k = 1; ; k=k+1) {
                if (a[k, k] != 0) #M1;
                nat m;
                for(m = k +1;;m = m + 1) {
                        if (m>n) #2;
                        if (a[m, k] != 0) break
                };
                { b = b for (I1n j) {case k: b[m] case m: b[k]} ||
                  a = a for (I1n i, j) {case (k, k..n): a[m, j]  case (m, k..n): a[k, j]}
                };
M1:             { b = b for (I1n j) {case  k: b[k] / a[k, k] }||
                  a = a for (I1n i, j) { case (k, k): 1 case (k, k+1..n): a[k, j] / a[k, k]}
                };
                { b = b for (I1n i) {case  k+1..n: b[i] - b[k] × a[i, k]} ||
                  a = a for (I1n i, j) {case (k+1..n, k): 0
                                        case (k+1..n, k+1..n): a[i, j] - a[k, j] × a[i, k]}
                };
                if (k = n) break
        }
        for(nat k = n; k != 1; k = k - 1) {
                b = b for (I1n i) {case 1..k-1: b[i] - b[k] × a[i, k]} ||
                a = a for (I1n i, j) {case (1..k-1, k): 0 }
        }
        #1
}
```

На четвертом этапе раскрываются операции с массивами. Для формы вида a = a **for** каждая **case**–часть определяет независимый параллельный цикл. За исключением первых двух форм, реализующих обмен элементов. Там возникает групповой оператор присваивания, который раскрывается с помощью дополнительной переменной. Получаем итоговую программу на императивном расширении языка **P**:

```
Lin( MATR a, VEC b:  b : )
{       for(nat k = 1; ; k=k+1) {
                if (a[k, k] != 0) #M1;
                nat m;
                for(m = k +1;;m = m + 1) {
                        if (m>n) #2;
                        if (a[m, k] != 0) break
                }
                {  | b[k], b[m] | = | b[m], b[k] | ||
                   for (nat j= k..n) | a[k, j], a[m, j] | = | a[m, j], a[k, j] |
                }
M1:             {  b[k] = b[k] / c[k, k] ||
                   a[k, k] = 1 ||
                   for (nat j= k+1..n) a[k, j] = a[k, j] / a[k, k]
                }
                {  for (nat i= k+1..n) b[i] = b[i] - b[k] × a[i, k] ||
                   for (nat j= k+1..n, k) a[k, j] =0 ||
                   for (nat i= k+1..n, nat j= k+1..n) a[i, j] = a[i, j] - a[k, j] × a[i, k]
                };
                if (k = n) break
        }
        for(nat k = n step -1 to 2) {
                for (nat i=1..k-1) b[i] = b[i] - b[k] × a[i, k] ||
                for (nat i=1..k-1) a[i, k] = 0
        }
        #1
}
```

Такие операторы (отмеченные красным цветом), как обнуление k-го столбца, ранее в программах subtrLines и subtrCol, и присваивание a[k, k] = 1, являются избыточными, поскольку соответствующие элементы матрицы a не используются в дальнейших вычислениях. Удаление указанных фрагментов программы – нетривиальная трансформация, требующая специального потокового анализа на элементах матрицы.

Все циклы, кроме полученных устранением рекурсии, допускают параллельное исполнение. Наиболее значимым является оператор цикла:

```
for (nat i= k+1..n, nat j= k+1..n) a[i, j] = a[i, j] - a[k, j] × a[i, k]
```

Здесь проводится наибольший объем вычислений. Данный цикл допускает реализацию в форме матричного параллелизма.

5. Параллелизм предикатных программ

Распараллеливание программ не является самоцелью – оно применяется исключительно в целях повышения быстродействия программ.

Классом предикатных программ является класс программ-функций [8], не взаимодействующих с внешним окружением программы. В предикатных программах нет физического параллелизма, который возможен в автоматных программах [5, 9, 13], являющихся продолжением параллельных физических процессов, протекающих в разных точках пространства и осуществляющих обмен информацией. Но есть параллелизм другого рода, логической природы.

5.1. Источники параллелизма

Параллелизм предикатных программ имеется в языке P_0 [12], из которого построен язык предикатного программирования P [3]. Источником параллелизма в языке P_0 являются два оператора: параллельный оператор и оператор каррирования.

Параллельный оператор B(x: y) || C(x: z) допускает параллельное независимое исполнение подоператоров B(x: y) и C(x: z). При этом исполнение B(x: y) и исполнение C(x: z) никак между собой не взаимодействуют, поскольку наборы y и z не содержат общих переменных. Ввиду тождества:

$$\{B(x: y) \,||\, C(x: z)\};\ F(y, z: u) \equiv F(B(x), C(x): u)$$

параллелизм присутствует при вычислении значений разных аргументов в вызове предиката или функции. В частности, параллелизм возможен при вычислении аргументов любой бинарной операции, например, плюс «+».

Если предикатная программа транслируется на язык Си или C++, параллельный оператор кодируется в виде последовательного оператора. В подавляющем большинстве случаев издержки на организацию параллельных процессов превышают выигрыш от распараллеливания.

Параллелизм простых операторов может быть частично учтен в архитектуре широких команд вычислительных систем «Эльбрус» [14]. Адекватная реализация параллельного оператора возможна лишь на уровне интегральных схем или его аналога ПЛИС (FPGA).

Вторым источником параллелизма (векторного и матричного) является оператор каррирования [12] – аналог лямбда-функции в языках функционального программирования. Оператор каррирования для массива записывается в виде конструкций: определение массива и определение частей массива. Оптимизирующая трансформация этих конструкций дает циклы с векторным или матричным параллелизмом.

Параллелизм, реализуемый параллельным оператором и определением массива, будем называть *логическим параллелизмом* в противовес физическому параллелизму в автоматных программах.

5.2. В классе программ-функций только логический параллелизм. Распараллеливания нет в предикатном программировании

Оператор суперпозиции, условный оператор, параллельный оператор и конструкция «определения массива» определяют базис языка P и являются естественными формами построения алгоритма [7]. Параллельный оператор и определение массива являются двумя формами логического параллелизма предикатных программ. Предикатная программа не нуждается в распараллеливании. Она изначально параллельна. Задача реализации в том, чтобы донести этот параллелизм до конченой программы, получаемой в результате применения набора оптимизирующих трансформаций.

Универсальность предикатного программирования можно сформулировать следующим тезисом. Если для некоторой математической задачи, постановку которой можно формализовать в виде предусловия и постусловия, существует алгоритм решения задачи, то этот алгоритм можно записать в виде предикатной программы.

Апробация предикатного программирования на большом числе задач показала, что построение предикатных программ не только потенциально возможно. Технология предикатного программирования имеет существенные преимущества перед традиционной технологией императивного программирования.

Справедлив другой тезис. Для всякой математической задачи, постановку которой можно формализовать в виде предусловия и постусловия, и для всякой императивной программы, которая решает данную задачу, существует предикатная программа, решающая данную задачу, причем императивная программа получается из предикатной программы применением некоторого набора оптимизирующих трансформаций.

Отсюда следует чисто математическая природа всех программ из класса программ-функций, т.е. класса невзаимодействующих программ. Всякой программе соответствует некоторый предикат, т.е. вычислимая логическая формула. Эта формула композируется из базисных операторов языка P_0. Среди композиций – параллельный оператор и определение массива. Другого параллелизма нет. При этом фрагменты программ, которые могут исполняться параллельно, между собой не взаимодействуют.

Итак, логический параллелизм полностью характеризует параллелизм не только предикатных, но и императивных программ для всего класса задач дискретной и вычислительной математики.

5.3. Параллельное и конкурентное программирование

Мы противопоставляем логический и физический параллелизмы программ. Логический параллелизм реализуется в классе программ-функций. Это самый большой класс программ, содержащий, в частности, программы для задач дискретной и вычислительной математики.

Физический параллелизм может присутствовать в программах класса программ-процессов [8] (или реактивных систем), реализующих постоянное взаимодействие с окружением программы. Этот класс также называется классом *автоматных программ*. Важнейшим подклассом является класс контроллеров систем управления. Исполнение автоматной программы и ее параллельных подпрограмм инициируется различными физическими процессами через сообщения и датчики в окружении программы. Разные физические процессы обычно протекают параллельно. Соответственно, параллельно исполняются различные инициируемые ими подпрограммы. Эти подпрограммы могут обращаться к общим данным, что может стать причиной конфликта по доступу.

Если предикатной программе, а также императивной программе для класса задач вычислительной математики соответствует предикат, то автоматной программе соответствует конечный автомат [5, 9, 13]. Различная природа предикатных и автоматных программ предопределяет разные технологии программирования для этих классов программ. Язык автоматного программирования универсален и позволяет закодировать любой алгоритм. Но не следует использовать его для программ-функций – программа станет существенно сложнее. В частности, не рекомендуется программировать в автоматном стиле чисто вычислительные блоки, появляющиеся в составе контроллеров систем управления. Различные особенности технологии автоматного программирования описаны в работе [5, разд.4].

Параллельное программирование для автоматных программ – неудачный и неадекватный перевод для concurrent programming. Это вызывает путаницу, в частности, является причиной того, что параллельное программирование для вычислительной математики и concurrent programming рассматриваются вместе под общим заголовком «в области параллельного и распределенного программирования» (стр.24) в Программе фундаментальных исследований:

http://www.ras.ru/FStorage/Download.aspx?id=62d335ba-2aea-4803-85ee-fd0cd37aba4b

В последнее время, наряду с термином «распределенное программирование» часто используется «конкурентное программирование».

6. Сравнение с подходами потокового параллелизма

Кризис программирования 1960-ых гг. не был успешно преодолен. Появившиеся новые технологии и языки программирования лишь смягчили остроту этого кризиса. В этом причина затянувшейся стагнации в области программной инженерии. Особая острота кризиса программирования в параллельном программировании. Большое число архитектур вычислительных машин, языков программирования и технологий. Высокая сложность и трудоемкость программирования, проблемы надежности. Дороговизна разработки новых архитектур. Все это отмечалось в многочисленных обзорах по параллельному программированию.

В обзоре [4] рассматриваются языки и системы потокового параллелизма с ориентацией на несколько разных платформ. Как средство решения проблем эффективности и удобства программирования поставлена задача построения единого универсального максимально абстрактного языка для математического описания алгоритмов, с которого можно автоматически (с возможными подсказками) транслировать в разные языки существующих платформ.

Дополнительный интерес к работе [4] вызван рассмотрением там алгоритма разложения Холецкого для вычисления квадратного корня матрицы. Схема этого алгоритма во многом похожа на схему Гаусса-Жордано, используемую в нашей работе. А это значит, что алгоритм Гаусса-Жордано можно было бы записать на языке UPL(G) [4] в виде асинхронно-потоковой реализации аналогично алгоритму разложения Холецкого в работе [4]. В асинхронно-потоковой модели результаты промежуточных вычислений не записываются в память, а доставляются непосредственно аргументам для всех операторов программы, где эти результаты далее используются. Памяти нет – данные прикрепляются непосредственно к аргументам операторов. Оператор срабатывает лишь при поступлении всех его аргументов.

Интригующим было бы сравнение алгоритма Гаусса-Жордано с применением матричного параллелизма и асинхронно-потоковой реализации этого алгоритма на языке UPL(G). Однако, несмотря на многочисленные попытки в мире, до сих пор нет успешной асинхронно-потоковой архитектуры.

Эффективность алгоритма Гаусса-Жордано зависит главным образом от эффективности реализации цикла (см. конец разд. 4):

for (**nat** i= k+1..n, **nat** j= k+1..n) a[i, j] = a[i, j] - a[k, j] × a[i, k]

Предельная эффективность здесь достигается применением специализированного матричного процессора. Единственный способ достичь подобного результата в асинхронно-потоковой архитектуре, например для языков Пифагор [16, 17] или Sisal [15], это суметь положить параллельные списки Пифагора или потоки Sisal'а на матричный процессор. Использовать декларированные преимущества неявного параллелизма здесь не удастся. Очередной вычисленный элемент a[i, j] весьма проблематично упреждающе использовать на следующих итерациях цикла по k. В частности, необходимо гарантировать, что строка матрицы, содержащая a[i, j], не будет обмениваться. Кроме того, надо не попасть в цикл нормирования по диагональному элементу. Необходимо также обеспечить доступность элементов a[k, j] и a[i, k]. В принципе, подобное реализуемо применением изощренного потокового анализа, однако вряд ли обеспечит приемлемую эффективность.

7. Заключение

В настоящей работе в технологии предикатного программирования представлена реализация параллельной программы решения системы линейных уравнений по схеме Гаусса-Жордано. Без распараллеливания. Поскольку предикатная программа изначально параллельна благодаря наличию в языке P параллельного оператора и определения массива. Задача лишь в том, чтобы донести исходный параллелизм до конечной императивной программы.

В языке P не предусмотрено других специальных средств реализации распараллеливания. Разумеется, следует выбрать адекватный алгоритм, соответствующий эффективной реализации на целевой платформе. Например, для параллельного суммирования массива чисел необходимо выбрать подходящую модификацию каскадной схемы суммирования. Но при этом программа остается чисто предикатной, не отягощенной специальными конструкциями подобно языкам в работе [4], в частности, позволяет провести дедуктивную верификацию.

Реализация параллелизма проводится полностью на стадии оптимизирующей трансформации предикатной программы. Реализация с выходом на пакеты MPI и OpenMP не представляет особой сложности. Реализация для ПЛИС [18] и интегральных схем потребует применения более мощного и глубокого потокового анализа трансформируемой программы. В частности, необходимо будет реализовать протяжку диапазонов значений скалярных переменных [18]. Для реализации на ПЛИС выбирается соответствующий алгоритм. Предикатная программа определенным образом структурируется с ориентацией на ПЛИС.

При реализации на базе языка Си структуризация обычно реализуется автоматически. Однако в предикатном программировании оптимизации подобного рода возлагаются на программиста. В дальнейшем следует исследовать возможность реализации специальных структурирующих трансформаций.

Работа выполнена при поддержке РФФИ, грант № 16-01-00498.

Список литературы

1. Вшивков В.А., Маркелова Т.В., Шелехов В.И. Об алгоритмах сортировки в методе частиц в ячейках // Научный вестник НГТУ. 2008. Т. 4 (33). С. 79-94.
2. Каблуков И.В., Шелехов В.И. Реализация оптимизирующих трансформаций в системе предикатного программирования // Системная информатика, № 11. — Новосибирск, 2017. — С. 21-48. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/opttransform4.pdf (дата обращения: 22.10.2018).
3. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. Версия 0.12. Новосибирск, 2013. 28с., [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/plang14.pdf (дата обращения: 22.10.2018).
4. Климов А.В. Обзор подходов к созданию мульти-платформенной среды параллельного программирования. Научный сервис в сети Интернет: труды XIX Всероссийской научной конференции. М.: ИПМ им. М.В.Келдыша, 2017. С. 260-276. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://keldysh.ru/abrau/2017/19.pdf
5. Тумуров Э.Г., Шелехов В.И. Технология автоматного программирования на примере программы управления лифтом // «Программная инженерия», Том 8, № 3, 2017. – С.99-111. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/lift1.pdf (дата обращения: 22.10.2018).
6. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. С. 14-21.
7. Шелехов В.И. Доказательное построение, верификация и синтез предикатных программ // Знания-Онтологии-Теории (ЗОНТ-2017), Том 2. — Институт Математики СО РАН, Новосибирск, 2018. — С. 156-165. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/lbase.pdf (дата обращения: 22.10.2018).
8. Шелехов В.И. Классификация программ, ориентированная на технологию программирования // «Программная инженерия», Том 7, № 12, 2016. — С. 531–538. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/prog.pdf (дата обращения: 22.10.2018).
9. Шелехов В.И. Оптимизация автоматных программ методом трансформации требований // «Программная инженерия», №11, 2015. – С. 3-13. [Электронный ресурс]. Систем. требования:

Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/req_k.pdf (дата обращения: 22.10.2018).

10. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препр. / ИСИ СО РАН. № 164).
11. Шелехов В.И. Разработка программы построения дерева суффиксов в технологии предикатного программирования. Новосибирск, 2004. 52с. (Препр. / ИСИ СО РАН; N 115).
12. Шелехов В.И. Семантика языка предикатного программирования // Знания-Онтологии-Теории (ЗОНТ-2015). — Новосибирск, 2015. — С 206-215. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf (дата обращения: 22.10.2018).
13. Шелехов В.И. Язык и технология автоматного программирования // «Программная инженерия», №4, 2014. – С. 3-15. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/automatProg.pdf (дата обращения: 22.10.2018).
14. Волконский В.Ю., Брегер А.В., Бучнев А.Ю., Грабежной А.В., Ермолицкий А.В., Муханов Л.Е., Нейман-заде М.И., Степанов П.А., Четверина О.А. Методы распараллеливания программ в оптимизирующем компиляторе / ЗАО «МЦСТ», 2013. 28с. URL: http://www.mcst.ru/metody-rasparallelivaniya-programm-v-optimiziruyushhem-kompilyatore (дата обращения: 23.11.2018).
15. Касьянов В. Н., Стасенко А. П. Язык программирования Sisal 3.2 // Методы и инструменты конструирования программ. Новосибирск: ИСИ СО РАН, 2007. С. 56– 134.
16. Легалов А.И. Функциональный язык для создания архитектурно-независимых параллельных программ // Вычислительные технологии. – 2005. – № 1 (10). – С. 71–89.
17. Легалов А.И., Васильев В.С., Матковский И.В., Ушакова М.С. Инструментальная поддержка создания и трансформации функционально-потоковых параллельных программ // Труды Института системного программирования РАН, том 29, вып. 5, 2017, С. 165-184.
18. Cardoso J.M.P., Diniz P.C. Compilation Techniques for Reconfigurable Architectures. Springer. 2009. 230p.
19. Shelekhov V. I. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements // Automatic Control and Computer Sciences. 2011. Vol. 45, No. 7, P. 421–427.