В работе [1] предложен модифицированный вариант алгоритма Валианта [2] для синтаксического анализа, основанный на перемножении матриц. Основным недостатком этого алгоритма является сложность разделения его на независимые потоки. Нами предложена модификация этого алгоритма, которая отчасти решает эту проблему. Описание этой модификации разделено на несколько этапов: в разделе 1 введена терминология, немного отличающаяся от той, что используется в статье [1]. Далее, в разделе 2 переформулирован оригинальный алгоритм в рамках этой терминологии. Разработанная модификация алгоритма описана в секции 3. В секции 4 обсуждаются возможности параллельной реализации исходного и модифицированного алгоритмов.

1 Терминология

Пусть $G = (\Sigma, N, R, S)$ — контекстно-свободная грамматика в нормальной форме Хомского и $w = a_1 \dots a_n$ — строка, причем $n + 1 = 2^k$. С некоторыми изменениями алгоритм, предложенный в работе [1], применим также для булевых и стохастических грамматик. Подробнее адаптация алгоритма для использования с другими типами грамматик рассмотрена в статье [1] и здесь отдельно рассматриваться не будет. Ограничение на длину строки (для обоих рассмотренных алгоритмов) введены скорее из соображений простоты изложения, так как оба алгоритма довольно легко обобщаются на строки произвольной длины.

Определение 1.1. Пусть $G = (\Sigma, N, R, S)$ — контекстно-свободная грамматика, а $A \in N$ — нетерминальный символ, принадлежащий этой грамматике. Тогда определим $L_G(A) \subset \Sigma^*$, как язык, заданный грамматикой $G_A = (\Sigma, N, R, A)$. Также для двух нетерминалов $B, C \in N$ определим конкатенацию соответствующих языков $L_G(B)$ и $L_G(C)$, как конкатенации всех возможных пар строк из этих языков:

$$L_G(B)L_G(C) = L_G(BC) = \{w_1 w_2 \,|\, w_1 \in L_G(B), w_2 \in L_G(C)\}\,.$$

Известно, что конкатенация двух контекстно-свободных языков также является контекстно свободным языком [3]. Также в том случае, если из контекста однозначно можно восстановить грамматику G, в обозначении $L_G(\cdot)$ может опускаться нижний индекс.

Пусть T — треугольная матрица $(n+1) \times (n+1)$ с элементами $T[i,j] \subseteq N;\ 0 \leqslant i < j \leqslant n$. Цель рассматриваемого алгоритма — заполнить ячейки матрицы T так, чтобы выполнялось следующее условие:

$$T[i,j] = \{A \,|\, a_{i+1} \dots a_j \in L(A)\}, \quad \text{при } 0 \leqslant i < j \leqslant n\,.$$

Тогда принадлежность строки w к грамматике G будет определятся принадлежностью стартового нетерминала S к ячейче $T[0,n]$.

Вспомогательная матрица P с элементами $P[i,j] \subseteq N \times N$ такого же размера, как и матрица T, в результате должна быть заполнена значениями

$$P[i,j] = \{(B,C) \,|\, a_{i+1} \dots a_j \in L(B)L(C)\}, \quad 0 \leqslant i < j \leqslant n\,,$$

Определение 1.2. Будем называть (i,j) *корректной* парой индексов, если для них выполнены условия $0 \leqslant i < j \leqslant n$. Определим $\mathcal{I}$, как множество всех корректных индексов.

Заметим, что алгоритм в принципе рассматривает только те ячейки матриц T и P, которые задаются корректными парами индексов.

Определение 1.3. Назовем *(квадратной) подматрицей* такой набор корректных пар индексов $S = \{(i,j)\} \subset \mathcal{I}$, что существуют корректная пара индексов (a,b) и $size > 0$, для которых выполнены следующие условия: $a \geqslant size - 1$, $b \leqslant n + 1 - size$, а также пара (i,j) принадлежит множеству S тогда и только тогда, когда $a - size < i \leqslant a$ и $b \leqslant j < b + size$. Тогда $size$ — *размер*, а пара индексов (a,b) — *вершина* этой подматрицы.

Определение 1.4. Для подматрицы m с вершиной (a,b) и размером s определим множество индексов $\mathcal{I}_m \subset \mathcal{I}$, *влияющее на подматрицу* m, как

$$\mathcal{I}_m = \{(i,j) \subset \mathcal{I} \,|\, a - s < i; j \leqslant a\} \cup \{(i,j) \subset \mathcal{I} \,|\, b \leqslant i;\ j < b + s\}\,.$$

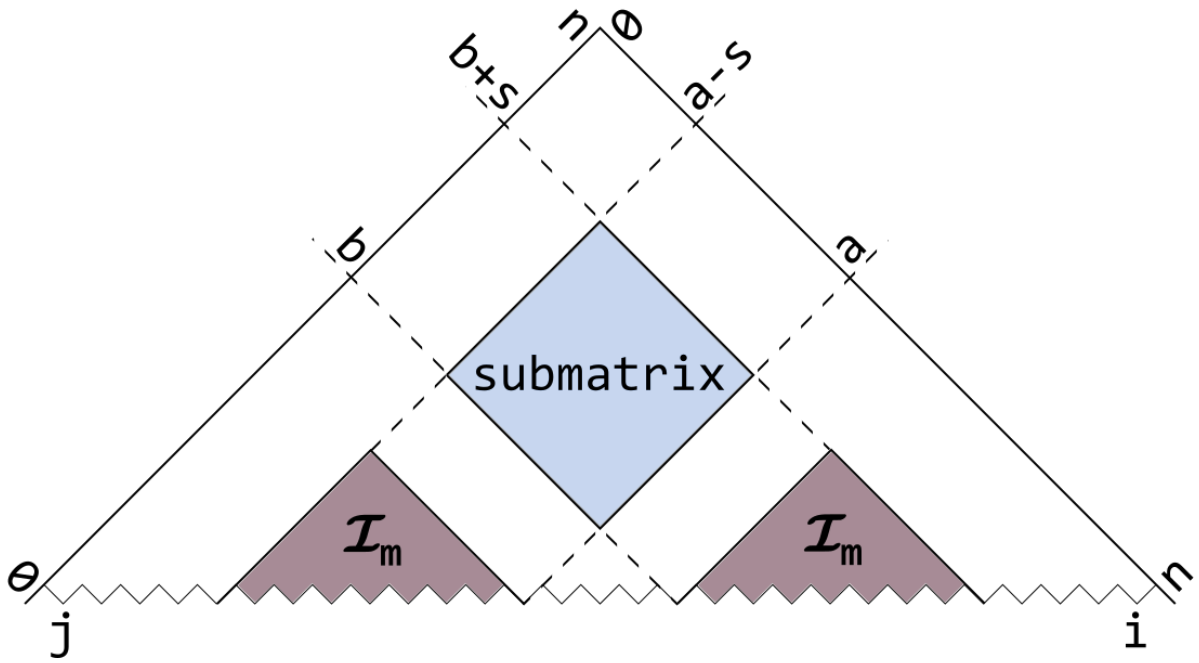

Рис. 1: Иллюстрация к определениям 1.3 и 1.4: (a,b) — вершина подматрицы submatrix размером $(s \times s)$; $\mathcal{I}_m$ — зона, влияющая на подматрицу submatrix

Определение 1.5. Пусть S — подматрица. Будем обозначать через $T[S]$ и $P[S]$ подматрицы матриц T и P, соответствующие множеству индексов S.

Листинг 1: Вспомогательные функции обработки подматриц

```
1  function size(m)
2  function bottomCell(m)
3
4  function shift(m, i, j)
5
6  function leftSubmatrix(m)
7  function topSubmatrix(m)
8  function rightSubmatrix(m)
9  function bottomSubmatrix(m)
10
11 function rightNeighbor(m)
12 function leftNeighbor(m)
13 function rightGrounded(m)
14 function leftGrounded(m)
```

Для описания алгоритмов нам необходимо задать ряд вспомогательных функций для оперирования подматрицами. Эти функции представлены в листинге 1.

Функции $size(m)$ и $bottomCell(m)$ вычисляют размер и вершину подматрицы m соответственно. Следующие четыре функции $leftSubmatrix(m)$ — $bottomSubmatrix(m)$ возвращают одну из подматриц, делящих исходную подматрицу m на четыре части, как показано на рисунке 2.

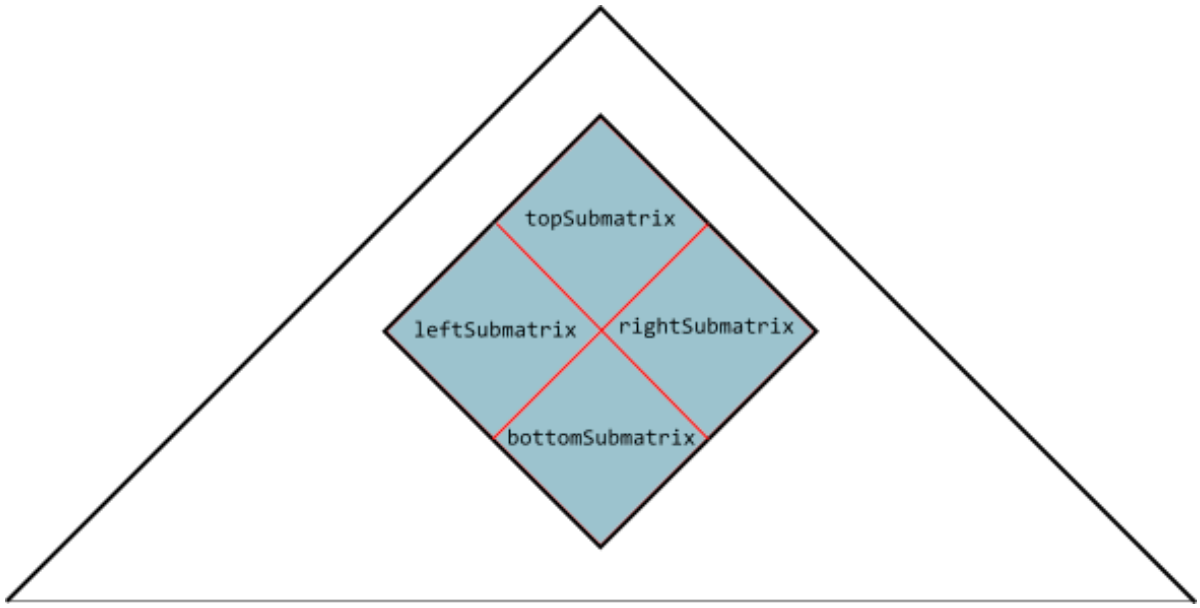

Рис. 2: Иллюстрация к листингу 1: разбиение матрицы на 4 компоненты

Функция $shift(m, i, j)$, примененная к целым числам i, j и подматрице m с вершиной (a, b) и размером s, возвращает подматрицу, сдвинутую относительно исходной на i и j по каждой из осей соответственно. Точнее, функция возвращает подматрицу размера s с вершиной $(a + i, b + j)$. На числа i и j накладываются естественные ограничения основанные на том, что результирующая подматрица не может выходить за границы множества $\mathcal{I}$.

В свою очередь функции $rightNeighbor(m)$ и $leftNeighbor(m)$ возвращают подматрицы, сдвинутые относительно исходной по одной из осей, как показано на рисунке 3. Таким образом, вызов $rightNeighbor(m)$ или $leftNeighbor(m)$ эквивалентен вызову $shift(m, size(m), 0)$ или

$shift(m, 0, -size(m))$ соответственно. Функции $rightGrounded(m)$ и $leftGrounded(m)$ (рисунок 3) также возвращают подматрицы, сдвинутые относительно m, и эквивалентны вызовам функций $shift(m, b-a-1, 0)$ или $shift(m, 0, -b+a+1)$ соответственно (здесь (a, b) — вершина матрицы m). Заметим, что для вершины (i, j) любой их двух матриц $rightGrounded(m)$ и $leftGrounded(m)$ выполняется равенство $i+1 = j$. Это означает, что вершина расположена в самой «нижней» из использующихся диагоналей всей матрицы.

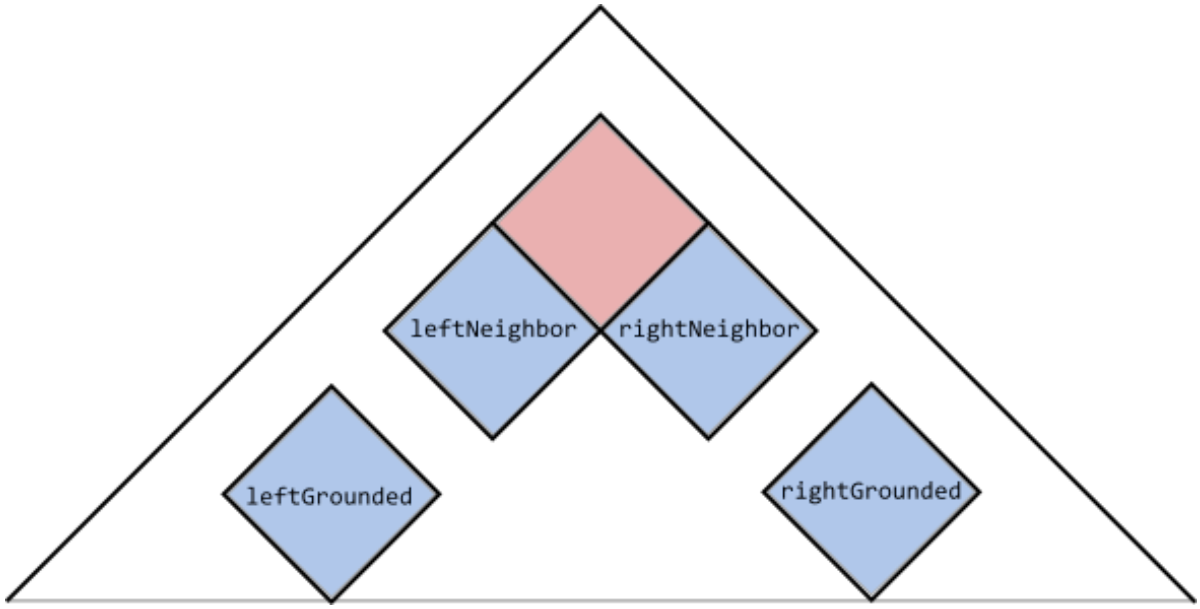

Рис. 3: Иллюстрация к листингу 1: соседи матрицы

Еще одна важная процедура — *performMultiplication* — реализует основную опрацию алгоритма — переможение подматриц. Ее тривиальная реализация представлена в листинге 2. Эта процедура принимает на вход множество упорядоченных троек подматриц (m, m_1, m_2) и для каждой из этих троек выполняет уножение $T[m_1] \times T[m_2]$, добавляя результат этого умножения в подматрицу $P[m]$.

Листинг 2: Процедура, использующаяся для перемножения матриц

```
1  procedure performMultiplications(task)
2    foreach (m, m1, m2) in task do
3      P[m] = P[m] ∪ (T[m1] × T[m2])
```

Под умножением матриц в данном случае имеется в виду операция определённая ниже.

Определение 1.6. Пусть $\mathcal{A}, \mathcal{B}$ — матрицы с элементами из N, причем $\mathcal{A}$ имеет размер $n \times k$, а $\mathcal{B}$ размер $k \times m$. Тогда результатом умножения матриц $\mathcal{A}$ и $\mathcal{B}$ является матрица $\mathcal{C}$ размера $n \times m$ с элементами из $N \times N$ и элементами, вычисленными по формуле

$$\mathcal{C}[i, j] = \{(A, B) \mid \exists k : A \in \mathcal{A}[i, k], B \in \mathcal{B}[k, j]\}.$$

2 Алгоритм синтаксического анализа, основанный на перемножении матриц

В данном разделе описание алгоритма, взятого из статьи [1] сформулировано в определённых выше терминах и представлено в виде псевдокода, приведённого в следующем листинге 3.

Здесь, процедура $compute(i, j)$ принимает на вход такие i, j, что $(i, j-1)$ — корректная пара индексов и записывает значения во все ячейки (i', j') матрицы T такие, что (i', j') — корректная пара индексов, $i' \geqslant i$ и $j' < j$.

Процедура $complete(m)$, в свою очередь, принимает на вход подматрицу m и определена только для подматриц с размером, являющимся степенью двойки. $complete(m)$ вычисляет все значения матрицы T на переданном ей множестве индексов m, при выполнении некоторых дополнительных условий. Во-первых, матрица T должны быть корректно заполнена для всех пар индексов из множества $\mathcal{I}_m$. Во-вторых для всех $(i, j) \in m$ текущее значение $P[i, j]$ должно быть следующим:

$$\{(B, C) \mid \exists k : a < k < b;\ a_{i+1} \dots a_k \in L(B) \text{ и } a_{k+1} \dots a_j \in L(C)\},$$

где (a, b) — вершина подматрицы m.

Листинг 3: Алгоритм синтаксического анализа, основанный на перемножении матриц

```
1  procedure main():
2      compute(n + 1, n + 1)
3
4  procedure compute(i, j):
5      if j − i ⩾ 4 then
6          compute (i, (i+j)/2)
7          compute ((i+j)/2, j)
8      denote m = submatrixByBottomCellAndSize ((i+j)/2 − 1, (i+j)/2), (j−i)/2)
9      complete(m)
10
11 procedure complete(m)
12     denote (i, j) = bottomCell(m)
13     if size(m) = 1 and i + 1 = j then
14         T[i, j] = {A | A → a_j ∈ R}
15     else if size(m) = 1 then
16         T[i, j] = {A | ∃(B, C) ∈ P[i, j] : A → BC ∈ R}
17     else if size(m) > 1 then
18         denote B = bottomSubmatrix(m), L = leftSubmatrix(m),
19                R = rightSubmatrix(m), T = topSubmatrix(m)
20         complete(B)
21         performMultiplications({(L, leftGrounded(L), B)})
22         complete(L)
23         performMultiplications({(R, B, rightGrounded(R))})
24         complete(R)
25         performMultiplications({(T, leftGrounded(T), R)})
26         performMultiplications({(T, L, rightGrounded(T))})
27         complete(T)
```

Отдельно заметим, что процедура $performMultiplications$, предназначенная, вообще говоря, для выпонения нескольких перемножений сразу, в этом алгоритме вызывается только для множеств мощности один (то есть выполняет роно одно перемножение за один вызов функции). Это накладывает ограничения на возможность параллельной реализации рассматриваемого алгоритма и предлагаемая в данной работе можификация позволяет снять эти ограничения.

3 Модифицированный алгоритм

Далее предложена модификация исходного алгоритма синтаксического анализа, предназначенная для его более естественной адаптации к задаче поиска подстрок, выводимых в заданной грамматике.

Главная процедура *main* сначала обрабатывает нижний слой матрицы T (клетки (i, j), для которых $i + 1 = j$), записывая в него корректные значения. Потом, с помощью функции *constructLayer*, разбивает матрицу T на слои, как показано на рисунке 4 (каждый слой состоит из набора подматриц, у каждой из которых отброшена нижняя четверть — *bottomSubmatrix*). Полученные слои обрабатываются последовательно снизу вверх, с помощью процедуры *completeVLayer*, заполняя тем самым всю матрицу T.

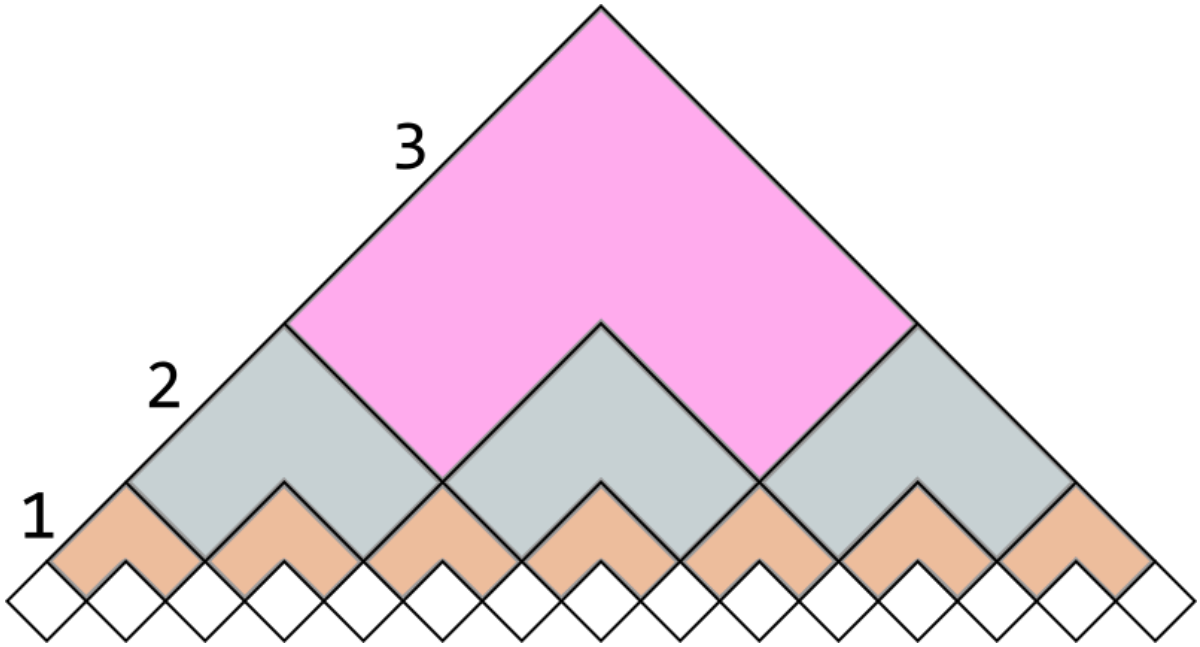

Рис. 4: Первичное разбиение на слои

Процедура *completeVLayer*, в свою очередь, принимает на вход набор подматриц M. Эти подматрицы не должны пересекаться, а также для любых двух подматриц $m_1, m_2 \in M$; (a_i, b_i), s_i — вершина и размер m_i соответственно, должно выполняться $s_1 = s_2$ и $b_1 - a_1 = b_2 - a_2$. Для каждого элемента m множества M процедура достраивает матрицу T для трех верхних четвертей ($leftSubmatrix(m)$, $rightSubmatrix(m)$ и $topSubmatrix(m)$). Для корректной работы этой функции, во-первых, необходимо, чтобы для любой $m \in M$ ячейки $T[i, j]$ были построены для $(i, j) \in bottomSubmatrix(m)$ и для $(i, j) \in \mathcal{I}_m$. Во-вторых, требуется выполнение ограничения на матрицу P, аналогичного ограничению в случае процедуры *complete* оригинального алгоритма, а именно: для любой $m \in M$ и для всех $(i, j) \in m$ текущее значение $P[i, j]$ должно быть следующим:

$$\{(B, C) \mid \exists k : a < k < b;\ a_{i+1} \dots a_k \in L(B) \text{ и } a_{k+1} \dots a_j \in L(C)\},$$

где (a, b) — вершина подматрицы m.

Третья процедура — *completeLayer* — тоже принимает на вход набор подматриц M, но для каждого элемента m этого набора достраивает матрицу T для всей подматрицы m. Ограничения на входные данные такие же, как и у процедуры *completeVLayer*. Для корректной работы этой функции необходимо, чтобы для любой $m \in M$ ячейки $T[i, j]$ были построены для $\mathcal{I}_m$, а так же выполнение того же требования на P, что и в предыдущем случае.

Листинг 4: Модифицированный алгоритм синтаксического анализа

```
1  procedure main():
2    for ℓ in {1, ..., n} do
3      T[ℓ − 1, ℓ] = {A | A → a_ℓ ∈ R}
4    foreach 1 ⩽ i < k do
5      denote layer = constructLayer(i)
6      completeVLayer(layer)
7
8  procedure constructLayer(i):
9    denote A = submatrixByBottomCellAndSize((2^i − 1, 2^i), 2^i)
10   return {B | B ⊂ I; ∃k ⩾ 0 : B = shift(A, k2^i, k2^i)}
11
12 procedure completeLayer(M):
13   if ∀ m ∈ M, size(m) = 1 then
14     denote cells = {bottomCell(m) | m ∈ M}
15     foreach {(i, j) ∈ cells | i + 1 ≠ j} do
16       T[i, j] = {A | ∃(B, C) ∈ P[i, j] : A → BC ∈ R}
17   else
18     denote bottomLayer = {bottomSubmatrix(m) | m ∈ M}
19     completeLayer(bottomLayer)
20     completeVLayer(M)
21
22 procedure completeVLayer(M):
23   denote leftSubLayer = {leftSubmatrix(m) | m ∈ M}
24   denote rightSubLayer = {rightSubmatrix(m) | m ∈ M}
25   denote topSubLayer = {topSubmatrix(m) | m ∈ M}
26
27   denote firstMultiplicationTask =
28     {(m, m1, m2) | m ∈ leftSubLayer, m1 = leftGrounded(m), m2 = rightNeighbor(m)}
29     ∪ {(m, m1, m2) | m ∈ rightSubLayer, m1 = leftNeighbor(m), m2 = rightGrounded(m)}
30   denote secondMultiplicationTask =
31     {(m, m1, m2) | m ∈ topSubLayer, m1 = leftGrounded(m), m2 = rightNeighbor(m)}
32   denote thirdMultiplicationTask =
33     {(m, m1, m2) | m ∈ topSubLayer, m1 = leftNeighbor(m), m2 = rightGrounded(m)}
34
35   performMultiplications(firstMultiplicationTask)
36   completeLayer(leftSubLayer ∪ rightSubLayer)
37   performMultiplications(secondMultiplicationTask)
38   performMultiplications(thirdMultiplicationTask)
39   completeLayer(topSubLayer)
```

Процедура *main* реализуется через *completeVLayer* очевидным образом. Для построения множества подматриц, составляющих слой под номером i, используется процедура *constructLayer*(i). Опишем подробнее принцип ее работы. Сначала заметим, что подматрицы уровня i должны иметь размер 2^i. Для того, чтобы построить необходимый слой сначала строится первая матрица слоя с вершиной $(2^i - 1, 2^i)$ и размером 2^i. После этого в слой добавляются все подматрицы, получающиеся из исходной сдвигом на $k * 2^i, k \geqslant 0$ по каждой из осей.

Процедура *completeLayer* по сути аналогична процедуре *complete* из алгоритма, взятого за основу, за исключением того, что она выполняется сразу для нескольких матриц. Основная разница состоит в том, что случай $size(m) = 1$ и $i + 1 = j$ в модифицимрованном алгоритме уже разобран (первые две строки процедуры *main*). Остается отдельно разобрать

случай $size(m) = 1$ и $i + 1 \neq j$. Иначе, матрицы разбиваются на четыре части и сначала следует рекурсивный вызов от *bottomSubmatrix* (для всех переданных матриц), а затем вызов процедуры *completeVLayer*, которая обрабатывает верхние части матриц.

Процедура *completeVLayer* для каждой из переданных матриц сначала выполняет два перемножения (соответствует 21 и 23 строкам в алгоритме из статьи), затем вызывает процедуру *completeLayer* от *rightSubmatrix* и $leftSubmatrix$ (22 и 24 строки оригинального алгоритма), далее выполняет оставшиеся два умножения (строки 25 и 26) и, наконец, вызывает *completeLayer* от оставшейся части *topSubmatrix* (строка 27).

Заметим, что процедура *performMultiplications* в общем случае, в отличие от исходного алгоритма, вызывается от множеств, состоящих из нескольких элементов.

4 Параллельная реализация

Рассмотрим процедуру $performMultiplications$ более подробно. Во-первых заметим, что матрица T на практике может представляться в виде нескольких булевых матриц: по одной матрице T_A на каждый нетерминал $A \in N$. Тогда значения в ячейках матрицы T задаются следующим образом: $T[i,j] = \{A \in N \mid T_A[i,j] = TRUE\}$. Аналогично, матрица P представляется в виде набора булевых матриц P_{BC}, по одной матрице на каждую пару нетерминалов $(B,C) \in N \times N$. Ее значения, в свою очередь, определяются так: $P[i,j] = \{(B,C) \in N \times N \mid P_{BC}[i,j] = TRUE\}$.

Исходя из этого, функция $performMultiplication$ может быть реализована так, как показано в листинге 5.

Листинг 5: Процедура, использующаяся для перемножения матриц

```
1  procedure performMultiplications(task)
2    foreach (m, m1, m2) in task do
3      foreach (B, C) in N × N do
4        P_BC[m] = P_BC[m] ∪ (T_B[m1] × T_C[m2])
```

Итого, количество перемножений матриц, соответствующее каждому вызову процедуры $performMultiplication$, равно мощности множества $task \times N \times N$. При этом все $n = |task \times N \times N|$ умножений независимы, то есть их можно совершать в произвольном порядке. Таким образом, возможно выполнять эти умножения параллельно без затрат на синхронизацию данных. При этом в модифицированном алгоритме, в отличие от исходного, мощность множества *task* больше единицы, что влечет за собой бóльшую степень параллелизма.

Следет заметить, что процедура *compute* исходного алгоритма является по сути древовидной рекурсией, в которой два рекурсивных вызова (строки 6 и 7 листинга 3) независимы (обрабатывают непересекающиеся множества индексов). Следовательно, их тоже можно совершать параллельно.

С точки зрения практики существует несколько различных широко распространённых платформ для параллельной реализации рассмотренных алгоритмов: (многоядерный) центральный процессор (CPU), графический процессор общего назначения (GPGPU). При этом, возможны несколько уровней примерения параллелизма: параллельное умножение двух матриц и параллельное выполнение перемножения нескольких пар матриц.

В результате, можно рассмотреть три различных способа реализации процедуры *performMultiplications*: параллельно на GPGPU, параллельно на CPU и последовательно. Важно то, что для достижения наилучшей практической производительности необходимо

комбинировать три способа в рамках одного алгоритма, а именно: матрицы очень большого размера (с количеством строк $s \geqslant M_{GPU}$) умножать на GPU, матрицы поменьше (с количеством строк s, назодящимся в границах $M_{Parallel} \leqslant s < M_{GPU}$) умножать параллельно на CPU и оставшиеся совсем маленькие матрицы умножать последовательно. Очевидно что, для каждого из двух алгоритмов оптимальные значения M_{GPU} и $M_{Parallel}$ могут быть разными.

Для того, чтобы проверить заявленные перимущества представленного алгоритма по возможности распараллеливания нами были реализованы исходный и модифицированный алгориты. Для умножения матриц на CPU и GPGPU используются одни и те же процедуры. По результатам ряда запусков были подобраны оптимальные значения $M_{GPU} = 64$ и $M_{Parallel} = 16$ для базового алгоритма и $M_{GPU} = 32$ и $M_{Parallel} = 1$ для модифицированного. В результате базовый алгоритм работал 7.1 и 34.3 секунды на строках длины 1023 и 2047 соответственно. Время работы модифицированного алгоритма: 5.6 и 23.2 секунды.

5 Заключение

Представленная модификация алгоритма облегчает задачу написания параллельной версии алгоритма, что было проверено на практике. Однако текущая реализация не является оптимальной и необходимо работать над её улучшением. Также в дальнейшем неоюбходимо провести более полное сравнение производительности на синтетических данных и апробацию на реальных данных.

Также, у представленной модификации есть еще одно преимущество: если мы хотим адаптировать алгоритм для задачи поиска подстроки длины не большей, чем какое-то заданное $m \ll n$, тогда нам надо строить матрицы T и P только для индексов $i + j \leqslant n + m$. В этом случае в оригинальном алгоритме придется производить большое количество «холостых» рекурсивных вызовов. В модифицированном алгоритме этого отчасти можно избежать, изменив верхнюю границу в цикле for (строка 4 листинга 4).

Список литературы

[1] Alexander Okhotin, “Parsing by matrix multiplication generalized to Boolean grammars”, *Theoretical Computer Science*, V. 516, p. 101–120, January 2014

[2] Leslie Valiant, “General context-free recognition in less than cubic time”, *Journal of Computer and System Sciences*, 10:2 (1975), p. 308–314.

[3] Hopcroft J. E., Motwani R., Ullman J. D., “Introduction to automata theory, languages, and computation”, 2007.