Санкт-Петербургский государственный университет

Кафедра системного программирования

Сусанина Юлия Алексеевна

Распараллеливание алгоритмов синтаксического анализа, основанных на матричных операциях

Курсовая работа

Научный руководитель:
к. ф.-м. н., доцент Григорьев С. В.

Санкт-Петербург
2018

Оглавление

Введение

Синтаксический анализ — это процесс определения принадлежности некоторой последовательности лексем языку, порождаемому грамматикой. Его классической областью применения является изучение преобразований языков программирования, а именно разбор исходного кода в процессе трансляции. В последнее время синтаксический анализ стал применяться для исследования последовательностей ДНК и РНК в биоинформатике [5]. Например, для определения принадлежности организма какому-либо семейству можно использовать его вторичную структуру рРНК [6], которая описывается с помощью некоторой грамматики. Более того, из-за больших объемов данных от синтаксического анализа требуется высокая производительность, которой можно добиться с помощью использования параллелизма.

Алгоритм синтаксического анализа Валианта [7], работающий с контекстно-свободными грамматиками, который позднее был расширен А.С. Охотиным до семейства булевых грамматик [3], является одним из наиболее подходящих решений для таких задач. Данный алгоритм строит матрицу разбора, элементы которой отвечают за выводимость конкретной подстроки. Валиант добился существенного увеличения производительности алгоритма за счет переноса большей части вычислений на перемножение подматриц исходной матрицы, которое позволяет эффективно использовать параллельные архитектуры. Основной недостаток этого алгоритма — сложность разделения его на независимые потоки, которые могли бы оперировать различными матрицами.

На кафедре системного программирования в лаборатории языковых инструментов была предложена модификация алгоритма А.С. Охотина [12], которая частично решает эту проблему и позволяет использовать параллелизм как на уровне матричных операций, так и за счет распараллеливания вычислений для целых массивов подматриц. Однако приемлемая реализация данного алгоритма отсутствует.

1. Постановка задачи

Целью данной работы является исследование и реализация модифицированного алгоритма А.С. Охотина. Для достижения цели были поставлены следующие задачи:

- реализовать модифицированный алгоритм, а также исходный алгоритм А.С. Охотина;
- дать теоретическую оценку эффективности использования параллельных вычислений в модифицированном алгоритме;
- провести экспериментальное исследование модифицированного алгоритма.

2. Обзор

В этом разделе будут кратко описаны алгоритмы синтаксического анализа, рассматриваемые в рамках данной работы, а также проект YaccConstructor, с помощью которого эти алгоритмы реализуются.

2.1. Синтаксический анализ, основанный на матричных операциях

Задача синтаксического анализа — это проверка принадлежности строки $a_1...a_n$ языку, определенному некоторой грамматикой $G = (\Sigma, N, R, S)$, где Σ, N — алфавиты терминалов и нетерминалов соответственно, R — конечное множество правил, S $\in$ N — стартовый нетерминал.

Определение 1. Пусть $L_G(A)$ — это язык, заданный грамматикой $G_A = (\Sigma, N, R, A)$.

Представленный далее алгоритм Валианта [7] работает с контекстно-свободными грамматиками в нормальной форме Хомского и строит верхнетреугольную матрицу размера $n \times n$ для конкретной строки $a_1...a_n$, где n — длина входной строки. Ее элементами являются множества нетерминалов и отвечают за выводимость конкретной подстроки

$$T[i, j] = \{A \in N | a_{i+1}...a_j \in L_G(A)\}$$

для $0 \leq i < j \leq n$.

Элементы данной матрицы вычисляются, начиная с $T[i-1, i] = \{A | A \to a_i \in R\}$.

Затем заполняются остальные элементы

$$T[i, j] = f(P[i, j]),$$

где $P[i, j] = \bigcup_{k=i+1}^{j-1} T[i, k] \times T[k, j]$, а $f(P) = \{A | \exists A \to BC \in R : (B, C) \in P\}$.

Таким образом, для определения принадлежности строки языку, заданному грамматикой G, необходимо, чтобы стартовый нетерминал S

содержался в $T[0, n]$.

Самой трудоемкой операцией в приведенном выше алгоритме является вычисление $P[i, j]$. За счет изменения порядка вычислений Валианту [7] удалось улучшить алгоритм так, чтобы большая часть вычислений выполнялась с помощью перемножения двух максимально возможных матриц, что заметно увеличило скорость работы алгоритма.

Алгоритм 1: Синтаксический анализ, основанный на матричных операциях (алгоритм Охотина)

Input: Булева грамматика в нормальной форме
$G = (\Sigma, N, R, S), w = a_1...a_n, n \geq 1, a_i \in \Sigma$, где n + 1 — степень двойки

```
1  main():
2  compute(0, n + 1);
3  accept if and only if S ∈ T_{0,n}

4  compute(l, m):
5  if m − l ≥ 4 then
6      compute(l, (l+m)/2);
7      compute((l+m)/2, m)
8  end
9  complete(l, (l+m)/2, (l+m)/2, m)

10 complete(l, m, l', m'):
11 if m − l = 4 and m = l' then
12     T_{l,l+1} = {A | A → a_{l+1} ∈ R};
13 end
14 else if m − l = 1 and m < l' then
15     T_{l,l'} = f(P_{l,l'});
16 end
17 else if m − l > 1 then
18     B = (l, (l+m)/2, (l+m)/2, m), B' = (l', (l'+m')/2, (l'+m')/2, m'),
       C = ((l+m)/2, m, l', (l'+m')/2), D = (l, (l+m)/2, l', (l'+m')/2),
       D' = ((l+m)/2, m, (l'+m')/2, m'), E = (l, (l+m)/2, (l'+m')/2, m');
19     complete(C);
20     P_D = P_D ∪ (T_B × T_C);
21     complete(D);
22     P_{D'} = P_{D'} ∪ (T_C × T_{B'});
23     complete(D');
24     P_E = P_E ∪ (T_B × T_{D'});
25     P_E = P_E ∪ (T_D × T_{B'});
26     complete(E)
27 end
28 complete(l, (l+m)/2, (l+m)/2, m)
```

Асимптотическая сложность алгоритма составляет $O(|G|MM(n)log(n))$, где n — длина входной строки, |G| — размер входной грамматики, MM(n) — количество операций необходимых для перемножения двух матриц размера n × n.

Определение 2. Обозначим (l, m, l', m') подматрицу матрицы $n \times n$, такую что $l \leq i < m$ и $l' \leq j < m'$.

Алгоритм Валианта был позже упрощен Александром Охотиным [3], что позволило ему работать в более общем случае, а именно, для конъюнктивных и булевых грамматик (Листинг 1).

Однако, как следует из определения функции *complete*, за один вызов данной функции перемножается только одна пара матриц, что накладывает определенные ограничения на его параллельную реализацию, и представленная далее модификация позволяет частично снять это ограничение.

2.2. Модифицированный алгоритм

В данном разделе рассматривается модификация алгоритма Охотина [12].

Функция *completeLayer* (строки 11-21 алгоритма 2) почти аналогична функции *complete* исходного алгоритма (строки 10-28 алгоритма 1). Однако она допускает одновременное независимое перемножение массивов матриц. Это достигается путем разбиения первоначальной матрицы на слои подматриц меньшего размера (функция constructLayer строки 9-10 алгоритма 2) и реорганизации вычислений.

С этими изменениями появляется возможность использования параллелизма на уровне слоев подматриц(строки 26-28, 29, 31-32 алгоритма 2), что должно положительно сказаться на эффективности работы алгоритма.

Алгоритм 2: Модифицированный алгоритм

Input: Булева грамматика в нормальной форме
$G = (\Sigma, N, R, S), w = a_1...a_n, n \geq 1, a_i \in \Sigma$, где n + 1 — степень двойки

main():

for $l \in \{1, \ldots, n\}$ **do**
    $T_{l,l+1} = \{A | A \to a_{l+1} \in R\}$
end
for $1 \leq i < k$ **do**
    layer = *constructLayer(i)*;
    completeVLayer(layer)
end

constructLayer(i):

$\{B | \exists k \geq 0 : B = (k * 2^i, (k+1) * 2^i, (k+1) * 2^i, (k+2) * 2^i)\}$

completeLayer(M):

if $\forall (l, m, l', m') \in M \quad (m - l = 1)$ **then**
    for $(l, m, l', m') \in M$ **do**
        $T_{l,l'} = f(P_{l,l'})$;
    end
end
else
    bottomLayer = $\{(\frac{l+m}{2}, m, l', \frac{l'+m'}{2}) | (l, m, l', m') \in M\}$;
    completeLayer(bottomLayer);
    completeVLayer(M)
end

completeVLayer(M):

leftSubLayer = $\{(l, \frac{l+m}{2}, l', \frac{l'+m'}{2}) | (l, m, l', m') \in M\}$;
rightSubLayer = $\{(\frac{l+m}{2}, m, \frac{l'+m'}{2}, m') | (l, m, l', m') \in M\}$;
topSubLayer = $\{(l, \frac{l+m}{2}, \frac{l'+m'}{2}, m') | (l, m, l', m') \in M\}$;
multiplicationTask1 =
$\{(l, m, l', m'), (l, m, m+1, m'-m-l'+1), (m, 2*m-1, l', m') | (l, m, l', m') \in leftSubLayer\} \cup$
$\{(l, m, l', m'), (l, m, 2*l'-m', l'), (l+l'-m-1, l'-1, l', m') | (l, m, l', m') \in rightSubLayer\}$;
multiplicationTask2 =
$\{(l, m, l', m'), (l, m, m+1, m'-m-l'+1), (m, 2*m-1, l', m') | (l, m, l', m') \in topSubLayer\}$;
multiplicationTask3 =
$\{(l, m, l', m'), (l, m, 2*l'-m', l'), (l+l'-m-1, l'-1, l', m') | (l, m, l', m') \in topSubLayer\}$;
performMultiplications(multiplicationTask1);
completeLayer(leftSubLayer $\cup$ *rightSubLayer)*;
performMultiplications(multiplicationTask2);
performMultiplications(multiplicationTask3);
completeLayer(topSubLayer)

2.3. Проект YaccConstructor

Реализация и тестирование данных алгоритмов проводятся в рамках проекта YaccConstructor [2], разрабатываемого на кафедре системного программирования [10] в лаборатории языковых инструментов [11] СПбГУ. YaccConstructor является платформой для исследований в области синтаксического анализа, разработанной на языке `F#`.

3. Реализация алгоритмов

Алгоритм А.С. Охотина и его модификация, рассмотренные в предыдущей главе были реализованы в рамках исследовательского проекта YaccConstructor. В данном разделе описываются детали практической реализации.

Одной из задач данной работы было исследование возможностей новой версии алгоритма, поэтому было решено реализовать оба алгоритма, чтобы затем уже сравнить их производительности на практике. Предполагается увидеть выигрыш в скорости работы модификации, который связан с возможностью, в отличии от исходного алгоритма, независимого перемножения большого количества пар матриц.

Алгоритмы были реализованы на платформе .NET на языке программирования `F#`.

В проекте используется язык описания грамматик YARD, которые перед запуском алгоритма преобразуются в нормальную форму Хомского. Все необходимые преобразования ранее уже были реализованы в проекте YaccConstructor.

Рассмотрим представление матриц T и P в реализуемых алгоритмов. На практике данные матрицы можно представить в виде нескольких булевых матриц, где каждой булевой матрице T_A соответствует нетерминал A входной грамматики G и матрице P_{BC} — пара нетерминалов (B,C). Таким образом, значения в ячейках теперь задаются следующим образом.

$T[i,j] = \{A \in N | T_A[i,j] = 1\}$;

$P[i,j] = \{(B,C) \in N \times N | P_{BC}[i,j] = 1\}$

Это позволяет работать с числовыми значениями, а не с множествами нетерминалов.

Теперь каждый вызов функции performMultiplications реализуемых алгоритмов представляет перемножение булевых матриц для всех пар нетерминалов(алгоритм 3). Все перемножения и заполнение ячеек матриц T_A и P_{BC} могут производиться независимо, то есть их можно выполнять параллельно без необходимости синхронизации данных.

Алгоритм 3: Функция performMultiplications после изменения представления матриц T и P

Input: Множество подматриц task

```
1 performMultiplications(task):
2 for (m, m1, m2) ∈ task do
3     for (B, C) ∈ N × N do
4         P^m_BC = P^m_BC(T^m1_B × T^m2_C)
5     end
6 end
```

Для параллельного перемножения матриц использовалась библиотека cuBLAS (CUDA Basic Linear Algebra Subroutine library) [8], реализующая набор базовых функций линейной алгебры, эффективно использующих возможности графических процессоров (GPU). Для интеграции .NET-приложений и cuBLAS использовалась библиотека managedCUDA [9].

4. Теоретическая оценка эффективности использования параллельных вычислений

При разработке параллельных алгоритмов важным моментом является оценка эффективности использования параллельных вычислений. В данном разделе мы дадим теоретическую оценку таких характеристик, как ускорение и загруженность параллельной версии модифицированного алгоритма.

Лемма 1. *Пусть длина входной строки $2^k - 1$. Тогда в Алгоритме 2 количество перемножений для подматриц размера $2^{k-i} \times 2^{k-i}$, где $i \in \{2, \ldots, k\}$, составляет $2^{2i-1} - 2^i$.*

Доказательство. Изменения в модификации по сравнению с исходным алгоритмов А.С. Охотина состоят в реорганизации порядка вычислений [12]. При этом количество перемножений подматриц не меняется и составляет $2^{2i-1} - 2^i$, для матриц размера $2^{k-i} \times 2^{k-i}$, где $i \in \{2, \ldots, k\}$, как уже было доказано в [3]. □

Теорема 1. *Пусть G — контекстно-свободная грамматика в нормальной форме Хомского, n — длина строки. Тогда Алгоритм 2 строит таблицу разбора T для этой грамматики и этой строки за время $O(|G| \cdot BMM(n) \cdot log(n))$, где BMM(n) — количество операций необходимых для перемножения двух булевых матриц размера $n \times n$.*

Доказательство. Так как количество операций осталось без изменений (Лемма 1), доказательство строится аналогично доказательству для алгоритма 1 в [3]. □

Теперь рассмотрим паралельную версию алгоритма. Мы будем считать, что слой подматриц, которые необходимо перемножить, обрабатывается параллельно и затрачивает время равное умножению одной пары матриц.

Лемма 2. *Пусть длина входной строки $2^k - 1$. Тогда в параллельной версии Алгоритма 2 количество перемножений для подматриц размера $2^{k-i} \times 2^{k-i}$, где $i \in \{2, \ldots, k\}$, составляет 3^{i-1}.*

Доказательство.

База.

При i = 2 подматрицами достигают наибольшого размера, соответственно их перемножение происходит 3 раза (вызов функции performMultiplication строки 29, 31-32 Алгоритма 2), а все дальнейшие вызовы функции происходят для подматриц меньшего размера.

Переход.

Пусть мы знаем, что количество перемножений для подматриц размера $2^{k-i} \times 2^{k-i}$ составляет 3^{i-1}. Теперь рассмотрим матрицы размера $2^{k-(i+1)} \times 2^{k-(i+1)}$. Подматрицы размера $2^{k-i} \times 2^{k-i}$ разделяются на 3 непересекающихся слоя (верхняя подматрица, нижняя подматрица и правая и левая подматрицы размера $2^{k-(i+1)} \times 2^{k-(i+1)}$), затем в строках 19, 30 и 33 Алгоритма 2 от каждого из этих слоев вызывается функция completeLayer, в которой происходит основное перемножение матриц. Таким образом, количество перемножений для матриц размера $2^{k-(i+1)} \times 2^{k-(i+1)}$ составляет $3 \cdot 3^{i-1} = 3^{(i+1)-1}$. □

Теорема 2. *Пусть G — контекстно-свободная грамматика в нормальной форме Хомского, n — длина строки. Тогда параллельная версия Алгоритма 2 строит таблицу разбора T для этой грамматики и этой строки за время $O(|G| \cdot BMM(n)$, где BMM(n) — количество операций необходимых для перемножения двух булевых матриц размера $n \times n$.*

Доказательство.

Пусть n = $2^k - 1$, а BMM(n) = $n^\omega \cdot f(n)$, где $\omega \geq 2$ и $f(n) = n^(O(1))$.

Согласно Лемме 2 количество перемножений для подматриц размера $2^{k-i} \times 2^{k-i}$, где $i \in \{2, \ldots, k\}$, составляет 3^{i-1}, и каждое из них еще домножается на константу C = O(|G|).

Тогда общее количество операций можно оценить следующим образом:

$$C \sum_{i=2}^{k} 3^{i-1} \cdot BMM(2^{k-i}) = C \sum_{i=2}^{k} 3^{i-1} \cdot 2^{\omega(k-i)} \cdot f(2^{k-i})$$
$$\leq 2^{\omega k} \cdot f(2^k) \sum_{i=2}^{k} 3^{i-1} \cdot 2^{-\omega i} = BMM(2^k) \sum_{i=2}^{k} 3^{i-1} \cdot 2^{-\omega i}$$

Осталось оценить сумму $\sum_{i=2}^{k} 3^{i-1} \cdot 2^{-\omega i}$. Так как $\omega \geq 2$, то $2^\omega > 3$, что означает что ряд сходится.

$$\sum_{i=2}^{k} 3^{i-1} \cdot 2^{-\omega i} \leq \sum_{i=2}^{\infty} 3^{i-1} \cdot 2^{-\omega i} = \frac{3 \cdot 2^{-\omega}}{-3+2^{\omega}}$$

Более того, он ограничен константой, что приводит к верхней оценке работы алгоритма $O(|G| \cdot BMM(n))$. $\square$

Теперь перейдем к оценке эффективности использования параллелизма в алгоритме 2. Для оценки данной характеристики будем использовать следующие критерии:

- ускорение $S_p = \frac{T_0}{T_p}$ (T_0 — время исполнения последовательной программы программы, T_p — время исполнения распараллеленной программы на p процессорах);
- загруженность $E_p = \frac{S_p}{p}$ (средняя доля времени выполнения алгоритма, в течение которой процессоры реально используются для решения задачи).

В данном случае количество процессоров p будет составлять длину максимального массива матриц, который мы хотим обработать за один шаг, то есть $2^k - 2$. Для модификации алгоритма А.С. Охотина и его распараллеленной версии ускорение будет составлять $log(n)$, а загруженность — $\frac{log(n)}{p}$. Разбиение матрицы разбора в процессе ее вычисления на обрабатывающиеся параллельно подслои, количество которых как раз равно $log(n)$, интуитивно и дает нам полученную выше оценку ускорения.

Таким образом, была дана теоретическая оценка эффективности использования параллелизма, однако простейшим примером проблемы, которая появляется при реализации, можно назвать время, затрачиваемое на отправление и получение данных с хоста при работе с GPU, из-за чего достижение высокой скорости работы алгоритма на практике оказывается не совсем простой задачей.

5. Эксперименты

В рамках данной работы были проведены исследования по сравнению производительности алгоритма А.С. Охотина и его модифицированной версии.

Компьютер, на котором производились замеры, обладает следующими характеристиками:

- операционная система: Windows 10 Pro;
- процессор: Intel(R) Core(TM) i7-4790 CPU @ 3.60 GHz;
- объем оперативной памяти: 16.0 GB;
- видеокарта: NVIDIA GeForce GTX 1070, 1920 cores.

Первой грамматикой для сравнения была выбрала грамматика G1 (Листинг 1). Данная грамматика имеет прежде всего теоретический интерес, потому что, как и большинство грамматик для задач биоинформатики, является сильно неоднозначной и позволяет оценить время работы алгоритмов на простейшем примере.

```
S : S S S | S S | B
```

Листинг 1: Грамматика $G1$

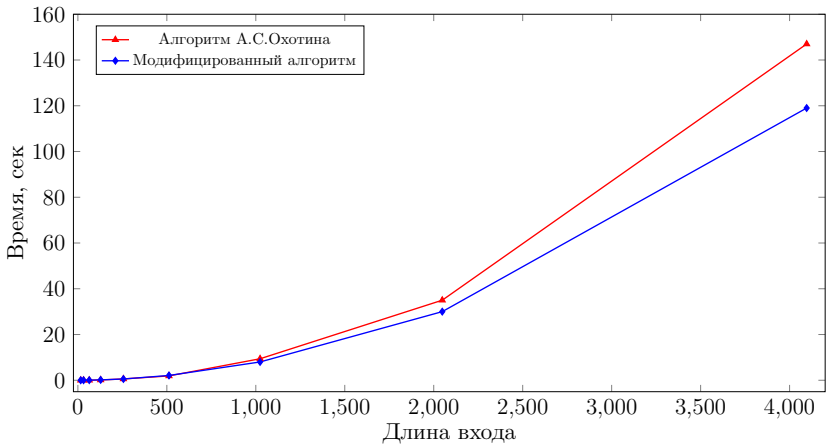

Рис. 1: Сравнительный анализ алгоритмов для грамматики $G1$

Далее реализованные алгоритмы сравнивались на грамматике G2 (Листинг 2), с помощью которой задается вторичная структура тРНК. Размер данной грамматики значительно увеличивается во время преобразования ее в нормальную форму Хомского, из-за чего значительно увеличивается время работы алгоритмов.

```
stem<s>:              [<Start>]
      A stem<s> U     full: folded any?
    | U stem<s> A
    | C stem<s> G        a_5_8 : any*[5..8]
    | G stem<s> C        a_1_3 : any*[1..3]
    | G stem<s> U
    | U stem<s> G        folded: stem<(a_1_3 stem<any*[7..10]>
    | s                                  a_1_3 stem<a_5_8>
                                         any*[3..6]
any: A | U | G | C                       stem<a_5_8>)>
```

Листинг 2: Грамматика $G2$

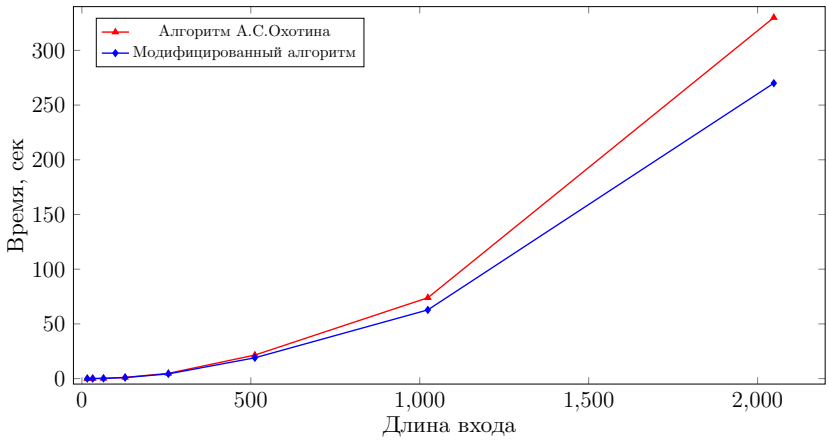

Рис. 2: Сравнительный анализ алгоритмов для грамматики $G2$

Результаты сравнительного анализа алгоритмов для грамматик G1 и G2 представлены на рис.1 и рис.2 соответственно.

Проведенные эксперименты показали, что модифицированная версия алгоритма А.С. Охотина может проигрывать на строках небольшой длины. Однако с увеличением количества лексем во входной строке разница во времени работы реализованных алгоритмов начинает увеличиваться, модификация показывает более высокую производительность, чем исходный алгоритм А.С. Охотина.

6. Заключение

В ходе данной работы были получены следующие результаты:

- реализованы алгоритм А.С. Охотина и его модифицированная версия на языке программирования `F#` с использованием библиотеки для параллельных вычислений cuBLAS в рамках исследовательского проекта YaccConstructor;
- получена теоретическая оценка эффективности использования параллельных вычислений в модифицированном алгоритме: ускорение параллельной версии в сравнении с последовательной составляет $log(n)$, а загруженность — $\frac{log(n)}{2^k-2}$;
- проведен сравнительный анализ алгоритма А.С. Охотина и модифицированной версии, который показал что модификация показывает лучшие результаты на строках большой длины.

Исходный код данной работы можно найти в ответвлений репозитория проекта YaccConstructor (https://github.com/SusaninaJulia/YaccConstructor), где ведутся дальнейшие усовершенствования реализации алгоритма, выходящие за рамки курсовой работы.

Дальнейшее направление работ

В дальнейшем необходимо добиться снижения времени работы алгоритмов путем применения более эффективных структур данных, например разреженных матриц. Также можно исследовать возможность применения парадигмы «разделяй и властвуй» для модифицированного алгоритма, как это сделано в работах [1] и [4].

Список литературы

[1] Bernardy J.-P., Claessen K. Efficient divide-and-conquer parsing of practical context-free languages // ICFP '13 Proceedings of the 18th ACM SIGPLAN international conference on Functional programming. — 2013. — P. 111–122.

[2] Kirilenko I., Grigorev S., Avdiukhin D. Syntax Analyzers Development in Automated Reengineering of Informational System // St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems. — 2013. — Vol. 174, no. 3. — P. 94–98.

[3] Okhotin A. Parsing by matrix multiplication generalized to Boolean grammars // Theoretical Computer Science. — 2014. — Vol. 516. — P. 101–120.

[4] Olausson T. Implementing incremental and parallel parsing. — 2014.

[5] Rivas E., Eddy S.R. The language of RNA: a formal grammar that includes pseudoknots // Bioinformatics. — 2000. — Vol. 16(4). — P. 334–340.

[6] Sükösd Z., Andersen E.S., R. Lyngsø. SCFGs in RNA Secondary Structure Prediction: A Hands-on Approach. — 2013.

[7] Valiant L.G. General context-free recognition in less than cubic time // Journal of Computer and System Sciences. — 1975. — Vol. 10(2). — P. 308–315.

[8] cuBLAS, CUDA Toolkit Documentation. — URL: `https://docs.nvidia.com/cuda/cublas/index.html` (online; accessed: 13.05.2018).

[9] managedCuda. — URL: `https://kunzmi.github.io/managedCuda` (online; accessed: 13.05.2018).

[10] Кафедра Системного Программирования. — URL: `http://se.math.spbu.ru/SE` (online; accessed: 13.05.2018).

[11] Лаборатория языковых инструментов. — URL: https://research.jetbrains.org/ru/groups/plt_lab (online; accessed: 13.05.2018).

[12] Явейн А. Разработка алгоритма синтаксического анализа через умножение матриц. — URL: https://github.com/YaccConstructor/articles/blob/master/InProgress/Yaveyn_alg/jbalg.pdf (online; accessed: 13.05.2018).