

УДК 004.442

Исследование эффективности параллельного преобразования лексемных деревьев в абстрактное синтаксическое представление

Никитин Владимир Леонидович

Соколов Александр Павлович (*научный руководитель*)

МГТУ им. Н.Э. Баумана, Москва, Россия

Современные компиляторы являются сложными программными системами, включающими множество этапов обработки исходного текста, среди которых важнейшее место занимает построение абстрактного синтаксического дерева. Этот этап предполагает трансформацию конкретного синтаксического представления, хранящего лексемную структуру исходного кода в более компактную и абстрактную форму, пригодную для дальнейшего семантического анализа и генерации кода. Представлено исследование возможности распараллеливания алгоритма преобразования лексемного дерева в синтаксическое и реализация его многопоточной версии. Результаты вычислительного эксперимента подтверждают, что предложенный подход позволяет добиться ускорения до 1,8 раз при работе с входными данными. Отмечено, что эффективность масштабирования ограничивается структурными характеристиками исходного дерева, в частности средней шириной и числом независимых подузлов.

Ключевые слова: параллельные вычисления, компиляция, синтаксический анализ, абстрактное синтаксическое дерево, многопоточность

Введение. Компиляторы современных языков программирования включают в себя множество этапов анализа исходного кода, начиная от лексического анализа и заканчивая генерацией машинного кода. В этом процессе построение абстрактного синтаксического дерева (англ. Abstract Syntax Tree, AST) играет ключевую роль, так как оно используется для всех последующих трансформаций и анализов кода [1].

AST представляет собой представление синтаксической структуры программы и строится на основе промежуточного дерева — лексемного дерева (англ. Raw Lexeme Tree — RLT), которое сохраняет структуру и местоположение лексем [2]. Однако при большом объеме исходных файлов RLT может стать громоздким и сложным для обработки.

Последовательные алгоритмы преобразования RLT в AST могут не справляться с такими случаями эффективно. Это приводит к увеличению времени компиляции. Решением данной проблемы является внедрение параллельных методов, которые позволяют одновременно обрабатывать несколько независимых частей дерева.

Настоящая работа посвящена разработке, реализации и исследованию эффективности параллельного алгоритма преобразования RLT в AST.

Постановка задачи. Алгоритм преобразования получает дерево RLT на вход и производит AST [3]. Далее подробнее рассматривается структура RLT. Каждый узел дерева определяется в виде структуры данных двух видов: а) структуры, где каждое поле является набором других узлов (рис. 1); б) структуры, содержащей позицию в исходном файле (рис. 2).

```
1 struct Function {  
2     keyword: Keyword,  
3     id: Identifier,  
4     params: Vec<Parameter>,  
5     body: Body  
6 }
```

Рис. 1. Пример структуры-узла для синтаксиса функции

```
1 struct Keyword {  
2     length: u32,  
3     offset: u32 // отступ от начала  
4     файла  
5 }
```

Рис. 2. Пример структуры-листа для синтаксиса ключевого слова

Такая архитектура дерева позволяет легко строить его в ходе синтаксического анализа с помощью рекурсивных нисходящих парсеров. Само же RLT необходимо для получения точного местоположения той или иной лексемы и используется в основном для создания диагностик.

В отличие от RLT, дерево AST сформировано по-другому: узлы в синтаксическом дереве задаются небольшими независимыми структурами. Дополнительно каждому узлу присваивается индекс. Каждый узел хранится в едином массиве, а связи между ними формируются с помощью отдельного графа. С использованием такого подхода становится возможным динамическое создание дерева; кроме того, упрощается поиск в дереве. Традиционные подходы — визиторы, рекурсивные обходы и прочие — требуют значительно большего дополнительного кода для работы.

Формально задача преобразования может быть сформулирована следующим образом: пусть у всех узлов N дерева RLT есть множество $\{r_i\}$ — множество дочерних узлов и для каждого вида узла определена функция $f: N \rightarrow A$, где A — множество деревьев AST. Тогда вычисление функции f сводится к конструированию дерева

$$A_0 < f(r_1) < f(r_2) < \dots < f(r_n), \quad (1)$$

где A_0 — базовое поддерево; $a \prec b$ — операция присоединения поддерева b к дереву a в качестве дочернего. Так как все вызовы функции f независимы друг от друга, то они могут быть выполнены параллельно. Операции присоединения должны быть выполнены последовательно, так как зависят от результата предыдущей операции.

Программная реализация. На основании формулы (1) была разработана программная реализация параллельного преобразования (рис. 3). Параллельность достигается за счет использования пула потоков (англ. thread pool) [4]. При этом потоки используют стратегию «похищения» задач (англ. work stealing) [5]. Благодаря этому нагрузка на потоки распределяется динамически и равномерно.

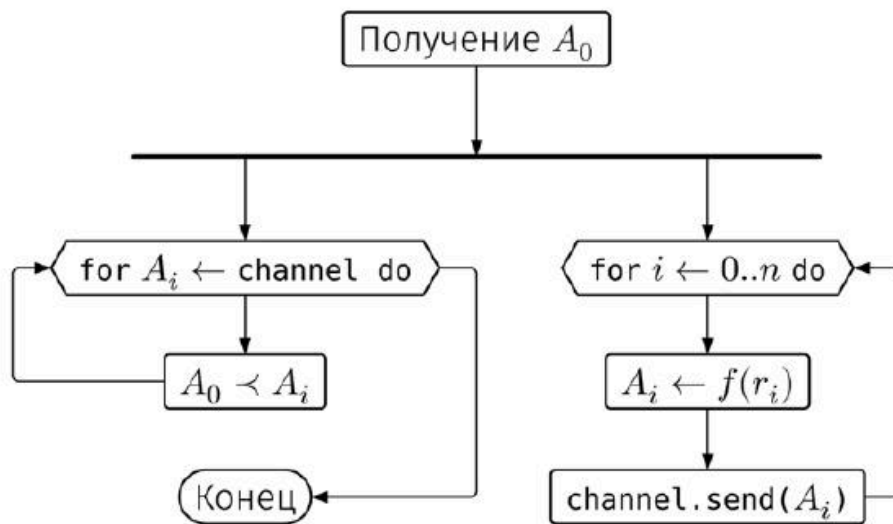


Рис. 3. Блок-схема параллельного преобразования узла лексемного дерева

Для синхронизации операций присоединения используется канал синхронизации (англ. multiple producers, single consumer channel — MPSC channel).

Вычислительный эксперимент. Для измерения эффективности параллельного алгоритма был поставлен вычислительный эксперимент. В качестве исходных данных было сгенерированное случайным образом дерево RLT со следующими свойствами: средняя ширина дерева — 2,32, максимальная ширина — 16, высота дерева — 12. Результаты получены на процессоре 12th Gen Intel® Core™ i7-12650H × 10, ОС — Linux 6.12.26.

В качестве измеряемых характеристик выбраны ускорение (рис. 4) и эффективность (рис. 5) от числа потоков p и время работы в зависимости от размера задачи (рис. 6). Размер задачи напрямую зависит от количества узлов в дереве RLT, поэтому для этого оно было увеличено в $n = 2, 5, 10, 50, 200$ раз, а также было зафиксировано количество потоков $p = 9$.

В табл. 1, 2 представлены соответствующие значения измеряемых характеристик.

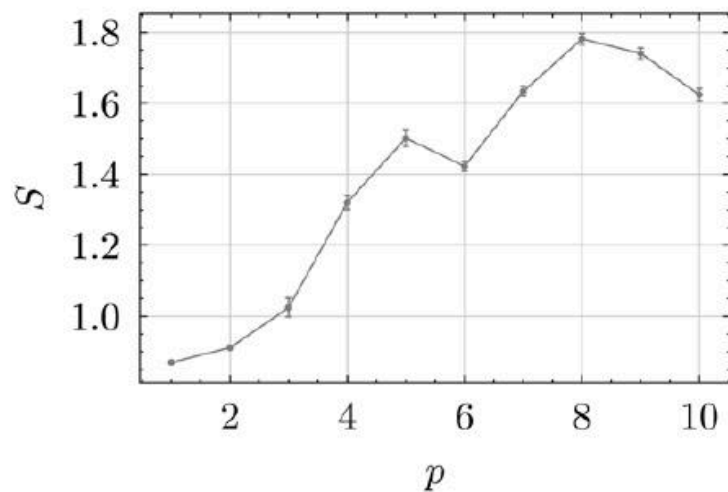


Рис. 4. График зависимости ускорения S от числа потоков p

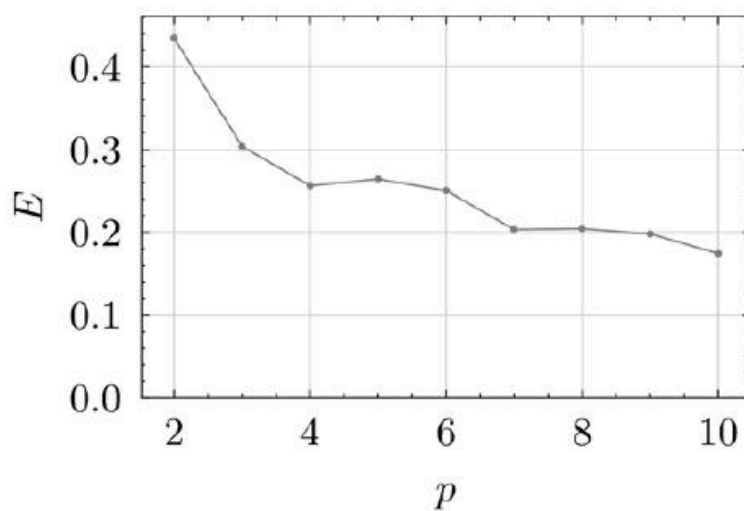


Рис. 5. График эффективности E от числа потоков p

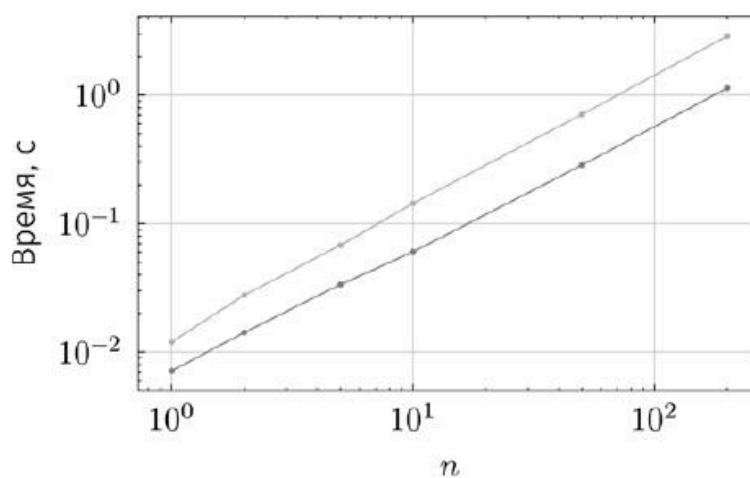


Рис. 6. Log-log график времени работы в зависимости от сложности задачи:
синяя линия — параллельная версия, оранжевая — последовательная

Таблица 1

Численные значения ускорения и эффективности от числа потоков

Количество потоков p	Ускорение S	Эффективность E
1	0,870	0,870
2	0,912	0,456
3	1,024	0,341
4	1,321	0,33
5	1,502	0,30
6	1,423	0,237
7	1,634	0,223
8	1,782	0,223
9	1,741	0,193
10	1,625	0,162

Таблица 2

Время выполнения последовательного и параллельного алгоритмов на разных размерах данных

Коэффициент увеличения n	Время посл. t_s , с.	Время парал. t_p , с.	Ускорение $S_{p=9}$
1	0,012	0,007	1,679
2	0,028	0,014	1,953
5	0,068	0,034	2,022
10	0,144	0,06	2,386
50	0,709	0,286	2,483
200	2,888	1,137	2,541

Из первых двух графиков видно, что параллельный алгоритм показывает более плохие результаты, чем последовательный, обеспечивая ускорение ниже 1, при потоках $p < 3$. Это связано с высокими накладными расходами по синхронизации. Кроме того, из-за структуры алгоритма, доля последовательных операций $f \approx 50\%$. Стоит отметить, что удалось добиться ускорения по сравнению с последовательной версией — до 1,8 раз при $p = 8$.

Заключение. В результате работы получены следующие результаты: анализ алгоритма преобразования показал, что есть возможность его распараллеливания; реализованный параллельный алгоритм преобразования RLT в AST работает корректно; проведенные вычислительные эксперименты по-

казали эффективность параллельного алгоритма — достигнуто ускорение до 1,8 раз при 8 потоках по сравнению с последовательной версией. Используемый подход можно применять и для параллельной обработки других древовидных форматов — json, xml и прочих.

Литература

- [1] Färnstrand L. Parallelization in Rust with fork-join and friends. Creating the ForkJoin framework. Master's thesis in Computer Science and Engineering. Gothenburg, Chalmers University of Technology, 2015.
- [2] Hsu A.W. The key to a data parallel compiler. USA, ACM, 2016.
- [3] Dubach C. Lecture 7: Parse Tree and Abstract Syntax. URL: <https://www.cs.mcgill.ca/~cs520/2021/slides/7-ast.pdf> (accessed 05.03.2025).
- [4] Божко А.Н., Волосатова Т.М., Грошев С.В., Карпенко А.П., Мартынюк В.А. *Основы автоматизированного проектирования*. Москва, ИНФРА-М, 2017, 327 с.
- [5] *Оценка эффективности параллельных алгоритмов*. URL: <http://bigor.bmstu.ru/?cnt/?doc=Parallel/ch030102.mod/?cou=Parallel/base.cou> (дата обращения 18.04.2025).

Research of the effectiveness of parallel transformation of lexeme trees into an abstract syntactic representation

Nikitin Vladimir Leonidovich

Sokolov Alexandr Pavlovich (*scientific advisor*)

BMSTU, Moscow, Russia

Modern compilers are sophisticated software systems that involve multiple stages of processing source code. Among these stages, the most significant is the construction of an abstract syntax tree (AST). This process involves transforming a specific syntactic representation, which captures the lexical structure of the source code, into a more concise and abstract form that is suitable for further semantic analysis and code generation. This paper presents a study of the possibility of parallelizing the algorithm for converting a lexeme tree into a syntactic tree and implementing its multithreaded version. The results of the computational experiment confirm that the proposed approach allows achieving up to 1.8 times speedup when dealing with input data. It is noted that the efficiency of scaling is limited by structural characteristics of the initial tree, in particular, the average width and the number of independent subnodes.

Keywords: parallel computing, compilation, syntax analysis, abstract syntax tree, multithreading