

## ОСНОВНЫЕ АЛГОРИТМЫ КОМПИЛЯТОРА ДЛЯ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

Альбатша Ахмад Мухаммад Хусайн, старший преподаватель  
Национальный исследовательский университет Высшая школа экономики  
(Россия, г. Москва)

DOI:10.24412/2500-1000-2025-3-1-195-199

***Аннотация.** Параллельные программы стали приобретать все большую важность и значимость в последние годы из-за стремительного роста информационных массивов и как следствие необходимости быстрой обработки больших объемов данных. В тоже время, традиционные методы компиляции, разработанные для последовательных программ, не гарантируют правильность (последовательную согласованность) преобразований компилятора при применении к параллельным программам. В связи с этим, статья посвящена анализу основных алгоритмов компилятора для такого рода программ, позволяющих применять методы распараллеливания и оптимизации.*

***Ключевые слова:** код, компилятор, алгоритм, параллельная программа.*

Разработка компиляторов для параллельных программ очень важна в контексте реализации всего потенциала современных вычислений. Параллельные вычисления – это подход, который подразумевает использование несколько процессоров или компьютерных систем для решения задачи. Он широко используется в различных областях, таких как научные вычисления, анализ данных и машинное обучение, поскольку позволяет значительно сократить время, необходимое для решения сложных задач. Параллелизм разбивает задачи на более мелкие подзадачи, используя для этого аппаратное и программное обеспечение. Среди преимуществ параллельных вычислений отдельно следует выделить – повышение производительности, ускорение обработки данных, рост эффективности, масштабируемость и эффективное решение задач с интенсивными вычислениями [1].

В тоже время, необходимо отметить, что компиляция и запуск параллельных программ гораздо сложнее, чем работа с последовательными программами, что предопределяет необходимость использования более совершенных алгоритмов компиляторов. Существует два способа, с помощью которых код может выполнять различные задачи параллельно и иметь связь между ними: общая память и передача сообщений. В рамках модели параллельного программирования с общей памятью все потоки в задании имеют доступ к глобальному адресному пространству. Связь между потоками осуществляется через чтение

и запись общих переменных, а не через явные операции связи [2]. Процессоры могут обращаться к общей переменной параллельно без какого-либо фиксированного упорядочивания обращений, что приводит к скачкам данных и недетерминированному поведению. Передача сообщений – это когда используется группа машин, каждая со своим собственным центральным процессором, памятью и копией операционной системы. Файловые системы/диски по-прежнему могут быть общими.

Скачки данных и синхронизация делают невозможным применение классических методов оптимизации и анализа компилятора непосредственно к параллельным программам, поскольку классические методы не учитывают обновления общих переменных в потоках, отличных от анализируемого. Классические оптимизации могут изменить смысл программ, когда они применяются к параллельным программам с общей памятью [3].

Таким образом, необходимость более подробного изучения возможностей высокоуровневой оптимизации кода параллельных программ с помощью компиляторов, а также их способности обеспечивать многопоточность посредством автоматического распараллеливания, предопределила выбор темы данной статьи.

Алгоритмы, позволяющие компиляторам реализовывать автоматическое распараллеливание, что дает возможность обнаруживать циклы, которые могут быть безопасно и эффективно выполнены параллельно, а также

генерировать многопоточный код, рассматривают в своих трудах Черноног В.В., Дьячков И.Л., Добров А.Д., Логунов Б.А., Харин И.А., Mary W. Hall, Margaret Martonosi.

Изучением методов компиляции параллельных программ, которые имеют расширенные возможности для приватизации массивов, символического и нелинейного тестирования зависимости данных, распознавания идиом, межпроцедурного и символического анализа программ занимаются Баглий А.П., Кривошеев Н.М., Штейнберг Б.Я., Афанасьев В.О., Бородин А.Е., Белеванцев А.А., Xavier Martorell, Jesús Labarta, Nacho Navarro, José Oliver.

Детальное описание возможностей компилятора для выявления и использования параллелизма в коде, оптимизации структур циклов входит в круг научных интересов Владимирова К.И., Тарасова Ю.В., Городней Л.В., Tiago Carneiro Pessoa, Jan Gmys, Francisco Heron.

Имеющиеся на сегодняшний день наработки и публикации, безусловно заложили ключевые основы понимания и роли компиляторов для параллельных программ. Однако, с развитием вычислительных мощностей, ряд вопросов требует дополнительной проработки и более детального анализа. Так, например, нерешенной проблемой при проектировании компиляторов для параллельных машин является анализ зависимости данных. Отдельного внимания заслуживают вопросы, связанные с формализацией требований, которые необходимо выполнить компилятору для распараллеливания циклов.

Таким образом, цель статьи заключается в изучении основных алгоритмов компилятора для параллельных программ.

**Результаты исследования.** Прежде всего, необходимо отметить, что алгоритмы компилятора для параллельных программ направлены на оптимизацию каждого из выполняемых ими этапов с целью эффективного использования оборудования и мощностей, что в конечном итоге позволяет получить высокопроизводительный код [4]. Конкретные техники и алгоритмы зависят от модели параллелизма и возможностей целевого оборудования.

Компиляторы используют различные техники и алгоритмы, позволяющие им оптимизировать код для параллельного выполнения. Рассмотрим более подробно существующие алгоритмы на примере конкретных задач, которые решают компиляторы.

#### 1. Разворачивание циклов

Разворачивание циклов подразумевает их разбиение на несколько меньших аналогов, что позволяет выполнять несколько итераций параллельно. Это уменьшает накладные расходы цикла и дает возможность увеличить параллельность.

Например, в простом цикле: `с` заложена возможность выполнять две итерации одновременно при наличии достаточного количества ресурсов. В тоже время, следует отметить, что подобного рода алгоритм может эффективно анализировать только циклы с относительно простой структурой. Например, он не в состоянии установить потокобезопасность цикла, который содержит вызовы внешних функций, поскольку не знает, есть ли у вызова функции побочные эффекты, которые вносят зависимости.

```
for (int i = 0; i < N; i++) {
// Loop body
}
- Loop unrolling might transform it into:
for (int i = 0; i < N; i += 2) {
// Loop body for i
// Loop body for i+1
}
...
```

В данном случае используется более сложный алгоритм для разворачивания циклов, которые предлагают компиляторы Intel® C++ и

Fortran. Задача этих алгоритмов заключается в проведении анализа потока данных в циклах, чтобы определить, какие из них могут быть

безопасно и эффективно выполняться параллельно. Например, атрибут `concurrency_safe` можно использовать в компиляторе Intel® C, чтобы утверждать, что функция безопасна для параллельного выполнения, без неожиданных побочных эффектов или конфликтов доступа к памяти между несколькими вызовами функции. Другой способ, в C или Fortran, – вызвать межпроцедурную оптимизацию с помощью опции компилятора – `Qipo` (Windows) или `-ipo` (Linux or macOS). Это дает компилятору возможность оценить вызываемую функцию на предмет побочных эффектов.

То, что цикл можно распараллелить, не означает, что его следует распараллеливать. Компилятор использует модель затрат с пороговым параметром, чтобы решить, распараллеливать ли цикл. Опции компилятора – `Qpar-threshold[n]` (Windows) и `-par-threshold[n]` (Linux) настраивают этот параметр. Значение

$n$  варьируется от 0 до 100, где 0 означает всегда распараллеливать безопасный цикл, независимо от модели затрат, а 100 указывает компилятору распараллеливать только те циклы, для которых выигрыш в производительности весьма вероятен. Значение  $n$  по умолчанию консервативно и равно 100; иногда снижение порога до 99 может привести к значительному увеличению числа распараллеливаемых циклов. Прагма `#parallel always` (!DIR\$ PARALLEL ALWAYS в Fortran) может быть использована для переопределения модели затрат для отдельного цикла [5].

Примеры работы данных компиляторов представлены ниже.

Ключи `-Qopt-report-phase:par` `-Qopt-report[:n]` (Windows) или `-qopt-report-phase=par` `-qopt-report[=n]` (Linux), где  $n$  - от 1 до 5, показывают, какие циклы были распараллелены:

```
LOOP BEGIN at par.f90(4,1)
  remark #17109: LOOP WAS AUTO-PARALLELIZED
LOOP END
```

Компилятор также сообщит, какие циклы не удалось распараллелить, и причину этого, как в следующем примере:

```
LOOP BEGIN at par.f90(4,1)
  remark #17104: loop was not parallelized: existence of parallel
dependence
LOOP END
```

Также это иллюстрируется следующим образом:

```
void add (int k, float *a, float *b)
{
  for (int i = 1; i < 10000; i++)
    a[i] = a[i+k] + b[i];
}
```

Следующее сообщение показывает, что цикл распараллелен:

```

LOOP BEGIN at add.cpp(4,1)
  remark #17109: LOOP WAS AUTO-PARALLELIZED
  remark #17101: parallel loop shared={ } private={ } firstprivate={ b k
a i }
lastprivate={ } firstlastprivate={ } reduction={ }
LOOP END

```

## 2. Создание прототипа языка параллельных шаблонов

Прототип компилятора языка параллельных шаблонов (PPL) позволяет проверить применимость паттернового подхода к реальным задачам и средам. Подход PPL нацелен на более широкий спектр приложений и оборудования, отделяя параллельную семантику от используемого оборудования, при этом

оставляя степень параллелизма и использования на усмотрение оптимизации [6]. Цель PPL – обеспечить гораздо более широкую применимость, сосредоточившись на всем приложении при генерации оптимизированного кода.

На рисунке 1 представлена общая структура компилятора.

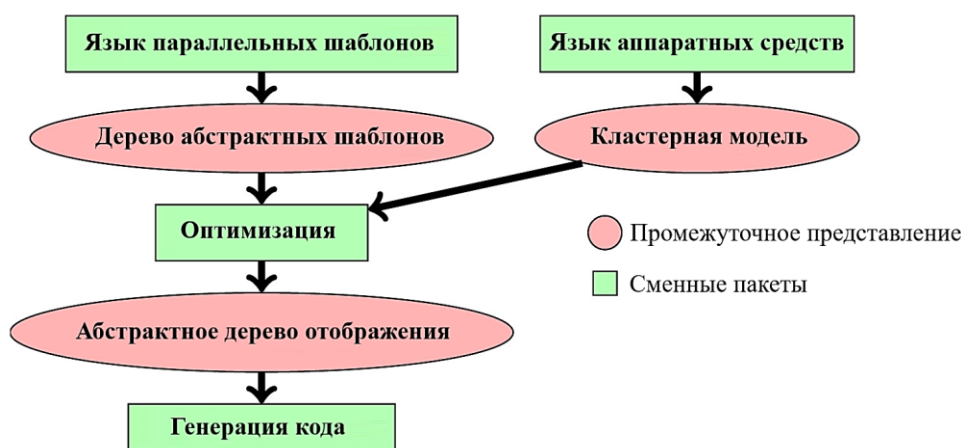


Рис. 1. Схема работы компилятора для создания прототипа языка параллельных шаблонов

Работа компилятора, представленного на рисунке 1 включает в себя пять последовательных этапов:

1. Оценка данных, которые были введены пользователем.

(а) с использованием языка, ориентированного на конкретную область, проводится анализ исходных тексты приложений.

(б) язык аппаратного обеспечения: на основе объектной нотация JavaScript осуществляется описание аппаратного обеспечения целевого кластера.

2. Создание дерева абстрактных шаблонов

3. Оптимизация: выполнение глобальных оптимизаций дерева абстрактных шаблонов. Это может быть переупорядочивание либо же наглядное аппаратное отображение.

4. Создание дерева абстрактного отображения: на этом этапе осуществляется генерация дерева используя результаты оптимизации, что позволяет реализовать необходимую синхронизацию и обеспечить передачу данных.

5. Написание оптимизированного кода на C++.

Считается, что подход PPL обладает широким потенциалом, благодаря его текущему рабочему процессу, который отлично подходит для статических кодов.

3. *Векторизация* позволяет оптимизировать код для архитектур с одной инструкцией и несколькими данными. В данном случае алгоритмы компилятора обеспечивают преобразование скалярных операций в векторные, когда одна инструкция параллельно оперирует не-

сколькими элементами данных. В результате пользоваться может получить существенное повышение производительности.

На последнем этапе проводимого исследования, по мнению автора, целесообразно акцентировать внимание на перспективах развития алгоритмов компилятора для параллельных программ. Эксперты считают, что будущее компиляторов для параллельных машин будет зависеть от развития аппаратных и программных технологий. Одной из наиболее значимых тенденций является появление гетерогенных вычислений, которые предполагают использование в одной системе нескольких типов процессоров. Помимо этого, еще один вектор развития предусматривает более

широкое использование машинного обучения в алгоритмах компиляторов. Благодаря машинному обучению компиляторы могут научиться принимать лучшие решения о том, как оптимизировать код на основе прошлых данных о производительности.

**Заключение.** Компиляторы играют важнейшую роль в разработке параллельных программ. Помимо задачи трансляции и оптимизации программ, они предоставляют полезную инфраструктуру для создания широкого спектра инструментов повышения производительности кода. В статье на примере конкретных задач, которые решают компиляторы для параллельных программ, рассмотрены алгоритмы их реализации.

#### Библиографический список

1. Алеева В.Н. Автоматизированное проектирование и исполнение эффективных программ для численных алгоритмов // Вестник Южно-Уральского государственного университета. – 2023. – Т. 12, № 3. – С. 31-49.
2. Баглий А.П. Преобразования программ в оптимизирующей распараллеливающей системе для распараллеливания на распределенную память // Инженерный вестник Дона. – 2022. – № 12 (96). – С. 266-285.
3. Jiange Zhang, Qing Yi. Compiler-driven approach for automating nonblocking synchronization in concurrent data abstractions // Concurrency and Computation: Practice and Experience. – 2023. – Vol. 36, Iss. 5. – P. 45-49.
4. Braedy Kuzma, Ivan Korostelev. Fast matrix multiplication via compiler-only layered data reorganization and intrinsic lowering // Software: Practice and Experience. – 2023. – Vol. 53, Iss. 9. – P. 87-93.
5. Matt Windsor, Alastair F. Donaldson High-coverage metamorphic testing of concurrency support in C compilers // Software Testing, Verification and Reliability. – 2022. – Vol. 32, Iss. 4. – P. 130-134.
6. Misun Yu. An operator scheduler for deep-learning compilers supporting multiple heterogeneous processing units // ETRI Journal. – 2023. – № 4. – P. 97-105.

## BASIC COMPILER ALGORITHMS FOR PARALLEL PROGRAMMES

**Albatsha Ahmad Muhammad Husain, Senior Lecturer**  
**National Research University Higher School of Economics**  
**(Russia, Moscow)**

**Abstract.** *Parallel programs have become more and more important in recent years because of the rapid growth of information arrays and as a consequence the necessity of fast processing of large amounts of data. At the same time, traditional compilation methods developed for sequential programs do not guarantee correctness (sequential consistency) of compiler's transformations when applied to parallel programs. In this connection the article is devoted to the analysis of basic compiler algorithms for such programs allowing to apply parallelisation and optimisation methods for code creation.*

**Keywords:** *code, compiler, algorithm, parallel program.*