Разработка препроцессора языка программирования python для распределенной обработки данных

В.С. Симонов[1], М.С. Хайретдинов[1,2]

[1]*Новосибирский государственный технический университет (Новосибирск, Россия)*
[2]*Институт вычислительной математики и математической геофизики СО РАН (Новосибирск, Россия)*

Аннотация

Постановка проблемы. Появление парадигм распределенных вычислений потребовало эволюции методов разработки программного обеспечения, более того, возникла необходимость эффективного преобразования традиционно последовательных приложений в распараллеливаемые и масштабируемые архитектуры. Это преобразование является ключевым в использовании вычислительной мощности современных многоядерных и распределенных систем. Python, с его обширной экосистемой и простотой, становится мощным языком, который облегчает этот переход. Результаты. В этой статье рассматриваются методологии и разработка инструмента препроцессора Python, предназначенного для преобразования последовательных приложения в формы, подходящие для крупномасштабных вычислительных сред. Практическая значимость. Благодаря подробному анализу и внедрению мы демонстрируем использование функций Spark и сторонних библиотек для автоматизации этого перехода, что делает распределенные вычисления более доступными для более широкого круга приложений и разработчиков. Препроцессор включает в себя набор правил, которые определяют различные шаблоны ввода кода (например, циклический переход по списку) и преобразуют код, пригодный для платформ параллельной обработки данных. Результаты. В данной статье представлен препроцессор Python, предназначенный для автоматизации преобразования из последовательных приложений на Python в те, которые используют возможности распределенных вычислительных сред.

Ключевые слова

Формальные языки, распределенные вычисления, параллельное программирование, python,

Введение

В области разработки программного обеспечения переход от последовательных или однопоточных приложений к приложениям, использующим распределенные вычислительные ресурсы, таким как многопоточные, параллельные или облачные среды, сопряжен с рядом трудностей. Тем не менее, это необходимая эволюция, позволяющая справляться с растущими объемами данных и вычислительными требованиями современных приложений. Последовательные приложения работают

7^{th} International Workshop on Information, Computation, and Control Systems for Distributed Environments (ICCS-DE 2025), July 7–11, 2025, Irkutsk, Russia
v.s.simonov@corp.nstu.ru (Simonov V.); marat@opg.sscc.ru (Khairetdinov M.)
0009-0004-0452-7819 (Simonov V.)

ICCS-DE 2025 Workshop Proceedings (iccs-de.icc.ru)
DOI: 10.47350/ICCS-DE.2025.14

поэтапно, выполняя одну операцию за раз. Этот подход, несмотря на свою простоту, становитя все более неадекватным для решения задач, требующих обработки больших наборов данных или выполнение сложных вычислений в разумные сроки [1].

MapReduce - популярная модель для разработки распределенных приложений. Она отличается разнообразием и высокой эффективностью работы с такими реализациями [2,3,4,5]. Переход к парадигмам распределенных вычислений, таким как параллельная обработка и облачные вычисления, может значительно сократить время выполнения за счет разделения задач на более мелкие блоки, которые могут выполняться одновременно.

Распределенные вычисления охватывают целый ряд архитектур, включая параллельные, грид-, облачные и распределенные вычисления. Каждая парадигма обладает уникальными преимуществами в обработке обширных вычислений и больших наборов данных на нескольких вычислительных модулях.

Роль Python в параллельных и распределенных вычислениях определяется его богатым набором библиотек и фреймворков, таких как PySpark, интерфейс передачи сообщений (MPI) для Python (mpi4py), Dask и Joblib. Эти инструменты абстрактны сложность взаимодействия и управления процессами делает Python идеальным языком для разработки масштабируемых приложений.

Последовательные приложения, обычно предназначенные для однопоточных сред, не используют многоядерные современные процессоры или возможности распределенных систем, что приводит к неэффективным вычислениям. Преобразование этих приложений в параллельную или распределенную архитектуру может значительно повысить производительность и масштабируемость.

Распределенные вычисления охватывают целый ряд архитектур, включая параллельные, грид-, облачные и распределенные вычисления. Каждая парадигма обладает уникальными преимуществами в обработке обширных вычислений и больших наборов данных на нескольких вычислительных модулях.

Роль Python в параллельных и распределенных вычислениях определяется его богатым набором библиотек и фреймворков, таких как PySpark, интерфейс передачи сообщений (MPI) для Python (mpi4py), Dask и Joblib. Эти инструменты абстрактны сложность взаимодействия и управления процессами делает Python идеальным языком для разработки масштабируемых приложений.

1. Постановка задачи

Преобразование последовательного приложения в распределенное вычислительное приложение вручную сопряжено с преодолением ряда трудностей:

1. Сложность кода: параллельный или распределенный код создает сложности, связанные с такими проблемами, как синхронизация, совместное использование данных и шаблоны взаимодействия.
2. Сложность отладки: отладка параллельных и распределенных систем по своей сути является более сложной задачей из-за параллельных операций и проблем с синхронизацией.

3. Знание инфраструктуры: разработчики должны иметь глубокое представление о целевой крупномасштабной вычислительной среде.
4. Оптимизация производительности: простое преобразование распределенный или параллельный запуск приложения не гарантирует повышения производительности; требуется тщательная оптимизация.

Для решения этих задач мы предлагаем препроцессор Python, который позволяет абстрагироваться от сложностей преобразования последовательного кода в формат, подходящий для распределенных вычислений.

2. Структура препроцессора и реализация

Препроцессор анализирует последовательный код на языке Python, находит фрагменты, которые можно распределить и выполняет преобразование кода, включая соответствующие библиотеки и внося необходимые изменения.

2.1. Обзор

Препроцессор Python, разработанный в данной работе, представляет собой инструмент, который анализирует и преобразует код Python, написанный для последовательного выполнения, в форму, которая может быть выполнена в параллельной или распределенной среде. Его базовая архитектура показана на рисунке 1.

Он включает в себя функции анализа кода, распознавания образов и автоматического преобразования кода. Препроцессор, предназначен для преобразования входного кода в эффективный код в рамках парадигмы MapReduce, поддерживаемый платформой PySpark. Конвейер выполнения состоит из трех основных модулей, архитектура которых будет описана далее.

Анализатор программ. Данный начальный этап включает в себя разбор входного кода с помощью абстрактного синтаксического дерева (AST) и выполнение статического анализа программы. Цель этого этапа - определить сегменты кода, которые подходят для перевода. Для каждого идентифицированного сегмента анализатор программ выполняет две ключевые задачи:

1. Создание описания пространства поиска, используя высокоуровневое промежуточное представление (IR). Это описание направляет процесс синтеза в следующем модуле , определяя границы и правила для создания корректной сводки программы.
2. Генерация условий проверки (VC), чтобы гарантировать, что сводная информация о синтезированной программе семантически эквивалентна исходному входному коду, тем самым гарантируя правильность процесса преобразования.

Поиск эквивалента. На этом этапе препроцессор синтезирует сводки программ на основе пространства поиска и VC, предоставляемых анализатором программ [10]. Данный модуль воплощает основную логику для преобразования высокоуровневой программной логики в форму, которая легко преобразуется в исполняемый код для

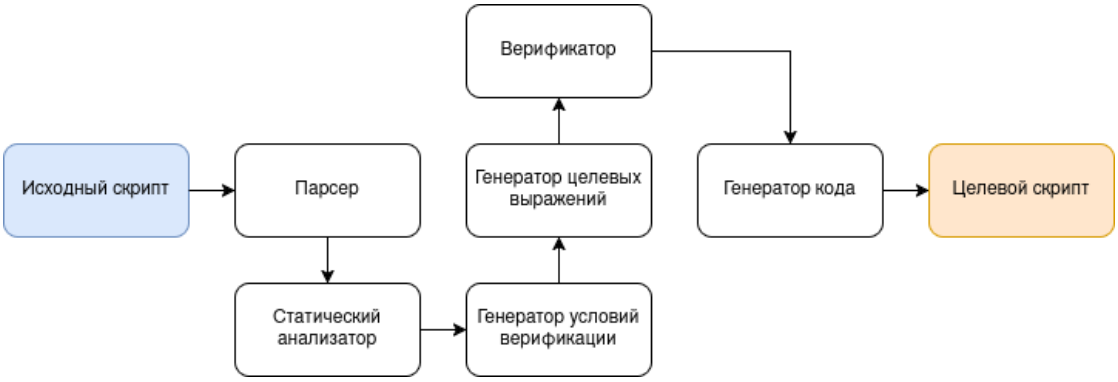

Figure 1: Архитектура системы.

платформ параллельных вычислений. В нем используется алгоритм поэтапного синтеза для эффективной навигации в пространстве поиска, гарантирующий, что процесс поиска корректных сводок программ будет быстрым и точным. Цель состоит в том, чтобы создать сводку, отражающую суть оригинала вычисления таким образом, чтобы обеспечить максимальную эффективность при развертывании в рамках платформы MapReduce.

Генерация кода. После того, как найдено и проверено краткое описание подходящей программы, последним шагом является преобразование этого краткого описания в исполняемый код. Этот модуль легко адаптируется и поддерживает генерацию кода для трех основных платформ MapReduce: Spark, Hadoop. Такое разнообразие позволяет препроцессору быть универсальным и применимым в различных вычислительных средах. Кроме того, этот модуль отвечает за генерацию кода, который отслеживает статистику данных во время выполнения. Эта функция позволяет осуществлять динамический выбор между различными стратегиями внедрения на основе показателей производительности в реальном времени, что еще больше оптимизирует эффективность выполнения переведенных программ [6].

Вместе эти модули образуют комплексный конвейер для системы препроцессора, предназначенный для оптимизации процесса перехода от высокоуровневого кода к эффективному, распараллеливаемому исполняемому коду, подходящему для распределенных задач обработки данных. Этот подход делает упор на автоматизацию в обеспечении семантической эквивалентности за счет использования условий проверки и направлен на оптимизацию производительности во время выполнения путем адаптивного выбора между различными стратегиями выполнения на основе статистики текущих данных.

2.2. Восходящая верификация

Восходящая верификация [11, 12] представляет собой методологию, направленную на анализ семантики программного кода путём его преобразования в абстрактные спецификации высокого уровня. Данный подход заключается в генерации

постусловий для отдельных участков кода, формально описывающих их влияние на выходные переменные. Разрабатываемый язык спецификаций ориентирован на решение двух ключевых задач:

1. Формирование простых абстракций, совместимых с целевой предметно-ориентированной платформой (DSL). Это означает, что отбрасываются корректные абстракции, не поддающиеся преобразованию в синтаксис целевого языка. Соответственно, язык спецификаций должен избегать конструкций, не имеющих прямых аналогов в DSL.
2. Генерация содержательных абстракций, поддерживающих параллельную обработку данных. Из рассмотрения исключаются спецификации, ориентированные исключительно на последовательные вычисления.

2.3. Генерация кода

Препроцессор также включает механизмы для выборки входных данных и динамического переключения между исходным кодом и оптимизированной версией на основе показателей производительности.

Это объяснение подчеркивает логику препроцессора и его подход к автоматизации оптимизации кода Python для распределенных вычислительных сред, предоставляя как общий обзор его возможностей, так и некоторое представление о технических деталях, связанных с его работой. Препроцессор не только упрощает переход от автономного кода на Python к оптимизированной распределенной версии но также включает в себя ряд функциональных возможностей, предназначенных для повышения производительности и обеспечения надежности в различных вычислительных средах.

Препроцессор разумно распределяет данные по узлам, принимая во внимание размер, сложность и вычислительные требования каждой задачи, тем самым сводя к минимуму накладные расходы и максимально повышая эффективность обработки данных. Используя базовую архитектуру распределенных вычислительных платформ, таких как Spark, этот инструмент реализует стратегии обеспечения отказоустойчивости, такие как информация о происхождении для RDD, что позволяет вычислениям корректно восстанавливаться после сбоев. отказы узлов без существенных потерь производительности.

3. Результаты

В качестве примера того, как работает препроцессор, приведем код из листингов 1 и 2. В листинге 1 функция select предназначена для поиска по списку записей (предполагается, что они хранятся в переменной с именем table) в поисках тех записей, первый столбец которых соответствует указанному ключу. Когда найдено совпадение, эта запись добавляется в список результатов, который возвращается в конце работы функции.

Listing 1: Выбор записи из таблицы.

```
def select(table, key):
    result = []
    for record in table:
        if record.columns[0] == key:
            result.append(record)
    return result
```

В листинге 2 приведен код после запуска препроцессора. И теперь код стал пригоден для выполнения на PySpark.

Listing 2: Предварительно обработанный код для PySpark.

```
def select(rdd, key):
    result = rdd.flatMap(lambda data_index:
        [data_index] if data_index.columns[0]
        == key else [])
    return result.collect()
```

3.1. Производительность

Здесь описан один из тестов. Среднее время обработки каждого фрагмента кода, составляющее 7,3 минуты , в сравнении со значительно меньшим средним временем, составляющим 1,8 минуты, свидетельствует о неравномерном распределении времени компиляции. Это указывает на то, что, хотя многие компиляции выполняются довольно быстро, некоторые из них занимают непропорционально больше времени, возможно , из-за сложных задач оптимизации или проверки, которые препроцессор выполняет для определенных тестов.

Заключение

Разработанный в этом исследовании препроцессор Python представляет собой значительный шаг на пути к тому, чтобы сделать распределенные вычисления более доступными для широкого спектра приложений. Автоматизируя процесс преобразования, этот инструмент не только упрощает переход от последовательной архитектуры к параллельной и распределенной, но и открывает новые возможности для повышения производительности и масштабируемости. По мере дальнейшего развития крупномасштабных вычислений такие инструменты, как этот препроцессор, будут играть решающую роль, позволяя приложениям использовать весь потенциал современных вычислительных инфраструктур.

References

[1] J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM 51, 01.01.2008, c. 107–113.

[2] Apache Spark 2025. (2025). [Электронный ресурс]. URL: https://spark.apache.org (дата обращения 28.03.2025).
[3] A.V. Aho, M.S. Lam, R. Sethi, J. D. Ullman. Compilers: Principles, Techniques, and Tools (2Nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, июнь 2006, с. 38–39.
[4] W. Blume, R. Eigenmann, J. Hoeflinger, D.A. Padua, P. Petersen, L. Rauchwerger, and P. Tu. Automatic Detection of Parallelism: A grand challenge for high performance computing. IEEE PDT 2, 3 1994, с. 37–46. 10.18127/j19997493-201904-08
[5] Simonov V. S. Automatic decomposition of a sequential algorithm for MapReduce Frameworks / V. S. Simonov, M. S. Khairetdinov. – ISBN 978-1-6654-6480-2. DOI 10.1109/SIBIRCON56155.2022.10017034. – Text : electronic // International multi-conference on engineering, computer and information sciences (SIBIRCON–2022) : proc., Novosibirsk–Yekaterinburg, 11–13 Nov. 2022. – IEEE, 2022. – с. 1780–1783.
[6] A. Alexandrov, A. Katsifodimos, G. Krastev, V. Markl. 2016. Implicit Parallelism Through Deep Language Embedding. SIGMOD Rec. 45, 1 (Июнь 2016), с. 53–54.
[7] ROSE. An open source compiler infrastructure. [Электронный ресурс]. URL: http://rosecompiler.org/ROSE$_{H}TML_{R}eference/index.html$(18.03.2025).
[8] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.
[9] R. Bodik, B. Jobstmann. 2013. Algorithmic program synthesis: introduction. International Journal on Software Tools for Technology Transfer 15 (2013), с. 399–403.
[10] Alvin Cheung, Armando Solar-Lezama and Samuel Madden (2013): Optimizing Database-backed Applications with Query Synthesis. In: Proceedings of the 34th ACM SIGPLAN Conference on Pro- gramming Language Design and Implementation, PLDI '13, ACM, New York, NY, USA, с. 3–14, doi:10.1145/2491956.2462180.
[11] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky and Armando Solar-Lezama (2016): Verified Lifting of Stencil Computations. SIGPLAN Not. 51(6), с. 711–726, doi:10.1145/2980983.2908117.