IMPERATIVE CODE CONVERSION METHOD FOR PARALLEL DATA PROCESSING PLATFORMS

V. S. Simonov*, M. S. Khairetdinov*,**

*Novosibirsk State Technical University,
630087, Novosibirsk, Russia
**Institute of Computational Mathematics and Mathematical Geophysics SB RAS,
630090, Novosibirsk, Russia

DOI: 10.24412/2073-0667-2023-3-68-80
EDN: HGORYY

There are many data processing platforms that allow sequential programs to access parallel processing capabilities. To benefit from the advantages of such platforms, existing code has to be rewritten into domain-specific languages that each platform supports. This transformation, a tedious and error-prone process, also requires developers to choose the right platform that optimizes performance based on a specific workload.

This article describes a formal method, the result of which on imperative code is code suitable for execution in a parallel data processing system, for example, Hadoop, implementing the MapReduce paradigm. Given a sequential code fragment, a method is used to output a high-level summary expressed in our the language of the program specification, which is then compiled for execution in Apache Spark [1]. We demonstrate that the method allows you to convert imperative code into suitable for execution on the Apache Spark platform. Translated results are executed 1.3 times faster on average than sequential implementations, and also scale better for large datasets.

As computing becomes more ubiquitous, storage becomes cheaper, and data collection tools become more sophisticated, more data is being collected today than ever before. Data-driven advances are becoming increasingly common in various scientific fields. Thus, efficient analysis and processing of huge data sets is a huge computational task.

For processing very large data sets, many parallel data processing platforms have been developed [1–5], and new ones continue to be developed [5–7]. Most parallel data processing frameworks come with domain-specific optimizations , which are provided either through the library application programming interface (API) [1–4, 6, 7], or using a high-level domain-specific language: domain-specific language (DSL), so that users can express their calculations [5, 8]. Calculations expressed using such API or DSL calls are more efficient due to the optimization of platforms for a specific domain [8–11].

However, many of the problems associated with this approach often make frameworks related to a specific subject area inaccessible to non-specialists, such as researchers studying physical or social sciences. First, domain-specific optimization for various workloads requires an expert to determine in advance the most appropriate structure for a given piece of code. Secondly, end users often have to learn new APIs or DSLs [1–3, 6, 7, 12] and transform existing code to take advantage of the advantages provided by some platforms. This requires not only considerable time and resources, but is also fraught with errors in the code. Moreover, even users who want to transform their applications must first understand the purpose of the code that could have been written by others, and manually written low-level optimizations in the code often hide high-level intentions. Finally, even after learning new APIs and rewriting code, newly emerging frameworks often turn newly written code into outdated

applications. Users then have to repeat this process in order to keep up with new advances, which requires considerable time, which would be better spent on promoting scientific discoveries.

One way to improve the availability of parallel data processing platforms involves creating compilers that automatically convert applications written in common general-purpose languages (such as C, Java or Python) into high-performance parallel processing applications such as Hadoop or Spark. These compilers allow users to write their applications in familiar general-purpose languages and allow the compiler to reassign parts of their code to high-performance DSL [13–15]. Then applications can use the performance of these specialized frameworks without the additional cost of learning to program individual DSLs. But such compilers do not exist for all cases, and their creation can be very difficult.

Key words: imperative code, parallel data processing.

References

1. Apache Spark. [Electron res.]: `https://spark.apache.org`. Accessed: 2023-01-19.
2. Apache Hadoop. [Electron res.]: `http://hadoop.apache.org`. Accessed: 2023-01-19.
3. Apache Storm. [Electron res.]: `http://storm.apache.org`. Accessed: 2023-01-19.
4. GraphLab Create. [Electron res.]: `https://dato.com/`. Accessed: 2023-01-20.
5. MongoDB. [Electron res.]: `https://www.mongodb.org`. Accessed: 2023-01-19.
6. Akidau T., Bradshaw R., Chambers C., Chernyak S., Fernandez-Moctezuma R. J., Lax R., McVeety S., Mills D., Perry F., Schmidt E., Whittle S. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing // Proceedings of the VLDB Endowment 8, 2015. P. 1792–1803.
7. TensorFlow. [Electron res.]: `http://tensorflow.org/`. Accessed: 2023-01-20.
8. Ragan-Kelley J., Barnes C., Adams A., Paris S., Durand F., Amarasinghe S. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines // Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2013. PLDI'13, ACM, New York, NY, USA. P. 519–530, DOI: 10.1145/2491956.2462176.
9. Apache Hive. [Electron res.]: `http://hive.apache.org`. Accessed: 2023-01-20.
10. Solar-Lezama A., Arnold G., Tancau L., Bodik R., Saraswat V., Seshia S. Sketching Stencils // Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007. PLDI '07, ACM, New York, NY, USA. P. 167–178, DOI: 10.1145/1273442.1250754.
11. Arvind K. Sujeeth A.K., Kevin J. Brown K.J., Lee H., Rompf T., Chafi H., Odersky M., Olukotun K. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific. 2014.
12. Hoare C. A. R. An Axiomatic Basis for Computer Programming // Communications of the ACM 12(10), 1969. P. 576–580, DOI: 10.1145/363235.363259.
13. Cheung A., Solar-Lezama A., Madden S. Optimizing Database-backed Applications with Query Synthesis // Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2013. PLDI '13, ACM, New York, NY, USA. P. 3–14, DOI: 10.1145/2491956.2462180.
14. Kamil S., Cheung A., Itzhaky S., Solar-Lezama A. Verified Lifting of Stencil Computations // SIGPLAN Not. 2016. 51(6), P. 711–726, DOI: 10.1145/2980983.2908117.
15. Radoi C., Fink S. J., Rabbah R., Sridharan M. Translating Imperative Code to MapReduce // Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '14, 2014. ACM, New York, NY, USA. P. 909–927, DOI: 10.1145/2660193.2660228.

16. Ernst M. D., Perkins J. H., Guo P. J., McCamant S., Pacheco C., Tschantz M. S., Xiao C. The Daikon System for Dynamic Detection of Likely Invariants. Sci. Comput. Program. 2007.

17. Srivastava S., Gulwani S. Program Verification Using Templates over Predicate Abstraction // Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, '09, 2009. ACM, New York, NY, USA. P. 223–234, DOI: 10.1145/1542476.1542501.

МЕТОД ПРЕОБРАЗОВАНИЯ ИМПЕРАТИВНОГО КОДА ДЛЯ ПЛАТФОРМ ПАРАЛЛЕЛЬНОЙ ОБРАБОТКИ ДАННЫХ

В. С. Симонов*, М. С. Хайретдинов*,**

*Новосибирский государственный технический университет,
630087, Новосибирск, Россия
**Институт вычислительной математики и математической геофизики СО РАН,
630090, Новосибирск, Россия

УДК 004.89
DOI: 10.24412/2073-0667-2023-3-68-80
EDN: HGORYY

Существует множество платформ для обработки данных, которые позволяют последовательным программам получать доступ к возможностям параллельной обработки. Чтобы извлечь выгоду из преимуществ таких платформ, существующий код приходится переписывать на языки, специфичные для конкретной предметной области, которые поддерживает каждая платформа. Данное преобразование — утомительный и подверженный ошибкам процесс — также требует от разработчиков выбора нужной платформы, которая оптимизирует производительность с учетом конкретной рабочей нагрузки.

В данной статье описывается формальный метод, результатом применения которого на императивном коде являются эквивалентные инструкции, пригодные для исполнения в системе параллельной обработки данных, например, Hadoop, реализующей парадигму MapReduce. Метод применяется для вывода высокоуровневой сводки, выраженной на нашем языке спецификации программы, которая затем компилируется для выполнения в Apache Spark [1]. Было показано, что метод позволяет преобразовать императивный код в пригодный для исполнения на платформе Apache Spark. Приведенные результаты выполняются в среднем в 3,3 раза быстрее, чем последовательные реализации, а также лучше масштабируются для больших наборов данных.

Ключевые слова: императивный код, параллельная обработка данных.

Введение. Поскольку вычислительная техника становится все более повсеместной, хранилище — более дешевым, а инструменты сбора данных — более сложными, сегодня собирается больше данных, чем когда-либо прежде. Достижения, основанные на данных, становятся все более распространенными в различных научных областях. Таким образом, эффективный анализ и обработка огромных наборов данных представляют собой трудоемкую вычислительную задачу.

Для обработки очень больших наборов данных было разработано множество платформ параллельной обработки данных [1–5], и новые продолжают дорабатываться [5–7]. Большинство фреймворков параллельной обработки данных поставляются с оптимизациями для конкретной предметной области, которые предоставляются либо через библиотечный application programming interface (API) [1–4, 6, 7], либо с помощью высокоуровневого языка, специфичного для конкретной предметной области: domain-specific language (DSL),

чтобы пользователи могли выражать свои вычисления [5, 8]. Программный код, выраженный с помощью таких вызовов API или DSL, более эффективен благодаря оптимизации платформ для конкретной предметной области [8–12].

Один из способов улучшить доступность платформ параллельной обработки данных включает создание транслятора, который автоматически преобразует код, написанный на распространенных языках общего назначения (таких как C, Java или Python), в код, пригодный для высокопроизводительных приложений параллельной обработки; Hadoop или Spark. Данные компиляторы позволяют пользователям создавать свои приложения, используя знакомые языки общего назначения, а компилятору — переназначать части их кода на высокопроизводительные DSL [13–15]. В результате коду становится доступна вся производительность вычислительного кластера, а пользователь — освобождается от необходимости изучать платформы параллельной обработки данных. Подобные компиляторы пригодны не для всех вариантов исходного кода, и их создание может оказаться очень сложным.

1. Постановка задачи. В данной статье демонстрируется применение нового метода для автоматического преобразования последовательных фрагментов кода в MapReduce. Под фрагментами кода подразумеваются конструкции на базе цикла. В качестве входных данных метод требует фрагменты программы, написанные на языке общего назначения, и использует программный синтез для автоматического поиска доказуемо корректных сводок кода. Сводки — выраженные на нашем языке спецификации программы — представляют семантику фрагмента входного кода. Найденные сводки затем используются для перевода исходного входного кода в целевой высокопроизводительный DSL.

Концепция восходящего метода проверок корректности кода ранее применялась к приложениям баз данных [13] и трафаретным вычислениям [14]. В данной работе применяется подход к преобразованию императивного кода последовательной обработки данных для использования платформ параллельной обработки данных Apache Spark. Постановка задачи известна и впервые была предложена в компиляторе MOLD [15], который преобразует последовательный Java-код для выполнения в Apache Spark. MOLD использует заранее определенные правила перезаписи для поиска в пространстве эквивалентных реализаций Apache Spark. Он сканирует входной код на наличие шаблонов, затем подбирает правила перезаписи, — подход, сопряженный со многими ограничениями. Например, это требует априорного определения сложных правил перезаписи, которые могут быть чрезвычайно хрупкими при изменении шаблона кода. Для сравнения, настоящий метод анализирует семантику программы, а не синтаксис программы, что делает его устойчивым к изменениям шаблона кода. Мы также не полагаемся на заранее определенные правила перевода и, таким образом, можем находить новые фрагменты кода, о существовании которых пользователь никогда не подозревал.

2. Структура метода. В данном разделе раскрывается структура метода преобразования последовательных фрагментов кода в задачи MapReduce. Мы рассмотрим концепцию восходящей верификации в 2.1 и опишем язык спецификации программы, который используется для выражения резюме программы. В 2.2 мы объясняем, каким образом проверяется, что идентифицированные фрагменты кода сохраняют программную семантику исходного фрагмента кода. В 2.3 демонстрируется процесс поиска, который применяется для поиска сводок программ, в то время как в 2.3.1 объясняется, как происходит выборка подходящих фрагментов кода для трансляции. Наконец, в 2.3.3 демонстрируется

конечный результат (генерация кода) после того, как было выведено краткое описание программы.

2.1. *Восходящая верификация.* Восходящая верификация [13, 14] — общая техника, которая выводит семантику кода, написанного на языке общего назначения, «поднимая» его до сводок, выраженных с использованием языка высокого уровня. Метод предполагает определение резюме фрагментов кода на нашем языке спецификации программы в форме постусловий, которые описывают влияние фрагмента кода на его выходные переменные. Целями нашего языка спецификации программы являются:

— Создание тривиальных сводок, которые можно перевести на целевую платформу DSL. Это исключает действительные сводки, которые не могут быть переведены. Следовательно, язык должен опускать конструкции, которые не могут быть легко переведены на целевой язык.

— Создание нетривиальных сводок, отвечающих за параллельную обработку данных. Очевидно, что это исключает сводки, которые ответственны за последовательные вычисления.

Имея в виду данные цели, предполагаемые сводки должны иметь вид:

$$\forall \in outputVariables.v = reduce(map(data, f_m), f_r)[id_y], \tag{1}$$

где *data* — это повторяющийся набор входных данных. Функция map выполняет итерацию по данным при вызове функции f_m для каждого элемента. f_m принимает в качестве входных данных элемент из данных и генерирует потенциально несколько пар «ключ-значение». Затем, map собирает и возвращает пары «ключ-значение», сгенерированные вызовами f_m. Функция reduce принимает эти пары «ключ-значение», группирует их по ключу и вызывает f_r для каждого ключа и всех значений, соответствующих этому ключу. Функция f_r агрегирует все значения для данного ключа и выдает единственную пару «ключ-значение». Как и map, reduce собирает все агрегированные пары «ключ-значение» и возвращает ассоциативный массив, который сопоставляет идентификатор каждой переменной с ее конечным значением. Идентификатор переменной — это уникальный идентификатор, который присваивается каждой выходной переменной. Требуется, чтобы резюме (т. е. постусловия) имели форму, описанную в (1) для удобства трансляции в задачи Spark.

2.2. *Проверка эквивалентности.* Найденные сводки должны быть семантически эквивалентны фрагменту входного кода. Устанавливается достоверность выведенных постусловий, используя условия проверки в стиле логики Хоара [12], которые представляют собой предварительные условия фрагмента кода, которые должны быть истинными, чтобы установить постусловие того же фрагмента кода при всех возможных исполнениях. Генерировать условия проверки для простых операторов присваивания и условных выражений несложно. Например, рассмотрим императивное программное утверждение $x := y+3$. Чтобы показать, что возможное постусловие $x > 10$ является допустимым постусловием, мы должны доказать, что $y + 3 > 10$ истинно, прежде чем оператор будет выполнен. В данном случае, $y + 3 > 10$ называется условием проверки для этого постусловия. Вычислить условие проверки легко для простых утверждений. Однако для цикла вычисление условий проверки становится более сложным, поскольку необходим инвариант цикла. Инвариант цикла — это гипотеза, которая утверждает, что постусловие истинно независимо от того, сколько раз повторяется цикл. Логика Хоара утверждает, что следующие три утверждения должны выполняться для:

1) $\forall \sigma.preCondition(\sigma) \rightarrow loopInvariant(\sigma)$

2) $\forall\sigma.loopInvariant(\sigma) \wedge loopCondition(\sigma) \rightarrow loopInvariant(body(\sigma))$
3) $\forall\sigma.loopInvariant(\sigma) \wedge \neg loopCondition(\sigma) \rightarrow postCondition(\sigma)$

Утверждение 1 означает, что инвариант цикла должен быть истинным, когда предварительное условие истинно для всех состояний программы (σ), т. е. инвариант цикла должен быть истинным перед входом в цикл. Утверждение 2 означает, что для всех состояний σ инвариант цикла должен быть истинным перед входом в цикл, и что для всех возможных состояний программы σ — цикл продолжается. Инвариант цикла остается истинным после еще одного выполнения цикла body; (здесь body(σ) возвращает новое состояние программы после выполнения тела цикла в σ). Утверждение 3 означает, что если инвариант цикла истинен и если цикл завершается, то постусловие должно быть истинным для всех возможных состояний программы.

Две проблемы влияют на идентификацию постусловий для фрагментов кода, которые включают циклы. Во-первых, должны быть синтезированы как инварианты цикла, так и постусловие. Однако, в отличие от предыдущей работы по поиску инвариантов [16, 17], сейчас необходимо найти инварианты цикла, которые логически достаточно сильны только для того, чтобы установить обоснованность постусловия, т. е. те, которые удовлетворяют утверждению 3. Этого просто достичь благодаря конкретной форме постусловия. Кроме того, установление достоверности найденных инвариантов и постусловий требует проверки всех возможных состояний программы, что усложняет задачу синтеза.

2.3. *Поиск сводок.* Метод предполагает стремление вывести резюме для каждого фрагмента кода, где каждое резюме является постусловием формы, описанной в 2.1. В данном разделе описывается, каким образом используется синтез для поиска постусловий и инвариантов цикла, которые им требуются для доказательства правильности постусловий.

$$preCondition(hR,hG,hB,i) \equiv hR = [0..0] \wedge hB = [0..0] \wedge i = 0 \tag{2}$$

$$\begin{aligned} &postCondition(data,hR,hG,B) \equiv \\ &\quad \forall 0 <= j < hR.length.hR[j] = reduce(map(data, f_m), f_r)[(0,j)] \\ &\quad \wedge \forall 0 <= j < hR.length.hR[j] = reduce(map(data, f_m), f_r)[(1,j)] \\ &\quad \wedge \forall 0 <= j < hR.length.hR[j] = reduce(map(data, f_m), f_r)[(2,j)] \end{aligned} \tag{3}$$

$$\begin{aligned} &loopInvariant(data,hR,hG,B) \equiv \\ &\quad \forall 0 <= j < hR.length.hR[j] = reduce(map(data[0:i], f_m), f_r)[(0,j)] \\ &\quad \wedge \forall 0 <= j < hR.length.hR[j] = reduce(map(data[0:i], f_m), f_r)[(1,j)] \\ &\quad \wedge \forall 0 <= j < hR.length.hR[j] = reduce(map(data[0:i], f_m), f_r)[(2,j)] \end{aligned} \tag{4}$$

2.3.1. *Генерация условий верификации.* В 2.2 мы объяснили три условия проверки, которым должна удовлетворять обобщенная сводка. Данные условия проверки включают предварительное условие, постусловие и инвариант цикла для фрагмента кода. Предварительные условия генерируются путем извлечения с помощью статического анализа программы состояния (значения входных и выходных переменных) непосредственно перед началом выполнения цикла. Когда значение переменной перед запуском цикла не может быть определено, метод генерирует новую переменную для представления начального

значения. Инвариант цикла имеет форму, аналогичную постусловию (см. 2.1); однако, в отличие от постусловия, которое вызывает map и reduce для всей коллекции данных, инвариант цикла вызывает map и reduce только для подмножества коллекции, которое до сих пор было пройдено циклом. Кроме того, инвариант цикла включает в себя выражение, описывающее поведение счетчиков цикла.

Выражения 2, 3 и 4 являются предварительным условием, постусловием и инвариантом цикла, сгенерированных для теста 3D-гистограммы. Функции, инвариантные к постусловию и циклу, описывают поведение, которое должно быть истинным для того, чтобы тела f_m и f_r были истинными. Например, постусловие гласит, что для каждого индекса j в $hR[j]$ значение должно быть эквивалентно значениям map и reduce функций в точке $(0,j)$.

2.3.2. *Определение пространства поиска.* В данном разделе описывается, каким образом определяется генерация грамматики, которая используется в процессе синтеза для построения функций f_m и f_r. Динамически генерируя грамматику для каждого фрагмента кода, ограничивается пространство сводок, в котором синтезатор должен выполнять поиск.

Напомним, что функция f_m принимает в качестве параметров набор входных данных и индекс в коллекцию и возвращает набор пар «ключ-значение»; конструирует тело f_m, используя операторы emit и условные обозначения. Текущий прототип не создает реализации f_m, которые включают циклы. Основываясь на наших вычислительных экспериментах, мы обнаружили, что использование того же количества инструкций emit в качестве выходных переменных во фрагменте кода хорошо работает в качестве отправной точки. Затем количество инструкций emit может быть увеличено, если решение не может быть найдено. В целом, однако, стоит придерживаться консервативного подхода, чтобы избежать реализаций с избыточными операторами emit, поскольку они генерируют ненужные данные в случайном порядке, что снижает производительность. Каждый оператор emit создает пару «ключ-значение»; ключом и значением может быть любое выражение, составленное одной из наших грамматик выражений или кортежей таких выражений. Функция f_r сводит все значения, выдаваемые map для данного ключа, к одному значению.

Генерируется грамматика выражений для каждого примитивного типа данных, найденного во фрагменте кода. Каждая грамматика может быть использована для генерации выражений, которые вычисляются до значения ее типа. Выражения сформулированы с использованием операторов и вызовов функций из исходного фрагмента кода. Входные переменные, цикл счетчики и литералы из фрагмента кода используются в качестве терминалов. Для арифметических типов синтезатору позволено генерировать новые константы. Кроме того, генерируется грамматика выражений для построения выражения свертки в f_r и выражения счетчика циклов в инварианте цикла.

Все составленные грамматики выражений ограничены заданным уровнем рекурсии, который может указать пользователь. Рекурсивная граница грамматики определяет, сколько раз синтезатору разрешается расширять нетерминалы при формулировании выражения. Если синтезатор не может найти решение, грамматики выражений могут быть постепенно расширены путем введения новых операторов и функций, которые не были найдены во фрагменте кода или увеличивают рекурсивную привязку к грамматике. В табл. 1 показана грамматика, составленная для теста 3D-гистограммы после 2 итераций расширения грамматики.

2.3.3. *Процедура поиска.* Несмотря на все ограничения пространства поиска, пространство возможных резюме остается большим. Поэтому, чтобы ускорить поиск, разделяется

Таблица 1

Правила вывода

$emit\_map$	::=	$emit(expr, expr) \mid if(bool\_expr)emit(expr, expr)$
$expr$	::=	$int\_expr \mid bool\_expr \mid (expr, expr)$
int_expr	::=	$int\_term \mid data[int\_expr] \mid int\_expr + int\_expr \mid int\_expr/int\_expr$
int_term	::=	$int\_lit \mid int\_cntr$
$bool\_expr$	::=	$true \mid false \mid int\_expr == int\_expr \mid bool\_expr \wedge bool\_expr$
f_m	::=	$emit\_map; emit\_map; emit\_map$
fld_exp	::=	$fld\_term \mid fld\_exp + fld\_exp$
fld_term	::=	$int\_lit \mid value \mid v$
$loop\_expr$	::=	$loop\_term <= loop\_term <= loop\_term$
$loop\_term$	::=	$loop\_cntr \mid int\_lit \mid data.length$
f_r	::=	$value = int\_lit; for(vinvalues)value = fld\_expemit(key, value)$

процесс проверки на две части: сначала он использует процедуру ограниченной проверки для поиска инвариантов-кандидатов и постусловий. Для инвариантов-кандидатов и постусловий, которые проходят процедуру ограниченной проверки, затем используется доказательство теоремы для установления надежности для всех входных состояний программы. Если программа проверки теоремы завершается неудачей (из-за тайм-аута) или возвращает unsat, синтезатор продолжает поиск нового резюме кандидата в том же пространстве поиска. Когда он больше не находит резюме кандидатов, синтезатор расширяет грамматику, чтобы увеличить пространство поиска. Он делает это либо путем добавления новых нетерминалов, увеличения рекурсивной границы для грамматики, либо путем увеличения количества отправлений, выполняемых f_m, как обсуждалось ранее. Параметры конфигурации, указанные пользователем, управляют этим итеративным расширением пространства поиска. В конце концов, синтезатор либо находит проверяемую сводку, исправляет или останавливает попытки преобразовать код. Предложенный метод также отделяет процедуру синтеза от формальной верификации и использует разные инструменты для каждой из двух подзадач. Эта методология хорошо работает на практике, сокращая время синтеза.

3. Оценка результатов. Здесь описывается наш прототип реализации настоящего метода и демонстрируются результаты, полученные в результате применения метода в соответствии различным критериям. Стоит упомянуть, что Hadoop/MapReduce ориентировано на пакетную обработку данных, а Spark — в т. ч. и на потоковую. В данном разделе сравниваются производительности последовательных тестов, аналогичных тестов с применением Hadoop и последовательных тестов, преобразованных с помощью настоящего метода (исполняемые Apache Spark).

3.1. *Контрольные показатели.* Мы оценили производительность метода по следующим четырем критериям. Данные тесты были взяты из набора тестов Phoenix [7] и представляют собой традиционные задачи, которые могут быть распараллелены путем преобразования с использованием парадигмы MapReduce.

— **Сложение**: сумма целочисленных значений элементов списка.

— **Подсчет слов**: подсчет числа вхождений каждого слова в файле.

— **Вхождение строки**: определяет, содержится ли набор из двух строк в основном тексте. Возвращается логическое значение для каждой строки в качестве выходных дан-

Таблица 2

Результаты экспериментов

Показатель	Анализ программы	Синтез	Итерации грамматики
Сложение	$<$ 1с	13с	1
Подсчет слов	$<$ 1с	44с	1
Вхождение строки	$<$ 1с	1406с	2
3D Гистограмма	$<$ 1с	2355с	2

ных. Как и подсчет слов, этот тест также выполняет итерацию по каждому слову во входном файле.

— **3D Гистограмма**: генерация трехмерной гистограммы, которая подсчитывает частоту каждого цветового компонента RGB в изображении. Выходные данные представляют собой массив для каждого цветового компонента, который содержит частоту каждого значения интенсивности.

Тесты считывают входные данные из текстового файла, сохраненного в HDFS. Для найденных решений Hadoop класс org.apache.hadoop.mapred.FileInputFormat используется для чтения и разделения данных между несколькими функциями разделения данных.

3.2. *Масштабируемость.* В табл. 2 показано среднее время (более 5 запусков), необходимое для обобщения сводных данных по каждому из четырех контрольных показателей. Компилятор синтезировал реализации Spark для всех тестов в течение часа. Более простые тесты, такие как суммирование и подсчет слов, были преобразованы менее чем за минуту и потребовали всего одной итерации генерации грамматики. Ни один тест не требовал более двух итераций для успешного выполнения, что показано в табл. 2.

Были созданы нетривиальные реализации, которые используют параллелизм, предлагаемый Hadoop MapReduce. Чтобы оценить качество оптимизации, мы сравнили производительность во время выполнения исходных последовательных реализаций с созданными реализациями. Результаты замеров производительности разработанного решения показаны на рис. 1. Представлены все четыре теста производительности. Однако данные реализации могут быть не самыми эффективными.

Листинг 1. Тест построения 3D гистограммы.

```
public static int[][] histogram(List<Pixel> image, int[] hR, int[] hG, int[] hB) {
    for (int i = 0; i < image.size(); i += 1) {
        int r = image.get(i).r;
        int g = image.get(i).g;
        int b = image.get(i).b;
        hR[r]++; hG[g]++; hB[b]++;
    }

    int[][] result = new int[3][];
    result[0] = hR;
    result[1] = hG;
    result[2] = hB;
    return result;
}
```

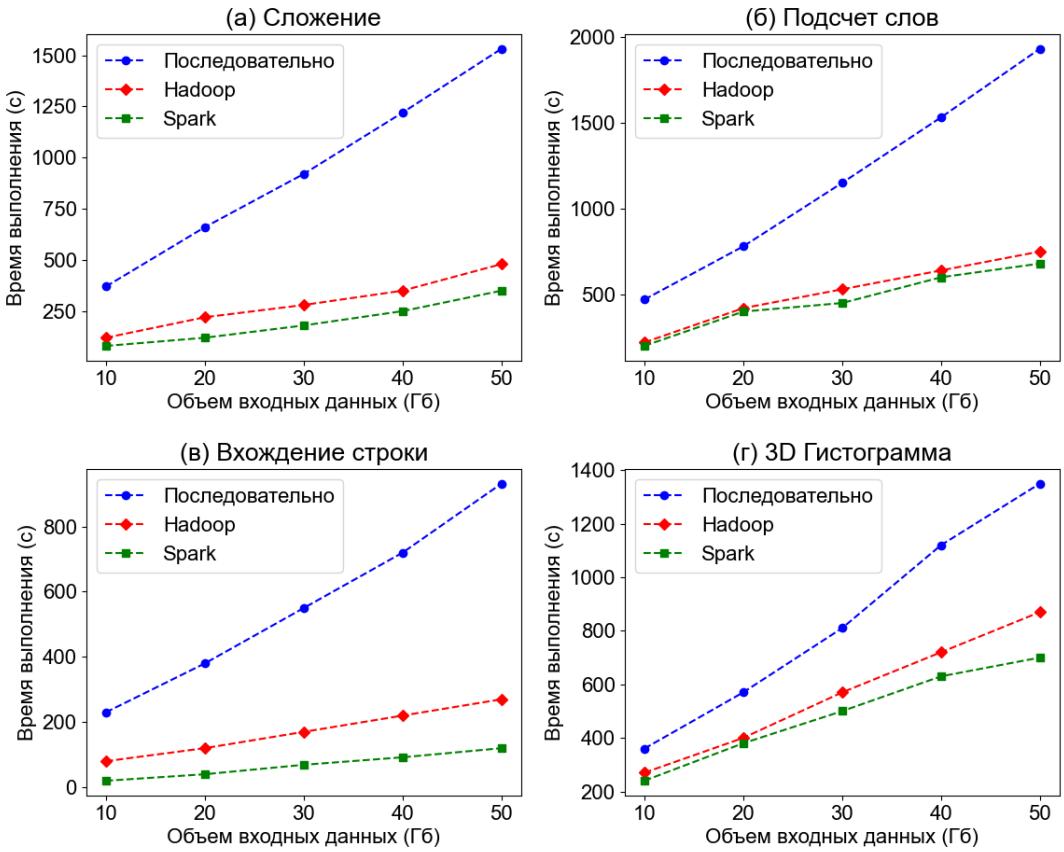


Рис. 1. Сравнение производительностей

Листинг 2. Результат работы транслятора.

```
public static int[][] histogram(JavaRDD<Pixel> rdd, int[] hR, int[] hG, int[] hB) {
    int i = 0;
    Map<Tuple2<Integer,Integer>,Integer> output = rdd.flatMapToPair(image_i -> {
        List<Tuple2<Tuple2<Integer,Integer>,Integer>> emits = new
            ArrayList<Tuple2<Tuple2<Integer,Integer>,Integer>>();
        emits.add(new Tuple2(new Tuple2(0, image_i.g),1));
        emits.add(new Tuple2(new Tuple2(1, image_i.b),1));
        emits.add(new Tuple2(new Tuple2(2, image_i.r),1));
        return emits.iterator();
    }).reduceByKey((v1,v2) -> v1 + v2).collectAsMap();
    for (Tuple2<Integer,Integer> output_i : output.keySet()) {
        if (output_i._1 == 0) {
            hG[output_i._2] = output.get(output_i);
        }
        if (output_i._1 == 1) {
            hB[output_i._2] = output.get(output_i);
        }
        if (output_i._1 == 2) {
            hR[output_i._2] = output.get(output_i);
        }
    }
```

```
    int[][] result = (int[][])(new int[3][]);
    ((int[][])result)[0] = (int[])hR;
    ((int[][])result)[1] = (int[])hG;
    ((int[][])result)[2] = (int[])hB;
    return (int[][])result;
}
```

Для теста построения 3D Гистограммы (листинг 1, 2) альтернативной реализацией Hadoop было бы выдавать для каждого пикселя во входных данных пары «ключ-значение» формы (интенсивность, цвет). Затем Hadoop сгруппировал бы данные по 256 значениям интенсивности. Агрегирование включало бы простой подсчет количества повторений каждого цвета (красный, зеленый или синий), что отображается для заданной клавиши.

Заключение. В данной работе был представлен метод, позволяющий автоматически переназначать императивный код для выполнения с помощью Apache Spark. Метод использует восходящую верификацию для преобразования фрагментов кода в исходной программе в высокоуровневое представление, которое затем может быть переведено для генерации эквивалентных задач Hadoop для распределенной обработки данных. Был протестирован прототип на основе разработанного метода, и оценена его производительность на нескольких тестах MapReduce. Наши эксперименты показывают, что метод может переводить все входные тесты, и созданные программы могут выполняться в среднем в 3,3 раза быстрее по сравнению с их последовательными аналогами. Тем не менее, реализация метода имеет проблемы. В табл. 2 видно, что существуют примеры входных данных, синтез которых занимает много времени. Наблюдается и различие между временем работы Hadoop и Spark, что объясняется использованием HDFS в первом случае.

Список литературы

1. Apache Spark. [Electron res.]: `https://spark.apache.org`. Accessed: 2023-01-19.
2. Apache Hadoop. [Electron res.]: `http://hadoop.apache.org`. Accessed: 2023-01-19.
3. Apache Storm. [Electron res.]: `http://storm.apache.org`. Accessed: 2023-01-19.
4. GraphLab Create. [Electron res.]: `https://dato.com/`. Accessed: 2023-01-20.
5. MongoDB. [Electron res.]: `https://www.mongodb.org`. Accessed: 2023-01-19.
6. Akidau T., Bradshaw R., Chambers C., Chernyak S., Fernandez-Moctezuma R. J., Lax R., McVeety S., Mills D., Perry F., Schmidt E., Whittle S. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing // Proceedings of the VLDB Endowment 8, 2015. P. 1792–1803.
7. TensorFlow. [Electron res.]: `http://tensorflow.org/`. Accessed: 2023-01-20.
8. Ragan-Kelley J., Barnes C., Adams A., Paris S., Durand F., Amarasinghe S. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines // Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2013. PLDI'13, ACM, New York, NY, USA. P. 519–530, DOI: 10.1145/2491956.2462176.
9. Apache Hive. [Electron res.]: `http://hive.apache.org`. Accessed: 2023-01-20.
10. Solar-Lezama A., Arnold G., Tancau L., Bodik R., Saraswat V., Seshia S. Sketching Stencils // Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007. PLDI '07, ACM, New York, NY, USA. P. 167–178, DOI: 10.1145/1273442.1250754.

11. Arvind K. Sujeeth A. K., Kevin J. Brown K. J., Lee H., Rompf T., Chafi H., Odersky M., Olukotun K. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific. 2014.

12. Hoare C. A. R. An Axiomatic Basis for Computer Programming // Communications of the ACM 12(10), 1969. P. 576–580, DOI: 10.1145/363235.363259.

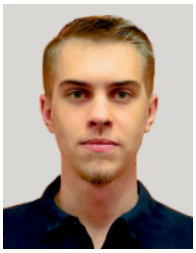
13. Cheung A., Solar-Lezama A., Madden S. Optimizing Database-backed Applications with Query Synthesis // Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2013. PLDI '13, ACM, New York, NY, USA. P. 3–14, DOI: 10.1145/2491956.2462180.

14. Kamil S., Cheung A., Itzhaky S., Solar-Lezama A. Verified Lifting of Stencil Computations // SIGPLAN Not. 2016. 51(6), P. 711–726, DOI: 10.1145/2980983.2908117.

15. Radoi C., Fink S. J., Rabbah R., Sridharan M. Translating Imperative Code to MapReduce // Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '14, 2014. ACM, New York, NY, USA. P. 909–927, DOI: 10.1145/2660193.2660228.

16. Ernst M. D., Perkins J. H., Guo P. J., McCamant S., Pacheco C., Tschantz M. S., Xiao C. The Daikon System for Dynamic Detection of Likely Invariants. Sci. Comput. Program. 2007.

17. Srivastava S., Gulwani S. Program Verification Using Templates over Predicate Abstraction // Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, '09, 2009. ACM, New York, NY, USA. P. 223–234, DOI: 10.1145/1542476.1542501.



Симонов Виктор Сергеевич, e-mail: simonws@ya.ru. Аспирант НГТУ —- Новосибирского государственного технического университета, ассистент кафедры вычислительной техники НГТУ. Научные интересы: формальные языки, математическое моделирование, вычислительные системы и сети, информационные технологии.

Simonov Victor Sergeevich, e-mail: simonws@ya.ru. Postgraduate student of Novosibirsk State Technical University (NSTU), assistant of the Department of Computer Engineering of NSTU. Research interests: formal languages, mathematical modeling, computer systems and networks, information technology.

Хайретдинов Марат Саматович, e-mail: `marat@opg.sscc.ru`.

Выпускник КАИ — Казанского авиационного института им. А. Н. Туполева (ныне КНИТУ — Казанский национальный исследовательский технический университет), доктор технических наук, профессор кафедры вычислительной техники НГТУ, главный научный сотрудник ИВМ и МГ СО РАН. Научные интересы: волновые процессы, проблемы взаимодействия геофизических полей, нелинейные процессы, статистическая обработка сигналов, математическое моделирование, методы оптимизации сложных систем, информационные технологии, мониторинговые сети. Автор и соавтор более 250 научных работ, из них 7 монографий.

Khairetdinov Marat. Graduate of the Kazan Aviation Institute named after A. N. Tupolev (now Kazan National Technical University), Doctor of Technical Sciences, Professor of the Department of Computing Engineering, NSTU, Chief Researcher, ICM & MG SB RAS. Research interests: seismic-acoustic-optical wave processes, problems of the interaction of geophysical fields, nonlinear processes, statistical signal processing, computing. Author and co-author of more than 250 scientific papers, including 7 monographs.

Дата поступления — 04.04.2023