2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

doi 10.26089/NumMet.v27r331

УДК 04.272,
004.4'242

Определение в системе SAPFOR устранимых зависимостей по массивам в последовательных Fortran-программах для их эффективного распараллеливания на вычислительные кластеры

А. С. Колганов
Московский государственный университет имени М. В. Ломоносова,
факультет вычислительной математики и кибернетики,
Москва, Российская Федерация
Институт прикладной математики имени М. В. Келдыша (ИПМ РАН),
Москва, Российская Федерация
ORCID: 0000-0002-1384-7484, e-mail: alexander.k.s@mail.ru

О. Ю. Никитин
Московский государственный университет имени М. В. Ломоносова,
факультет вычислительной математики и кибернетики,
Москва, Российская Федерация
ORCID: 0009-0002-4776-6305, e-mail: olegnik725@gmail.com

Аннотация: Процесс автоматизации распараллеливания последовательных программ сталкивается с проблемой устранения зависимостей по данным, которые препятствуют параллельному выполнению циклов. Одним из эффективных методов решения этой проблемы является приватизация переменных, позволяющая создавать локальные копии данных для каждой итерации цикла. В данной статье представлен разработанный алгоритм статического анализа для определения приватизируемых массивов в программах на языке Fortran и его реализация в системе автоматизированного распараллеливания SAPFOR (System FOR Automated Parallelization). Предложенный алгоритм основан на анализе графа потока управления с удалением обратных ребер, решении уравнений потока данных и применении операции свертки вложенных циклов. Алгоритм был протестирован на четырех прикладных программах, входящих в пакет NAS Parallel Benchmarks. Данное исследование является шагом на пути к созданию полностью автоматической системы распараллеливания, способной минимизировать участие разработчика в процессе подготовки параллельной версии программы.

Ключевые слова: SAPFOR (System FOR Automated Parallelization), автоматизация распараллеливания для кластерных систем, устранимые зависимости, анализ зависимостей массивов, параллельные вычисления, DVM (Distributed Virtual Memory), кластеры с графическими процессорами.

Для цитирования: Колганов А.С., Никитин О.Ю. Определение в системе SAPFOR устранимых зависимостей по массивам в последовательных Fortran-программах для их эффективного распараллеливания на вычислительные кластеры // Вычислительные методы и программирование. 2026. **27**, № 3. 489–513. doi 10.26089/NumMet.v27r331.

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

Detection within the SAPFOR system of eliminable array dependencies in sequential Fortran programs for efficient parallelization on computing clusters

Alexander S. Kolganov
Lomonosov Moscow State University,
Faculty of Computational Mathematics and Cybernetics,
Moscow, Russia
Keldysh Institute of Applied Mathematics of RAS,
Moscow, Russia
ORCID: 0000-0002-1384-7484, e-mail: alexander.k.s@mail.ru

Oleg Yu. Nikitin
Lomonosov Moscow State University,
Faculty of Computational Mathematics and Cybernetics,
Moscow, Russia
ORCID: 0009-0002-4776-6305, e-mail: olegnik725@gmail.com

Abstract: The process of automating the parallelization of sequential programs faces the problem of eliminating data dependencies that prevent the parallel execution of loops. One of the effective methods to solve this problem is a variable privatization, which allows creating local copies of data for each loop iteration. This paper presents the implementation in the system SAPFOR (System FOR Automated Parallelization) of a static analysis algorithm for identifying privatizable arrays in Fortran programs. The proposed algorithm is based on the control flow graph analysis with back edge removal, solving data flow equations, and applying nested loop convolution. The algorithm was tested on four application programs from the NAS Parallel Benchmarks suite. This research is a step towards creating a fully automatic parallelization system capable of minimizing developer involvement in the process of preparing a parallel version of a program.

Keywords: SAPFOR (System FOR Automated Parallelization), automation of parallelization for clusters, eliminable dependencies, array dependence analysis, parallel computing, DVM (Distributed Virtual Memory), GPU clusters.

For citation: A. S. Kolganov, O. Yu. Nikitin, "Detection within the SAPFOR system of eliminable array dependencies in sequential Fortran programs for efficient parallelization on computing clusters," Numerical Methods and Programming. **27** (3), 489–513 (2026). doi 10.26089/NumMet.v27r331.

1. Введение. Развитие современных вычислительных систем сопровождается ростом числа используемых в них процессорных ядер и ядер графических ускорителей. Иными словами, стремительный рост подобных систем ведет к развитию параллелизма. Это требует разработки эффективных методов преобразования последовательных программ в параллельные. Для того чтобы преобразованные программы проводили вычисления параллельно на нескольких потоках и эффективно использовали ресурсы вычислительной системы, а результаты выполнения параллельной и последовательной версий совпадали, необходимо обеспечить корректный процесс чтения и записи данных [1]. При неправильной синхронизации доступа к разделяемым ресурсам могут возникнуть конфликты чтения-записи и записи-записи, когда несколько потоков одновременно обращаются к одной и той же области памяти. Одним из проявлений таких конфликтов является состояние гонки (race condition) — ситуация, при которой результат выполнения программы зависит от случайного порядка чередования операций потоков во времени. Например, состояние гонки возникает, когда один поток считывает значение переменной в тот момент, когда другой поток параллельно ее изменяет. В этом случае итоговое значение переменной и, следовательно, результа-

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

ты последующих вычислений становятся недетерминированными: в зависимости от порядка выполнения потоков программа может выдать как корректный, так и некорректный результат.

Одним из подходов к решению таких задач является приватизация переменных — преобразование, при котором переменные или массивы заменяются на локальные копии для каждого параллельного потока. Это позволяет устранить конфликтующие обращения к данным и расширить возможности параллельного выполнения программы. Приватизация часто применяется в циклах, где одни и те же переменные могут многократно переиспользоваться, создавая зависимости по данным и препятствуя одновременному исполнению нескольких витков цикла.

Приватизация активно используется в различных технологиях параллельного программирования, таких как OpenMP [2], OpenACC [3], XcalableMP [4], где разработчик может явно указывать список приватных переменных. Аналогичные концепции применяются и в других системах, включая компиляторные оптимизации высокого уровня [5, 6]. Даже современные компиляторы с поддержкой GPU (CUDA [7], GCC) не всегда способны автоматически определить приватизируемые переменные, оставляя эту задачу программисту.

В большинстве случаев разработчик вручную выполняет поиск и устранение зависимостей по данным, препятствующих параллельному выполнению кода. Системы автоматизированного распараллеливания последовательных программ могут облегчить этот процесс. Они используют методы статического и динамического анализа исходного кода, которые позволяют создать параллельную версию программы и обнаружить участки, препятствующие распараллеливанию. В качестве примера такого инструмента можно рассмотреть систему автоматизированного распараллеливания Fortran-программ SAPFOR (System FOR Automated Parallelization), разрабатываемую в Институте прикладной математики имени М. В. Келдыша РАН при активном участии студентов и аспирантов факультета ВМК МГУ имени М. В. Ломоносова.

Система SAPFOR использует методы статического анализа исходного кода, строит абстрактное синтаксическое дерево (AST) и промежуточное представление (IR) в виде трехадресных команд, после чего выполняет набор преобразований, направленных на подготовку программы к распараллеливанию. Результатом работы системы является многопоточная версия программы в модели DVMH [8] или диагностическая информация с указанием причин, препятствующих распараллеливанию.

Однако в системе SAPFOR до настоящего времени отсутствовали алгоритмы автоматического определения приватизируемых массивов. Пользователю приходилось вручную анализировать циклы программы и указывать приватные переменные с помощью соответствующих директив анализа системе SAPFOR. Это ограничивало степень автоматизации и требовало от разработчика глубокого понимания структуры программы и возникающих в ней зависимостей по данным. Более того, последующие преобразования, такие как удаление приватных переменных [9], также требуют предварительной идентификации приватных массивов. Таким образом, автоматическое определение приватизируемых массивов является необходимым предшествующим этапом для построения полностью автоматической цепочки преобразований, ведущей к эффективной параллельной программе. В данной работе рассматривается алгоритм статического анализа циклов, позволяющий автоматически выявлять приватизируемые массивы в последовательных Fortran-программах. Алгоритм основан на анализе графа потока управления с удалением обратных ребер, решении уравнений потока данных и применении операции свертки вложенных циклов. Для компактного представления множеств обращений к элементам массивов используется модель диапазонов изменения индексных выражений, что позволяет эффективно обрабатывать многомерные массивы большого размера без поэлементного перечисления.

Целью настоящей работы является разработка и реализация алгоритма автоматического определения приватизируемых массивов в последовательных Fortran-программах, позволяющего полностью исключить ручной ввод информации о приватных переменных в системе SAPFOR и обеспечить основу для построения полностью автоматической цепочки преобразований, ведущей к эффективной параллельной программе.

Статья имеет следующую структуру. В разделе 2 рассматриваются существующие подходы к автоматическому определению приватизируемых переменных в различных системах распараллеливания и компиляторах, а также обосновывается необходимость разработки собственного алгоритма для системы SAPFOR. Раздел 3 содержит краткое описание системы SAPFOR и ее архитектуры. В разделе 4 вводятся основные понятия и определения, необходимые для описания алгоритма: приватизация переменных, анализ потока данных, анализ на основе областей и операция свертки циклов. Раздел 5 посвящен представлению обращений к массивам в виде диапазонов и операциям над ними. В разделе 6 подробно

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

излагается разработанный алгоритм поиска приватизируемых массивов, включая сбор локальной информации, решение уравнений потока данных и формирование агрегированных множеств. Раздел 7 описывает вспомогательные методы, расширяющие применимость алгоритма: работу с динамическими параметрами задачи, распространение констант через элементы массивов и подстановку процедур на уровне промежуточного представления. В разделе 8 представлены результаты тестирования алгоритма на программах из пакета NAS Parallel Benchmarks и оценка эффективности автоматического определения приватных массивов. В заключении подводятся итоги работы и намечаются направления дальнейших исследований.

2. Обзор существующих систем. В области статического анализа и автоматизированного преобразования кода активно ведутся исследования. Решение задачи автоматизированного поиска и устранения зависимостей между параллельными областями позволяет увеличить степень параллелизма и ускорить работу разрабатываемых программ. Различные исследовательские и промышленные системы предлагают свои подходы к решению данной задачи, отличающиеся степенью точности анализа, уровнем межпроцедурной обработки и используемыми моделями представления данных.

Рассмотрим метод определения приватизируемых переменных на основе самозависимых определений. В статье [10] вводится понятие самозависимых определений (self-dependent definitions, SDD). Это определение массива, при котором производится запись в один и тот же элемент массива на разных витках цикла. На листинге 1 показан пример самозависимого определения.

Листинг 1. Пример самозависимого определения
Listing 1. An example of a self-dependent definition

```
1 DO i = 2, N
2    A(2) = ...
3 END DO
```

Далее вводятся несамозависимые определения (non-self-dependent definitions, NSDD), при которых отсутствует запись на разных витках цикла в один и тот же элемент. Автор работы [10] формулирует два условия приватизации массива:

- все самозависимые определения данного массива должны быть приватизируемыми, то есть значения, создаваемые этими определениями, не должны использоваться между разными витками цикла;
- самозависимые определения не должны пересекаться с несамозависимыми по множеству элементов массива, то есть элементы, записываемые самозависимыми определениями, не должны конфликтовать с элементами, записываемыми несамозависимыми определениями.

Если оба условия выполняются, массив рассматривается как кандидат на приватизацию. Проверка проводится с помощью анализа графа потока управления. Сначала определяются элементы массива, которые могут быть прочитаны раньше, чем будут определены в рамках того же витка цикла. Затем проверяется, что ни один элемент, записанный самозависимым определением на каком-либо витке цикла, не используется для чтения на более поздних витках цикла, то есть не является “upward exposed” по отношению к ним.

Можно выделить следующие ограничения данного подхода. Анализ условных операторов упрощен: фактическое содержимое условных конструкций обычно не анализируется. Сложные индексные выражения требуют символьных операций над множествами. Межпроцедурный анализ в рассматриваемой работе выносится за рамки, предполагается подстановка процедур. В случае когда невозможно определить, на какой итерации элемент массива определяется в последний раз, массив не подлежит приватизации.

В статье [11] предлагается динамический метод DOALL Test для определения приватизируемых переменных. Статический анализ может быть недостаточно точным из-за отсутствия информации о значениях переменных на этапе компиляции. Вместо этого компилятор внедряет в программу специальный проверочный код, который срабатывает во время ее выполнения.

Этот проверочный код осуществляет мониторинг фактических обращений к данным: отслеживает, какие элементы массивов и скалярные переменные читаются и записываются на каждом витке цикла. На основе собранной информации динамически проверяется, возникают ли зависимости по данным между разными витками. Если зависимости отсутствуют или могут быть устранены с помощью приватизации, цикл выполняется параллельно.

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

Метод поддерживает приватизацию как скалярных переменных, так и массивов. Если каждый виток цикла может использовать собственную копию переменной или массива, не влияя на результаты других витков, тест динамически выполняет приватизацию. Это означает, что для каждого витка создается отдельная копия данных, что позволяет избежать конфликтов при параллельном выполнении.

Таким образом, DOALL Test переносит часть работы с этапа компиляции на этап выполнения программы, что позволяет учесть информацию, недоступную при статическом анализе (например, реальные значения границ циклов или динамические адреса массивов). Однако такой подход вносит дополнительные накладные расходы на выполнение проверочного кода, что является его основным недостатком.

Система Polaris [12, 13] — одна из первых систем автоматизированного распараллеливания программ Fortran 77. В отличие от ранних прототипов, которые были ориентированы преимущественно на демонстрацию отдельных оптимизаций, Polaris проектировался как целостная компиляторная среда, способная работать с реальными прикладными научно-техническими кодами. Это сделало его важной опорной точкой в исследованиях методов автоматизированной приватизации.

В системе Polaris процесс приватизации начинается со сбора информации об обращениях к памяти в теле цикла. Анализируется набор читаемых и записываемых элементов массивов, а также последовательность этих операций в пределах одного витка. При этом рассматриваются все вложенные циклы. Затем выполняется проверка межитерационных конфликтов для тех пар обращений, которые потенциально могут образовывать зависимости по данным.

Вместо отдельного анализа каждого индексного выражения при обращении к массивам система Polaris строит аппроксимацию множества элементов массива, к которым происходит обращение в пределах одного витка цикла. Такой подход позволяет эффективнее проверять пересечение обращений и оказывается особенно полезным для вложенных циклов, где индексные выражения могут зависеть сразу от нескольких итерационных переменных циклов.

Идеи, заложенные в системе Polaris, послужили отправной точкой для разработки алгоритма, предлагаемого в данной статье. Однако предложенное решение включает ряд существенных отличий и расширений, необходимых для работы в составе системы SAPFOR с реальными программами на языке Fortran, которые будут рассмотрены далее.

Более поздние исследования в области автоматического определения приватных массивов развивали методы символьного анализа для повышения точности и расширения класса анализируемых программ. В работе [14] предлагается метод символьного анализа потоков данных для массивов, основанный на использовании охраняемых областей массивов (guarded array regions). Ключевая особенность подхода состоит в том, что информация из условных операторов (IF-условий) представляется в виде специальных охранных выражений (guards) и учитывается при построении и распространении областей массива по графу потока управления. Это позволяет работать со сложными случаями, недоступными для более ранних методов, включая программы с произвольными графами потока управления и ациклическими графами вызовов.

Охранные области массивов сохраняют простоту теоретико-множественных операций для регулярных областей в типичных случаях и расширяют их возможности в сложных за счет использования охраняющих условий для обработки символьных выражений и нерегулярных форм массивов. Кроме того, скалярные значения, входящие в индексные выражения и границы циклов, подставляются непосредственно в процессе распространения информации, что позволяет точно разрешить неоднозначности при выполнении операций над множествами. Проведенные эксперименты показали, что предложенный анализ позволяет успешно выявлять приватные массивы в циклах, дающих существенный вклад в общее время выполнения (от 3% до 90% в зависимости от программы).

Дальнейшее развитие методов анализа массивов связано с распространением на них SSA-формы (Static Single Assignment). В [15] предложено расширение SSA-представления, получившее название Region Array SSA. Авторы этой работы отмечают, что, несмотря на широкое применение SSA для скалярных переменных в современных компиляторах, аналогичный прогресс в анализе потоков данных для массивов отсутствует. Предложенное представление использует символьный дескриптор доступа к памяти (symbolic memory access descriptor), позволяющий агрегировать обращения к элементам массива в крупных, в том числе межпроцедурных, контекстах программы. На основе этого представления реализован базовый алгоритм достижимых определений (reaching definitions) для областей массивов. Применение разработанного анализа к задачам распространения констант для массивов и приватизации массивов продемонстрировало измеримое ускорение выполнения тестовых программ. Данный подход позволяет более точно представ-

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

лять отношения между определениями и использованиями областей массивов, что особенно важно для корректного определения приватных массивов в сложных вложенных циклах.

Отдельное направление в области автоматического распараллеливания представляют компиляторы, основанные на полиэдральной модели [16]. К наиболее известным инструментам этого класса относятся Pluto [17], PPCG [18] и Polly [19]. Эти системы выполняют глобальные преобразования циклов (например, изменение порядка вложенности, разворачивание, слияние) на основе полиэдрального представления итерационного пространства. Однако этот подход к приватизации существенно отличается от статического анализа, рассматриваемого в настоящей работе.

В полиэдральных компиляторах приватные переменные, как правило, не выявляются с помощью отдельного анализа потока данных. Вместо этого системы опираются на полиэдральную модель зависимостей и преобразуют код таким образом, чтобы зависимости, требующие приватизации, исчезали за счет перестройки вычислений. Например, если цикл содержит редукционную операцию или временный массив, компилятор может автоматически преобразовать код к виду, где такая переменная становится локальной для каждого витка цикла [20]. Однако этот подход накладывает жесткие ограничения на структуру программы: индексные выражения должны быть аффинными, границы циклов — инвариантными относительно выполнения, а управляющий поток — предсказуемым [17] (отсутствие операторов безусловного перехода и сложных условий).

Кроме того, полиэдральные преобразования вносят значительные изменения в код программы, зачастую делая его трудно читаемым для человека. Это противоречит концепции системы SAPFOR, которая сохраняет высокоуровневую структуру программы и предполагает возможность интерактивного участия разработчика в процессе распараллеливания. В то же время полиэдральные компиляторы практически не решают задачу автоматического определения приватизируемых массивов как таковую — для них это лишь побочный эффект перестройки циклов, а не полноценный анализ.

Таким образом, полиэдральный подход не заменяет статический анализ приватизации переменных, необходимый в системах распараллеливания, где преобразование кода должно быть прозрачным для пользователя и применимо к более широкому классу программ. В настоящей работе, напротив, приватизация рассматривается как отдельная задача, решаемая методами анализа потока данных, что позволяет выявлять приватные массивы без глобальной перестройки кода.

Наконец, рассмотрим как задача автоматической приватизации решается в современных компиляторах, поддерживающих модель OpenACC. OpenACC — это высокоуровневая модель параллельного программирования для гетерогенных систем, позволяющая разработчику указывать участки кода, предназначенные для выполнения, в том числе на графических ускорителях с помощью директив. В стандарте OpenACC предусмотрены две основные директивы для распараллеливания циклов: `PARALLEL` и `KERNELS`. Директива `PARALLEL` требует от программиста явных указаний: необходимо вручную определить распределение данных и приватные переменные, что возлагает всю ответственность за корректность полученного параллельного кода на разработчика. В противоположность ей, директива `KERNELS` декларирует подход “черного ящика”: компилятор самостоятельно анализирует размеченные данной директивой участки кода, выявляет параллелизм и выполняет необходимые оптимизации, включая приватизацию данных, без вмешательства пользователя.

Однако на практике способность компиляторов (например, NVFORTRAN из HPC SDK) к автоматическому решению таких сложных задач, как определение приватизируемых массивов на основе глобального анализа потока данных, крайне ограничена. Рассмотрим простой пример на языке Fortran, демонстрирующий эту проблему (листинг 2).

В рассматриваемом примере массив `LOCAL(2)` на каждом витке цикла получает новые значения и используется только в пределах текущего витка. Очевидно, что он может быть приватизирован. Однако при компиляции этого кода с помощью NVFORTRAN из состава NVIDIA HPC SDK версии 12.9 с флагом `-Minfo=accel` получается вывод, приведенный в листинге 3.

Компилятор явно указывает причину отказа от распараллеливания: “Complex loop-carried dependence of local prevents parallelization. Parallelization would require privatization of array local(:)”. Несмотря на то, что по семантике программы массив `LOCAL` является приватным для каждого витка цикла, компилятор не выполняет достаточного статического анализа для вывода этого факта, даже при том, что размеры массива известны, он является локальным для рассматриваемой подпрограммы и требует ручного вмешательства от пользователя. В результате цикл выполняется последовательно (“14, !$acc loop seq”), что полностью нивелирует все преимущества использования ускорителя для данного цикла.

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

Листинг 2. Пример цикла с приватным массивом `LOCAL(2)`
Listing 2. Example of a loop with a private array `LOCAL(2)`

```
PROGRAM MAIN
    IMPLICIT NONE
    INTEGER, PARAMETER :: N = 1000
    REAL, DIMENSION(N) :: A, B
    REAL :: LOCAL(2)
    INTEGER :: I

    DO I = 1, N
        A(I) = REAL(I)
    END DO

!$ACC KERNELS
    DO I = 2, N-1
        LOCAL(1) = A(I-1) + A(I)
        LOCAL(2) = A(I) + A(I+1)
        B(I) = (LOCAL(1) + LOCAL(2)) * 0.5
    END DO
!$ACC END KERNELS

    PRINT *, 'B(500) = ', B(500)
END PROGRAM MAIN
```

Листинг 3. Вывод компилятора NVFORTRAN для цикла с директивой `KERNELS`
Listing 3. NVFORTRAN compiler output for a loop with the `KERNELS` directive

```
main:
   13, Generating implicit copyin(a(:)) [if not already present]
       Generating implicit copyout(b(2:999),local(:)) [if not already present]
   14, Complex loop carried dependence of local prevents parallelization
       Parallelization would require privatization of array local(:)
       Accelerator serial kernel generated
       Generating NVIDIA GPU code
       14, !$acc loop seq
   14, Complex loop carried dependence of local prevents parallelization
       Parallelization would require privatization of array local(:)
```

Для успешного распараллеливания этого цикла программист вынужден либо переключиться на директиву `PARALLEL` и вручную добавить клаузулу `PRIVATE(LOCAL)`, либо использовать директиву `ACC LOOP PRIVATE(LOCAL)` внутри региона `KERNELS`. В обоих случаях требуется ручное вмешательство и понимание модели устройства памяти графического ускорителя.

Исследование практики использования OpenACC [21] показывает, что разработчики регулярно сталкиваются с ситуациями, когда компилятор не способен доказать независимость витков цикла. В таких случаях единственным выходом является ручное указание приватных переменных, что требует глубокого понимания кода программы и моделей памяти ускорителей.

Таким образом, хотя идея директивы `KERNELS` предполагает полную автоматизацию, реальные компиляторы по-прежнему полагаются на явное управление приватизацией со стороны программиста. Система SAPFOR, напротив, осуществляет полноценный статический анализ потока данных для автоматического выявления приватизируемых массивов, что является шагом к реализации идеи, заложенной в `KERNELS`, но на более высоком уровне и без привязки к конкретной модели параллелизма.

3. Система SAPFOR. Важным дополнением к DVM-системе является система автоматизированного распараллеливания SAPFOR [22] для программ на языках Fortran и C. SAPFOR анализирует про-

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

грамму с помощью методов статического анализа, то есть работая только с исходным кодом программы без его запуска и отслеживания выполнения. На основе полученных данных система осуществляет преобразование кода и генерацию параллельной версии, расставляя в коде программы DVMH-директивы.

Автоматизированное распараллеливание программ основывается на возможности выявления данных, которые могут быть распределены между вычислительными устройствами, и циклов, витки которых могут быть выполнены параллельно. Для этого требуется точный анализ исходного кода программы с выявлением в нем зависимостей по данным и управлению. Как было сказано ранее, часто автоматизированное распараллеливание программы в исходном виде через расстановку в ее коде директив параллелизма оказывается либо невозможным, либо приводит к неэффективному параллельному коду (например, с большим количеством обменов данными между процессами). В ряде случаев решить указанные проблемы помогают преобразования исходной программы в рамках последовательного кода, которые приводят ее к виду, более подходящему для автоматизированного распараллеливания. Данный вид программы будем называть потенциально параллельным.

Процесс распараллеливания программы с помощью системы SAPFOR является итеративным и состоит из этапов анализа и преобразования кода. Выполнив тот или иной анализ, пользователь получает набор диагностик и сообщений системы, на основе которых принимает решение о дальнейших действиях по созданию параллельной версии программы. Этап преобразований заключается в трансформации исходного кода последовательной программы или построении ее параллельной версии. Если некоторый анализ или преобразование выполнить не удается, система выдает сообщения о проблемах с привязкой к строкам исходной программы.

В зависимости от полученных результатов пользователь может изменить набор преобразований и порядок проведения анализа, получив в итоге другую версию параллельной программы. Следует отметить, что оптимальной последовательности преобразований участков кода зачастую не существует даже в рамках одной программы (данный аспект рассмотрен в статье [23]). К одному участку кода могут быть применимы несколько преобразований, дающих разный эффект.

Этапы анализа и преобразования кода в системе SAPFOR представляют собой наборы проходов. Проход — это функциональный модуль, решающий отдельную простую задачу (например, построение графа вызовов процедур или подстановку функций в места их вызова). В общем случае работа прохода состоит из двух фаз, на первой из которых каждый файл исходной программы единообразно обрабатывается, а на второй происходит объединение собранных данных. Наличие всех фаз для прохода не является обязательным.

Проходы могут иметь зависимости между собой. Они возникают из-за того, что для выполнения одного прохода могут потребоваться результаты анализа или предварительные преобразования программы, полученные в ходе другого прохода. Для указания зависимостей между ними используется менеджер проходов, представляющий собой структуру, в которой хранятся прямые упорядоченные зависимости между теми или иными проходами. Косвенные зависимости выводятся из прямых автоматически. При запуске пользователем некоторого прохода система SAPFOR выполняет все проходы, от которых он прямо или косвенно зависит, после чего осуществляет вызванный пользователем проход [24].

4. Основные понятия и определения.

4.1. Определение приватных переменных. Приватизация переменной — это преобразование программы, при котором каждому параллельно выполняемому потоку или процессу предоставляется собственная независимая копия переменной. В терминах OpenMP приватизированная переменная представляет собой объект, для которого каждый параллельный контекст выполнения использует отдельную область памяти.

Пусть в цикле L происходит обращение к массиву A. Массив A считается приватизируемым для цикла L, если выполнено одно из следующих условий:

- каждому обращению к элементу массива A для чтения предшествует присваивание значения этому элементу в том же витке цикла L;
- разные витки цикла L могут обращаться к одним и тем же элементам массива A, но при этом не возникает зависимостей типа “read after write” между витками цикла (то есть чтение всегда происходит после записи в том же витке).

4.2. Анализ потока данных. Выполнение программы можно рассматривать как последовательность преобразований состояния переменных. Анализ потока данных направлен на исследование передачи

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

значений переменных между инструкциями вдоль путей выполнения программы. Потоком управления будем называть порядок выполнения команд программы.

Под базовым блоком будем понимать совокупность максимальной последовательности инструкций, которая выполняется строго последовательно и обладает следующими особенностями [25]:

- поток управления может войти в блок только через его первую инструкцию (отсутствуют переходы внутрь блока);
- поток управления покидает блок после выполнения его последней инструкции (за исключением, возможно, самой последней инструкции, которая может быть переходом).

Для базового блока B введем следующие множества:

- $def[B]$ включает переменные, которые получают последнее определение внутри B (то есть их значение обязательно изменяется при выполнении блока);
- $use[B]$ содержит переменные, значения которых используются в B до того, как они будут определены в этом же блоке;
- $kill[B]$ включает все переменные, имеющие хотя бы одно определение в B (независимо от того, является ли это определение последним).

Говорят, что базовый блок B_1 доминирует над базовым блоком B_2, если любой путь из входного узла в графе потока управления, ведущий в B_2, проходит через B_1. Графом потока управления называется ориентированный граф, вершины которого соответствуют базовым блокам, а ребра — возможным переходам управления между ними. Если после последней инструкции блока B_1 управление может быть передано на первую инструкцию блока B_2, то в графе существует ребро от B_1 к B_2.

Для анализа потока данных с каждой точкой программы связывается значение потока данных — абстракция множества возможных состояний программы в этой точке. В данной работе анализируется, какие переменные и области массивов определены к моменту выполнения каждой инструкции. Значение потока данных перед выполнением инструкции s обозначается $in[s]$, а после выполнения — $out[s]$. Для базового блока B через $in[B]$ и $out[B]$ обозначаются значения перед первой инструкцией блока и после последней соответственно.

Анализ графа потока управления можно осуществлять как в прямом направлении (от точки получения входных данных программы до места завершения работы и формирования результата), так и в обратном направлении. Для базового блока B решение уравнений потока данных в прямом направлении по графу потока управления имеет следующий вид:

$$\begin{aligned} in[Enter] &= \varnothing, \\ in[B] &= \bigcap_{P \in pred(B)} out[P], \\ out[B] &= \big(in[B] \setminus kill[B]\big) \cup def[B]. \end{aligned} \tag{1}$$

где множество $pred(B)$ — множество предшественников B в графе потока управления.

Оператор сбора (в данном случае — пересечение) определяет, как объединяется информация, приходящая в блок по разным путям графа потока управления. В данном случае необходимо использовать пересечение, поскольку что свойство “переменная определена” должно выполняться на всех путях, ведущих в данную точку.

Передаточная функция f_B описывает, как базовый блок B преобразует входное значение потока данных в выходное. Для базового блока, состоящего из инструкций $s_1, s_2, \ldots, s_n$, передаточная функция является суперпозицией передаточных функций отдельных инструкций: $f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}$.

4.3. Анализ на основе областей. Традиционный анализ потока данных работает на уровне базовых блоков и отдельных инструкций. Однако для анализа вложенных циклов и крупных фрагментов программы удобнее использовать анализ на основе областей. Этот подход обобщает выполнение все более крупных областей программы, начиная с базовых блоков и заканчивая целыми процедурами.

Областью назовем связный подграф графа потока управления, имеющий единственную входную вершину (заголовок), которая доминирует над всеми остальными вершинами области, и, возможно, несколько выходных вершин. Важным свойством области является то, что если существует путь от некоторой вершины m до вершины n внутри области, не проходящий через заголовок, то вершина m также принад-

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

лежит этой области. В контексте данной работы область может соответствовать либо базовому блоку как наименьшей области, содержащей последовательность инструкций, либо циклу — области, включающей все базовые блоки, входящие в тело цикла, а также вложенные в него циклы.

Топологическая сортировка [26] — это упорядочивание вершин направленного ациклического графа, при котором для каждого ребра $u \to v$ вершина u предшествует вершине v. Для анализа потока данных важно, что если из графа потока управления удалить обратные ребра, ведущие к заголовку цикла, то полученный граф становится ациклическим. Это позволяет применить топологическую сортировку и вычислить множества in и out для всех вершин за один проход, без итеративной сходимости как в общем случае.

Анализ на основе областей позволяет строить передаточные функции для каждой области, которые обобщают поведение всех инструкций внутри нее. Вложенный цикл сначала анализируется как самостоятельная область, а затем свертывается — заменяется одной вершиной с агрегированными множествами. Это позволяет при анализе внешнего цикла рассматривать вложенный цикл как единый базовый блок, что существенно упрощает вычисления и делает анализ масштабируемым для программ с глубокой вложенностью циклов и большим количеством инструкций.

4.4. Операция свертки цикла. При анализе вложенных циклов возникает необходимость обобщить информацию о внутреннем цикле для использования во внешнем. Для этого применяется операция свертки. Пусть M — цикл, вложенный в цикл L. После завершения анализа цикла M соответствующий ему подграф графа потока управления заменяется одной вершиной $new\,B$, которая при анализе внешнего цикла L рассматривается как обычный базовый блок.

Пусть G — это граф потока управления, а L — цикл `DO` в графе G. Тогда $COLLAP(G, L)$ обозначает граф потока управления, в котором подграф, соответствующий циклу L, свернут в одну вершину. Операция свертки обозначается как $G' = COLLAP(G, M)$, где G' — граф, в котором подграф цикла M заменен вершиной $new\,B$. В рассматриваемых далее формулах предполагается, что для графа потока управления выполнена топологическая сортировка и обход происходит на уровне инструкций, составляющих базовые блоки.

Для свернутой вершины $new\,B$ агрегированные множества, обобщающие поведение цикла M, вычисляются следующим образом. Множество $use_b[M]$ содержит переменные, значения которых используются внутри цикла M, но определены вне его или на предыдущих витках (требуются на входе в цикл). Его можно задать в соответствии с формулой:

$$use_b[M] = \bigcup_{t \in M} (use[t] \setminus in[t]).$$

Множество $def\,[new\,B]$ содержит переменные, которые определены на всех возможных путях выхода из цикла M (гарантированно определены после выполнения цикла). Оно задается по формуле:

$$def\,[new\,B] = \bigcap_{t \in exits(M)} out[t].$$

Множество $use[new\,B]$ содержит переменные, используемые в цикле M, но не определенные гарантированно внутри него. Задается по формуле:

$$use[new\,B] = use_b[M] \setminus def\,[new\,B]. \tag{2}$$

Наконец, множество $pri[M]$ содержит переменные, используемые внутри цикла, но не требующие значения на входе в него (кандидаты на приватизацию). Оно определяется по формуле:

$$pri[M] = \bigcup_{t \in M} use[t] \setminus use_b[M].$$

Рассмотрим последовательность шагов для выполнения свертки для цикла L. Для каждого вложенного цикла M рекурсивно выполняется его анализ и свертка. В результате M заменяется вершиной $new\,B$ с найденными множествами $def\,[new\,B]$, $use[new\,B]$ и $pri[M]$. В графе потока управления цикла L удаляются обратные ребра, ведущие к заголовку L. Полученный граф становится ациклическим. Выполняется топологическая сортировка вершин графа (базовых блоков и свернутых вершин, представляющих вложенные

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

циклы). Решаются уравнения потока данных в топологическом порядке, при этом свернутые вершины обрабатываются так же, как обычные базовые блоки с использованием их агрегированных множеств. После вычисления *in* и *out* для всех вершин цикла L определяеются его собственные агрегированные множества $def_b[L]$, $use_b[L]$ и $pri[L]$ по выше рассмотренным формулам.

Благодаря свертке анализ вложенных циклов выполняется за линейное (относительно количества базовых блоков) время, а не за экспоненциальное. Кроме того, свертка позволяет абстрагироваться от внутренней структуры вложенного цикла при анализе внешнего, избежать повторного анализа одного и того же цикла при его использовании в разных контекстах, корректно обрабатывать приватные переменные, которые могут быть созданы во внутреннем цикле и использованы во внешнем (или наоборот). В данной работе свертка применяется для анализа массивов: множества *def*, *use* и *pri* оперируют не скалярными переменными, а областями массивов, представленными в виде диапазонов.

5. Представление обращений к массивам и операций над ними. Одному обращению к массиву в тексте программы может соответствовать множество элементов. Каждый элемент можно рассматривать как отдельную переменную, и тогда алгоритм приватизации не требует каких-либо модификаций. Однако такой подход приводит к значительным накладным расходам: объем хранимой информации становится пропорционален числу элементов массива, что неприемлемо для массивов большого размера и многомерных массивов. Поэтому рассмотрим их более компактное представление, при котором множественные обращения к элементам массива в цикле заменяются на диапазон обращений.

Обращение к каждому измерению массива осуществляется с помощью индексных выражений, значения которых могут меняться между витками цикла. При обращении к элементам массива в цикле их индексные выражения часто изменяются по линейному правилу. В таком случае множество обрабатываемых элементов удобно описывать не списком значений, а параметрами соответствующей линейной зависимости. Диапазон обращений к измерению массива описывается структурой, состоящей из тройки значений *start*, *step* и *tripCount*. Они определяют множество значений одного индексного выражения в виде арифметической последовательности. Такой способ представления позволяет компактно описывать как одиночное обращение к элементу массива, так и диапазон элементов, обрабатываемый в цикле:

$$\{start + i \cdot step \mid 0 \leqslant i < tripCount\}.$$

Если значение *tripCount* равно 1, то структура описывает либо обращение в цикле с одним витком, либо обращение по константному индексу. Обращение к области массива описывается в виде вектора таких троек, где i-му измерению массива соответствует i-й элемент этого вектора.

Таким образом, все обращения к элементам массива внутри цикла можно представить в виде набора диапазонов, задающих декартово произведение всех значений индексных выражений. Использование такого представления позволяет рассматривать массив не как множество отдельных элементов, а как область. Множественные операции в таком случае заменяются операциями между диапазонами. Ниже рассмотрим необходимые операции над диапазонами.

5.1. Пересечение диапазонов. Пусть имеются две арифметические последовательности, описывающие два разных обращения к одному и тому же массиву:

$$D_1 = \{start_1 + i \cdot step_1 \mid 0 \leqslant i < tripCount_1\},$$
$$D_2 = \{start_2 + j \cdot step_2 \mid 0 \leqslant j < tripCount_2\}.$$

Операция пересечения должна построить новый диапазон или новую арифметическую последовательность, которая описывает все значения из пересечения множеств D_1 и D_2. Для этого необходимо найти все решения задачи вида:

$$start_1 + X \cdot step_1 = start_2 + Y \cdot step_2.$$

Пересечение находится в два этапа. На первом этапе выполняется поиск частного решения, то есть набора (i, j), при котором значения двух арифметических последовательностей совпадают. Если такой набор не найден, то пересечение считается пустым. Если частное решение найдено, то на втором этапе строится общее решение как решение линейного уравнения с двумя неизвестными. Ниже приведен алгоритм 1 поиска всех решений в форме псевдокода. Результатом является новый диапазон D_3, представляющий множество значений, образованных пересечением диапазонов D_1 и D_2.

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

Алгоритм 1. Вычисление пересечения двух диапазонов массива

Algorithm 1. Calculating the intersection of two array regions

Вход: два диапазона D_1 и D_2
Выход: диапазон $D_3 = D_1 \cap D_2$ или $\varnothing$, если пересечение отсутствует
function *DimensionIntersection*
  $solution \leftarrow FindParticularSolution(D_1, D_2)$
  if $solution = \varnothing$ **then**
    return $\varnothing$
  end if
  $x_0 \leftarrow solution[0]$
  $y_0 \leftarrow solution[1]$
  $g \leftarrow \gcd(D_1.step, D_2.step)$
  $c \leftarrow D_2.step/g$
  $d \leftarrow D_1.step/g$
  $tX_{\min} \leftarrow -x_0/c$
  $tX_{\max} \leftarrow (D_1.tripCount - 1 - x_0)/c$
  $tY_{\min} \leftarrow -y_0/d$
  $tY_{\max} \leftarrow (D_2.tripCount - 1 - y_0)/d$
  $t_{\min} \leftarrow \max(tX_{\min}, tY_{\min})$
  $t_{\max} \leftarrow \min(tX_{\max}, tY_{\max})$
  if $t_{\min} > t_{\max}$ **then**
    return $\varnothing$
  end if
  $D_3.start \leftarrow D_1.start + x_0 \cdot D_1.step$
  $D_3.step \leftarrow c \cdot D_1.step$
  $D_3.tripCount \leftarrow t_{\max} + 1$
  return D_3
end function

5.2. Разность диапазонов. Для получения разности необходимо вычислить пересечение двух диапазонов. Полученный пересечением результат делится на три типа участков (рис 1):

- *Left* — подмножество значений из первого диапазона, которые меньше значений пересечения;
- *Right* — подмножество значений из первого диапазона, которые больше значений пересечения;
- *Center* — подмножество значений из первого диапазона, которые лежат между значениями пересечения. Таких диапазонов может существовать несколько, поскольку может быть несколько участков, которые содержат значения первого диапазона и ограничены значениями пересечения.

Рис. 1. Разность диапазонов

Fig. 1. Range difference

Для описания полученных участков можно сохранить исходное значение *step*, что приведет к сокращению вычислений, но разобьет участок *Center* на множество более мелких участков. Ниже приведен алгоритм 2 для вычисления разности двух диапазонов в форме псевдокода.

Алгоритм 2. Вычисление разности двух диапазонов массива

Algorithm 2. Calculating the difference of two array regions

Вход: два диапазона D_1 и D_2
Выход: дножество диапазонов R, описывающих $D_1 \setminus D_2$
function *DimensionDifference*
  $inter \leftarrow DimensionIntersection(D_1, D_2)$

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

```
    if inter = ∅ then
        return D_1
    end if
    R ← ∅
    if D_1.start < inter.start then
        добавить в R участок D_1, расположенный до начала inter
    end if
    if inter.step > D_1.step then
        for каждого промежутка между соседними элементами inter do
            добавить в R участок D_1, лежащий внутри промежутка
        end for
    end if
    end_1 ← D_1.start + D_1.step · (D_1.tripCount − 1)
    end_inter ← inter.start + inter.step · (inter.tripCount − 1)
    if end_inter < end_1 then
        добавить в R участок D_1, расположенный после конца inter
    end if
    return R
end function
```

5.3. Область массива. Область массива описывает все обращения к его элементам. Она представляется в виде вектора диапазонов, где каждый диапазон задает закон изменения индексного выражения в соответствующем измерении массива. Рассмотрим операции пересечения и разности для областей массива.

Операция пересечения (*elementsIntersection*) принимает на вход две области массива. Для каждой пары соответствующих диапазонов (по одному от каждой области) проверяется наличие пересечения. Если пересечение существует для всех измерений, результатом является новая область, содержащая результат пересечения для каждого измерения. Если хотя бы для одного измерения пересечение отсутствует, результат считается пустым.

Операция разности (*elementsDifference*) определяет множество элементов первой области, не принадлежащих второй. На вход подаются две области P и Q. Результатом является набор областей, описывающих части P, не покрываемые Q.

Сначала проверяются тривиальные случаи. Если область P пуста, результат также пуст. Если пуста область Q, результат совпадает с P, поскольку вычитать нечего. Далее находится пересечение $P \cap Q$ (операция *elementsIntersection*). Если пересечение отсутствует, разность совпадает с P. Если пересечение существует, алгоритм последовательно рассматривает каждое измерение области P. Для каждого измерения вычисляется одномерная разность соответствующих диапазонов. Поскольку исключение части области по одному измерению может порождать несколько новых диапазонов, результат представляется в виде набора областей. Каждая область в этом наборе описывает одну многомерную часть P, оставшуюся после вычитания Q.

Операция разности используется при добавлении новых областей во множество обращений к массиву, а также при вычислении разности множеств обращений.

5.4. Множество обращений к массиву. Для описания обращений к массиву недостаточно просто сопоставить каждому измерению диапазон. В программе могут существовать различные независимые обращения к разным участкам одного и того же массива. Рассмотрим пример (листинг 4), в котором в одном фрагменте цикла происходит обращение ко всем элементам массива $A(I, J)$, а в другом фиксируется одно измерение, и итерации выполняются только по второму измерению, например, рассматривается $A(I, 1)$. Однако в контексте анализа потока данных (множества *def*, *use*, *in* и *out*) все такие обращения к одному массиву должны рассматриваться совместно.

Для этого вводится понятие множества обращений к массиву — *AccessingSet*. Оно хранит набор областей, каждая из которых описывает один связный участок массива. Область задается вектором диапазонов, где каждый диапазон соответствует одному измерению массива и описывает закон изменения индексного выражения. Набор таких областей позволяет представить произвольные (в том числе несвязные) множества элементов массива.

Введенное множество *AccessingSet* является основной структурой для работы с областями массивов. Оно скрывает низкоуровневые вычисления над отдельными измерениями массива и предоставляет более

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

Листинг 4. Пример обращения к массивам
Listing 4. Example of accessing arrays

```
DO I = 1, N
    A(I, 1) = I + 1
    DO J = 1, M
        A(I, J) = A(I, J) + 10
    END DO
END DO
```

высокоуровневые операции над множествами обращений к элементам массивов. Эти операции используются при построении множеств *def*, *use*, *in* и *out* и при определении приватизируемых массивов.

В качестве иллюстрации рассмотрим представление следующего множества обращений к массиву $AccessingSet = \{R_0, R_1, \ldots, R_k\}$, где каждая область R_i имеет вид $R_i = (D_{i0}, D_{i1}, \ldots, D_{in})$, а D_{ij} описывает множество значений j-го индексного выражения. На рис. 2 изображено графическое представление *AccesingSet*, хранящее обращения к элементам двумерного массива. Далее рассмотрим основные необходимые операции над множеством обращений к *AccesingSet*.

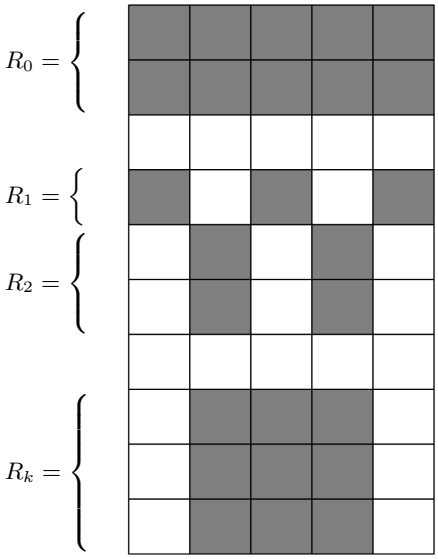

Рис. 2. Графическое представление *AccesingSet*
Fig. 2. Graphical representation of *AccesingSet*

1. Поиск непокрытых частей области массива. Данная процедура получает на вход многомерную область (вектор диапазонов) и множество областей. Она возвращает те части исходной области, которые не покрываются ни одной областью из переданного множества. Алгоритм последовательно вычитает каждую область из текущего множества. Ниже приведен алгоритм 3 этой процедуры в форме псевдокода.

Алгоритм 3. Поиск непокрытых частей области массива
Algorithm 3. Finding uncovered parts of an array region

Вход: область *element*, множество областей *allElements*
Выход: множество областей *result*, не покрытых элементами из *allElements*
function *FindUncovered*
  result ← {*element*}
  for каждого *currentElement* ∈ *allElements* **do**
    newTails ← ∅
    for каждого *tail* ∈ *result* **do**
      intersection ← *elementsIntersection*(*tail*, *currentElement*)
      diff ← *elementsDifference*(*tail*, *intersection*)
      if *diff* ≠ ∅ **then**
        newTails ← *newTails* ∪ *diff*
      end if
    end for
    result ← *newTails*
  end for
  return *result*
end function

2. Проверка вложенности области в множество. Данная операция выполняет проверку, полностью ли заданная многомерная область массива содержится в текущем множестве областей *AccesingSet*.

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

Проверка выполняется с помощью операции поиска непокрытых частей области массива. Если после последовательного вычитания всех областей из множества не остается непокрытых частей, область полностью покрывается множеством. Такой подход позволяет корректно обрабатывать случаи, когда исходная область покрывается не одной, а несколькими областями из множества. Ниже приведен алгоритм 4 этой процедуры в форме псевдокода.

Алгоритм 4. Проверка вложенности области в множество

Algorithm 4. Checking if a region is contained in a set

Вход: область *element*, множество областей *allElements*
Выход: **true**, если *element* полностью покрывается множеством *allElements*
function *ContainsElement*
    $tails \leftarrow FindUncovered(element, allElements)$
    if $tails = \varnothing$ **then**
        return true
    else
        return false
    end if
end function

3. Поиск частей области, покрываемых множеством. Данная операция определяет, какие части заданной области покрываются текущим множеством областей. На вход передается область массива. Для каждого элемента из множества вычисляется пересечение с этой областью. Если пересечение не пусто, оно добавляется в результат. Таким образом, метод возвращает набор областей, которые одновременно принадлежат исходной области и текущему множеству *AccesingSet*. Ниже приведен алгоритм 5 этой процедуры в форме псевдокода.

Алгоритм 5. Поиск частей области, покрываемых множеством

Algorithm 5. Finding parts of a region covered by a set

Вход: область *element*, множество областей *allElements*
Выход: множество частей *result*, покрываемых элементами из *allElements*
function *FindCoveredBy*
    $result \leftarrow \varnothing$
    for каждого $currentElement \in allElements$ **do**
        $intersection \leftarrow ElementsIntersection(element, currentElement)$
        if $intersection \neq \varnothing$ **then**
            $result \leftarrow result \cup \{intersection\}$
        end if
    end for
    return *result*
end function

4. Добавление области в множество AccessingSet. Данная операция добавляет новую область массива в множество *AccesingSet*. Перед вставкой она определяет, какие части новой области еще не покрыты уже существующими областями. Для этого вызывается операция *FindUncovered*. Это позволяет хранить только непересекающиеся области и обеспечивает единственность представления каждого участка массива. Ниже приведен алгоритм 6 этой процедуры в форме псевдокода.

Алгоритм 6. Добавление области в множество *AccessingSet*

Algorithm 6. Inserting a region into *AccessingSet*

Вход: область *element*, множество областей *allElements*
Выход: обновленное множество *allElements*
function *Insert*
    $tails \leftarrow FindUncovered(element, allElements)$
    $allElements \leftarrow allElements \cup tails$

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

```
    return allElements
end function
```

5. Операция объединения в множество AccessingSet. Данная операция строит объединение текущего множества областей с другим множеством областей. Если одно из множеств пусто, результатом является другое множество. В общем случае создается новое множество, в которое последовательно добавляются все области из обоих исходных множеств. Добавление выполняется через операцию *Insert*, поэтому перекрывающиеся области автоматически не дублируются. Ниже приведен алгоритм 7 этой операции в форме псевдокода.

Алгоритм 7. Объединение двух множеств обращений

Algorithm 7. Union of two access sets

```
Вход: множества областей P и Q
Выход: множество result = P ∪ Q
function Union
    if Q = ∅ then
        return P
    end if
    if P = ∅ then
        return Q
    end if
    result ← ∅
    for каждого element ∈ Q do
        Insert(result, element)
    end for
    for каждого element ∈ P do
        Insert(result, element)
    end for
    return result
end function
```

6. Операция пересечения множеств AccessingSet. Данная операция вычисляет пересечение текущего множества областей с другим множеством областей. Она последовательно рассматривает каждую область из текущего множества. Если область целиком содержится во втором множестве, она добавляется в результат полностью. В противном случае вычисляются части этой области, которые покрываются вторым множеством, и они добавляются в результат. Результатом является множество областей, принадлежащих обоим исходным множествам. Ниже приведен алгоритм 8 этой процедуры в форме псевдокода.

Алгоритм 8. Пересечение двух множеств обращений

Algorithm 8. Intersection of two access sets

```
Вход: множества областей P и Q
Вход: множества областей P и Q
Выход: множество result = P ∩ Q
function Intersect
    result ← ∅
    if P = ∅ или Q = ∅ then
        return result
    end if
    for каждого element ∈ P do
        if ContainsElement(Q, element) then
            result ← result ∪ {element}
        else
            coveredBy ← FindCoveredBy(Q, element)
            if coveredBy ≠ ∅ then
                result ← result ∪ coveredBy
            end if
        end if
    end for
    return result
end function
```

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

7. Операция разности множеств AccessingSet. Данная операция вычисляет разность двух множеств *AccessingSet*. Она возвращает те части текущего множества, которые не входят во второе множество. Сначала вычисляется пересечение первого множества со вторым. Затем для каждой области из первого множества определяются элементы, не покрытые этим пересечением. Непокрытые части формируют результирующее множество. Ниже приведен алгоритм 9 этой процедуры в форме псевдокода.

Алгоритм 9. Разность двух множеств обращений

Algorithm 9. Difference of two access sets

Вход: множества областей P и Q
Выход: множество $result = P \setminus Q$
function *Diff*
  if $Q = \varnothing$ **или** $P = \varnothing$ **then**
    return P
  end if
  $intersection \leftarrow Intersect(P, Q)$
  $uncovered \leftarrow \varnothing$
  for каждого $element \in P$ **do**
    $currentUncovered \leftarrow FindUncovered(intersection, element)$
    $uncovered \leftarrow uncovered \cup currentUncovered$
  end for
  return *uncovered*
end function

6. Алгоритм поиска приватных массивов в циклах. Определим этапы алгоритма поиска приватизируемых массивов. Алгоритм рекурсивно обрабатывает циклы программы, начиная с самых внутренних, и для каждого цикла L выполняет последовательность этапов, описанных ниже.

6.1. Сбор локальной информации. На первом шаге алгоритма выполняется сбор локальной информации об инструкциях, входящих в тело рассматриваемого цикла. Исходными данными для данного этапа являются промежуточное представление программы (IR) и граф потока управления цикла L с удаленными обратными ребрами. Для каждой инструкции t формируются множества $def[t]$, $use[t]$ и $kill[t]$ на основе данных из IR. Множества $in[t]$ и $out[t]$ на данном этапе инициализируются пустым множеством.

Если в процессе обхода обнаруживается вложенный цикл M, то к нему рекурсивно применяется тот же алгоритм анализа. После завершения обработки вложенного цикла полученные для него данные сохраняются в текущем графе путем операции свертки. При анализе внешнего цикла вложенный цикл рассматривается как один агрегированный базовый блок. Ниже приведен алгоритм 10 данного шага в форме псевдокода.

Алгоритм 10. Сбор локальной информации

Algorithm 10. Collecting local information

Вход: граф потока управления цикла L с удаленными обратными ребрами
Выход: множества $def[t]$, $use[t]$, $kill[t]$, $in[t]$, $out[t]$ для каждой инструкции t
function *CollectLocalInformation*
  for каждой инструкции $t \in body(L)$ в топологическом порядке **do**
    if t является заголовком вложенного цикла M **then**
      $(def[M], use[M]) \leftarrow privatize(M)$
      $L \leftarrow COLLAP(L, M)$
    else
      собрать локальные множества $def[t]$, $use[t]$, $kill[t]$
      $in[t] \leftarrow \varnothing$
      $out[t] \leftarrow \varnothing$
    end if
  end for
end function

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

6.2. Решение уравнений потока данных. После формирования множеств *def*, *use* и *kill* выполняется вычисление множеств *in* и *out* для инструкций тела цикла. Поскольку обратные ребра удалены, то граф потока управления рассматриваемого цикла является ориентированным ациклическим графом. Поэтому вычисление выполняется с помощью одного прохода по инструкциям цикла в топологическом порядке.

Для этого применяется оператор сбора и передаточная функция, описанные в формуле (1). Для каждой инструкции t множество $in[t]$ вычисляется оператором сбора на основе множеств *out* ее предшественников. После этого с помощью передаточной функции вычисляется множество $out[t]$. Ниже приведен алгоритм 11 данного шага в форме псевдокода.

Алгоритм 11. Решение уравнений потока данных

Algorithm 11. Solving data flow equations

Вход: граф потока управления цикла L; множества *def*, *use*, *kill*
Выход: множества $in[t]$ и $out[t]$ для каждой инструкции t
function *SolveDataFlow*
  for каждой инструкции $t \in body(L)$ в топологическом порядке **do**
    $in[t] \leftarrow \bigcap\limits_{p \in pred(t)} out[p]$
    $out[t] \leftarrow (in[t] \setminus kill[t]) \cup def[t]$
  end for
end function

6.3. Вычисление данных для тела цикла. Далее на основе вычисленных данных строятся множества приватизируемых переменных и множества, используемые на этапе свертки цикла. Множество $def_b[L]$ содержит переменные, которые гарантированно определены после выполнения тела цикла. Для этого используются множества *out* всех выходов из цикла, а результат строится как их пересечение.

Определения множества $use_b[L]$, содержащего переменные, заданные вне текущего витка, и множества $pri_b[L]$, состоящего из приватизируемых переменных цикла L, приведены в разделе 4.4. Ниже приведен алгоритм 12 данного этапа в форме псевдокода.

Алгоритм 12. Вычисление данных для тела цикла

Algorithm 12. Computing loop body data

Вход: граф потока управления цикла L; множества $in[t]$, $out[t]$, $def[t]$, $use[t]$
Выход: Множества $def_b[L]$, $use_b[L]$, $pri_b[L]$
function *ComputeLoopBodyData*
  $def_b[L] \leftarrow \bigcap\limits_{t \in exits(L)} out[t]$
  $use_b[L] \leftarrow \bigcup\limits_{t \in L} (use[t] \setminus in[t])$
  $pri_b[L] \leftarrow \bigcup\limits_{t \in L} use[t] \setminus use_b[L]$
end function

6.4. Создание агрегированных множеств. На заключительном этапе алгоритма поиска приватных массивов в цикле множества полученные на этапе вычисления данных для тела цикла преобразуются в итоговые множества, характеризующие весь цикл L как базовый блок.

Множеству $def[L]$ присваивается $def_b[L]$, а $use[L]$ вычисляется по формуле (2). Далее эти множества могут быть использованы при анализе объемлющего цикла. Ниже приведен алгоритм 13 заключительного этапа алгоритма поиска приватных массивов в цикле в форме псевдокода.

Алгоритм 13. Создание агрегированных множеств

Algorithm 13. Creating aggregated sets

Вход: множества $def_b[L]$, $use_b[L]$, цикл L
Выход: итоговые множества $def[L]$, $use[L]$
function *CreateAggregatedSets*
  $def[L] \leftarrow def_b[L]$

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

5: $use[L] \leftarrow use_b[L] \setminus def[L]$
6: **return** $(def[L], use[L])$
7: **end function**

7. Межпроцедурный анализ и неизвестные границы цикла. Для корректного анализа приватизируемости массивов необходимо знать границы циклов (начальное значение, конечное значение и шаг), поскольку именно они определяют диапазоны изменения индексных выражений массивов. Однако в реальных программах они могут быть заданы переменными, значения которых неизвестны на этапе статического анализа. В данном разделе рассматриваются два подхода к решению этой проблемы: использование динамических параметров задачи и распространение констант через элементы массивов.

7.1. Динамические параметры задачи. Во многих научно-технических программах размеры решаемой задачи задаются пользователем во время выполнения — через чтение из файла, со стандартного потока ввода или как результат предварительных вычислений. Такие переменные называются динамическими параметрами задачи. К ним относятся переменные, через которые напрямую или транзитивно выражаются размерности массивов и количество витков циклов.

Для того чтобы система SAPFOR могла определить границы цикла на этапе статического анализа, пользователь может указать значения динамических параметров с помощью специальной директивы (листинг 5).

Листинг 5. Пример использования директивы `PARAMETER`
Listing 5. Example of using the `PARAMETER` directive

```
READ(*, *) n, m, k
!$SPF ANALYSIS (PARAMETER (n=100, m=100, k=100))
DO i = 1, n
    DO j = 1, m
        DO l = 1, k
            A(i, j, l) = ...
        END DO
    END DO
END DO
```

Однако в больших программных комплексах ручной поиск всех таких переменных трудоемок. Для автоматизации этого процесса в системе SAPFOR ранее был реализован алгоритм поиска минимального множества таких переменных, через которые определяются параметры циклов и размерности массивов. Данный алгоритм выполняет следующие шаги:

- находит все операторы выделения массивов и заголовки циклов;
- для каждого такого оператора восходящим анализом по дереву доминаторов собирает множество переменных, от которых зависят размеры массивов или границы циклов;
- для каждой переменной проверяет единственность достигающего определения. Если определение единственно, анализ продолжается. Если определений несколько или они отсутствуют, переменная включается в результирующее множество динамических параметров;
- создает директивы `PARAMETER` в точках программы, где пользователь должен указать значения этих переменных.

Благодаря этому алгоритму пользователь получает подсказки о том, для каких переменных требуется уточнение, а также может указать их значения прямо в директивах, не изменяя исходный код программы. При этом гарантируется, что множество таких переменных будет минимальным, то есть пользователю не нужно будет специфицировать переменные, которые задают границы каждого из циклов по всей программе.

7.2. Распространение констант через элементы массивов. Однако описанный выше подход не работает, если границы цикла заданы не через скалярные переменные, а через обращения к элементам массивов (листинг 6).

В этом случае переменная `grid_points(1)` является элементом массива. Существующий в SAPFOR механизм распространения констант работает только со скалярными переменными и не отслеживает зна-

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

Листинг 6. Пример неизвестных границ цикла
Listing 6. An example of unknown loop boundaries

```
DO i = 1, grid_points(1)
    ! loop body
END DO
```

чения, хранящиеся в элементах массивов. Без дополнительной информации граница цикла остается неизвестной и анализ приватизируемости становится невозможным. Для устранения этого ограничения был разработан алгоритм распространения констант через элементы массивов. Его идея заключается в следующем:

- для каждого обращения к элементу массива с константным индексным выражением в заголовке цикла вводится временная скалярная переменная;
- присваивание значения данному элементу массива осуществляется в два этапа: сначала значение заносится во временную переменную, и лишь затем выполняется запись в сам массив;
- после такой замены параметры цикла оказываются выражены через скалярные переменные, и стандартное распространение констант становится применимым;
- после завершения анализа временные переменные удаляются, а программа восстанавливается в исходный вид. В итоге временные изменения не видны при последующих преобразованиях.

Алгоритм применяется только к тем обращениям к массиву, у которых все индексные выражения являются целочисленными константами, что гарантирует корректность замены. Этот алгоритм используется также в том случае, если данные массивы являются глобальными, например, объявлены в common-блоке или модуле. Это позволяет не ограничиваться только текущей процедурой. В листинге 7 приведен пример цикла с границей, заданной через элемент массива, до временного преобразования и после.

Листинг 7. Преобразование цикла с неизвестными границами
Listing 7. Transforming a loop with unknown bounds

```
! BEFORE TRANSFORMATION
grid_points(1) = 162
....
DO i = 1, grid_points(1)
    output(i) = i * 2
END DO


! AFTER TRANSFORMATION
grid_points(1) = 162
tmp_var = 162
....
DO i = 1, tmp_var
    output(i) = i * 2
END DO
```

7.3. Подстановка процедур на уровне IR. Дополнительной проблемой при анализе циклов является наличие вызовов процедур в теле цикла. Если вызываемая процедура содержит обращения к массивам или сама содержит циклы, ее необходимо проанализировать. В данной работе используется подход, при котором подстановка процедур выполняется на уровне промежуточного представления. Это означает, что тело вызываемой процедуры встраивается непосредственно в IR-инструкции анализируемого цикла. Такая подстановка не видна пользователю в выходной программе, но позволяет алгоритму приватизации обрабатывать тело процедуры, вызываемое из цикла.

7.4. Совместное использование методов. Описанные методы являются взаимодополняющими. Динамические параметры задачи позволяют определить значения скалярных переменных, от которых

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

зависят границы циклов и размерности массивов. Автоматический поиск таких переменных избавляет пользователя от необходимости вручную анализировать код.

Распространение констант через элементы массивов расширяет класс анализируемых программ на случаи, когда границы заданы через элементы массивов, что особенно часто встречается в научно-технических программах.

При совместном применении подстановки процедур на уровне промежуточного представления данные методы позволяют добиться того, что границы любого цикла, подпадающего под ограничения алгоритма приватизации, оказываются известны еще на стадии статического анализа. Это позволяет гарантировать корректное и предсказуемое поведение алгоритма автоматического определения приватизируемых массивов.

8. Исследование реализованного алгоритма поиска приватных массивов. Для проверки корректности и применимости разработанного алгоритма было проведено его исследование на программах из пакета тестов NAS Parallel Benchmarks (NPB) [27]. Для тестирования были выбраны четыре программы: BT (Block Tri-diagonal), SP (Scalar Pentadiagonal), LU (Lower-Upper) и EP (Embarrassingly Parallel). Эти программы охватывают различные вычислительные схемы и структуры циклов, что позволяет исследовать алгоритм на разнообразных сценариях использования приватных массивов. Программы BT, LU и SP содержат примерно 3000 строк кода на языке Fortran, программа EP — около 850 строк. Количество циклов в программах BT, LU и SP варьируется от 142 до 156, а в случае EP равно 28. Количество массивов, которое необходимо проанализировать, для программ BT, LU и SP варьируется от 38 до 42, а для EP равно 8.

Программа BT решает систему нелинейных дифференциальных уравнений в частных производных, используя блочную трехдиагональную схему с методом переменных направлений. Основной объем вычислений приходится на процедуры `x_solve`, `y_solve`, `z_solve` и `rhs`, каждая из которых содержит гнезда циклов с вложенностью до четырех уровней. В ходе анализа алгоритм выявил приватные массивы `fjac`, `njac`, `lhs` в процедурах `x_solve`, `y_solve`, `z_solve`. Также были найдены приватный массив `u_exact` в процедуре вычисления ошибки `error_norm` и приватные массивы `Pface` и `temp` в процедуре инициализации `initialize`.

Программа SP решает ту же систему уравнений, что и BT, но использует скалярную пятидиагональную схему. Ее структура во многом схожа с BT: основные вычисления выполняются в процедурах `x_solve`, `y_solve`, `z_solve` и `rhs`, каждая из которых содержит гнезда циклов с вложенностью до четырех уровней. В ходе анализа алгоритм выявил приватные массивы `lhsp`, `lhsm`, `lhs` в процедурах `x_solve`, `y_solve`, `z_solve`. Также были найдены приватный массив `u_exact` в процедуре вычисления ошибки `error_norm` и приватные массивы `Pface` и `temp` в процедуре инициализации `initialize`.

Программа LU решает систему уравнений методом симметричной последовательной верхней релаксации. Основной объем вычислений приходится на процедуры `ssor`, `blts`, `buts`, `jacld`, `jacu` и `rhs`, каждая из которых содержит гнезда циклов с вложенностью до трех уровней. В ходе анализа алгоритм выявил приватные массивы `flux`, `utmp`, `rtmp` в процедуре `rhs`, приватные массивы `tmat` и `tv` в процедурах `ssor`, `blts` и `buts`, приватные массивы `ue_1jk`, `ue_nx0jk`, `ue_i1k`, `ue_iny0k`, `ue_ij1` и `ue_ijnz` в процедуре `setiv` и приватные массивы `temp1` и `temp2` в процедуре инициализации `setbv`.

Программа EP представляет собой тест, в котором вычисляется большое количество случайных чисел с последующим выполнением трудоемких математических операций. Основной объем вычислений приходится на главный цикл по числу генерируемых псевдослучайных чисел. В ходе анализа алгоритм выявил приватный массив `x`. Массив `x` используется для хранения последовательности генерируемых случайных чисел.

Разработанный в данной работе алгоритм автоматического определения приватизируемых массивов позволяет полностью исключить ручной ввод информации о приватных переменных. Чтобы оценить его качество, проанализируем результаты применения преобразования удаления приватных переменных, которое становится полностью автоматическим благодаря разработанному алгоритму. Как показано в работе [9], данное преобразование демонстрирует высокую эффективность на рассмотренных программах из пакета NAS Parallel Benchmarks.

В частности, для программы LU удалось добиться ускорения выполнения в 3.3 раза на GPU и в 1.6 раз на 32 потоках CPU, при этом потребление памяти сократилось почти в 5.3 раза. В программе EP, где удаление единственного массива оказалось возможным, ускорение на GPU достигло двух порядков, а потребление памяти снизилось на пять порядков (с 64 ГБ до 0.001 ГБ). Для программы BT удаление

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

части переменных дало ускорение почти в 2.8 раз на GPU и почти в 3.2 раза на CPU. Для программы SP ускорение не таким высоким — 18% на GPU и 28% на CPU. Это объясняется тем, что удаляемые массивы в программе SP имеют значительно меньший размер, чем в программе BT. Однако, несмотря на меньший размер, их удаление привело к более заметному сокращению потребляемой памяти (16% для программы SP, в то время как для программы BT разница в потреблении памяти отсутствовала).

Таким образом, предлагаемый в настоящей статье алгоритм обеспечивает необходимую информацию для проведения высокоэффективных преобразований, что подтверждается кратным ускорением выполнения программ и значительным сокращением используемой памяти. Это позволяет органично встроить разработанный алгоритм в общую цепочку автоматизированных преобразований системы SAPFOR, сделав ее более мощным инструментом для автоматического распараллеливания.

9. Заключение. В данной статье разработан и описан алгоритм автоматического определения приватизируемых массивов в циклах последовательных Fortran-программ, а также представлена его реализация в системе автоматизированного распараллеливания SAPFOR. Задача приватизации является одной из ключевых при подготовке программы к параллельному выполнению, поскольку она позволяет устранить зависимости по данным, препятствующие распараллеливанию циклов, без изменения общей логики программы. Однако до настоящего времени этот процесс требовал ручного вмешательства разработчика, что ограничивало степень автоматизации и увеличивало трудоемкость подготовки параллельных версий.

В рамках исследования был разработан и описан алгоритм статического анализа циклов, реализующий автоматический поиск приватизируемых массивов. Алгоритм основан на анализе графа потока управления с удалением обратных ребер, решении уравнений потока данных и операции свертки вложенных циклов. Сформулированы условия его применимости, а также особенности использования в составе системы автоматизированного распараллеливания SAPFOR. Для представления обращений к массивам использовано компактное описание областей через диапазоны изменения индексных выражений, что позволило избежать поэлементного хранения информации и обеспечить применимость алгоритма к массивам большого размера. Дополнительно предложен вспомогательный метод распространения констант через элементы массивов, расширяющий возможности анализа на циклы с границами, заданными через элементы массивов.

Разработанный алгоритм был реализован в системе SAPFOR в виде отдельного прохода преобразования. Его работа включает построение представления цикла в виде графа потока управления, сбор локальной информации об обращениях к элементам массивов, решение уравнений потока данных и формирование результата анализа в виде директивы `!$SPF ANALYSIS (PRIVATE(A1, A2,..., An))`, содержащей имена приватизируемых массивов.

Реализованный алгоритм был протестирован на четырех прикладных программах из пакета тестов NAS Parallel Benchmarks: BT, LU, SP и EP. В процессе анализа в каждой из этих программ были обнаружены приватизируемые массивы, что подтверждает корректность работы алгоритма и его применимость к реальным научно-техническим программам различной структуры и сложности.

Полученные результаты являются не только самостоятельным результатом, но и необходимой основой для последующих преобразований. В частности, в статье [9] было предложено преобразование удаления приватных переменных, которое позволяет заменить обращения к приватному массиву прямым вычислением необходимых значений, устраняя избыточное потребление памяти и сокращая количество обращений к данным. Применение этого преобразования к программам LU и EP позволило добиться кратного ускорения выполнения программы на графических ускорителях и кратного сокращения потребления памяти. Автоматическое определение приватизируемых массивов, предлагаемое в настоящей статье, позволяет полностью исключить ручной ввод информации о приватных переменных и сделать преобразование удаления приватных переменных полностью автоматическим. В результате разработанный алгоритм органично встраивается в общую цепочку автоматизированных преобразований системы SAPFOR.

Таким образом, система автоматизированного распараллеливания программ SAPFOR была дополнена алгоритмом автоматического определения приватизируемых массивов в последовательных Fortran-программах. Данное исследование является важным этапом на пути к созданию полностью автоматической системы распараллеливания, способной минимизировать участие разработчика в процессе подготовки параллельной версии программы. В отличие от полностью автоматических компиляторов на основе полиэдральной модели, SAPFOR сохраняет возможность интерактивного участия разработчика, что делает ее применимой к более широкому классу программ. Разработанный алгоритм приватизации сокращает

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

объем ручной работы и приближает систему к главной цели — созданию инструмента, способного автоматически генерировать эффективные параллельные версии сложных научно-технических программ.

Список литературы

1. *Таненбаум Э.С., Бос Х.* Современные операционные системы. 4-е изд. И.Д. “Питер”, 2015.
2. OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 5.1, 2020. https://www.openmp.org/specifications/. (Дата обращения: 14 августа 2026).
3. OpenACC-Standard.org. The OpenACC Application Programming Interface, Version 3.3, 2022. https://www.openacc.org/specification. Cited August 10, 2026.
4. XcalableACC Language Specification, Version 1.0. RIKEN AICS and University of Tsukuba, 2017. http://xcalablemp.org/download/XACC/xacc-spec-1.0.pdf. Cited August 10, 2026.
5. Cetus: A Parallelizing Source-to-Source Compiler for C Programs. https://sites.udel.edu/cetus-cid/. Cited August 10, 2026.
6. *Grosser T., Groesslinger A., Lengauer C.* Polly — performing polyhedral optimizations on a low-level intermediate representation // Parallel Processing Letters. 2012. **22**, N 04. Article Number 1250010. doi 10.1142/S0129626412500107.
7. CUDA Toolkit Documentation. NVIDIA Corporation. https://docs.nvidia.com/cuda/. Cited August 10, 2026.
8. DVM-система | Система разработки параллельных программ. Документация по языкам C-DVMH и Fortran-DVMH. http://dvm-system.org/ru/docs/. (Дата обращения: 10 августа 2026).
9. *Колганов А.С., Гусев Г.Д.* Реализация преобразования удаления приватных переменных в последовательных Fortran-программах для их эффективного распараллеливания на вычислительные кластеры в системе SAPFOR // Вычислительные методы и программирование. 2025. **26**, № 1. 58–84. doi 10.26089/NumMet.v26r105.
10. *Li Z.* Array privatization for parallel execution of loops // Proceedings of the 6th ACM International Conference on Supercomputing (ICS ’92), Washington D.C. USA, July 19–24, 1992. New York: ACM, 1992. pp. 313–322. doi 10.1145/143369.143426.
11. *Rauchwerger L., Padua D.* The privatizing DOALL test: a run-time technique for DOALL loop identification and array privatization // Proceedings of the 8th International Conference on Supercomputing (ICS ’94), Manchester, England, July 11–15, 1994. New York: ACM, 1994. pp. 33–43. doi 10.1145/181181.181254.
12. *Tu P., Padua D.* Automatic array privatization // Proc. in Languages and Compilers for Parallel Computing. LCPC 1993. Editors Banerjee U., Gelernter D., Nicolau A., Padua D. Lecture Notes in Computer Science, vol 768. Springer, Berlin, Heidelberg, 1994, pp. 500–521. doi 10.1007/3-540-57659-2_29.
13. *Blume B., Eigenmann R., Faigin K., et al.* Polaris: The Next Generation in Parallelizing Compilers // Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing. Ithaca, NY, USA, 1994. 459–474.
14. *Gu J., Li Z.* Efficient interprocedural array data-flow analysis for automatic program parallelization // IEEE Transactions on Software Engineering. 2000. **26**, N 3. 244–261. doi 10.1109/32.842950.
15. *Rus S., He G., Alias C., Rauchwerger L.* Region Array SSA // Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT ’06), Seattle, WA, USA, September 16–20, 2006. New York: ACM, 2006. pp. 43–52. doi 10.1145/1152154.1152165.
16. *Feautrier P., Lengauer C.* Polyhedron Model // Encyclopedia of Parallel Computing. New York: Springer, 2011. 1581–1592.
17. *Bondhugula U., Hartono A., Ramanujam J., Sadayappan P.* A practical automatic polyhedral parallelizer and locality optimizer // ACM SIGPLAN Notices. 2008. **43**, N 6. 101–113. doi 10.1145/1379022.1375595.
18. *Verdoolaege S., Juega J.C., Cohen A., et al.* Polyhedral parallel code generation for CUDA // ACM Transactions on Architecture and Code Optimization. 2013. **9**, N 4. Article Number 54. doi 10.1145/2400682.2400713.
19. *Grosser T., Hoefler T.* Polly-ACC transparent compilation to heterogeneous hardware // Proceedings of the International Conference on Supercomputing, Istanbul, Turkey, June 1–3, 2016. New York: ACM Press, 2016. 1–13. doi 10.1145/2925426.2926286.
20. *Lattner C., Adve V.* LLVM: A compilation framework for lifelong program analysis and transformation // Proceedings of the International Symposium on Code Generation and Optimization (CGO’04), San Jose, USA, March 20–24, 2004. IEEE Press, 2004. pp. 75–86. doi 10.1109/CGO.2004.1281665.
21. *Wienke S., Springer P., Terboven C., an Mey D.* OpenACC — first experiences with real-world applications // Proceedings of the 18th International Conference on Parallel Processing (Euro-Par 2012), Rhodes Islands, Greece, August 27–31, 2012. Berlin: Springer Berlin, 2012. pp. 859–870. doi 10.1007/978-3-642-32820-6_85.

22. Система Автоматизированной Параллелизации ФОРтран-программ (САПФОР). http://keldysh.ru/dvm/SAPFOR/. (Дата обращения: 10 августа 2026).
23. *Бахтин В.А., Жукова О.Ф., Катаев Н.А., Колганов А.С., и др.* Автоматизация распараллеливания программных комплексов // Труды XVIII Всероссийской научной конференции "Научный сервис в сети Интернет", Новороссийск, 19–24 сентября, 2016. М.: ИПМ им. М.В. Келдыша, 2016. 76–85. doi 10.20948/abrau-2016-31.
24. *Колганов А.С.* Автоматизация распараллеливания Fortran-программ для гетерогенных кластеров. Автореф. дисс. канд. физ.-мат. наук. М.: ИПМ им. М.В. Келдыша, 2020.
25. *Ахо А.В., Лам М.С., Сети Р., Ульман Д.Д.* Компиляторы: принципы, технологии и инструментарий. 2-е изд. М.: И.Д. Вильямс, 2008.
26. *Kahn A.B.* Topological sorting of large networks // Communications of the ACM. 1962. **5**, N 11. 558–562. doi 10.1145/368996.369025.
27. NAS Parallel Benchmarks. https://www.nas.nasa.gov/software/npb.html. Cited August 10, 2026.

Получена 10 июня 2026 г. | *Принята* 14 июля 2026 г. | *Опубликована* 17 августа 2026 г.

Информация об авторах

Александр Сергеевич Колганов — к.ф.-м.н., науч. сотр.; 1) Московский государственный университет имени М. В. Ломоносова, факультет вычислительной математики и кибернетики, Ленинские горы, 1, стр. 4, 119991, Москва, Российская Федерация; 2) Институт прикладной математики имени М. В. Келдыша (ИПМ РАН), Миусская пл., 4, 125047, Москва, Российская Федерация.

Олег Юрьевич Никитин — магистр; Московский государственный университет имени М. В. Ломоносова, факультет вычислительной математики и кибернетики, Ленинские горы, 1, стр. 4, 119991, Москва, Российская Федерация.

References

1. A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. (Pearson, Boston, 2015).
2. OpenMP Architecture Review Board, OpenMP Application Programming Interface Version 5.1 (2020). https://www.openmp.org/specifications/. Cited August 14, 2026.
3. OpenACC-Standard.org, The OpenACC Application Programming Interface, Version 3.3 (2022). https://www.openacc.org/specification. Cited August 10, 2026.
4. XcalableACC Language Specification, Version 1.0. RIKEN AICS and University of Tsukuba, (2017). http://xcalablemp.org/download/XACC/xacc-spec-1.0.pdf. Cited August 10, 2026.
5. Cetus: A Parallelizing Source-to-Source Compiler for C Programs. https://sites.udel.edu/cetus-cid/. Cited August 10, 2026.
6. T. Grosser, A. Groesslinger, and C. Lengauer, "Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation," Parallel Processing Letters **22** (04), Article Number 1250010 (2012). doi 10.1142/S0129626412500107.
7. CUDA Toolkit Documentation. NVIDIA Corporation. https://docs.nvidia.com/cuda/. Cited August 10, 2026.
8. DVM-system | System for developing parallel programs. Documentation for C-DVMH and Fortran-DVMH Languages. http://dvm-system.org/ru/docs/. Cited August 10, 2026.
9. A. S. Kolganov and G. D. Gusev, "Implementation of Private Variables Contraction Transformation of Sequential Fortran Programs for their Effective Parallelization into Computing Clusters in the SAPFOR," Numerical Methods and Programming **26** (1), 58–84 (2025). doi 10.26089/NumMet.v26r105.
10. Z. Li, "Array Privatization for Parallel Execution of Loops," in *Proceedings of the 6th ACM International Conference on Supercomputing (ICS '92), Washington D.C. USA, July 19–24, 1992* (ACM, New York, NY, USA, 1992), pp. 313–322. doi 10.1145/143369.143426.
11. L. Rauchwerger and D. Padua, "The privatizing DOALL test: A run-time technique for DOALL loop identification and array privatization," in *Proceedings of the 8th International Conference on Supercomputing (ICS '94), Manchester, England, July 11–15, 1994* (ACM, New York, NY, USA, 1994), pp. 33–43. doi 10.1145/181181.181254.

2026, **27** (3), 489–513. doi 10.26089/NumMet.v27r331

12. P. Tu, D. Padua, "Automatic array privatization," in *Banerjee U., Gelernter D., Nicolau A., Padua D. (eds) Languages and Compilers for Parallel Computing. LCPC 1993.* (Lecture Notes in Computer Science, vol 768. Springer, Berlin, Heidelberg, 1994), pp. 500–521. doi 10.1007/3-540-57659-2_29.
13. B. Blume, R. Eigenmann, K. Faigin, et al., "Polaris: The Next Generation in Parallelizing Compilers," in *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing* (Ithaca, NY, USA, 1994), pp. 459–474.
14. J. Gu and Z. Li, "Efficient Interprocedural Array Data-Flow Analysis for Automatic Program Parallelization," IEEE Transactions on Software Engineering **26** (3), 244–261 (2000). doi 10.1109/32.842950.
15. S. Rus, G. He, C. Alias, and L. Rauchwerger, "Region Array SSA," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT '06), Seattle, WA, USA, September 16–20, 2006* (ACM, New York, NY, USA, 2006), pp. 43–52. doi 10.1145/1152154.1152165.
16. P. Feautrier and C. Lengauer, "Polyhedron Model," in *Encyclopedia of Parallel Computing* (Springer, New York, 2011), pp. 1581–1592.
17. U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," ACM SIGPLAN Notices **43** (6), 101–113 (2008). doi 10.1145/1379022.1375595.
18. S. Verdoolaege, J. C. Juega, A. Cohen, et al., "Polyhedral Parallel Code Generation for CUDA," ACM Transactions on Architecture and Code Optimization **9** (4), Article Number 54 (2013). doi 10.1145/2400682.2400713.
19. T. Grosser and T. Hoefler, "Polly-ACC Transparent Compilation to Heterogeneous Hardware," in *Proceedings of the International Conference on Supercomputing, Istanbul, Turkey, June 1–3, 2016* (ACM Press, New York, NY, USA, 2016), pp. 1–13. doi 10.1145/2925426.2926286.
20. C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04), San Jose, USA, March 20–24, 2004* (IEEE Press, 2004), pp. 75–86. doi 10.1109/CGO.2004.1281665.
21. S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC — First Experiences with Real-World Applications," in *Proceedings of the 18th International Conference on Parallel Processing (Euro-Par 2012), Rhodes Islands, Greece, August 27–31, 2012* (Springer Berlin, Berlin, 2012), pp. 859–870. doi 10.1007/978-3-642-32820-6_85.
22. System for Automated Parallelization of FORtran programs (SAPFOR). http://keldysh.ru/dvm/SAPFOR/. Cited August 10, 2026.
23. V. A. Bakhtin, O. F. Zhukova, N. A. Kataev, A. S. Kolganov, et al., "Automation of Parallelization of Software Complexes," in *Proc. XVIII All-Rus. Sci. Conf. on "Scientific Service on the Internet", Novorossiysk, September 19–24, 2016* (KIAM RAS, Moscow, 2016), pp. 76–85. doi 10.20948/abrau-2016-31.
24. A. S. Kolganov, *Automation of Parallelization of Fortran Programs for Heterogeneous Clusters* Candidate's Dissertation in Mathematics and Physics. (Keldysh Inst. Applied Math., Moscow, 2020).
25. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools, 2th ed.* (Pearson/Addison Wesley, Boston, 2007).
26. A. B. Kahn, "Topological sorting of large networks," Communications of the ACM **5** (11), 558–562 (1962). doi 10.1145/368996.369025.
27. NAS Parallel Benchmarks. https://www.nas.nasa.gov/software/npb.html. Cited August 10, 2026.

Received	*Accepted*	*Published*
June 10, 2026	*July 14, 2026*	*August 17, 2026*

Information about the authors

Alexander S. Kolganov — Ph.D., Scientist; 1) Lomonosov Moscow State University, Faculty of Computational Mathematics and Cybernetics, Leninskie Gory, 1, building 4, 119991, Moscow, Russia; 2) Keldysh Institute of Applied Mathematics of RAS, Miusskaya ploshchad', 4, 125047, Moscow, Russia.

Oleg Yu. Nikitin — Master; Lomonosov Moscow State University, Faculty of Computational Mathematics and Cybernetics, Leninskie Gory, 1, building 4, 119991, Moscow, Russia.