2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

doi 10.26089/NumMet.v26r105 УДК 04.272, 004.4'242

Реализация преобразования удаления приватных переменных в последовательных Fortran-программах для их эффективного распараллеливания на вычислительные кластеры в системе SAPFOR

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

Г. Д. Гусев
Московский государственный университет имени М. В. Ломоносова,
Москва, Российская Федерация
ORCID: 0000-0002-3133-0649, e-mail: gr@medhim.ru

Аннотация: Процесс автоматизированного распараллеливания программ может быть существенно затруднен из-за их структуры и оптимизации под последовательное выполнение. Из-за этого полученная параллельная версия может быть неэффективной, а в некоторых случаях распараллеливание оказывается и вовсе невозможным. Решить указанные проблемы помогают преобразования исходного кода последовательных программ. В данной статье рассматривается разработка алгоритма преобразования последовательных Fortran-программ "удаление приватных переменных" и его реализация в системе автоматизированного распараллеливания SAPFOR (System FOR Automated Parallelization). Применение реализованных преобразований в системе SAPFOR продемонстрировано на четырех прикладных программах, входящих в пакет NAS Parallel Benchmarks.

Ключевые слова: SAPFOR (System FOR Automated Parallelization), автоматизация распараллеливания на кластер, автоматизация преобразований, параллельные вычисления, DVM (Distributed Virtual Memory), кластеры с графическими процессорами.

Для цитирования: 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.

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

Implementation of Private Variables Contraction Transformation of Sequential Fortran Programs for their Effective Parallelization into Computing Clusters in the SAPFOR

Alexander S. Kolganov
Keldysh Institute of Applied Mathematics of RAS, Moscow, Russia
ORCID: 0000-0002-1384-7484, e-mail: alexander.k.s@mail.ru
Grigorii D. Gusev
Lomonosov Moscow State University, Moscow, Russia
ORCID: 0000-0002-3133-0649, e-mail: gr@medhim.ru

Abstract: The process of automated parallelization of programs can be significantly complicated due to their structure and optimization for sequential execution. Because of this, the resulting parallel version may be ineffective, and in some cases parallelization turns out to be completely impossible. Transformations of the source code of sequential programs help to solve these problems. This article discusses the development of an algorithm for transformation of sequential Fortran programs called "removing of private variables" and its implementation in the SAPFOR automated parallelization system (System FOR Automated Parallelization). The application of the implemented transformations in the SAPFOR system is demonstrated on four application programs included in the NAS Parallel Benchmarks package.

Keywords: SAPFOR (System FOR Automated Parallelization), parallelization automation for clusters, transformation automation, parallel computing, DVM (Distributed Virtual Memory), GPU clusters.

For citation: A. S. Kolganov, 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.

1. Введение. Многие современные научные и технические задачи, решаемые на вычислительных системах, требуют обработки большого количества данных и выполнения большого объема вычислений над этими данными. Для решения подобных задач необходимо использование высокопроизводительных вычислительных систем. В настоящее время все такие системы являются многопроцессорными и нацелены на выполнение параллельных программ. Однако написание параллельных программ существенно отличается по сложности от написания последовательных и требует от программиста дополнительных знаний технологий параллельного программирования и особенностей архитектуры системы, на которой разрабатываемая программа будет выполняться. Ситуация осложняется наличием различных параллельных архитектур (таких как системы с распределенной и общей памятью, гибридные системы) и технологий параллельного программирования (таких как OpenMP [1], MPI [2], CUDA [3] и др.).

Кластером называется вычислительная система, состоящая из множества вычислительных узлов (каждый из которых может быть многопроцессорным), соединенных высокоскоростной сетью. Многие современные высокопроизводительные вычислительные системы являются гибридными кластерами, в узлах которых помимо центрального процессора (central processing unit, CPU) используется ускоритель в виде графического сопроцессора (graphics processing unit, GPU) или многоядерного высокопроизводительного сопроцессора. Примерами таких ускорителей являются графические сопроцессоры фирмы NVIDIA или AMD и многоядерные сопроцессоры Intel Xeon Phi и Intel Data Center GPU. Для достижения параллелизма на всех уровнях гибридной вычислительной системы и максимальной эффективности

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

ее использования программисту требуется знание и одновременное применение нескольких технологий параллельного программирования.

Особое место среди средств создания параллельных программ занимают модели, основанные на добавлении в программы на высокоуровневых последовательных языках (таких как Fortran и C) спецификаций параллелизма (также называемых директивами), осуществляющих распараллеливание последовательной программы. Указанные спецификации оформляются в виде специальных комментариев в Fortran-программах и прагм в C-программах и не воспринимаются обычными последовательными компиляторами. Подобные высокоуровневые модели позволяют получить параллельную программу из последовательной с минимальным изменением кода, иметь одну версию программы как для последовательного, так и для параллельного выполнения, а также упрощают переносимость программ с одной архитектуры вычислительной системы на другую. При этом высокоуровневость таких моделей позволяет абстрагироваться от конкретных низкоуровневых технологий параллельного программирования. С помощью спецификаций описывается структура вычислений программы и имеющиеся в ней зависимости по данным, выбор же конкретных технологий параллельного программирования и организация параллельных вычислений осуществляется компилятором. Это позволяет иметь одну параллельную версию программы для выполнения на вычислительных системах с общей и распределенной памятью, на гибридных системах и в любом возможном их сочетании.

Системы автоматизированного распараллеливания последовательных программ призваны облегчить для программиста процесс создания параллельной программы. Такие системы с помощью статического анализа собирают данные о вычислительной структуре программы и имеющихся в ней зависимостях по данным и на основе полученной информации генерируют параллельную версию программы. Однако процесс автоматизированного распараллеливания последовательной программы может быть существенно затруднен из-за ее структуры, имеющихся в ней зависимостей по данным и оптимизаций под последовательное выполнение. Это может приводить к неэффективности полученной параллельной версии программы или к невозможности автоматизированного распараллеливания вообще. Решить обозначенные проблемы в ряде случаев помогают преобразования исходной последовательной программы, позволяющие изменить в ней структуру вычислений или зависимости по данным, которые препятствуют получению эффективной параллельной версии.

2. Обзор существующих систем. Можно выделить два основных подхода, применяемых при разработке инструментов, автоматизирующих распараллеливание программ. Подход, наиболее понятный для пользователя и поэтому используемый в SAPFOR в силу ее направленности на поддержание процесса интерактивного распараллеливания, заключается в последовательном изменении кода программы (выполнение нескольких шагов преобразования программы), чтобы впоследствии привести ее к параллельному виду. Многие из выполняемых преобразований оказываются общими для разных программ и могут быть автоматизированы (подстановка функций, разделение и слияние циклов, разворот цикла и др. [4]). Часть из этих преобразований уже реализована в системе SAPFOR и при необходимости может быть запрошена пользователем. Альтернативой этому подходу является второй подход — использование математической модели, описывающей поведение распараллеливаемой программы (модель многогранников, или полиэдральная модель [5]). Этот подход позволяет сформировать оптимизационную задачу, решение которой фактически описывает переход от исходной программы к ее параллельной версии за один шаг преобразования (т.е. на вход подается исходная версия программы, а на выходе получается ее параллельная версия).

Одним из основных недостатков второго подхода является большое количество ограничений, накладываемых на распараллеливаемую программу (рассмотренных в работе [5]): в результате такой подход годится, в первую очередь, для локального применения к хорошо структурированным участкам кода (SCoP). Кроме того, степень изменений, вносимых в код программы, которые порождает решение оптимизационной задачи, оказывается такова, что пользователь фактически лишается возможности участвовать в процессе распараллеливания, даже если результирующая параллельная программа остается на языке высокого уровня. Таким образом, альтернативный подход должен применяться при разработке автоматически распараллеливающих компиляторов [5–9], а не инструментов автоматизации.

Компиляторы Pluto [5] и PPCG [6] направлены на преобразование только C-программ, при этом Pluto выполняет распараллеливание только на многоядерные процессоры (с использованием OpenMP), в то время как PPCG опирается на CUDA или OpenCL [10] для отображения программ на графические ускорители. Оба инструмента сохраняют высокий уровень исходного языка, но вносимые изменения

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

негативно сказываются на возможности восприятия кода неподготовленным программистом. Существенным недостатком является необходимость явно задавать фрагменты исходного кода, которые должны быть оптимизированы. Для этого используется специальная директива `scop`. Так как оптимизация выполняется локально, то от задания данной директивы существенно зависит результат распараллеливания. Например, если соседние гнезда циклов поместить в разные области распараллеливания, то PPCG добавит в код программы вызовы CUDA, выполняющие обмены с графическим ускорителем в начале и в конце каждой области: глобальная оптимизация обменов не выполняется. Кроме того, Pluto и PPCG не обрабатывают участки кода, содержащие редукционные операции, и оставляют соответствующий код последовательным. В этом случае вычисления не могут быть полностью выполнены на ускорителях и требуются дополнительные обмены данными с центральным процессором.

Другим примером применения модели многогранников является инструмент Polly [7] и его дальнейшее расширение для отображения программ на графические ускорители Polly-ACC [8]. Данные инструменты используют LLVM [11] для анализа и преобразования программ и доступны в составе компилятора Clang. Использование низкоуровневого представления (LLVM IR) лишает пользователя возможности каклибо изменять код параллельной программы, но при этом расширяет применимость данных инструментов к языкам, для которых может быть построено LLVM IR. В отличие от Pluto и PPCG, оптимизируемые участки кода определяются автоматически и также выполняется попытка оптимизировать обмены с графическим ускорителем. Отдельно была реализована возможность распараллеливания редукционных операций [12]. Хотя оптимизируемые участки кода определяются автоматически, выполнить распараллеливание удается не всегда: инструменты опираются на статический анализ кода, возможности которого ограничены, особенно если в программах используются указатели и адресная арифметика. Чтобы понять, как привести программу в распараллеливаемую форму, пользователю может потребоваться изучить возникшие проблемы анализа, описываемые в терминах низкоуровневого представления LLVM IR. Еще одним интересным инструментом является Apollo [9], осуществляющий спекулятивное распараллеливание программы во время выполнения. Но в данном инструменте реализована возможность распараллеливания только для многоядерных процессоров.

Примером высокоуровневой технологии параллельного программирования, основанной на добавлении в код программы спецификаций параллелизма, является DVM-система [13], созданная в ИПМ имени М. В. Келдыша РАН при активном участии аспирантов и студентов факультета ВМК МГУ имени М. В. Ломоносова. Она предназначена для разработки параллельных программ научно-технических расчетов. Модель DVMH является расширением модели DVM для гибридных кластеров. В модели используются высокоуровневые директивные языки Fortran-DVMH и C-DVMH [14], которые представляют собой расширения стандартных языков Fortran-95 и C-99 спецификациями параллелизма в виде специальных комментариев и прагм соответственно. Компилятор DVMH осуществляет распараллеливание программы с использованием технологий параллельного программирования OpenMP, MPI и CUDA. Далее в данной работе рассматривается только часть DVM-системы, относящаяся к созданию параллельных программ на языке Fortran-DVMH. Общий вид специальных комментариев в языке Fortran-DVMH (также называемых директивами), представлен в листинге 1.

Листинг 1. Общий вид директивы в языке Fortran-DVMH
Listing 1. General form of the directive in the Fortran-DVMH language

```
1 !DVM$ <DVMH-directive>
```

Модель DVMH предполагает распараллеливание программ с помощью параллелизма по данным, т.е. с помощью выявления в программе данных, которые могут быть распределены на разные процессоры и обработаны на этих процессорах параллельно. Вычислительная система представляется в виде массива виртуальных процессоров (в общем случае многомерного), который задает пользователь при запуске программы на выполнение. Такой массив также называется решеткой процессоров. С помощью директивы `DISTRIBUTE` пользователь задает отображение массивов, использующихся в программе, на решетку виртуальных процессоров. При этом каждое измерение массива либо распределяется на одно из измерений решетки процессоров, либо распределяется на каждый виртуальный процессор целиком (размножается). Модель предполагает порождение одного MPI-процесса на каждый виртуальный процессор. Если измерение массива распределяется на одно из измерений решетки виртуальных процессоров, то размер данного

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

измерения решетки определяет количество процессов, между которыми будет распределено данное измерение массива. Произведение размеров измерений решетки процессоров определяет общее количество процессов, на которых будет выполняться программа.

Через директиву `ALIGN` задается отображение (выравнивание) элементов одного массива на элементы другого массива. Соответствующие друг другу элементы массивов будут распределены на один виртуальный процессор. Также с помощью данной директивы можно выровнять массив по DVMH-шаблону. По одному шаблону может быть выровнено несколько массивов. DVMH-шаблон задает индексное пространство, которое определяет отображение элементов выровненных по данному шаблону массивов на множество виртуальных процессоров и, в отличие от массивов, не занимает места в памяти. Переменные, для которых не введено распределение с помощью директив `DISTRIBUTE` и `ALIGN`, автоматически распределяются на каждый виртуальный процессор (размножаются).

Параллельный цикл в модели DVMH рассматривается как массив витков цикла. Количество измерений массива определяется степенью тесновложенности данного цикла (тесновложенные циклы в языке Fortran описаны в разделе 3). С помощью директивы `PARALLEL` задается отображение витков цикла на элементы некоторого распределенного массива. Каждый виток цикла будет выполнен на том виртуальном процессоре, на который был распределен соответствующий витку элемент массива. При этом должно соблюдаться правило собственных вычислений: виртуальный процессор имеет право присваивать значения только в собственные элементы массива, т.е. в те элементы, которые распределены на данный процессор [14].

Вычислительным регионом (или просто регионом) в модели DVMH называется часть программы с одним входом и одним выходом, которая может быть выполнена на одном или нескольких вычислительных устройствах. В частности, регион может быть выполнен на графическом ускорителе. В листинге 2 приведен общий вид региона, который состоит из блока операторов, заключенных в пару директив. В списке спецификаций `<specs>` указываются входные, выходные и локальные данные для региона.

Листинг 2. Общий вид региона в модели DVMH
Listing 2. General form of the parallel region in FDVMH model

```
1 !DVM$ REGION[<specs>]
2 !        Block of statements
3 !DVM$ END REGION
```

Важным дополнением к DVM-системе является система автоматизированного распараллеливания Fortran- и C-программ SAPFOR [15]. С помощью методов статического анализа (т.е. работая только с исходным кодом программы, без его запуска и анализа его выполнения) система анализирует программу и на основе полученных данных осуществляет преобразование ее кода и генерацию параллельной версии через расстановку в коде программы DVMH-директив. В данной статье рассматривается только часть системы SAPFOR, относящаяся к распараллеливанию программ на языке Fortran.

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 невозможным либо полученная параллельная программа оказывается неэффективной (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. Такой вид программы будем называть потенциально параллельным.

Процесс распараллеливания программы с помощью системы SAPFOR является итеративным и состоит из этапов анализа и преобразований. Выполнив тот или иной анализ, пользователь получает набор диагностик и сообщений системы, на основе которых предпринимает дальнейшие шаги по получению 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.

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

В зависимости от полученных результатов пользователь может изменить набор и порядок анализа и преобразований и получить другую версию параллельной программы. Следует отметить, что оптимальной последовательности анализа и преобразований участков кода зачастую не существует даже в рамках одной программы (данный аспект рассмотрен в статье [16]). К одному участку кода могут быть применимы несколько преобразований, дающих разный эффект.

Этапы анализа и преобразований 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, выполненные другим проходом. Для указания зависимостей между проходами 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 выполняет вызванный пользователем проход [17].

3. Используемые понятия. Циклом в языке Fortran называется повторное выполнение блока операторов и конструкций. Конструкции (иначе называемые составными операторами) состоят из нескольких операторов и используются для выполнения управляющих действий, таких как ветвления или циклы. Блок операторов и конструкций цикла называется телом цикла. Однократное выполнение тела цикла называется витком цикла или итерацией. В языке Fortran есть три вида циклов: с параметром, с предусловием и с постусловием [18].

Распараллеливанием цикла является распределение множества его витков между узлами вычислительной системы. Поскольку при распараллеливании с помощью системы SAPFOR для распределения витков между узлами необходимо точно знать число витков цикла, распараллелен может быть только цикл с параметром.

В листинге 3 приведен пример цикла с параметром. Целочисленная переменная `I` называется итерационной переменной цикла или параметром. Целочисленные выражения `A` и `B` задают начальное и конечное значения параметра. Будем называть их границами цикла. Необязательное целочисленное значение `C` задает шаг цикла, который по умолчанию равен единице. Совокупность значений `A`, `B` и `C` назовем заголовком цикла. Также для удобства обозначения будем говорить “цикл по переменной I”, имея в виду цикл, итерационной переменной которого является переменная I.

Листинг 3. Общий вид цикла с параметром в языке Fortran
Listing 3. General form of the DO loop in Fortran

```
1       DO I = A, B, C
2 !        Block of statements
3       ENDDO
```

Тесновложенным называется цикл, состоящий из нескольких последовательно вложенных друг в друга циклов так, что тело каждого цикла (кроме внутреннего) состоит из одного оператора — вложенного в него другого цикла. Тесновложенный цикл является одним из вариантов гнезда циклов. Внешний цикл гнезда назовем находящимся на первом уровне вложенности. Вложенный в него цикл — находящимся на втором уровне вложенности и т.д. Уровень внутреннего цикла определяет максимальный уровень тесной вложенности всего гнезда, иначе называемой степенью тесновложенности цикла. Далее в данной работе рассматриваются только циклы с параметром и не проводится различий между циклом и тесновложенным гнездом циклов, если не указано обратное.

Переменные, использующиеся в теле цикла, который потенциально может быть распараллелен, делятся на приватные и неприватные для данного цикла. Приватной называется переменная, которая на каждом витке цикла получает некоторое значение и это значение используется только на данном витке. В случае массива это означает, что все элементы данного массива получают некоторые значения на каждом витке цикла и эти значения используются только на нем.

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

Для указания системе SAPFOR информации об анализе программы или о необходимости выполнения преобразований пользователь может использовать директивы, которые оформляются в программе в виде специальных комментариев. Общий вид директивы системы SAPFOR представлен в листинге 4. В данном примере `<SPF-directive>` — это тип директивы (один из `ANALYSIS`, `TRANSFORM`, `CHECKPOINT`, `PARALLEL`, `PARALLEL_REG`, `END PARALLEL_REG`), а `<specs>` — список спецификаций, разделенных запятыми.

Листинг 4. Общий вид директивы системы SAPFOR
Listing 4. General form of the SAPFOR's directives

```
!$SPF <SPF-directive>(<specs>)
```

Директива типа `ANALYSIS` сообщает системе дополнительную информацию о том участке кода программы, перед которым она расположена, и используется в сочетании со спецификациями `PRIVATE (<privates>)`, `REDUCTION(<red list>)` и `PARAMETER(<par list>)`. Директива со спецификацией `PRIVATE` может быть расположена перед циклом, в таком случае в списке `<privates>` через запятую указываются переменные, приватные для данного цикла.

Зависимостью по данным называется ситуация, при которой инструкция (оператор) в программе записывает данные в то же место в памяти, в которое записывает или из которого считывает данные другая инструкция [19]. Отдельно отметим, что если две инструкции считывают данные из одного и того же места в памяти, то такая зависимость по данным между инструкциями не препятствует распараллеливанию циклов.

Будем рассматривать выражения, представленные в аффинной форме $a * I + b$, где I — скалярная переменная, а a и b — целочисленные константы. Переменную I будем называть переменной данного аффинного выражения. Если обращение к массиву внутри цикла по переменной I имеет в своем j-м измерении индексное выражение, являющееся аффинным с переменной I, и константа a в данном выражении не равна нулю, то будем говорить, что такое обращение к массиву выровнено с циклом по переменной I по j-му измерению.

4. Представление программы и ее анализ в системе SAPFOR. Абстрактным синтаксическим деревом (или просто синтаксическим деревом) программы называется упорядоченное ориентированное дерево, внутренние узлы которого сопоставлены с операторами и операциями программы, а листья — с их операндами [19]. Синтаксическое дерево представляет собой иерархическую синтаксическую структуру программы. Оно строится синтаксическим анализатором и является удобной структурой для преобразования исходного кода программы. Добавление, удаление, перестановка или изменение операторов программы и операций в них сводятся к соответствующим действиям с узлами синтаксического дерева программы. Анализ программы в таком представлении может быть реализован как обход дерева в глубину или в ширину. Измененная версия программы получается через восстановление ее кода по измененному синтаксическому дереву.

Для представления исходного кода Fortran-программы в виде абстрактного синтаксического дерева в системе SAPFOR используется объектно-ориентированная библиотека Sage++ [20], написанная на языке C++. Узлами синтаксического дерева программы являются объекты классов Sage++, соответствующие операторам программы. Работа с синтаксическим деревом осуществляется через методы классов.

Абстрактное синтаксическое дерево является удобным представлением программы для преобразования ее исходного кода, однако оно слишком сложное для анализа. Поэтому компиляторы и системы анализа кода генерируют некоторое более простое по структуре представление, которое называется промежуточным представлением (Intermediate representation, IR) программы. Его можно рассматривать как программу для абстрактной вычислительной машины. Промежуточное представление характеризуется небольшим набором инструкций схожей структуры и используется для выполнения различного анализа кода программы, оптимизаций и генерации машинного кода в компиляторах. Оно строится путем обхода синтаксического дерева программы и является его линеаризованным представлением [19].

Система SAPFOR использует промежуточное представление программы, называемое трехадресным кодом. Это последовательность инструкций вида

`x = y op z` ,

где `x`, `y` и `z` являются именами в программе, константами или генерируемыми временными переменными, а `op` — это оператор. Название представления происходит от данного общего вида инструкций, в кото-

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

ром используется три операнда. Для работы с массивами используются инструкции `REF`, `LOAD` и `STORE`. Инструкция

```
REF i
```

кладет в стек значение `i`, являющееся индексом, по которому осуществляется обращение к массиву в некотором его измерении. Инструкция

```
LOAD x m k
```

достает из стека `k` значений индексов, по которым соответственно осуществляется обращение к `k`-мерному массиву `m`, и сохраняет значение элемента массива в переменную `x`. Инструкция

```
STORE m k y
```

аналогично инструкции `LOAD` достает из стека `k` значений индексов и присваивает элементу массива `m` значение `y`. Например, для оператора

```
x(i) = y(i, j, k) + 42
```

будет сгенерирован трехадресный код, представленный в листинге 5. В примере видны временные переменные `_reg1` и `_reg2`, в которые сохраняются промежуточные значения, получаемые при выполнении оператора.

Листинг 5. Трехадресный код для оператора `x(i) = y(i, j, k) + 42`
Listing 5. Three-address code for the operator `x(i) = y(i, j, k) + 42`

```
1 REF i
2 REF j
3 REF k
4 LOAD _reg1 y 3
5 _reg2 = _reg1 + 42
6 REF i
7 STORE x 1 _reg2
```

Трехадресные инструкции выполняются последовательно, если иное не требуется инструкциями условного или безусловного перехода. Инструкции условного перехода имеют вид

```
IF_FALSE x then goto n
```

и осуществляют передачу потока управления к инструкции с номером `n` (каждая инструкция трехадресного кода имеет уникальный номер), если значение `x` ложно. Инструкции безусловного перехода имеют вид

```
GOTO n
```

и осуществляют передачу потока управления к инструкции с номером `n`.

После генерации трехадресный код представляется в виде ориентированного графа, узлами которого являются блоки кода, называемые базовыми блоками. Базовый блок (basic block, BB) — это максимальная последовательность следующих друг за другом инструкций, обладающая следующими свойствами:

- поток управления может входить в базовый блок только через первую инструкцию блока, т.е. переходы внутрь блока отсутствуют;
- управление покидает блок без останова или ветвления, за исключением, возможно, последней инструкции блока.

Если поток управления после последней инструкции базового блока B_1 может быть передан на первую инструкцию базового блока B_2, то вершины графа, соответствующие блокам B_1 и B_2, соединяются ориентированным ребром. Имеются две ситуации, когда может существовать такое ребро: если существует условный или безусловный переход от конца блока B_1 к началу блока B_2 или если B_2 следует непосредственно за B_1 в исходном порядке трехадресных инструкций, а блок B_1 не заканчивается безусловным переходом. К построенному графу обычно добавляются два фиктивных блока, называемые входом и выходом. Они не соответствуют каким-либо исполняемым инструкциям программы, но оказываются удобными при выполнении анализа кода по данному графу. Вход соединяется ребром с первым выполняемым узлом графа, содержащим первую инструкцию трехадресного кода. С выходом соединяется блок, содержащий последнюю инструкцию программы, и блоки, оканчивающиеся инструкцией останова. Построенный граф называется графом потока управления программы (control flow graph, CFG).

Граф потока управления используется для выполнения различного анализа кода программы, называемого анализом потоков данных. Это методы, собирающие информацию о потоках данных вдоль путей

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

выполнения программы [19, раздел 9.2]. Например, они используются для поиска общих подвыражений (разных инструкций, вычисляющих одинаковые значения) или мертвого кода (инструкций, результат вычисления которых в программе не используется).

Выполнение программы можно рассматривать как ряд преобразований состояния программы, каждое из которых характеризуется множеством значений всех переменных. Каждое выполнение инструкции промежуточного кода переводит программу в новое выходное состояние. Входное состояние связано с точкой программы перед инструкцией, выходное — с точкой программы после инструкции. Путем выполнения от точки p_1 до точки p_n называется последовательность точек программы $p_1, p_2, \ldots, p_n$ таких, что для каждого $i = 1, \ldots, n-1$ выполнено одно из двух условий:

- p_i и p_{i+1} являются точками, непосредственно предшествующей и следующей за инструкцией в базовом блоке;
- p_i является концом некоторого базового блока B_1 (следует за последней инструкцией B_1), а p_{i+1} является началом базового блока B_2 (находится перед первой инструкцией B_2), и B_1 соединен с B_2 ребром в графе потока управления.

Отметим, что если граф потока управления содержит циклы, то различных путей выполнения между двумя точками программы может быть неограниченно много.

При анализе потока данных с каждой точкой программы связывается значение потока данных, представляющее собой абстракцию множества всех возможных состояний программы, которые могут быть в данной точке. Выбирается передаточная функция, которая определяет, как каждая инструкция программы меняет значение потока данных. На основе передаточной функции для отдельной инструкции выводится передаточная функция для всего базового блока. Задачей анализа потока данных является определение его значения во всех точках программы (или только в точках начала и конца базовых блоков) по заданным начальным значениям. Важным примером анализа потока данных является анализ достигающих определений.

Определением переменной a называется инструкция, которая присваивает или может присваивать значение этой переменной. При этом говорят, что инструкция порождает определение переменной. Определение d переменной a достигает некоторой точки p в программе, если существует путь s от точки, следующей за d, к точке p, вдоль которого определение d не уничтожается. Определение d будет уничтожено на пути s, если на s существует иное определение d' переменной a. В этом случае говорят, что определение d' уничтожает определение d [19].

Анализ достигающих определений позволяет для каждого базового блока получить множество определений, достигающих его (т.е. достигающих точки перед первой инструкцией блока) и покидающих его (т.е. достигающих точки после последней инструкции блока). Отметим несколько важных особенностей анализа достигающих определений:

- Базовый блок может достигать несколько определений одной переменной. Например, когда поток управления может прийти в данный блок из нескольких базовых блоков, каждый из которых порождает свое определение переменной.
- Неинициализированные переменные. В программе возможна ситуация, когда инструкция читает значение (возможно) неинициализированной переменной. Чтобы отличать такое обращение к переменной, вводится специальное определение для всех переменных, порождаемое входным базовым блоком графа потока управления. В системе SAPFOR такое определение задается константой `UNINIT`. Оно отличается от остальных определений только тем, что не порождается конкретной инструкцией программы и может, как и остальные определения, распространяться по графу потока управления программы или уничтожаться инструкциями в базовых блоках.
- Анализ достигающих определений работает только для скалярных переменных. Поскольку анализ полагается на то, что каждая инструкция программы обращается на чтение или запись к переменной, уничтожая и порождая при этом ее определения, он оказывается неспособным различить обращения к разным элементам одной переменной-массива. Для определения, к какому элементу массива идет обращение в инструкции, требуется знать возможные значения индексных выражений в этом обращении. Использование же определений переменных из этих выражений практически бесполезно, поскольку в другом месте программы индексные выражения при обращении к этому же массиву могут содержать другие переменные с другими определениями. И даже если в обоих местах программы идет обращение к одному и тому же элементу массива, анализ достигающих определе-

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

ний оказывается бессилен это выявить и может предоставить лишь общую информацию о том, какая инструкция обращается на чтение или запись в массив.

5. Реализация преобразования удаления приватных переменных в системе SAPFOR. В данном разделе описывается алгоритм преобразования последовательных Fortran-программ "удаление приватных переменных". Главное требование, которому должно удовлетворять преобразование — корректность, т.е. исходная и преобразованная программы должны быть эквивалентны. Эквивалентными будем называть программы, которые имеют равные выходные данные при одинаковых входных данных.

Поскольку изначальной целью преобразования удаления приватных переменных является приведение цикла к потенциально параллельному виду последовательной программы, то реализация данного преобразования рассматривается только для потенциально параллельных циклов, т.е. таких циклов с параметром (возможно тесновложенных), которые могут быть преобразованы в параллельные циклы системой SAPFOR с помощью директивы распределения вычислений `PARALLEL` [17]. Такие циклы должны обладать следующими дополнительными свойствами:

- иметь ненулевое количество витков;
- иметь прямоугольное индексное пространство (это означает, что границы всех уровней гнезда циклов могут быть вычислены до начала выполнения цикла и они не меняются во время выполнения цикла);
- не содержать операторов перехода `GOTO` внутрь цикла и за его пределы;
- не содержать операторов ввода/вывода и останова;
- измерения используемых в цикле массивов индексируются только аффинными выражениями;
- между витками цикла могут быть только зависимости по редукционным переменным, а также аффинные зависимости по распределенному массиву;
- не содержать вызовов процедур и функций, внутри которых находятся другие потенциально параллельные циклы.

Отметим, что потенциально параллельный цикл может содержать вызовы процедур и функций в своем теле. Однако, для удовлетворения описанным выше свойствам, данные процедуры и функции не должны содержать потенциально параллельных циклов и операторов ввода-вывода и останова. Проверка выполнения данного условия будет осуществляться межпроцедурным анализом.

5.1. Общая схема преобразования. Удалением приватной переменной цикла называется такое его преобразование, при котором выражения, стоящие в правых частях определений-присваиваний в переменную в теле цикла, подставляются на места использования этих определений, т.е. на места обращений на чтение к данной переменной в теле цикла. При этом сами операторы присваивания в переменную удаляются из цикла, если после подстановки выражений они становятся мертвым кодом. Поскольку преобразование заключается в подстановке выражений из операторов присваивания в переменную, на цикл налагается условие, что других операторов, меняющих значение этой переменной, в теле цикла нет, в том числе нет операторов вызова процедур или функций, которые меняют удаляемую переменную через свои OUT-параметры. Неявное использование массива через common-блоки процедурами, вызываемыми из преобразуемого цикла, также не допускается. Здесь и далее для удобства под определением удаляемой переменной в зависимости от контекста будем понимать или выражение, стоящее в правой части присваивания в переменную, или определение переменной в терминах достигающих определений. Все вхождения переменной в операторы тела цикла будем называть обращениями к переменной, разделяя их на обращения на запись и обращения на чтение. Оператором DEF будем называть любой оператор присваивания в удаляемую переменную в теле цикла, оператором USE — любой оператор, в котором есть обращение на чтение из удаляемой переменной. Выражением оператора DEF будем называть выражение, стоящее справа от знака равно в операторе DEF.

Преобразование удаления приватной переменной можно разделить на четыре основных фазы. На первой требуется построить множество пар операторов DEF–USE, т.е. операторов, присваивающих некоторое значение в удаляемую переменную и использующих его. На второй фазе необходимо для каждой пары операторов DEF–USE проверить, возможно ли подставить выражение оператора DEF в место обращения на чтение из удаляемой переменной в операторе USE, чтобы такая подстановка была корректной. На третьей фазе непосредственно выполняется подстановка выражений оператора DEF в оператор USE для каждой пары операторов, для которой эта подстановка будет корректной. На заключительной четвертой фазе выполняется удаление мертвого кода, который мог образоваться в результате преобразований третьей фазы.

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

Заметим, что в случае, когда приватная переменная цикла является скалярной, реализация данного преобразования относительно проста. Множество пар операторов DEF–USE строится с помощью анализа достигающих определений и последующего обхода графа потока управления с целью выявления операторов, использующих определения данной переменной. Если такой оператор USE достигает единственное определение, то соответствующий ему оператор DEF найден. Для проверки возможности подставить выражение оператора DEF в оператор USE необходимо проверить, что для каждой переменной из выражения оператора DEF все ее определения, достигающие оператор DEF, достигают оператор USE и при этом оператор USE не достигают никакие другие определения данной переменной. Для скалярных переменных это также можно сделать по результатам анализа достигающих определений.

Для массивов в общем случае это уже неразрешимая задача (об анализе достигающих определений для массива подробнее будет сказано ниже), но можно сделать упрощение в виде проверки, что на всех путях выполнения от оператора DEF до оператора USE нет присваиваний в данный массив. Также нельзя выполнять подстановку, если оператор DEF или USE является рекурсивным определением переменной (т.е. когда в выражении справа от знака равно есть использование определяемой переменной, иначе говоря, оператор DEF одновременно является и оператором USE и наоборот). Удаление такой переменной будет некорректным, поскольку она не является приватной для цикла, в теле которого находится ее рекурсивное определение, и при этом нет инициализации данной переменной (т.е. оператора DEF, который бы уничтожал все достигающие его определения данной переменной, не используя их). Такое может быть, если внутри внешнего преобразуемого цикла есть не тесновложенный цикл, содержащий рекурсивное определение некоторой скалярной переменной, являющейся приватной для внешнего цикла. Подстановка выражений оператора DEF в оператор USE для построенных пар операторов не требует преобразования подставляемых выражений, поэтому является тривиальной. Удаление мертвого кода для присваиваний в скалярные переменные также является стандартным алгоритмом, реализуемым через анализ потока данных под названием “анализ живых переменных” и схожим с анализом достигающих определений.

Описанные преобразования уже реализованы в системе SAPFOR под названиями “подстановка выражений” и “удаление мертвого кода”, первое из которых осуществляет построение множества пар операторов DEF–USE для скалярных переменных и подстановку выражений для них (причем не только для приватных переменных циклов, но и вообще для тела всей функции), а второе — удаляет мертвый код (неиспользуемые определения скалярных переменных, а также некоторые другие бесполезные операторы, например циклы с пустым телом). Поэтому в данной статье преобразование удаления приватных переменных рассматривается только для переменных-массивов. К тому же, при использовании системы SAPFOR для распараллеливания различных программ именно удаление приватных массивов представляет основной интерес и возможность для преобразования цикла к потенциально параллельному виду. Далее в статье под удаляемой приватной переменной цикла всегда подразумевается массив.

5.2. Построение множества пар операторов DEF–USE. Как было отмечено в разделе 4, анализ достигающих определений работает только для скалярных переменных, поэтому в исходном виде применить его для анализа достижения определений элементов массива нельзя. В общем случае задача соотнесения определений элементов массива и мест их использования в коде является неразрешимой, поскольку требует знания значений индексных выражений в обращении к массиву для определения, к какому элементу массива идет обращение, что не всегда можно сделать статическим анализом кода. Поэтому предложенное решение данной задачи имеет определенные ограничения, которые будут рассмотрены далее. Предложенное решение для построения множества пар DEF–USE операторов удаляемого массива основывается на введении множества временных скалярных переменных, соотнесении их с элементами (или множествами элементов) массива и выполнении анализа достигающих определений для введенных переменных. Рассмотрим его реализацию на примере цикла из листинга 6.

Введем понятие маски фиксированных измерений массива — булевого вектора длины, совпадающей с размерностью переменной (количеством измерений массива). Каждому обращению к переменной сопоставляется маска фиксированных измерений, i-й элемент вектора которой равен $true$ тогда и только тогда, когда индексное выражение в i-м измерении обращения к переменной является целочисленной константой (для удобства здесь и далее будем нумеровать измерения массива так, как они указываются в определении массива и обращении к нему — слева направо, начиная с единицы). Например, для всех обращений к приватной переменной `A` в примере из листинга 6 маской фиксированных измерений будет вектор $< true, false >$, потому что во всех обращениях к переменной в первом измерении в качестве ин-

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

Листинг 6. Пример цикла для иллюстрации понятия маски фиксированных измерений массива `A`
Listing 6. An example of a loop to illustrate the concept of a fixed-dimension mask for array `A`

```
1  !$SPF ANALYSIS(PRIVATE(A))
2        DO I = 1, N
3           DO J1 = 1, M
4              A(1, J1) = I + J1
5              A(2, J1) = I * J1
6           ENDDO
7           DO J2 = 2, M - 1
8              B(1, J2, I) = A(1, J2 - 1) * I
9              B(2, J2, I) = A(2, J2 + 1) * I
10          ENDDO
11       ENDDO
```

дексного выражения используется целочисленная константа `1` или `2`, а во втором измерении — выражения `J1`, `J2 - 1` или `J2 + 1`.

На основе всех обращений к переменной в цикле строится минимальная маска фиксированных измерений как поэлементное логическое "И" векторов масок фиксированных измерений. Таким образом, если i-й элемент вектора минимальной маски фиксированных измерений равен $true$, то во всех обращениях к переменной в цикле индексное выражение в i-м измерении является целочисленной константой. Назовем данное измерение переменной "фиксированным". Отметим, что фиксированных измерений у массива может не быть вообще.

Наличие фиксированного измерения позволяет "разделить" исходную переменную на несколько различных переменных, каждой из которых соответствует своя константа в фиксированном измерении удаляемой переменной. Введем понятие вектора фиксированных индексов обращения к переменной. Это целочисленный вектор, размерность которого равна количеству элементов `true` в минимальной маске фиксированных измерений и значения элементов которого равны целочисленным константам, стоящим в индексных выражениях фиксированных измерений в данном обращении к массиву. Например, векторы фиксированных индексов обращений к переменной `A` в строках 4 и 5 в листинге 6 равны $< 1 >$ и $< 2 >$ соответственно. Векторы фиксированных индексов обращений к переменной `A` в строках 8 и 9 также будут равны $< 1 >$ и $< 2 >$, что является основой для построения соответствия между определениями элементов массива и местами их использования в коде.

"Разделение" массива на несколько скалярных переменных через отождествление обращений к массиву с одинаковыми векторами фиксированных индексов позволяет использовать анализ достигающих определений для построения соответствия между определениями переменной и местами их использования. Для этого тело цикла преобразуется следующим образом. Строится множество всех различных векторов фиксированных индексов для удаляемой переменной (для переменной `A` из листинга 6 это множество равно $\{< 1 >, < 2 >\}$). Для каждого вектора из построенного множества создается соответствующая ему скалярная переменная. Векторам $< 1 >$ и $< 2 >$ для переменной-массива `A` будут сопоставлены скалярные переменные `A_1` и `A_2` соответственно. Это позволяет каждому обращению к удаляемому массиву сопоставить скалярную переменную.

Конечно, отождествление двух обращений к массиву, если они совпадают по вектору фиксированных индексов, в общем случае неверно, ведь они могут различаться по значениям индексов из нефиксированных измерений. Однако, как было отмечено выше, в общем случае данная задача является неразрешимой для статического анализа. Чтобы уточнить множество циклов, к которым данный подход применим, наложим на удаляемую приватную переменную цикла следующее ограничение: индексные выражения нефиксированных измерений во всех обращениях на запись к удаляемому массиву должны состоять из одной итерационной переменной какого-либо цикла, вложенного в преобразуемый. Индексные выражения разных обращений к массиву могут содержать разные итерационные переменные.

Далее перед каждым оператором DEF (т.е. обращением на запись в удаляемую переменную) ставится оператор присваивания в соответствующую этому обращению скалярную переменную. Например, перед присваиванием

```
A(1, J1) = I + J1
```

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

Листинг 7. Промежуточное преобразование цикла
Listing 7. Intermediate loop transformation

```
1  !$SPF ANALYSIS(PRIVATE(A))
2        DO I = 1, N
3           DO J1 = 1, M
4              A_1 = 0
5              A(1, J1) = I + J1
6              A_2 = 0
7              A(2, J1) = I * J1
8           ENDDO
9           DO J2 = 2, M - 1
10             TMP = A_1
11             B(1, J2, I) = A(1, J2 - 1) * I
12             TMP = A_2
13             B(2, J2, I) = A(2, J2 + 1) * I
14          ENDDO
15       ENDDO
```

в строке 4 в листинге 6 будет вставлено присваивание

```
A_1 = 0 .
```

Перед каждым оператором, в котором есть обращение на чтение из удаляемой переменной, ставится оператор присваивания из соответствующей этому обращению скалярной переменной в созданную временную переменную `TMP` (т.е. ставится оператор, использующий значение введенной скалярной переменной). Например, перед оператором

```
B(1, J2, I) = A(1, J2 - 1) * I
```

в строке 8 будет вставлено присваивание

```
TMP = A_1 .
```

При этом в памяти сохраняется соответствие между вставленными операторами присваивания и исходными операторами цикла, к которым они относятся. Для цикла из листинга 6 результат расстановки дополнительных присваиваний представлен в листинге 7.

После расстановки дополнительных операторов присваивания во временные переменные выполняется построение графа потока управления и анализ достигающих определений для текущей функции. При этом анализ проводится в предположении, что все циклы имеют ненулевое количество итераций, что соответствует требованию к потенциально параллельному циклу, описанному в начале данного раздела. Такое предположение существенно улучшает точность анализа достигающих определений, исключая “лишние” определения, которые достигают базовых блоков. Например, базовый блок, соответствующий телу цикла по переменной `J2` в листинге 7, при каноническом анализе достигают не менее двух определений переменной `A_1`: определение из строки 4 и те определения, которые достигли входа в цикл по переменной `I` (например, определение неинициализированной переменной UNINIT). Если же анализ проводится в предположении, что каждый цикл выполняется как минимум одну итерацию, то все определения, достигшие входа в цикл по переменной `I`, уничтожаются в теле цикла по переменной `J1` и цикла по переменной `J2` достигает только одно определение `A_1` из строки 4.

По результатам анализа достигающих определений строится множество пар операторов DEF–USE для удаляемой переменной. Для каждого расставленного оператора присваивания OP_{USE} вида `TMP = A_K`, соответствующего некоторому оператору USE, находится множество S_{DEF} определений переменной `A_K`, которые достигают базового блока с оператором OP_{USE}. Если количество определений во множестве S_{DEF} отлично от единицы, это означает, что оператор OP_{USE} достигают более одного определения переменной `A_K` и выполнить подстановку удаляемой переменной в оператор USE, соответствующий оператору OP_{USE}, невозможно. Если же во множестве S_{DEF} осталось единственное определение переменной `A_K`, неравное UNINIT, то оно однозначно определяет оператор вида `A_K = 0`, которому соответствует исходный оператор цикла DEF. Таким образом определяется пара DEF–USE исходных операторов цикла, в

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

первом из которых происходит обращение на запись в удаляемую переменную, а во втором — обращение на чтение, причем оба обращения соответствуют одному вектору фиксированных индексов. После построения множества пар операторов DEF–USE расставленные операторы присваивания во временные скалярные переменные можно удалить.

Листинг 8. Пример цикла с рекурсивным определением приватной переменной `A`
Listing 8. Example of a loop with a recursive definition of a private variable `A`

```
1  !$SPF ANALYSIS(PRIVATE(A))
2        DO I = 1, N
3           DO J1 = 1, M
4              A_ = 0
5              A(J1) = J1
6           ENDDO
7           DO J2 = 1, M
8              A_ = 0
9              TMP = A_
10             A(J2) = A(J2) + J2
11          ENDDO
12          DO J3 = 1, M
13             TMP = A_
14             B(J3, I) = A(J3)
15          ENDDO
16          C(I) = A(2)
17       ENDDO
```

Предложенный алгоритм позволяет корректно соотнести операторы DEF и USE для удаляемой переменной, однако он слишком консервативен и не позволяет решить данную задачу для следующего встречаемого на практике случая, проиллюстрированного в листинге 8 (уже после расстановки операторов присваивания во временные скалярные переменные `A_` и `TMP`). В этом примере определение массива `A` в цикле по переменной `J2` является рекурсивным, т.е. каждый элемент массива определяется на основе своего предыдущего значения. Такой оператор DEF является одновременно и оператором USE, причем векторы фиксированных индексов обращений на чтение и запись из массива `A` совпадают (в данном случае они пустые, поскольку вектор фиксированных измерений равен $< false >$). Анализ достигающих определений покажет, что оператор чтения из временной переменной `A_` на строке 9 достигают два определения переменной, порожденные на строках 4 и 8. Однако подстановка выражения оператора DEF на строке 5 в оператор USE на строке 10 корректна, поскольку и оба цикла по переменным `J1` и `J2` присваивают значение в каждый элемент массива `A` лишь единожды.

Чтобы исключить данную ситуацию, когда анализ достигающих определений не позволяет построить DEF–USE пары для рекурсивных определений, нужно на этапе построения множества определений S_{DEF} исключить из него определение, относящееся к этому же оператору OP_{USE}. Тогда в рассматриваемом примере оператор на строке 9 будет достигать единственное определение, порожденное оператором на строке 4, что позволит корректно построить пару DEF–USE операторов на строках 5 и 10 соответственно.

Однако дальнейшая подстановка выражения оператора DEF на строке 10 в оператор USE на строке 14 может быть выполнена только после подстановки в паре операторов на строках 5 и 10. Поэтому если для оператора USE соответствующий ему оператор DEF является рекурсивным определением, данная пара операторов DEF–USE отбрасывается. Заметим, что после выполнения подстановки операторов в паре на строках 5 и 10 оператор DEF на строке 10 перестанет быть рекурсивным, поэтому подстановка его выражения в оператор USE на строке 14 будет возможна. Таким образом, удаление переменных, у которых есть рекурсивные определения, может осуществляться последовательным выполнением алгоритма несколько раз.

Исключение рекурсивного определения из множества достигающих определений корректно, поскольку оба определения порождаются циклами, которые присваивают значение каждому элементу удаляемого массива единожды и находятся на одном уровне вложенности в преобразуемый цикл. Первое условие выполняется потому, что обращения к удаляемому массиву в каждом из вложенных циклов выровнены с

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

ним, т.е. содержат в индексном выражении первого (и единственного) измерения итерационную переменную данных циклов. Это накладывает дополнительное требование к преобразуемому циклу — все обращения к удаляемой переменной в теле цикла должны иметь одинаковое выравнивание с вложенными в него циклами. Таким образом, для каждого нефиксированного измерения все обращения к удаляемому массиву должны быть выровнены по данному измерению с циклами, находящимися на одинаковом уровне вложенности, либо индексное выражение данного измерения не должно содержать итерационных переменных.

Листинг 9. Пример цикла для иллюстрации понятия рекурсивного определения массива `A`
Listing 9. An example of a loop to illustrate the concept of recursive definition of array `A`

```
1  !$SPF ANALYSIS(PRIVATE(A))
2        DO I = 1, N
3           DO J1 = 1, M
4              A(J1) = 0
5           ENDDO
6           DO K = 1, 10
7              DO J2 = 1, M
8                 A(J2) = A(J2) + J2
9              ENDDO
10             DO J3 = 1, M
11                B(J3, K, I) = A(J3) * K
12             ENDDO
13          ENDDO
14       ENDDO
```

Рассмотрим данное требование на примере цикла из листинга 9, в котором оно не выполняется. Обращение к переменной `A` на строке 4 выровнено по первому измерению с циклом по переменной `J1`, находящимся на втором уровне вложенности в цикл по переменной `I`. Обращения же к переменной `A` на строках 8 и 11 выровнены по первому измерению с циклами по переменным `J2` и `J3` соответственно, находящимися на третьем уровне вложенности. Разное выравнивание обращений к переменной с циклами не позволяет выполнить ее удаление в данном примере. Отметим, что в цикле из листинга 8 все обращения (кроме находящегося на строке 16) к переменной `A` выровнены по первому измерению с циклами, находящимися на втором уровне вложенности. Обращение к переменной `A` на строке 16 в индексном выражении первого измерения не имеет итерационных переменных, поэтому выравнивание его с каким-либо циклом не требуется и подстановка в него выражения из оператора на строке 10 будет корректной.

5.3. Проверка корректности подстановки пары операторов DEF–USE. После построения множества пар операторов DEF–USE для каждой пары требуется выполнить проверку, возможно ли корректно подставить выражение из оператора DEF в оператор USE. Это возможно только если определения всех переменных из выражения оператора DEF (кроме определений итерационных переменных) достигают оператор USE. Проверка данного свойства является нетривиальной, поскольку в выражении оператора DEF могут быть как скалярные переменные, так и массивы. Итерационные переменные исключаются из рассмотрения, потому что при подстановке выражения оператора DEF они будут заменены на индексные выражения из обращения к массиву в операторе USE, о чем будет подробнее сказано в разделе 5.4.

Для проверки того факта, что все определения скалярных переменных (кроме итерационных) из выражения оператора DEF достигают оператора USE, используется анализ достигающих определений, который был проведен для текущей функции на предыдущей фазе преобразования. Выбираются множества достигающих определений RD_{DEF} и RD_{USE} базовых блоков, в которых находятся операторы DEF и USE соответственно. Далее для каждой скалярной переменной V из выражения DEF проверяется, что во множествах RD_{DEF} и RD_{USE} содержится строго по одному определению переменной V и определения V во множествах RD_{DEF} и RD_{USE} совпадают.

Для проверки, что все определения массивов из выражения оператора DEF достигают оператор USE, можно не выполнять анализ достигающих определений, который был проведен для удаляемого массива, а ограничиться проверкой присваиваний в массивы на пути выполнения S от оператора DEF до оператора USE. Операторы пары DEF и USE находятся внутри преобразуемого цикла (причем оператор

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

DEF находится в коде выше оператора USE, иначе удаляемый массив не был бы приватным), однако каждый из них может находиться внутри отдельного цикла, вложенного в преобразуемый и не содержащего второй оператор пары. Тогда путей выполнения от оператора DEF до оператора USE в общем случае неограниченно много, но достаточно просмотреть лишь один, включающий в себя все операторы данного вложенного цикла. Назовем цикл охватывающим оператор DEF (USE), если он содержит в себе оператор DEF (USE) и не содержит оператор USE (DEF). Максимальным из охватывающих циклов оператора DEF (USE) назовем такой охватывающий цикл, который является охватывающим и не содержится в другом цикле, охватывающем данный оператор. Аналогично минимальный охватывающий цикл — это цикл, который не содержит в себе другого охватывающего цикла для данного оператора.

Для построения пути выполнения S, включающего в себя все операторы между операторами DEF и USE, а также все операторы из циклов, максимальных из охватывающих их (если они существуют), используется следующий алгоритм. Множество циклов, вложенных в преобразуемый цикл, образует дерево, корнем которого является сам преобразуемый цикл. Минимальные охватывающие циклы для операторов DEF и USE, если они существуют, являются листьями этого дерева. Тогда задача поиска максимальных охватывающих циклов операторов DEF и USE сводится к задаче поиска ближайшего общего предка для двух листьев дерева. Потомки такой вершины, содержащие операторы DEF и USE, и будут максимальными из охватывающих данные операторы циклов. Алгоритмы решения этой задачи являются общеизвестными и не представляют интереса в настоящей работе, поэтому их описание не приводится. Проверка существования охватывающего цикла для оператора DEF или USE также является тривиальной.

Отметим, что путь S представляет собой сплошной участок кода программы, поскольку, как было сказано выше, оператор DEF стоит в коде выше оператора USE и путь S включает в себя все операторы между DEF и USE и операторы максимальных из охватывающих их циклов, если такие существуют. Поэтому при анализе на уровне синтаксического дерева путь S описывается двумя указателями — на первый и на последний оператор пути.

Далее из выражения оператора DEF и обращения к удаляемому массиву в операторе USE конструируется выражение E, которое непосредственно будет подставлено в оператор USE (если проверка возможности подстановки даст положительный результат). Подробно конструирование этого выражения будет описано в разделе 5.4, здесь лишь отметим, что в нем происходит замена итерационных переменных на индексные выражения из обращения к удаляемому массиву в операторе USE. Далее для каждого обращения R к какому-либо массиву M из выражения E необходимо проверить, что на всем пути S нет присваиваний в элементы данного массива, которые используются в обращении R (отметим, что в выражении E нет обращений к удаляемому массиву, поскольку по построению пар операторов DEF–USE оператор DEF не может быть рекурсивным). Для этого требуется использовать некоторый метод, позволяющий определить, относятся ли два обращения к одному массиву M к разным его элементам, даже если в индексных выражениях используются итерационные переменные циклов.

Сопоставим каждому индексному выражению V из обращения к массиву M пару из двух аффинных выражений, задающих начало и конец диапазона возможных значений выражения V. По условию потенциальной параллельности преобразуемого цикла выражение V является аффинным и имеет общий вид $a*I+b$. Если оно является целочисленной константой (т.е. константа a равна нулю) или переменная I в нем не является итерационной, то оба аффинных выражения диапазона одинаковы и равны V. Если же переменная I является итерационной, то внутри цикла она меняется в диапазоне от $start$ до end — нижней и верхней границ цикла соответственно (отметим, что они могут быть не только целочисленными константами). Тогда пара аффинных выражений диапазона возможных значений V задается выражениями $a*start+b$ и $a*end+b$.

Таким образом, задача определения, относятся ли два обращения к массиву к разным его элементам, сводится к задаче определения, пересекаются ли построенные диапазоны значений индексных выражений из данных обращений в каждом измерении. Если удается показать, что диапазоны не пересекаются хотя бы в одном из измерений, то данные обращения к массиву относятся к разным его элементам, и поэтому выражение оператора DEF может быть корректно подставлено в оператор USE. Если же показать это не удается, то для консервативности алгоритма такие обращения к массиву считаются потенциально обращающимися к одним и тем же элементам массива и подстановка для данной пары операторов DEF–USE запрещается. В общем случае такая задача достаточно сложна, поскольку переменные, используемые в индексных выражениях обращений к массиву и границах цикла, могут зависеть друг от друга и/или определяться в других функциях программы. Для точного и полного анализа требуется межпроцедурный

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

анализ достигающих определений и интерпретатор выражений из этих определений либо динамический анализ.

5.4. Подстановка выражений и удаление мертвого кода. После проверки того факта, что для некоторой пары операторов DEF–USE подстановка выражения оператора DEF в оператор USE корректна, осуществляется конструирование выражения E и подстановка его на место обращения к удаляемому массиву в операторе USE. Рассмотрим обращение REF_{w} на запись в удаляемый массив в операторе DEF и обращение REF_{r} на чтение из него в операторе USE. Строится отображение V_j -> U_j, где V_j — индексное выражение j-го нефиксированного измерения обращения REF_{w}, а U_j — индексное выражение j-го измерения обращения REF_{r}. По условиям, наложенным ранее на цикл, индексные выражения в нефиксированных измерениях обращения REF_{w} являются итерационными переменными циклов, в которые вложен оператор DEF, поэтому выражение V_j является переменной. Выражение E конструируется путем замены в выражении оператора DEF переменной V_j на выражение U_j для каждого j-го нефиксированного измерения, в соответствии с построенным отображением. Как было сказано ранее, данное преобразование можно представить как подстановку функции, телом которой является выражение оператора DEF, а параметрами — переменные V_j.

После выполнения подстановок выражений операторов DEF, некоторые (а возможно, и все) из генерируемых ими определений удаляемой переменной становятся мертвым кодом, который следует удалить из цикла. Для этого строится множество S векторов фиксированных индексов для всех обращений на чтение к удаляемому массиву в операторах USE. Далее проверяются все обращения на запись в удаляемый массив в операторах DEF. Если вектор фиксированных индексов такого обращения отсутствует во множестве S, то определение, генерируемое данным оператором, не используется в цикле, поэтому данный оператор может быть удален как мертвый код. Тем самым преобразование удаления приватных переменных является выполненным.

5.5. Удаление массива с аффинными индексными выражениями в операторе DEF. Рассмотренный выше алгоритм работает только в том случае, если индексные выражения нефиксированных измерений в обращениях к удаляемому массиву в операторах DEF являются итерационными переменными какого-либо цикла, вложенного в преобразуемый. Однако на практике применения реализованного преобразования с помощью системы SAPFOR встретился случай, когда индексные выражения нефиксированных измерений удаляемого массива в операторах DEF являются аффинными выражениями вида $a * I + b$, в которых коэффициент a при итерационной переменной цикла I не равен нулю и единице и коэффициент b (свободный член) не равен нулю. При этом фиксированных измерений у массива нет и единственный способ корректно построить пары операторов DEF и USE — по индексным выражениям аффинного вида. Пример такого цикла приведен в листинге 10.

Листинг 10. Пример цикла с аффинными индексными выражениями
Listing 10. Example of a loop with regular index expressions

```
1 !$SPF ANALYSIS(PRIVATE(A))
2       DO I = 1, N
3          A(2*I - 1) = I
4          A(2*I) = I * I
5          B(I) = A(2*I - 1) + A(2*I)
6       ENDDO
```

Отметим, что, строго говоря, массив `A` не является приватным для цикла в данном ранее каноническом определении приватной переменной, поскольку не все его элементы получают свое значение на каждой итерации цикла. Однако значения, которые получает часть его элементов (два элемента в нашем примере) на каждой итерации цикла, используются только на данной итерации, поэтому отнесем данную переменную к приватной для цикла. Несмотря на то что индексные выражения обращений к массиву в операторах DEF не соответствуют ограничениям, введенным в разделе 5.2, массив может быть удален. Для этого можно применить описанный ранее алгоритм, дополнив его следующими изменениями.

Требования к индексным выражениям нефиксированных измерений удаляемого массива можно ослабить так, чтобы коэффициент a при итерационной переменной I был отличен от нуля. Далее, в исходном алгоритме временные скалярные переменные вводились в соответствии со множеством векторов

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

фиксированных индексов. В нашем примере вектором фиксированных индексов можно назвать вектор из всех коэффициентов всех аффинных индексных выражений. Поэтому множество временных переменных строится по всем различным коэффициентам индексных выражений. Для данного примера векторами фиксированных индексов являются $< 2, -1 >$ и $< 2, 0 >$, поэтому могут быть введены следующие переменные: `A_2_M1` и `A_2_0`. Буква `M` в названии переменной означает знак минус перед соответствующим коэффициентом, поскольку в языке Fortran не позволяется использовать символ знака минус (дефис) в названиях переменных.

Также на этапе конструирования выражения для подстановки в оператор USE в исходном алгоритме строилось отображение V_j -> U_j, где V_j — индексное выражение j-го нефиксированного измерения обращения к удаляемому массиву на запись, а U_j — индексное выражение j-го измерения обращения на чтение. В случае с аффинными индексными выражениями данное отображение должно строиться так, что V_j и U_j — это итерационные переменные из индексных выражений нефиксированных измерений обращений на запись и на чтение соответственно.

Описанные изменения в исходном алгоритме позволяют применить его для удаления приватной переменной из рассматриваемого примера.

6. Исследование реализованного преобразования. В данном разделе рассмотрим применение реализованного преобразования удаления приватных переменных в процессе автоматизированного распараллеливания программ LU, BT, SP и EP из пакета тестов NAS Parallel Benchmarks 3.3 в системе SAPFOR [21].

6.1. Преобразование программы LU. Программа LU решает систему линейных уравнений итерационным методом Гаусса–Зейделя. В основном файле программы вызывается процедура SSOR, которая содержит основной итерационный цикл программы, откуда вызывается процедура RHS, выполняющая примерно половину всех вычислений в программе. Процедура RHS состоит из трех схожих по структуре циклов, упрощенный пример первого из которых приведен в листинге 11. В этом цикле на основе массивов `U` и `RHO_I` вычисляются значения приватного массива `FLUX`, которые используются для вычисления значения результирующего массива `RSD`. Цикл может быть распараллелен, поскольку витки циклов по переменным `K` и `J` можно выполнять независимо ввиду отсутствия зависимостей по данным.

Листинг 11. Упрощенный вид первого цикла из процедуры RHS программы LU
Listing 11. Simplified view of the first loop from the RHS procedure of the LU program

```
!$SPF ANALYSIS(PRIVATE(FLUX))
      DO K = 2, NZ - 1
         DO J = JST, JEND
            DO I = 1, NX
               FLUX(1,I) = U(2,I,J,K)
               FLUX(2,I) = U(2,I,J,K) * RHO_I(I,J,K) * C1
            ENDDO

            DO I = IST, IEND
               DO M = 1, 2
                  RSD(M,I,J,K) = RSD(M,I,J,K) -
     >               C2 * (FLUX(M,I+1) - FLUX(M,I-1))
               ENDDO
            ENDDO
         ENDDO
       ENDDO
```

Как было описано ранее, для распараллеливания цикла необходимо задать распределение использующихся в нем массивов на решетку виртуальных процессоров и затем выполнить отображение витков цикла на распределенные данные. Массив `FLUX` является приватным для цикла, поэтому может быть размножен между виртуальными процессорами без необходимости выполнения обменов данными между ними. Для распределения массивов `U`, `RHO_I` и `RSD` подходит следующее выравнивание: множество элементов массива `RHO_I` с индексами `J` и `K` во втором и третьем измерениях (`RHO_I(*,J,K)`) соответствует множествам элементов массивов `U` и `RSD` с индексами `J` и `K` в третьем и четвертом измерениях соответственно

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

(`U(*,*,J,K)` и `RSD(*,*,J,K)`). Тогда массив `RHO_I` может быть распределен по второму и третьему измерениям на двумерную решетку процессоров, а массивы `U` и `RSD` — по третьему и четвертому измерениям. Нераспределенные измерения размножаются. Витки цикла распределяются так, что `(K,J)`-й виток выполняется на том виртуальном процессоре, на который распределены множества элементов `RHO_I(*,J,K)`, `U(*,*,J,K)` и `RSD(*,*,J,K)`.

Листинг 12. Упрощенный вид второго цикла из процедуры RHS программы LU
Listing 12. Simplified view of the second loop from the RHS procedure of the LU program

```
!$SPF ANALYSIS(PRIVATE(FLUX))
      DO K = 2, NZ - 1
         DO I = IST, IEND
            DO J = 1, NY
               FLUX(1,J) = U(3,I,J,K)
               FLUX(2,J) = U(2,I,J,K) * RHO_I(I,J,K) * C3
            ENDDO

            DO J = JST, JEND
               DO M = 1, 5
                  RSD(M,I,J,K) = RSD(M,I,J,K) -
     >               C4 * (FLUX(M,J+1) - FLUX(M,J-1))
               ENDDO
            ENDDO
         ENDDO
       ENDDO
```

Однако на практике такое распределение массива `RSD` оказывается неприменимым из-за того, что следующие два цикла программы RHS имеют другое выравнивание по используемым в них массивам `U`, `RHO_I` и `RSD`. Упрощенный вид второго цикла приведен в листинге 12. В нем могут быть параллельно выполнены витки циклов по переменным `K` и `I`, однако для этого потребуется ввести новое выравнивание и распределение для массивов: `(K,I)`-й виток должен выполняться на том виртуальном процессоре, на который распределены множества элементов `RHO_I(I,*,K)`, `U(*,I,*,K)` и `RSD(*,I,*,K)`. Таким образом, для распараллеливания этого цикла требуется распределить массив `RHO_I` по первому и третьему измерениям, массивы `U` и `RSD` — по второму и четвертому измерениям, а оставшиеся измерения размножить.

Отметим, что для первых двух циклов процедуры RHS можно выбрать общее распределение данных: массив `RHO_I` распределить по третьему измерению, а массивы `U` и `RSD` — по четвертому измерению, и выполнять `K`-й виток каждого из циклов на том виртуальном процессоре, на который распределены множества элементов `RHO_I(*,*,K)`, `U(*,*,*,K)` и `RSD(*,*,*,K)`. Однако такое распределение данных не подойдет для третьего цикла процедуры RHS. Несоответствие распределений данных, подходящих для распараллеливания циклов процедуры, требует перераспределения данных между их выполнением, что повлечет увеличение накладных расходов на выполнение параллельной программы и существенно уменьшит ее эффективность. Оптимальным решением возникшей проблемы было бы приведение всех циклов к виду, для которого можно было бы подобрать общее распределение данных. В нашем случае требуется привести все циклы к тесновложенному виду, позволяющему выполнить распределение массива `RHO_I` по всем трем измерениям на трехмерную решетку виртуальных процессоров, а массивы `U` и `RSD` распределить по второму, третьему и четвертому измерениям.

Для приведения цикла из листинга 11 к оптимальному для распараллеливания виду можно разделить его на два тесновложенных цикла, первый из которых со степенью тесновложенности три вычислял бы значения массива `FLUX`, а второй со степенью тесновложенности четыре вычислял бы значения результирующего массива `RSD`. Причем каждый из этих циклов можно будет выполнять параллельно на всех уровнях тесновложенности. Для этого требуется выполнить преобразование, называемое “разделение цикла”. Оно заключается в замещении цикла несколькими циклами с совпадающими заголовками, тела которых являются разбиением тела исходного цикла. Под разбиением понимается некоторое деление множества операторов тела исходного цикла на непересекающиеся непустые подмножества, при этом такое, что разделение цикла по нему приводит программу к эквивалентному с исходной программой виду.

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

Однако рассматриваемый цикл не может быть разделен, поскольку вложенные в него циклы по переменной `I` имеют зависимость по переменной `FLUX`.

Чтобы избавиться от зависимости по приватной переменной-массиву `FLUX`, к циклу можно применить преобразование “расширение приватных переменных”. Оно расширит приватный массив `FLUX`, добавив к нему еще два измерения и выровняв их с циклами по переменным `K` и `J`. Для этого перед циклом необходимо разместить директиву системы SAPFOR

```
!$SPF TRANSFORM(EXPAND(K, J))
```

и запустить проход преобразования “расширение приватных переменных”. Далее для разделения цикла необходимо перед циклом разместить директиву

```
!$SPF TRANSFORM(FISSION(K, J))
```

и запустить проход “разделение циклов”. Проведя аналогичные преобразования со всеми циклами процедуры RHS, получим множество тесновложенных циклов, для которого можно выбрать общее распределение данных, позволяющее избежать перераспределений данных между выполнением циклов. Рассмотренная последовательность преобразований программы позволяет привести ее к виду, который может быть автоматизированно распараллелен системой SAPFOR достаточно эффективно — без лишних перераспределений данных между виртуальными процессорами и без лишних обменов между MPI-процессами.

Однако преобразование “расширение приватных переменных” имеет существенный недостаток. Поскольку оно заменяет приватные переменные цикла на переменные большей размерности, то общий объем оперативной памяти, требуемый для выполнения программы, увеличивается. Если вычислительная система обладает необходимым объемом оперативной памяти, то такое решение оказывается вполне допустимым, поскольку позволяет распараллелить программу без лишних перераспределений данных в ней, которые могут существенно замедлять ее выполнение. Но увеличение объема потребляемой памяти программы накладывает более строгое ограничение на максимальный размер задачи, которая может быть решена с помощью данной программы на заданной вычислительной системе.

Существует более оптимальное решение по приведению цикла из листинга 11 к тесновложенному виду. Заметим, что массив `FLUX` используется в цикле только для того, чтобы сохранить результат промежуточных вычислений на каждой `(K,J)`-й итерации цикла, который потом используется для вычисления значений элементов массива `RSD` на этой же итерации. Это свойство его использования в теле цикла и определяет его приватность для данного цикла. При этом сохранять результат промежуточных вычислений в массив `FLUX` нет никакой необходимости (кроме, может быть, удобства написания и отлаживания кода программистом) — значение элементов результирующего массива `RSD` может быть вычислено одним выражением (возможно, довольно длинным). Отсюда возникает идея преобразования кода тела цикла под названием “удаление приватных переменных”. Приватный массив можно удалить, подставив выражения, результат которых сохраняется в элементы данного массива, на место обращения к нему на чтение в других выражениях. Однако в рассматриваемом цикле непосредственно подставить выражения, результат которых присваивается в массив `FLUX` на строках 5 и 6, в место чтения из массива в строке 12 оказывается невозможным, поскольку в строке 12 происходит чтение результатов разных выражений из строк 5 и 6 в зависимости от значения итерационной переменной `M` вложенного цикла.

Решить указанную проблему помогает преобразование “разворачивание цикла”. Оно заключается в замещении цикла его телом, повторенным столько раз, сколько итераций в данном цикле, при этом в каждом повторении тела цикла вместо итерационной переменной подставляется ее очередное значение. В системе SAPFOR для выполнения данного преобразования необходимо разместить перед циклом директиву

```
!$SPF TRANSFORM(UNROLL)
```

и вызывать проход “разворачивание цикла”. После проведенного преобразования можно однозначно определить, какое из выражений на место каких обращений к массиву `FLUX` следует подставить, чтобы удалить его. Такое преобразование можно рассматривать как подстановку функции, параметрами которой являются итерационные переменные из обращения к удаляемому массиву на запись, а телом — выражение, стоящее в правой части присваивания в массив. Также отметим, что цикл, присваивающий значения массиву `FLUX`, после преобразования окажется мертвым кодом, поскольку эти значения нигде больше не используются, поэтому он будет удален.

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

Удаление приватных переменных позволяет привести цикл из рассматриваемого примера к тесновложенному виду, при котором возможно распределение данных программы без дополнительных коммуникаций и получение эффективной параллельной версии. При этом оно уменьшает объем оперативной памяти, требуемый для выполнения цикла, поскольку удаляет его приватную переменную. Задача сохранения промежуточных значений в процессе вычисления длинного выражения переносится на компилятор и может быть решена им более эффективно, чем сохранение этих значений в оперативной памяти (например, через сохранение их на регистрах процессора).

Можно отметить, что если, не выполняя разворот вложенных циклов, попытаться запустить проход удаления приватных переменных, то система SAPFOR выдаст сообщение, что не удалось найти маску фиксированных измерений для удаляемого массива и поэтому удаление не может быть выполнено.

После расстановки директив системы SAPFOR и разворачивания вложенных циклов преобразование удаления приватных переменных успешно удаляет приватный массив FLUX из данных циклов. Отметим лишь отличия реальных циклов программы от приведенных примеров. Первое — определения удаляемого массива зависят от приватных скалярных переменных, которые определяются в циклах, охватывающих данные операторы DEF. Поэтому определения данных скалярных переменных не достигают операторов USE. Решить возникшую проблему помогает вызов прохода подстановки выражений перед выполнением прохода удаления приватных переменных в системе SAPFOR (напомним, что вызов прохода подстановки выражений, как и прохода отмены подстановки, выполняется автоматически при запуске пользователем преобразования “удаление приватных переменных”). Второе — в преобразуемых циклах несколько вложенных циклов, в первом из которых находятся операторы DEF удаляемого массива, во втором — операторы USE, в третьем — снова операторы DEF, в оставшихся — снова операторы USE. При этом рекурсивных определений у удаляемого массива нет. Реализованный алгоритм построения пар операторов DEF–USE успешно работает для данных вложенных циклов.

Листинг 13. Упрощенный вид третьего цикла из процедуры RHS программы LU
Listing 13. Simplified view of the third loop from the RHS procedure of the LU program

```
!$SPF ANALYSIS(PRIVATE(FLUX,UTMP,RTMP))
      DO  J = JST,JEND
         DO  I = IST,IEND
            DO  K = 1,NZ
               UTMP(2,K) = U(2,I,J,K)
            ENDDO
            DO  K = 1,NZ
               FLUX(2,K) = UTMP(2,K) * U41
            ENDDO
            DO  K = 2,NZ - 1
               RTMP(2,K) = RSD(2,I,J,K) - FLUX(2,K + 1)
            ENDDO
            DO  K = 2,NZ - 1
               RTMP(2,K) = RTMP(2,K) + TZ3 * FLUX(2,K) - UTMP(2,K)
            ENDDO
            DO  K = 4,NZ - 3
               RSD(2,I,J,K) = RTMP(2,K) - DSSP * UTMP(2,K - 2)
            ENDDO
         ENDDO
      ENDDO
```

Третий цикл функции RHS отличается от первых двух наличием трех приватных массивов FLUX, UTMP и RTMP, причем у массива RTMP есть рекурсивные определения. Его упрощенный вид приведен в листинге 13. Определения массива FLUX зависят от массива UTMP, поэтому он может быть удален только после удаления массива UTMP. Определения массива RTMP зависят от массива FLUX. Реализованный алгоритм выбора очередности удаления приватных массивов корректно определяет порядок их удаления в нашем случае: сначала UTMP, затем FLUX и в последнюю очередь RTMP. Массивы UTMP и FLUX не имеют рекурсивных определений, поэтому их удаление осуществляется за одно выполнение прохода преобразования.

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

Массив RTMP за первое выполнение прохода удаляется частично, о чем проход выводит соответствующее сообщение. Для полного удаления приватного массива RTMP требуется выполнить данный проход еще два раза. При этом в результате подстановок размер самого длинного оператора с двух строк фиксированной длины (ограниченной в 80 символов) увеличился до 25 строк, что иллюстрирует трудоемкость такого преобразования при его выполнении вручную.

После удаления приватных переменных в циклах функции RHS эти циклы с помощью прохода разделения циклов разбиваются на несколько тесновложенных циклов, которые эффективно распараллеливаются системой SAPFOR без лишних перераспределений данных. Для преобразования остальной части программы LU к оптимальному для распараллеливания виду используются преобразования подстановки функций, слияния циклов и сужения приватных переменных.

6.2. Преобразование программ SP и BT. Программы SP и BT решают системы линейных уравнений для пятидиагональной и трехдиагональной матриц соответственно. Их структуры схожи — в основном итерационном цикле программы вызываются вычислительные процедуры COMPUTE_RHS, X_SOLVE, Y_SOLVE и Z_SOLVE и затем процедура ADD, прибавляющая вычисленный на текущем шаге результат к основному. Для преобразования циклов процедуры COMPUTE_RHS к тесновложенному виду достаточно преобразования разделения циклов. Циклы процедур X_SOLVE, Y_SOLVE и Z_SOLVE имеют более сложную структуру и используют приватные массивы для вычисления результирующих значений выходного массива. После подстановки процедур, в которых происходит инициализация части приватных массивов, и выполнения разворачивания и слияния вложенных циклов, некоторые из приватных массивов могут быть удалены. Часть приватных массивов удалить нельзя, поскольку для них нельзя подобрать маску фиксированных измерений. Поэтому для эффективного распараллеливания данных циклов оставшиеся приватные переменные также приходится расширять и затем разделять циклы для приведения их к тесновложенному виду. Данный ряд преобразований не позволяет полностью избавиться от приватных массивов в основных вычислительных циклах программы и избежать их расширения, однако приводит данные циклы к более эффективному для автоматизированного распараллеливания через систему SAPFOR виду.

6.3. Преобразование программы EP. Программа EP представляет собой цикл, в теле которого используется приватный массив X, заполняемый псевдослучайными числами в процедуре VRANLC, для вычисления трудоемких математических операций извлечения квадратного корня и взятия логарифма от данных чисел. При этом все витки цикла могут выполняться параллельно, а приватный массив X может быть удален. Для этого необходимо выполнить подстановку процедуры VRANLC. Упрощенный вид цикла после подстановки процедуры приведен в листинге 14.

Листинг 14. Упрощенный вид цикла из программы EP
Listing 14. Simplified view of the loop from the EP program

```
1  !$SPF ANALYSIS(PRIVATE(X))
2        DO  K = 1,NP
3           DO  I_0 = 1, 2 * NK
4              Y = T + 1
5              T = Y * Y
6              X(I_0) = T
7           ENDDO
8           DO  I = 1, NK
9              X1 = 2.D0 * X(2 * I - 1) - 1.D0
10             X2 = 2.D0 * X(2 * I) - 1.D0
11             T1 = X1** 2 + X2** 2
12          ENDDO
13       ENDDO
```

Однако приватный массив X не может быть удален без дополнительных преобразований — определение переменной T, которая используется в выражении оператора DEF на строке 6, не достигает операторов USE на строках 9 и 10, поскольку оно уничтожается оператором на строке 5 в теле охватывающего оператор DEF цикла. Чтобы определение переменной T достигало операторов USE, необходимо объединить циклы по переменным I_0 и I. Данные циклы имеют разные границы, поэтому для их объединения

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

требуется привести их к одинаковым границам. Для этого можно уменьшить вдвое число витков цикла по переменной `I_0`, продублировав операторы в его теле. В настоящий момент в системе SAPFOR нет подобного преобразования, поэтому его можно выполнить только вручную. При этом необходимо корректно изменить присваивания в массив `X`, поскольку после сокращения числа витков вдвое на каждом витке цикла присваиваются значения сразу в два элемента массива. Результат сокращения числа витков и объединения циклов представлен в листинге 15.

Листинг 15. Цикл из программы EP после объединения вложенных циклов
Listing 15. Loop from EP program after merging nested loops

```
1  !$SPF ANALYSIS(PRIVATE(X))
2        DO  K = 1,NP
3           DO  I = 1, NK
4              Y = T + 1
5              T = Y * Y
6              X(2 * I - 1) = T
7              Y = T + 1
8              T = Y * Y
9              X(2 * I) = T
10             X1 = 2.D0 * X(2 * I - 1) - 1.D0
11             X2 = 2.D0 * X(2 * I) - 1.D0
12             T1 = X1** 2 + X2** 2
13          ENDDO
14       ENDDO
```

После объединения вложенных циклов индексные выражения первого (и единственного) измерения всех обращений к приватному массиву `X` являются аффинными, с итерационной переменной цикла `I`. Поэтому для удаления данного массива может быть применено преобразование удаления приватных переменных с модификацией основного алгоритма, описанной в разделе 5.5. Однако в представленном в листинге 15 цикле массив `X` не может быть удален полностью, поскольку определение переменной `T` на пути от оператора DEF на строке 6 до оператора USE на строке 10 будет уничтожено оператором присваивания на строке 8. Чтобы избежать этого, можно сократить длину пути между операторами DEF и USE и перенести оператор присваивания в переменную `X1` со строки 10 сразу после оператора DEF на строке 6. В настоящий момент в проходе удаления приватных переменных не реализовано преобразование перестановки операторов для решения подобных проблем, не позволяющих выполнить подстановку в паре операторов DEF–USE, поэтому данное преобразование также необходимо выполнить вручную. После перестановки оператора USE проход удаления приватных переменных корректно удаляет приватный массив `X`. Результат удаления представлен в листинге 16.

Листинг 16. Результат преобразования цикла из программы EP
Listing 16. The result of the transformation of the loop from the EP program

```
1  !$SPF ANALYSIS(PRIVATE(X))
2        DO  K = 1,NP
3           DO  I = 1, NK
4              Y = T + 1
5              T = Y * Y
6              X1 = 2.D0 * ((T + 1) * (T + 1)) - 1.D0
7              Y = T + 1
8              T = Y * Y
9              X2 = 2.D0 * ((T + 1) * (T + 1)) - 1.D0
10             T1 = X1** 2 + X2** 2
11          ENDDO
12       ENDDO
```

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

7. Анализ эффективности полученных параллельных версий с помощью SAPFOR. Полученные с помощью автоматизированных преобразований и распараллеливания в системе SAPFOR версии программ LU, BT, SP и EP были запущены на гибридной вычислительной системе, имеющей два 16-ядерных центральных процессора Intel Xeon Gold 6142 v4 и графический ускоритель NVIDIA Tesla V100. Для каждой из программ было проведено сравнение двух версий: версии, полученной с применением реализованного преобразования удаления приватных переменных, и версии, полученной без применения реализованного преобразования, т.е. максимально эффективной версии, которую можно было получить через систему SAPFOR до реализации в ней прохода удаления приватных переменных (далее в тексте — версия, полученная через преобразование расширения приватных переменных [22]). Результаты сравнительных запусков версий программ на классе размера задачи C приведены в табл. 1. Для каждой версии программы запуск был проведен на 32 нитях CPU и на одном GPU.

Для программы LU наблюдается ускорение почти в 3.3 раза на GPU и в 1.6 раз на CPU. Объем потребляемой памяти уменьшился почти в 5.3 раза. Данные результаты объясняются тем, что расширение приватных переменных выполняется сразу на два измерения большой размерности (соответствующим двум циклам гнезда с большим количеством витков). Поэтому данное преобразование для программы LU оказывается очень затратным по памяти. Удаление приватных переменных, помимо уменьшения объема потребляемой памяти, уменьшает количество кэш-промахов при выполнении циклов, поскольку исключает из них обращения на запись и чтение из приватных переменных, что ускоряет выполнение программы.

Как было сказано выше при описании программы EP, ее структура достаточно проста. В программе используется всего один одномерный приватный массив внутри основного вычислительного цикла, поэтому его удаление снижает потребление памяти программой до почти минимально возможного, показывая снижение на 4 порядка по сравнению с расширением данного массива. Тело цикла содержит вычислительно сложные операции, поэтому после удаления приватной переменной программа показывает существенное ускорение на GPU (на два порядка по сравнению с 10% на CPU), поскольку графический ускоритель содержит большое количество производительных ядер и хорошо справляется с потоковой вычислительно сложной обработкой данных.

Как было описано ранее, программы SP и BT схожи по своей структуре и необходимым для их распараллеливания преобразованиям. Поскольку в обеих программах не удается удалить все приватные массивы и необходимо выполнять расширение оставшихся, то разница в потребляемой версиями данных программ памяти не так значительна, как для программ LU и EP: 16% для программы SP, а для

Таблица 1. Результаты сравнительных запусков (п.п. — приватная переменная)

Table 1. Results of comparative runs (p.v. — private variable)

Программа Program	Преобразование Transformation	1 CPU время выполнения, с 1 CPU execution time, s	32 потока CPU время выполнения, с 32 threads execution time, s	Потребление памяти, ГБ Memory consumption, GB
LU	Удаление п.п. P.v. deletion	14.2	200.8	2.1
	Расширение п.п. P.v. extension	46.2	321.5	11.1
EP	Удаление п.п. P.v. deletion	0.1	14.5	0.001
	Расширение п.п. P.v. extension	11.9	16.2	64
BT	Удаление п.п. P.v. deletion	20.4	270.0	16
	Расширение п.п. P.v. extension	56.3	859.0	16
SP	Удаление п.п. P.v. deletion	21.0	146.1	4.2
	Расширение п.п. P.v. extension	25.7	201.0	5

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

программы BT разница в потребляемой памяти вообще отсутствует. Однако удаление части приватных переменных позволяет получить существенное ускорение для программы BT — почти в 2.8 раза на GPU и почти в 3.2 раза на CPU. Для программы SP ускорение меньше — 18% для GPU и 28% для CPU, что объясняется тем, что удаляемые массивы в программе SP имеют существенно меньший размер, чем в программе BT (однако, несмотря на меньший размер, их удаление оказывается более заметным по сокращению объема потребляемой программой памяти, чем удаление приватных массивов в программе BT).

Данные результаты свидетельствуют о том, что реализованное преобразование удаления приватных переменных при его применении в процессе автоматизированного распараллеливания рассматриваемых программ в системе SAPFOR позволяет получить более эффективные параллельные версии данных программ. Для всех рассматриваемых программ было достигнуто ускорение при выполнении как на GPU, так и на CPU, для большинства программ также было достигнуто сокращение потребления памяти. Наиболее показательным является применение преобразования в процессе распараллеливания программы LU, где при минимальном количестве ручных изменений исходного кода с помощью реализованного преобразования получается удалить все приватные массивы из части основных по трудоемкости вычислительных циклов, что дает кратное ускорение выполнения программы и кратное сокращение объема потребляемой памяти.

Для применения реализованного преобразования на программе EP потребовалось выполнить несколько преобразований вручную (сокращение числа витков цикла и перестановку операторов в теле цикла), что требует от пользователя системы SAPFOR существенно большего участия в процессе распараллеливания и показывает пути для возможного дальнейшего улучшения системы через реализацию в ней данных преобразований. Применение преобразования удаления приватных переменных к программам BT и SP не позволило удалить все приватные переменные основных циклов, поэтому не дало существенного снижения объема потребляемой программами памяти, однако позволило добиться кратного ускорения выполнения для программы BT.

8. Заключение. В статье рассмотрено преобразование последовательных Fortran-программ “удаление приватных переменных”. Предложен алгоритм реализации данного преобразования, который оформлен в виде проходов системы автоматизированного распараллеливания SAPFOR.

Реализованное преобразование исследовано в процессе автоматизированного преобразования и распараллеливания программ LU, BT, SP и EP из пакета тестов NAS Parallel Benchmarks. Суммарный объем преобразованных программ составил примерно 14000 строк, из которых реализованным преобразованием было изменено около 1100 строк. В результате применения данного преобразования в программах LU и EP удалось удалить все приватные массивы основных вычислительных циклов, в программах BT и SP — только часть из них. При этом в большинстве случаев для применения реализованного преобразования не требуется существенного изменения исходного кода программы (кроме расстановки директив системы SAPFOR), однако для его применения к программе EP потребовалось выполнить часть трансформаций кода вручную, что показывает пути для возможного дальнейшего улучшения реализованного преобразования и реализации в системе SAPFOR новых преобразований.

Полученные с помощью реализованного преобразования параллельные версии данных программ были запущены на гибридной вычислительной системе, а также выполнено сравнение с версиями, полученными без применения данного преобразования. Результаты демонстрируют высокую эффективность версий, полученных с помощью реализованного преобразования, показывая на некоторых программах кратное снижение объема потребляемой памяти и кратное ускорение при выполнении как на CPU, так и на GPU.

Список литературы

1. OpenMP Specifications for Parallel Programming. https://www.openmp.org/specifications/. Cited January 21, 2025.
2. MPI Documents. https://www.mpi-forum.org/docs/. Cited January 21, 2025.
3. CUDA Toolkit Documentation 12.6 Update 3. https://docs.nvidia.com/cuda/. Cited January 21, 2025.
4. *Wolfe M.* High performance compilers for parallel computing. New York: Addison-Wesley, 1995.
5. *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.

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

6. *Verdoolaege S., Juega J.C., Cohen A., et al.* Polyhedral parallel code generation for CUDA // ACM Trans. Archit. Code Optim. 2013. **9**, N 4. Article Number 54. doi 10.1145/2400682.2400713.
7. *Grosser T., Groesslinger A., Lengauer C.* Polly — performing polyhedral optimizations on a low-level intermediate representation // Parallel Processing Letters. 2012. **22**, N 4. Article Number 1250010. doi 10.1142/S0129626412500107.
8. *Grosser T., Hoefler T.* Polly-ACC transparent compilation to heterogeneous hardware // Proc. 2016 Int. Conf. on Supercomputing, Istanbul, Turkey, June 1–3, 2016. New York: ACM Press, 2016. 1:1–1:13. doi 10.1145/2925426.2926286.
9. *Caamano J.M.M., Sukumaran-Rajam A., Baloian A., et al.* APOLLO: automatic speculative POLyhedral loop optimizer // Proc. 7th Int. Workshop on Polyhedral Compilation Techniques (IMPACT), Stockholm, Sweden, January 23, 2017. https://inria.hal.science/hal-01533692v1. Cited January 21, 2025.
10. OpenCL Specification. https://www.khronos.org/registry/OpenCL/. Cited January 21, 2025.
11. *Lattner C., Adve V.* LLVM: a compilation framework for lifelong program analysis & transformation // Proc. Int. Symp. on Code Generation and Optimization (CGO'04), San Jose, USA, March 20–24, 2004. doi 10.1109/CGO.2004.1281665.
12. *Doerfert J., Streit K., Hack S., Benaissa Z.* Polly's polyhedral scheduling in the presence of reductions // Proc. 5th Int. Workshop on Polyhedral Compilation Techniques, Amsterdam, The Netherlands, January 19, 2015. https://www.impact-workshop.org/impact2015/papers/impact2015-doerfert.pdf. Cited January 21, 2025.
13. Описание DVM-системы. http://dvm-system.org/. (Дата обращения: 21 января 2025).
14. Документация по языкам C-DVMH и Fortran-DVMH. http://dvm-system.org/ru/docs/. (Дата обращения: 21 января 2025).
15. Описание системы SAPFOR. http://keldysh.ru/dvm/SAPFOR/. (Дата обращения: 21 января 2025).
16. *Бахтин В.А., Жукова О.Ф., Катаев Н.А. и др.* Автоматизация распараллеливания программных комплексов // Труды XVIII Всероссийской научной конференции "Научный сервис в сети Интернет", 19–24 сентября 2016, Новороссийск. М.: ИПМ им. М.В. Келдыша, 2016. 76–85.
17. *Колганов А.С.* Автоматизация распараллеливания Фортран-программ для гетерогенных кластеров: автореф. дисс. канд. физ.-мат. наук. М.: ИПМ им. М.В. Келдыша, 2020.
18. *Бартеньев О.В.* Современный Фортран. М.: ДИАЛОГ–МИФИ, 2000.
19. *Ахо А.В., Лам М.С., Сети Р., Ульман Дж.Д.* Компиляторы: принципы, технологии и инструментарий. М.: Вильямс, 2018.
20. *Bodin F., Beckman P., Gannon D., et al.* Sage++: an object-oriented toolkit and class library for building Fortran and C++ restructuring tools. https://www.researchgate.net/publication/2725408_Sage_An_Object-Oriented_Toolkit_and_Class_Library_for_Building_Fortran_and_C_Restructuring_Tools. Cited January 21, 2025.
21. NAS Parallel Benchmarks. https://www.nas.nasa.gov/software/npb.html. Cited January 21, 2025.
22. *Колганов А.С., Гусев Г.Д.* Преобразование последовательных Fortran-программ для их распараллеливания на гибридные кластеры в системе SAPFOR // Вычислительные методы и программирование. 2022. **23**, № 4. 288–310. doi 10.26089/NumMet.v23r418.

Поступила в редакцию
1 октября 2024 г.

Принята к публикации
14 января 2025 г.

Информация об авторах

Александр Сергеевич Колганов — к.ф.-м.н., научн. сотр.; Институт прикладной математики имени М. В. Келдыша РАН (ИПМ РАН), Миусская пл., д. 4, 125047, Москва, Российская Федерация.

Григорий Дмитриевич Гусев — магистр; Московский государственный университет имени М. В. Ломоносова, Ленинские горы, 1, стр. 4, 119991, Москва, Российская Федерация.

References

1. OpenMP Specifications for Parallel Programming. https://www.openmp.org/specifications/. Cited January 21, 2025.
2. MPI Documents. https://www.mpi-forum.org/docs/. Cited January 21, 2025.
3. CUDA Toolkit Documentation 12.6 Update 3. https://docs.nvidia.com/cuda/. Cited January 21, 2025.

2025, **26** (1), 58–84. doi 10.26089/NumMet.v26r105

4. M. Wolfe, *High Performance Compilers for Parallel Computing* (Addison-Wesley, New York, 1995).
5. U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer," ACM SIGPLAN Not. **43** (6), 101–113 (2008). doi 10.1145/1379022.1375595.
6. S. Verdoolaege, J. C. Juega, A. Cohen, et al., "Polyhedral Parallel Code Generation for CUDA," ACM Trans. Archit. Code Optim. **9** (4), Article Number 54 (2013). doi 10.1145/2400682.2400713.
7. T. Grosser, A. Groesslinger, and C. Lengauer, "Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation," Parallel Process. Lett. **22** (04), Article Number 1250010 (2012). doi 10.1142/S0129626412500107.
8. T. Grosser and T. Hoefler, "Polly-ACC Transparent Compilation to Heterogeneous Hardware," in *Proc. 2016 Int. Conf. on Supercomputing, Istanbul, Turkey, June 1–3, 2016.* (ACM Press, New York, 2016), pp. 1:1–1:13. doi 10.1145/2925426.2926286.
9. J. M. M. Caamano, A. Sukumaran-Rajam, A. Baloian, et al., "APOLLO: Automatic Speculative POLyhedral Loop Optimizer," in *Proc. 7th Int. Workshop on Polyhedral Compilation Techniques (IMPACT), Stockholm, Sweden, January 23, 2017.* https://inria.hal.science/hal-01533692v1. Cited January 21, 2025.
10. OpenCL Specification. https://www.khronos.org/registry/OpenCL/. Cited January 21, 2025.
11. C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. Int. Symp. on Code Generation and Optimization (CGO'04), San Jose, USA, March 20–24, 2004.* doi 10.1109/CGO.2004.1281665.
12. J. Doerfert, K. Streit, S. Hack, and Z. Benaissa, "Polly's Polyhedral Scheduling in the Presence of Reductions," in *Proc. 5th Int. Workshop on Polyhedral Compilation Techniques, Amsterdam, The Netherlands, January 19, 2015.* https://www.impact-workshop.org/impact2015/papers/impact2015-doerfert.pdf. Cited January 21, 2025.
13. Description of DVM-system. http://dvm-system.org/. Cited January 21, 2025.
14. Documentation for C-DVMH and Fortran-DVMH Languages. http://dvm-system.org/ru/docs/. Cited January 21, 2025.
15. Description of SAPFOR System. http://keldysh.ru/dvm/SAPFOR/. Cited January 21, 2025.
16. V. A. Bakhtin, O. F. Zhukova, N. A. Kataev, et al., "Automation of Parallelization of Software Complexes," in *Proc. Conf. on Scientific Service on the Internet, Novorossiysk, Russia, September 19–24, 2016.* (Keldysh Inst. Applied Math., Moscow, 2016), pp. 76–85.
17. 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).
18. O. V. Barten'ev, *Modern Fortran* (DIALOG-MEPHI, Moscow, 2000) [in Russian].
19. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison-Wesley, Boston, 2006; Williams, Moscow, 2018).
20. F. Bodin, P. Beckman, D. Gannon, et al., *Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools.* https://www.researchgate.net/publication/2725408_Sage_An_Object-Oriented_Toolkit_and_Class_Library_for_Building_Fortran_and_C_Restructuring_Tools. Cited January 21, 2025.
21. NAS Parallel Benchmarks. https://www.nas.nasa.gov/software/npb.html. Cited January 21, 2025.
22. A. S. Kolganov and G. D. Gusev, "Transformation of Sequential Fortran Programs for Their Parallelization into Hybrid Clusters in the SAPFOR," Numerical Methods and Programming **23** (4), 288–310 (2022). doi 10.26089/NumMet.v23r418.

Received
October 1, 2024

Accepted for publication
January 14, 2025

Information about the authors

Alexander S. Kolganov — Ph.D., Scientist; Keldysh Institute of Applied Mathematics of RAS, Miusskaya ploshchad', 4, 125047, Moscow, Russia.

Grigorii D. Gusev — M.Sc.; Lomonosov Moscow State University, Leninskie Gory, 1, building 4, 119991, Moscow, Russia.