Научная статья

УДК 004.052.42
DOI 10.25205/1818-7900-2024-22-1-5-20

Построение комплекса автоматизированной отладки фрагментированных программ

Андрей Юрьевич Власенко[1], Михаил Антонович Мичуров[2]
Василий Дмитриевич Царёв[3], Максим Андреевич Курбатов[4]

[1-4] Новосибирский государственный университет
[1] Институт вычислительной математики и математической геофизики СО РАН
Новосибирск, Россия

[1]a.vlasenko@g.nsu.ru, https://orcid.org/0000-0001-8247-5736
[2]m.michurov@g.nsu.ru, https://orcid.org/0000−0002−0404−6802
[3]v.tsarev@g.nsu.ru
[4]m.kurbatov@g.nsu.ru

Аннотация

В статье кратко изложена концепция фрагментированного программирования, а также принципиальное устройство системы автоматического конструирования параллельных программ LuNA (Language for Numerical Algorithms) и одноименного языка программирования. Описаны такие свойства системы LuNA, как возможность работы на вычислительных системах разных типов, динамическая балансировка нагрузки на узлы и процессорные ядра вычислительного кластера и другие.

Целью описываемой в статье работы является создание программного комплекса отладки фрагментированных программ в системе LuNA. В связи с этим приведен ряд ошибок, специфических для фрагментированных программ. Перечислены подходы к отладке параллельных программ и проанализирована их применимость к фрагментированным программам.

Подробно изложена реализация методов статического анализа и «посмертного анализа» в авторских средствах автоматизированной отладки фрагментированных программ для системы LuNA. Два средства статического анализа используют такие промежуточные представления, как абстрактное синтаксическое дерево и граф информационных зависимостей. Третье средство базируется на оригинальном методе статического анализа, заключающемся в генерации Prolog-программы, содержащей набор фактов об исходной LuNA-программе и соответствующие ошибочным ситуациям правила. При последующем запуске Prolog-программы факты проверяются на соответствие правилам, в результате чего пользователю выдаются сообщения о найденных ошибках.

Представлена модель интеграции созданных средств в единый комплекс автоматизированной отладки фрагментированных программ, а также утилита автоматизированного тестирования инструментов отладки LuNA-программ.

Ключевые слова

фрагментированное программирование, система LuNA, логические ошибки, автоматизированная отладка, статический анализ, «посмертный анализ».

Финансирование

Исследования выполнены в рамках государственного задания ИВМиМГ СО РАН 0251-2022-0005.

Для цитирования

Власенко А. Ю., Мичуров М. А., Царёв В. Д., Курбатов М. А. Построение комплекса автоматизированной отладки фрагментированных программ // Вестник НГУ. Серия: Информационные технологии. 2024. Т. 22, № 1. С. 5–20. DOI 10.25205/1818-7900-2024-22-1-5-20

Constructing a Complex for Automated Debugging of Fragmented Programs

Andrey Yu. Vlasenko[1], Mihail A. Michurov[2]
Vasiliy D. Tsarev[3], Maksim A. Kurbatov[4]

[1-4]Novosibirsk State University
[1]Institute of Computational Mathematics and Mathematical Geophysics SB RAS
Novosibirsk, Russian Federation

[1]a.vlasenko@g.nsu.ru, https://orcid.org/0000-0001-8247-5736
[2]m.michurov@g.nsu.ru, https://orcid.org/0000−0002−0404−6802
[3]v.tsarev@g.nsu.ru
[4]m.kurbatov@g.nsu.ru

Abstract

The paper is devoted to the authors' tools for automated debugging of fragmented programs in the LuNA (Language for Numerical Algorithms) system. The LuNA system uses approach of fragmented programming, which allows the researcher to design parallel programs in an automated mode.

The main problem discussed in the paper is the detection of errors specific to LuNA programs. The operation algorithms of four developed tools for automated debugging of LuNA programs are described. Three of them are based on the static analysis approach and differ in the intermediate representations used. The first uses an abstract syntax tree (AST). The second uses a data dependency graph (DDG). The third tool generates a Prolog program containing a set of facts about the original LuNA program and rules corresponding to erroneous situations. Due to the use of various intermediate representations, each of the static analysis tools is able to detect its own types of errors. The last tool uses the "post-mortem analysis" approach. Each process of a parallel program collects a trace file during its operation and at the end of the work these files are analyzed by a special utility.

The paper also describes the developed utility for testing automated debugging tools. The set of test programs with various errors, as well as error-free programs, has been developed for this utility.

The last part describes the designed mechanism for integrating the developed automated debugging tools into a single software package.

Keywords

fragmented programming, logical errors, automated debugging, static analysis, "post-mortem analysis"

Funding

This work was carried out under state contract with ICMMG SB RAS 0251-2022-0005.

For citation

Vlasenko A. Y., Michurov M. A., Tsarev V. D., Kurbatov M. A. Constructing a complex for automated debugging of fragmented programs. *Vestnik NSU. Series: Information Technologies*, 2024, vol. 22, no. 1, pp. 5–20 (in Russ.) DOI 10.25205/1818-7900-2024-22-1-5-20

Введение

Технология фрагментированного программирования подразумевает разделение программы на части, называемые фрагментами кода (ФК). Во время работы программы каждый запущенный экземпляр ФК представляет собой самостоятельную динамическую единицу, называемую фрагментом вычислений (ФВ или CF – calculation fragment). При этом если по коду программы ФК1 предшествовал ФК2, то из этого не следует, что соответствующий ФВ2 будет выполнен строго после ФВ1. Если между ними нет зависимостей по данным, то эти ФВ могут исполняться параллельно, и ФВ2 может завершиться даже раньше, чем ФВ1. ФВ могут быть запущены на различных узлах мультикомпьютера, ядрах мультипроцессора или графических ускорителях.

Еще одной важной сущностью во фрагментированной программе является фрагмент данных (ФД или DF – data fragment), представляющий собой некий аналог переменной в языках императивного программирования. Особенность ФД состоит в том, что это объект единствен-

ного присваивания. Любой ФВ потребляет на вход некоторое множество ФД (в том числе и пустое) и выдает на выходе другие ФД, которые, в свою очередь, могут быть потреблены следующими ФВ. Таким образом, ФВn является зависимым по данным от ФВm, если среди входных аргументов ФВn есть ФД, которые вычисляет ФВm, либо существует последовательность ФВ1, ФВ2,... , ФВk таких, что ФВ1 зависим по данным от ФДm, ФВ2 – от ФВ1, …, ФВn – от ФВk. ФД, как и ФВ, должны обладать возможностью миграции между узлами мультикомпьютера в целях перераспределения вычислительной нагрузки.

Система LuNA и одноименный язык, разрабатываемые в ИВМиМГ СО РАН и на кафедре параллельных вычислений ФИТ НГУ, реализуют технологию фрагментированного программирования [1; 2]. При помощи языка LuNA описывается рекурсивно-перечислимое множество триплетов <*in*, *mod*, *out*>, где *in* – множество входных ФД, *out* – множество выходных, а *mod* – модуль, реализующий ФК, который вычисляет *out*, если ему на вход поступило *in*. Если множество *in* не полное (не все входные ФД уже вычислены), то *mod* просто ожидает готовности недостающих ФД. ФД могут иметь тип целое число, число с плавающей точкой и строковый тип.

В описываемой системе ФК могут быть как структурированными (на языке LuNA), так и атомарными (на C/C++ или Fortran). Для ФК определены входные и выходные переменные, являющиеся его формальными параметрами. Также ФК не должен иметь «побочных эффектов», т. е. не должен изменять ничего (глобальных переменных, переменных среды, системных настроек и т. д.), кроме своих выходных параметров. Данное требование, как и требование единственности присваивания значений ФД, необходимо для обеспечения возможности исполнения порождаемых ФВ в произвольном порядке, в том числе и параллельно.

Фрагментированная программа в системе LuNA может работать на параллельной вычислительной системе за счет перераспределения ФВ по узлам вычислительного кластера, процессорным ядрам одного узла и графическим ускорителям. При этом динамическим распределением ФВ по вычислительным устройствам занимается runtime-система. В этом заключается балансировка нагрузки.

Листинг 1. Пример ошибочной LuNA-программы

```
1: import c_print(real) as print; // импорт атомарных ФК на C/C++
2: import c_init(real, name) as init;  // инициализация ФД
(второго аргумента) вещественным числом
3: import c_bar(string) as bar;

4: sub main(int a){
5:     df a, b, c, d, e;  // объявление ФД
6:     foo(a);
7:     bar(1);
8:     init(1, b);
9:     init(2, b);
10:    print(e);
11: }

12: sub foo(name x){
13:    init(1, x);
14:    print(x);
15: }
```

Создание LuNA-программ сопряжено с риском написания ошибочного кода. Ввиду особенностей языка LuNA-программам свойственны специфические семантические ошибки, не характерные для стандартных языков и технологий параллельного программирования.

Авторами статьи была создана база данных с ошибками, их описанием и исходным кодом, их воспроизводящим. На текущий момент база данных содержит 22 различные ошибки, из которых 4 будут рассмотрены ниже.

Демонстрация алгоритмов и структур будет произведена на основе программы из листинга 1, содержащей следующие ошибки.

1. Попытка использования неинициализированного ФД. Эта ошибка приводит к зависанию программы, поскольку ФВ, ожидающий получения неинициализированного ФД, вынужден ожидать его бесконечно. Пример: строка 10 – ФД *e*.

2. Повторная инициализация ФД. В случае инициализации уже инициализированного ФД программа может завершиться преждевременно. Такую ситуацию пользователю может быть тяжело обнаружить визуально в сложной программе с большим количеством ветвлений. Пример: строка 9 – ФД *b* уже был инициализирован в строке 8.

3. Неиспользуемый ФД. Данная ситуация не является ошибочной в полном смысле слова, но тем не менее является примером неоптимального поведения и, что важнее, может указывать на наличие иных ошибок, связанных с указанным ФД. Пример: строка 5 – ФД *d*.

4. Несоответствие типов аргументов при вызове атомарного ФК. Возникает в случае передачи в ФК аргумента иного типа, нежели ожидался. Пример: строка 7 – несоответствие константной единицы и типа *string*.

Фрагментированное программирование – новый перспективный подход для построения программ, способных эффективно использовать высокопроизводительные вычислительные ресурсы. А в связи с тем что программам в этой технологии свойственны специфические ошибки, острым вопросом встает создание средств отладки. Авторы данной статьи задались целью разработки комплекса средств автоматизированного обнаружения ошибок в LuNA-программах. Автоматизированная отладка существенно ускорит процесс разработки как для новых, так и для уже опытных пользователей системы LuNA. Программный комплекс автоматизированной отладки будет построен на основе интеграции четырех созданных инструментальных средств (AST-analyzer, DeGSA, Prolog-analyzer и luna-trace), каждое из которых будет описано ниже.

1. Методы отладки

Для обнаружения ошибок в программах существуют различные инструментальные средства, которые можно классифицировать по нескольким используемым методам. С целью выбора подходов, пригодных к использованию при построении средств автоматизированной отладки, был выполнен нижеследующий обзор и анализ.

Самым распространенным является метод диалоговой отладки, который используют такие популярные системы, как TotalView и Linaro Distributed Debugging Tool (DDT)[1]. При работе с подобной системой пользователь расставляет в программе контрольные точки, движется по шагам, анализируя изменения значений переменных и др. Такой анализ хорошо знаком большинству программистов, соответственно системы диалоговой отладки обладают низким входным порогом использования. Однако этот подход плохо применим в условиях больших суперкомпьютерных центров, поскольку там пользователи ставят свои задачи в очередь и на этом теряют над ними контроль. Возможности интерактивного взаимодействия с исполняющейся программой при этом сильно ограничены. Кроме того, на сложных параллельных программах встречаются такие ситуации, когда на малых размерностях при запуске небольшого количества процессов программа отрабатывает корректно, а при переходе на сотни процессов проявляется ошибка. Методом диалоговой отладки выявлять подобные ошибки крайне трудоемко.

[1] Get started with DDT. URL: https://docs.linaroforge.com/23.0.2/html/forge/ddt/get_started_ddt/index.html

В связи с этим перспективными представляются автоматизированные методы отладки, не требующие непосредственного диалога пользователя с работающей программой. Некоторые из них будут изложены ниже.

Статический анализ выполняется над исходным кодом или над одним из промежуточных представлений программы (абстрактным синтаксическим деревом – AST [3], трехадресным кодом, графом потока управления и др.). Таким образом, статический анализ производится до непосредственного запуска программы.

Глубина статического анализа может варьироваться от определения поведения отдельных операторов до анализа всего имеющегося исходного кода. В коммерческой среде статический анализ главным образом применяется для поиска уязвимостей в системах, критичных к безопасности, однако при помощи данного подхода возможно обнаруживать различные семантические ошибки. Примерами статических анализаторов являются Clang-Tidy, PVS-Studio, PC-LINT и др.

Очевидными преимуществами данного метода являются:

– возможность полного покрытия кода – при динамическом анализе программа проходит только один из всех возможных вариантов;

– проверка на ошибки без запуска анализируемого приложения (параллельные программы для суперкомпьютерных систем очень часто работают весьма значительное время);

– независимость от конкретного компилятора и среды исполнения.

По этим причинам три разрабатываемых и описанных ниже средства обнаружения ошибок в LuNA-программах используют подход статического анализа. Принципиальное их различие – в промежуточных представлениях, на базе которых осуществляется анализ.

К минусам же метода статического анализа относятся так называемые «ложные срабатывания», когда средство выдает предупреждения на фактически нормальное поведение.

Оригинальным и перспективным вариантом статического анализа является генерация по исходному коду в одном языке программы на каком-либо ином языке (как правило – в другой парадигме), после чего производится стандартная компиляция с последующим запуском или интерпретация сгенерированной программы. В процессе работы обнаруживаются заданные правилами ошибочные ситуации [4]. Такой подход, в частности, применяется в еще одном инструментальном средстве отладки фрагментированных программ, генерирующем Prolog-программу и описанном далее в п. 5.

Еще одним подходом к автоматизированному обнаружению ошибок в программах (в том числе и параллельных) является сравнительная отладка. Суть этого метода заключается в том, что инструментальному средству дается 2 версии программы: эталонная и отлаживаемая. Средство сравнивает значения переменных обеих программ в определенных пользователем контрольных точках и сигнализирует при несовпадении. В том числе возможна и такая ситуация, что за эталонную будет взята последовательная программа, а за отлаживаемую – параллельная. Сравнению чаще всего подвергаются трассы обеих программ с записанными значениями переменных в контрольных точках. Характерными представителями данного подхода являются отладчик Guard [5] и система DVM [6]. Данный метод на текущем этапе не рассматривается в качестве возможного для внедрения в разрабатываемый комплекс автоматизированной отладки LuNA-программ в связи с тем, что эталонная версия, как правило, отсутствует.

Следующий метод – автоматизированный (или динамический) контроль корректности [7], при котором анализируемая программа проверяется инструментальным средством на соответствие определенным правилам, заложенным в инструментальном средстве. Примерами таких средств являются Intel Trace Analyzer and Collector (ITAC) и MUST [8], проверяющие MPI-программу на соблюдение ограничений MPI-стандарта. Данный анализ можно выполнять по ходу работы программы (в online-режиме) или по собранной трассе после завершения программы («посмертный анализ»). Поскольку ряд семантических ошибок крайне сложно обнаружить методом статического анализа, то одно из разрабатываемых инструментальных средств отладки

использует именно подход автоматизированного контроля корректности (см. п. 3), благодаря чему расширяется охват возможных проблем в LuNA-программах.

Последним рассмотренным в данной статье методом будет верификация программ на моделях (Model Checking) [9]. При использовании данного подхода формулируются требования к анализируемой системе (в частном случае – программе), для чего зачастую применяется аппарат темпоральной логики. В то же время строится модель на некотором псевдоязыке, включающая только те свойства системы, которые могут повлиять на соблюдение требований в процессе функционирования системы. Данная модель обладает конечным числом состояний и представима в виде графа. Затем верификатор выполняет исчерпывающий поиск в данном графе тех путей, которые ведут к нарушению требований (так называемых «контрпримеров»). Одной из основных проблем метода верификации программ на моделях является «комбинаторный взрыв» пространства состояний – экспоненциальный рост числа состояний и путей в результирующем графе в зависимости от количества параллельных ветвей исследуемой программы и количества переменных, принимаемых во внимание в модели. Одним из самых популярных верификаторов является Spin [10]. Данный метод в последующем также планируется к внедрению в разрабатываемый комплекс инструментальных средств отладки LuNA-программ.

2. Анализ программы по собранной трассе

Система LuNA состоит из нескольких взаимодействующих модулей и позволяет легко внедрять новые модули, ответственные за какую-либо функциональность. Такой подход к проектированию системы позволил внедрить модуль трассировки WorkflowTracer, записывающий информацию об исполнении программы в файлы трассы. Собранная информация в дальнейшем анализируется утилитой luna_trace [7] с целью обнаружения семантических ошибок, имевших место во время работы программы.

Модуль WorkflowTracer предоставляет интерфейс для уведомления о значимых для последующего анализа событиях, таких как создание и удаление ФВ и ФД, пересылка ФВ между узлами и т. п., и собирает трассу на каждом узле.

По завершении работы программы на каждом вычислительном узле создается три файла трассы со следующей информацией:

1) список ФВ, выполненных на данном узле;
2) список ФД, созданных на данном узле;
3) список ФВ, выполнение которых на данном узле было начато, но не завершилось.

Зависание системы может быть вызвано наличием ошибки использования неинициализированного ФД. В случае же, когда зависание вызвано иными причинами (например, бесконечным циклом в коде атомарного ФВ на C++), пользователь может остановить работу программы, послав сигнал SIGINT (например, используя комбинацию «Ctrl» + «C»). В этом случае вся информация, накопленная в буферах отложенной записи для файлов трасс, будет записана на диск.

В исполнительной runtime-системе LuNA за остановку отвечает модуль IdleStopper, реализующий адаптированную версию алгоритма Дейкстры – Шольтена [11]. Алгоритм оперирует понятиями «наличия» и «отсутствия» работы на вычислительных узлах. Когда на всех узлах не остается работы, система может быть остановлена. В системе LuNA наличие работы на каждом узле определяется счетчиком незавершенных ФВ на данном узле. Если ФВ ожидает данные, то он не учитывается счетчиком. Таким образом, модуль IdleStopper будет считать, что на узле не осталось работы, если на нем либо все ФВ завершены, либо остались только те, которые ожидают данные. Когда на всех узлах не осталось работы, то IdleStopper завершает

программу. По наличию ожидавших данные ФВ WorkflowTracer делает вывод о зависании системы.

Утилита анализа трасс, названная luna_trace, агрегирует трассы, собранные на узлах мультикомпьютера во время работы программы, и производит их анализ для выявления семантических ошибок. Утилита предоставляет информацию не только о факте наличия ошибок как таковых, но, по возможности, и о причинах возникновения ошибок с привязкой к исходным кодам LuNA-программы, а также о возможных способах исправления.

Поиск причин зависаний

Под зависшим ФВ будем понимать ФВ, выполнение которого не может завершиться при текущем состоянии исполнительной системы (например, если нет необходимых ФД). Под причиной зависания понимается зависший ФВ, не имеющий информационных зависимостей от других зависших ФВ.

Для поиска причин зависания строится ориентированный граф информационных зависимостей между зависшими ФВ. Тогда причинами зависаний будут ФВ, соответствующие вершинам, не имеющим исходящих ребер.

В зависимости от опций запуска luna_trace может как вывести информацию обо всех зависших ФВ, так и только о причинах. Возможна ситуация, когда в непустом графе нет ни одной вершины, не имеющей исходящих ребер, т. е. причины зависания выделить невозможно. В этом случае luna_trace выводит информацию обо всех зависших ФВ, а также сообщение о наличии циклических зависимостей между ФВ.

Потенциальные места инициализации данных

Инструментальное средство luna_trace при обнаружении попытки использования неинициализированного ФД выполняет поиск мест в коде программы, где могла быть пропущена инициализация. Этот поиск основан на обнаружении мест инициализации *похожих* ФД.

Пусть существуют множества ФД:

$DF_1 = \{ \textbf{name}[i] \mid i \in I \}$,

$DF_2 = \{ \textbf{name}[j] \mid j \in J, I \cap J = \varnothing \}$.

Пусть ФД из множества DF_1 не были инициализированы во время работы программы, а ФД из множества DF_2 были инициализированы множеством ФВ CF, причем все ФВ из множества CF выполняли один и тот же код (например, операцию инициализации *i*-го индекса ФД в цикле по *i*). Тогда можно предположить, что ФД из множества DF_1 должны были быть инициализированы фрагментами вычислений, выполняющими тот же код, что и ФВ из множества CF. Собранная во время выполнения программы трасса содержит информацию, по которой можно восстановить потенциальные места инициализации в коде.

Поиск случаев множественной инициализации данных

В файлах трассы для создаваемых во время работы программы ФВ имеется информация об идентификаторах входных и выходных ФД. Наличие этой информации позволяет обнаружить повторяющиеся ФД среди выходных, что соответствует ошибке множественной инициализации.

3. Статический анализ на базе AST

Для применения статического анализа необходимо иметь удобное представление исходного кода. В качестве промежуточного представления можно использовать абстрактное синтаксическое дерево (AST), трехадресный код, граф потока управления (CFG) и др. [12]. Различные представления подходят для обнаружения ошибок различных типов. В связи с этим в создаваемый комплекс автоматизированной отладки будут включены 3 разрабатываемых авторами статических анализатора (AST-analyzer, DeGSA – Dependence Graph Static Analyzer, Prolog-analyzer).

Абстрактное синтаксическое дерево строится по известному алгоритму [13]. Это конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены с операторами языка программирования, а листья – с соответствующими операндами.

Другой важной функцией анализа AST является определение структуры программы. Например, можно определить, какие функции присутствуют в программе и какие аргументы они принимают, или выяснить, какие классы определены в коде.

По коду из листинга 1 AST-analyzer строит следующее дерево (не принципиальные элементы опущены) (рис. 1).

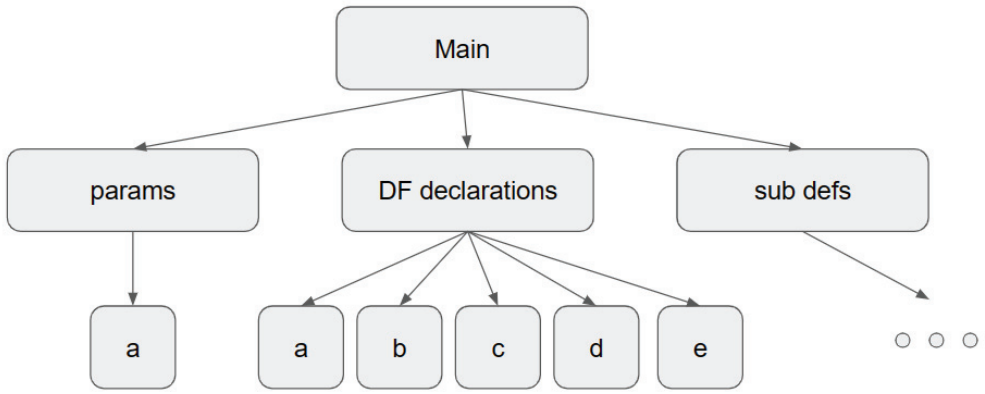

Рис. 1. AST для программы (1)
Fig. 1. AST for program (1)

Здесь корнем дерева является ФК Main, у которого есть параметр *a*, декларации фрагментов данных (5шт.) и вызовы других фрагментов кода.

Далее совершается обход этого AST в глубину с сохранением в контейнер всех параметров и объявлений ФД. После этого линейным поиском выявляются все повторяющиеся декларации.

AST-analyzer способен обнаруживать такие ошибки, как:

1) импорт двух функций под одним именем;
2) отсутствие функции main;
3) ФД с одинаковыми именами в одной области видимости;
4) несовпадение количества аргументов при объявлении ФК и его вызове.

Однако не все ошибки можно обнаружить методом статического анализа по AST. Например, это касается ошибки «Несоответствие типов аргументов при вызове структурированного ФК», поскольку не всегда тип ФД можно определить до запуска программы. Таким образом, если X, Y – ФД и тип хотя бы одного из них не известен, то невозможно определить тип ариф-

метического выражения ($X + Y$, $X * Y$ и т. д.). Возможно, X или Y будут инициализированы строковым типом, и выражение вообще не будет иметь смысл. Поэтому если найдена декларация и вызов фрагмента кода, то при выявлении несовпадения типов параметров анализатор предупредит программиста о возможной ошибке. Другим примером ошибки, для обнаружения которой не подходит такое промежуточное представление, как AST, является попытка использования неинициализированного ФД.

4. Статический анализ на базе графа зависимостей по данным

Одним из вариантов альтернативного промежуточного представления, которое было рассмотрено авторами на предмет возможности применения к системе LuNA, является *граф потока управления* (control-flow graph, CFG) [14]. В традиционных императивных языках программирования (таких как C++ или Java) CFG представляет собой ориентированный граф, вершинами которого являются базовые блоки, а дуги указывают передачу управления от одного базового блока к другому. CFG строится из трехадресного кода, который, в свою очередь, строится из AST.

Однако при построении трехадресного кода LuNA-программы возникают трудности, самой главной из которых является недетерминированность порядка выполнения ФВ. Трехадресный код опирается на конкретный порядок выполнения команд, и, как следствие, CFG тоже этого требует. Это условие делает CFG неприменимым для статического анализа LuNA-программ.

В связи с этим авторами в качестве альтернативы был выбран граф зависимостей по данным [15] (data-dependence graph, DDG), который применяется в различных языках программирования (в частности для оптимизаций) и представляет собой ориентированный граф, где вершинами выступают некоторые операции, а дугами – зависимости операндов операций от результатов вычислений других операций. Такой граф также возможно построить из AST. Авторское инструментальное средство, использующее DDG для обнаружения семантических ошибок в LuNA-программе, носит название DeGSA (Dependence Graph Static Analyzer).

При построении DDG был выбран подход, при котором каждая вершина графа представляет собой оператор, который имеет входные вершины (от которых он зависит) и выходные (которые зависят от него), а также ссылается на все операторы, которые находятся в его теле (далее – «дочерние» вершины).

В качестве примера ниже продемонстрирован DDG для программы из листинга 1 (рис. 2):

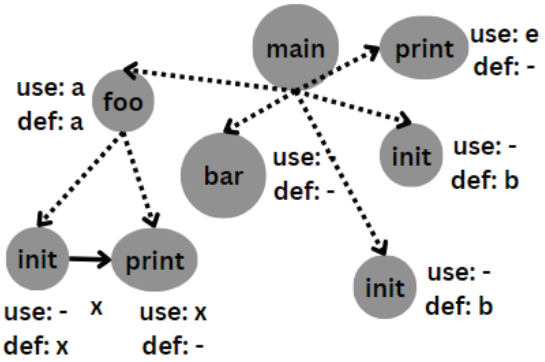

Рис. 2. Граф зависимостей по данным для программы (1)
Fig. 2. Data-dependence graph for program (1)

В этом графе сплошными стрелками указаны дуги зависимостей по данным, а пунктирными стрелками – нахождение одной вершины внутри блока второй (например, *foo* и *bar* находятся внутри блока *main*).

Ниже рассмотрен алгоритм построения и применения DDG в средстве DeGSA, разделяющийся на четыре этапа.

1. Линейный обход всех атомарных ФК и анализ их сигнатур. А именно поиск и сохранение информации о том, какие аргументы при вызове атомарного ФК будут использованы, а какие инициализированы.

2. Создание вершин. Вершина создается только после того, как все операторы в теле текущего оператора были также пройдены, и для них были созданы вершины. Процесс создания различается в зависимости от типа оператора. Для примера далее рассмотрен оператор *cf*, т. е. оператор применения ФК – порождения ФВ. Если ФК является атомарным, то вершина создается сразу. По сигнатуре атомарного ФК можно однозначно определить, является ли каждый из аргументов инициализируемым (имеет тип *name*) или используемым (любой другой тип). Если ФК является структурированным, то функция рекурсивно запускает себя для анализа операторов в теле текущего и порождает дочерние вершины, после чего создает и вершину текущего оператора. Для каждой вершины сохраняется информация, какие именно ФД были использованы и инициализированы в соответствующем ей операторе. Также на этом этапе сохраняется информация о вершинах, содержащихся в блоке текущей вершины (эта связь и указывается на рис. 2 пунктирными стрелками).

3. Рекурсивный обход всех созданных вершин и их связывание. Дуга (сплошная стрелка) от первой вершины до второй создается в случае, если у первой вершины ФД инициализируется, а у второй он же используется (в примере на рисунке в *init* инициализируется ФД *x*, а в *print* – используется).

4. Поиск ошибок.

Для демонстрации возможностей полученного графа ниже рассмотрен поиск двух видов возможных ошибок на примере программы из листинга 1.

1. Повторная инициализация ФД. DeGSA обнаруживает подобные ситуации путем поиска по графу в глубину всех вершин, где ФД указан как используемый. На рис. 2 можно заметить, что от вершины *main* идут две дуги к вершинам *init*, каждая из которых инициализирует ФД *b*, что является ошибкой.

2. Неиспользуемый ФД. Здесь DeGSA ищет вершины, из которых нет исходящей дуги с меткой некоторого инициализируемого ФД. Анализ графа позволяет заметить, что из двух вершин, инициализирующих ФД *b*, не идет никаких дуг, что позволяет заключить, что фрагмент данных *b* не используется и может быть удален.

5. Статический анализ с использованием Prolog

В рамках еще одного разрабатываемого инструмента статического анализа LuNA-программ Prolog-analyzer было решено использовать язык логического программирования Prolog. Использование Prolog позволяет формализовать семантические ошибки в виде лаконичных правил. Еще одним плюсом данного подхода является то, что при проверке этих правил Prolog инстанцирует входящие в правила свободные переменные. Это позволяет получить как информацию о наличии ошибок, так и информацию о том, для каких значений входных ФД найденные ошибки имеют место.

Формат представления LuNA-программы

Существуют работы, в которых языки логического программирования (в частности, Prolog и Datalog) используются для статического анализа [16]. В данной работе статический анализа-

тор по AST LuNA-программы строит набор фактов (часть базы знаний), описывающих инициализацию и использование ФД.

В качестве анализируемых абстракций были выбраны циклы инициализации и использования, а также вызовы атомарных ФК, инициализирующих и использующих одиночные ФД (не содержащие индексов). Для представления LuNA-программы используются следующие факты:

– *df_init_single(InitID, Name)* – инициализация одиночного ФД;

– *df_use_single(UseID, Name)* – использование одиночного ФД;

– *df_init_range(InitId, Name, boundary(InitStart, InitStartType), boundary(InitEnd, InitEndType))* – инициализация диапазона индексов (цикл инициализации) для ФД *Name*, начиная от *InitStart* и заканчивая *InitEnd*;

– *df_use_range(UseId, Name, boundary(UseStart, UseStartType), boundary(UseEnd, UseEndType))* – использование диапазона индексов (цикл использования) для ФД *Name*, начиная от *UseStart* и заканчивая *UseEnd*.

Во всех четырех видах фактов *InitID* и *UseId* – уникальные идентификаторы мест инициализации и использования ФД в программе, а *Name* – имя ФД. Параметры {*Init*|*Use*} {*Start*|*End*} *Type* могут принимать значения «const», «df» и «expr».

В данный момент реализована генерация базы знаний для ограниченного подмножества языка LuNA: поддерживаются вызовы ФК (как атомарных, так и структурированных) с параметрами типа *name* и параметрами примитивных типов, и оператор *for*.

Ниже рассматривается ошибка использования неинициализированных ФД. Правила для поиска двух ее разновидностей, записываемые в код на Prolog:

– *use_of_uninitialized_df(Name, Details)* – использование неинициализированного одиночного ФД;

– *invalid_df_use_range(Name, Details)* – некорректные границы цикла использования ФД.

Name – имя ФД, *Details* – подробная информация об ошибке, например, информация о том, верхняя или нижняя граница цикла использования была некорректной, а также конкретные выражения, использованные в качестве границы.

Выражения, используемые в качестве границ, сравниваются с помощью библиотеки CLP(FD) (Constraint Logic Programming over Finite Domains). Различаются ситуации, когда границы циклов инициализации и использования можно сравнить однозначно (например, *N* и *N + 1*) и когда это невозможно без знания конкретных значений выражений (например, *100* и *N*).

Автоматизация генерации базы знаний и выполнения запросов

Инструментальное средство Prolog-analyzer автоматизирует процесс генерации Prolog-представления LuNA-программы и проверки выполнимости правил для поиска ошибок. Оно реализовано на языке Python как утилита командной строки.

Средство анализа генерирует и сохраняет информацию об использовании ФД (места использования/инициализации). Затем генерируется база знаний на языке Prolog. После к базе знаний выполняются запросы (в частности проверяется выполнимость правил *invalid_df_use_range* и *use_of_uninitialized_df*), и выводится подробная информация об обнаруженных ошибках.

6. Утилита автоматизированного тестирования инструментальных средств отладки LuNA-программ

Для тестирования создаваемых средств отладки LuNA-программ на корректность работы была создана утилита автоматизированного тестирования. Утилита получает на вход директо-

рию, содержащую файл конфигурации *test_config.json*. Конфигурация теста состоит из четырех основных частей:

1) набор команд, которые нужно выполнить перед тестом (поле «arrange»);

2) команда, работа которой тестируется (поле «act»);

3) проверки, которые необходимо выполнить над выводом тестируемой команды (поле «assert»);

4) список имен потоков вывода («stderr», «stdout»), которые нужно сохранить в файлы на диске (поле «save»).

В листинге 2 приводится пример конфигурации теста. В данном случае требуется, чтобы вывод анализатора был пустым, так как в LuNA-программе нет ошибок, которые он мог бы обнаружить.

Листинг 2. Конфигурация теста для программы, не содержащей ошибок

```
0: {
1:  "arrange": [
2:    {
3:      "platform": "Linux",
4:      "command": "./build.sh"
5:    }
6:  ],
7:  "act": {
8:        "command":  "prolog-analyzer  --project-dir=.  --build-
dir=bin",
09:    "timeout": 10
10:  },
11:  "assert": {
12:    "stderr": {"empty": true},
13:    "stdout": {"empty": true}
14:  }
15: }
```

Проверки могут производиться над потоками вывода (stdout) и ошибок (stderr). В качестве проверки можно указать регулярное выражение, которое должно содержаться в выводе; путь к файлу шаблона, содержимое которого должно совпадать с выводом команды; факт того, что в поток не должно ничего выводиться, как в листинге 2.

Для проверки корректности разрабатываемых анализаторов был создан набор тестов, каждый из которых содержит LuNA-программу, скрипт для ее компиляции и файл *test_config.json*. В созданный набор тестов вошли LuNA-программы, содержащие все 22 выделенные ошибки, а также безошибочные программы.

После того как вывод всех средств отладки будет осуществляться в едином JSON-формате (см. п. 7), шаблоны проверок в тестах будут переписаны и утилита автоматизированного тестирования сможет унифицированным образом работать с любым из описанных средств отладки.

7. Интеграция утилит автоматизированной отладки в программный комплекс

Прорабатывается интеграция описанных выше анализаторов в единый комплекс автоматизированной отладки LuNA-программ, способный применять различные подходы к отладке и агрегировать результаты их работы. В целях данной интеграции ведется работа по унификации вывода разработанных анализаторов в JSON-формате, что позволит объединять вывод

и предоставлять пользователю максимально полный и информативный отчет обо всех ошибках в LuNA-программе.

Анализаторы также будут взаимодействовать друг с другом, обмениваясь вспомогательной информацией о найденных и/или возможных ошибках. Примером такого взаимодействия является проверка значений конкретных выражений во время работы программы в случае, когда от конкретного значения будет зависеть наличие или отсутствие ошибочного поведения. Например, Prolog-analyzer устанавливает возможную ошибку несоответствия границ циклов использования и инициализации в зависимости от значения некоторого параметра *N*. luna_trace, в свою очередь, будет проверять факт наличия этой ошибки, поскольку может получить информацию о реальном значении, которое *N* принимало в программе.

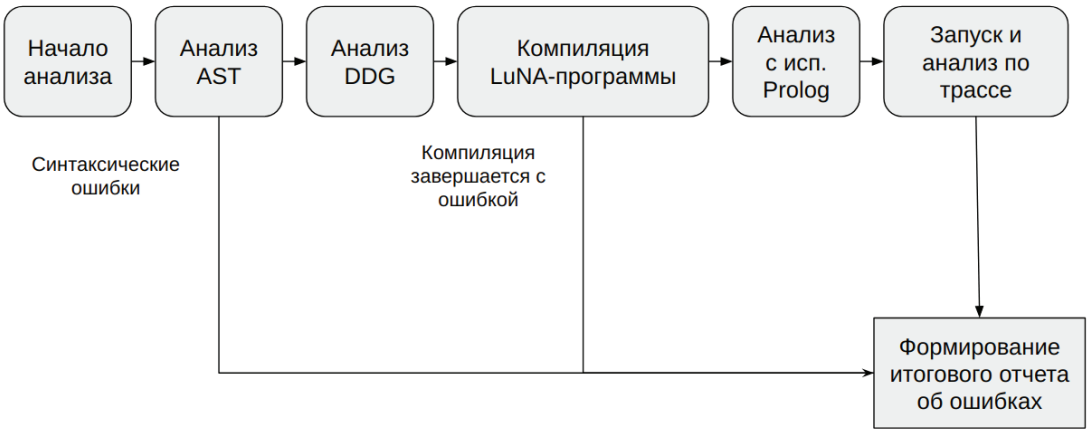

Рис. 3. Схема работы комплекса автоматизированной отладки
Fig. 3. The operation scheme of the automated debugging complex

Схема работы программного комплекса автоматизированной отладки изображена на рис. 3. Сначала запускается AST-analyzer. На данном этапе будут обнаружены такие ошибки, как отсутствие объявления ФК *main*, импортирование нескольких атомарных ФК под одним именем, и другие. Информация об обнаруженных ошибках будет записываться в JSON-файл отчета. Затем запускается DeGSA. На этом этапе будут обнаруживаться более сложные ошибки, такие как повторная инициализация ФД, а также неиспользуемые ФД. Соответственно будет дополнен JSON-файл отчета. Далее LuNA-программа проходит стадии парсинга и препроцессинга с последующей генерацией ее внутреннего представления, используемого компилятором. Данное представление также используется для генерации Prolog-программы третьим статическим анализатором. Автоматизированное выполнение запросов к составленной на следующем шаге базе знаний на Prolog позволит обнаружить еще ряд ошибок, которые дополнят JSON-отчет. Наконец, LuNA-программа будет скомпилирована и запущена, после чего luna_trace произведет анализ трассы и завершит заполнение файла отчета. Затем JSON-файл будет потреблен программным средством, формирующим человекочитаемый отчет о найденных ошибках и являющийся конечным результатом работы создаваемого программного комплекса.

Заключение

Инструментальные средства разрабатываемого программного комплекса автоматизированной отладки LuNA-программ на данный момент способны обнаруживать ряд часто встречающихся ошибок. Однако авторами ведутся работы по расширению функциональных

возможностей данных средств и, в первую очередь, в отношении поиска других ошибок, характерных для LuNA-программ. Благодаря тому что разрабатываемые анализаторы следуют подходу автоматизированной отладки, их становится возможным интегрировать в рамках единого комплекса. А благодаря тому что они используют различные методы (статического и «посмертного» анализа) или различные представления исходного кода, каждый из них подходит для обнаружения своего множества ошибок. Таким образом анализаторы будут дополнять друг друга, формируя единый JSON-файл найденных ошибок. Данный файл будет подаваться на вход средству, формирующему человекочитаемый отчет об ошибках для пользователя.

Для проверки сохранения контроля корректности анализаторов разработана утилита тестирования, включающая тесты на все известные авторам ошибки в LuNA-программах. Благодаря тому что все анализаторы будут выдавать информацию об ошибках в едином формате, утилита сможет однообразно проверять все разрабатываемые инструментальные средства отладки.

В ближайшие планы авторов входит унификация вывода разрабатываемых средств в JSON-формате и создание средства автоматической генерации подробного отчета об ошибках по JSON-файлу. Также программный комплекс будет протестирован на «больших» LuNA-программах численного моделирования для оценки величины накладных расходов.

Список литературы

1. **Malyshkin V., Perepelkin V., Lyamin A.** Trace Balancing Technique for Trace Playback in LuNA System // Malyshkin V. (Eds) Parallel Computing Technologies. PaCT 2023. Lecture Notes in Computer Science. 2023. Vol. 14098. Springer, Cham. P. 4250. DOI: 10.1007/978-3-031-41673-6_4
2. **Малышкин В. Э.** Технология фрагментированного программирования // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2012. № 46 (305).С. 4555. DOI: 10.14529/cmse120104
3. **Зубов М. В., Пустыгин А. Н., Старцев Е. В.** Применение универсальных промежуточных представлений для статического анализа исходного программного кода // Доклады Томского гос. ун-та систем управления и радиоэлектроники. 2013. № 1 (27). С. 64–68.
4. **Flederer F., Ostermayer L., Seipel D., Montenegro S.** Source Code Verification for Embedded Systems using Prolog // Electronic Proceedings in Theoretical Computer Science. 2016. Vol. 234. P. 88–103. DOI: 10.4204/EPTCS.234.7
5. **Abramson D. A., Watson G., Le P. D.** Guard: a tool for migrating scientific applications to the .NET Framework // Sloot P. M. A., Tan C. J. K., Dongarra J. J., & Hoekstra A. G. (Eds.) Computational Science – ICCS 2002: International Conference Amsterdam, The Netherlands, April 21–24, 2002 Proceedings. P. 834843. (Lecture Notes in Computer Science; Vol. 2330). Springer. DOI: 10.1007/3-540-46080-2_88
6. **Bakhtin V. A., Kataev N. A., Kolganov A. S., Yakobovskiy M. V., Zaharov D. A.** Automation of programming for promising high-performance computing systems // Malyshkin V. (eds) Parallel Computing Technologies. PaCT 2023. Lecture Notes in Computer Science. Cham: Springer, 2023. Vol. 14098. P. 3–17.
7. **Malyshkin V., Vlasenko A., Michurov M.** Automated Debugging of Fragmented Programs in LuNA System // Proc. of Mathematical Modeling and Supercomputer Technologies 2022. Communications in Computer and Information Science. 2022. Vol. 1750. P. 266–280. DOI: 10.1007/978-3-031-24145-1_22
8. **Protze J., Hilbrich T., Schulz M., de Supinski B. R.** MPI Runtime Error Detection with MUST: A Scalable and Crash-Safe Approach // Proc. of 43rd International Conference on

Parallel Processing Workshops. Minneapolis, MN, USA, 2014, p. 206–215. DOI: 10.1109/ICPPW.2014.37
9. **Карпов Ю. Г.** MODEL CHECKING. Верификация параллельных и распределенных программных систем. СПб.: БХВ-Петербург, 2010. 560 с.
10. **Holzmann G. J., Joshi R.** Model-Driven Software Verification // Graf S., Mounier L. (Eds) Model Checking Software. Proc. of SPIN 2004. Lecture Notes in Computer Science. Vol. 2989. Springer, Berlin, Heidelberg. P. 76–91. DOI: 10.1007/978-3-540-24732-6_6
11. **Fokkink W.** Distributed Algorithms, second edition: An Intuitive Approach. MIT Press, 2018.272 p.
12. **Касьянов В. Н., Евстигнеев В. А.** Графы в программировании: обработка, визуализация и применение. СПб.: БХВ-Петербург, 2003. 1104 с.
13. **Белеванцев А. А.** Многоуровневый статический анализ исходного кода программ для обеспечения качества программ // Программирование. 2017.Т. 43, № 6. С. 3–25.
14. **Krinke J.** Information Flow Control and Taint Analysis with Dependence Graphs. Vancouver: Proc. of 3d International Workshop on Code Based Software Security Assessments (CoBaSSA). 2007. P. 6–9.
15. **Kuck D. J., Kuhn R. H., Padua D. A., Leasure B., Wolfe M.** Dependence graphs and compiler optimizations // Proc. of the 8th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL). 1981. P. 207–218. DOI: 10.1145/567532.567555
16. **Smaragdakis Y., Bravenboer M.** Using Datalog for Fast and Easy Program Analysis // de Moor O., Gottlob G., Furche T., Sellers A. (Eds) Datalog Reloaded. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2010. Vol. 6702. P. 245–251. DOI: 10.1007/978-3-642-24206-9_14

References

1. **Malyshkin V., Perepelkin V., Lyamin A.** Trace Balancing Technique for Trace Playback in LuNA System. In: *Malyshkin, V. (eds) Parallel Computing Technologies. PaCT 2023. Lecture Notes in Computer Science*, 2023, vol. 14098. Springer, Cham., pp. 42–50. DOI: 10.1007/978-3-031-41673-6_4.
2. **Malyshkin V. E.** Technology of fragmented programming. *Vestnik YuUrGU. Series: Computational Mathematics and Software Engineering*, 2012, no. 46 (305), pp. 45–55 (in Russ.). DOI: 10.14529/cmse120104
3. **Zubov M. V., Pustygin A. N., Starcev E. V.** The use of universal intermediate representations for static analysis of the source code. *Reports of Tomsk State University of Control Systems and Radioelectronics*, 2013, no. 1(27), pp. 64–68.
4. **Flederer F., Ostermayer L., Seipel D., Montenegro S.** Source Code Verification for Embedded Systems using Prolog. *Electronic Proceedings in Theoretical Computer Science*, 2016, vol. 234, pp. 88–103. DOI: 10.4204/EPTCS.234.7
5. **Abramson D. A., Watson G., Le P. D.** Guard: a tool for migrating scientific applications to the .NET Framework. In *P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, & A. G. Hoekstra (Eds.) Computational Science – ICCS 2002*: International Conference Amsterdam, The Netherlands, April 21–24, 2002 Proceedings, pp. 834843. (Lecture Notes in Computer Science; Vol. 2330). Springer. DOI: 10.1007/3-540-46080-2_88
6. **Bakhtin V. A., Kataev N. A., Kolganov A. S., Yakobovskiy M. V., Zaharov D. A.** Automation of programming for promising high-performance computing systems. In: *Malyshkin V. (eds) Parallel Computing Technologies. PaCT 2023. Lecture Notes in Computer Science*, Cham: Springer, 2023, vol. 14098, pp. 3–17.
7. **Malyshkin V., Vlasenko A., Michurov M.** Automated Debugging of Fragmented Programs in LuNA System. *Proc. of Mathematical Modeling and Supercomputer Technologies 2022.*

Communications in Computer and Information Science, 2022, vol. 1750, pp. 266–280. DOI: 10.1007/978-3-031-24145-1_22

8. **Protze J., Hilbrich T., Schulz M., de Supinski B. R.** MPI Runtime Error Detection with MUST: A Scalable and Crash-Safe Approach. *Proc. of 43rd International Conference on Parallel Processing Workshops*. Minneapolis, MN, USA, 2014, p. 206–215. DOI: 10.1109/ICPPW.2014.37
9. **Karpov J. G.** MODEL CHECKING. Verification of parallel and distributed software systems. Saint Petersburg, BHV-Peterburg publ., 2010, 560 p. (in Russ.)
10. **Holzmann G. J., Joshi R.** Model-Driven Software Verification. In: *Graf S., Mounier, L. (eds) Model Checking Software. Proc. of SPIN 2004. Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2004, vol. 2989, p. 7691. DOI: 10.1007/978-3-540-24732-6_6
11. **Fokkink W.** Distributed Algorithms, second edition: An Intuitive Approach. MIT Press, 2018, 272 p.
12. **Kas'janov V. N., Evstigneev V. A.** Graphs in programming: processing, visualization and application. Saint Petersburg, BHV-Peterburg publ., 2003, 1104 p. (in Russ.)
13. **Belevancev A. A.** Multilevel static analysis of the source code to ensure the quality of programs. *Programming and Computer Software*, 2017, vol. 43, no. 6, p. 325. (in Russ.)
14. **Krinke J.** Information Flow Control and Taint Analysis with Dependence Graphs. Vancouver: Proc. of 3d International Workshop on Code Based Software Security Assessments (CoBaSSA), 2007, p. 69.
15. **Kuck D. J., Kuhn R. H., Padua D. A., Leasure B., Wolfe M.** Dependence graphs and compiler optimizations. *Proc. of the 8th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, 1981, p. 207218. DOI: 10.1145/567532.567555
16. **Smaragdakis Y., Bravenboer M.** Using Datalog for Fast and Easy Program Analysis. In: de Moor O., Gottlob G., Furche T., Sellers A. (eds) Datalog Reloaded. Lecture Notes in Computer Science.. Springer, Berlin, Heidelberg, 2010, vol. 6702, p. 245251. DOI: 10.1007/978-3-642-24206-9_14

Сведения об авторах

Власенко Андрей Юрьевич, кандидат технических наук, доцент кафедры параллельных вычислений ФИТ НГУ, научный сотрудник лаборатории синтеза параллельных программ ИВМиМГ СО РАН

Мичуров Михаил Антонович, ассистент кафедры параллельных вычислений ФИТ НГУ

Царёв Василий Дмитриевич, студент ФИТ НГУ

Курбатов Максим Андреевич, студент ФИТ НГУ

Information about the Authors

Andrey Yu. Vlasenko, Candidate of Technical Sciences, Associate Proffessor of the Parallel Computations Department, Novosibirsk State University (Novosibirsk, Russian Federation)

Mihail A. Michurov, Assistant of the Parallel Computations Department, Novosibirsk State University (Novosibirsk, Russian Federation)

Vasiliy D. Tsarev, Student of the Faculty of Information Technologies, Novosibirsk State University (Novosibirsk, Russian Federation)

Maksim A. Kurbatov, Student of the Faculty of Information Technologies, Novosibirsk State University (Novosibirsk, Russian Federation)

Статья поступила в редакцию 29.12.2023;
одобрена после рецензирования 08.02.2024; принята к публикации 08.02.2024

The article was submitted 29.11.2023;
approved after reviewing 08.02.2024; accepted for publication 08.02.2024