ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ №4(35), 2017, с. 221–244

УДК 519.685+519.682.2+004.032.22

А. И. Адамович, Анд. В. Климов

Как создавать параллельные программы, детерминированные по построению? Постановка проблемы и обзор работ

Аннотация. Одна из основных проблем, делающих параллельное программирование ненадежным, трудозатратным, подверженным ошибкам, а программы трудно отлаживаемыми, — недетерминированность процессов и результатов вычислений, когда несколько исполнений одной программы с одинаковыми входными данными могут выдавать разные результаты из-за другого порядка взаимодействия параллельных процессов. В связи с бурным ростом сложности программ для суперкомпьютеров, в последнее десятилетие приобретает популярность и становится всё более актуальной идея параллельных вычислений с детерминированностью, гарантированной языком и системой программирования.

В статье анализируется проблема, как сделать параллельное программирование как можно более детерминированным, и дается обзор некоторых подходов к ее решению. Также обсуждается задача разработки системы, предоставляющей возможность писать как детерминированный, так и недетерминированный код с гарантиями прикладному программисту, что его программа будет детерминированной.

Ключевые слова и фразы: модели параллельных вычислений, детерминированные программы, функциональное программирование, объектно-ориентированное программирование.

Введение

Одна из фундаментальных причин, почему параллельное программирование существенно сложнее последовательного, к которому привыкли разработчики за десятилетия развития информационных технологий, — то, что оно в общем случае недетерминированное: несколько исполнений программы с одними и теми же исходными данными могут давать различные результаты из-за различающегося порядка

Исследование выполнено в рамках НИР (госзадание ФАНО РОССИИ, регистрационный номер АААА-А17-117040610375-5)[1].

Исследование выполнено при поддержке РФФИ (проект № 16-01-00813-а)[2].

DOI: 10.25209/2079-3316-2017-8-4-221-244

вычислений и взаимодействия параллельных процессов. Это делает ошибки трудно воспроизводимыми, отладку намного более сложной, а созданные программы менее надежными, трудно сопровождаемыми и изменяемыми.

Параллельное программирование[1] принципиально не может полностью избавиться от недетерминированности, так как соответствующие средства программирования — процессы, «потоки» (*англ.* threads), их взаимодействие через общий ресурс — требуются для эффективной реализации на современной аппаратуре, а также из-за распределенного характера приложений, функционирующих в реальном мире.

Несмотря на это, можно и нужно стараться как можно больше увеличить долю детерминированного кода, значительно упрощая отладку и дальнейшее сопровождение приложения. Программисты часто решают эту задачу руками (и мозгами), например, разрабатывая приложения средствами MPI, вставляют барьеры, чтобы получать понятные состояния после барьеров.

Возникает вопрос: насколько эту деятельность можно автоматизировать?

В статье дается обзор известных авторам подходов и программных инструментов (языков, сред выполнения программ), гарантирующих детерминированность процессов и результатов вычислений, а также представлено возможное направление их дальнейшего развития и интеграции, в рамках которого авторы проводят и свои работы [**1**–**3**].

Симптоматично название одной из недавних статей: «Параллельное программирование должно быть по умолчанию детерминированным» («Parallel Programming Must Be Deterministic by Default») [**4**]. Мы разделяем основной тезис ее авторов.

Дальнейший материал этой статьи состоит из двух основных частей:

---

[1] Мы не противопоставляем понятия «parallel» и «concurrent» так, как это принято в англоязычной литературе, и используем слово «параллельной» для всей области, которую «там» именуют «parallel *and* concurrent». По нашим наблюдениям, в русскоязычной литературе это разделение проводится, как правило, только если эту нужно для конкретного обсуждения, и более того, для слова «concurrent» не сложилось благозвучного перевода, кроме как вызывающего побочные ассоциации «конкурентный».

- в разделе 1 дается обзор известных нам методов, языков и систем, в которых ставится и как-то решается задача обеспечения детерминированности программ. Обзор начинается с чисто функционального программирования, допускающего, благодаря отсутствию побочных эффектов, параллельную реализацию с сохранением детерминированной семантики. Затем переходим к языкам с понятиями процесса и побочного эффекта и отмечаем работы, разрабатывающие методы статического анализа по выявлению детерминированности программы. Обзор заканчивается достаточно свежей работой, разрабатывающей тему детерминированности процессов, взаимодействующих через переменные, монотонно изменяющиеся на некоторой решетке;
- в разделе 2 обсуждается задача разработки системы, предоставляющей возможность писать как детерминированный, так и потенциально недетерминированный код.

В заключении резюмируются основные тезисы статьи.

1. Обзор работ по детерминированным параллельным вычислениям

Идея противопоставления детерминированных и недетерминированных параллельных вычислений достаточно старая и восходит к 1960–70-ым годам, когда зарождались основные идеи моделей вычисления и парадигм программирования. Лозунг скорого перехода к массовому параллельному программированию и проблема поиска подходящих программных средств громко звучали уже в 1970-е годы. Но до начала 2000-х годов, пока благодаря закону Мура промышленное программирование удовлетворялось языками, ориентированными на последовательные вычисления со сравнительно небольшим объемом кода, связанным с организацией взаимодействия процессов и параллелизма, эти работы были не столь актуальными, как сейчас. Отметим, что нетривиальные параллельные архитектуры 1970–80-ых годов практически умерли (например, Manchester Data Flow Computer [5], Connection Machine, Transputer), хотя их идеи разошлись, и не исключено их возрождение на новом витке технологий.

В 2000-е годы и особенно в последнее десятилетие наблюдается взрывной рост числа работ по параллельным вычислениям, включая разработку средств детерминированного программирования. Перечислим основные выявленные нами подходы.

1.1. Детерминированный параллелизм функциональных языков

Языки и модели вычислений без побочных эффектов (типа функциональных) или с ограниченными побочными эффектами (например, логическое программирование), не мешающими параллельному выполнению большой части кода программы, являются основой всех детерминированных моделей вычислений. Достаточно широкое использование функционального программирования показывает, что детерминированный параллелизм не является слишком узкой областью и оправдана задача его расширения за пределы чисто функционального и логического программирования с сохранением их полезных свойств.

Этот подход был популярен с зарождения функциональных и логических языков 1970–80-ые годы. Для многих из них разрабатывались хотя бы исследовательские реализации с неявным параллелизмом и «автоматическим» распараллеливанием (кавычки, поскольку и там есть проблемы, например, чрезмерное измельчение гранул параллелизма и, как следствие, плохо управляемый, взрывной рост числа параллельных процессов).

Среди большого разнообразия публикаций по этому направлению особенно интересен сборник, подводящий итоги по состоянию на конец 1990-х годов [6]. Из современных работ следует отменить, что интенсивно ведутся исследования для языка Haskell по расширению самого языка и его реализации средствами параллелизма, не нарушающими «функциональную чистоту» языка[2]. Авторы данной статьи, имея большой опыт разработки систем функционального программирования, также отдали дань этому подходу и в те времена разрабатывали системы на близких принципах [7, 8].

В рамках функционального подхода появились и специфические модели вычислений, такие как Data Parallel Computation (от зарождения [9] до современных реализаций, например, в языке Haskell[3]) и модель Data Flow с чисто функциональным языком Id [10].

Не выходя за рамки функциональной модели вычислений, строятся проблемно-ориентированные декларативные языки для конкретных областей применения. Например, таким является язык Норма [11, 12] для решения сеточных задач некоторого класса из математической

[2] https://wiki.haskell.org/Parallel
[3] https://wiki.haskell.org/GHC/Data_Parallel_Haskell

физики. Специализированные конструкции этих языков и отсутствие понятий процесса и побочного эффекта дают возможность осуществлять весьма глубоких анализ и преобразования таких программ вплоть до эффективного кода для разнообразных архитектур современных процессоров [12].

Из функционального подхода возникли и некоторые конструкции детерминированных параллельных вычислений в современных языках. Примером может служить параллельный вызов подпрограмм, называемый в ряде языков «future» (например, в языке X10 фирмы IBM[4]). Другой пример — шаблон программирования Map-Reduce, ставший особенно знаменитым из-за применения фирмой Google в своем поисковике и в системах обработки «больших данных» [13].

1.2. Неизменяемость

В императивных и объектно-ориентированных языках функциональному программированию соответствует ограничение на язык (или стиль программирования), когда объекты не изменяются после создания и инициализации. В этом случае, с точки зрения распараллеливания, сохраняются все хорошие свойства функциональных языков. Неизменяемые объекты являются одним из тех неотъемлемых свойств большинства функциональных языков, которые обеспечивают изящество и простоту реализации параллелизма, основанного на использовании функционального подхода к программированию.

Поэтому неизменяемые (*англ.* immutable) переменные — как простые переменные, так и структуры, а также ссылки только-на-чтение (*англ.* read-only references) — уже в течение некоторого периода времени находятся в фокусе внимания исследователей и разработчиков. Например, Джошуа Блох во втором издании своей книги «Java. Эффективное программирование» [14] посвятил отдельный раздел рассуждениям о необходимости в максимально возможной степени отказаться от изменяемости данных. Среди преимуществ, обеспечиваемых неизменяемыми объектами, Блох отмечает их простоту, влекущую, как следствие, безопасность в отношении потоков (*англ.* thread-safe) и эффективность при совместном доступе.

Не удивительно поэтому, что в последние годы, когда многоядерные параллельные аппаратные архитектуры вычислительных систем стали скорее правилом, чем исключением, предпринимаются усилия по внедрению концепции неизменяемых переменных и в императивные языки программирования.

[4] http://x10-lang.org/

Одним из примеров таких усилий является включение понятия неизменяемости в язык программирования Rust [15]. В этом языке переменные являются неизменяемыми по умолчанию, а для того, чтобы значения переменных можно было изменять, такие переменные должны быть описаны с использованием ключевого слова `mut`. В язык Rust включены также тщательно проработанные средства управления изменяемостью переменных на основе понятия владения (*англ.* ownership) и механизма заимствования (*англ.* borrowing).

Внедрению концепции неизменяемости в те объектно-ориентированные императивные языки программирования, которые практически используются более-менее значительным числом разработчиков, предшествовало значительное число научных публикаций на данную тему.

В работе [16] упоминается целый спектр вариантов неизменяемости. *Неизменяемый объект* не может быть изменен. Если все объекты некоторого класса являются неизменяемыми, то этот класс называется неизменяемым. *Ссылка только-на-чтение* не может быть использована для изменения того объекта, на который она указывает. *Чистые методы* не имеют побочных эффектов, видимых из участков кода, находящихся вовне их вызовов[5]. *Неприсваиваемость* (*англ.* non-assignability) является слабым частным случаем неизменяемости; для описания неприсваиваемых переменных и полей в языке Java используется ключевое слово `final`. *Глубокая* (*англ.* deep) неизменяемость подразумевает рекурсивность этого свойства — в отличие от поверхностной (*англ.* shallow); то же относится и к ссылкам только-на-чтение. И, наконец, можно говорить об *неизменяемости представления* (*англ.* representation immutability) и *абстрактной неизменяемости* (*англ.* abstract immutability). Во втором случае конкретное представление объекта в памяти может быть изменено, без изменения свойств объекта, видимых из остальной программы. Например, в языке Java в экземпляре класса String есть поле «хеш-код», которое инициализируется (изменяется) в какой-то момент жизни этого объекта. При этом абстрактное значение данного экземпляра — сама символьная строка — не меняется.

Аналогичная — хотя и не во всем совпадающая с приведенной — классификация вариантов неизменяемости приводится в работе [17]. Интересным отличием данной классификации является упоминание о

[5] Это является более сильной гарантией неизменяемости, чем описание всех параметров метода как рекурсивно неизменяемых ссылок только-на-чтение.

возможности во время инициализации структуры данных *ослабить* (*англ.* relax) ограничения на изменяемость, а по завершении процесса инициализации — восстановить (*англ.* enforce). Эта особенность полезна для инициализации циклических структур данных.

В целом, статья [16] содержит обширный обзор работ, относящихся к неизменяемости, в том числе и сравнение целого ряда языков программирования, реализующих данное понятие. Работа [17] может служить в качестве ее дополнения, как содержащая более поздние сведения.

Приведем несколько примеров исследований, посвященных реализации понятия неизменяемости в языках программирования. В статье [18] описан язык Joe_3, в котором понятие неизменяемости реализуется с использованием концепции владения. Авторы характеризуют Joe_3 как основанный на классах объектно-ориентированный язык с синтаксисом, понятным читателю, имеющему представление о понятии *типов владения* (*англ.* ownership types) и знакомому с Java-подобными языками. В чуть более поздней статье [19] представлен язык OIGJ, являющийся расширением языка Java реализациями понятий владения и неизменяемости. Работа [20] посвящена Glacier — системе аннотации типов для языка Java, направленной на расширение данного языка реализацией понятия неизменяемости. Интересной особенностью данной работы являются приведенные в ней сведения об исследовании, посвященном удобству данного программного средства для практических целей.

В целом, анализ публикаций, посвященных реализации понятия неизменяемости, позволяет говорить о том, что уже в самое ближайшее время данное понятие может стать обычным для широко распространенных языков программирования и облегчить распараллеливание программ.

1.3. Статические методы обеспечения детерминированности

Разрешая побочный эффект вместе с параллелизмом, можно добиваться детерминированности, вводя определенные ограничения, проверяемые компилятором. Это удобно делать, расширяя систему типов понятиями, связанными с наличием/отсутствием побочных эффектов и их областью действия. Такие системы типизации называются «effect systems»[6].

[6] https://en.wikipedia.org/wiki/Effect_system

В данной области уже выполнен (и продолжается) ряд теоретических работ по специально расширенным лямбда-исчислениям, но больше всего заслуживает внимания попытка довести этот подход до практики в рамках Java-подобного языка группой из Иллинойского университета, опубликовавшей в 2009 году статью-манифест с ярким названием «Параллельное программирование должно быть детерминированным по умолчанию» [4] и разрабатывающей соответствующий язык DPJ (Deterministic Parallel Java)[7] [21].

Еще одна интересная серия работ по детерминированному параллелизму с помощью систем типизации с эффектами выполняется в University of California San Diego [22]. В ней используются зависимые типы (*англ.* dependent types, refinement types) следующего специального вида: либо $\nu = 0$, либо $\nu < n$, либо $\nu > n$, где ν — переменная, на которую накладывается ограничение, n — целая константа. Оказывается, вывод и проверка типов в такой системе достаточны для определения независимости побочных эффектов над массивами и выявления детерминированности при распараллеливании некоторого практического класса программ. Такая система типов названа ее авторами «liquid types» созвучно словосочетанию «logically qualified types» (а отнюдь не означает «жидкие типы»).

Сюда же можно отнести и большую линию работ по методам статического анализа параллельных (точнее — concurrent) программ на предмет отсутствия «гонок» между процессами, имеющими доступ к общим данным. Это большая специальная область, выходящая за пределы данной статьи.

1.4. Обеспечение детерминированности с помощью операций над данными

Следующий подход к обеспечению детерминированности параллельных программ, в отличие от методов предыдущего раздела, не использует никаких статических средств анализа. За основу берется чисто функциональный язык программирования, быть может, со всевозможными расширениями, не нарушающими функциональность и распараллеливаемость, например, статическое однократное присваивание. Затем для взаимодействия параллельных процессов предоставляются специальные структуры данных с так определенными операциями, чтобы не нарушалась детерминированность. Оказывается, это не только возможно, но такая идея породила целое направление.

Отметим самые интересные (на наш взгляд) из этих работ:

[7] http://dpj.cs.illinois.edu/DPJ/

- I-структуры (*англ.* I-structures) [**10**, **23**];
- сети Кана (*англ.* Kahn networks) [**24**], TStreams, Concurrent Collections [**25**];
- структуры данных, основанные на решетках (*англ.* lattice-based data structures) [**26**, **27**].

В данной категории самыми простыми являются I-структуры (*англ.* I-structures) [**23**]. Они определяются как ячейки памяти, имеющие в начальный момент значение «не определено», а потом лишь один раз принимающее значение. Повторная запись равного значения допустима и не меняет состояние, а в случае присваивания неравного значения возникает исключение (ошибка). Попытка чтения неопределенного значения блокирует процесс до момента присваивания. В оригинальных работах [**10**, **23**] эти ячейки являются элементами массивов, которые, в свою очередь, также могут быть элементами массивов. Однако такие ячейки могут быть и полями объектов и других структур данных. Аппаратная поддержка операций над I-структурами присутствует в классических dataflow-компьютерах [**5**] и позднее использовалась в некоторых как потоковых, так и универсальных архитектурах [**28**]. Доказано, что параллельные программы, состоящие из процессов, взаимодействующих только через I-структуры, являются детерминированными.

Взаимодействие процессов через каналы (*англ.* channels) или потоки (*англ.* streams), в случае, когда запись разрешена лишь одному процессу, также гарантирует детерминированность. Считается, что это направление началось с сетей Кана (*англ.* Kahn networks) [**24**]. Последняя из наиболее известных систем такого рода — Concurrent Collections [**25**]. Специальными данными, над которыми строятся детерминированные параллельные вычисления, здесь являются *коллекции* (*англ.* collections) трех видов: *шага* (*англ.* step collections), *данных* (*англ.* data collections) и *управления* (*англ.* control collections). В этих терминах специфицируются графы вычислений в стиле потоков данных. Каждый элемент коллекции (называемый здесь экземпляром, *англ.* instance) изменяется монотонно однократным присваиванием от неопределенного состояния к определенному, гарантируя детерминированность.

У приведенных выше подходов есть общая черта: переменные, объекты, через которые осуществляется взаимодействие параллельных процессов, изменяют свое состояние монотонно, только вверх на

некоторой полурешетке[8] от неопределенного состояния к «всё более определенному». При этом верхний элемент решетки ($\top$) обозначает «переопределено»; в программе это соответствует ошибке, выработке исключения. Например, ячейки I-структур с целыми числами в качестве значений, описывается решеткой, называемой «плоской», состоящей из нижнего элемента «не определено» ($\bot$), не сравнимых между собой целых чисел и верхнего элемента «переопределено» ($\top$). При выполнении операции присваивания значения y в переменную со значением x, в нее записывается наименьшая верхняя грань значений x и y. Если полученный результат оказывается верхним элементом $\top$, то вырабатывается исключение.

Эта идея в общем виде была проработана в диссертации Lindsey Kuper [**26**] и в публикациях вместе с ее коллегами [**27**, **29**, **31**]. Она доказала детерминированность параллельных вычислений для процессов, взаимодействующих через переменные, принимающие значения из произвольной (полу)решетки.

В такой модели вычислений, основанной на решетках, есть тонкость: когда операция чтения `get` из переменной может выдать значение, а не быть заблокированной в ожидании следующего его уточнения? Значение x можно считать готовым и его следует выдать, когда между ним и верхним элементом решетки $\top$ нет промежуточных значений. Действительно, если позднее в переменную будет присвоено значение, не сравнимое с x, переменная перейдет в состояние $\top$, будет выработано исключение, и тогда будет все равно, было ли выдано значение x и какое. Если же таких присваиваний не будет, то вычисления продолжатся, используя данное значение x.

Если же значение переменной еще может изменится вверх, но не стать $\top$, его можно прочитать и использовать в дальнейших вычислениях, только если каким-то образом известно, что присваиваний в данную переменную больше не будет и ее значение достигло неподвижной точки. Такое состояние, называемое «покоем» (*англ.* quiescense), в общем случае нелегко обнаружить.

---

[8] `https://en.wikipedia.org/wiki/Semilattice`. Точнее говоря, используется полурешетка с верхним ($\top$) и нижним ($\bot$) элементами. Решетка не требуется, так как изменение значений переменных происходит в одну сторону снизу вверх. В дальнейшем для краткости мы говорим «решетка» вместо «полурешетка».

В диссертации [**26**] предложено два способа выявления покоя для программ с ослабленной формой детерминированности — *квазидетерминированных* (*англ.* quasi-deterministic). Так названы программы, которые могут завершаться недетерминированно либо выработкой исключения, либо выдачей всегда одного и того же значения.

Первый способ — принудительная «заморозка» переменной на текущем значении операцией `freeze`, запрещающей дальнейшие изменения переменной и вызывающей исключение в будущем при попытке присвоить другое значение. После выполнения `freeze` операции чтения `get` из этой переменной всегда будут выполняться без блокировки.

Второй способ основан на использовании обработчиков (*англ.* handlers), которые программа должна зарегистрировать на переменной и которые вызываются при ее изменении. Считается, что когда у переменной не активен ни один обработчик, неподвижная точка и покой достигнуты. Чтобы дождаться состояния покоя переменной, вызывается операция `quiesce`.

В диссертации [**26**] предложены и другие интересные применения и расширения модели параллельных вычислений, основанной на решетках. В частности, при распределенных вычислениях, переменные со значениями на решетках можно копировать в несколько узлов, изменять независимо, а потом объединить полученные значения, вычислив их наименьшую грань. Здесь используется техника *сходящихся реплицированных типов данных* (*англ.* convergent replicated data types, CvRDT) [**30**].

Работы [**26**, **27**, **29**, **31**] показывают, что идея использования структур данных, основанных на решетках, весьма плодотворна и ее следует развивать дальше для расширения класса параллельных программ, детерминированных по построению.

В последние годы наблюдается взрывной рост работ по детерминированным параллельным вычислениям, и в этом обзоре отражены далеко не все из них.

2. Постановка задачи разработки системы детерминированного параллельного программирования

Как видно из приведенного выше в разделе 1 обзора, задача разработки языков и систем детерминированного параллельного программирования еще далека от полноценного решения и работы находятся на исследовательской стадии. Из-за сложности параллельного программирования идеальная среда разработки (*англ.* integrated development

environment, IDE) должна предоставлять различные языки и инструменты, с одной стороны, для разработчиков приложений (у которых одна забота — выразить содержательную логику, функциональность программы в удобных детерминированных терминах), а с другой стороны, для системных программистов, экспертов в параллельном программировании (которые смогут реализовывать базовые понятия, программные компоненты, пользуясь всем арсеналом недетерминированного параллельного программирования).

Отметим также точку зрения авторов Concurrent Collections [**25**]. Они выделяют еще одну роль, поддержанную их инструментами, — эксперт по настройке (*англ.* tuning expert), который отображает спецификацию программы, разработанную прикладным экспертом (*англ.* domain expert), в эффективный код для конкретной архитектуры, сохраняя эквивалентность исходной спецификации и кода программы. При этом язык и среда разработки должны автоматически проверять отсутствие формальных ошибок при таком отображении. Мы согласны, что эта роль действительно важна, поскольку не следует рассчитывать, что в обозримом будущем могут быть построены системы автоматического отображения параллельных программ достаточно широкого класса в эффективный низкоуровневый код. Однако, мы ее сейчас не рассматриваем, но полагаем, что решение обсуждаемой здесь задачи поможет продвинуться и в разработке автоматизированных инструментов для построения эффективных параллельных программных реализаций с гарантиями эквивалентности исходным спецификациям.

2.1. Анализ существующих методов

Методы, представленные выше, подразделяются на два подхода:

(*1*) расширения функциональных языков операциями над новыми данными с сохранением детерминированности (этому посвящены все разделы обзора, кроме раздела 1.3);
(*2*) ограничения императивных языков статическими методами до детерминированного параллелизма (раздел 1.3).

Второму подходу в нашем обзоре уделено меньше внимания, так как нам кажется, что он еще не настолько разработан, чтобы получить применение для достаточно широкого класса программ. Естественно предположить, что прорыв произойдет после включения в массовые языки программирования средств записи утверждений и

доказательств о программах, а также средств проверки и поиска доказательств. Детерминированность — это лишь одно из нетривиальных свойств, и вряд ли удастся разработать для него специализированные, более эффективные и удобные средства, чем для верификации программ в общем виде. В настоящее время бурно развиваются и уже на подходе к практике расширения системы типизации языков *зависимыми типами* (*англ.* dependent types)[9], позволяющими изображать любые конструктивные свойства программ, включая эквивалентность результатов при любом порядке вычислений. Эти методы окажут решающее влияние на все методы и инструменты программирования, в том числе на резкое повышение надежности параллельного программирования. Но пока это дело будущего.

Уже имеющиеся достижения в рамках первого подхода более прагматичны: сохранять детерминированность функциональных программ легче, чем выявлять ее и проверять в императивных. Заметим, что нынешние системы этого типа имеют следующую общую структуру. Их авторы разрабатывают библиотеки «нижнего уровня» для использования прикладными программистами на языке высокого уровня (например, Haskell), но сами реализуют их на языках системного программирования, таких как C/C++, или с использованием небезопасных (*англ.* unsafe) средств реализации языка Haskell. Таким образом, наблюдается большой разрыв между двумя уровнями разработки детерминированных программ.

Если бы существовал конечный набор понятий и конечный объем кода таких библиотек, которых хватило бы надолго для эффективной реализации достаточно большого класса прикладных задач, то эту ситуацию можно было бы считать приемлемой. Однако есть все основания полагать, что такого конечного набора понятий и базовых библиотек не существует. Поэтому надо рассчитывать на то, что библиотеки будут постоянно развиваться вместе с новыми классами прикладных задач и новыми компьютерными архитектурами, в которые нужно эффективно отображать программы.

2.2. Двухуровневый подход к обеспечению детерминированности

На основе анализа, проведенного выше в разделе 2.1, можно сделать вывод о том, что *детерминированное параллельное программирование принципиально является двухуровневым*. Причем если

[9] См. таблицу языков с поддержкой зависимых типов на странице `https://en.wikipedia.org/wiki/Dependent_type`.

гарантированно детерминированный «верхний уровень» представлен в настоящее время хорошими современными языками программирования, для которых бурно развиваются новые методы и инструменты создания и сопровождения программ, такие, как верификация, то «нижний уровень» требует искусства и больших усилий разработчика, чтобы дать программисту «верхнего уровня» гарантии того, что библиотеки обладают всеми нужными свойствами, включая детерминированность и эффективность на параллельной аппаратуре.

Это означает, что улучшать, развивать и автоматизировать надо сразу оба уровня, в то время как обычно (по крайней мере, в публикациях) акцент ставится на красоту и изящество средств «верхнего уровня», а с «нижним уровнем» если и можно ознакомиться, то только в том случае, если автор выставил открытый код. *Уровни должны быть объединены общей системой понятий и языковыми конструкциями, различаясь только там, где это существенно.*

Таким образом, код на языке, объединяющем детерминированное и недетерминированное параллельное программирование, должен четко подразделяться на две части — гарантированно детерминированную и потенциально недетерминированную:

- *верхний уровень — детерминированная часть*: прикладной код на подмножестве языка, который пишет специалист в данном предметной области, которому гарантируется детерминированность любой его программы, использующей библиотеки нижнего уровня;
- *нижний уровень — недетерминированная часть*: библиотеки, создаваемые квалифицированными программистами для определенных классов задач на универсальном языке с богатым набором изобразительных средств, позволяющем кодировать недетерминированные параллельные алгоритмы. Авторы библиотек гарантируют детерминированность параллельных программ их пользователям, кодирующим на подмножестве языка на верхнем уровне.

2.3. Погружение в объектно-ориентированное программирование

Для разделения кода на уровни естественно воспользоваться современными методами модульного программирования. Самые устоявшиеся и привычные понятия для структуризации кода предлагает объектно-ориентированное программирование. Для реализации и

дальнейшего развития описанных в обзоре методов, среди объектно-ориентированных языков следует выбрать такой, у которого есть подмножество, представляющее полноценный чисто функциональный язык. Например, C++ не годится для этой цели, так как функциональные языки требуют такой свободы манипулирования со значениями, которая предполагает сборку мусора в реализации, чего у C++ нет. Java вполне годится, начиная с первых версий, а Java 8 — тем более, поскольку в нее внесены основные языковые конструкции и библиотечные понятия, привычные программистам на функциональных языках.

Если типы данных и операции «нижнего уровня» изображать в виде классов, возникает следующая схема построения системы детерминированного параллельного программирования.

(*1*) За основу берется универсальный объектно-ориентированный язык типа Java. Он, как и многие другие современные языки, содержит в себе подъязык, обеспечивающий функциональное программирование. Этот подъязык может быть формально определен, а компилятор может проверять, что фрагмент программы реализован на этом подъязыке. Такой функциональный подъязык обладает свойством детерминированности параллельных вычислений даже в самом общем случае, когда все вызовы подпрограмм (методов в классах) вызываются параллельно.

(*2*) Расширяем чисто функциональный подъязык теми понятиями из универсального объектно-ориентированного языка, которые не нарушают детерминированность параллельных вычислений, а именно: ссылочными значениями и операцией создания объектов, которые не изменяются после создания (*англ.* immutable objects). Принадлежность кода такому расширенному подъязыку проверяем компилятором.

(*3*) Специальными аннотациями отмечаем классы объектно-ориентированной программы, принадлежащие полному языку, считая по умолчанию, что класс принадлежит подъязыку.

(*4*) Разрешаем использовать весь универсальный объектно-ориентированный язык включая средства явного параллелизма только квалифицированным программистам, создающим библиотеку базовых классов, при использовании которых в рамках подъязыка не нарушается детерминированность. Только те классы, для которых выполняется это свойство, разрешаем использовать прикладному программисту.

В общем случае не существует универсальных формальных и алгоритмических методов проверки того, что данный базовый класс гарантирует детерминированность использующих его программ. В нынешних условиях проверку и поиск доказательства данного свойства приходится выполнять вручную. Мы рассчитываем, что в будущем удастся разработать (и даже вставить в компилятор) методы построения доказательств детерминированности, которые смогут проверяться автоматически для достаточно широкого множества классов, но ближняя цель — наработать как методы программирования в рамках такого языка, так и методы «ручного» доказательства детерминированности.

Простейший пример класса, через объекты которого передается информации в виде побочного эффекта, но при этом не нарушается детерминированность параллельных вычислений (если этот класс использовать в рамках подмножества языка Java, соответствующего языку функционального программирования) приведен в [**32**].

2.4. Перспективы

Мы полагаем, что классы, гарантирующие детерминированность использующих их программ, не накладывают чрезмерных ограничений на программирование приложений. Они позволяют создавать и обрабатывать достаточно сложные структуры данных, выходящие далеко за пределы возможностей обычного функционального программирования (например, представлять и обрабатывать графы).

Классы, описывающие объекты, изменяющиеся монотонно на некоторых решетках, — также достаточно богатое средство программирования. Например, в статье [**8**] показано, что с помощью «монотонных объектов» можно реализовать динамические частичные вычисления, повышая тем самым эффективность счета.

Более того, естественно ожидать, что детерминированный код будет лучше поддаваться глубокому анализу, преобразованиям и эффективному отображению в параллельные архитектуры компьютеров, чем недетерминированный.

Заключение

Общая цель работ, представленных в этой статье, — снижение трудоемкости параллельного программирования за счет как можно бóльшего увеличения доли кода, детерминированность которого проверяется и гарантируется инструментальными средствами.

Работы, представленные в обзоре, продвигаются к решению этой задачи с двух сторон:

(*1*) расширения функциональных языков операциями над новыми данными с сохранением детерминированности (этому посвящены все разделы обзора, кроме раздела 1.3);
(*2*) ограничения императивных языков статическими методами до детерминированного параллелизма (раздел 1.3).

Несомненно, эти линии развития постепенно сольются.

Сейчас работы второго направления посвящены, в основном, анализу областей действия побочных эффектов, чтобы выявлять невзаимодействующие параллельные процессы. Дальнейшее развитие статических методов будет нацелено на построение средств доказательства детерминированности. Возможно для этого будет использованы более богатые системы типизации с зависимыми типами.

Работы первого направления уже ближе к практике. Из обзора видны следующие основные круги расширения чисто функциональной модели вычислений новыми типами данных и операциями, сохраняющими детерминированность параллельных вычислений:

- всё начинается с функциональных языков и их параллельных реализаций;
- этот опыт переносится на популярные языки программирования путем выделения в них чисто функциональных подмножеств;
- добавляются неизменяемые переменные, объекты и другие структуры данных, а также всевозможные средства без побочных эффектов (здесь много различных работ, из которых в обзоре были упомянуты далеко не все);
- в язык вносятся такие операции над общими данными, которые обеспечивают средства взаимодействия процессов, не нарушающие детерминированность. Исторически первыми такими структурами были каналы и очереди в сетях Кана [**24**]. Затем в работах по dataflow архитектурам появились I-структуры [**23**], которые используются и в некоторых более поздних системах команд компьютеров [**28**]. Появляются языки и системы программирования, использующие все такие структуры и добивающиеся эффективной реализации, например, Concurrent Collections [**25**]. Возникает обобщение структур данных, через которые происходит взаимодействие процессов, до элементов некоторых решеток. Их значения монотонно изменяются в процессе вычислений в сторону уточнения [**26**, **27**].

Выкристаллизовывается методика программирования, обеспечивающая детерминированность приложений за счет разделения программного кода на два уровня: «высокий» прикладной — детерминированный по построению и «низкий» системный — возможно недетерминированный.

Во второй части статьи идея системы детерминированного программирования, основанной на этой методике, раскрыта более подробно. В рамках объектно-ориентированного программирования низкий уровень реализуется в виде библиотек классов, а верхний уровень — это подмножество объектно-ориентированного языка примерно соответствующего чисто функциональному языку.

Благодарности. *Авторы признательны С. А. Романенко, высказавшему много ценных замечаний и предложившему конкретные улучшения текста статьи, а также первым читателям Ю. А. Климову и Арк. В. Климову за содержательные советы.*

Список литературы

[1] А. И. Адамович. «Струи как основа реализации понятия Т-процесса для платформы JVM», *Программные системы: теория и приложения*, **6**:4 (27) (2015), с. 177–195, URL: http://psta.psiras.ru/read/psta2015_4_177-195.pdf ↑ 222

[2] А. И. Адамович. «Язык программирования Ajl: автоматическое динамическое распараллеливание для платформы JVM», *Программные системы: теория и приложения*, **7**:4 (31) (2016), с. 83–117, URL: http://psta.psiras.ru/read/psta2015_4_177-195.pdf ↑ 222

[3] А. И. Адамович, Анд. В. Климов. «Об опыте использования среды метапрограммирования Eclipse/TMF для конструирования специализированных языков», *Научный сервис в сети Интернет*, Труды XVIII Всероссийской научной конференции (19–24 сентября 2016 г., г. Новороссийск), ИПМ им. М.В. Келдыша, М., 2016, с. 3–8, URL: http://keldysh.ru/abrau/2016/45.pdf ↑ 222

[4] R.L̇. Bocchino (Jr.), V. S. Adve, S. V. Adve, M. Snir. “Parallel Programming Must Be Deterministic by Default”, *Fifth USENIX Conference on Hot Topics in Parallelism*, HotPar’09, USENIX Association, Berkeley, CA, USA, 2009, pp. 4–4. ↑ 222,228

[5] J. R. Gurd, C. C. Kirkham, I. Watson. “The Manchester Prototype Dataflow Computer”, *Commun. ACM*, **28**:1 (1985), pp. 34–52. ↑ 223,229

[6] K. Hammond, G. Michelson (eds.). *Research Directions in Parallel Functional Programming*, Springer-Verlag, London, UK, UK, 2000. ↑ 224

[7] С. М. Абрамов, А. И. Адамович, М. Р. Коваленко. «Т-система — среда программирования с поддержкой автоматического динамического распараллеливания программ. Пример реализации алгоритма построения изображений методом трассировки лучей», *Программирование*, **25**:2 (1999), с. 100–107. ↑ 224

[8] And. V. Klimov. “Dynamic Specialization in Extended Functional Language with Monotone Objects”, *SIGPLAN Not.*, **26**:9 (1991), pp. 199–210. ↑ 224, 236

[9] W. D. Hillis, G. L. Steele (Jr.). “Data Parallel Algorithms”, *Commun. ACM*, **29**:12 (1986), pp. 1170–1183. ↑ 224

[10] R. S. Nikhil. *ID Language Reference Manual*, Tech. Rep. CSG Memo 284-2, MIT Lab. for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1991. ↑ 224, 229

[11] А. Н. Андрианов, К. Н. Ефимкин, И. Б. Задыхайло. «Непроцедурный язык для решения задач математической физики», *Программирование*, 1991, №2, с. 80–84. ↑ 224

[12] А. Н. Андрианов, Т. П. Баранова, А. Б. Бугеря, К. Н. Ефимкин. «Трансляция непроцедурного языка Норма для графических процессоров», *Препринты ИПМ им. М.В. Келдыша*, 2016, 073, 24 с. ↑ 224, 225

[13] *Hadoop: MapReduce Tutorial*, https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html, Apache Hadoop, 2017. ↑ 225

[14] Дж. Блох. *Java. Эффективное программирование*, Лори, М., 2014, ISBN: 978-5-85582-348-6. ↑ 225

[15] *The Rust Programming Language*, Mozilla Research (Accessed Nov. 2017), URL: https://www.rust-lang.org ↑ 226

[16] A. Potanin, J. Östlund, Y. Zibin, M. D. Ernst. “Immutability”, *Aliasing in Object-Oriented Programming*, eds. D. Clarke, J. Noble, T. Wrigstad, Springer-Verlag, Berlin–Heidelberg, 2013, pp. 233–269. ↑ 226, 227

[17] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, F. Shull. “Exploring Language Support for Immutability”, *Proceedings of the 38th International Conference on Software Engineering*, ICSE ’16, ACM, New York, NY, USA, 2016, pp. 736–747. ↑ 226, 227

[18] J. Östlund, T. Wrigstad, D. Clarke, B. Åkerblom. “Ownership, Uniqueness, and Immutability”, *Objects, Components, Models and Patterns*, 46th International Conference, TOOLS EUROPE 2008 (Zurich, Switzerland, June 30–July 4, 2008), eds. R. F. Paige, B. Meyer,, Springer, Berlin–Heidelberg, 2008, pp. 178–197. ↑ 227

[19] Y. Zibin, A. Potanin, P. Li, M. Ali, M. D. Ernst. “Ownership and Immutability in Generic Java”, *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA ’10, ACM, New York, NY, USA, 2010, pp. 598–617. ↑ 227

[20] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, J. Sunshine. “Glacier: Transitive Class Immutability for Java”, *Proceedings of the 39th International Conference on Software Engineering*, ICSE ’17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 496–506. ↑ 227

[21] R.Ł. Bocchino (Jr.), V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, M. Vakilian. “A Type and Effect System for Deterministic Parallel Java”, *SIGPLAN Not.*, **44**:10 (2009), pp. 97–116. ↑ 228

[22] M. Kawaguchi, P. Rondon, A. Bakst, R. Jhala. “Deterministic Parallelism via Liquid Effects”, *ACM SIGPLAN Not.*, **47**:6 (2012), pp. 45–54. ↑ 228

[23] Arvind, R. S. Nikhil, K. K. Pingali. “I-structures: Data Structures for Parallel Computing”, *ACM Trans. Program. Lang. Syst.*, **11**:4 (1989), pp. 598–632. ↑ 229,237

[24] G. Kahn. “The Semantics of Simple Language for Parallel Programming”, *IFIP Congress*, 1974, pp. 471–475. ↑ 229,237

[25] M. G. Burke, K. Knobe, R. Newton, V. Sarkar. “Concurrent Collections Programming Model”, *Encyclopedia of Parallel Computing*, ed. D. Padua, Springer, Boston, MA, 2011, pp. 364–371. ↑ 229,232,237

[26] L. Kuper. *Lattice-based Data Structures for Deterministic Parallel and Distributed Programming*, Ph.D. Thesis, 2015, URL: http://www.cs.indiana.edu/~lkuper/papers/lindsey-kuper-dissertation.pdf ↑ 229,230,231,237

[27] L. Kuper, A. Todd, S. Tobin-Hochstadt, R. R. Newton. “Taming the Parallel Effect Zoo: Extensible Deterministic Parallelism with LVish”, *ACM SIGPLAN Not.*, **49**:6 (2014), pp. 2–14. ↑ 229,230,231,237

[28] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, B. Smith. “The Tera Computer System”, *SIGARCH Comput. Archit. News*, **18**:3b (1990), pp. 1–6. ↑ 229,237

[29] L. Kuper, A. Turon, N. R. Krishnaswami, R. R. Newton. “Freeze After Writing: Quasi-deterministic Parallel Programming with LVars”, *ACM SIGPLAN Not.*, **49**:1 (2014), pp. 257–270. ↑ 230,231

[30] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski. “Conflict-free Replicated Data Types”, *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS’11, Springer-Verlag, Berlin–Heidelberg, 2011, pp. 386–400. ↑ 231

[31] L. Kuper, R. R. Newton. “LVars: Lattice-based Data Structures for Deterministic Parallelism”, *2nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC’13, ACM, New York, NY, USA, 2013, pp. 71–84. ↑ 230,231

[32] Анд. В. Климов. «Детерминированные параллельные вычисления с монотонными объектами», *Научный сервис в сети Интернет: многоядерный компьютерный мир. 15 лет РФФИ*, Труды Всероссийской научной конференции (24–29 сентября 2007 г., г. Новороссийск), Изд-во Московского университета, М., 2007, с. 212–217. ↑ 236

Рекомендовал к публикации *к.ф.-м.н. С. А. Романенко*

Пример ссылки на эту публикацию:

А. И. Адамович, Анд. В. Климов. «Как создавать параллельные программы, детерминированные по построению? Постановка проблемы и обзор работ», *Программные системы: теория и приложения*, 2017, **8**:4(35), с. 221–244. URL: http://psta.psiras.ru/read/psta2017_4_221-244.pdf

Об авторах:

Алексей Игоревич Адамович

Старший научный сотрудник ИПС им. А.К. Айламазяна РАН. Работает в области инструментальных средств для разработки параллельных приложений. Разработчик первой параллельной версии Т-системы, ведущий разработчик параллельного отладчика tdb. Активный участник суперкомпьютерного проекта «СКИФ» Союзного государства России и Беларуси.

e-mail: lexa@adam.botik.ru

Андрей Валентинович Климов

Заведующий сектором методов анализа и преобразования программ Института прикладной математики им. М.В. Келдыша РАН. Области интересов: разработка и реализация языков программирования, особенно функциональных; методы преобразования программ, метавычисления, суперкомпиляция; приложения этих методов к разработке инструментов, облегчающих труд программистов.

e-mail: klimov@keldysh.ru

Alexei Adamovich, Andrei Klimov. *How to create deterministic by construction parallel programs? Problem statement and survey of related works.*

Abstract. One of the main problems that make parallel programming unreliable, labor-intensive, error-prone, and programs difficult to debug, is the non-determinism of processes and results of computation, when several runs of the same program with the same input data can produce different results because of different order of interaction of parallel processes. In connection with the rapid growth in the complexity of programs for supercomputers, the idea of parallel computations with determinism, guaranteed by language and a programming system has become more popular in the last decade and is becoming more vital.

The problem of how to make parallel programming as deterministic as possible is analyzed. An overview of some approaches to solving it is given. The task of developing a system that provides an opportunity to write both deterministic and nondeterministic code with guarantees to the application programmer that his program is deterministic, is discussed. (*In Russian*).

Key words and phrases: parallel computation models, deterministic programs, functional programming, object-oriented programming.

References

[1] A. I. Adamovich. "Fibers as the basis for the implementation of the notion of the T-process for the JVM platform", *Program Systems: Theory and Applications*, **6**:4 (27) (2015), pp. 177–195 (in Russian), URL: http://psta.psiras.ru/read/psta2015_4_177-195.pdf

[2] A. I. Adamovich. "The Ajl programming language: the automatic dynamic parallelization for the JVM platform", *Program Systems: Theory and Applications*, **7**:4 (31) (2016), pp. 83–117 (in Russian), URL: http://psta.psiras.ru/read/psta2015_4_177-195.pdf

[3] A. I. Adamovich, And. V. Klimov. "On the experience of using the environment metaprogramming Eclipse/TMF for designing specialized languages", *Nauchnyy servis v seti Internet*, Trudy XVIII Vserossiyskoy nauchnoy konferentsii (19–24 sentyabrya 2016 g., g. Novorossiysk), IPM im. M.V. Keldysha, M., 2016, pp. 3–8 (in Russian), URL: http://keldysh.ru/abrau/2016/45.pdf

[4] R.L̇. Bocchino (Jr.), V. S. Adve, S. V. Adve, M. Snir. "Parallel Programming Must Be Deterministic by Default", *Fifth USENIX Conference on Hot Topics in Parallelism*, HotPar'09, USENIX Association, Berkeley, CA, USA, 2009, pp. 4–4.

[5] J. R. Gurd, C. C. Kirkham, I. Watson. "The Manchester Prototype Dataflow Computer", *Commun. ACM*, **28**:1 (1985), pp. 34–52.

[6] K. Hammond, G. Michelson (eds.). *Research Directions in Parallel Functional Programming*, Springer-Verlag, London, UK, UK, 2000.

[7] S. M. Abramov, A. I. Adamovich, M. R. Kovalenko. "T-System - An Environment Supporting Automatic Dynamic Parallelization of Programs: An Example of the Implementation of an Image Rendering Algorithm Based on the Ray Tracing Method", *Programmirovaniye*, **25**:2 (1999), pp. 100–107 (in Russian).

DOI: 10.25209/2079-3316-2017-8-4-221-244

[8] And. V. Klimov. “Dynamic Specialization in Extended Functional Language with Monotone Objects”, *SIGPLAN Not.*, **26**:9 (1991), pp. 199–210.

[9] W. D. Hillis, G. L. Steele (Jr.). “Data Parallel Algorithms”, *Commun. ACM*, **29**:12 (1986), pp. 1170–1183.

[10] R. S. Nikhil. *ID Language Reference Manual*, Tech. Rep. CSG Memo 284-2, MIT Lab. for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1991.

[11] A. N. Andrianov, K. N. Yefimkin, I. B. Zadykhaylo. “Non-procedural language for solving problems of mathematical physics”, *Programmirovaniye*, 1991, no.2, pp. 80–84 (in Russian).

[12] A. N. Andrianov, T. P. Baranova, A. B. Bugerya, K. N. Yefimkin. “Nonprocedural NORMA language translation for GPUs”, *Preprinty IPM im. M.V. Keldysha*, 2016, 073 (in Russian), 24 p.

[13] *Hadoop: MapReduce Tutorial*, Apache Hadoop, 2017, URL: https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

[14] J. Bloch. *Effective Java*, 2nd Ed., Addison-Wesley, 2008, 346 p.

[15] *The Rust Programming Language*, Mozilla Research (Accessed Nov. 2017), URL: https://www.rust-lang.org

[16] A. Potanin, J. Östlund, Y. Zibin, M. D. Ernst. “Immutability”, *Aliasing in Object-Oriented Programming*, eds. D. Clarke, J. Noble, T. Wrigstad, Springer-Verlag, Berlin–Heidelberg, 2013, pp. 233–269.

[17] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, F. Shull. “Exploring Language Support for Immutability”, *Proceedings of the 38th International Conference on Software Engineering*, ICSE ’16, ACM, New York, NY, USA, 2016, pp. 736–747.

[18] J. Östlund, T. Wrigstad, D. Clarke, B. Åkerblom. “Ownership, Uniqueness, and Immutability”, *Objects, Components, Models and Patterns*, 46th International Conference, TOOLS EUROPE 2008 (Zurich, Switzerland, June 30–July 4, 2008), eds. R. F. Paige, B. Meyer,, Springer, Berlin–Heidelberg, 2008, pp. 178–197.

[19] Y. Zibin, A. Potanin, P. Li, M. Ali, M. D. Ernst. “Ownership and Immutability in Generic Java”, *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA ’10, ACM, New York, NY, USA, 2010, pp. 598–617.

[20] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, J. Sunshine. “Glacier: Transitive Class Immutability for Java”, *Proceedings of the 39th International Conference on Software Engineering*, ICSE ’17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 496–506.

[21] R.L̇. Bocchino (Jr.), V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, M. Vakilian. “A Type and Effect System for Deterministic Parallel Java”, *SIGPLAN Not.*, **44**:10 (2009), pp. 97–116.

[22] M. Kawaguchi, P. Rondon, A. Bakst, R. Jhala. “Deterministic Parallelism via Liquid Effects”, *ACM SIGPLAN Not.*, **47**:6 (2012), pp. 45–54.

[23] Arvind, R. S. Nikhil, K. K. Pingali. “I-structures: Data Structures for Parallel Computing”, *ACM Trans. Program. Lang. Syst.*, **11**:4 (1989), pp. 598–632.

[24] G. Kahn. “The Semantics of Simple Language for Parallel Programming”, *IFIP Congress*, 1974, pp. 471–475.

[25] M. G. Burke, K. Knobe, R. Newton, V. Sarkar. “Concurrent Collections Programming Model”, *Encyclopedia of Parallel Computing*, ed. D. Padua, Springer, Boston, MA, 2011, pp. 364–371.

[26] L. Kuper. *Lattice-based Data Structures for Deterministic Parallel and Distributed Programming*, Ph.D. Thesis, 2015, URL: http://www.cs.indiana.edu/~lkuper/papers/lindsey-kuper-dissertation.pdf

[27] L. Kuper, A. Todd, S. Tobin-Hochstadt, R. R. Newton. “Taming the Parallel Effect Zoo: Extensible Deterministic Parallelism with LVish”, *ACM SIGPLAN Not.*, **49**:6 (2014), pp. 2–14.

[28] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, B. Smith. “The Tera Computer System”, *SIGARCH Comput. Archit. News*, **18**:3b (1990), pp. 1–6.

[29] L. Kuper, A. Turon, N. R. Krishnaswami, R. R. Newton. “Freeze After Writing: Quasi-deterministic Parallel Programming with LVars”, *ACM SIGPLAN Not.*, **49**:1 (2014), pp. 257–270.

[30] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski. “Conflict-free Replicated Data Types”, *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS’11, Springer-Verlag, Berlin–Heidelberg, 2011, pp. 386–400.

[31] L. Kuper, R. R. Newton. “LVars: Lattice-based Data Structures for Deterministic Parallelism”, *2nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC’13, ACM, New York, NY, USA, 2013, pp. 71–84.

[32] And. V. Klimov. “Deterministic parallel computations with monotone objects”, *Nauchnyy servis v seti Internet: mnogoyadernyy komp’yuternyy mir. 15 let RFFI*, Trudy Vserossiyskoy nauchnoy konferentsii (24–29 sentyabrya 2007 g., g. Novorossiysk), Izd-vo Moskovskogo universiteta, M., 2007, pp. 212–217 (in Russian).

Sample citation of this publication:

Alexei Adamovich, Andrei Klimov. “How to create deterministic by construction parallel programs? Problem statement and survey of related works”, *Program systems: Theory and applications*, 2017, **8**:4(35), pp. 221–244. (*In Russian*). URL: http://psta.psiras.ru/read/psta2017_4_221-244.pdf