УДК 004.272.2
DOI: 10.46548/21vek-2021-1056-0008

ТЕОРЕТИЧЕСКИЕ АСПЕКТЫ ВЕРИФИКАЦИИ БЛОЧНО-СИНХРОННЫХ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ ДЛЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ, ПОСТРОЕННЫХ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ *BIGDATA*

Синев Михаил Петрович, кандидат технических наук, доцент кафедры «Вычислительная техника»
Пензенский государственный университет
(440026, Россия, Пенза, улица Красная, 40, e-mail: mix.sinev@gmail.com)
Трокоз Дмитрий Анатольевич, кандидат технических наук, доцент, проректор по научной работе
Мартышкин Алексей Иванович, кандидат технических наук, доцент,
заведующий кафедрой «Программирование»
Пензенский государственный технологический университет
(440039, Россия, Пенза, проезд Байдукова/ул. Гагарина, д. 1а/11,
e-mails: dmitriy.trokoz@gmail.com , alexey314@yandex.ru)

Аннотация. В статье приведено описание теоретических аспектов верификации блочно-синхронных параллельных программ для вычислительных систем, построенных с использованием технологии *BigData*. Мы являемся свидетелями того, как стремительно растет интерес к проблеме верификации программного обеспечения. Сейчас разработчикам программного обеспечения доступен широкий спектр методов доказательства программных свойств на популярных языках программирования, и средств, позволяющих автоматизировать проверку корректности приведенных доказательств. Процесс верификации программ является сложным и дорогим с финансовой точки зрения. Несмотря на это, есть ряд преимуществ, которые делают верификацию незаменимой в некоторых случаях, даже в сравнении с традиционным тестированием. В постановочной части исследования определен объект исследования – семантика языков программирования и инструментария для создания блочно-синхронных параллельных программ, а также поставлена цель исследования – описание абстрактного вычислителя, выполняющего программы на специальном языке, являющимся подмножеством языка С, который можно использовать для верификации программ. В ходе проведенных по тематике статьи исследований получены важные результаты, среди которых аналитический обзор по проблеме актуальности верификации параллельных программ. Обоснован выбор модели построения блочно-синхронных параллельных программ для создания производительного программного обеспечения. Выявлены случаи, когда формальная верификация программного обеспечения целесообразнее тестирования. Проанализированы актуальные публикации западных авторов, занимающихся исследованиями по проблемам верификации блочно-синхронных параллельных программ. Показано краткое описание семантики языка *BSPC*. В заключении сформулированы основные выводы по проделанной работе.

Ключевые слова: большие данные, верификация, модель, параллельная программа, параллельная система, проверка, распределенная система, семантика, тестирование, язык программирования.

THEORETICAL ASPECTS OF VERIFICATION OF BLOCK-SYNCHRONOUS PARALLEL PROGRAMS FOR COMPUTING SYSTEMS BUILT USING *BIGDATA* TECHNOLOGY

Sinev Mihail Petrovich, candidate of technical sciences,
associate professor of sub-department «Computer engineering»
Penza State University
(440026, Russia, Penza, Krasnaya Street, 40, e-mail: mix.sinev@gmail.com)
Trokoz Dmitriy Anatolevich, candidate of technical sciences, docent, vice-rector for scientific work
Martyshkin Alexey Ivanovich, candidate of technical sciences, docent, head of sub-department «Programming»
Penza state technological University
(440039, Russia, Penza, BaydukovProyezd / Gagarin Street, 1a/11,
e-mails: dmitriy.trokoz@gmail.com , alexey314@yandex.ru)

Abstract. The article describes the theoretical aspects of verification of block-synchronous parallel programs for computing systems built using *BigData* technology. We are witnessing a rapidly growing interest in the problem of software verification. Now software developers have access to a wide range of methods for proving program properties in popular programming languages, and tools that allow automating the verification of the correctness of the proofs given. The software verification process is complicated and expensive from a financial point of view. Despite this, there are a number of advantages that make verification indispensable in some cases, even in comparison with traditional testing. In the staged part of the study, the object of research is defined - the semantics of programming languages and tools for creating block-synchronous parallel programs, and the purpose of the study is to describe an abstract computer that executes programs in a special language, which is a subset of the C language that can be used for program verification. In the course of the research conducted on the subject of the article, important results were obtained, including an

analytical review on the problem of the relevance of verification of parallel programs. The choice of a model for building block-synchronous parallel programs for creating productive software is justified. Cases have been identified when formal software verification is more appropriate than testing. The current publications of Western authors engaged in research on the problems of verification of block-synchronous parallel programs are analyzed. A brief description of the semantics of the *BSPC* language is shown. In conclusion, the main conclusions on the work done are formulated.

Keywords: big data, verification, model, parallel program, parallel system, verification, distributed system, semantics, testing, programming language.

Введение. В настоящее время наблюдается стремительно возрастающий интерес к проблеме верификации программного обеспечения. В распоряжении разработчиков имеется богатый инструментарий методов доказательства программных свойств на популярных языках программирования, а также средств, позволяющих автоматически проверять корректность доказательств [1-3]. Как правило, верификация программ – сложный и дорогой процесс. Несмотря на это, есть некоторые преимущества, делающие ее незаменимой в ряде случаев в сравнении с традиционным тестированием:

1. Гарантии поведения, предоставляемые формально верифицированным программным обеспечением, обычно более обоснованы, чем полученные обычным тестированием. В случае управляющих систем экономически выгоднее верифицировать программу, снизив риск необратимой поломки дорогого устройства вследствие некорректного поведения программы.

2. В случае, когда необходимо доказать определенные свойства преобразований или их композиций, описывающихся логикой второго порядка и выше, тестирование неэффективно. Такие задачи возникают, например, при построении систем операционных преобразований, использующихся в продуктах, как *Google Docs* [4].

3. Многопоточные программы практически невозможно протестировать. Это связано с тем, что порядок выполнения отдельных инструкций каждый раз отличается

Объект исследования – семантика языков программирования и инструментария для создания блочно-синхронных параллельных программ [5].

Основная **цель** исследования – описание абстрактного вычислителя, выполняющего программы на специальном языке, являющимся подмножеством языка *C*, который можно использовать для верификации программ. Кроме того, он включает в себя конструкции для написания блочно-синхронных параллельных программ, семантика которых также формально описана. Для достижения поставленной цели в статье решаются задачи: изучение стандарта языка *C*; проведение анализа существующих описаний семантики языка *C*; изучение описаний модели блочно-синхронного параллелизма и ее реализаций.

Материалы и результаты исследования. Сегодня процессоры подошли к теоретическому максимуму производительности последовательных вычислений, обусловленному физическими ограничениями, которые накладываются на схемотехническую базу ЭВМ. По этой причине производители процессоров стремятся дать программистам больше возможностей для горизонтального масштабирования программных продуктов, предоставляя им дополнительные ядра и процессоры. Существующие программные и аппаратные решения в последнее время обладают высокой популярностью, что дает основания считать, что востребованность блочно-синхронного параллелизма будет расти, ввиду ее продвижения крупными компаниями, например, *Google*, *Microsoft* и т.п.

Впервые концепция блочно-синхронного параллелизма предложена в [6]. Позже в [7] описана спецификация библиотеки *BSPlib*, которую предлагалось использовать как набор примитивов для блочно-синхронных параллельных программ. Формальное описание частей *BSPLib* описано в [8], где показана семантика реализации библиотеки совместно с минималистичным языком программирования.

Долгое время производительность компьютеров росла, подчиняясь закону Мура, который в современном толковании звучит так: каждые 24 месяца количество транзисторов в интегральной схеме удваивается. Вместе с этим увеличивалось и быстродействие схем, в том числе и благодаря увеличению частоты, на которой они работают. Это позволяло увеличивать производительность программ, написанных в фон Неймановской парадигме (общая память, процессор, последовательное выполнение команд), не переписывая их, а лишь перенося их на более быстродействующие компьютеры. Сегодня наступил тот момент, когда процесс увеличения быстродействия последовательных вычислений существенно тормозит то, что компьютеры в своей миниатюризации дошли до состояния, когда уменьшение транзисторов произвести все труднее из-за разнообразных физических ограничений [9]. Возможности по дальнейшей миниатюризации полностью не исчерпаны [10]. Однако повысить производительность компьютера потенциально можно и на имеющейся схемотехнической базе – добавив дополнительные ядра в процессоры.

Многие существующие программы не адаптированы для выполнения в многопроцессорной среде. Они никак не могут задействовать дополнительные ресурсы в силу своей последовательностной природы. Писать параллельные программы сложнее, поскольку параллельные процессы обычно имеют общие ресурсы и могут конкурировать за них. В этой ситуации программисты обычно прибегают или к различным примитивам синхронизации (которые, в конечном счете, тормозят работу программы и могут служить причинами различных блоков – *spinlock, deadlock* и т.д.), или пользуются *lock-free* алгоритмами и структурами данных, которые крайне сложны в создании и требуют специальных навыков и высокой квалификации.

Вопрос разработчиков программного обеспечения «Как сделать параллельное программирование проще?» волнует по сей день. Универсального и оптимального с точки зрения производительности решения, конечно, нет. Различные исследователи вносят свои предложения моделей вычислений, которые достаточно производительны в большинстве случаев и существенно облегчают процесс написания параллельных программ, делая их поведение более предсказуемым. Создание такой модели должно стимулировать развитие параллельных вычислений. Одной из таких моделей является модель блочно-синхронного параллелизма. Она призвана играть ту же роль для параллельных вычислений, что и фон Неймановская модель для последовательных – быть универсальной абстракцией, точкой зрения, с которой удобно рассматривать построение алгоритмов. Эта модель может быть реализована программно или аппаратно.

Модель блочно-синхронного параллелизма. Модель блочно-синхронного параллелизма предлагает модель вычислений для параллельного программирования, которая должна обладать свойствами: быть независимой от аппаратной платформы; обеспечивать предсказуемое быстродействие; улучшать предсказуемость поведения программы и простоту написания; предоставлять максимально простую модель оценки сложности выполнения программы; быть легко масштабируемой. То, что модель блочно-синхронного параллелизма удовлетворяет этим требованиям, показано исследователями в работах еще в 90-х годах [11]. Их предсказания подтвердились в 2000-е годы со взрывным ростом популярности технологий, основанных на этой модели, например, *Apache Spark* и используемого в таких компаниях, как *Alibaba* и др [12].

Структурно абстрактный вычислитель для модели блочно-синхронного параллелизма (рис. 1) состоит из:

1. Набора процессоров – компонентов, способных на вычисления и локальные транзакции с памятью. Каждый процессор обладает доступом к памяти, изолированной от других процессоров;

2. Межпроцессорной сети, которая позволяет осуществлять обмен данными между областями памяти, принадлежащими разным процессорам;

3. Набора средств для обеспечения синхронизации всей системы.

При описании блочно-синхронной модели вычислений обычно используется термин «процессор». При программной реализации неважно, как будут реализованы эти процессоры: с помощью потоков (нитей) или изолированных процессов.

Модель разделяет вычисления на супершаги, каждый из которых делится на три составляющих:

1. Многочисленные процессы производят вычисления независимо друг от друга

2. Процессы обмениваются данными, необходимыми для продолжения вычислений

3. Происходит барьерная синхронизация процессоров.

Итак, модель блочно-синхронного параллелизма разделяет две важные стадии параллельных вычислений: вычисления и синхронизацию. Этот подход применим к различным параллельным архитектурам, включая архитектуры с распределенной памятью и мультипроцессорные архитектуры с общей памятью.

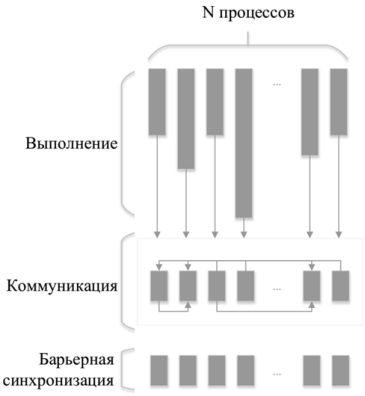

Рисунок 1 – Модель блочно-синхронного параллелизма

Модель блочно-синхронного параллелизма также предоставляет достаточно простой подход к оценке алгоритмической сложности. Стоимость выполнения программы складывается из суммы стоимостей супершагов. Стоимость одного супершага для p процессоров можно оценить выражением:

$$\max_{i=1,\dots,p} w_i + \max_{i=1,\dots,p}(h_i g) + l, \quad (1)$$

Здесь w_i обозначает стоимость локальных вычислений в процессе i, h_i – суммарное количество сообщений, которые послал и принял процесс i. Параметр g вводится для оценки производительности сети обмена сообщениями так, что для сообщения единичной длины время доставки выражается как hg. Обозначив соответствующие максимумы прописными буквами и просуммировав по количеству супершагов S получим выражение для оценки стоимости алгоритма:

$$W = Hg + Sl = \sum_{s=1}^{S} w_s + \sum_{s=1}^{S} h_s + Sl, \quad (2)$$

Среди альтернативных решений, нацеленных на те же проблемы, что и модель блочно-синхронного параллелизма, можно выделить спецификацию *MPI*, описанную в [13] и в [14-16]. И блочно-синхронная модель, и *MPI* являются описаниями среды для создания масштабируемых и переносимых параллельных программ. Главным их отличием является отсутствие концепции супершагов в *MPI*. Для программирования в модели блочно-синхронного параллелизма часто используется один из вариантов реализации спецификации *BSPlib* [6]. Существуют различные реализации этой спецификации, наиболее полный обзор приведен в [17]; в числе прочих упомянута реализация *BSP* on *MPI*, создающая набор примитивов для написания блочно-синхронных параллельных программ поверх реализации *MPI*. Благодаря тому, что реализации *MPI* существуют практически под все распространенные платформы, адаптация соответствующей реализации *BSPlib* не представляет трудностей. Библиотека *BSPlib* реализует метод *SPMD*, являющийся подкатегорией класса *MIMD* классификации вычислительных систем по Флинну. Она поддерживает два способа обмена данными между процессами: *DRMA* (прямой до-

ступ к удаленной памяти) и *BSMP* (блочно-синхронная передача сообщений).

Поскольку одной из целей создания *BSPlib* была простота написания параллельных программ, библиотека предоставляет программисту очень небольшой набор универсальных примитивов, а именно функции, среди которых, например:

– *void bsp_begin(int maxprocs).* Старт параллельных вычислений;

– *void bsp_end(void).* Завершение параллельных вычислений;

– *int bsp_abort(const char* format, …).* Аварийное завершение всех процессов;

– *int bsp_nprocs(void).* Возвращает число процессов;

– *int bsp_pid(void).* Возвращает идентификатор процесса.

При запуске программа начинает свою работу в однопоточном режиме. С помощью *bsp_init* можно указать функцию, которая содержит логику работы многопоточной части программы. Эта функция должна начинаться вызовом *bsp_begin* и заканчиваться вызовом *bsp_end.*

Функция *bsp_begin* отмечает начало многопоточной части и в качестве аргумента принимает количество процессов, которые необходимо создать. После этого код внутри библиотеки создает необходимое количество копий исходного процесса, которые продолжают свою работу независимо. Каждая из них обладает своей областью памяти.

Супершаг для процесса завершается вызовом функции *bsp_sync.* Гарантировано, что все действия, запланированные на супершаге, будут совершены после вызова *bsp_sync.*

С помощью функции *bsp_push_reg* область памяти помечается как доступная другим процессам для записи. Это запланированное действие, которое совершается в конце супершага. Функция *bsp_pop_reg* отменяет этот эффект; ее действие также отложено до конца супершага.

Если каждый процесс p_i на протяжении шага выполнил функцию *bsp_push_reg* с аргументами (x_i, s_i), то на следующем супершаге:

1. В *i*-ом процессе область памяти (x_i, s_i) будет считаться доступной другим процессам для чтения и записи;

2. Эти области будут логически связаны, и для любого процесса *p*<функции *bsp_get* и *bsp_put* могут использовать соответствующий адрес *x*< для того, чтобы обратиться к памяти процесса *k*, а именно к ее области (x_k, s_k).

Функция *bsp_get* позволяет запросить данные у удаленного процесса. Операция копирования будет произведена скрытым для программиста образом в конце супершага, соответственно, будет получено то состояние, в котором находились ячейки памяти на момент завершения супершага.

Функция *bsp_put* позволяет отправить данные удаленному процессу. Операция копирования будет произведена скрытым для программиста образом в конце супершага, но благодаря буферизации будет использоваться то состояние, в котором находились ячейки памяти на момент вызова функции *bsp_put.*

Функции *bsp_hpput* и *bsp_hpget* являются более высокопроизводительными версиями *bsp_put* и *bsp_get* соответственно. Это достигается за счет отсутствия буферизации. Иными словами, эффект *bsp_hpput* или *bsp_hpget* может произойти в любой момент до завершения супершага, а значит для обеспечения корректности нужно гарантировать выполнение следующих инвариантов:

1. При вызове *bsp_hpput(i, x, …)* в некотором процессе код в *i*-ом процессе не должен быть зависим от содержимого региона памяти по адресу *x*. Обычная версия bsp_put гарантирует отсутствие изменений *x* вплоть до синхронизации.

2. При вызове *bsp_hpput(i, x, …)* в некотором процессе нельзя допускать изменения памяти в области по адресу *x* вплоть до синхронизации. Обычная версия *bsp_put* сразу скопирует содержимое этой памяти во внутренний буфер библиотеки, так что содержимое *x* после этого можно произвольным образом менять.

3. При вызове *bsp_hpget(i, x, …)* в некотором процессе нельзя допускать зависимость от содержимого региона памяти по адресу *x*. Обычная версия *bsp_put* гарантирует отсутствие изменений *x* вплоть до синхронизации.

Современные методы описания семантики. В настоящее время наибольшее распространение получили три способа описания семантики формального языка [18, 19]:

1. Операционная, где каждому структурному элементу языка ставится в соответствии функция, преобразующая состояние абстрактного вычислителя;

2. Денотационная, где программа моделируется как функция на определенных математических объектах;

3. Аксиоматическая, где конструкциям языка ставятся в соответствие правила вывода, использующиеся для вывода формул, описывающих результаты выполнения этой конструкции. Прежде всего в настоящее время это логика Хоара и основанная на ней логика разделения, удобная для доказательств утверждений про состояние кучи: отсутствие утечек памяти, двойного освобождения памяти и т.д.

Операционную семантику описывают одним из двух способов:

1. Семантика большого шага описывает конечный результат работы абстрактного вычислителя после выполнения предложения языка. Результатом интерпретации программы на основании описания семантики большого шага является конечное состояние абстрактного вычислителя.

2. Семантика малого шага описывает процесс упрощения выражения на указанном языке. Результатом интерпретации программы на основании описания семантики малого шага является последовательность шагов абстрактного вычислителя с информацией о них (трассировка).

Описание, предложенное нами, предназначается

для формализации в среде *Coq*, реализующей вариант конструктивной логики. В силу того, что описывается семантика потенциально не завершающихся программ, мы прибегнем к описанию семантики малого шага.

Средство интерактивного доказательства теорем *Coq*. Описание, предложенное нами, предназначается для формализации в среде *Coq*, реализующей вариант конструктивной логики. В силу того, что описывается семантика потенциально не завершающихся программ, мы прибегнем к описанию семантики малого шага.

Coq – средство интерактивных доказательств, основанное на разновидности теории типов Мартина-Лефа. Он следует парадигме «высказывания – типы, доказательства – термы», известной как изоморфизм Карри-Говарда. Кроме того, эта система обогащена т.н. индуктивными определениями предикатов и типов данных.

Приведем пример описания синтаксиса и семантики языка арифметических выражений с помощью индуктивных определений в *Coq*:

```
Inductive expr: =
| Add: expr-> expr-> expr
| Mul: expr-> expr-> expr
| Lit: nat->expr.
Inductive bs_interpret (e:expr) (n:nat) :=
| IAdd: forall x y ix    iy,
    e = Add x y ->
    bs_interpret x ix ->
    bs_interpret y iy ->
    n = ix + iy ->
    bs_interpret e n
| IMul: forall x y ix iy,
    e = Mul x y ->
    bs_interpret x ix ->
    bs_interpret y iy ->
    n = ix * iy ->
    bs_interpret e n
| ILit: forall x,
    e = Lit x -> n = x ->
    bs_interpret e n.
```

На сегодняшний день с наиболее полным формальным описанием семантики *BSPlib* можно ознакомиться в работе [6], где предлагается операционная семантика языка *BSP-IMP*, состоящего из нескольких конструкций, типичных для императивных языков программирования, а также операций *get* и *put*, схожих с *bsp_get* и *bsp_put,* и [20], где предлагается расширение классического языка *IMP*, используемого во многих курсах языков программирования за рубежом. Он обладает очень простой семантикой и моделью памяти.

Заключение. В ходе подготовки статьи получены следующие результаты:

1. Выполнен аналитический обзор по проблеме актуальности верификации параллельных программ. Обоснован выбор модели построения блочно-синхронных параллельных программ для создания надежного и высокопроизводительного программного обеспечения. Выявлены такие случаи, когда формальная верификация программного обеспечения целесообразнее тестирования.

2. Выполнен обзор актуальных публикаций зарубежных авторов, занимающихся исследованиями по вопросам верификации блочно-синхронных параллельных программ.

3. Приведено описание семантики языка BSPC, аппроксимирующей язык C с библиотекой BSPLib.

СПИСОК ЛИТЕРАТУРЫ:

1. Шилов Н.В., Городняя Л.В., Марчук А.Г. Параллельное программирование среди других парадигм программирования // Прикладная информатика. – 2011. – № 1 (31). – С. 120-129.

2. Основы параллельного программирования: учебно-методическое пособие / С. А. Немнюгин. – Санкт-Петербург: Соло, 2007. – 103 с.

3. Городняя Л.В. О классификациях парадигм программирования и параллельном программировании // Образовательные ресурсы и технологии. – 2016. – № 2 (14). – С. 138-144.

4. Sun C., Ellis C. Operational transformation in real-time group editors: issues, algorithms, and achievements // CSCW '98: Proceedings of the 1998 ACM conference on Computer supported cooperative workNovember 1998 Pages 59–68https://doi.org/10.1145/289444.289469.

5. Котов В.Е. Проблемы развития параллельного программирования // Проблемы информатики. 2016. – № 2 (31). – С. 70-78.

6. W.F. McColl "Scalability portability and predictability: The BSP approach to parallel programming" Future Generation Computer Systems vol. 12 pp. 265-272 1996.

7. J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling, BSPlib: The BSP Programming Library, Parallel Computing, 24 (1998), pp. 1947–1980.

8. Julien Tesson, Frédéric Loulergue. Formal Semantics for the DRMA Programming Style Subset of the BSPlib Library. Seventh International Conference on Parallel Processing and Applied Mathematics (PPAM 2007), Workshop on Language-Based Parallel Programming Models, Sep 2007, Gdansk, Poland. pp.1122-1129, DOI: 10.1007/978-3-540-68111-3_119.

9. Roy K. Leakage Current Mechanisms and Leakage Reduction Techniques in DeepSubmicrometer, Vol. 91, №. 2, 2003. pp. 305-327.

10. A single-atom transistor | Nature Nanotechnology [Электронный ресурс]. – URL: https://www.nature.com/articles/nnano.2012.21 (дата обращения: 1.11.2021).

11. Hill J.M.D., McColl W.F. Questions and answers about BSP // Scientific Programming, Vol. 6, No. 3, 1997. pp. 249-274.

12. The Apache Software Foundation Announces Apache™ Spark™ as a Top-Level Project: The Apache Software Foundation Blog [Электронный ресурс]. – URL: https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces50 (дата обращения: 1.11.2021).

13. Gropp W., Lusk E., Skjellum A. Using MPI: Portable Parallel Programming with the Message Passing Interface, 1994.

14. Богословский Н.Н., Данилкин Е.А. Массовый открытый онлайн-курс «Введение в параллельное программирование с использованием OPENMP и MPI» // Хроники объединенного фонда электронных ресурсов Наука и образование. – 2016. – № 7 (86). – С. 13.

15. Антонов А.С. Параллельное программирование с использованием технологии MPI. – М.: МГУ, 2004. –71 с.

16. Жидаков В.П. Модели параллельного программирования для организации массовых параллельных вычислений // Телекоммуникационные и вычислительные системы: Труды конференции. – 2015. – С. 174-175.

17. What Bulk Synchronous Parallel (BSP) software tools have been developed? - Quora [Электронный ресурс] URL: https://www.quora.com/unanswered/What-Bulk-Synchronous-Parallel-BSP-software-tools-have-been-developed (дата обращения: 1.11.2021).

18. Title Software Foundations. Authors Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjberg, Brent Yorgey. Publisher: University of Pennsylvania, 2013.

19. Reynolds J.C. Separation Logic: A Logic for Shared Mutable Data Structures // LICS'02, 2002.

20. Gava F., Fortin J. Two Formal Semantics of a Subset of the Paderborn University BSPlib // Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2009. pp. 44-51.

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 19-07-00516 А.

Статья поступила в редакцию 05.11.2021
Статья принята к публикации 07.12.2021