

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра параллельных вычислений

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Иванченко Данилы Валерьевича

Тема работы:

**РАЗРАБОТКА И РЕАЛИЗАЦИЯ СПЕЦИАЛИЗИРОВАННОГО АЛГОРИТМА
РАСПРЕДЕЛЕННОЙ ДИНАМИЧЕСКОЙ СБОРКИ МУСОРА В СИСТЕМЕ LUNA**

«К защите допущена»

Заведующий кафедрой,

д.т.н., профессор

Малышкин В.Э. /.....

(ФИО)/ (подпись)

«31»...мая.....2023г.

Руководитель ВКР

к.т.н.,

доцент каф. ПВ ФИТ НГУ

Власенко А.Ю. /.....

(ФИО) / (подпись)

«20»...мая.....2023г

Соруководитель

ст. преп. каф. ПВ ФИТ НГУ

Перепёлкин В.А. /.....

(ФИО)/ (подпись)

«20»...мая.....2023г.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)
Факультет информационных технологий
Кафедра параллельных вычислений
(название кафедры)

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

УТВЕРЖДАЮ

Зав. кафедрой Малышкин В.Э.
(фамилия, И., О.)

.....
(подпись)

«7»...ноября.....2022г.

ЗАДАНИЕ

НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту(ке) Иванченко Даниле Валерьевичу, группы 19207

(фамилия, имя, отчество, номер группы)

Тема Разработка и реализация специализированного алгоритма распределенной динамической сборки мусора в системе LuNA

(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 07.11.2022 № 0330

Срок сдачи студентом готовой работы 20 мая 2023 г.

Исходные данные (или цель работы):

разработать и реализовать в виде программных модулей системы фрагментированного программирования LuNA специализированный алгоритм распределенной динамической сборки мусора

Структурные части работы:

обзор литературы, постановка задачи, разработка и реализация алгоритма, тестирование

Руководитель ВКР
доцент каф. ПВ ФИТ НГУ,
к.т.н.,
Власенко А.Ю. /.....
(ФИО) / (подпись)

«7»...ноября.....2022г.

Задание принял к исполнению
Иванченко Д.В. /.....
(ФИО студента) / (подпись)
«7»...ноября.....2022г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ,
Перепёлкин В.А. /.....
(ФИО) / (подпись)

«7»...ноября.....2022г

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ	4
ВВЕДЕНИЕ.....	5
1 Анализ предметной области	8
1.1 Обзор динамических алгоритмов сборки мусора в распределенных системах	8
1.1.1 Алгоритмы на основе подсчета ссылок	8
1.1.2 Алгоритмы на основе трассировки.....	9
1.2 Обзор систем автоматического конструирования параллельных программ.....	10
1.3 Итог	12
2 Динамический алгоритм распределенной сборки мусора.....	13
2.1 Термины и определения.....	13
2.2 Постановка задачи	13
2.2.1 Требования, предъявляемые к алгоритму	15
2.3 Описание предлагаемого решения.....	15
2.3.1 Индексированные фрагменты данных	20
2.3.2 Обработка операторов if, for, while	21
2.3.3 Идентификаторы фрагментов вычислений.....	25
2.4 Характеристика предлагаемого решения	26
3 Программная реализация	27
3.1 Структура транслятора.....	27
3.2 Структура исполнительной системы	28
3.3 Модификация исполнительной системы.....	29
3.4 Модификация транслятора	31
4 Тестирование.....	32
4.1 Сравнение потребления памяти и времени работы программы	32
ЗАКЛЮЧЕНИЕ	37
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ	39
ПРИЛОЖЕНИЕ А	42
ПРИЛОЖЕНИЕ Б.....	51

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Мультимпьютер – вычислительная система, состоящая из множества автономных компьютеров, называемых узлами, соединенных сетью и работающих совместно для выполнения вычислительных задач.

Вычислительный узел – узел мультимпьютера.

Мусор – объект в памяти, созданный программой, который после некоторого момента времени больше не используется или не может быть достигнут программой, но все еще занимает память компьютера.

Сборка мусора – форма автоматического управления памятью, основной задачей которой является освобождение памяти, занятой мусором.

Фрагмент вычислений – единица программы, содержащая описание входных/выходных фрагментов данных и кода (модуля, процедуры) этого фрагмента [3].

Оператор – синтаксическая единица языка программирования, означающая некоторое исполняемое действие. Операторами являются вызовы, объявления фрагментов данных, условные операторы, циклы. Оператор может быть связан с порождением фрагментов вычислений.

Фрагмент данных – агрегат из переменных [3].

Фрагментированная программа [10] – это набор $\langle CF, DF, \rho, In, Out \rangle$, где

1. CF – множество фрагментов вычислений (ФВ) – агрегированных операций множественного срабатывания.

2. DF – множество фрагментов данных (ФД) – агрегированных переменных множественного присваивания.

3. $\rho \subseteq CF \times CF$ – отношения строгого частичного порядка на множестве CF , определяющее порядок исполнения ФВ.

4. $In, Out: CF \rightarrow 2^{DF}$ – функции, сопоставляющие каждому фрагменту вычислений конечное множество входных и выходных фрагментов данных соответственно.

ВВЕДЕНИЕ

В настоящее время идет процесс создания новых и развития старых систем автоматического конструирования программ. Данные системы позволяют писать программы на языках высокого уровня, скрывая от пользователя детали низкоуровневого программирования, сетевого взаимодействия и др., тем самым накладывая на программиста более низкие требования к его квалификации, умению создавать корректные и эффективные программы. Под эффективной программой здесь и далее в работе подразумевается программа, время работы и объем потребляемой памяти которой являются приемлемыми для пользователя программы.

Одним из важных классов таких систем являются системы автоматического конструирования и исполнения параллельных программ, целью которых является упростить написание параллельных программ для распределенных систем.

Так как реализация параллельной программы осуществляется в общем случае в два этапа – трансляция и исполнение, важную роль играет процесс исполнения сконструированной параллельной программы. В частности, исполнение может осуществляться под управлением некоторой распределенной исполнительной системы, обеспечивающей динамические свойства программы.

Помимо задачи управления процессом исполнения программы, перед системой встает задача эффективного управления памятью. Одной из подзадач эффективного управления памятью является задача своевременного освобождения динамически выделенной памяти. Одним из решений данной задачи является использование сборщика “мусора”, который позволяет автоматически освобождать память, тем самым избавляя программиста от части проблем, связанных с ручным управлением памятью программы.

Задача сборки мусора может быть решена во время трансляции с помощью статического анализатора или динамически решаться исполнительной системой при ее наличии. Статический анализ может лишь частично решить проблему

потребления памяти в силу отсутствия у транслятора полной информации, доступной во время исполнения.

Подходы к сборке мусора, используемые в таких языках как Java, Go или C# не могут быть в полной мере использованы для такого класса систем, так как объекты с одного узла могут ссылаться на объекты других узлов в системе, следовательно, использование последовательных и многопоточных алгоритмов, которым необходима информация об использовании объектов, приведет к росту коммуникаций внутри сети, а следовательно, и к снижению скорости работы программы.

Более того, проблема сборки мусора в распределенных системах не может быть эффективно решена в общем случае ввиду алгоритмической сложности данной задачи и невозможности алгоритма соответствовать нефункциональным требованиям, предъявляемым к различным задачам. В силу этого, научное сообщество занимается разработкой специализированных и эвристических алгоритмов, которые будут эффективно работать на некотором классе вычислительных задач.

Система фрагментированного программирования LuNA [2], разрабатываемая в ИВМиМГ СО РАН, является одной из представительниц класса систем для автоматического построения параллельных программ. Данная система является подходящей площадкой для исследования проблемы распределенной сборки мусора из-за особенностей архитектуры, которые позволяют системе накапливать частные решения различных проблем в области распределенных систем.

Целью работы является разработка и реализация специализированного алгоритма распределенной динамической сборки мусора в системе LuNA.

Для достижения данной цели были поставлены следующие задачи:

1. Проанализировать существующие решения для сборки мусора.
2. Сформировать требования к разрабатываемому алгоритму и его нефункциональным свойствам при исполнении LuNA программ
3. Разработать алгоритм, удовлетворяющую требованиям из п. 2;

4. Реализовать разработанный алгоритм и встроить его в систему LuNA;
5. Провести экспериментальное исследование эффективности алгоритма на реальных вычислительных задачах.

Научная новизна работы состоит в том, что был разработан и реализован алгоритм динамической распределенной сборки мусора, учитывающий особенности систем фрагментированного программирования, в частности, системы LuNA, а также эффективно освобождающий мусор на итерационных задачах численного моделирования.

Практическая ценность работы заключается в том, что новый системный специализированный алгоритм динамической сборки мусора в системе LuNA позволил сократить потребление памяти во время исполнения различных прикладных задач и повысить эффективность исполнения фрагментированных программ.

Проделанная работа вносит вклад в решение проблемы сборки мусора в распределенных системах.

Настоящая работа состоит из введения, четырех глав и заключения. Во введении описывается общая проблема, которую призван решить алгоритм, актуальность проблемы, цель работы, практическая значимость и научная новизна. В первой главе рассматриваются родственные системы параллельного программирования, организация модулей сборки мусора в них, фиксируется результат обзора и формулируются требования. Во второй главе описывается модель нового специализированного алгоритма динамической сборки мусора. В третьей главе описываются детали реализации. В четвертой главе приведены результаты экспериментального исследования производительности алгоритма. В заключении представлены итоги работы.

1 Анализ предметной области

Динамический сборщик мусора является важным компонентом многих современных распределенных систем, которые выполняются на многих узлах сети. Ниже перечислены основные требования, которые предъявляются к динамическому алгоритму сборки мусора в распределенной системе.

Масштабируемость: динамический сборщик мусора должен быть масштабируемым для поддержки распределенных систем, работающих на многих узлах. Это означает, что он должен быть способен работать на многих узлах сети и управлять большим количеством объектов и данных.

Эффективность: динамический сборщик мусора должен предоставлять компромисс между уменьшением потребляемой памяти программы и накладными расходами по его работе.

Надежность: динамический сборщик мусора должен быть надежным и обеспечивать сохранность данных в распределенной системе.

На основе вышеизложенных критериев произведен обзор алгоритмов динамической сборки мусора в распределенных системах.

1.1 Обзор динамических алгоритмов сборки мусора в распределенных системах

1.1.1 Алгоритмы на основе подсчета ссылок

Стандартный алгоритм [7] подсчета ссылок создает для каждого объекта в системе счетчик ссылок на него. При создании новой ссылки на объект в системе, копировании существующей ссылки отправляется сообщение на узел, где хранится объект, для того чтобы увеличить или уменьшить счетчик ссылок. Такой подход сталкивается с проблемой потери сообщений в системе, что может привести к несогласованной работе алгоритма. Решения данной проблемы путем дублирования сообщений может привести к увеличению количества коммуникаций в сети. Также данный подход не уничтожает объекты с циклическими зависимостями.

Взвешенный алгоритм подсчета ссылок [5, 11] использует другую модель подсчета. При создании объекта ему присваивается вес, который

инициализируется максимально возможным значением, которое может храниться в системе. При создании ссылки ее вес будет также равен максимальному значению. При дублировании ссылки, исходная ссылка отдаст половину веса новой ссылке. При удалении ссылки, вес глобального объекта уменьшится на вес, равного весу ссылки. В итоге, вес глобального объекта будет равен сумме весов всех ссылок на него. Однако, данный подход сталкивается с тем, что при инициализации веса объекта значением 2^n , где n – количество бит под хранение значения, только n ссылок на объект может быть создано, так как каждое копирование ссылки в два раза уменьшает вес ссылки.

Алгоритм на основе списка ссылок [6] вместо подсчета создает запись о созданной ссылке на глобальный объект, новой или дублированной, тем самым решая проблему дублирования сообщений, когда счетчик ссылок может уменьшиться два раза. Таким образом, количество записей в списке ссылок соответствует количеству существующих в системе ссылок на объект. Данный алгоритм требует больших расходов памяти для поддержания соответствующего списка для каждого глобального объекта в системе.

1.1.2 Алгоритмы на основе трассировки

Алгоритмы на основе подсчета ссылок не могут собирать мусор при возникновении циклических зависимостей. Однако, алгоритмы на основе трассировки способны решать данную проблему и успешно собирать мусора внутри циклов зависимостей.

Стандартный алгоритм сборки мусора на основе трассировки [8] работает в две фазы: маркировка (Mark) и удаления (Sweep). Маркировка предполагает анализ графа объектов для выявления достижимых объектов, фаза Sweep удаляет объекты, которые остались недостижимыми после фазы Mark. В случае распределенных систем, некоторый главный вычислительный узел (master) отправляет сообщения на другие узлы о необходимости приостановке изменений объектов и проведении фазы маркировки. Каждый узел может удалить локальный мусор, глобальный же мусор требует фазы синхронизации между узлами. Когда узел-инициатор сборки мусора получает сообщения от всех

узлов о том, что фаза маркировки глобальных объектов на них завершена, то система может вступить в фазу удаления, которая может быть проведена локально, так как узлы уже обменялись информацией о достижимых объектах.

Алгоритм трассировки с использованием временных меток Хьюза [12] дополнительно к маркировке достижимых объектов дает им временную метку, связанную с временем начала фазы маркировки на узле. Алгоритм использует наблюдение о том, что “живые” объекты будут иметь свежую временную метку относительно текущего времени, мусор же будет иметь константную временную метку. Для корректной работы алгоритма в распределенной системе необходимо добиться синхронизации часов на узлах. Во время работы алгоритма происходит локальная фаза маркировки и очистки, после чего узлы обмениваются информацией о достижимых объектах. Если временная метка ссылки на объект больше, чем текущая на узле, то она обновляется на новую. В итоге, после фазы синхронизации, объекты, временная метка которых будет старше, чем текущее время с некоторым запасом по времени, будут удалены локально при фазе удаления. Данный алгоритм подвержен проблеме замедления сборки мусора в системе из-за задержки сборки мусора на локальных узлах.

Для получения полной картины об области алгоритмов распределенной сборки мусора, стоит рассмотреть их использование в системах автоматического конструирования параллельных программ.

1.2 Обзор систем автоматического конструирования параллельных программ

Система Legion [4] основана на понятии региона (region) – набора локальных данных, которые не зависят от других регионов. Также вводится понятие задач, которые представляют собой некоторые вычисления над регионами. Таким образом система Legion извлекает параллелизм из программы путем нахождения независимых задач и их распараллеливания внутри регионов. Такой подход дает возможность разделять описание вычислительной части алгоритма от того, каким образом он будет исполняться.

Влиять на производительность программы можно путем описания

интерфейсов (mappers), которые позволяют пользователю предоставить системе описание того, каким образом задачи накладываются на регионы.

В Legion для сборки мусора используется алгоритм распределенного подсчета ссылок, т.е. регион очищается только после того, как не осталось незавершенных задач, которые работают с данным регионом.

Язык X10 [22], разработанный компанией IBM, реализует асинхронную модель разделенного глобального адресного пространства (APGAS) [13]. Представленная модель разделяет глобальную память на ячейки, которые могут ссылаться друг на друга, а также привязываться к конкретным потокам или процессам. Потоки могут выполнять атомарные и асинхронные операции над ячейками, переключать место выполнения, дожидаться завершения работы других потоков. Такая модель упрощает работу с параллелизмом и распределенными системами, а также позволяет сохранять хорошую производительность при масштабировании системы. Сам язык расширяет подмножество языка Java, предоставляя дополнительные возможности для работы с массивами и процессами.

Сборка мусора в X10, аналогично таким языкам, работающими в общей памяти, как Java [19], Go [18], C# [17] происходит локально, т.е. при наличии глобальной ссылки, которая отправляется на другой узел, объект перестает быть достижимым сборщиком мусора. Для автоматического управления памятью используется отслеживающий сборщик мусора, который за первый проход по цепочке ссылок из стека маркирует достижимые объекты, за второй проход удаляются оставшиеся недостижимые объекты.

LuNA (Language for Numerical Algorithms) [3] – система фрагментированного программирования, разрабатываемая в ИВМиМГ СО РАН. Язык LuNA создан для описания сложных вычисленных моделей. Программа в LuNA представляется множеством фрагментов вычислений и фрагментов данных. В LuNA используется похожий подход, что и в системе Legion: описание алгоритма и управление его исполнением разделены. В LuNA влиять на исполняемый код можно с помощью рекомендаций в коде программы.

В LuNA встроено автоматическое удаление фрагментов данных при использовании рекомендации, которая задает максимальное количество использований фрагмента данных. Также в LuNA использовался алгоритм сборки мусора на основе областей видимости. Данный алгоритм удалял фрагменты данных при завершении фрагмента вычислений и всех его дочерних фрагментов, которые используют фрагмент данных. Таким образом момент удаления фрагмента данных во многих случаях может оказаться далек от фактического последнего потребления.

1.3 Итог

На основе вышесказанного можно сделать вывод, что область исследований распределенной сборки мусора содержит большое количество подходов по решению данной задачи и до сих пор исследуется. Тем не менее, существующие алгоритмы не закрывают проблему в полной мере, это приводит к необходимости разработать собственный специализированный алгоритм, который будет эффективным на узком классе практически значимых задач.

2 Динамический алгоритм распределенной сборки мусора

2.1 Термины и определения

Для понимания задачи, необходимо ввести следующие понятия:

Фрагмент данных (ФД) – агрегат из переменных [3]. Каждый фрагмент является неизменяемым и инициализируется единожды.

Фрагмент вычислений (ФВ) – исполняемая единица программы, содержащая описание входных и выходных фрагментов данных и исполняемого кода фрагмента без побочных эффектов [3].

Выражение – синтаксическая единица языка программирования, выполнение которой приводит к вычислению ее значения. Является комбинацией одной или более констант, фрагментов данных и операций, которая может быть интерпретирована в соответствии с правилами конкретного языка.

Запрос фрагмента данных [3] – механизм отправки сообщения от одного узла другому для получения фрагмента данных. Формат сообщения: пара $\langle dfid, cfid \rangle$, где $dfid$ и $cfid$ это идентификаторы ФД и ФВ соответственно. Ответ – пара $\langle r, val \rangle$, где r – запрос, а val – значение ФД.

Потребление фрагмента данных – использование фрагмента данных при исполнении кода фрагмента вычислений.

Рекомендация – механизм ручного управления поведением системы исполнения, используется чтобы сообщить компилятору или системе дополнительную информацию. В зависимости от назначения рекомендации, может иметь список аргументов, описанных в виде выражений.

Жизненный цикл фрагмента данных – набор всех потреблений фрагмента данных.

2.2 Постановка задачи

Перед формулировкой задачи, стоит рассмотреть следующее:

Фрагментированная программа в системе LuNA состоит из множества фрагментов вычислений и множества фрагментов данных. Фрагмент вычислений описывается входными и выходными фрагментами данных, а также описания кода фрагмента. Исполнением программы занимается исполнительная

система, которая может динамически создавать и удалять фрагменты данных на узлах, а также перемещать между узлами фрагменты вычислений и фрагменты данных.

Ход исполнения фрагментированной программы зависит от информационных зависимостей между фрагментами вычислений, т.е. фрагмент вычислений, который производит фрагмент данных будет выполнен до фрагмента вычислений, который потребляет данный фрагмент данных. Для того, чтобы разрешить эти зависимости в распределенной системе используется механизм запроса фрагмента данных на узел для выполнения фрагмента вычислений.

Для каждого фрагмента данных существует набор фрагментов вычислений, которые его потребляют. Через такой набор можно выразить жизненный цикл фрагмента данных (см. Рисунок 1).

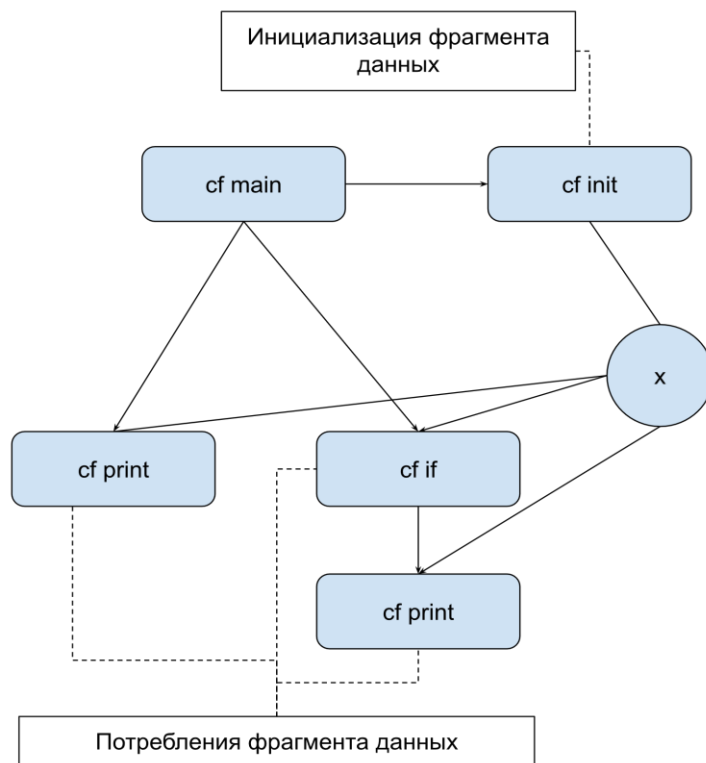


Рисунок 1 – Пример жизненного цикла фрагмента данных

Таким образом, задача динамической сборки мусора в распределенной системе подразумевает отслеживание жизненного цикла фрагмента, нахождение

момента времени, после которого фрагмент больше не будет использоваться в программе, и последующее удаление фрагмента.

2.2.1 Требования, предъявляемые к алгоритму

1. Алгоритм должен гарантировать корректное выполнение программы, то есть освобождение фрагмента данных должно производиться только после момента времени, когда он больше не будет потребляться.

2. Алгоритм должен снижать потребление памяти и не приводить к большому увеличению времени работы системы на некотором практически значимом классе задач.

2.3 Описание предлагаемого решения

Основная идея алгоритма лежит в отслеживании жизненного цикла фрагмента данных с помощью обработки запросов на получение фрагмента данных от потребляющих фрагментов вычислений и удаление фрагмента данных при получении и обработке запросов от всех фрагментов вычислений из набора потребляющих ФВ. Общая схема решения представлена на рисунке 2.

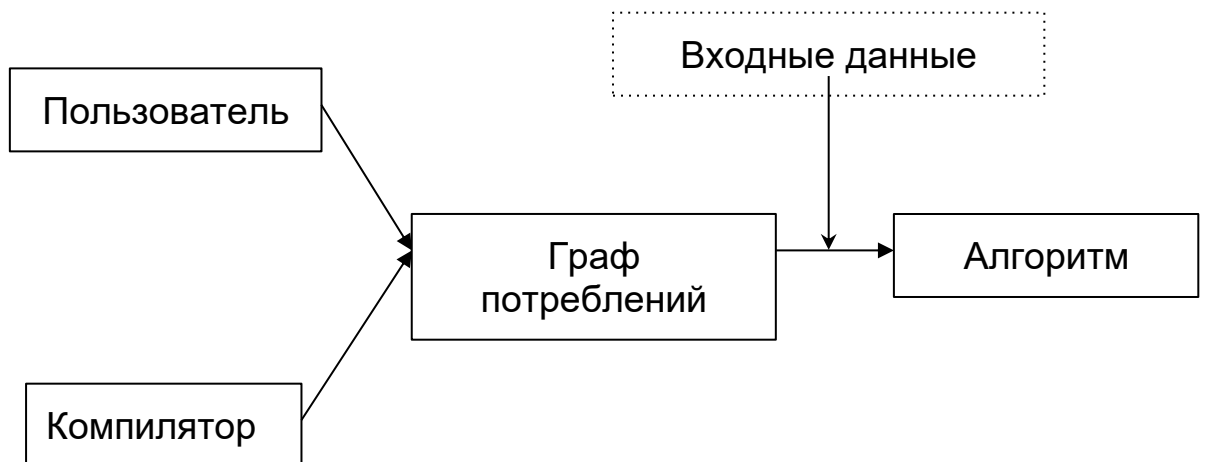


Рисунок 2 – Общая схема решения

Входными данными алгоритма является граф потребления, который может быть задан пользователем или построен на этапе компиляции путем анализа абстрактного синтаксического дерева программы. Конкретный алгоритм построения входного графа не относится к данной работе. Вершинами графа являются операторы программы, которые соответствуют фрагментам

вычислений, потребляющим фрагмент данных. Дугами графа представляют собой информационные зависимости между соответствующими фрагментами вычислений. Например, для программы на рисунке 3 граф потреблений будет иметь вид как на рисунке 4.

Граф инициализируется в момент порождения фрагмента данных, после этого отправляется вместе с фрагментом на узел хранения.

```
1. sub main()  
2. {  
3.     df a;  
4.     init(a);  
5.     cf a: consume(a);  
6.  
7.     if(a > 0) {  
8.         cf b: print(a);  
9.         cf c: print(a);  
10.    }  
11. }
```

Рисунок 3 – Пример фрагментированной программы

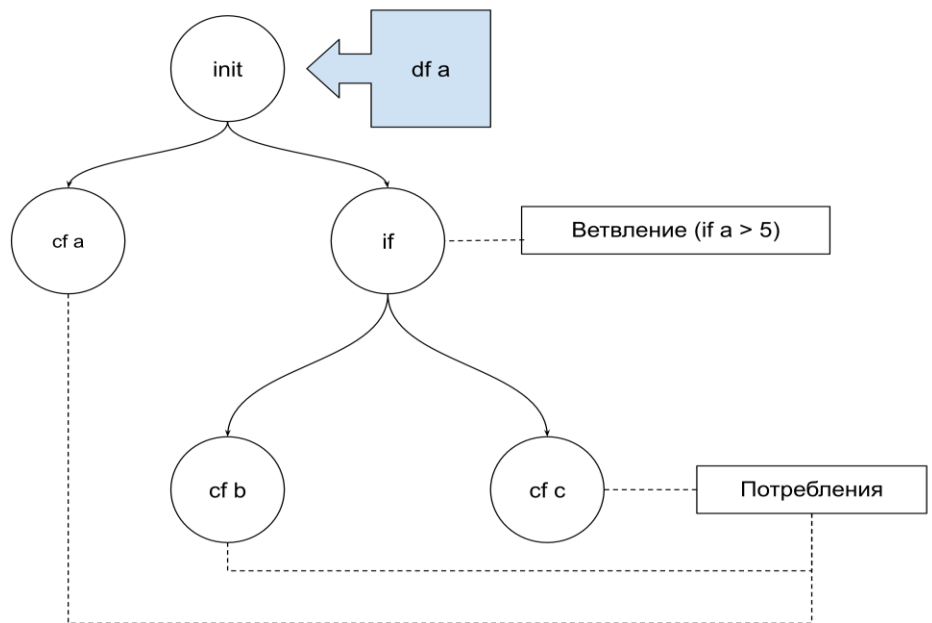


Рисунок 4 – Пример графа потреблений фрагмента a

Блок-схема алгоритма представлена на рисунке 5. Основным этапом

работы алгоритма является обработка на узле хранения фрагмента данных сообщений на запрос фрагмента или сообщений от управляющих операторов if, for или while, внутри тел которых фрагмент может быть потреблен.

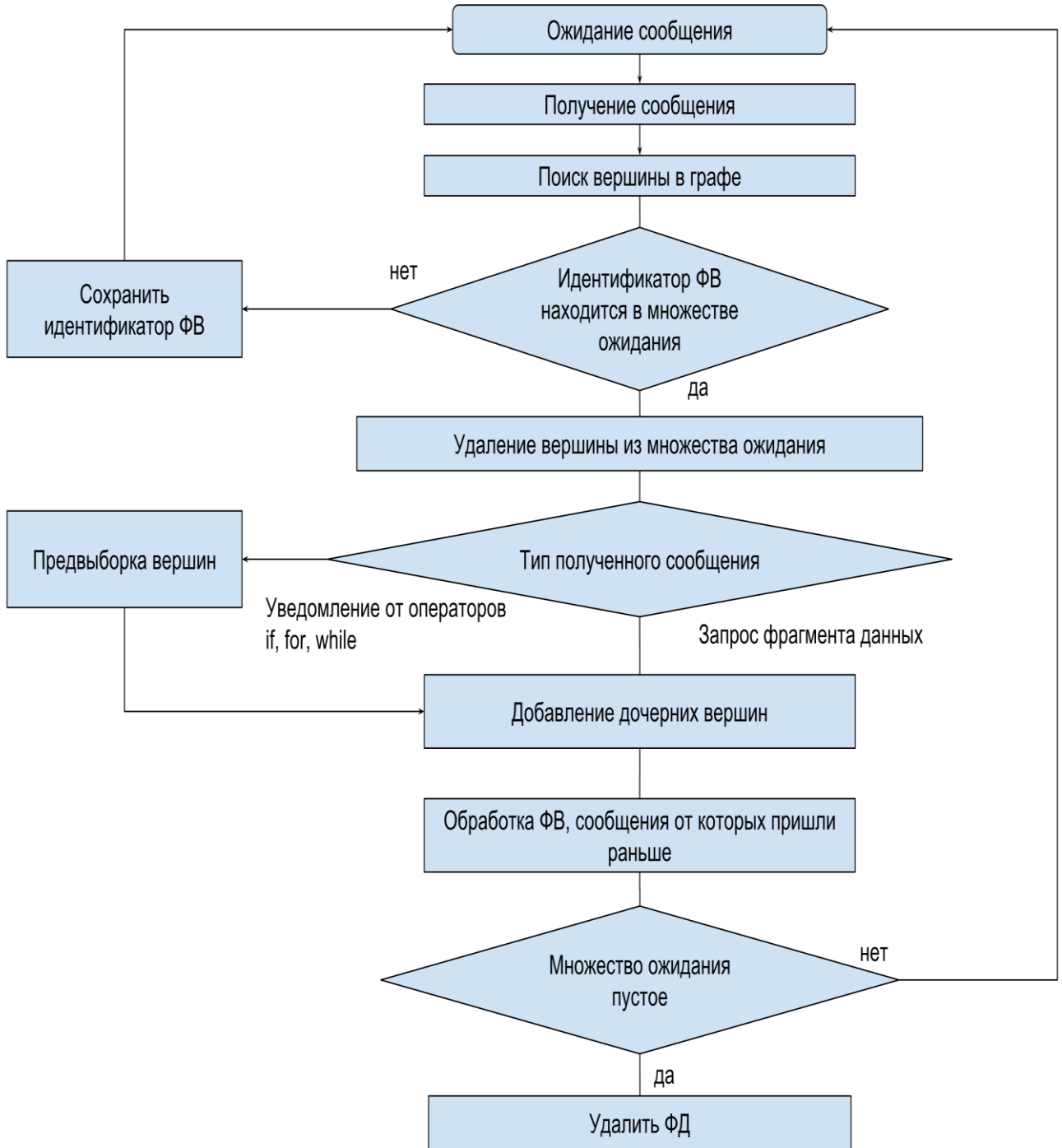


Рисунок 5 – Блок-схема алгоритма

На основе идентификатора ФВ, полученного из сообщения, алгоритм удаляет соответствующую вершину из графа потреблений. Сообщения от

операторов `if`, `for`, `while`, дополнительно к идентификатору ФВ содержат информацию о значении выражений, которые определяют работу операторов, то есть условного выражения для операторов `if`, `while` и выражения переменной цикла для операторов `for` и `while`. На основе этой информации на этапе исполнения можно сделать вывод о потреблении фрагмента данных фрагментами вычислений, которые будут или не будут порождаться в телах `if`, `for`, `while`. Таким образом, с помощью данных из уведомлений от управляющих операторов возможно удалить не только вершину, соответствующую самому оператору, но еще и все ее дочерние вершины.

Для нахождения соответствующей вершины в графе, алгоритм использует множество ожидающих вершин. До первого сообщения о запросе фрагмента, данное множество содержит вершины, соответствующие фрагментам вычислений, которые напрямую зависят от порождающего ФВ. Далее, при получении запроса, поиск вершины происходит именно в множестве ожидания. Найденная вершина удаляется из множества, а ее дочерние вершины в множество добавляются.

Если сообщение от ФВ пришло раньше, чем соответствующая вершина была добавлена в очередь ожидания, то идентификатор ФВ и дополнительная информация сохраняется, при следующем добавлении вершин в множество снова произойдет поиск вершины для сохраненного идентификатора.

После того, как очередь опустела, жизненный цикл фрагмента данных закончился, следовательно исполнительная система может удалить фрагмент данных. Для наглядного представления о работе алгоритма, рассмотрим его на примере программы из рисунка 3.

Алгоритм начинает свою работу после того, как граф потреблений вместе с фрагментом вычислений были получены на узле хранения. В этот момент, все дочерние вершины корня, который является процедурой создания фрагмента данных, добавляются в множество вершин, ожидающих запросов от соответствующих фрагментов вычислений (см. Рисунок 6).

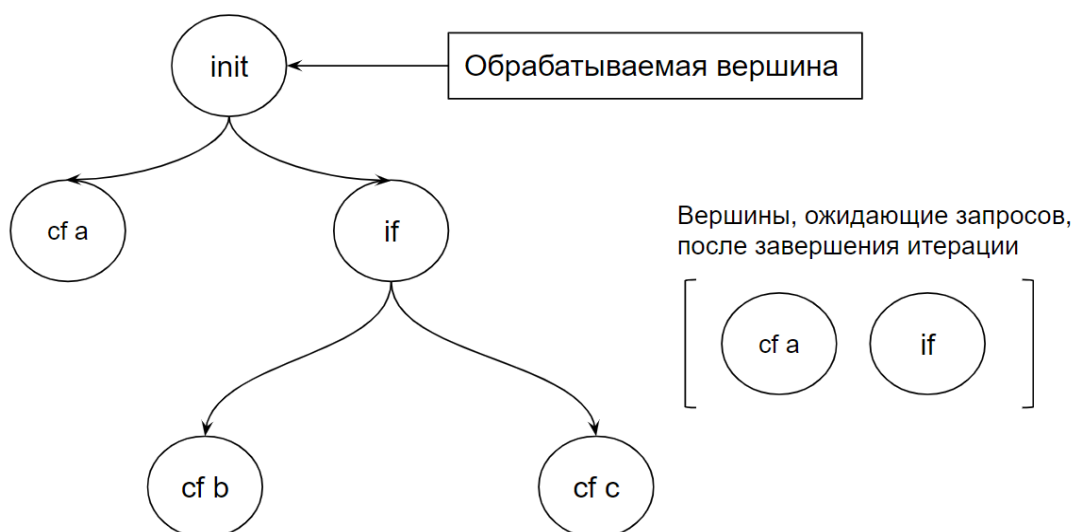


Рисунок 6 – Состояние множества ожидания после миграции графа на узел

Далее, алгоритм ожидает запросов на получение фрагмента данных. После получения уведомления о выполнении условия оператора `if` граф и множество ожидающих вершин будут выглядеть так, как на рисунке 7. Если же условия `if` ложно, то дочерние узлы вершины `if` не добавляются в очередь ожидания. В итоге, после обработки всех запросов, множество ожидания станет пустым, следовательно, фрагмент данных `a` может быть удален.

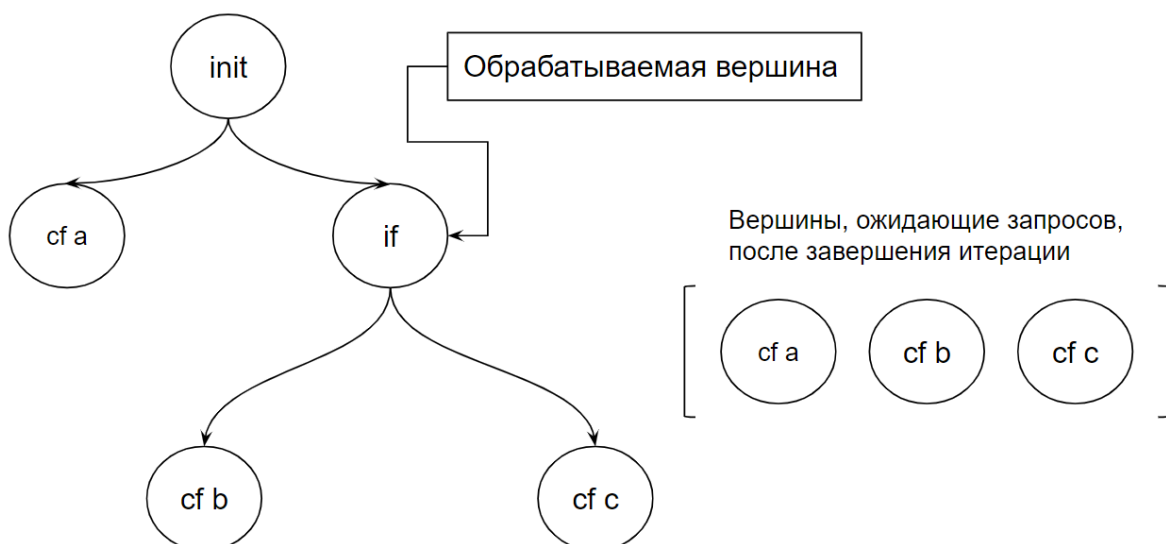


Рисунок 7 – Состояние множества ожидания после получения запроса/уведомления.

Далее будут рассмотрены детали алгоритма, которые характерны для работы LuNA программы в условиях распределенной системы.

2.3.1 Индексированные фрагменты данных

Система LuNA не имеет такой структуры данных, как массив, в системе используется понятие индексированного имени. Косвенной адресацией назовем запросы фрагментов данных, необходимых для вычисления значения выражения индекса.

По причине косвенной адресации индексированного фрагмента данных на этапе компиляции, невозможно построить граф потреблений каждого такого фрагмента данных. Поэтому в алгоритме для индексированных имен используется общий граф потреблений, который объединяет все потребления для разных фрагментов данных с общим именем.

Например, для программы на рисунке 8 граф потреблений фрагмента данных A на рисунке 9 включает в себя потребление $\text{print}(A[x - 1])$ для всех индексированных фрагментов данных в диапазоне индексов 0 до x . Так как на этапе компиляции ничего не известно о значении фрагмента x , то вершина графа, соответствующая фрагменту вычислений $\text{print}(A[x - 1])$ будет общей для всех индексированных фрагментов данных с общей частью A . В момент запроса $A[x - 1]$, будет происходить широковещательный запрос на другие узлы, чтобы произвести итерацию обхода и удалить из списка ожидания.

Описанный способ – крайне неэффективный в контексте общего решения для сборки мусора из-за большого количества накладных расходов, поэтому алгоритм предоставляет возможность игнорировать фрагменты, имена которых используют косвенную адресацию, таким образом, пользователь должен самостоятельно удалить фрагмент данных.

```

1. sub main()
2. {
3.     df A, x;
4.     init(A);
5.     init(x);
6.
7.     cf a: print(A[x - 1]);
8.     for i=0..x {
9.         cf b: print(A[i]);
10.    }
11. }

```

Рисунок 8 – Листинг программы с индексированными фрагментами данных.

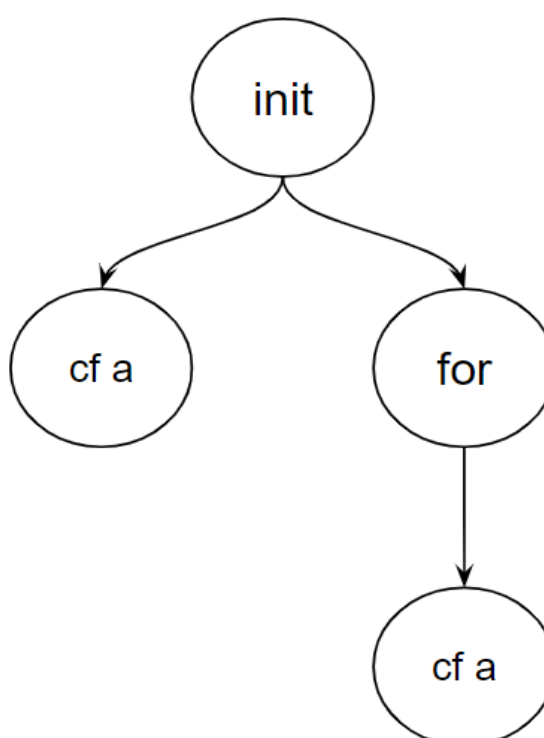


Рисунок 9 – Общий граф потреблений для индексированных фрагментов данных

2.3.2 Обработка операторов *if*, *for*, *while*

Особенность динамической сборки мусора на основе жизненного цикла фрагмента данных в том, что в некоторых случаях он определяет и очищает мусор намного раньше, чем, например, сборка мусора на основе областей видимости. Это достигается за счет того, что алгоритм может подстроиться под ход исполнения фрагментированной программы. Ход исполнения же во многом зависит от таких операторов как *if*, *for*, *while*.

В контексте системы LuNA вышеуказанные операторы содержат тело,

которое характеризуется набором фрагментов вычислений, которые запустятся в зависимости от выполнения условий операторов. В силу этого, внутри тела могут потребляться несколько различных фрагментов данных, графы которых включают в себя вершины, соответствующие операторам if, for, while. Следовательно, для того, чтобы полностью обойти граф, необходимо уведомить каждый фрагмент данных, который потребляется в теле операторов, о выполнении/не выполнении условия.

В то время как оператор if может обойтись одним уведомлением графов фрагментов данных, то операторы частично-рекурсивного и примитивного-рекурсивного перечисления, т.е. while и for соответственно, требуют немного другого подхода.

Особенностью оператора while является то, что он рекурсивно порождает все фрагменты вычислений в теле и свою копию, пока условное выражение истинно. Таким образом, вершина, соответствующая оператору while в графе потреблений, должна иметь петлю.

Так, например граф на рисунке 11 соответствует программе на рисунке 10.

```
1. sub main() {
2.     df a;
3.     init(a);
4.
5.     while(a[i] > 0), i=0..out {
6.         cf a: print(a[i]);
7.         cf b: consume(a[i]);
8.     }
9. }
```

Рисунок 10 – Листинг программы с оператором while

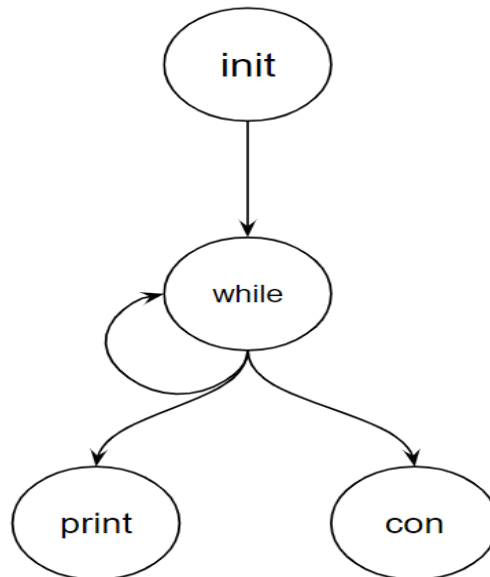


Рисунок 11 – Граф потребления для программы с оператором while

Следовательно, пока условия while выполняется, уведомления о истинности условия приходят на необходимые узлы, тем самым вершина, соответствующая while будет каждый раз добавляться во множество вершин, ожидающих запроса/уведомления. Тем самым, после завершения выполнения оператора while, фрагменты данных могут быть удалены.

Иначе дело обстоит с оператором for. Конструкция for <counter> = <first_expr> .. <last_expr> <body> предписывает породить новые фрагменты вычислений для каждого значения в целочисленном множестве между first_expr и last_expr. Таким образом решение по уведомлению каждого графа потреблений фрагментов данных на каждой итерации является неэффективным и ведет к снижению производительности программы. Поэтому для индексированных фрагментов данных в программе, индексы которых не зависят от других фрагментов данных, а зависят лишь от констант или счетчиков операторов for и while был добавлен механизм предвыборки вершин в графе потреблений.

Основной идеей механизма предвыборки является отправление границ <first_expr> и <last_expr> для счетчика на узел хранения при работе операторов for и while. Данные значения используются при добавлении дочерних вершин в множество ожидания. Предвыборка проверяет, находятся ли индекс фрагмента

данных в диапазоне от `<first_expr>` до `<last_expr>`, при нахождении в диапазоне, фрагмент вычислений добавляется в очередь ожидания. Для того, чтобы работать со сложными выражениями, которые представляют индекс индексированных фрагментов данных, в вершинах графа потреблений необходимо хранить дополнительную информацию о выражении. Для обработки фрагментов данных, не являющимися индексированными и фрагментов данных, индексы которых являются константами, необходимо точно знать количество итераций цикла. В случае `for`, если константный индекс лежит в границах счетчика, то мы ожидаем, что запрос на фрагмент данных придет еще $(\text{<last_expr>} - \text{<first_expr>})$ раз. Для этого, дополнительно к идентификатору вершины и выражения, представляющего индекс, вершина, находящаяся в множестве ожидания, должна хранить количество ожидаемых запросов для экономии памяти. По умолчанию для всех вершин данный параметр равен 1.

Отдельно стоит рассмотреть случай вложенных операторов `for`, `while`, а также их комбинации с `if`. На этапе предвыборки вершин в очередь ожидания в случае, если дочерней вершиной `for` (`while`) является другой `for` (`while/if`), то старый удаляется, при этом в дочерний `for` (`while/if`) передается информация о границах родительского и он сохраняется в очередь ожидания. Далее, после получения границ от дочернего оператора `for`, каждый индекс фрагмента данных проверяется на принадлежность соответствующему диапазону.

На примере фрагмента кода на рисунке 12 рассмотрим описанный выше механизм. Здесь выражения `<first_expr>` и `<last_expr>` равны 0 и 10 соответственно, а фрагменты вычислений `print1` и `print2` потребляют фрагмент данных с индексами $i + 1$ и 0 соответственно. Тогда при получении уведомления от оператора `for`, будет произведена проверка индекса фрагмента данных в диапазоне 1 и 11, следовательно, для всех индексированных фрагментов, индексы которых лежат в данном диапазоне произойдет добавление вершины, соответствующей `print1` в очередь ожидания. В случае `print2`, индекс 0 тоже находится в диапазоне, следовательно, вершина, с идентификатором `print2` будет добавлена 10 раз.

Таким образом такой подход к обработке фрагментов данных без косвенной адресации позволяет не использовать широковещательную рассылку при каждом употреблении фрагмента данных внутри тела операторов `for` и `while`.

```
1. for i=0..10 {  
2.     print1(A[i+1]);  
3.     print2(A[0]);  
4. }
```

Рисунок 12 – Листинг фрагмента кода с обращением к индексированным фрагментам данных по неизвестному и известному заранее индексам.

2.3.3 Идентификаторы фрагментов вычислений

Для корректной работы алгоритма требуется уникальность идентификатора фрагмента вычислений в области видимости фрагмента данных. Наивным решением данной проблемы является использование уже существующих в системе LuNA идентификаторов фрагментов вычислений, но данные идентификаторы уникальны только области видимости, в которой фрагмент вычислений был порожден. В случае использования встроенных идентификаторов ФВ, алгоритм сборки мусора в программе на рисунке 13 отработает неправильно, в зависимости от последовательности прихода уведомлений от внутренних операторов `if`. Предположим, что условие для второго вложенного оператора `if` окажется ложным, а первого `if` (`x == 0`) – истинным. Также допустим, что оба данных оператора уже находятся в множестве ожидания, у них у обоих одинаковый идентификатор `cf b`. Тогда, если уведомление о выполнении условия `if` для первого вложенного оператора приходит раньше, чем уведомления о втором операторе, то может случиться так, что дочерние вершины с идентификаторами `print1` и `print2` будут добавлены в очередь ожидания, но запрос от них уже никогда не придет. Следовательно, фрагмент данных не будет удален.

```

1. sub main()
2. {
3.     df x;
4.     init(x);
5.
6.
7.     if(x > 0) {
8.         cf b: if (x == 0) {
9.             print1(a);
10.        }
11.    }
12.
13.    if (x > 0) {
14.        cf b: if (x == 1) {
15.            print1(x);
16.            print2(x);
17.        }
18.    }
19. }

```

Рисунок 13 – Листинг программы с одинаковыми идентификаторами для порождающих фрагментов вычислений

Решить описанную выше проблему можно, добавляя идентификатору номер строки, на которой встречается оператор в тексте программы, а также сдвиг в этой строке. В итоге, идентификатор в своей сути позволяет отличать конструкции операторов на уровне синтаксиса, что позволяет добиться корректной работы алгоритма.

2.4 Характеристика предлагаемого решения

Данное решение позволяет динамически отслеживать жизненный цикл фрагментов данных. Главным плюсом такого подхода является то, что алгоритм хорошо подстраивается под ход исполнения программы и очищает фрагмент данных именно в тот момент, после которого он уже не будет использоваться. Минусом алгоритма является то, что он неэффективно работает с косвенной адресацией индексированных фрагментов данных. Данный недостаток накладывает ограничения на спектр применимости алгоритма в общем случае. Но стоит отметить, что алгоритм может быть применим к классу вычислительных задач, которые не зависят от косвенной адресации, например, перемножение матриц или реализация метода прогонки.

3 Программная реализация

Перед описанием программной реализации алгоритма необходимо описать структуру исполнительской системы и транслятора LuNA. В разделах 3.1 и 3.2 описаны структура транслятора и исполнительской системы LuNA. В разделах 3.3 и 3.4 описаны изменения, которые были внесены в систему для реализации алгоритма динамической сборки мусора, в приложении А приведено руководство программиста, в приложении Б – описание программы.

3.1 Структура транслятора

Транслятор в системе LuNA имеет конвейерную структуру. Транслятор LuNA получает на вход описание фрагментированного алгоритма и множества фрагментов кода в виде набора исходных файлов на языках LuNA и C++, а на выход выдает исполняемое представление заданного ФА в виде динамически подключаемой библиотеки (файл с расширением .so).

LuNA программа описывается в виде набора LuNA файлов, также реализации фрагментов кода фрагментов вычислений могут быть написаны на C++.

Препроцессор LuNA обрабатывает входные LuNA и C++ файлы, удаляя комментарии, подставляя включаемые файлы, делая макроподстановки. Далее происходят этапы синтаксического и лексического анализа, итогом работы которых является абстрактное синтаксическое дерево программы в виде JSON [20]. После этого построенного абстрактного синтаксического дерева модифицируется с учетом пользовательских рекомендаций о ходе исполнения программы. В конце этапа трансляции происходит генерация внутреннего представления LuNA программы в виде последовательного кода, компиляция этого представления последовательным компилятором в стандартную динамическую библиотеку. Полученная библиотека используется исполнительской системой при распределенном исполнении фрагментированного алгоритма.

Части транслятора, отвечающие за препроцессинг, генерацию последовательного C++ кода на основе абстрактного синтаксического дерева и

рекомендаций реализована на языке Python. Итоговый код, полученный после работы транслятора, компилируется C++ компилятором. Для лексического и синтаксического анализа используются утилиты flex [24] и bison [23] соответственно.

Транслятор LuNA является однопроходным. Каждый оператор в теле каждой подпрограммы LuNA программы обрабатывается рекурсивно.

3.2 Структура исполнительной системы

Исполнительная система является распределенной параллельной программой и написана на языке C++. Для реализации коммуникаций используется библиотека MPICH [14], реализующая стандарт MPI [15].

Исполнительная система состоит из двух модулей: сетевого и исполнительного. Исполнительный модуль занимается исполнением полученного представления фрагментированного алгоритма. Сетевой модуль предоставляет возможность отправки и получения сообщений между узлами в сети.

Исполнительный модуль начинает свою работу с выполнения реализации подпрограммы `sub main`, аналогом функции `main`. Далее, из `sub main` происходит рекурсивное порождение новых реализаций фрагментов вычислений. В итоге, ход исполнения LuNA-программы можно представить в виде ориентированного графа задач, которые исполняются набором потоков.

Сетевой модуль ожидает получения сообщений от других узлов и, в зависимости от типа сообщения, вызывает соответствующую функцию-обработчик. В системе существуют следующие виды сообщений:

1. Запрос фрагмента данных
2. Поступление какого-то фрагмента данных
3. Получение запрашиваемого фрагмента данных
4. Миграция кода фрагмента вычислений и его состояния.
5. Завершение работы системы

Также модуль отправляет сообщения на другие узлы при необходимости, например, при запросе фрагментов данных, необходимых для выполнения

фрагмента вычислений.

Для работы с фрагментами данных система использует два ассоциативных массива `posts` и `requests`, где `posts` хранит фрагменты данных с доступом по идентификатору фрагмента данных, а `requests` – сохраняет обратные вызовы (`callback`), которые будут выполнены при получении фрагмента данных с необходимым идентификатором. При запросе фрагмента данных соответствующая запись будет добавлена в массив `requests` на удаленном узле, при получении сообщения. В массив `posts` новая запись добавляется при получении нового фрагмента данных (получение фрагмента данных по запросу, миграция фрагмента данных с узла на узел).

3.3 Модификация исполнительной системы

Дополнительно к наборам `posts` и `requests`, был добавлен новый ассоциативный массив, который представляет собой множество ожидания. Ключом в массиве является идентификатор фрагмента данных, а значением – набор указателей на вершины графа, соответствующих фрагментам вычислений, от которых ожидаются сообщения о запросе фрагмента данных.

Был изменен формат сообщения при запросе фрагмента данных, теперь, помимо идентификатора запрашиваемого фрагмента данных и обратного вызова, сообщение содержит идентификатор запрашивающего фрагмента вычислений.

Теперь при получении сообщения о запросе фрагмента данных происходит поиск вершины графа в наборе, соответствующему идентификатору фрагмента данных из сообщения, после этого, в множество ожидания добавляются новые вершины, а сама вершина удаляется из множества ожидания.

Класс, представляющий вершину графа потреблений, имеет следующие поля:

1. `cf_id` – идентификатор фрагмента вычислений
2. Массив указателей на дочерние вершины графа потреблений
3. `req_count` – ожидаемое количество запросов от фрагмента вычислений
4. Массив диапазонов индексов для работы предвыборки в случае

вложенных операторов if, for, while.

5. `consumption_cond` – функция проверки соответствия индексам (см. Рисунок 14).

Функция проверки индексов принимает аргументами массив диапазонов и массив значений индексов для конкретного фрагмента данных, для каждого значения индекса проверяется, попадает ли он в диапазон. Данная функция генерируется на этапе компиляции и используется при выполнении предвыборки вершин в множество ожидания.

Были введены новые виды сообщений для получения уведомлений от операторов if, for, while. Помимо идентификатора фрагмента вычислений, сообщение от оператора if представляет собой идентификатор фрагмента вычислений и результат условия оператора, для оператора for – диапазон значений `<first_expr> – <last_expr>`, для оператора while – значение условия, `<first_expr> – <last_expr>`. Данным сообщениям были добавлены соответствующие обработчики, которые аналогично обычным запросам ищут соответствующую вершину в множестве ожидания, но в случае if и while, дочерние вершины теперь могут не добавляться в множество ожидания, если условие ложно. В случае for и while, если фрагмент данных является индексруемым именем, то дочерние вершины могут добавляться/не добавляться в множество ожидания в зависимости от результата работы метода `consumption_cond` соответствующей вершины. В случае, если вершина является оператором if, for или while, диапазон из сообщения добавляется в массив диапазонов дочерней вершины.

```
1. [](std::vector<std::pair<int, int>>& ranges, std::vector<int>& idx){
2.return (ranges[0].first <= idx[0] && idx[0] <= ranges[0].second);
3.}
```

Рисунок 14 – Листинг генерируемой функции проверки нахождения индекса в диапазоне

3.4 Модификация транслятора

Для тестирования алгоритма была добавлена отдельная рекомендация для

исполнительной системы, через которую алгоритм может получить входные данные в виде графа потреблений фрагмента данных. Синтаксис рекомендации имеет следующий вид (см. Рисунок 15):

`gc_graph <df_id>=>{<init_id>=>{<consume_id>=>{...},<consume_id>=>{...}}`, где `df_id` – идентификатор фрагмента данных, для которого строится граф потреблений, `init_id` – идентификатор фрагмента кода, который инициализирует фрагмент данных, `consume_id` – идентификаторы фрагментов кода, в которых происходит потребление фрагмента данных. Стрелка `=>` указывает на наличие дочерних вершин в графе потреблений. На основе рекомендации, в момент генерации последовательного C++ кода, происходит инициализация объекта графа потребления для заданного фрагмента данных путем рекурсивного обхода схемы из рекомендации. Данный объект отправится на узел хранения вместе с фрагментом данных при его инициализации.

```
1. sub main() {  
2.     df x;  
3.     init(x) @ {  
4.         gc_graph: x=>{init=>{cons1, cons2=>{cons3}}};  
5     };  
6. };
```

Рисунок 15 – Синтаксис рекомендации для построения дерева потреблений

Уникальность идентификаторов для операторов `if`, `for`, `while` предоставляется с помощью добавления уникального номера к текущему идентификатору оператора, т.е происходит нумерация операторов `if`, `for`, `while`.

4 Тестирование

4.1 Сравнение потребления памяти и времени работы программы

Для первого теста была выбрана программа, которая инициализирует матрицу заданного размера и во вложенном цикле выводит значения i строки и j столбца, а также $i + 1$ строки и $j + 1$ столбца. Тестирование проводилось на кластере ИВЦ НГУ [16].

Цель теста – проверить работу алгоритма и в частности, работу предвыборки для циклов. Листинг программы теста представлен на рисунке 16.

```
1. #!/usr/bin/luna
2.
3. import c_init(int, name) as init;
4. import c_iprint(int) as iprint;
5.
6. sub main(int mWidth, int mHeight)
7. {
8.     df x;
9.     for i=0..mHeight {
10.         for j=0..mWidth {
11.             init(mWidth*i + j, x[i][j]) @
12.             {gc_graph: x => {init => {for1 => {for2 => {a, b}}}}};
13.         }
14.     }
15.     for i=0..mHeight - 1 {
16.         for j=0..mWidth - 1 {
17.             cf a: iprint(x[i][j]);
18.             cf b: iprint(x[i + 1][j + 1]);
19.         }
20.     } @ {
21.         locator_cyclic: 0;
22.     }
23. }
```

Рисунок 16 – Листинг программы test_1

Встроенная утилита top [21] была использована для получения информации о потреблении памяти, т. к. она позволяет получать информацию о процессах в системе в реальном времени. В силу того, что память, очищаемая во время исполнения программы с помощью delete или free не отдается операционной системе, а маркируется процессом как свободная и остается у него для дальнейшего использования. В итоге, графики потребления памяти для всех

вариантов исполнения программы будут неубывающими.

Для теста был выбран размер матрицы 10000x10000. Данного размера достаточно, чтобы задача выполнялась более 30 секунд. На рисунке 17 представлен график, который по оси Y показывает количество памяти, потребляемое программой, а по оси X – время с шагом 0.1 секунды для разных вариантов исполнения программы.

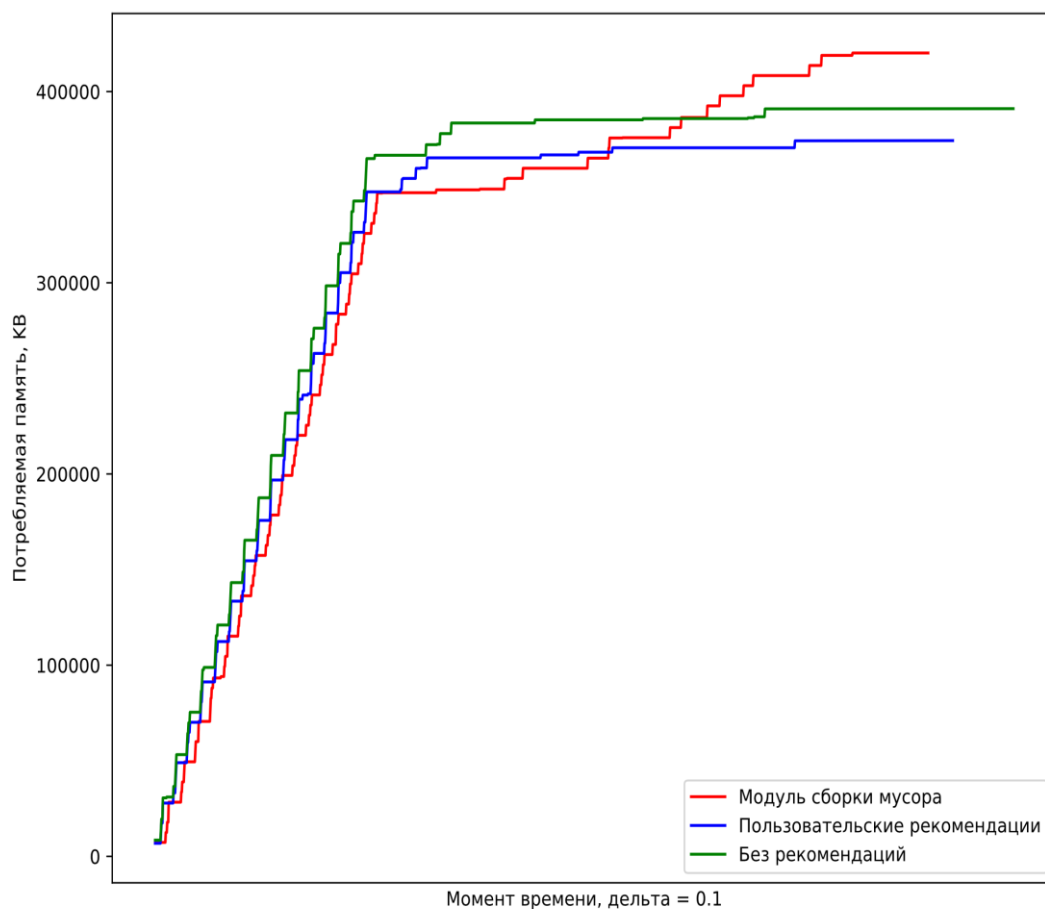


Рисунок 17 – График потребления памяти для разных вариантов запуска теста 1

На основе графиков на рисунке 17 и данных из таблицы 1 можно сразу увидеть, что вариант без использования рекомендаций потребляет больше памяти и работает быстрее, чем варианты со сборкой мусора. Это объясняется тем, что полученное ускорение за счет локальности памяти не компенсирует временные затраты на выполнение алгоритма.

Таблица 1 – Сравнение времени работы программы для разных вариантов управления памятью, сек

Без рекомендаций	Пользовательские рекомендации	Сборщик мусора
41.33	43,81	46, 51

Также график показывает, что пиковое потребление памяти для варианта без рекомендаций намного больше, чем для вариантов, использующих сборку мусора или пользовательские рекомендации. В таблице 2 представлены пиковые значения для каждого варианта.

Таблица 2 – Сравнение пикового значения потребляемой памяти для разных вариантов управления памятью, кб

Без рекомендаций	Пользовательские рекомендации	Сборщик мусора
406159	352175	375291

На основе данных из таблицы, видно, что разница в потреблении памяти между программой без рекомендаций и программой со сборщиком мусора составляет 30868 кб (7 %). Такой разрыв вызван тем, что при окончании жизненного цикла фрагмента данных он не удаляется, а остается храниться на узле до завершения программы. Разница в 23116 кб (6 %) между пользовательскими рекомендациями и модулем сборки мусора обусловлена тем, что для алгоритма требуется хранить дополнительную информацию в виде вершин для каждого фрагмента данных.

Также в алгоритм был протестирован на задаче моделирования эволюции самогравитирующего пылевого протопланетного диска методом частиц-в-ячейках [9]. В тестируемом приложении имеется пространство моделирования, ограниченное прямоугольным параллелепипедом, в котором происходит движение модельных частиц. Частицы, обладающие массой и координатами, взаимодействуют с гравитационным полем, дискретизируемым на регулярной

прямоугольной сетке. Шаг моделирования является фиксированным. На каждом шаге на сетке вычисляется плотность, после этого решается уравнение Пуассона методом Зейделя для нахождения гравитационного потенциала, затем вычисляются новые координаты и скорости частиц в зависимости от действующих на них сил.

Рисунок 18 является хорошей иллюстрацией того, что сборка мусора приносит в систему дополнительные накладные расходы, которые напрямую влияют на скорость работы программы. Вариант запуска без рекомендаций работает 38 секунд, когда время выполнения при работе сборщика мусора составило 65 секунд. Тем не менее, сборщик мусора снижает общее потребление памяти системой. В данном случае, пиковое потребление памяти при запуске со сборщиком мусора и при запуске без рекомендаций различается в 9172 кб.

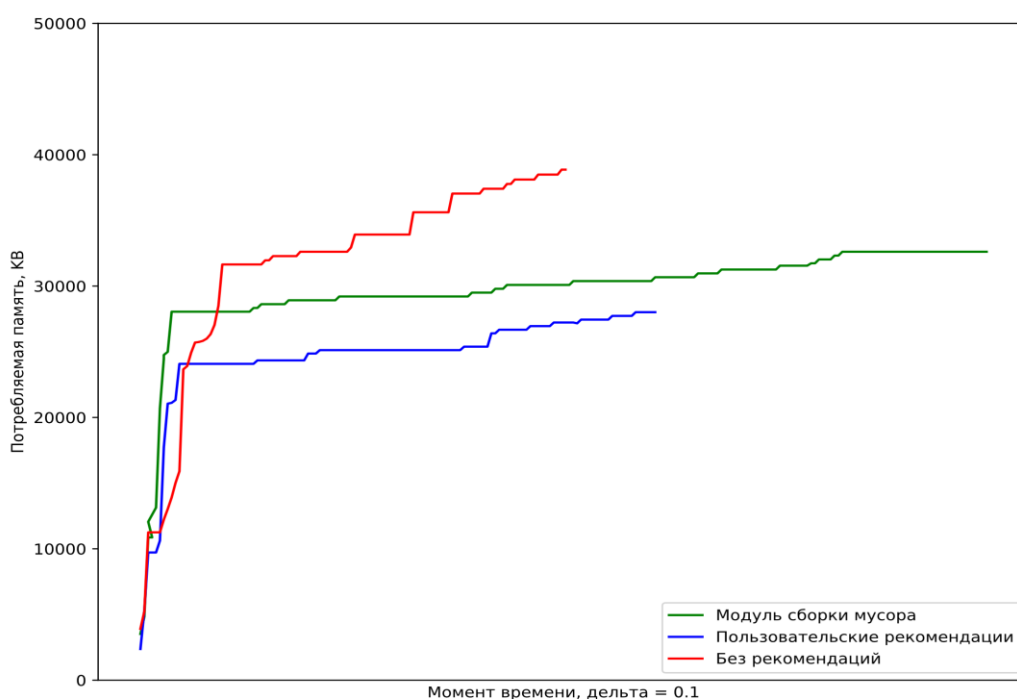


Рисунок 18 – График потребления памяти для разных вариантов запуска задачи моделирования самогравитирующего пылевого протопланетного диска методом частиц-в-ячейках

Результаты тестирования подтвердили работоспособность алгоритма сборки мусора для задач численного моделирования. Разработанный алгоритм

способен снизить потребление памяти программой на некотором классе вычислительных задач.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы был разработан, реализован и встроен в систему фрагментированного программирования LuNA специализированный алгоритм динамической распределенной сборки мусора, который снижает потребление памяти программой на итерационных методах численного моделирования.

Данная работы была опубликована на 61-й Международной научно-студенческой конференции, г. Новосибирск, 2023 г. и была награждена дипломом третьей степени [1].

Защищаемые положения:

1. Разработан и реализован специализированный алгоритм динамической распределенной сборки мусора в системе LuNA;
2. Проведено экспериментальное исследование работоспособности алгоритма динамической распределенной сборки мусора системе LuNA.

Разработка и реализация алгоритма динамической распределенной сборки мусора в системе LuNA снизила потребление памяти и улучшило производительность программ. Кроме того, практическая ценность работы подтверждается результатами тестов.

В дальнейшем планируется продолжить работу над автоматизацией сборки мусора в системе фрагментированного программирования LuNA. Возможные направления развития:

1. Разработка динамического сборщика мусора, который эффективно работает с косвенной адресацией;
2. Разработка универсального гибридного сборщика мусора.

Выпускная квалификационная работа выполнена мной самостоятельно и с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) и опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием для недопуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно»

Иванченко Данила Валерьевич

ФИО студента

Подпись студента

«_____» _____ 20 __ г.
(заполняется от руки)

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Иванченко, Д.В. Разработка и реализация специализированного алгоритма распределенной динамической сборки мусора в системе LuNA. / Д.В. Иванченко // Информационные технологии. Параллельные вычисления : Материалы 61-й Междунар. науч. студ. конф. 17–26 апреля 2023 г. /Новосиб. гос. ун-т. (в печати)
2. Малышкин В. Э. Технология фрагментированного программирования / В.Э. Малышкин // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. – 2012. – N 46(305). – С. 45 – 55.
3. Перепелкин В. А. Система LuNA автоматического конструирования параллельных программ численного моделирования на мультикомпьютерах: дис. канд. тех. наук 05.13.11 / Перепелкин В. А. – Новосибирск., 2022. – 135 с.
4. Bauer, M. Legion: Expressing Locality and Independence with Logical Regions / M. Bauer, S. Treichler, E. Slaughter, A. Aiken // SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis – 2012. – N 66 – P. 1 – 11.
5. D. I. Bevan, “Distributed garbage collection using reference counting,” in Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe. London, UK: Springer-Verlag, 1987, – P. 176 – 187.
6. D. Plainfosse and M. Shapiro, “A survey of distributed garbage collection techniques,” in Proc. Int. Workshop on Memory Management, Kinross, Scotland (UK), 1995.
7. G. E. Collins. A method for overlapping and erasure of lists. Communications of the ACM, 3(12). – P. 655 – 657, 1960.
8. Khayri A. Mohamed-Ali. Object-oriented storage management and garbage collection in distributed processing systems. Academic Dissertation, Royal Institute of Technology, Dept. of Computer Systems, Stockholm, Sweden, 1984.
9. Kireev S. A parallel 3D code for simulation of self-gravitating gas-dust systems // International Conference on Parallel Computing Technologies 2009. — Berlin,

Heidelberg : Springer, 2009.— С. 406–413.

10. Malyshkin V.E. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem / V.E. Malyshkin, V.A. Perepelkin // Proceedings of the 11th Conference on Parallel Computing Technologies.— Springer, 2011. – Vol. 6873 of Lecture Notes in Computer Science. – P. 53-61.

11. P. Watson and I. Watson, “An efficient garbage collection scheme for parallel computer architectures,” in Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe. London, UK: Springer-Verlag, 1987, P. 432–443.

12. R. J. M. Hughes, “A Distributed Garbage Collection Algorithm,” in Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture, Nancy, France, 1985, P. 256–272.

13. V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu. The Asynchronous Partitioned Global Address Space model. AMP, 2010.

14. W. Gropp, E. Lusk, and R. Thakur. Using MPI-2: Advanced Features of the MessagePassing Interface. — MIT Press, 1999.

15. William Gropp, Ewing Lusk, Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface / MIT Press: 3 изд. — 2014. — 336 С. — ISBN-13 978-0262527392.

16. Информационно-вычислительный центр Новосибирского государственного университета [Электронный ресурс]. – Режим доступа: <http://nusc.nsu.ru/> (дата обращения 18.05.2023).

17. Сборка мусора .NET — [Электронный ресурс] – Режим доступа: <https://learn.microsoft.com/ru-ru/dotnet/standard/garbage-collection/> (дата обращения: 25.05.2023).

18. A Guide to the Go Garbage Collector — [Электронный ресурс] – Режим доступа: <https://go.dev/doc/gc-guide> (дата обращения: 28.04.2023).

19. Java Garbage Collection Basics — [Электронный ресурс] – Режим доступа: <https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>

(дата обращения: 12.04.2023).

20. JSON – Режим доступа: <https://www.json.org/json-en.html> (дата обращения: 25.05.2023).

21. Linux manual page. top(1) — Linux manual page — [Электронный ресурс] / – Режим доступа: <https://man7.org/linux/man-pages/man1/top.1.html> (дата обращения: 25.05.2023).

22. The X10 Programming Language — [Электронный ресурс] — Режим доступа: <http://x10-lang.org/> (дата обращения: 15.04.2023).

23. Ubuntu Manpage: bison – GNU Project parser generator (yacc replacement) — [Электронный ресурс] – Режим доступа: <https://manpages.ubuntu.com/manpages/focal/man1/bison.1.html> (дата обращения: 11.05.2023).

24. Ubuntu Manpage: flex – fast lexical analyzer generator — [Электронный ресурс] – Режим доступа: <https://manpages.ubuntu.com/manpages/focal/man1/lex.1.html> (дата обращения: 11.05.2023).

ПРИЛОЖЕНИЕ А

МОДУЛЬ ДИНАМИЧЕСКОЙ СБОРКИ МУСОРА ДЛЯ СИСТЕМЫ “LuNA”

РУКОВОДСТВО ПРОГРАММИСТА

Листов 9
Новосибирск 2023

СОДЕРЖАНИЕ

Аннотация	444
1 Назначение и условия программы	455
1.1 Назначение программы	455
1.2 Функции, выполняемые программой	455
1.3 Условия, необходимые для выполнения программы	455
2 Характеристика программы	466
2.1 Описание основных характеристик программы	466
2.2 Описание основных особенностей программы	466
3 Обращение к программе	477
3.1 Описание процедур вызова программы	477
4 Входные и выходные данные	488
4.1 Организация используемой входной информации	488
4.2 Организация используемой выходной информации	488
5 Сообщения.....	49
6 Лист регистрации изменений	500

АННОТАЦИЯ

В данном документе приведено руководство программиста для модуля динамической сборки мусора в системе LuNA. Исходными языками программы является C++ и Python. Средства разработки – редакторы исходного кода CLion и PyCharm от компании JetBrains.

Основной функцией программы является удаление фрагментов данных при завершении их жизненного цикла. Оформление программного документа «Руководство оператора» произведено по требованиям ГОСТ 19.504-79 «ЕСПД. Руководство программиста» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением N 1)».

1 Назначение и условия программы

1.1 Назначение программы

Модуль динамической сборки мусора в системе LuNA предназначен для автоматического освобождения динамической памяти.

1.2 Функции, выполняемые программой

Программа на основе графа потреблений отслеживает жизненный цикл фрагментов данных и удаляет их после завершения цикла.

1.3 Условия, необходимые для выполнения программы

Чтобы программа могла исполняться, необходимо наличие у программиста компиляторов `g++`, `mpicxx`, библиотеки `openmpi`, интерпретатора Python версии 2.7, а также утилит `flex` и `bison`.

2 Характеристика программы

2.1 Описание основных характеристик программы

Модуль сборки мусора входит в состав исполнительной системой системы LuNA и производит отслеживание жизненного цикла фрагментов данных на основе графа потреблений для своевременного освобождения памяти.

2.2 Описание основных особенностей программы

Разработанная версия модуля на момент написания руководства использует рекомендации системы LuNA для построения входного графа потреблений фрагмента данных.

3 Обращение к программе

3.1 Описание процедур вызова программы

Генерация кода для графа потреблений на основе рекомендации происходит при вызове функции `parse_gc_graph` из файла `fcmp2`. Для отправки графа потреблений фрагментом вычислений используется функция `senf_gc_graph` класса `CF` исполнительной системы, для получения используется функция `save_gc_tree` этого же класса.

Функции `notify_for`, `notify_while`, `notify_if` из файла `fcmp2` генерируют соответствующий C++ код для отправки уведомлений фрагментами вычислений. На стороне исполнительной системы для отправки уведомлений фрагментами вычислений используются функции `notify_for`, `notify_if`, `notify_while` класса `CF`. Для обработки `u` используются функции `check_notify_if`, `check_notify_for`, `check_notify_while` класса `RTS`.

После получения сообщения или уведомления от фрагмента вычислений для обновления множества ожидания соответствующего фрагмента данных используется функция `update_gc_graph`.

Вершину графа представляет класс `Node`.

4 Входные и выходные данные

4.1 Организация используемой входной информации

Входная информация – абстрактное синтаксическое дерево компилируемой программы на языке системы LuNA с использованием соответствующей рекомендации для построения графа потреблений фрагмента данных в формате json.

4.2 Организация используемой выходной информации

Отсутствует.

5 Сообщения

В случае, если программа в системе LuNA компилируется с флагом `--verbose`, то при удалении фрагмента данных сборщиком мусора в терминал будет выведено сообщение в формате: “DF with ID: `<df_id>` was deleted”.

При невозможности построения графа потреблений фрагмента данных на основе рекомендации в терминал выводится следующее сообщение: “Unable to construct gc graph for DF `<df>`”.

ПРИЛОЖЕНИЕ Б

ДИНАМИЧЕСКИЙ СБОРЩИК МУСОРА ДЛЯ СИСТЕМЫ «LuNA»

ОПИСАНИЕ ПРОГРАММЫ

Листов 13

Новосибирск 2023

СОДЕРЖАНИЕ

Аннотация	53
1 Общие сведения	54
1.1 Обозначение и наименование программы	54
1.2 Программное обеспечение, необходимое для функционирования	54
программы	54
1.3 Языки программирования	54
2 Функциональное назначение	55
2.1 Назначение программы	55
2.2 Сведения о функциональных ограничениях на применение	55
3 Описание логической структуры	56
3.1 Алгоритм программы	56
3.2 Используемые методы	56
3.3 Структура программы	56
3.4 Связи между составными частями программы	58
3.5 Связи программы с другими программами	58
4 Используемые технические средства	59
5 Вызов и загрузка	60
6 Входные данные	61
7 Выходные данные	62
8 Лист регистрации изменений	63

АННОТАЦИЯ

В данном документе приведено описание модуля динамической сборки мусора в системе LuNA. Пользователь имеет возможность использовать модуль для автоматического освобождения динамической памяти LuNA-программы.

Чтобы программа могла исполняться, необходимо наличие у программиста компиляторов g++, mpicxx, библиотеки openmpi, интерпретатора Python версии 2.7, а также утилит flex и bison.

Оформление программного документа «Описание программы» произведено по требованиям ГОСТ 19.402-78 «ЕСПД. Описание программы» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением N 1)»

1 Общие сведения

1.1 Обозначение и наименование программы

Полное наименование программы – модуль динамической сборки мусора для системы «LuNA»

1.2 Программное обеспечение, необходимое для функционирования программы

Для функционирования программы необходимо наличие установленных компиляторов g++, mpісxx, библиотеки openssl, интерпретатора Python версии 2.7, а также утилит flex и bison.

1.3 Языки программирования

Исходными языками программы является C++ и Python. Для разработки использовались редакторы исходного кода CLion и PyCharm от компании JetBrains.

2 Функциональное назначение

2.1 Назначение программы

Программа предназначена для автоматического освобождения динамической памяти, выделенной для фрагментов данных, при исполнении LuNA-программ.

2.2 Сведения о функциональных ограничениях на применение

Программа не предназначена для запуска на платформах, на которых отсутствуют компиляторы g++, mpicxx, библиотеки openmpi, интерпретатора Python версии 2.7, а также утилиты flex и bison.

3 Описание логической структуры

3.1 Алгоритм программы

Программа на основе рекомендаций для исполнительной системы LuNA строит граф потреблений для каждого фрагмента данных. Далее, программа отслеживает жизненный цикл фрагментов данных, используя граф потреблений. После того, как того, как фрагмент данных завершил свой жизненный цикл, он удаляется модулем сборки мусора

3.2 Используемые методы

Алгоритм динамической сборки использует граф потреблений для определения момента времени, когда фрагмент данных может быть удален.

Алгоритм обрабатывает сообщения о запросе фрагмента данных от фрагментов вычислений, на основе идентификаторов фрагмента данных и фрагмента вычислений находит соответствующую вершину в множестве ожидания и удаляет ее, при этом добавляя дочерние вершины в множество ожидания. Когда множество ожидания опустело, фрагмент данных может быть удален.

3.3 Структура программы

Программа состоит из нескольких классов, которые были добавлены к существующим классам исполнительной системы LuNA, дополнительно к этому были модифицированы некоторые функции и методы классов исполнительной системы.

Таблица 4 – Описание классов программы

Класс	Описание
Node	Класс соответствует вершине графа потреблений.
RTS	Класс исполнительной системы. В его работу были добавлены обработка дополнительных форматов сообщений: <ol style="list-style-type: none">1. TAG_GC_NODE, соответствующий приходу графа потреблений на узел2. TAG_GC_NOTIFY_IF - уведомлению от фрагмента

	<p>вычислений, представляющий оператор if</p> <p>3. TAG_GC_NOTIFY_FOR - уведомлению от фрагмента вычислений, представляющий оператор for</p> <p>4. TAG_GC_NOTIFY_WHILE - уведомлению от фрагмента вычислений, представляющий оператор while</p> <p>Также для обновления множества вершин, которые ожидают прихода сообщений от соответствующих им фрагментов вычислений был добавлен метод <code>update_gc_graph</code>.</p> <p>Для обработки уведомлений от фрагментов вычислений операторов if, for, while были добавлены методы <code>check_notify_if</code>, <code>check_notify_for</code>, <code>check_notify_while</code> соответственно.</p> <p>Для отправки и получения графа потреблений в виде указателя на корень графа были добавлены методы <code>save_gc_graph</code> и <code>send_gc_graph</code></p>
CF	<p>Класс, представляющий фрагмент вычислений в системе. Для него были добавлен метод <code>send_gc_tree</code>, позволяющий фрагменту вычислений отправлять граф потреблений вместе с фрагментом данных. В данном методе используется описанная выше функция <code>send_gc_tree</code> класса RTS.</p> <p>Также были добавлены методы <code>notify_if</code>, <code>notify_for</code>, <code>notify_while</code> для отправки уведомлений фрагментами вычислений, которые представляют операторы if, for и while соответственно</p>
Id	<p>Класс, представляющий идентификатор фрагмента вычислений в системе. Для данного класса был добавлен метод <code>get_array_index</code> получения значения индекса в определенной позиции.</p>

3.4 Связи между составными частями программы

Связь между классами программы осуществляется при помощи вызова функций и методов объектов.

3.5 Связи программы с другими программами

Для запуска программы необходимо наличие установленной системы LuNA.

4 Используемые технические средства

Режим работы – консольное приложение. Для работы программы необходимо устройство со следующими требованиями:

1. Оперативную память объемом 2 ГБ и выше

5 Вызов и загрузка

Запуск программы осуществляется во время запуска LuNA-программы путем набора команды в терминале `luna <название .fa файла>`.

6 Входные данные

Входные данные программы – абстрактное синтаксическое дерево компилируемой программы на языке системы LuNA с использованием соответствующей рекомендации для построения графа потреблений фрагмента данных в формате json.

7 Выходные данные

Отсутствуют.

