**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ**

**УНИВЕРСИТЕТ**

**«МЭИ»**

Институт автоматики и вычислительной техники

Кафедра прикладной математики

**В Ы П У С К Н А Я Р А Б О Т А**

**Б А К А Л А В Р А**

по направлению 01.03.02 «Прикладная математика и информатика»

Тема:

реализация упреждающего параллелизма в интерпретаторе FPTL.

Студент А-05-14 Зубов М.И.

группа подпись фамилия, и., о.

Научный руководитель д.т.н., профессор Кутепов В. П.

должность звание подпись фамилия, и., о.

Консультант\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

должность звание подпись фамилия, и., о.

Консультант\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

должность звание подпись фамилия, и., о.

«Работа допущена к защите»

Зав. кафедрой ПМ д.т.н., профессор А.П.Еремеев

Дата\_\_\_\_\_\_\_\_\_\_\_\_\_

Москва 2018

# АННОТАЦИЯ

2018г. Зубов М.И.

Москва, НИУ «МЭИ»

Тема работы: реализация упреждающего параллелизма в интерпретаторе FPTL.

Объём работы с приложениями составляет 43 страницы.

Работа содержит 9 иллюстраций, 6 таблиц, 4 приложения.

Выполнена реализация, расширяющая возможности уменьшения времени параллельного выполнения функциональных программ на языке FPTL.

1) Разработан метод и алгоритм реализации упреждающих вычислений значений функций.

2) Метод реализован в виде расширения существующего интерпретатора языка.

3) Выполнены эксперименты на примерах FPTL программ, показывающие эффективность разработанных программных средств, нацеленных на ускорение выполнения упреждающих программ на FPTL.

Оглавление

[ВВЕДЕНИЕ 4](#_Toc519119658)

[1. ЯЗЫК FPTL И ЕГО РЕАЛИЗАЦИЯ 6](#_Toc519119659)

[1.1. Описание языка 6](#_Toc519119660)

[1.2. Интерпретатор языка 10](#_Toc519119661)

[2. УПРЕЖДАЮЩИЕ ВЫЧИСЛЕНИЯ И ОЦЕНКА ИХ ВЛИЯНИЯ НА УСКОРЕНИЕ ВЫПОЛНЕНИЯ ПРОГРАММ 18](#_Toc519119662)

[3. СОЗДАНИЕ ОПЕРАЦИОННЫХ СРЕДСТВ ДЛЯ РЕАЛИЗАЦИИ УПРЕЖДАЮЩИХ ВЫЧИСЛЕНИЙ 21](#_Toc519119663)

[3.1. Основной механизм 21](#_Toc519119664)

[3.2. Механизм перемещения задач из очереди упреждения в основную 23](#_Toc519119665)

[3.3. Механизм отмены задач 24](#_Toc519119666)

[4. РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТАЛЬНЫХ ИССЛЕДОВАНИЙ 27](#_Toc519119667)

[4.1. Задача с двумя сложными ветвями 27](#_Toc519119668)

[4.2. Задача с одной сложной ветвью и возрастающей сложностью 28](#_Toc519119669)

[4.3. Исследование влияния разработанного механизма на время выполнения программ без сложных условных конструкций 31](#_Toc519119670)

[ЗАКЛЮЧЕНИЕ 33](#_Toc519119671)

[СПИСОК ИСТОЧНИКОВ 34](#_Toc519119672)

[Приложение 1 35](#_Toc519119673)

[Приложение 2 36](#_Toc519119674)

[Приложение 3 37](#_Toc519119675)

[Приложение 4 38](#_Toc519119676)

## ВВЕДЕНИЕ

FPTL (Functional Parallel Typified Language) – язык функционального параллельного программирования, разработанный для параллельного выполнения программ, при написании которых программисту не требуется работать с инструментами для организации параллельных вычислений и явно указывать части программы, требующие параллельного выполнения.

Упреждающий параллелизм – параллельное выполнение частей программы, необходимость вычислений которых определяется результатом предшествующих операций (условный оператор – типичный случай). В программе с большим количеством сложных условных конструкций он является мощным инструментом для сокращения времени её выполнения [1].

**Целью** данной работы является разработка и реализация механизма упреждающего параллелизма в интерпретаторе FPTL.

Были поставлены и решены следующие **задачи**:

* определены условия, необходимые для эффективной работы механизма упреждающих вычислений;
* выполнен анализ внутреннего устройства текущей версии интерпретатора;
* определены способы модификации архитектуры интерпретатора;
* созданы операционные средства для эффективной реализации механизма упреждающего параллелизма;
* выполнена проверка корректности работы разработанных программных средств и проведены исследования достигаемых значений по ускорению выполнения программ.

В **первой** главе приведено описание языка и его основных возможностей, проведён анализ внутреннего устройства современной версии интерпретатора, построена упрощённая схема его работы и определены части кода, требующие модификации.

Во **второй** главе работы проведена оценка теоретически возможного ускорения при использовании упреждающих вычислений их эффективность в зависимости от параметров исполняемой программы, а также определены требования для корректной и эффективной реализации упреждающего параллелизма.

В **третьей** главе описана логика работы разработанных программных средств и их взаимодействия с остальными компонентами интерпретатора разрабатываемого расширения.

В **четвёртой** главе проведены тестирование и эксперименты для исследования ускорения, которое достигается при использовании разработанных программных средств управления упреждающими вычислениями.

В **заключении** перечислены результаты данной работы и подведены итоги экспериментальных исследований.

## ЯЗЫК FPTL И ЕГО РЕАЛИЗАЦИЯ

### Описание языка

FPTL – язык со строгой неявной динамической типизацией. Данные в нём представляются в виде кортежей – последовательностей разнотипных элементов. Функции в языке являются типизированными -арными соответствиями между кортежами данных, где – длина входного кортежа функции, – выходного. Константы – функции арности (0, 1) [1].

Для формирования новых кортежей в FPTL используются функция полного копирования кортежа данных *id* и функция , которая выбирает из него только -й элемент (1).

На данный момент в языке есть 4 встроенных типа данных: *bool*, *real*, *int*, *string*. А также существует понятие неопределенного значения - специального типа данных, необходимого для корректного выполнения программы в случае возникновения в функциях неоднозначных ситуаций. Обозначим его , тогда для любого кортежа выполняется свойство [2].

FPTL основан на композиции функций путем применения рекурсивных определений и трёх операций композиции [3].

1. Последовательная композиция «•»:

; .

1. Операция конкатенации кортежей значений функций (параллельная композиция) «\*»:

; .

1. Операция условной композиции «->»:

;

Все операции композиции обладают свойством ассоциативности и имеют следующий приоритет выполнения: «•», «\*», «->» (от наивысшего к меньшему).

Комбинируя последовательные и параллельные операции, программист может регулировать степень параллелизма в программе [4].

В примере 3 показано, как с помощью эквивалентных преобразований программы можно изменять количество операций последовательной и параллельной композиции.

В FPTL есть возможность представлять любой структурный тип данных. Для определения абстрактного типа данных (АТД) используются операции композиции, объединения множеств «++», функции-конструкторы и обратные к ним функции-деструкторы [3; 5; 6].

Операция объединения множеств «++» может быть использована только в блоке описания АТД и имеет следующее определение:

;

Все ключевые слова, начинающиеся с «c\_», определяются в языке как конструкторы, а начинающиеся с «~c\_», как деструкторы.

Конструктор – функция, которая представляет входные данные строго определённого в блоке описания АТД вида в качестве единицы данных нового типа.

Деструктор нераздельно связан с соответствующим конструктором. В общем случае, он определяется следующим образом:

В языке можно создавать параметризованные типы данных. Для этого после названия АТД необходимо добавить идентификаторы типов. Они должны всегда начинаться с символа «'» [3]. Описание параметризованного типа данных в программе можно увидеть в примере 1.

Схожим образом в FPTL задаются функционалы: после названия указываются функциональные параметры. Перед именем функционала обязательно должно быть ключевое слово «fun», которое означает, что следующая за ним функция имеет свой лексический контекст и может переопределять имена функциональных переменных, описанные в схеме. В примере 5 показано определение функционала.

С помощью макроса «@» можно заменять имя функции, в контексте (области видимости) которой он находится.

FPTL-программы имеют следующую структуру:

* блоки описания абстрактных типов данных (пример 1), могут быть опущены;
* блок описания функций (пример 2);
* блок вызова функциональных схем (пример 4).

Блоки программы обязательно должны описываться в приведённом выше порядке.

*Пример 1.* Описание АТД.

Data List['t]

{

List = c\_nil ++ 't \* List['t].c\_cons

}

В данном примере задаётся абстрактный тип данных – список элементов типа «'t».

*Пример 2.* Описание схемы.

Scheme ListConcat

{

@ = Concat.print;

Fun Concat

{

@ = [1].~c\_nil ->

[2], [2].~c\_nil ->

[1], ([1].~c\_cons.[1] \* ([1].~c\_cons.[2] \* [2]).@).c\_cons;

}

}

В этом примере задана функциональная схема, выполняющая объединение двух списков, описание которых производится в примере 1.

*Пример 3*. Эквивалентные преобразования.

Scheme ListConcat

{

ListConcat = Concat.print;

Fun Concat

{

Concat = [1].~c\_nil ->

[2], [2].~c\_nil ->

[1], ([1].~c\_cons \* [2]).([1] \* ([2] \* [3]).Concat).c\_cons;

}

}

Схема, эквивалентная схеме из примера 2.

*Пример 4.* Инициация выполнения схемы.

Application

List1 = (1 \* (2 \* c\_nil).c\_cons).c\_cons;

List2 = (3 \* (4 \* c\_nil).c\_cons).c\_cons;

%ListConcat(List1, List2)

В примере создаются два списка и вызывается схема их объединения.

*Пример 5.* Объявление функционала.

Fun Filter[fPredicate]

{

@ = [1].~c\_nil -> c\_nil, [1].~c\_cons ->

([1].~c\_cons \* [2]).(([1]\*[3]).fPredicate ->

([1] \* ([2]\*[3]).Filter).c\_cons, ([2]\*[3]).Filter);

}

Данный функционал располагает элементы списка относительно заданного в соответствии с передаваемой в него функцией сравнения.

### Интерпретатор языка

Проведём анализ внутреннего устройства текущей версии интерпретатора.

Для начала выполнения FPTL-программы из аргументов командной строки с помощью пакета «*program\_options*» библиотеки «*boost*» [10] считываются параметры для запуска:

1. путь к файлу с кодом программы;
2. количество создаваемых рабочих потоков;
3. принудительное отключение сборщики мусора;
4. вывод информации о работе сборщика мусора;
5. размер оперативной памяти в мегабайтах, отводимой для хранения недавно созданных объектов, также называемых «молодым поколением»;
6. размер памяти в мегабайтах для хранения объектов, существующих продолжительное время – «старое поколение»;
7. коэффициент заполнения памяти, выделенной под «старое поколение», при превышении которого в нём будет производиться сборка мусора.

Обязательным для указания является только параметр, задающий путь к исполняемой программе, значения остальных параметров при отсутствии соответствующего аргумента устанавливаются в значения по умолчанию. При этом будет использоваться 1 рабочий поток, сборщик мусора будет включен, информация о его работе выводиться не будет, под «молодое» поколение будет отведено 20Мб, под «старое» 100Мб, коэффициент заполнения будет равен 0,75.

Если аргументы указаны верно и файл по указанному пути существует, считывается код программы. Иначе будет выведено соответствующее сообщение об ошибке.

Далее осуществляется лексический и синтаксический разбор FPTL-программы, результатом которого является абстрактное синтаксическое дерево (АСД) [7, 8]. После этого производится семантическая проверка правильности кода программы. При нахождении каких-либо ошибок выводится информация о них и их местоположении в коде программы. Более подробное описание происходящих на данном этапе процессов приведено в таблице 1.1.

Таблица 1.1. Описание класса «*Support*» и его методов глобальной области видимости.

|  |
| --- |
| **Класс*****Support*** |
| Осуществляет сбор и хранение информации при синтаксическом разборе и проверке семантической правильности кода. |
| **Метод *getInternalForm*(*string*)** |
| Составляет таблицу имён - хеш-таблицу, которая ставит в соответствие каждой лексеме внутренний идентификатор парсера. Далее с помощью лексического анализатора, созданного при помощи «*Flex*», и синтаксического анализатора, который сгенерирован с использованием «*GNU Bison*», строит по входной строке абстрактное синтаксическое дерево.  Если АСД построено успешно, производит семантическую проверку: проверяет наличие объявлений используемых функций и структур данных, их доступность в текущем контексте и отсутствие идентичных имён, далее производит поиск и пометку рекурсивных функций.  Возвращает семантически корректное АСД или пустой указатель в случае обнаружения ошибок. |
| **Метод *getErrorList*()** |
| Выводит найденные в коде программы ошибки и информацию об их местоположении. |

После этого по АСД производится построение дерева, узлами которого являются функциональные схемы. Класс ответственный за этот процесс описан в таблице 1.2.

Затем по нему генерируется внутреннее представление программы – последовательность команд, которая в дальнейшем будет выполняться. В таблице 1.3 приведено описание происходящих на данном этапе процессов.

Представление программы передаётся в вычислитель схемы. Описание вычислителя приведено в таблице 1.4.

Табл.1.2. Описание класса «*FSchemeGenerator*» и его глобальных методов.

|  |
| --- |
| **Класс *FSchemeGenerator*** |
| Предназначен для построения дерева функциональных схем по АСД. |
| **Группа методов *visit*(*nodeType*)** |
| Генерирует функциональную схему для узла определённого типа.  Для констант создаётся класс *FConstantNode*, в который входят поля с описанием типа и значением константы.  Для строковых констант создаётся класс *FStringConstant*, в который входит поле со строкой.  Для операции получения элемента кортежа создаётся класс *FTakeNode*, в который входит поле с индексом элемента.  Для встроенных функций создаётся класс *FFunctionNode*, в который входят поля с именем и экземпляром функции.  Для пользовательских функций запускается генерация схем для их параметров и определения.  Для конструкторов создаётся класс *FFunctionNode*, в который входят поля с его именем и экземпляром функции, которая проверяет соответствие типов входного кортежа сигнатуре конструктора и создаёт АТД.  Для деструкторов создаётся класс *FFunctionNode*, в который входят поля с его именем и экземпляром функции, которая проверяет соответствие АТД данному деструктору и разворачивает АТД в кортеж.  Для переменных из блока вызова функциональных схем запускается процесс генерации схем их определения.  Для определений производится генерация схемы внутренних узлов, создаётся пара имя-схема и добавляется в множество схем определений.  Для операции условной композиции создаётся класс *FConditionNode*, в который входят поля со схемами ветвей и условия.  Для операции последовательной композиции создаётся класс *FSequentialNode*, в который входят поле со схемой предшествующих узлов и поле со схемой последующих узлов. Если подряд идёт несколько операций последовательной композиции, создаётся цепочка классов *FSequentialNode.*  Для операции параллельной композиции создаётся класс *FParallelNode*, в который входят поле со схемой узлов, находящихся слева от операции, и поле со схемой узлов, находящихся справа от операции. |

Табл.1.3. Описание класса «*Generator*» и его глобальных методов.

|  |
| --- |
| **Класс *Generator*** |
| Предназначен для генерации внутреннего представления программы (последовательности команд для выполнения) из дерева функциональных схем. |
| **Метод *generate(FSchemeNode)*** |
| Инициирует генерацию внутреннего представления программы. |
| **Группа методов *visit*(*FSchemeNode*)** |
| Генерирует внутреннее представление схемы определённого типа.  Для параллельной композиции инициируется генерация представлений для схем левой и правой ветвей. Если обе ветви являются «сложными», то создаются экземпляры классов *ParFork* и *ParJoin*, с помощью которых при выполнении представления для одной из ветвей будет создана отдельная задача. Ветвь считается «сложной», если она содержит рекурсивные функции или другие «сложные» функции.  Для последовательной композиции инициируется генерация представлений для схем предшествующих операций и последующих и создаются классы внутреннего представления *SeqBegin* и *SeqEnd*. Если подряд идёт несколько операций последовательной композиции, создаётся цепочка классов *SeqAdvance.* В каждом из этих классов хранится ссылка на следующее за ним представление.  Для условной композиции инициируется генерация представлений для схем ветвей и условия и создаются классы внутреннего представления *CondStart* и *CondChoose*. В *CondStart* хранится указатель на представление условия. В *CondChoose* на представления ветвей. При выполнении производится вычисление и проверка условия и выбирается одна из веток для дальнейшего выполнения.  Для пользовательских функций инициируется генерация представления и создаются классы внутреннего представления *RecFn* и *Ret*. В *RecFn* хранятся поля с именем функции, со ссылкой на представление следующей схемы и со ссылкой на представление данной функции. При выполнении ссылка на представление следующей схемы передаётся в класс *Ret* через вспомогательный стек. После этого инициируется выполнение данной функции и следующей схемы.  Для встроенных функций создаётся класс внутреннего представления *BasicFn*, в который передаются значения для полей со ссылкой на представление следующей схемы, именем функции, позицией в коде и экземпляром функции. При выполнении вычисляется значение функции и добавляется в стек вычисленных значений.  Для операции получения элемента кортежа создаётся класс внутреннего представления *GetArg*, который хранит индекс элемента и ссылку на представление следующей схемы. При выполнении добавляет значение элемента в стек вычисленных значений.  Для констант создаётся класс внутреннего представления *Constant*, который хранит её значение и ссылку на представление следующей схемы. При выполнении добавляет значение константы в стек вычисленных значений.  Для строк создаётся класс внутреннего представления *BasicFn*, в который передаются значения для полей со ссылкой на представление следующей схемы, именем функции, позицией в коде и экземпляром функции, которая выделяет в куче память, копирует в неё строку и возвращает структуру с информационными полями и методами для работы с кучей. |

Табл.1.4. Описание класса «*SchemeEvaluator*» и его глобальных методов.

|  |
| --- |
| **Класс *SchemeEvaluator*** |
| Производит вычисления программы, заданной функциональной схемой. Порождает, владеет и управляет всеми потоками. |
| **Метод *run(SExecutionContext, numEvaluators)*** |
| Инициирует выполнение программы.  Создаёт экземпляр сборщика мусора, заданное количество потоков и исполнительных единиц (*EvaluatorUnit*) и контекст выполнения программы (основное задание).  Контекст выполнения программы добавляется в очередь одной из исполнительных единиц. После этого потокам назначаются вычислительные единицы и начинается их выполнение.  После завершения работы всех потоков удаляет вычислительные единицы и выводит время выполнения программы. |
| **Метод *stop()*** |
| Инициирует завершение всех потоков. Потоки могут остановиться только после того, как завершат выполнение текущей задачи, так как прерывание выполнения задачи не безопасно и может привести к таким проблемам, как неопределённое состояние программы, утечки памяти, попытки доступа к внешним участкам памяти. |

|  |
| --- |
| **Метод *findJob(EvaluatorUnit)*** |
| Поиск задания в очередях других вычислителей. Возвращает нулевой указатель, если заданий нет. |
| **Метод *markRoots(ObjectMarker)*** |
| Запуск процесса пометки данных всех актуальных заданий, для последующей сборки мусора. |

Вычислитель инициализирует сборщик мусора, создаёт рабочие потоки соответственно количеству, заданному в параметрах запуска, и назначает каждому созданному потоку свою исполнительную единицу.

Каждая исполнительная единица содержит свою неблокирующую очередь задач, список выполняющихся задач, по которому сборщик мусора производит маркировку, и набор методов для работы с ними. Методы исполнительной единицы приведены в таблице 1.5.

Внутреннее представление программы предаётся в очередь задач одной из исполнительных единиц, и начинается процесс его выполнения.

Табл.1.5. Описание класса «***EvaluatorUnit***» и его глобальных методов.

|  |
| --- |
| **Класс *EvaluatorUnit*** |
| Вычислитель схем. |
| **Метод *evaluateScheme()*** |
| Запускает процесс поиска задачи, пока не будет произведена остановка потока. После остановки выводит информацию о количестве выполненных данным потоком операций с заданиями. |
| **Метод *addJob (SExecutionContext)*** |
| Добавляет задачу в список ожидающих выполнения и в очередь задач. |
| **Метод *join()*** |
| Ожидает завершения выполнения верхней в списке ожидающих выполнения задачи. Пока происходит ожидание и текущая задача не может быть завершена, запускает процесс поиска другой задачи для выполнения. |

|  |
| --- |
| **Метод *safePoint()*** |
| Производит приостановку потока для маркировки данных, если запланирована сборка мусора. |
| **Метод *stealJob()*** |
| Получение в своей очереди задачи для другой исполнительной единицы. |
| **Метод *schedule()*** |
| Производит поиск задачи для выполнения. |
| **Метод *markDataRoots(ObjectMarker \* marker)*** |
| Пометка данных всех актуальных заданий, порождённых текущей исполнительной единицей, для последующей сборки мусора. |

Если при выполнении операции параллельной композиции обе соединяемые ею ветви внутреннего представления являются «сложными», то для выполнения одной из них создаётся отдельная задача, а вторая продолжает выполняться в текущем потоке. Ветвь считается «сложной», если она содержит рекурсивные функции или другие «сложные» функции. В случае, если задание было создано, операция композиции может быть завершена только после завершения этого задания и копирования результатов его выполнения в текущий контекст. Поэтому после выполнения своей ветви, пока ожидается выполнение задачи, создавший её поток начинает поиск и выполнение других задач. Если созданную задачу ещё не начал выполнять другой поток, то на выполнение будет взята она, так как поиск задач начинается в очереди своей исполнительной единицы.

Выполнение программы завершается, когда будет завершено выполнение основной задачи. После этого будет выведена информация о работе и завершены созданные потоки. Упрощённая схема работы интерпретатора показана на рисунке 1.1.

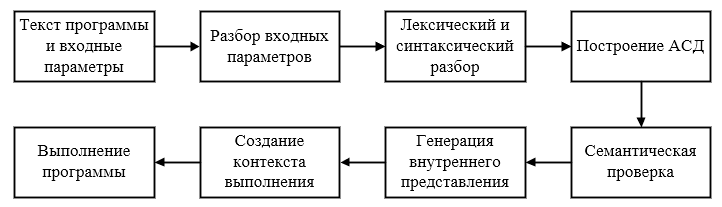


Рис.1.1. Общая схема работы интерпретатора.

Можно выделить следующие части интерпретатора, которые потребуют изменений в ходе реализации:

* метод генерации внутреннего представления операции условной композиции;
* метод, отвечающий за создание новых задач;
* класс исполнительной единицы;
* классы внутреннего представления узлов.

## УПРЕЖДАЮЩИЕ ВЫЧИСЛЕНИЯ И ОЦЕНКА ИХ ВЛИЯНИЯ НА УСКОРЕНИЕ ВЫПОЛНЕНИЯ ПРОГРАММ

Эффективность упреждающих вычислений во многом зависит от особенностей самой выполняемой программы, таких как вычислительная сложность различных её частей, степень ветвления в ней, вероятность того или иного сценария выполнения программы.

Непосредственно от реализации зависит время, требуемое для синхронизации, для создания копий данных, которые изменяются в частях программы, выполняемых с упреждением, и время, необходимое для прерывания упреждающих вычислений, результаты выполнения которых в дальнейшем не понадобятся.

Пусть для некоторой программы – среднее время выполнения условия «if» и ветвей «then», «else» в условной конструкции соответственно, –вероятность выполнения ветви «then», а – вероятность выполнения ветви «else». Тогда среднее время выполнения условной конструкции в этой программе при последовательном выполнении задаётся формулой (1), а при параллельном выполнении формулой (2). Тогда ускорение определяется формулой (3).

(1)

(2)

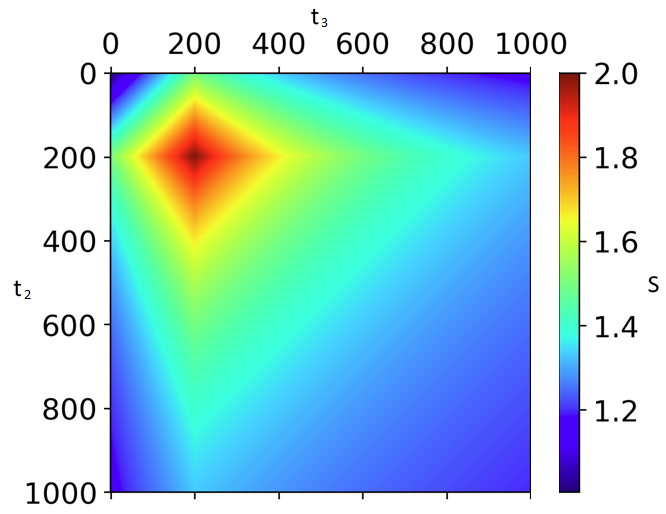


Рис.1. Ускорение при и .

Нетрудно заметить, что наибольшее ускорение достигается при и равно 2, при этом эффективность . Рисунок 1 иллюстрирует, как зависит ускорение от времени выполнения условия и ветвей if и then. Видно, что при увеличении разницы между временами выполнения ветвей и условия, ускорение достаточно резко убывает.

Очевидно, что при малой вычислительной сложности условия нет смысла использовать упреждающий параллелизм, как и в случае, если обе ветви «then» и «else» не являются вычислительно сложными, так как время, затрачиваемое на прерывание вычислений, передачу данных и синхронизацию, будет превышать возможный выигрыш от реализации параллелизма.

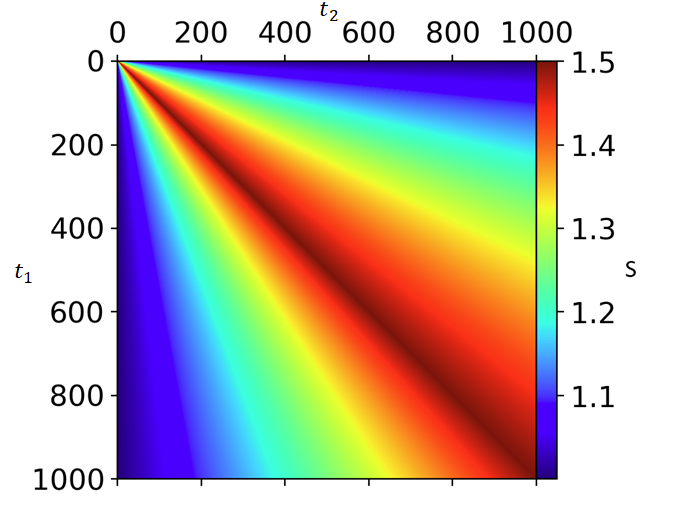


Рис.2. Ускорение при , и .

Наиболее частой является ситуация, когда только одна из ветвей вычислительно сложная. Пусть , тогда среднее время параллельного выполнения будет задаваться формулой (4), а ускорение формулой (5).

(4)

(5)

Наибольшее ускорение достигается при , причём оно сильно зависит от вероятности выбора «долгой» ветви. При равновероятном выборе ветвей максимально возможное среднее ускорение будет равно 1,5, а эффективность . Этот случай показан на рисунке 3. Видно, что если и различаются не более чем в 2 раза, то ускорение будет не меньше 1,3, а эффективность .

Чтобы эффективность выполнения программы в целом не снижалась, неупреждающие задачи должны иметь более высокий приоритет и должен быть механизм для прерывания выполняемых упреждающих задач после того, как станет известно, что их результаты не потребуются.

Также нужно отметить, что для получения корректных результатов упреждающие задачи должны быть явно различимы с неупреждающими и не должны оказывать влияния на общие с другими узлами ресурсы до момента вычисления условия и выбора «правильной» ветви [9].

## СОЗДАНИЕ ОПЕРАЦИОННЫХ СРЕДСТВ ДЛЯ РЕАЛИЗАЦИИ УПРЕЖДАЮЩИХ ВЫЧИСЛЕНИЙ

### Основной механизм

В генераторе внутреннего представления (класс *Generator*) при посещении условного узла схемы производится проверка: если условие сложное, то для сложных ветвей производится генерация представления только узлов, находящихся в этой ветви; иначе вместе с узлами, следующими за условной конструкцией.

После этого создаётся экземпляр класса внутреннего представления «*CondChoose*». В его конструктор передаются представление узлов, следующих за условной конструкцией, и представления простых ветвей при их наличии или нулевые указатели.

Далее генерируется внутреннее представление условия вместе с созданным ранее экземпляром класса. Это представление и представления сложных ветвей при их наличии или нулевые указатели, передаются в конструктор класса «*CondStart*», который будет являться результирующим внутренним представлением условной конструкции.

При выполнении сгенерированного представления в классе «*CondStart*» для ветвей, на которые получен ненулевой указатель, с помощью метода «*spawn*» создаются отдельные задачи и начинается выполнение условия. У порождённых таким образом задач флаг упреждения и флаг, показывающий начало нового уровня вложенности упреждающих задач, выставляются в истинное значение.

В классе «*CondChoose*» на основании вычисленного результата условия производится выбор «правильной» ветви. Если указатель на «неверную» ветвь нулевой, то есть она сложная и для неё ранее была создана отдельная задача, то эта задача отменяется. Если указатель на представление «верной» ветви ненулевой, то есть она простая, начинается её выполнение. Иначе вызывается метод «*join*» контекста выполнения (класс *SExecutionContext*), который ожидает завершения задачи вычисления ветви и копирует результат в конец кортежа данных. После этого начинается выполнение узлов, следующих за условной конструкцией.

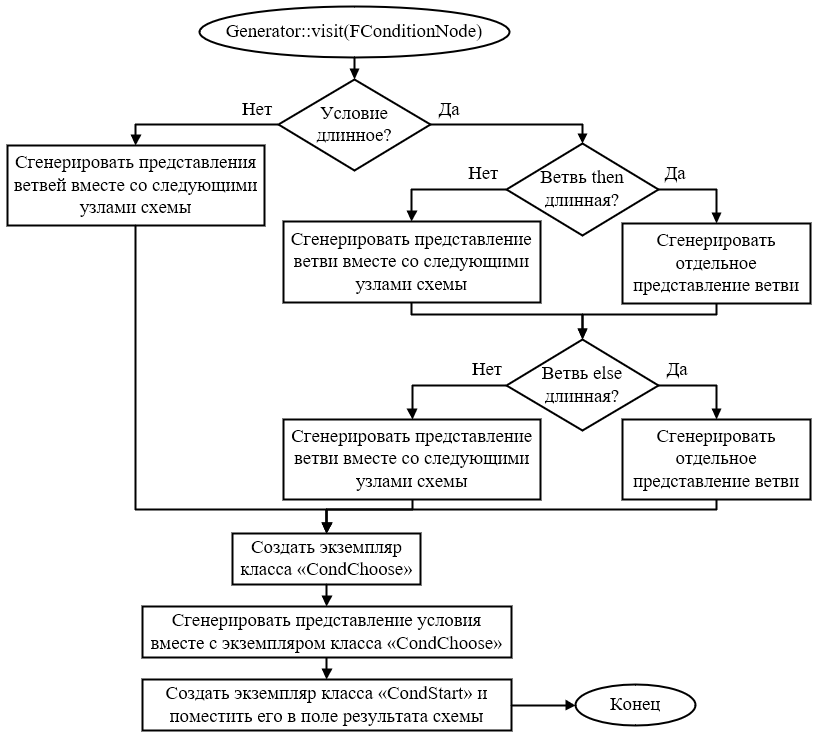


Рис.3.1. Алгоритм обработки узла условной конструкции генератором внутреннего представления.

В классе исполнительной единицы (*EvaluatorUnit*) был создан дополнительный экземпляр неблокирующей очереди для хранения упреждающих задач, который имеет более низкий приоритет при поиске задач для выполнения.

Также в классе исполнительной единицы были реализованы новые методы:

* «*addForkJob*(*task*)». Добавляет задачу в список задач, которые не должны быть удалены сборщиком мусора, после этого на основании флага упреждения добавляет задачу в соответствующую очередь выполнения.
* «*stealAnticipationJob*». Позволяет другому вычислителю взять задачу из упреждающей очереди.

В методе «*schedule*», который осуществляет поиск задачи для выполнения, после просмотра основных очередей добавлен поиск по очередям упреждения. При получении упреждающей задачи выполнение производится только в случае, когда флаг готовности ложен, то есть задача не отменена, и флаг упреждения истинен – задача не перенесена в основную очередь.

В класс вычислителя схемы добавлен метод «*findAnticipationJob*», который позволяет осуществлять поиск задачи в упреждающих очередях других вычислительных единиц.

### Механизм перемещения задач из очереди упреждения в основную

В классе исполнительной единицы (*EvaluatorUnit*) был создан метод «*moveToMainOrder*(*task*)».

Если родительская задача не является упреждающей, выставляет флаг упреждения в ложное состояние с целью предотвратить выполнение этой задачи из упреждающей очереди. После этого, если задача ещё не завершена и не выполняется, добавляет её в основную очередь и инициирует перемещение порождённых ею задач, которые не являются новым уровнем упреждающих задач. Флаг нового уровня вложенности упреждения выставляется в ложное значение, чтобы при перемещении родительской задачи в основную очередь эта задача также переместилась, если родительская задача была упреждающей.

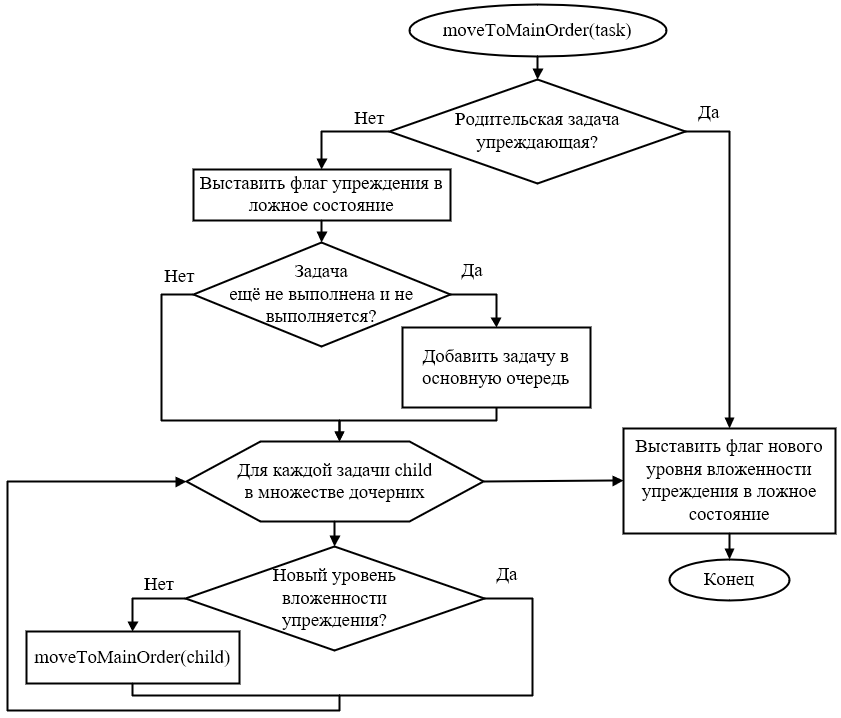


Рис.3.2. Алгоритм перемещения задачи в основную очередь.

### Механизм отмены задач

В классе исполнительной единицы (*EvaluatorUnit*) были реализованы методы:

* «*cancelFromPendingEnd*(*pos*)». Инициирует удаление задачи из позиции *pos* с конца списка выполняющихся задач (нумерация начинается с единицы). Задача удаляется из множества дочерних задач родителя, чтобы при удалении родительской задачи не был инициирован повторный процесс удаления задачи. После этого задача удаляется из списка задач, которые не должны быть удалены сборщиком мусора.
* «*cancel*(*task*)». Если задача ещё не выполнена, выставляет флаг готовности в истинное значение, чтобы выполнение задачи не началось. После этого, если задача уже выполняется, начинается замена представлений узлов задачи на завершающую операцию с помощью группы методов «zeroing». Указатели на заменённые представления при этом сохраняются в контексте выполнения для предотвращения их преждевременного удаления. Отменяемая задача корректно завершит порождённые задачи и удалит их из списка выполняющихся задач.

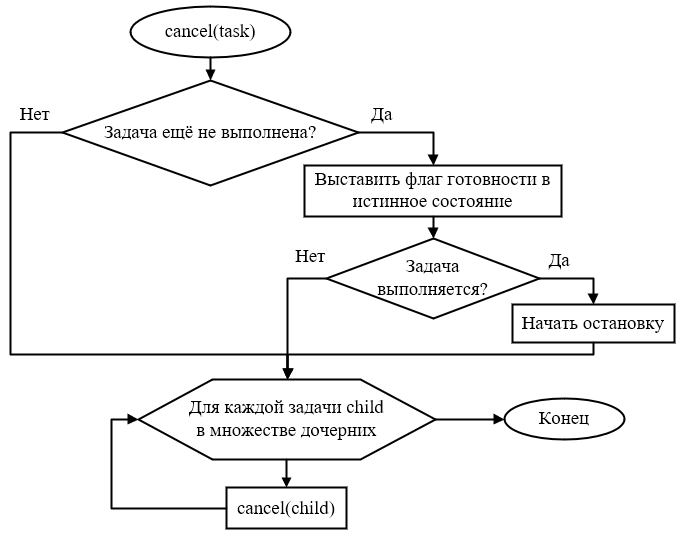


Рис.3.3. Алгоритм работы отмены задания.

В методе «*spawn*» дополнительно к своим старым функциям теперь устанавливается флаг упреждения порождаемой задачи на основании значения этого флага в родительской задаче и добавляется указатель на порождаемую задачу в множество порождённых задач родителя. Для хранения дочерних задач используется множество, а не хеш-таблица, так как в подавляющем большинстве случаев количество порождаемых задач невелико и операции на сбалансированном бинарном дереве будут производиться быстрее [11], чем вычисления хеш-значений.

В метод «*join*» контекста выполнения (*SExecutionContext*) перед копированием результата добавлена проверка, что ожидаемая задача не отменена.

В методе «*join*» исполнительной единицы (*EvaluatorUnit*) пред удалением задачи из списка выполняющихся задач производится её удаление из множества порождённых задач родителя. Это необходимо, чтобы не было инициировано удаление уже завершённой задачи, если родительская задача будет отмена.

## РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТАЛЬНЫХ ИССЛЕДОВАНИЙ

Исследование эффективности работы разработанного механизма и его реализации проводились на компьютерной системе, характеристики которой приведены в таблице 4.1. На время тестирования работа фоновых процессов сводилась к минимуму для получения более точных результатов. Все приводимые величины времени выполнения программы – медианы из 15 запусков.

Табл.4.1. Характеристики тестовой компьютерной системы.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Процессор | | | | | | | |
| Intel Xeon E5-2670; 3,0ГГц; 8 вычислительных ядер | | | | | | | |
| Кеш-память: | L1 Instruction | | L1 Data | | L2 | | L3 |
| 8 x 32Кб | | 8 x 32Кб | | 8 x 256Кб | | 20Мб |
| ОЗУ | | | | | | | |
| 48ГБ 1333МГц Quad-channel 9-9-9-24-174-2T | | | | | | | |
| Операционная система | | | | | | | |
| Название | | Разрядность | | Версия | | Сборка | |
| Windows 10 | | x64 | | 1803 | | 17134.112 | |

### Задача с двумя сложными ветвями

Данный эксперимент позволяет оценить эффективность разработанных операционных средств при выполнении условных конструкций с вычислительно сложными ветвями и условием.

Запускаемая FPTL-программа представляет из себя условную конструкцию, в условии и ветвях которой содержатся рекурсивные функции для нахождения суммы числового ряда, имеющие одинаковую вычислительную сложность. При использовании 3 рабочих потоков должно достигаться ускорение близкое к 2.

Алгоритм программы:

Если , вернуть ,

иначе , где – случайное число в диапазоне [0, 1].

В Приложении 1 приведён код данной программы.

Табл.4.2. Ускорение и время выполнения программы в зависимости от количества используемых рабочих потоков.

|  |  |  |  |
| --- | --- | --- | --- |
| Количество используемых рабочих потоков | 1 | 2 | 3 |
| Время выполнения, с | 10,6 | 5,4 или 10,7  среднее 8,1 | 5,4 |
| Ускорение | 1,00 | 1,31 | 1,96 |

Видно, что полученные значения ускорения близки к теоретическому максимуму, показанному в главе 2, что говорит о высокой эффективности разработанных операционных средств.

### Задача с одной сложной ветвью и возрастающей сложностью

В качестве примера использования разработанного механизма в реальных задачах используется программа численного интегрирования методом трапеций [13], оптимизированная для упреждающих вычислений.

В условной конструкции одна длинная ветвь, вероятность выполнения которой высока при большой точности вычислений. Каждая итерация содержит две сложные задачи: вычисление интеграла с шагами h и h/2. Сложность задач с каждой итерацией возрастает в четыре раза. Погрешность оценивается по правилу Рунге [13]. В реализации интерпретатора без упреждающих вычислений итерации выполняются последовательно.

В качестве подынтегральной взята функция. Интегрирование проводится на интервале с точностью .

Алгоритм программы выглядит следующим образом:

1. , где – левая граница отрезка интегрирования, – правая;
2. Если , то переходим к пункту 3, иначе к пункту 4.

Здесь – точность, – значение интеграла функции , вычисленное методом трапеции с шагом , а – с шагом .

1. Выводим результат .
2. , переходим к пункту 2.

Код FPTL-программы находится в Приложении 2.

Табл.4.3. Ускорение и время выполнения программы численного интегрирования в зависимости от реализации и количества используемых рабочих потоков.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Количество используемых рабочих потоков | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Время выполнения в интерпретаторе без упреждающих вычислений, с | 9,6 | 6,5 | 6,5 | 6,5 | 6,5 | 6,5 | 6,5 | 6,5 |
| Ускорение | 1,00 | 1,48 | 1,48 | 1,48 | 1,48 | 1,48 | 1,48 | 1,48 |
| Время выполнения в интерпретаторе с упреждающими вычислениями, с | 9,6 | 6 | 5,4 | 5 | 5,2 | 5,4 | 5,6 | 5,8 |
| Ускорение | 1,00 | 1,60 | 1,78 | 1,92 | 1,85 | 1,78 | 1,71 | 1,66 |

Из таблицы 4.3 и графиков на рисунках 4.1 и 4.2 можно заметить, что при использовании более 4 рабочих потоков происходит замедление выполнения программы. Это связано с увеличением количества узлов функциональной схемы на каждой следующей итерации и количеством одновременно выполняющихся упреждающих задач, что, в свою очередь, приводит к появлению дополнительных накладных расходов на управление, в частности, при их отмене в момент завершения вычислений.

Рис.4.1. Время выполнения программы численного интегрирования.

Рис.4.2. Ускорения выполнения программы численного интегрирования.

Для определения минимального теоретически возможного времени выполнения программы при использовании упреждающих вычислений было запущено вычисление интеграла с шагом, который используется на завершающей итерации. Время выполнения составило 4,9с. При использовании 4 рабочих потоков это значение практически достигается.

Из этого эксперимента можно сделать вывод о неэффективности использования большой глубины упреждения для такого типа задач.

### Исследование влияния разработанного механизма на время выполнения программ без сложных условных конструкций

Решим ту же задачу интегрирования, но используя алгоритм с высокой степенью зернистости.

Эта программа менее эффективна, так как из-за большого количества создаваемых задач, время, затрачиваемое на управление, занимает значительную часть времени выполнения программы. Однако такой алгоритм позволяет увидеть, насколько увеличились накладные расходы после реализации механизмов упреждающего параллелизма.

Чтобы выполнение программы длилось не больше нескольких минут, вычисления проводятся с меньшей точностью: .

Алгоритм программы:

1. , , , где – левая граница отрезка интегрирования, – правая, f(x) – подынтегральная функция;
2. вывести результат , где – точность.

Алгоритм :

1. , , , , ;
2. если , то переходим к пункту 3, иначе к пункту 4.
3. Возвращаем .
4. Возвращаем результат

.

Код FPTL-программы приведён в Приложении 3.

Табл.4.4. Ускорение и время выполнения программы численного интегрирования в зависимости от реализации и количества используемых рабочих потоков.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество используемых рабочих потоков | 1 | 2 | 4 | 8 |
| Время выполнения в интерпретаторе без упреждающих вычислений, с | 68,9 | 35 | 17,5 | 10,7 |
| Время выполнения в интерпретаторе с упреждающими вычислениями, с | 71 | 36 | 18 | 11 |
| Изменение времени выполнения | +3% | +3% | +3% | +3% |

Рис.4.3. Время выполнения программы численного интегрирования в зависимости от реализации и количества выделенных вычислительных ядер.

Из таблицы 4.4 и графика на рисунке 4.3 можно видеть, что реализованные средства для упреждающих вычислений увеличивают время выполнения программ с высокой степенью зернистости на 3%. Таким образом можно сделать вывод, что для сохранения максимальной производительности в задачах без сложных условных конструкций будет полезно отключать механизм упреждающих вычислений.

# ЗАКЛЮЧЕНИЕ

Целью данной работы являлась разработка и реализация механизма упреждающего параллелизма в интерпретаторе FPTL. Выделим основные результаты данной работы.

1. Описано устройство интерпретатора FPTL.
2. Исследовано влияние параметров выполняемой задачи на эффективность упреждающих вычислений.
3. Созданы программные средства для выполнения упреждающих вычислений в FPTL-программах.
4. Проведены экспериментальные исследования эффективности разработанной реализации.

Из проведённых экспериментов видно, что разработанные средства показывают себя весьма эффективно при выполнении программ, использующих упреждающие вычисления, и не оказывает сильного негативного влияния на время выполнения программ без сложных условных конструкций.

СПИСОК ИСТОЧНИКОВ

1. *Бажанов C.Е., Кутепов В.П., Шестаков Д.А.* Язык функционального параллельного программирования и его реализация на кластерных системах. // Программирование. 2005, № 5.
2. *Бочаров И.А., Кутепов В.П., Шамаль П.Н.* Система типового контроля программ на языке функционального программирования FPTL. // РФФИ №13-07-00810 2014г.
3. *Шамаль П.Н.* Разработка и исследование методов и программных средств параллельного выполнения функциональных программ на многоядерных компьютерах. //Диссертация на соискание ученой степени кандидата технических наук. 2014г.
4. *Бажанов C.Е., Кутепов В.П., Шестаков Д.А.* Структурный анализ и планирование процессов параллельного выполнения функциональных программ. // Изв. РАН. ТиСУ, 2005. № 6;
5. *Кутепов В.П., Фальк В.Н.* Направленные отношения: теория и приложения. // Изв. РАН. Техн. кибернетика. 1994. № 4, 5.
6. *Бажанов С.Е., Кутепов В.П., Шестаков Д.А*. Разработка и реализация системы функционального параллельного программирования на вычислительных системах. // Доклады международной научной конференции «Суперкомпьютерные системы и их применение» SSA’2004. Мн.: ОИПИ НАН Беларуси, 2004г.
7. *Ахо А.В., Лам М.С., Сети Р., Ульман Д.Д.* Компиляторы: принципы, технологии и инструментарий. 2 изд. // Москва, Вильямс, 2018г.
8. Abstract Syntax Tree. //Интернет ресурс: <https://ps-group.github.io/compilers/ast>
9. *Кутепов B.П, Фальк В.Н.* «Формы, языки представления, критерии и параметры сложности параллелизма.», 2010г.
10. Boost Library Documentation. //Интернет ресурс: https://www.boost.org/doc/libs/
11. Знай сложности алгоритмов. //Интернет ресурс: https://habr.com/post/188010/
12. Основы сборки мусора. //Интернет ресурс:

https://docs.microsoft.com/ru-ru/dotnet/standard/garbage-collection/fundamentals

1. *Амосов А.А., Дубинский Ю.А., Копченова Н.В.* Вычислительные методы для инженеров. // Москва, Высшая школа, 1994г.

# Приложение 1

Scheme Test

{

@ = TestSum.print;

TestSum = (([1] \* [2] \* 1).Sum(Func1) \* 0).greater ->

([1] \* [2] \* 1).Sum(Func2),

([1] \* [2] \* 1).Sum(Func3);

Func1 = [1].sin;

Func2 = [1].atan;

Func3 = [1].cos;

//Sum[f(x)](a,b,h) = if (a==b) f(a) elseif (a>b) 0 else f(a) + f(b) + Sum(a+h,b-h,h)

Fun Sum[fFunction]

{

@ = ([1] \* [2]).equal ->

[1].fFunction,

([1] \* [2]).greater ->

0,

(rand.fFunction \*

(rand.fFunction \*

( ([1] \* [3]).add \*

([2] \* [3]).sub \*

[3]

).Sum

).add

).add;

}

}

Application

%Test(0.0,30000000.0)

# Приложение 2

Scheme Test

{

@ = Integrate(Func); //(a, b, Eps, p)

//f(x) = sin(0,5\*x^3)\*sin(0.25\*x^2)\*sin(0.125\*x)

Func = ( (((([1]\*[1]).mul\*[1]).mul \* 0.5).mul.sin \*

(([1]\*[1]).mul \* 0.25).mul.sin).mul \*

([1] \* 0.125).mul.sin

).mul;

Fun Integrate[fFunction]

{

@ = ([3] \* [1] \* [2] \* ([2] \* [1]).sub).

( [1] \* [2] \* [3] \* ([4] \* 2).div \* ([4] \* 4).div

).Integr;

Integr = ( (([2] \* [3] \* [5]).Trp \* ([2] \* [3] \* [4]).Trp \* [1]

).(Rungreater -> true, false \* [1].print.("\n".print)) ->

( [1] \* [2] \* [3] \*

([5] \* 2).div \* ([5] \* 4).div

).Integr);

Rungreater = (Runge \* [3]).greater;

Runge = (([1] \* [2]).sub.abs \* 3).div;

//Trp[f(x)](a,b,h) = h\*((f(a)+f(b))/2+Sum(a+h,b-h,h))

Trp = ([3] \* ( ( ([1].fFunction \* [2].fFunction).add \* 2).div \*

(([1] \* [3]).add \* ([2] \* [3]).sub \* [3]).Sum

).add

).mul;

//Sum[f(x)](a,b,h) = if (a==b) f(a) elseif (a>b) 0 else f(a) + f(b) + Sum(a+h,b-h,h)

Sum = ([1] \* [2]).equal ->

[1].fFunction,

([1] \* [2]).greater ->

0,

([1].fFunction \*

([2].fFunction \*

(([1] \* [3]).add \* ([2] \* [3]).sub \* [3]).Sum

).add

).add;

}

}

Application

%Test(0.0 \* 25.0 \* 0.0000000001)

# Приложение 3

Scheme Test

{

@ = Integrate(Func).print;

//f(x) = sin(0,5\*x^3)\*sin(0.25\*x^2)\*sin(0.125\*x)

Func = ( ( ((([1]\*[1]).mul\*[1]).mul \* 0.5).mul.sin \*

(([1]\*[1]).mul \* 0.25).mul.sin).mul \*

([1] \* 0.125).mul.sin).mul;

// Параметры A\*B\*Eps - границы отрезка интегрирования и точность.

Fun Integrate[fFunction]

{

@ = ([3] \* [1] \* [2] \* [1].fFunction \* [2].fFunction

).(id \* ([2] \* [3] \* [4] \* [5]).Trp).Integr;

Integr = ( id \*

([2] \* [3]).Mid

).( id \*

[7].fFunction

).( id \*

([2] \* [7] \* [4] \* [8]).Trp \*

([7] \* [3] \* [8] \* [5]).Trp

).( id \*

([9] \* [10]).add

).( (([6]\*[11]).sub.abs \* 3).div \*

id \*

([1] \* 2).div

).(less ->

[12],

( ([13] \* [3] \* [8] \* [5] \* [9] \* [10]

).Integr \*

([13] \* [8] \* [4] \* [9] \* [6] \* [11]

).Integr

).add);

Mid = (([1] \* [2]).add \* 2.0).div;

//Trp(a, b, f(a), f(b)) = (b-a)\*(f(a)+f(b))/2

Trp = ((([2] \* [1]).sub \* ([3] \* [4]).add).mul \* 2.0).div;

}

}

Application

%Test(0.0 \* 25.0 \* 0.000001)

# Приложение 4

void **Generator::visit**(const FConditionNode \* node)

{

IfPtr thenBr = 0, elseBr = 0, thenBrFork = 0, elseBrFork = 0, cond;

if (disableAnt || !node->condition()->isLong())

{

thenBr = createSpan(node->trueBranch(), mTail);

elseBr = createSpan(node->falseBranch(), mTail);

}

else

{

if (node->trueBranch()->isLong())

thenBrFork = createSpan(node->trueBranch(), std::make\_shared<EndOp>());

else

thenBr = createSpan(node->trueBranch(), mTail);

if (node->falseBranch()->isLong())

elseBrFork = createSpan(node->falseBranch(), std::make\_shared<EndOp>());

else

elseBr = createSpan(node->falseBranch(), mTail);

}

auto choose = std::make\_shared<CondChoose>(thenBr, elseBr, mTail);

cond = createSpan(node->condition(), choose);

mResult = std::make\_shared<CondStart>(cond, thenBrFork, elseBrFork);

}

void **CondStart::exec**(SExecutionContext & ctx) const

{

ctx.controlStack.push\_back(ctx.arity);

if (mThen)

{

IFExecutionContext \*fork = static\_cast<IFExecutionContext &>(ctx).spawn(mThen.get());

fork->NewAnticipationLevel = 1;

fork->Anticipation = 1;

ctx.evaluator()->addForkJob(fork);

}

if (mElse)

{

IFExecutionContext \*fork = static\_cast<IFExecutionContext &>(ctx).spawn(mElse.get());

fork->NewAnticipationLevel = 1;

fork->Anticipation = 1;

ctx.evaluator()->addForkJob(fork);

}

mCond->exec(ctx);

}

void **CondChoose::exec**(SExecutionContext & ctx) const

{

auto arity = ctx.controlStack.back().OutArity;

ctx.controlStack.pop\_back();

// Берём сверху стека 1 аргумент - результат вычисления предиката.

DataValue cond = ctx.stack.back();

bool isUndefined = false;

int numArgs = ctx.arity - arity;

for (int i = 0; i < numArgs; ++i)

{

DataValue & arg = ctx.stack.back();

// Проверяем, содержится ли в кортеже неопределенное значение w для реализации семантики w\*a = a\*w = w.

if (arg.getOps() == undefined.getOps())

{

isUndefined = true;

}

ctx.stack.pop\_back();

}

ctx.arity = arity;

// Проверка условия.

if (numArgs > 0 && (isUndefined || (cond.getOps() == falseConst.getOps() && !cond.mIntVal)))

{

if (!mThen) // Если ненужная ветвь длинная - отменяем её вычисление.

ctx.evaluator()->cancelFromPendingEnd(1 + !mElse);

if (mElse) // Если верная ветвь короткая - начинаем её вычислять.

mElse->exec(ctx);

else

{

ctx.join();

mNext->exec(ctx);

}

}

else

{

if (!mElse) // Если ненужная ветвь длинная - отменяем её вычисление.

ctx.evaluator()->cancelFromPendingEnd();

if (mThen) // Если верная ветвь короткая - начинаем её вычислять.

mThen->exec(ctx);

else

{

ctx.join();

mNext->exec(ctx);

}

}

}

void **EvaluatorUnit::addForkJob**(SExecutionContext \* task)

{

pendingTasks.push\_back(task);

if (task->Anticipation.load(std::memory\_order\_acquire))

{

mAnticipationJobQueue.push(task);

mAnticipationJobsCreated++;

}

else addJob(task);

}

SExecutionContext \* **EvaluatorUnit::stealAnticipationJob**()

{

SExecutionContext \* elem = nullptr;

mAnticipationJobQueue.pop(elem);

return elem;

}

SExecutionContext \* **SchemeEvaluator::findAnticipationJob**(const EvaluatorUnit \* aUnit)

{

for (size\_t i = 0; i < mEvaluatorUnits.size(); i++)

{

if (mEvaluatorUnits[i] != aUnit)

{

SExecutionContext \* job = mEvaluatorUnits[i]->stealAnticipationJob();

if (job)

{

return job;

}

}

}

return nullptr;

}

void **EvaluatorUnit::moveToMainOrder**(SExecutionContext \* movingTask)

{

if (movingTask->Parent->Anticipation.load(std::memory\_order\_acquire))

{

movingTask->Anticipation.store(0, std::memory\_order\_release);

if (!movingTask->Ready && !movingTask->Working)

{

mJobQueue.push(movingTask);

mAnticipationJobsMoved++;

}

for (SExecutionContext \* child : movingTask->Childs)

{

if (!child->NewAnticipationLevel)

moveToMainOrder(child);

}

}

movingTask->NewAnticipationLevel.store(0, std::memory\_order\_release);

}

void **EvaluatorUnit::schedule**()

{

SExecutionContext \* context = nullptr;

if (mJobQueue.pop(context))

{

context->run(this);

mJobsCompleted++;

// Выполнили задание - проверяем не запланированна ли сборка мусора.

safePoint();

return;

}

context = mEvaluator->findJob(this);

if (context)

{

context->run(this);

mJobsStealed++;

mJobsCompleted++;

// Выполнили задание - проверяем не запланированна ли сборка мусора.

safePoint();

return;

}

if (!disableAnticipatory)

{

if (mAnticipationJobQueue.pop(context) && !context->Ready.load(std::memory\_order\_acquire) && context->Anticipation.load(std::memory\_order\_acquire))

{

context->Working.store(1, std::memory\_order\_release);

context->run(this);

mAnticipationJobsCompleted++;

// Выполнили задание - проверяем не запланированна ли сборка мусора.

safePoint();

return;

}

context = mEvaluator->findAnticipationJob(this);

if (context && !context->Ready.load(std::memory\_order\_acquire) && context->Anticipation.load(std::memory\_order\_acquire))

{

context->Working.store(1, std::memory\_order\_release);

context->run(this);

mAnticipationJobsStealed++;

mAnticipationJobsCompleted++;

// Выполнили задание - проверяем не запланированна ли сборка мусора.

safePoint();

return;

}

}

boost::this\_thread::sleep\_for(boost::chrono::milliseconds(1));

safePoint();

}

void **EvaluatorUnit::cancelFromPendingEnd**(const int backPos)

{

static std::mutex outputMutex;

SExecutionContext \* cancelTask = pendingTasks.at(pendingTasks.size() - backPos);

cancelTask->Parent->Childs.erase(cancelTask);

outputMutex.lock();

cancel(cancelTask);

outputMutex.unlock();

//Убираем из очереди ожидающих выполнения задач.

pendingTasks.erase(pendingTasks.end() - backPos);

}

void **EvaluatorUnit::cancel**(SExecutionContext \* cancelingTask)

{

cancelingTask->Parent->Childs.erase(cancelingTask);

cancelingTask->Canceled = 1;

if (!cancelingTask->Ready)

{ // Если задание ещё не выполнено, выставляем флаг готовности, чтобы никто не начал выполнение.

cancelingTask->Ready = 1;

// Если уже выполняется - запускаем процесс остановки.

if (cancelingTask->Working.load(std::memory\_order\_acquire))

{

cancelingTask->cancel();

}

mAnticipationJobsCanceled++;

}

for (SExecutionContext \* child : cancelingTask->Childs)

{

cancel(child);

}

}

IFExecutionContext \* **IFExecutionContext::spawn**(InternalForm \* forkBody)

{

IFExecutionContext \* fork = new IFExecutionContext(forkBody);

fork->Parent = this;

fork->Anticipation = this->Anticipation.load(std::memory\_order\_acquire);

this->Childs.insert(fork);

// Копируем стек.

for (int i = argPos; i < (argPos + argNum); i++)

{

fork->stack.push\_back(stack.at(i));

}

fork->argNum = argNum;

return fork;

}

SExecutionContext \***EvaluatorUnit::join**()

{

// Перед возможным выполнением новой задачи проверяем, не запланированна ли сборка мусора.

// Иначе может получаться ситуация, когда 1 поток пораждает задания и сразу же их берёт на выполнение,

// а все остальные ожидают сборки мусора.

safePoint();

SExecutionContext \* joinTask = pendingTasks.back();

if (joinTask->Anticipation.load(std::memory\_order\_acquire) && !joinTask->Parent->Anticipation.load(std::memory\_order\_acquire))

moveToMainOrder(joinTask);

while (!joinTask->Ready.load(std::memory\_order\_acquire))

{

schedule();

}

joinTask->Parent->Childs.erase(joinTask);

pendingTasks.pop\_back();

return joinTask;

}

void **SExecutionContext::join**()

{

auto joined = mEvaluatorUnit->join();

if (!joined->Canceled)

// Копируем результат.

for (int i = 0; i < joined->arity; ++i)

{

push(joined->stack.at(joined->stack.size() - joined->arity + i));

}

delete joined;

}