Исследование возможностей параллельной трансляции функциональных программ

Дацун Н.Н., Савков К.Г.
Донецкий национальный технический университет
datsun@pmi.donntu.edu.ua
cos_savkov@mail.ru

Abstract

Datsun N., Savkov K. "Research of opportunities of compiling parallel functional programs". *The article examines methods of paralleling of Lisp program translation process. Characteristics of possible acceleration of phase translation are received at parallel realization of the compiler functional programming language.*

Введение

Современные тенденции развития процессоров (многоядерность) привели к тому, что скоростной потенциал процессоров стал существенно зависеть от качества программного обеспечения. Однако критерии оценки результатов труда разработчиков программного и аппаратного обеспечения значительно отличаются. Задача оптимизации программного обеспечения часто возникает только после его коммерческого успеха. И никогда раньше успех массового программного обеспечения не зависел от его "распараллеливаемости". В связи с этим возникает необходимость рефакторинга [1] уже существующего программного обеспечения для более эффективного параллельного выполнения. Сам рефакторинг программного обеспечения требует экономических затрат и многих человеко/часов работы. Недостатки рефакторинга на данный момент перевешивают его преимущества, поэтому рефакторингом старого программного обеспечения практически не занимаются. Экономическая целесообразность рефакторинга зависит от качества и доступности средств автоматического распараллеливания последовательных программ. К таким средствам и относится рассмотренный в статье транслятор функционального языка Lisp с полной прозрачностью параллелизма. Он разработан для автоматизации процесса параллельного выполнения последовательных программ и может быть использован как инструмент для рефакторинга функциональных программ.

Целью работы является исследование возможности увеличения производительности транслятора функционального языка Lisp благодаря использованию параллельных вычислений. Основными задачами исследования являлись:

- определение основных путей увеличения производительности транслятора языка Lisp;
- создание транслятора языка Lisp с полной прозрачностью параллелизма.
- исследование результатов фаз трансляции последовательных программ в параллельный эквивалент.

Объектом исследования является транслятор функционального языка Lisp. Предметом исследования является часть транслятора – фаза лексического анализа, на которой возможно применение параллельных вычислений для увеличения производительности.

1 Анализ трансляторов языка Lisp, реализующих параллелизм вычислений, и методов управления задачами

Использование параллельных вычислений в функциональных языках не ново. Многие существующие трансляторы Lisp позволяют порождать новые процессы для выполнения подзадач. Первыми решениями для параллельного выполнения Lisp-программ были дополнительно введенные функции операционной системы по работе с процессами и другими системными объектами. Стандарт Common Lisp определил механизм вызова системных функций [2], однако в разных операционных системах эти функции были разными. Другим решением для распараллеливания программ на языке Lisp было введение дополнительных языковых конструкций для определения параллельного выполнения [3]. Эти расширения языка назывались условно прозрачными, так как без них результаты последовательного выполнения программы должны были совпадать с результатами параллельного ее выполнения. Одна из таких конструкций – "future" была занесена в стандарт языка для порождения отложенных вычислений. Информация о параллельных реализациях функциональных языков приведены в табл. 1.

Таблица 1. Параллельные реализации языков функционального программирования

Название (разработчик)	Платформа	Реализация параллелизма
Kyoto Common Lisp (университет Киото)	Unix 4.2 BSD на Sun Workstation	Функции ОС
Multilisp (университет Монреаля)	Unix – совместимая	Future (MPI)
Qlisp (Lucid Inc.)	Unix (Alliant FX/80)	Конструкции языка (процессы)
TS/Scheme (NEC)	STING для Silicon Graphics MIPS R4000	Конструкции языка (распределенная система)
BandaLisp (университет Сингапура)	SunOS, Sun Sparc Server 1000	Конструкции языка (компилятор)

Kyoto Common Lisp [4] предусматривает использование функции ОС для параллельного выполнения программы. Данная реализация Lisp является наиболее полной и мощной. Все последующие реализации Common Lisp основывались на частях этой Lisp-системы, а часть функций перешла в стандарт Common Lisp.

Multilisp [5] расширяет язык функционального программирования Scheme простой и элегантной парадигмой программирования. Параллелизм указывается в программе непосредственно благодаря использованию специальной формы “future” [3]. Данная форма изначально применяется только в Scheme, но она вполне подходит для использования в языке Lisp, хотя и не входит в стандартные функции языка:

(future expr)

Эта конструкция определяет отложенное вычисление expr, которое является независимой задачей, и должно быть вычислено до первого использования значения expr. Параллелизм вычисления следует из модели задач. Когда задача выполняет future, она порождает подзадачу (задачу потомка) для вычисления тела future. Основная же задача начинает выполнять продолжение future. Таким образом, параллелизм существует между вычислением тела и продолжением основной задачи.

Значение, возвращаемое в продолжение, есть вычисляемое (не обязательно вычисленное) значение тела. Чтобы избежать чтения несуществующего значения, используется специальный замещающий объект placeholder (для представления значения тела). Замещающий объект изначально пуст, он получает определенное значение только тогда, когда подзадача его вернет. Когда замещающий объект передается строгой функции, например, селектору car или функции if в качестве условия, он должен быть определен, чтобы получить значение. Если же он не определен, то задача откладывается до определения его подзадачей.

Multilisp использует модель общей памяти. Это означает, что задача может обращаться к любым данным, вне зависимости от того, где они были созданы.

В Qlisp [6] для распараллеливания вычислений введены специальные конструкции языка, при этом виртуальная машина динамически подстраивается под число занятых процессоров и предоставляет языковые средства синхронизации.

TS/SCHEME [7] – это распределенная система для объектно-ориентированного языка Lisp, для распараллеливания вычислений используются специальные конструкции языка.

В BandaLisp [8] для распараллеливания используются специально введенные конструкции для перевода списка в массивы и компиляция программы. Несмотря на то, что данная система рассчитана на последовательный диалект языка Lisp Scheme, в ней применяется оптимизирующая компиляция частей программы в машинный код. Результатами оптимизатора стали очень высокие показатели быстродействия программы, часто достигающие результатов при выполнении на параллельном Си. Целью проекта BandaLisp было опровержение того, что Lisp-программы значительно проигрывают их эквивалентам, написанным на Си. Несмотря на успешный результат этого проекта, данное негативное мнение о языке Lisp все еще распространено. С другой стороны, компиляция Lisp-программ в машинный язык возможна только при окончательно сформированном тексте программы, которое не использует некоторые ключевые особенности языка, например, динамическую генерацию текста программы.

Начиная с 1995 г., интерес к языку Lisp упал. Этим объясняется отсутствие публикаций за данный период. В период 2000-2007 гг. новых подходов к распараллеливанию вычислений в функциональных языках не сформировалось.

За указанный период появилось несколько реализаций Lisp с открытым кодом на различные платформы, при этом используемые методы остались те же. Основными преимуществами данных реализаций является их высокая производительность, мощные библиотеки встроенных функций и поддержка объектно-ориентированных технологий. Все реализации требуют использования дополнительных конструкций языка, непосредственно

указываемых при написании программы - необходимость явного указания параллелизма. Данная особенность делает невозможным использовать эти реализации при распараллеливании уже имеющихся последовательных программ без дополнительной переработки исходных текстов со стороны программиста, исключая таким образом рефакторинг функциональных программ.

В качестве системы-аналога в данной работе использован Qlisp [7], так как он содержит наиболее мощные конструкции для указания параллельного выполнения программы (среди рассмотренных выше трансляторов).

Подход, использованный в Qlisp – это параллельное выполнение очереди процессов. Программист должен явно указывать в программе, где возможно параллельное выполнение благодаря использованию специальных параллельных конструкций. Когда исполняемая программа выполняет инструкцию, указывающую на параллелизм, она добавляет набор новых задач в очередь для дальнейшего вычисления. Когда процессор завершает выполнение задачи, он переходит на следующую задачу в очереди. Основа параллелизма на очередях времени выполнения означает, что программа не написана или скомпилирована под конкретное число процессоров. Количество доступных процессоров может даже изменяться в ходе вычислений. Выполняемые задачи не обязаны быть схожей длины, так как после завершения короткой задачи процессор просто берет следующую из очереди. Для непосредственного ожидания возврата значения конструкции future применяется (realize-future form). После вызова конструкция вычисляет form, затем, если ее значение есть future, подождет до его реализации, потом вернет значение.

Самый простой способ включения параллелизма в Lisp-программу – использовать конструкцию (spawn prop form) .

Она создает новый процесс для вычисления form. Форма prop – специальный параметр, который вычисляется первым. Если его значение есть nil (ложь), то новый процесс не создается, а процесс, вычисляемый spawn, начнет выполнять form и вернет результат. Если prop не есть nil (истина), то новый процесс будет создан для вычисления form, а spawn вернет future, который затем будет реализован конкретным значением, после того, как новый процесс завершит его вычисление.

Все конструкции Qlisp для создания новых процессов используют похожие параметры для того, чтобы дать программисту способ ограничивать уровень параллелизма во время выполнения программы.

В MultiLisp [9] конструкция “future” реализована через MPI. В этой реализации рассматривались два основных метода управления задачами, называемые «Нетерпеливой задачей» и «Ленивой задачей». Благодаря реализации подхода «Ленивой задачи» стало возможным существенное увеличение производительности на MPI, по сравнению с мультипроцессорами с разделяемой памятью. Основным недостатком MPI являлась проблема синхронизации и передачи задач другим процессорам, так как были велики расходы процессорного времени. Этим и объясняется обилие параллельных реализаций Lisp для мультипроцессорных систем с общей памятью.

Создание нетерпеливой задачи (Eager task creation, ETC) есть прямая реализация future, используемая в некоторых параллельных Lisp-системах. Вычисление future немедленно создает в куче объект-задачу для представления порожденной задачи и делает ее доступной всем процессорам, добавляя ее в глобальную очередь задач (рабочую очередь). Объект-задача состоит из продолжения выполнения, показывающего состояние задачи. Это продолжение установлено на вычисление тела future, а затем - на определение замещающего объекта для результата. Когда процессор освобождается, он удаляет задачу из рабочей очереди, затем возобновляет выполнение ее продолжения. Для уменьшения загрузки рабочая очередь распределена между процессорами, и каждый процессор порождает и выполняет задачи из своей рабочей очереди. Когда процессор свободен, и его рабочая очередь пуста, он должен получить задачу из рабочей очереди другого процессора. Передачу задачи от “жертвы (victim)” к “вору (thief)” называют “кражей (steal)”.

Основным недостатком метода ETC является высокая загрузка управления задачами. Считая, что каждая задача случайным образом выполняется и завершается, для каждой задачи необходимо выполнение нескольких комплексных операций. При порождении задачи требуется создать объект-задачу и объект-заместитель, а также среду для запуска тела future (фрейм). После чего рабочую очередь нужно заблокировать, добавить задачу в рабочую очередь, разблокировать рабочую очередь. Создание объекта-заместителя также требует блокировки и разблокировки.

Общая стоимость перечисленных операций превышает сотни машинных команд. Производительность существующих реализаций метода ETC подтверждает это. Система Mul-T [10] была разработана для минимизации стоимости ETC. В этой системе это составляет около 130 машинных команд на задачу на Encore Multimax. В других системах стоимость еще выше (в Qlisp [6] около 1400 команд).

Из-за высокой стоимости распараллеливания результат существенно зависит от размера задач, так как для маленьких задач большая часть времени будет потрачена на бесполезное порождение задачи.

Создание ленивой задачи [11] (Lazy Task Creation, LTC) есть альтернатива реализации future. LTC уменьшает стоимость вычисления future благодаря более эффективной передаче отложенных задач. Новая задача порождается только в случае, если процессор простаивает. LTC достигает этого, адаптируя стековое расписание задач: в отсутствии свободных процессоров производит такой же порядок действий, что и последовательная программа (т.е. без future). Вычисление тела future немедленно начинается, а продолжение выполнения программы откладывается. Каждый процессор содержит свою структуру данных отложенных вычислений, подобную стеку - очередь ленивых задач (lazy task queue, LTQ). Когда вычисление тела завершается, последняя отложенная задача удаляется из очереди и начинает выполняться. В этом методе нет необходимости создавать объект-заместитель, так как продолжение может непосредственно получить значение, возвращенное телом.

В LTC кража задачи происходит следующим образом: убирается самое старое продолжение из LTQ-жертвы, создаются соответствующий объект-заместитель и новая задача, задача передается вору. Для связи между украденной задачей с ее потомком, вор должен вычислить продолжение задачи с созданным заместителем. Заместитель также должен храниться в порожденной задаче, чтобы правильно определить место для результата.

Кража самой старой задачи вместо самой новой, может сократить количество краж, потому что старые задачи обычно содержат в себе больше работы (вор будет дольше занят обработкой задач до следующей кражи). Это верно для программ, написанных с использованием параллельных алгоритмов по методу «разделяй и властвуй». Если алгоритм хорошо сбалансирован, объем работ украденной задачи будет соизмерим с суммой оставшихся в LTQ работ. Доступ к LTQ более эффективный, так как два конца LTQ могут работать независимо: жертва может добавлять продолжение в LTQ, в то время, как вор совершает кражу с другого конца.

2 Степени прозрачности параллелизма в существующих трансляторах языка Lisp

Существующие трансляторы языка Lisp являются сложными системами, которые можно разделить на две основные подсистемы – механизм языка Lisp и механизм виртуальной машины. Механизм языка должен корректно обрабатывать простые конструкции языка и взаимодействовать с виртуальной машиной для непосредственного выполнения программы. Взаимодействие с виртуальной машиной производится с помощью интерфейса. Виртуальная машина фактически привязана к аппаратуре, она поддерживает параллельное выполнение. С точки зрения программиста на языке Lisp интерфейс взаимодействия обладает понятием прозрачности [12]. Эта прозрачность может быть полной, промежуточной или вообще отсутствовать. Отсутствие прозрачности означает, что программист должен явно использовать специальные внутренние функции виртуальной машины для параллельного выполнения Lisp-программы (например, использование функций ОС в Kyoto Common Lisp [3]). Промежуточная степень прозрачности означает, что программист не обязан в точности знать специальные функции виртуальной машины, но должен декларировать в Lisp-программе параллельное выполнение с помощью специальных конструкций. Данные конструкции при промежуточной прозрачности интерфейса носят рекомендательный характер для виртуальной машины, так как на основании них виртуальная машина решает: выполнять программу параллельно или нет. Этот подход скрывает от программиста на языке Lisp особенности реализации виртуальной машины, однако не освобождает от необходимости применения специальных конструкций. Полная прозрачность интерфейса полностью скрывает от программиста факт параллельного выполнения программы на виртуальной машине, при этом не требует никаких дополнительных конструкций языка Lisp. Под прозрачностью параллелизма в этой статье подразумевается прозрачность интерфейса взаимодействия механизма языка Lisp с параллельной виртуальной машиной.

Современные параллельные реализации языка Lisp обладают хорошей производительностью и уделяют много внимания механизму распределения задач между процессорами для улучшения практических результатов. Однако существующие алгоритмы переносят ответственность за распараллеливание выполнения на плечи программиста, но это не всегда является оптимальным решением.

Обзор реализаций языка Lisp также выявил консервативное отношение к реализации трансляторов: новые трансляторы были созданы на основе уже существующих путем добавления и модификации реализации параллельного выполнения. Распространенным решением является добавление к уже существующему транслятору библиотеки для параллельного выполнения. Также большинство реализаций рассчитано на специальные процессоры и архитектуру компьютера, а не на архитектуру PC.

В статье рассматривается транслятор языка Lisp с полной прозрачностью параллелизма,

ориентированный на теоретические исследования. Задачи этих исследований - анализировать функциональные программы и определять пути увеличения их производительности.

3 Постановка задачи

Основываясь на полученных данных об уже существующих трансляторах, выделим основные требования к разрабатываемому транслятору:

– полная прозрачность параллелизма;
– ориентация сразу на несколько платформ;
– удобный механизм отладки параллельных приложений;
– возможность анализа программ для определения целесообразности распараллеливания.

Полная прозрачность параллелизма необходима для проведения автоматического рефакторинга уже существующих последовательных функциональных программ, так как в противном случае будет требоваться вмешательство программиста в тексты программ.

Кроссплатформенность является необходимым условием для любой виртуальной машины, так как эффективность ее работы существенно зависит от аппаратуры, на которой она выполняется. Нельзя ограничиваться традиционной архитектурой фон Неймана, так как для многих задач она не является эффективной

Современным стандартом параллельных вычислений является MPI. Практически все мультипроцессорные системы и кластеры его поддерживают. Реализации этой библиотеки существуют на многих платформах, поэтому она использована в трансляторе.

Отладочные средства систем программирования функциональных языков не достигли уровня современных отладчиков. С другой стороны, отладочная информация несет в себе накладные расходы, что не может не отразится на производительности транслятора в целом. Однако в современном программировании часто жертвуют производительностью во имя стабильности работы программ. Библиотека MPI содержит свои средства отладки, поэтому транслятор языка Lisp должен быть ориентирован на работу с ними.

4 Используемые методы и алгоритмы

Для определения параллельности вычисления S-выражений используется модификация метода непересекающегося множества переменных. Параллельное выполнение основано на парадигме портфеля задач.

Перед распараллеливанием программы проводится ее анализ. Анализ текста программы покажет пути увеличения производительности при ее выполнении, например:

– при вычислении выражения (* 2 (+ 1 1)) производится 5 вызовов функционала eval. Но таблица 2 показывает, что полезная работа выполняется только на шагах 1 и 3. Вызовы eval на шагах 2, 4, 5 являются бесполезными, так как они ничего не вычисляют;

Таблица 2. Порядок вычисления S-выражения

Порядок вызова/номер шага	Аргумент eval
1	(* 2 (+ 1 1))
2	2
3	(+ 1 1)
4	1
5	1

– при вычислении выражения (cons a b) нам не известны значения атомов a и b, в частном случае, если a и b не связаны со значением, то вызовы eval можно опустить;
– при выполнении выражения (+ (* 2 3) (- 5 2)) возможно параллельное выполнение (* 2 3) и (- 5 2). Это распараллеливание производится на этапе вычисления списка для функций типа EXPR или SUBR [13]. Так как это является составной частью механизма работы функционала APPLY, то именно в этой функции и должно производиться распараллеливание.

4.1 Модификация метода непересекающегося множества переменных

Множество записи процесса – это множество переменных, которым он присваивает значения (и, возможно, считывает их), а множество чтения процесса – это множество переменных, которые процесс считывает, но не изменяет. Множество ссылок процесса – это множество переменных, которые встречаются в утверждениях доказательства корректности данного процесса. Если множество записи одного процесса не пересекается со множеством ссылок другого процесса, и наоборот, то эти два процесса не могут влиять друг на друга [14].

Определим условия независимости двух процессов в языке Lisp. Пусть SExpressions – множество S-выражений для выполнения, Atoms – множество атомов. Пронумеруем все атомы и S-выражения, так чтобы каждый из них имел уникальный номер, например a_i - атом с номером i. Путем нумерации мы определим множество целочисленных идентификаторов атомов и S-выражений – AtomIDs и SExpressionIDs соответственно. Свойство атома есть тоже атом, таким образом, определим предикат $P(a_i, a_j)$, указывающий, имеется ли у атома a_i свойство a_j. Значением свойства атома может быть любое правильное S-выражение. Определим функцию

$V(a_i, a_j)$, которая возвращает свойство a_j атома a_i. S-выражение может содержать в себе атомы. В общем случае, до вычисления всего S-выражения невозможно точно определить, какие символьные атомы будут вычислены. Под вычислением подразумевается вызов функции eval для символьного атома, который требует получения связанного с атомом значения. Все атомы в Lisp могут быть вычислены (evaluated) для получения связанного с ними значения. Так в языке Mulisp символьные атомы связаны сами с собой, поэтому вычисление атома вернет его самого. В Common Lisp же изначально атомы не связаны, поэтому вычисление атома будет ошибкой [3]. Теоретически все символьные атомы, встречающиеся в S-выражении, могут быть вычислены. Кроме этого, у атомов могут быть взяты свойства и вычислены атомы в них.

Определим, какие атомы могут быть вычислены в S-выражении во множество атомов AtomsOfS:

$$\forall i \in AtomIDs \equiv a_i \in Atoms$$

$$\forall i \in SExpressionIDs \equiv s_i \in SExpressions$$

$$\forall a_i \in s_j, i \in AtomIDs, j \in SExpressionIDs \rightarrow a_i \in AtomsOfS_j$$

$$\forall a_i \in AtomsOfS_l, a_j \in AtomsOfS_l, P(a_i, a_j) \rightarrow$$

$$\forall a_k \in V(a_i, a_j), a_k \in AtomsOfS_l$$

$$\forall a_i \in AtomsOfS_j \equiv a_i \in Atoms, s_j \in SExpressions$$

Теперь AtomsOfS содержит все атомы, которые теоретически можно использовать.

Пусть необходимо выполнить два S-выражения параллельно – S_1 и S_2. По технике устранения взаимного вмешательства – множество ссылок одного процесса p_i не должно пересекаться с множеством записи другого процесса p_j для параллельного выполнения. Дадим предикату параллельного выполнения имя Parallel. Пусть R – множество чтения процесса, W – множество записи этого процесса, L – множество ссылок.

$$\forall p_i, L_i = R_i \bigcup W_i$$

$$Parallel(p_i, p_j) \equiv \begin{cases} L_i \bigcap W_j = \emptyset \\ L_j \bigcap W_i = \emptyset \end{cases}$$

$$L_i \bigcap L_j = (R_i \bigcup W_i) \bigcap L_j = (L_j \bigcap W_i) \bigcup (L_j \bigcap R_i)$$

$$L_i \bigcap L_j = L_i \bigcap (R_j \bigcup W_j) = (L_i \bigcap W_j) \bigcup (L_i \bigcap R_j)$$

$$(L_i \bigcap L_j = \emptyset) \equiv \begin{cases} L_j \bigcap W_i = \emptyset \\ L_j \bigcap R_i = \emptyset \end{cases}$$

$$(L_i \bigcap L_j = \emptyset) \equiv \begin{cases} L_i \bigcap W_j = \emptyset \\ L_i \bigcap R_j = \emptyset \end{cases}$$

$$(L_i \bigcap L_j = \emptyset) \rightarrow (\mathrm{L_j} \bigcap \mathrm{W_i} = \emptyset)$$

$$(L_i \bigcap L_j = \emptyset) \rightarrow (\mathrm{L_i} \bigcap \mathrm{W_j} = \emptyset)$$

$$(L_i \bigcap L_j = \emptyset) \rightarrow Parallel(p_i, p_j)$$

Для языка Lisp можно сказать, что AtomsOfS – это множество ссылок. Фактически, это множество содержит не только используемые ссылки L, но и неиспользуемые N. Каждый процесс в Lisp можно задать вычисляемым S-выражением.

$$\forall i \in SExpressions, p_i = s_i$$

$$\forall i \in SExpressions, AtomsOfS_i = L_i \bigcup N_i$$

Докажем, что если множества всех ссылок S-выражений не пересекается – то они могут выполняться параллельно.

$$(AtomsOfS_i \bigcap AtomsOfS_j = \emptyset) \rightarrow (\mathrm{L_i} \bigcup N_i) \bigcap (L_j \bigcup N_j) = \emptyset$$

$$((\mathrm{L_i} \bigcup N_i) \bigcap (L_j \bigcup N_j)) = \emptyset \rightarrow \mathrm{L_i} \bigcap (L_j \bigcup N_j) = \emptyset$$

$$(\mathrm{L_i} \bigcap (L_j \bigcup N_j) = \emptyset) \rightarrow (\mathrm{L_i} \bigcap L_j = \emptyset)$$

$$(L_i \bigcap L_j = \emptyset) \rightarrow Parallel(p_i, p_j)$$

$$(AtomsOfS_i \bigcap AtomsOfS_j = \emptyset) \rightarrow Parallel(p_i, p_j)$$

Данный подход требует более жестких условий для определения возможности параллельного выполнения. Множество записи другого процесса расширяется до множества ссылок, а множество ссылок расширяется неиспользуемыми ссылками. В этом остаются возможности увеличения производительности, такие как уменьшение множества неиспользуемых ссылок.

4.2 Представление задачи в портфеле задач

В существующих параллельных реализациях языка Lisp используется парадигма выполнения программ “портфель задач” [14]. Задача является независимой единицей работы. Задачи помещаются в портфель, разделяемый несколькими рабочими процессами. Каждый рабочий процесс выполняет основной цикл:

```
While (true)
 {получить задачу из портфеля;
  if (задач больше нет)
    break;
  выполнить задачу, возможно,
  порождая новые задачи;
  };
```

Рисунок 1 – Алгоритм работы портфеля задач

Этот подход можно использовать для реализации рекурсивного параллелизма; тогда задачи будут представлены рекурсивными вызовами. Парадигма портфеля задач имеет такие достоинства: простота (достаточно определить представление задачи и определить условие завершения работы алгоритма); масштабируемость программ (их можно использовать с любым числом процессоров, изменяя количество рабочих процессов); упрощение реализации балансировки нагрузки

Использование парадигмы портфеля задач в рассматриваемом трансляторе отличается от существующих трансляторов представлением задачи. Традиционно для представления задач используются внутренние структуры данных - очередь задач. В описываемом трансляторе сама задача и результат ее выполнения передаются в одном и том же виде – в виде текста программы. Использование текста программы вместо внутреннего представления замедляет вычисление результата в связи с тем, что будут заново выполняться фазы лексического и синтаксического анализа, а также загрузка программы во внутреннее представление списка. Однако при передаче текста задачи упрощается процесс отладки программ, так как программист может в явном виде увидеть непосредственно исполняемую программу, а также ее результат. Затраты процессорного времени на лексический и синтаксический анализ пропорциональны времени, необходимому для передачи структур, и также являются линейно зависимыми. Таким образом, все внутренние представления задачи могут быть описаны средствами языка. Логичным является передача тела задачи в явном виде.

Рассмотрим примеры задач, порождаемых функциональной программой. Простейшая задача – это S-выражение из встроенных функций и числовых атомов:

```
Вход : (+ 3 2)
Выход: 5
```

Более сложным вариантом передачи задачи является случай использования свойств атома. Свойства атома связывают его со значением или функцией. Явным способом связь атома со значением описывает функция SET. Если атом уже связан со значением, эквивалентной по результату будет SETQQ. Кроме значения, у атома могут быть и другие свойства, заданные псевдофункцией PUT. В общем случае можно передавать все глобальные связи атома с помощью PUT, но для человека удобнее было бы воспринимать SETQ или DEFUN [13]. Ниже представлена задача из S-выражения, содержащего встроенные функции и символьные атомы. При передаче данной задачи другому процессору необходима также передача окружения:

```
Вход: (CADR '(A B C))
Окружение входа: (DEFUN CADR (X) (CAR (CDR X)))
Выход: 'B
Окружение выхода: (DEFUN CADR(X) (CAR (CDR X)))
```

В общем случае окружение состоит из свойств атомов. При выполнении задачи окружение также может измениться, таким образом, оно должно возвращаться вместе с результатом.

Практически удобно использовать для передачи окружения функцию PROGN. Для предыдущего примера аналогом будет являться следующая конструкция:

```
Вход: (PROGN (DEFUN CADR(X)(CAR (CDR X))) (CADR '(A B C)))
Выход: (PROGN (DEFUN CADR(X)(CAR (CDR X))) 'B)
Упрощенный выход: 'B
```

Передача определения функции CADR в результат не требуется, так как оно не меняется.

Выше были описаны приемы, корректные для простых вызовов верхнего уровня. Логично использовать распараллеливание на задачи и внутри вызовов функций. Вызовы внутри функций отличаются от верхнего уровня наличием локальных связей для аргументов функции. Естественным средством языка для создания локальных связей является LAMBDA как определение функции. Таким образом, передаваемую задачу необходимо заключить в тело вызова функции. LAMBDA-вызов создаст локальные связи [15, 16]. Так как тело функции не обязано содержать одно S-выражение, а выполняется как PROGN, то дополнительно писать PROGN внутри тела не требуется:

```
Вход:((LAMBDA(X Y)(+ X Y)) '3 '2)
Выход: '5
```

Очевидно, что локальные связи уничтожатся при выходе из задачи, они не передаются в результат. Отмена вычислений при передаче результата требуется для корректной обработки при получении результата. Так как результат может храниться и в записанных свойствах атомов, то эти свойства будут записаны в результат в виде соответствующих вызовов SETQ, DEFUN или PUT. Полученное таким образом описание задачи полностью соответствует Lisp-программе. Получившееся в таком случае вычисление задачи есть не что иное, как преобразование текста программы.

Логично предположить, что расходы времени на передачу задачи другому процессору и получение результата могут превышать время вычислений. Таким образом, необходимы дополнительные данные для определения целесообразности передачи задачи.

4.3 Определение целесообразности параллельного выполнения

При принятии решения о том, посылать ли задачу для параллельного выполнения другому процессору или же не посылать, следует учесть время на передачу задачи и получение результата, а также время выполнения.

Пусть $T_з$ - время, необходимое на передачу задачи, $T_в$- время выполнения задачи процессором после получения, $T_р$ - время, необходимое на получение результата.

Из соображений здравого смысла нет необходимости передавать задачу другому процессору, если передавать ее и получать результат дольше, чем просто ее выполнить. Отсюда получим условие, при котором не следует передавать задачу другому процессору.

$$T_з + T_р \geq T_в$$

Время передачи данных между процессорами зависит от типа связи между ними. Например, многопроцессорная система с общей памятью выполнит передачу мгновенно, а передача задачи через сеть другому компьютеру займет значительно больше времени. Таким образом, используя этот критерий, мы можем определить, какому из процессоров передать задачу.

Передаваемая задача в рассматриваемом трансляторе языка Lisp представляет собой независимую программу в виде текста. Накладными расходами являются перевод текста во внутреннее представление, но он линейно зависит от размера передаваемого текста.

В действительности лексический анализ Lisp-программы и ее загрузка являются задачей линейной сложности. Сканер (лексический анализатор) [17] читает все символы программы последовательно. Таким образом, время выполнения лексического анализа пропорционально длине текста программы. Полученные в результате лексического анализа символьные атомы будут заноситься в символьную таблицу. Время занесения и поиска в таблице примем постоянным, так как при реализации таблицы символов в трансляторе использована хеш-таблица. Загрузчик совмещен с синтаксическим анализатором и работает методом рекурсивного спуска, используя грамматику с постоянным числом правил. Таким образом, для каждой лексемы в худшем случае будут перебраны все правила. Так как число порождающих правил постоянно, то время на загрузку каждой лексемы ограничено сверху некоторой постоянной. Таким образом, время выполнения загрузки пропорционально длине текста программы в лексемах, вместе с лексическим анализом получим задачу линейной сложности.

Накладные расходы на передачу можно связать с размером задачи, то есть с ее размером в байтах или в лексемах. Обозначим $T_п$- время, необходимое для передачи одной лексемы от одного процессора другому, $T_з$ - время, необходимое для загрузки лексемы (для перевода лексемы во внутреннее представление виртуальной Lisp-машины), а $T_в$ - время вывода лексемы из внутреннего представления. Пусть N_1 и N_2 - размеры входных и выходных данных (задачи и результата), T – время, необходимое для обработки задачи.

Оценки затрат времени при передаче задачи от процессора А процессору Б приведены в табл. 2. Верхние индексы обозначают названия процессоров. Все рассмотренные временные параметры в общем случае зависят от типа процессора, и для разных процессоров разные.

В таблице не учитывается время, необходимое для передачи данных по сети, так как оно не является процессорным временем. Однако при принятии решения на передачу задачи это тоже должно учитываться, как время, необходимое на выполнение задачи.

4.4 Параллельное выполнение задач без порождения подзадач

Параллельное выполнение программ возможно даже в случае, когда четко задан порядок действий. Рассмотрим программу:

```
(+ 1 1)
(* 2 5)
(- 3)
```

Очевидно, что результатом этой программы будут 2, 10, -3 соответственно. Учитывая, что все S-выражения независимые, мы можем выполнить их параллельно несколькими процессами, затем, собрав результат, вывести его в нужном порядке. Одним из самых простых решений является посылка каждого такого задания отдельному процессору, затем сбор результатов и вывод их в правильном порядке. Для эксперимента использовалась топология звезды, центральный процессор раздает задачи остальным и собирает их. Механизм раздачи задач прост: у центрального процессора хранятся данные о занятости процессоров. Если процессор свободен, то послать ему задачу, иначе смотреть следующий процессор. В том случае, если все процессоры и так уже заняты, выполнить задачу самому. Перед посылкой задач проводится сбор и вывод результатов. Сбор результатов и вывод в правильном порядке реализуется следующим образом: каждой задаче присваивается ее номер по порядку, после чего она отсылается другому процессору. Другой процессор не имеет информации о номере задачи, так как ему это не нужно. Дисциплина обработки задач FIFO, поэтому центральный процессор хранит для каждого процессора очередь номеров посланных задач, при получении результата просто извлекает номер из очереди. Все результаты хранятся в контейнере и выводятся в корректном порядке, если необходимо, ждут результата.

Были проведены эксперименты, использующие данный подход. Первым экспериментом была раздача всех задач процессорам по порядку и так далее циклично, то есть при N процессорах, кроме центрального, i-й процессор получал все задачи k, если k есть остаток от деления порядкового номера задачи на N, при нумерации процессоров от 0 [18].

Таблица 2. Затраты времени при параллельном выполнении фазы лексического анализа

Описание	Процессор А	Процессор Б
Передача/получение задачи	$N_1(T_в^A + T_п^A)$	$N_1(T_п^Б + T_з^Б)$
Передача/получение результата	$N_2(T_п^A + T_з^A)$	$N_2(T_в^Б + T_п^Б)$
Суммарное время работы	$N_1(T_в^A + T_п^A) + N_2(T_п^A + T_з^A)$	$N_1(T_п^Б + T_з^Б) + \mathrm{T} + N_2(T_в^Б + T_п^Б)$

Центральный процессор только раздавал задачи и собирал их результаты. Данный подход хорошо работал с задачами с одинаковым временем выполнения. При задачах с различным временем выполнения этот подход мог послать все сложные задачи одному процессору, при этом остальные процессоры вынуждены были бы его ожидать.

Для балансировки нагрузки на процессоры было решено посылать задачи только свободным процессорам, а пока они все заняты, выполнять задачи центральным. Это решило проблему предыдущего подхода, но создало в себе другую. Она заключалась в том, что в предыдущем подходе процессоры были полностью заняты задачами, после выполнения одной сразу же приступали к следующей, хотя центральный после раздачи всех задач занимался только сбором результатов. Теперь же маленькие задачи могли быть розданы всем процессорам, а одна большая начала бы выполняться на центральном. После завершения обработки маленьких задач все процессоры ожидали бы центральный процессор, так как тот был бы занят. Кроме того, выполнив одну задачу, процессоры простаивали до получения новой задачи.

Результирующим решением стало объединение этих подходов в один. Центральный процессор раздает каждому процессору до k задач; если все процессоры уже получили по k задач и заняты, то выполняет задачу сам. В общем случае можно и не загружать центральный процессор задачей, чтобы исключить возможность простоя других процессоров.

Перед выполнением задачи, каждый процессор получает список новых задач, а уже потом приступает к выполнению первой задачи из очереди. Худшим все равно остается случай: k самых больших задач на одном процессоре. Рассмотрим случай, когда все задачи уже розданы, и есть свободные процессоры. Тогда имеет смысл забрать задачу из очереди одного процессора и передать ее другому. Занятому процессору с избытком задач посылается специальное сообщение «выбросить последнюю пришедшую задачу», при этом копия самой задачи посылается свободному процессору. Если сообщение опоздало, и задача уже выполняется, то она будет выполнена дважды. Никаких конфликтов пришедший результат вызвать не может, результат игнорируется. Суммарное время выполнения при этом не увеличится.

4.5 Параллельное выполнение задач с порождением подзадач

При порождении подзадач задача откладывается до вычисления результатов подзадач. Таким образом, кроме непосредственно задач в портфеле появляются еще и результаты. Lisp не определяет удобной формы представления приостановленной задачи в состоянии выполнения, поэтому отложенные задачи хранятся в оперативной памяти тех процессоров, которые их выполняли. Хранить приостановленную задачу в оперативной памяти накладно, так как оперативная память является ценным ресурсом, а накапливать отложенные задачи в оперативной памяти еще менее желательно. Таким образом, одной из важных задач будем считать освобождение памяти от приостановленных задач. Приостановленные задачи ждут результатов от подзадач, поэтому примем, что результаты подзадач важны для продолжения выполнения задачи, а продолжение выполнения ранее отложенной задачи важнее, чем выполнение новой независимой задачи, ввиду того, что независимая задача тоже может быть приостановлена и отложена до вычисления результатов подзадач.

Пусть в портфеле находятся задачи и результаты, так как их представление идентично. Предлагается следующий алгоритм распределения задач из портфеля:

– рабочий процессор свободен и посылает запрос на новую задачу;
– портфель ищет результат, необходимый для продолжения задачи, отложенной на этом рабочем процессоре, и передает этот результат;
– рабочий процессор получает необходимый результат, и, если возможно, продолжает выполнение отложенной задачи.

В действительности необходимых результатов может не быть, тогда процессор получит независимую задачу. Данное описание алгоритма не является полным, однако его достаточно для распределения задач и результатов по процессорам.

4 Программная реализация транслятора LISP с полной прозрачностью параллелизма

Программная реализация описанных выше методов осуществлялась с использованием среды разработки Microsoft Visual C++ 7.1 .NET и библиотеки MPICH2 версии 1.0.5 для платформы Win32. Полученная реализация транслятора пригодна для исследования на предмет увеличения производительности Lisp-программ для кластеров, поддерживающих библиотеку MPI. Реализация содержит в себе возможности для дальнейшего расширения встроенных функций языка, а также для замены модулей программы, используя полиморфизм.

5 Экспериментальные исследования параллельной трансляции на фазе лексического анализа

Увеличение производительности транслятора на архитектуре IBM PC сталкивается с одной серьезной проблемой. Экспериментальные исследования, связанные с разработкой трансляторов языков функционального программирования проводились на машинах с другой архитектурой процессоров [3-11]. Для языка Lisp это связано с интенсивным обращением к оперативной памяти. Существующие на сегодняшний момент процессоры платформы IBM PC рассчитаны на обработку массивов в памяти. Массивы в памяти располагаются последовательно, поэтому существующий механизм кэширования памяти в таких процессорах успешно работает. В функциональных языках основным рабочим представлением данных является список, и его представление в памяти не является одним непрерывным участком данных. С этим связаны большие задержки в выполнении программ, так как архитектура памяти располагает к ускоренной работе с непрерывными большими участками данных, а не с разбросанными маленькими. Таким образом, реализованные в архитектуре IBM PC оптимизированные функции по работе с массивами данных в функциональных программах как таковые не используются. Расширенные наборы инструкций рассчитаны на обработку массивов. Однако операции запаковки списка в массив и распаковки массива в список практически теряют смысл в связи с тем, что преобразования списка в массив и назад осуществляется дольше, чем простое выполнение вычислений без преобразований.

Переход на процессоры марки Core 2 ф.Intel улучшает производительность при работе с памятью, но все же остается ориентированным на работу с массивами. Отсюда следует вывод о том, что существующие аппаратно-независимые трансляторы Lisp на IBM PC не используют все возможности процессора. Низкая загрузка связана с тем, что процессор будет тратить время на ожидание чтения из оперативной памяти. На уровне аппаратуры возможны различные оптимизации для увеличения эффективности транслятора, однако они требуют введения в программу транслятора аппаратно зависимых инструкций. Введение таких инструкций в текст функциональной программы сделает ее аппаратно-зависимой, что противоречит постановке задачи.

Попробуем увеличить производительность трансляции путем параллельного выполнения лексического анализа несколькими процессорами, используя общую таблицу символов на отдельном процессоре.

В качестве источника тестовых заданий был использован генератор функций для обращения к элементу списка. В языке Lisp для работы со списками используются функции вида:

```
(defun cddr (A) (cdr(cdr a)))
(defun cadr (A) (car(cdr a)))
(defun cdar (A) (cdr(car a)))
(defun caar (A) (car(car a)))
```

Каждая буква в названии функции обозначает соответствующий вызов в ее теле: символ “a” означает вызов селектора car, символ “d” – вызов cdr. Исходя из принципа построения данных функций, был создана программа, генерирующая заданное число подобных функций. Этот генератор функционирует как утилита разработанного транслятора языка Lisp.

Для тестирования при помощи генератора тестов было создано 65536 и 131072 функций. Такое количество функций связано с тем, что при меньших значениях велико значение погрешности во времени исполнения. В табл. 3-5 приводится время в секундах, необходимое для выполнения этапа лексического анализа программы. В поле “Количество процессов” в скобках указывается число рабочих процессоров, выполняющих чтение из файла текста исходной программы плюс один процессор, выполняющий функции общей таблицы символов.

Полученные данные показывают накладные расходы при переходе на MPI. Их можно заметить в разнице времени исполнения между первым и вторым столбцами таблиц 3 и 4. На основании этих данных было выдвинуто предположение о том, что при увеличении количества процессоров уменьшится время фазы лексического анализа. Как видно из этих таблиц, при дальнейшем увеличении количества процессоров суммарное время выполнения не увеличивается значительно. Это означает, что при увеличении числа процессоров задача лексического анализа хорошо распараллеливается.

Таблица 3. Суммарное время работы при тестировании 65536 функций вида CADADR

Кол-во процессов	1	2(1+1)	3(2+1)	5(4+1)	9(8+1)
Опыт 1	9,10	23,59	24,946	26,38	27,16
Опыт 2	9,16	24,14	24,16	26,09	29,98
Опыт 3	9,11	25,53	25,39	26,15	26,51
Среднее	9,13	24,42	24,83	26,21	27,83

Таблица 4. Суммарное время работы при тестировании 131072 функции вида CADADR

Кол-во процессов	1	2(1+1)	3(2+1)	5(4+1)	9(8+1)
Опыт 1	19,38	48,27	50,14	52,48	53,80
Опыт 2	19,57	48,33	50,75	51,79	53,38
Опыт 3	19,41	46,36	48,58	51,44	53,60
Среднее	19,45	47,56	49,83	51,90	53,59

В действительности же оказалось, что время полезной работы таблицы символов с использованием MPI практически очень близко ко времени работы лексического анализатора и таблицы символов на одном процессоре.

Пусть T_c - время полезной работы таблицы символов, а $T_л$ - время полезной работы лексического анализатора. Тогда общее время полезной работы T_p будет равно сумме полезных работ лексических анализаторов и таблицы символов. Пусть N – количество лексических анализаторов, тогда суммарное время полезной работы - $T_p = T_c + N * T_л$, тогда $T_л = \frac{T_p - T_c}{N}$

Таблица 5. Время выполнения лексического анализа

Количество процессоров	1	2	4	8
65536 функций	15,29	7,85	4,27	2,34
131072 функции	28,11	15,19	8,11	4,29

На основании этих данных мы можем найти зависимость ускорения выполнения лексического анализа от количества процессоров. Под ускорением будем понимать отношение времени выполнения задачи на одном процессоре ко времени ее выполнения параллельно на нескольких процессорах.

Проанализируем зависимость ускорения времени работы *S* от количества процессоров *N* при опыте с 65536 функциями (для случая 131072 функций результат аналогичен).

Для интерполяции по МНК подобран квадратичный многочлен:

$P_3(x) = 0.088 + 0.949x - 0.018x^2$

О полученной зависимости можно сказать, что с увеличением количества процессоров увеличение производительности постепенно падает. Более 26 процессоров использовать нецелесообразно, так как начиная с 27 процессора включительно ускорение уменьшается.

Проведенный анализ показал следующее. Увеличение производительности путем распараллеливания лексических анализаторов с синхронизацией символов в глобальной таблице символов считается тупиковым, так как максимальная загруженность приходится на таблицу символов. При этом даже один лексический анализатор выполняет всю свою работу быстрее таблицы символов. Таким образом, увеличение их количества не повлияет на суммарное время выполнения: все равно процессы сканеров будут ожидать процесс таблицы символов.

По результатам этого эксперимента сделан вывод о том, что поддержание работы глобальной таблицы символов несет в себе большие накладные расходы, чем собственно лексический анализ. Это показало нецелесообразность распараллеливания этапа лексического анализа данным способом – лексический анализатор и таблица символов должны выполняться на одной машине.

Полученные экспериментальные данные также можно использовать для целесообразности организации конвейерной обработки функциональной программы. Работу транслятора можно разбить на последовательные стадии:

– лексический анализ;
– синтаксический анализ/загрузка данных во внутреннее представление;
– виртуальная Lisp машина;
– вывод результатов.

Характер их работы последовательный и происходит в том порядке, в каком они перечислены. Возможным решением по увеличению производительности транслятора является конвейерная обработка фаз трансляции. Теоретически, конвейерная обработка увеличивает производительность, на практике создание подобного конвейера сталкивается с некоторыми проблемами. Основной из них является механизм реализации конвейера. При использовании библиотеки MPI необходимые затраты процессорного времени для передачи сообщений, в особенности по локальной сети, превышают время, необходимое на проведение одной стадии конвейера. Это связано с простотой работы каждой стадии, так как по сложности они соизмеримы с формированием и посылкой пакетов (оба действия - линейной сложности). Кроме того, интерфейс MPI не содержит в себе средств синхронизации, такие как события, семафоры или мьютексы. Для реализации подобного конвейера более подходит

использование библиотеки Posix Threads, но комбинирование двух библиотек MPI и PThreads является излишним. С другой стороны, само использование таких мощных средств синхронизации, как семафоры, несет в себе накладные расходы, соизмеримые со стадиями работы конвейера. Данное заключение может показаться сомнительным, но более детальное исследование подобного конвейера потребует перевода всех конструкций транслятора на ассемблер ввиду простоты алгоритма и сравнения времени его работы с использованием объектов операционной системы.

Используя данные, полученные при распараллеливании стадии лексического анализа, можно сделать вывод о том, что организация подобного конвейера на уровне программного обеспечения не дает увеличения производительности. Теоретически, реализованный на аппаратном уровне конвейер действительно увеличил бы производительность. Второй проблемой является то, что лексический анализатор берет данные из файла, находящегося в устройстве ввода-вывода. Проблема заключается в том, что стадии работы транслятора, такие как лексический и синтаксический анализ с загрузкой, на практике проходят быстрее, чем само чтение лексем из файла. При запуске программы в ОС Windows XP данные стадии работы транслятора не загружали процессор полностью, что означает свободное процессорное время при проведении операций чтения. Реальным способом увеличения производительности лексического анализа является ускорение операций чтения исходного текста функциональной программы.

Выводы

Предложенные механизмы определения взаимного невмешательства ранее не использовались в трансляторах функционального языка Lisp. Использованные методы параллельного выполнения, распределения и представления задач не использовались в комплексе, что позволило получить новые данные об эффективности трансляторов языка Lisp. Предложенные методы используются для достижения полной прозрачности параллельного выполнения фазы трансляции, что является основной целью работы

Практические результаты экспериментов с разработанным транслятором могут быть использованы в качестве дополнительной информации при написании параллельных трансляторов. Разработанный транслятор языка Lisp [19] может быть использован для анализа функциональным программ с целью их распараллеливания и применен для рефакторинга.

Литература

1. Фаулер М. Рефакторинг: Улучшение существующего кода.- СПб: Символ, 2003.
2. McCarthy J. Lisp 1.5 Programmer's Manual. – Cambridge: MIT, 1985. – 106p.
3. Tanaka Tomoyuki, Uzuhara Shigeru. Futures and multiple values in parallel Lisp, Lisp Conference, 1992. – 16p.
4. Yuasa Taiichi, Hagiya Masami. Kyoto Common Lisp Report, Kyoto University, 1985. -87p.
5. Robert H. Halstead, Multilisp: a language for concurrent symbolic computation// ACM Transactions on Programming Languages and Systems. Volume 7. Issue 4, 1985. – pp. 501–538,
6. Goldman R., Gabriel R.P. Qlisp: Parallel processing in Lisp, Lucid Inc., 1993. – 24 p.
7. Jagannathan S. TS/Scheme: Distributed Data Structures in Lisp, Lisp and symbolic computation: An International Journal, 7, 1994. – pp. 283-305.
8. Feng M.D., Wong W.F., Yuen C.K. Compiling parallel Lisp for a shared memory multiprocessor. National University of Singapore, 1994. – 27p.
9. Freely M. A message passing implementation of lazy task creation. Montreal University, 1993. -16p.
10. Jr. David A. Kranz, Robert H. Halstead and Eric Mohr. Mul-T: A high-performance parallel lisp/ ACM Symposium on Programming Language Design and Implementation, 1989. - pp 81-90.
11. Mohr E. Lazy task creation: A technique for increasing the granularity of parallel programs/ ACM Conference on Lisp and Functional Programming, 1990. – pp. 185-197.
12. Чери С., Готлоб Г., Танка Л. Логическое программирование и базы данных.- М.: Мир, 1992. – 352 с.
13. Хювёнен Э., Сеппянен И. Мир Лиспа. Том 1 – Введение в язык Lisp и функциональное программирование – М: Мир, 1990. – 434 с.
14. Эндрюс Г. Основы многопоточного, параллельного и распределенного программирования. – М: Вильямс, 2003. – 512 с.
15 Хювёнен Э., Сеппянен И. Мир Лиспа. Том 2 – Методы и системы программирования – М: Мир, 1990. – 318 с.
16. Лавров С.С., Силагадзе Г.С. Автоматическая обработка данных. Язык Lisp и его реализация – М: Наука, 1978. – 176 с.
17. Ахо А., Сети Р., Ульман Дж. Компиляторы: Принципы, технологии и инструменты. – М: Вильямс, 2003. – 768 с.
18. Антонов А.С. Параллельное программирование с использованием технологии MPI. - М.: Изд-во МГУ, 2004. – 71 с.
19. Савков К.Г., Дацун Н.Н. Увеличение производительности транслятора языка Лисп благодаря использованию параллельных вычислений/ інформатика та комп'ютерні технології. Матеріали III науково-технічної конференції молодих учених та студентів. – Донецьк: ДонНТУ – 2007. - с. 486-489.