УДК 681.3

Исследование возможностей параллельного выполнения 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 possibilities of parallel interpretation of functional programs". *The article examines methods of paralleling of Lisp program interpretation process. Characteristics of possible acceleration of phase interpretation-execution are received at parallel realization of the compiler functional programming language.*

Введение

Рефакторинг [1] уже существующего программного обеспечения для более эффективного выполнения на параллельных архитектурах является актуальной задачей. Целесообразность рефакторинга определяется доступностью и качеством средств автоматического распараллеливания последовательных программ. К таким средствам относится описанный в [2] транслятор функционального языка Lisp [3] с полной прозрачностью параллелизма. Он предназначен для автоматизации процесса параллельной интерпретации (трансляции и выполнения) последовательных программ и может быть использован как инструмент для рефакторинга функциональных программ.

Целью работы является исследование возможности увеличения производительности транслятора функционального языка Lisp с использованием параллельных вычислений на фазе выполнения. Основными задачами исследования являлись:

- создание транслятора языка Lisp с полной прозрачностью параллелизма;
- исследование результатов на фазе выполнения функциональных программ.

Объектом исследования является транслятор функционального языка Lisp. Предметом исследования является фаза выполнения интерпретатора, на которой возможно применение параллельных вычислений для увеличения производительности.

1 Анализ трансляторов языка Lisp, реализующих параллелизм вычислений

В работе [2] авторы проанализировали возможности параллельных реализаций функциональных языков: Multilisp[4], Qlisp[5], BandaLisp[6], TS/SCHEME[7], Kyoto Common Lisp[8].

В качестве системы-аналога в данной работе использован Qlisp [5], так как он содержит наиболее мощные конструкции для указания параллельного выполнения программы по сравнению со всеми рассмотренными в [2] трансляторами функциональных языков. Подход, использованный в Qlisp и в трансляторе [2] – это параллельное выполнение очереди процессов. В работе [2] и в данной статье рассматривается транслятор языка Lisp с полной прозрачностью параллелизма, ориентированный на теоретические исследования. Задачи этих исследований - анализировать функциональные программы на фазе выполнения и определять пути увеличения их производительности.

2 Постановка задачи

Основываясь на данных о современных параллельных реализациях трансляторов функциональных языков, выделим основные требования к разрабатываемому транслятору:

- полная прозрачность параллелизма;
- удобный механизм отладки параллельных приложений;
- возможность анализа программ для определения целесообразности распараллеливания на фазе выполнения.

Для обеспечения автоматического рефакторинга уже существующих последовательных функциональных программ необходимо обеспечить полную прозрачность параллелизма, так как в противном случае требуется модификация программистом текстов программ.

MPI является современным стандартом параллельных вычислений, его поддерживают большинство мультипроцессорных систем и кластеров. Для многих платформ созданы реализации этой библиотеки, поэтому именно она использована в данном трансляторе. Отладочные

средства систем программирования функциональных языков гораздо слабее по своим возможностям по сравнению с современными отладчиками процедурных языков. Средства отладки параллельных программ предоставляет библиотека MPI, поэтому транслятор языка Lisp должен быть ориентирован на работу с ними. С другой стороны, внутреннее представление Lisp-программы, использованное в данном трансляторе, позволяет встроить него собственные отладочные средства.

3 Используемые методы и алгоритмы

В работе [2] были сформулированы основные методы и алгоритмы, используемые в параллельной реализации транслятора языка Lisp с полной прозрачностью параллелизма. Для определения параллельности вычисления S-выражений используется модификация метода непересекающегося множества переменных. Параллельное выполнение основано на парадигме портфеля задач.

В данном трансляторе задача и результат ее выполнения передаются в виде текста программы (единообразно). Использование текста программы вместо ее внутреннего представления замедляет вычисление результата (каждый раз будут выполняться фазы лексического и синтаксического анализа, а также загрузка программы во внутреннее представление списка). Но при передаче текста задачи упрощается процесс отладки программ, так как программист может в явном виде увидеть непосредственно исполняемую программу и ее результат [2].

3.1 Виртуальный ассемблер

При параллельном выполнении Lisp-программ может потребоваться остановка процесса, чтобы сохранить его состояние и перейти к другому процессу. В трансляторе применяется асинхронный ввод/вывод, то есть ни один из процессоров не находится в состоянии ожидания, вместо ожидания он переключается на следующую задачу, пока необходимые для продолжения выполнения результаты подзадач не переданы ему. Например, при вычислении (+ (+ 1 2) (+ 3 4)) процессор зайдет в вызов самой первой функции сложения. Там возникают условия для распараллеливания, т.е. (+ 1 2) будет выполнено им самим, а (+ 3 4) послано другому процессору. Возможно, что результаты переданной другому процессору задачи придут позже, чем процессор сам ее выполнит. Тогда ему придется ждать результата. Ожидание результата не приносит пользы, так как процессор простаивает. Таким образом, процессор просто отложит эту задачу до получения результата и начнет выполнять другую.

Основной сложностью в реализации отложенной задачи является то, что когда процессор дойдет до необходимости ожидания результата, он будет находиться в глубине рекурсивных вызовов. Один из путей решения этой проблемы - использование средств ОС для управления процессами. Но в данной работе уже используется MPI, поэтому использование потоков ОС излишнее (так как удобно было бы изначально пользоваться библиотекой Pthreads).

Таким образом, принято решение о замене рекурсии при выполнении на виртуальной машине на последовательное выполнение. На уровне аппаратуры, в частности процессора, используемая рекурсия представляет собой последовательное выполнение. Будем использовать точно такой же подход и для перевода рекурсии Lisp-программы в последовательное выполнение. Для этого организуем виртуальную машину так:

- процессорные регистры;
- стек;
- внутренние программы только для чтения.

Сохранение задачи в момент выполнения сводится к сохранению регистров процессора и содержимого стека. Кроме этого, к задаче относится Lisp-память и используемые свойства атомов. Этот подход широко используется в современных компьютерах на аппаратном уровне. Кроме того, при таком подходе виртуальная машина обладает встроенными средствами для отладки Lisp- программ. Использование рекурсии в виртуальной машине совместно с отладкой приложения в среде разработки Microsoft Visual Studio довольно удобно, но конечный пользователь не будет обладать должными средствами. Поэтому для него удобным будет показ рекурсивных вызовов с раскруткой стека, встроенный в программу транслятора.

Моделирование выполнения виртуальной Lisp-программы на последовательном виртуальном процессоре может показаться сложной задачей, но для моделирования достаточно упрощенной модели процессора.

Упрощенный процессор будет похож на настоящий в том плане, что у него будут примерно такие же регистры и команды. Разрядность регистров виртуального процессора на самом деле будут соответствовать аппаратной разрядности машины. Для 32-х разрядной ЭВМ все регистры будут 32-х разрядные. Особо важным для реализации является то, что представление целых чисел и указателей одинаковое по размеру. Упрощенный процессор будет иметь регистры:

- IP (instruction pointer) – адрес выполняемой команды;
- FLAGS – регистр флагов;

- RA, RB, RC, RD – регистры общего назначения.

При выполнении программы IP есть индекс ячейки памяти из внутренних программ. Стек и память для программ размещены отдельно, одно и то же значение индекса указывает на разные значения ячейки памяти в стеке или программе. Это сходно с существующими процессорами (сегмент кода и стека), но в нашей виртуальной машине отсутствует сегмент данных. Стек представлен стандартным контейнером “массивом”, верхушкой стека считается конец массива. Это упрощает работу со стеком, при этом делает ненужным указатель на верхушку стека, так как он и так хранится в контейнере.

Представление программы для такого процессора реализовано с использованием объектно-ориентированных технологий, в частности полиморфизма. Все команды хранятся в стандартном контейнере “массив” (vector), при этом текст программы на виртуальном ассемблере не меняется – текст загружается разово при запуске транслятора. Так как это массив, то он поддерживает непосредственную адресацию каждого элемента по индексу. Содержимое каждого элемента массива – это указатель на объект-команду. Все команды наследуются от одного и того же чистого виртуального класса команды. Каждая команда сама ответственна за свое выполнение, содержит функцию *execute*, которая и выполняет команду; в качестве аргумента ей передается указатель на виртуальный процессор.

Команды процессора сходны по написанию и смыслу с командами ассемблера, но снабжаются префиксами. Префиксов обычно столько же, сколько и аргументов у команды. Префиксы имеют следующее значение:

- v – непосредственное указание значения в тексте программы;

- r – указание ID регистра, который содержит значение;

- l – значение локальной переменной.

Ячейка содержит целое 32-х разрядное число, хотя ее содержимое может трактоваться некоторыми функциями, как указатель. При указании ID–регистра значение берется из соответствующего регистра. Все регистры имеют свой уникальный от других регистров ID–номер. Значение локальной переменной требует некоторого пояснения. Как известно – локальные переменные функций хранятся в стеке, при этом аргумент команды - индекс этой локальной переменной, считая от верхушки стека.

Основными функциями виртуальной машины, которые переписаны для выполнения на виртуальном процессоре, являются *eval* и *apply*. Вычисление функций типа EXPR, требующих вычисление аргументов, а также вычисление лямбда-выражений, производится внутри функции *apply* виртуальной машины. Разделение на параллельные задачи напрямую связано сфункцией *evallist*. В этой функции производится вычисление списка аргументов. При условии взаимного невмешательства каждого элемента списка аргументов, каждый аргумент может быть вычислен отдельным процессом. Исключение составляет случай с одним аргументом, так как его целесообразно вычислить на месте. Выполнение *evallist* можно описать следующим образом:

- для каждого вычисляемого аргумента, начиная со второго, порождается новая задача;

- выполнение программы продолжается до вычисления первого аргумента;

- если все следующие аргументы уже вычислены другими процессами и результат получен, то продолжить выполнение.

В случае, когда следующие аргументы еще не вычислены – задача откладывается до их получения. Процессор при этом не простаивает, а выполняет следующие задачи. Таким образом, у каждого процессора будут храниться отложенные задачи с указанием необходимых для их продолжения результатов, а также активные задачи, которые ждут процессорного времени. Пока процессор занят работой, к нему будут приходить задачи с других процессоров, и добавляться в очередь. Каждая задача будет иметь свой уникальный номер. Уникальность номера достигается тем, что он состоит из двух частей – номера процессора и порядкового номера задачи. Каждый процессор в системе имеет свой номер и свой счетчик порожденных задач. При порождении задач он увеличивается, а в связи с номером процессора является уникальным. Номер задачи является целым 32-х разрядным числом. При его переполнении он обнуляется и продолжает свой отсчет сначала. Хотя теоретически конфликт задач при совпадении номера задачи и процессора возможен, на практике это маловероятно. Если же таковое случится, то самым простым решением будет увеличить разрядность переменной, хранящей номер задачи.

Вызов функций в данном виртуальном процессоре в основном происходит следующим образом: в регистр RB записывается аргумент, а в регистр RA - результат. Если же требуется несколько аргументов, они могут быть переданы через регистры или стек.

Описания команд виртуального процессора приведены в таблицах 1-3.

Таблица 1 – Базовые команды виртуального ассемблера

Команда	Описание
Ocmprv Register Value	Вычитает из значения указанного регистра значение и устанавливает флаги равенства и знака. Содержимое регистра не меняется.
Ojmp Address	Безусловный переход на команду по адресу
Oje Address	Переход на команду по адресу, если установлен флаг равенства
Ojne Address	Переход на команду по адресу, если флаг равенства не установлен
OCall Address	Производит вызов функции по непосредственно заданному адресу в памяти программы. В стек помещается адрес следующей команды и указатель на инструкции переходит на указанный адрес.
Omovrl Register Local	Записывает в регистр Register значение локальной переменной Local
Opopr Register	Выталкивает со стека значение в регистр
Opushr Register	Вталкивает в стек значение регистра
Oret N	Выталкивает из стека N элементов, затем выталкивает из стека адрес перехода и производит безусловный переход

Таблица 2 – Внутренние команды виртуального ассемблера

Команда	Описание
Oassertlambdaexpression	Проверяет, указывает ли RB на лямбда-выражение.
Ogetatomprops Prop	Возвращает в регистр RA значение свойства Prop атома RB
Olispevalsymbol	Вычисляет значение символьного атома RB, результат возвращается в RA
Olispgetsymbolid	Записывает в RA идентификатор символа RB
Olispgettype	Записывает в RA идентификатор типа RB
Olisppoplink	Восстанавливает связи в A-списке
Olisppushlink	Сохраняет связи A-списка и связывает формальные аргументы RA и фактические RB
Olispsystemcheck	Проверяет, указывает ли RB на вызов функции system. Если указывает, то завершает выполнение программы.
Orterror ptrError	Выводит сообщение ptrError и содержимое списка из RB на экран и аварийно завершает работу
Ovmcall	Берет из регистра RA указатель на внутреннюю функцию и вызывает ее с аргументом RB

Таблица 3 – Команды виртуального ассемблера для работы со списками

Команда	Описание
Olispcar	Вызывает car Lispa для RB, результат записывается в RA
Olispcdr	Вызывает cdr Lispa для RB, результат записывается в RA
Ocons	Вызывает cons из Lispa с аргументами RA и RB. Результат вызова записывается в регистр RA

3.2 Метод подготовки эксперимента параллельного выполнения Lisp-программ

Процесс идентификации задачи и фиксации времени ее выполнения может быть выполнен путем добавления специальной метки. Метка будет содержать порядковый номер процессора и порядковый номер задачи, а также время поступления задачи на обработку. Кроме того, ввиду идентичности задачи и результата, добавляется еще специальная информация для различения задач и результатов:

```
Буквенный префикс Номер процессора
. Номер задачи : Время поступления
задачи
```

Рисунок 1 – Формат метки

Буквенные префиксы – это T (Task – задача) и R (Result – результат). Так R5.6:7 – есть метка, обозначающая результат выполнения восьмой задачи шестого процессора в момент времени 7. Нумерация процессоров начинается с нуля, процессор номер 0 производит распределение задач по другим процессорам и протоколирует их работу. Нумерация задач начинается с единицы, время начинается с 0. На практике это должно быть время от начала работы процессора до того момента, когда он завершил выполнение задачи и сформировал результат. В данной работе показатель времени не обязан точно показывать фактическое время работы процессора, вместо этого реального времени можно ставить время, необходимое для моделирования операции. Это позволяет абстрагироваться от некоторых задержек, не

связанных с работой самой программы для получения теоретического результата. Для данной работы это важно ввиду того, что программа запускается в среде Microsoft Windows. Операционная система может в фоновом режиме запускать системные утилиты, проводить диагностику неполадок или принимать пакеты по сети, при этом эти задержки в выполнении не отразятся на результатах работы программы. Время работы процессоров может вычисляться по ходу выполнения Lisp-программы, например, увеличиваться на единицу при выполнении операции сложения. Таким образом, мы можем анализировать не только время выполнения программы, но и количества вызовов тех или иных внутренних функций транслятора. Применение данного подхода удобно и оправдано для моделирования, так как исключает влияние на результаты работы многих негативных факторов. В частности, для получения результатов моделирования можно использовать всего один процессор с переключением задач. Так как в общем случае показатель времени процессора может не соответствовать реальному времени его работы, будем называть его виртуальным, чтобы различать это время от реального практического. Время в метке - показатель виртуального времени процессора на момент передачи задачи или результата. Далее в тексте данной статьи время работы процессоров берется из метки и является виртуальным временем для моделирования.

Процессор и номер задачи обозначены для процессора, породившего задачу и ожидающего результата ее выполнения. Само же выполнение этой задачи может быть выполнено другим процессором. Передача задач и результатов происходит через общий портфель, находящийся на процессоре номер 0 с целью протоколирования выполнения.

Как и в классическом варианте портфеля задач, каждый свободный процессор будет запрашивать из портфеля задачи, но с указанием его собственного времени. Так, определив виртуальное время, необходимое на выполнение операции, мы используем его для моделирования выполнения. Каждый процессор в отдельности имеет свой собственный счетчик виртуального времени для вычисления статистических данных, таких как время загрузки и простоя процессора.

Рассмотрим пример вычисления выражения на трех рабочих процессорах, при этом будем считать, что время операции сложения равно 1, всем остальным временем можно пренебречь:

(+ (+ (+ 1 1) (+ 2 2)) (+ (+ 3 3) (+ 4 4)))

Первоначальная запись задачи будет T0.1:0, далее текст задачи. Все три рабочих процессора на данный момент находятся в виртуальном времени 0, делают запросы новых задач из портфеля. В портфеле пока находится только одна задача T0.1:0. Пусть она передана процессору номер 1. В процессе вычисления он разделит ее на две подзадачи:

T1.1:0 (+ (+ 1 1) (+ 2 2))

T1.2:0 (+ (+ 3 3) (+ 4 4))

Эти задачи будут посланы в портфель, а в самом процессоре номер 1 сохранится остановленная задача T0.1:0, требующая результатов R1.1 и R1.2.

Теперь в портфеле находится две задачи – T1.1:0 и T1.2:0. Пусть первая из них пойдет процессору номер 2, а вторая - процессору номер 3. Аналогично, процессор номер 2 породит:

T2.1:0 (+ 1 1)

T2.2:0 (+ 2 2)

В процессоре номер 2 будет храниться задача T1.1:0, ожидающая R2.1 и R2.2. Третий же процессор породит задачи:

T3.1:0 (+ 3 3)

T3.2:0 (+ 4 4)

Также он сохранит T1.2:0, ожидая R3.1 и R3.2

Теперь в портфеле хранится 4 задачи, пусть T2.1:0 пойдет на процессор номер 1, T2.2:0 - на процессор номер 2, T3.1:0 - на процессор номер 3. Эти задачи будут непосредственно решены, и будут получены результаты R2.1:1, R2.2:1, R3.1:1, которые будут посланы в портфель. Все 3 процессора выполнили сложение, поэтому их виртуальное время увеличилось на 1.

Портфель же задач содержит одну задачу, T3.2:0, которую желательно послать процессору еще во время 0, но так как таких процессоров уже нет, можно послать ее и процессору во время 1. Виртуальное время при моделировании будет трактоваться аналогично реальному: если у задачи время поступления 0, а посылается она процессору, который уже во времени 1, то это означает, что обработка задачи была отложена. Если же задача во времени 1, а процессор во времени 0, то посылка задачи процессору будет означать, что процессор простаивал, а только потом получил задачу. Приоритет посылки задач будет определяться по номеру процессора, первую задачу получит первый подходящий по номеру процессор. Это приводит к большей загрузке процессоров с меньшими номерами, без потери в суммарном времени выполнения.

Сначала портфель раздаст ожидаемые результаты процессорам при условии, что они не вызывают простоя процессора. Второй процессор получит R2.1 и R2.2, и сразу же начнет выполнение отложенной T1.1:0,

перейдет во время 2 и вернет R1.1:2 в портфель. Явно видно, что процессор 1 находится еще во времени 1, нет смысла возвращать ему результат R1.1:2, так как это приведет к простою до времени получения результата. Вместо этого процессор 1 получит T3.2:0, перейдет во время 2 и вернет в портфель R3.2:2. Третий процессор еще раньше получил R3.1:1, теперь получит R3.2:2, что означает его простой до получения результата. Получив результат, он начнет выполнение T1.2:0, вернув R1.2:3. Теперь остаются только R1.1:2 и R1.2:3, которые получит первый процессор, и закончит T0.1:0 с результатом R0.1:4

Полученный результат содержит суммарное время выполнения – 4. Затем для каждого процессора можно посчитать время полезной работы и время простоя. Полученный результат загрузки процессоров не является однозначным, так как, например, вместо третьего процессора мог работать первый.

Приведенный подход не является оптимальным, но он является простым и обеспечивает простые принципы балансировки нагрузки путем распределения задач на процессоры из портфеля. Предложенный алгоритм распределения задач из портфеля использован для проведения эксперимента. В ходе эксперимента для суммирования были получены достаточно правдоподобные результаты, что позволяет сделать предположение о пригодности данного метода для анализа более сложных программ.

Данный подход не гарантирует оптимального распараллеливания программы по процессорам и рассчитан на однородные процессоры. Результаты, полученные для занятости процессоров, могут быть использованы для анализа увеличения производительности, а также для определения оптимального количества процессоров при решении конкретной задачи.

Для определения оптимального количества процессоров для решения задачи фиксированного размера введем критерий оценки эффективности работы. Эффективность работы оценивается, используя утверждения:

- чем меньше суммарное время выполнения задачи, тем лучше;
- чем меньше простаивают процессоры, тем лучше.

В качестве эталона примем вариант выполнения задачи на одном процессоре. Зная время выполнения задачи на одном процессоре, мы можем определить выигрыш во времени при использовании нескольких процессоров в виде разности между временем выполнения на одном процессоре и временем выполнения задачи с использованием параллельных вычислений.

Введем две величины – стоимость ускорения решения задачи на одну секунду C и штраф за простой одного процессора в течение секунды P. Тогда для сравнения различных вариантов выполнения одной и той же задачи мы можем использовать числовую величину критерия эффективности:

$$(T_1 - T_n) * C - P * \sum_{i=1}^{n} w_i$$

где:

T_1 - время, необходимое на выполнение задачи на одном процессоре;

T_n- время, необходимое для выполнения задачи на n процессорах;

w_i - время простоя (ожидания) i-го процессора.

Если применить эту формулу для одного процессора, то получим 0, так как мы не получили никакого выигрыша по времени, но и не получили штрафов за простои процессора. Отсюда видно, что при значении этого критерия меньше нуля использование такого количества процессоров нецелесообразно ввиду их простоя. Если же критерий больше нуля, то выигрыш во времени перекрывает негативные последствия простоя процессоров. Открытой остается проблема подбора значений стоимости и величины штрафов. Их можно связать некоторым коэффициентом пропорциональности k, который будет показывать, на сколько выигрыш одной секунды времени при решении задачи важнее одной секунды простоя одного процессора, $k*C=P$. В таком случае можем переписать критерий, используя только один параметр k, если принять величину штрафа P постоянной:

$$(T_1 - T_n) * k * P - P * \sum_{i=1}^{n} w_i$$

$$(T_1 - T_n) * k - \sum_{i=1}^{n} w_i$$

Приведенный критерий приводится здесь для демонстрации простейшего применения полученных результатов моделирования. Более глубокий подход к определению оптимального количества процессоров требует большего количества данных и более мощного критерия эффективности, в данной же работе описаны только базовые предпосылки для подобных исследований.

5 Экспериментальные исследования параллельного выполнения функциональных программ

5.1 Предпосылки параллелизма на стадии выполнения

Опишем стандартные арифметические функции языка Lisp, к ним относятся сложение, умножение, вычитание и деление. В обычной записи сложения в виде (+ X_1 X_2 … X_n) подразумевается сумма элементов от X_1 до X_n включительно. С точки зрения математики, сложение есть функция двух аргументов.

Если бы в Lispe сложение было бы функцией только от двух аргументов, то оно выглядело бы так:

$$(+X_1\ (+X_2\ (+\ \ldots\ (+\ X_{n-1}\ X_n)\ldots).$$

Эта запись эквивалентна предыдущей, но сложение происходит от конца к началу – первыми складываются X_{n-1} и X_n, затем к ним прибавляется X_{n-2} и далее до X_1. Можно переназначить индексы таким образом, чтобы сложение было от начала к концу. Для N чисел требуется $N-1$ операция сложения, но при такой записи явно виден последовательный характер вычислений. Такая запись не подходит для распараллеливания, так как требует последовательных вычислений.

Изобразим ход вычислений в виде дерева, ветвями и корнем которого будут операции, а листьями аргументы. Как видно на рис.2-3, при одной форме записи сложение может выполняться только последовательно, при другой же возможно параллельное выполнение суммирования.

Считая сложение бинарной операцией, мы можем записать сложение восьми чисел таким образом:

$$(+X_1\ (+X_2\ (+\ X_3\ (+X_4\ (+X_5\ (+X_6\ (+\ X_7\ X_8)))))))$$

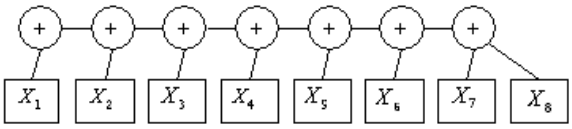

Рисунок 2 – Последовательное суммирование

$$(+\ (+\ (+\ X_1\ \ X_2)\ (+\ X_3\ X_4))\ (+\ (+\ X_5\ X_6)\ (+\ X_7\ X_8))))$$

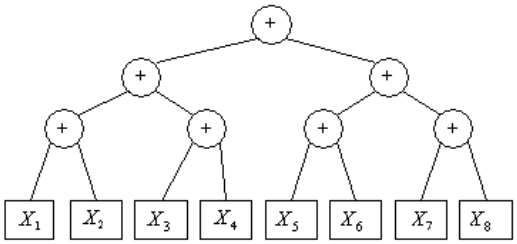

Рисунок 3 – Параллельное суммирование

Суммирование не является единственной операцией, которую можно выполнить подобным образом, к аналогичным операциям относятся произведение, поиск максимума или минимума. Вычитание и деление в Lisp также можно подвергнуть таким же преобразованиям. На рис. 4 показаны примеры форм записи операций сложения и вычитания. Аналогично сложению можно преобразовывать умножение, вычисление максимума или минимума, логические «и» и «или», операции объединения и пересечения множеств.

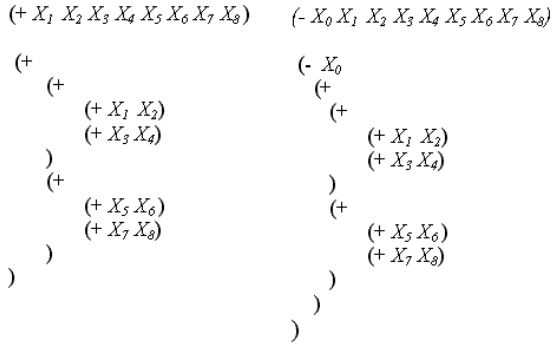

Рисунок 4 – Формы записи для вычисления суммы и разности

Эти операции требуют использования всех аргументов. Логические «и» и «или» могут быть вычислены аналогично, но для ускорения вычислений могут вернуть результат преждевременно, не вычисляя все аргументы. В описании языка Lisp о быстром вычислении логического «и» и «или» ничего не сказано, таким образом, оно является необязательным расширением языка. Функции сравнения в языке Lisp обобщены на произвольное количество аргументов, например, функция “>”:

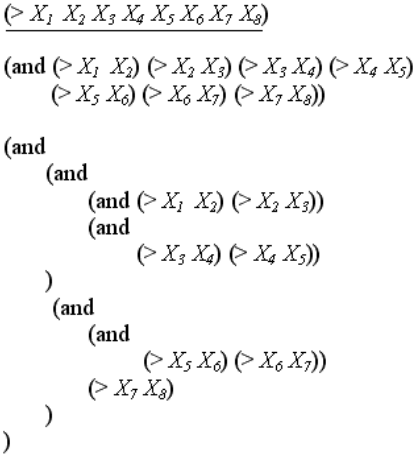

Рисунок 5 – Форма записи предиката “убывают”

Следует отметить, что приведенная для функции «больше», следующая форма пригодна и для «меньше», «больше или равно», «меньше или равно», «не равно», «равно». Что касается «равно», то возможно сведение этой операции, к

аналогичной со сложением, путем ввода специальной функции, назовем ее F. Во внутреннем представлении Lispa создадим специальный атом, недоступный для использования пользователем, назовем его NAN (not a number – не число), по аналогии с общепринятым сокращением. Основной особенностью этого атома является то, что:

$$F(A,B)=\begin{cases} A, A=B \\ NAN, A=NAN \\ NAN, B=NAN \\ NAN, A\neq B \end{cases}$$

Учитывая, что NAN не число, выражения, приведенные на рис. 5 и 6, эквивалентны:

```
(> X1 X2 X3 X4 X5 X6 X7 X8)

(numberp
    (F
        (F
            (F X1 X2)
            (F X3 X4)
        )
        (F
            (F X5 X6)
            (F X7 X8)
        )
    )
)
```

Рисунок 6 – Альтернативная форма записи предиката “убывают”

Показанный на рис. 6 эквивалент вызова функции “>” не совсем корректен, так как на рис. 5 значение X_2 он вычисляет дважды. В общем случае это важно, так как вычисление аргумента может являться трудоемкой задачей. Возникшую проблему можно устранить с помощью *lambda*-вызова:

```
(> X1 X2 X3 X4 X5 X6 X7 X8)

((lambda ((> A1 A2 A3 A4 A5 A6 A7 A8)
(and
    (and
        (and
            (> 'A1 'A2)
            (> 'A2 'A3))
        (and
            (> 'A3 'A4)
            (> 'A4 'A5))
    )
    (and
        (and
            (> 'A5 'A6)
            (> 'A6 'A7))
        (> 'A7 'A8)
    )
) X1 X2 X3 X4 X5 X6 X7 X8)
```

Рисунок 7 – Обобщенная форма записи предиката “убывают”

Описанные выше программы на языке Lisp при последовательном выполнении эквивалентны по результату. Однако эти преобразования делают возможным параллельное выполнение программ при условии независимости аргументов X_i. Используя транслятор языка как инструмент для анализа программ, можно подсчитать получаемый выигрыш во времени. Для этого введем время операции. В процессе трансляции *S*-выражений помимо непосредственного реального времени работы программы мы можем отдельно фиксировать и виртуальное время, которое задаем сами. Пусть время на операцию сложения (вычитания) – 1 единица (например, секунда). Практически это время гораздо меньше, но так как нас интересует параллельное вычисление этой операции, то остальными операциями мы можем пренебречь, т.е. принять их время равным нулю.

Для примера рассмотрим показанное на рис. 4 вычитание. На одном процессоре для вычисления разности девяти чисел потребуется 8 секунд: 1 секунда на вычитание и 7 секунд на сложение. Если же процессоров 2, то первые 6 сложений могут быть выполнены параллельно – по 3 на каждый процессор. Оставшееся сложение и вычитание выполняется последовательно и займет 2 секунды. Таким образом, вычитание произойдет за 3+2 – за 5 секунд. Другой процессор будет работать только 3 секунды и 2 секунды простаивать. Если же взять 4 процессора, то первые 4 сложения будут выполнены параллельно, следующие 2 сложения тоже, последнее сложение и вычитание будет выполнено последовательно. Время, за которое будет получен результат – 4 секунды. Не сложно убедиться, что 4 процессора для решения данной задачи – много, а дальнейшее увеличение количества процессоров при таком размере задачи не улучшит результат. За 4 секунды 4 процессора могли бы сделать 16 операций, а сделали только 8.

5.2 Результаты экспериментального моделирования параллельного выполнения Lisp-программ

Были выполнены эксперименты с функциональными программами, реализующими алгоритмы:

- сложение двух векторов;
- нахождение суммы последовательности чисел;
- сортировка методом слияния.

При моделировании время, необходимое на выполнение одной операции (сложения и сравнения соответственно) на виртуальном

процессоре, было принято равным одной единице (секунде). Полученные результаты моделирования приведены в таблице 4.

Таблица 4 – Значения времени на эксперимент (в секундах)

N	Сложение векторов	Сумма (256 элементов)	Сумма (4096 элементов)	Сортировка (256 элементов)	Сортировка (4096 элементов)
2	256	255	4095	1538	45057
3	128	128	2048	896	26623
4	86	103	1678	794	21709
5	64	65	1026	638	15359
6	52	60	930	738	17162
7	43	52	812	620	14166
8	37	49	861	670	14083
9	32	34	514	540	11262
10	29	37	497	620	14447
11	26	33	490	684	13329
12	24	31	435	724	12375
13	22	31	384	542	10505
14	21	30	393	686	12623
15	19	30	411	562	11023
16	18	28	366	650	12444
17	16	19	259	506	9469
18	16	24	277	528	12062

5.2.1 Сложение векторов

Эксперимент №1: вычисление суммы двух векторов размера 256. Зависимость времени выполнения от количества процессоров показана на рис. 8 (кривая показывает идеальный вариант времени выполнения).

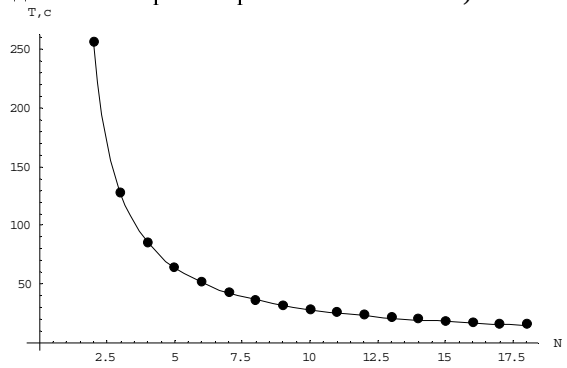

Рисунок 8 – Зависимость времени выполнения сложения векторов от количества процессоров

Экспериментальные данные полностью соответствуют ожиданиям от выполнения независимых задач, что свидетельствует о хорошем распределении задач по процессорам.

Ускорение времени выполнения в зависимости от количества процессоров представлено на рис. 9 (прямая показывает идеальный случай). На этом графике видны случаи отклонения результатов от ожидаемых. Это связано с тем, что число процессоров не кратно числу задач, в результате в конце задач не хватает для того, чтобы занять все процессоры и часть из них простаивает.

5.4.2 Вычисление суммы

Эксперимент №2: моделирование вычисления суммы 256 чисел. Зависимость времени выполнения от количества процессоров показана на рис. 10 (кривая показывает идеальный вариант времени выполнения).

Из рисунка можно определить следующую особенность – время выполнения на девяти процессорах меньше, чем на десяти, а на семнадцати процессорах меньше, чем на восемнадцати. Одной причиной хорошего времени для девяти и семнадцати процессоров является то, что задача размера 256 кратна числу рабочих процессоров, так как один процессор занят только распределением задач, то при девяти процессорах восемь рабочих (256 кратно 8), а при семнадцати процессорах 16 рабочих (256 кратно 16). Однако данная связь не объясняет тот факт, что при большем количестве процессоров суммарное время выполнения задачи увеличилось.

Протоколирование работы программы показало, что простои процессоров связаны с несовершенным распределением задач. Каждый рабочий процессор аккумулирует в себе приостановленные задачи. При получении необходимых результатов, выполнение этих задач продолжается. При передаче задач процессору не учитывается состояние его

отложенных задач, а также не учитываются его затраты на продолжение выполнения. Получив все необходимые результаты, процессор немедленно продолжает выполнение отложенной задачи. Простой происходит из-за того, что один процессор получил больше отложенных задач, чем остальные. При передаче задачи не учитывается, будет ли она отложена или же выполнится сразу, так как в общем случае это нельзя сказать заранее.

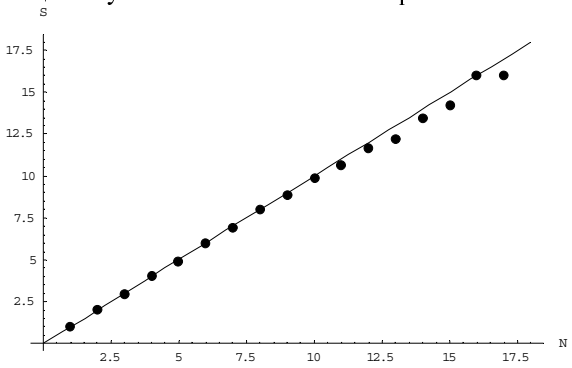

Рисунок 9 – Зависимость ускорения сложения векторов от количества рабочих процессоров

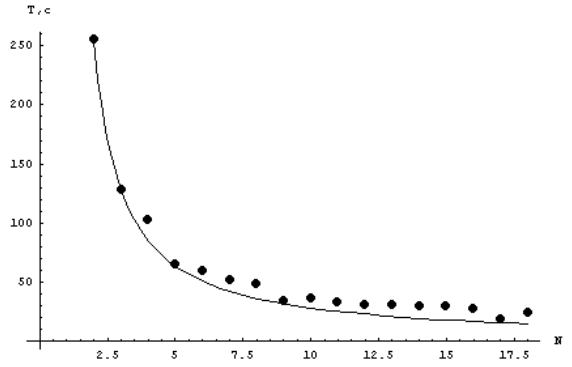

Рисунок 10 – Зависимость времени вычисления суммы 256 чисел от количества процессоров

При условии однородности выполняемых задач вполне оправданным является модификация алгоритма распределения задач между процессорами, которая будет учитывать количество отложенных процессором задач. На практике это легко устанавливается простым счетчиком, который увеличивается, когда процессору посылается задача и уменьшается, когда от процессора приходит результат. Данный счетчик будет показывать число отложенных задач процессора. При неоднородных задачах оценка времени, необходимого на выполнение продолжений требует значительного усложнения.

Ускорение времени выполнения в зависимости от количества процессоров представлено на рис. 11 (прямая показывает идеальный случай).

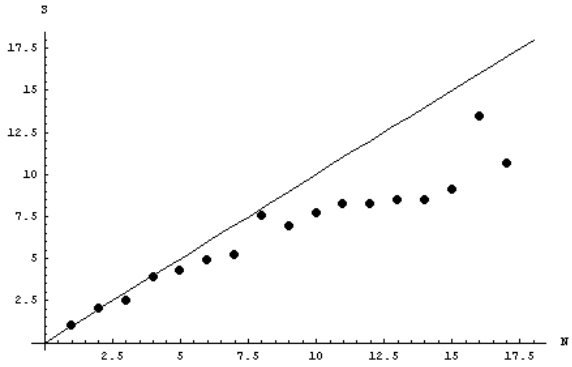

Рисунок 11 – Зависимость ускорения вычисления суммы 256 чисел от количества рабочих процессоров

Эксперимент №3: моделирование вычисления суммы 4096 чисел. Зависимость времени выполнения от количества процессоров и ускорение показана на рис. 12-13.

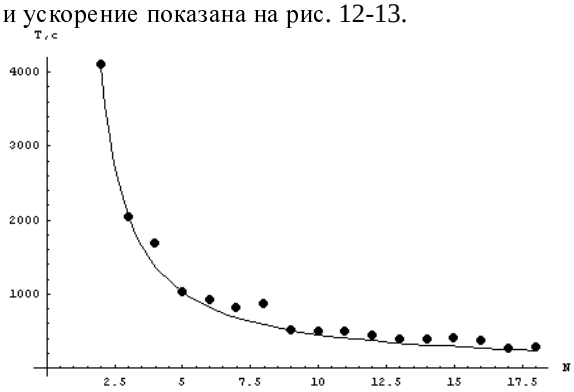

Рисунок 12 – Зависимость времени вычисления суммы 4096 чисел от количества процессоров

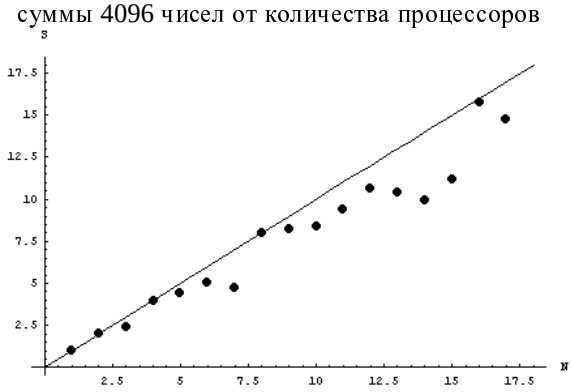

Рисунок 13 – Зависимость ускорения вычисления суммы 4096 чисел от количества рабочих процессоров

На рис. 13 явно выражены случаи кратности размеру задачи. Как видно, в остальных случаях увеличение производительности менее эффективно. Исходя из этого графика видно, что эффект рабочих процессоров в количестве от девяти до пятнадцати близок к показателю для восьми рабочих процессоров.

При увеличении размерности задачи общий вид графика улучшился, что свидетельствует о хорошей масштабируемости. Ускорения при количестве процессоров 2, 4, 8, 16 улучшились и приблизились к идеальному варианту, хотя результаты для 6, 13, 14, 17 рабочих процессоров мешают зависимости быть монотонно возрастающей.

5.4.3 Сортировка методом слияния

Эксперимент №4: моделирование сортировки по возрастанию 256 чисел. Для проведения эксперимента выбран алгоритм восходящей сортировки слиянием. Время выполнения такой сортировки пропорционально $N \log N$, и она нечувствительна к организации входных данных. Основным недостатком этого метода является дополнительное пространство памяти, однако при списочном представлении сами объекты не перемещаются, меняются лишь списочные ячейки, задающие порядок элементов.

Хотя сам алгоритм не чувствителен к организации входных данных – механизм слияния двух упорядоченных списков одного и того же размера в общем случае может приводить к различному количеству сравнений. Покажем это на примере:

А:	Б:
1 2 3 4 5	1 3 5 7 9
6 7 8 9 10	2 4 6 8 10

Рисунок 14 – Различные варианты слияния списков

Вариант А требует всего 5 операций сравнения, после чего один список станет пустым, а второй список уже упорядочен. Вариант Б требует 9 сравнений. В общем случае: пусть есть два списка M и N для слияния, в них n и m элементов соответственно, $n > 0, m > 0, n \geq m$. Для того, чтобы выбрать весь список M потребуется m операций сравнения. Чтобы выбрать весь список N потребуется n операций сравнения, тогда минимальным количеством сравнений будет m, так как $n \geq m$. В худшем случае можно выбрать все элементы списков кроме одного за $n+m-1$ операций сравнения. В эксперименте мы рассмотрим два случая – самый лучший и самый худший. Стоит отметить, что слияние двух больших списков будет производиться в самом конце работы алгоритма, когда будет происходить слияние двух упорядоченных половин исходного списка для сортировки.

На практике оказалось, что лучший случай отличается от худшего примерно в два раза. Это связано с тем, что в данном эксперименте слияние производится для списков одинаковой длины n, что дает n сравнений в лучшем случае и $2n-1$ в худшем. Получим соответствующие графики для времени выполнения и ускорения (рис. 15-16).

Получаемое ускорение не оправдывает аппаратные затраты.

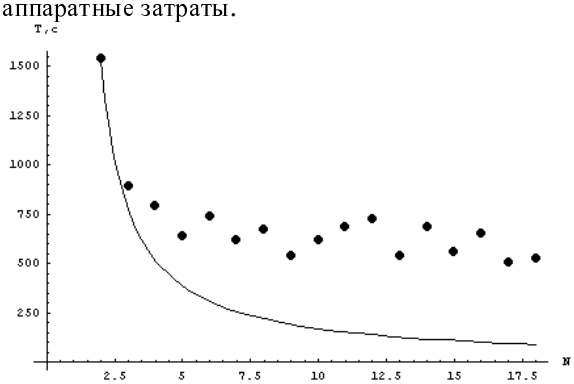

Рисунок 15 – Зависимость времени выполнения сортировки 256 чисел методом слияния от количества процессоров

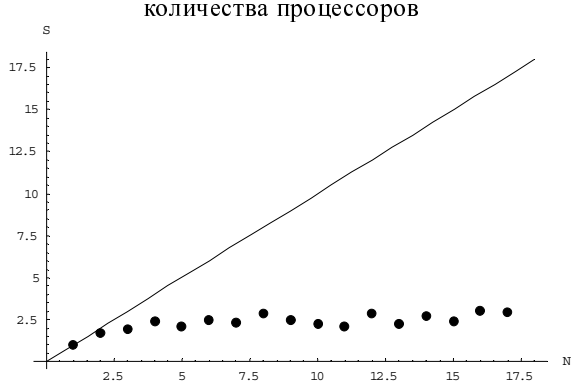

Рисунок 16 – Зависимость ускорения времени выполнения сортировки 256 чисел методом слияния от количества рабочих процессоров

Эксперимент №5: увеличим размер задачи сортировки с 256 до 4096 и проведем повторные измерения. Соответствующие графики для времени выполнения и ускорения приведены на рис. 17-18.

Операция слияния обычно проходит по спискам от начала, выбирает элемент, записывает его в начало списка результата и двигается по спискам к концу. С другой стороны аналогичного результата можно добиться, если идти от концов списков к началу и записывать результат в конец результирующего списка. Теоретически можно организовать проходы от начала списков и от концов одновременно. Это должно ускорить процесс слияния, однако в данной работе это не применяется, ввиду того, что в языке Lisp списки односвязные. Кроме того, в лучшем случае это позволяет ускорить решение задачи на двух процессорах в два раза, но дальнейшее распараллеливание на большее количество процессоров затруднительно.

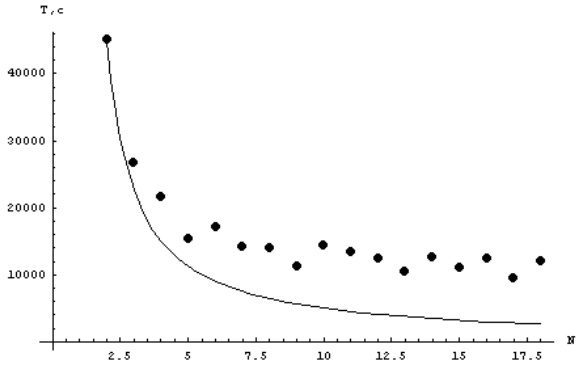

Рисунок 17 - Зависимость времени выполнения сортировки 4096 чисел методом слияния от количества процессоров

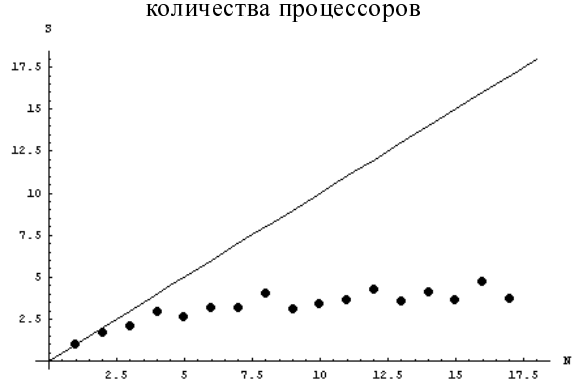

Рисунок 18 – Зависимость ускорения времени выполнения сортировки 4096 чисел методом слияния от количества рабочих процессоров

Хотя эффективность для задачи из 4096 чисел немного повысилась, все еще задача сортировки слиянием плохо распараллеливается; ускорение на двух-четырех процессорах еще имеет смысл делать, но большее количество процессоров дает лишь незначительный выигрыш во времени.

Выводы

Проведенные эксперименты показали работоспособность созданного транслятора с полной прозрачностью параллелизма для выполнения последовательных программ на примере сложения векторов, вычисления суммы и сортировки методом слияния. Полученные данные свидетельствуют об увеличении производительности для подобного рода задач. Как и следовало ожидать – множество независимых операций лучше всего параллельно выполняется, как показано экспериментом со сложением векторов. На основании экспериментов со сложением векторов и вычислением суммы можно сделать выводы для комплексных операций, например для умножения матриц. Умножение матриц можно рассматривать как множество независимых операций умножения и вычисления суммы. К подобным задачам также относятся моделирование нейросетей и вычисление скалярного произведения. Несмотря на положительные результаты для основных задач, стоит отметить, что многие задачи не способны рационально использовать большое количество процессоров, например сортировка слиянием плохо распараллеливается, и при увеличении числа процессоров результат не улучшается. Но даже для сортировки слиянием использование двух или четырех процессоров дает хорошее увеличение производительности. Стоит также отметить, что при большом количестве малых независимых задач, как показано на примере со сложением векторов, процессоры практически не простаивают, что делает подобные задачи очень полезными для параллельного выполнения одновременно с другими. Таким образом, при выполнении сортировки методом слияния вместе со сложением векторов, сложение будет занимать простаивающие процессоры, что позволит ликвидировать простои и увеличить эффективность выполнения программы в целом. Результаты параллельного выполнения на двух и четырех рабочих процессорах для всех проведенных экспериментов являются особенно полезными сегодня, когда рынок заполнен процессорами с двумя и четырьмя ядрами.

Результаты экспериментов с разработанным транслятором могут быть использованы в качестве дополнительной информации при написании параллельных трансляторов.

Литература

1. Фаулер М. Рефакторинг: Улучшение существующего кода.- СПб: Символ, 2003.
2. Дацун Н.Н., Савков К.Г. Исследование возможностей параллельной трансляции функциональных программ/ Наукові праці Донецького національного технічного університету. Серія “Інформатика, кібернетика і обчислювальна техніка” (ІКОТ-2008). Випуск 9 (132) – Донецьк: ДонНТУ. – 2008. – с.67-78.
3. McCarthy J. Lisp 1.5 Programmer’s Manual. – Cambridge: MIT, 1985. – 106p.
4. 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.
5. Goldman R., Gabriel R.P. Qlisp: Parallel processing in Lisp, Lucid Inc., 1993. – 24 p.
6. Feng M.D., Wong W.F., Yuen C.K. Compiling parallel Lisp for a shared memory multiprocessor. National University of Singapore, 1994. – 27p.
7. Jagannathan S. TS/Scheme: Distributed Data Structures in Lisp, Lisp and symbolic computation: An International Journal, 7, 1994. – pp. 283-305.
8. Yuasa Taiichi, Hagiya Masami. Kyoto Common Lisp Report, Kyoto University, 1985. – 87p.

Поступила в редколлегию 04.03.200