Федеральное государственное бюджетное образовательное учреждение
высшего образования «Сибирский государственный университет
телекоммуникаций и информатики» (СибГУТИ)

На правах рукописи

Кулагин Иван Иванович

Средства архитектурно-ориентированной оптимизации выполнения параллельных программ для вычислительных систем с многоуровневым параллелизмом

Специальность 05.13.15 —
«Вычислительные машины, комплексы и компьютерные сети»

Диссертация на соискание учёной степени
кандидата технических наук

Научный руководитель:
доктор технических наук, доцент
Курносов Михаил Георгиевич

Новосибирск — 2017

Оглавление

Стр.

Стр.

Стр.

Введение

Актуальность темы исследования и степень ее разработанности. Архитектура современных высокопроизводительных вычислительных систем (ВС) характеризуется многоуровневым параллелизмом – на множестве иерархических уровней системы реализуются различные формы параллельной обработки информации. На уровне множества вычислительных узлов (элементарных машин, ЭМ) реализуется параллелизм процессов (передача сообщений); на уровне многопроцессорного вычислительного узла с общей памятью – параллелизм потоков; на уровне суперскалярного процессорного ядра – параллелизм команд; на уровне векторных арифметико-логических устройств – параллелизм данных (SIMD-обработка). Развитие отечественных и зарубежных ВС идет по пути наращивания степени параллелизма на всех функциональных уровнях. Организация эффективного функционирования современных и перспективных ВС и достижение параллельными программами производительности близкой к пиковой требуют учета в системном инструментарии организации функционирования ВС форм параллелизма всех функциональных уровней.

Значительный вклад в развитие теории и практики вычислительных систем и средств организации их функционирования внесли выдающиеся ученые, среди которых В.С. Бурцев, В.А. Вальковский, А.П. Ершов, В.В. Воеводин, Э.В. Евреинов, А.В. Забродин, В.П. Иванников, М.Б. Игнатьев, А.В. Каляев, И.А. Каляев, Ю.Г. Косарев, В.В. Корнеев, Л.Н. Королев, А.О. Лацис, С.А. Лебедев, В.К. Левин, И.И. Левин, Г.И. Марчук, В.А. Мельников, Н.Н. Миренков, Д.А. Поспелов, И.В. Поттосин, И.В. Прангишвили, Д.В. Пузанков, Г.Г. Рябов, В.Б. Смолов, А.Н. Томилин, В.Г. Хорошевский, Б.Н. Четверушкин, Н.Н. Яненко, W. Carlson, B. Chamberlain, D. Culler, J. Dongarra, D. Grove, M. Herlihy, L. Lamport, T. Knight, T. Riegel, N. Shavit.

Анализ тенденций развития мощнейших высокопроизводительных ВС мира показывает, что развитие архитектуры ВС идет по пути увеличения степени параллелизма на всех уровнях иерархии системы: числа процессоров в ЭМ, числа ядер в процессорах, а также количества параллельно работающих скалярных и векторных АЛУ в процессорных ядрах. Эффективное использование всех ресурсов ВС, достижение (суб)пиковой устойчивой

производительности требуют разработки инструментальных средств (математических моделей, методов и алгоритмов) и параллельных программ, учитывающих многоуровневый параллелизм ВС.

С развитием мультиархитектуры ВС активно развиваются высокоуровневые средства параллельного программирования, ориентированные на сокращение трудозатрат при разработке параллельных программ (high productivity computing). В частности, языки параллельного программирования Unified Parallel C, Coarray Fortran, IBM X10, Cray Chapel, DVM, ParJava, T-система. Значительная часть таких языков реализует модель разделенного глобального адресного пространства (partitioned global address space, PGAS), которая позволяет абстрагироваться от явного использования в коде программ низкоуровневых операций передачи сообщений при обращении к объектам в памяти удаленных процессов (элементарных машин). Масштабируемость параллельных PGAS-программ на ВС во многом определяется эффективностью алгоритмов, реализованных в PGAS-компиляторах и их runtime-системах. В частности, остро стоят задачи, связанные с оптимизацией времени доступа в PGAS-программах к совместно используемым структурам данных, находящимся в памяти удаленных ЭМ системы. Решение этих задач связано с разработкой методов оптимизации информационных обменов в PGAS-языках на уровне вычислительной системы.

На уровне отдельного многопроцессорного вычислительного узла с общей памятью требуют своего решения задачи сокращения времени доступа ветвей параллельных программ к разделяемым структурам данных. Классические методы синхронизации типа «мьютекс» (взаимное исключение) и «семафор» подразумевают блокирование одновременного выполнения участка кода программы множеством потоков, а не защиту совместно используемой области памяти. Последнее, для значительного класса задач, приводит к образованию очередей ожидания доступа в критическую секцию и снижает масштабируемость программ. Развиваются альтернативные подходы: алгоритмы, свободные от блокировок (lock-free algorithms), и программная транзакционная память (ПТП, software transactional memory, STM). Модель транзакционной памяти получила как аппаратурную реализацию в процессорах IBM BlueGene/Q PowerPC, Intel Haswell (Intel TSX), Oracle Rock (SPARC v9), так и программную реализацию в современных компиляторах и runtime-библиотеках: GCC TM, LazySTM, TinySTM, DTMC, RSTM, STMX,

STM Monad. В программных реализациях транзакционной памяти актуальными являются задачи разработки методов и структур данных обнаружения конкурентного доступа потоков программы к разделяемым объектам в памяти.

Эффективное использование ресурсов ВС также требует учета параллелизма на уровнях суперскалярного ядра процессора и векторных арифметико-логических устройств. В связи с активным развитием наборов векторных SIMD-инструкций (Intel AVX, IBM AltiVec, ARM NEON/SVE, MIPS MSA) новую актуальность получили задачи (полу)автоматической векторизации циклов в параллельных программах на процессорах с короткими векторными регистрами.

На основании вышесказанного можно утверждать, что разработка средств архитектурно-ориентированной оптимизации выполнения параллельных программ для вычислительных систем с многоуровневым параллелизмом является актуальной и значимой темой.

Цель работы и задачи исследования. Целью работы является разработка и исследование средств архитектурно-ориентированной оптимизации выполнения параллельных программ для ВС с многоуровневым параллелизмом. В соответствии с целью определены следующие задачи исследования.

1. Для ВС с многоуровневым параллелизмом разработать средства оптимизации циклического доступа к информационным массивам в параллельных программах на базе модели разделенного глобального адресного пространства.

2. Для многопроцессорных вычислительных узлов с общей памятью разработать и исследовать методы сокращения времени выполнения транзакционных секций многопоточных программ в модели программной транзакционной памяти.

3. Разработать средства анализа эффективности векторизации циклов на архитектурах процессоров с короткими векторными регистрами.

4. Разработать пакет программ оптимизации функционирования ВС и использования их многоуровневого параллелизма для решения параллельных задач.

Научная новизна полученных результатов определяется учетом в созданных методах и алгоритмах форм параллелизма основных функциональных уровней ВС и динамических характеристик параллельных задач.

1. Оригинальность созданного алгоритма *Array Preload* трансформации конструкций циклической передачи потока управления подчиненным элементарным машинам заключается в возможности его применения к PGAS-программам, в которых на этапе компиляции неизвестно множество используемых элементов массивов. В отличие от известных, время выполнения кода, формируемого алгоритмом, не зависит от количества итераций циклов.

2. Новизна метода сокращения числа ложных конфликтов в многопоточных программах на базе программной транзакционной памяти заключается в (суб)оптимальной настройке таблиц обнаружения конфликтов с учетом шаблона доступа к памяти из параллельных программ.

3. В отличие от известных работ полученные классы трудно векторизуемых циклов из тестового набора ETSVC сформированы с учетом микроархитектуры современных наборов команд Intel AVX и AVX-512.

4. Созданный инструментарий анализа эффективности использования микроархитектурных возможностей ядер суперскалярных процессоров ВС позволяет анализировать загрузку суперскалярного конвейера архитектуры Intel 64 потоком инструкций с точностью до нескольких машинных команд.

Теоретическая и практическая значимость работы состоит в том, что разработанные в диссертации методы и алгоритмы реализованы в виде системного программного обеспечения анализа и оптимизации использования в параллельных программах многоуровневого параллелизма ВС.

1. Созданный алгоритм *Array Preload* трансформации конструкций циклической передачи потока управления подчиненным элементарным машинам реализован в виде расширения компилятора языка IBM X10. По сравнению со стандартными алгоритмами IBM X10 предложенный позволяет на кластерных ВС с сетями связи Gigabit Ethernet и InfiniBand QDR сократить время выполнения циклического доступа к элементам массивов в 1.2–2.5 раз.

2. Разработанный метод сокращения числа ложных конфликтов в многопоточных программах на базе программной транзакционной памяти реализован в расширении компилятора GCC. Реализация включает модуль инструментации кода целевой программы и модуль настройки параметров runtime-системы реализации ПТП по результатам профилирования программы. Использование предложенного метода позволяет сократить время

выполнения параллельных программ в среднем на 20%, что экспериментально показано на тестовых программах из пакета STAMP.

3. Создан инструментарий анализа эффективности использования микроархитектурных возможностей ядер суперскалярных процессоров ВС. Предложенные средства позволяют анализировать загрузку суперскалярного конвейера архитектуры Intel 64 потоком инструкций с точностью до нескольких команд ассемблера.

4. На процессорах с поддержкой наборов векторных инструкций Intel AVX и AVX-512 экспериментально установлены классы трудно векторизуемых циклов из тестового набора ETSVC. Построенное подмножество циклов составляет базисный набор для анализа эффективности ядер автовекторизаторов оптимизирующих компиляторов для векторных процессоров класса «регистр-регистр».

5. Программные средства внедрены в действующую мультикластерную ВС Центра параллельных вычислительных технологий СибГУТИ и Лаборатории вычислительных систем Института физики полупроводников им. А.В. Ржанова СО РАН (ИФП СО РАН).

Основные этапы исследования выполнены в ходе осуществления работ по проектам Российского фонда фундаментальных исследований №№ 15-07-00653, 15-37-20113; Совета Президента РФ по поддержке ведущих научных школ № НШ-2175.2012.9 (руководитель – чл.-корр. РАН В.Г. Хорошевский); грантов Новосибирской области для молодых ученых, стипендии Президента РФ для аспирантов.

Получено три свидетельства о государственной регистрации программ для ЭВМ. Результаты работы внедрены в учебный процесс СибГУТИ, в систему параллельного мультипрограммирования пространственно-распределенной ВС, что подтверждается соответствующими актами.

Методология и методы исследования. Для решения поставленных задач использовались методы теории вычислительных систем, теории алгоритмов, теории графов. Экспериментальные исследования осуществлялись путем моделирования на вычислительных кластерах с многоуровневым параллелизмом. Работа основана на результатах ведущей научной школы в области анализа и организации функционирования большемасштабных ВС (руководитель – чл.-корр. РАН В.Г. Хорошевский).

Положения и результаты, выносимые на защиту.

1. Алгоритм *Array Preload* трансформации конструкций циклической передачи потока управления подчиненным элементарным машинам, сокращающий время выполнения программ за счет опережающего копирования информационных массивов. В отличие от известных разработанный метод применим к PGAS-программам, в которых на этапе компиляции неизвестно множество используемых элементов массивов.

2. Программная реализация алгоритма *Array Preload*, обеспечивающая на кластерных ВС с сетями связи Gigabit Ethernet и InfiniBand QDR сокращение в параллельных программах на языке IBM X10 времени выполнения циклического доступа к элементам массивов в 1.2–2.5 раз.

3. Метод сокращения числа ложных конфликтов в многопоточных программах на базе программной транзакционной памяти, основанный на подборе (суб)оптимальных параметров таблиц обнаружения конфликтов по результатам предварительного профилирования параллельных программ.

4. Программная реализация метода сокращения числа ложных конфликтов в расширении компилятора GCC, обеспечивающая сокращение времени выполнения транзакционных секций в C/C++-программах в среднем на 20%.

5. Экспериментально построенные на архитектурах с поддержкой наборов инструкций Intel AVX/AVX-512 классы трудно векторизуемых циклов из тестового набора ETSVC. Полученное подмножество циклов составляет базисный набор для анализа эффективности ядер автовекторизаторов оптимизирующих компиляторов для векторных процессоров класса «регистр-регистр».

6. Инструментарий анализа эффективности использования микроархитектурных возможностей ядер суперскалярных процессоров ВС, позволяющий анализировать загрузку суперскалярного конвейера архитектуры Intel 64 потоком инструкций с точностью до нескольких команд ассемблера.

7. Инструментарий параллельного мультипрограммирования кластерной ВС, расширенный созданными автором пакетами оптимизации использования многоуровневого параллелизма в параллельных программах.

Личный вклад. Выносимые на защиту результаты получены соискателем лично. В совместных работах постановки задач и разработка методов их решения осуществлялись при непосредственном участии соискателя.

Степень достоверности и апробация результатов подтверждаются проведенными экспериментами и моделированием, согласованностью с данными, имеющимися в отечественной и зарубежной литературе, а также прошедшими экспертизами работы при получении грантов.

Основные результаты работы докладывались и обсуждались на международных, всероссийских и региональных научных конференциях, в их числе: международные конференции «Parallel Computing Technologies» (PaCT-2015, Petrozavodsk), «Параллельные вычислительные технологии» (ПаВТ-2016, г. Архангельск), «Суперкомпьютерные технологии: разработка, программирование, применение» (СКТ-2014, СКТ-2016, г. Геленджик), «Открытая конференция по компиляторным технологиям» (2015 г., г. Москва); российские конференции: «Актуальные проблемы вычислительной и прикладной математики» (АПВПМ-2015, г. Новосибирск), «Новые информационные технологии в исследовании сложных структур» (ICAM-2014, г. Томск), Сибирская конференция по параллельным и высокопроизводительным вычислениям (2015 г., г. Томск).

Публикации. По теме диссертации опубликовано 23 работы. Из них 4 – в журналах из перечня ВАК РФ; 2 – в изданиях, индексируемых Scopus и Web of Science. Получено 3 свидетельства о государственной регистрации программ для ЭВМ.

Глава 1. ВС с многоуровневым параллелизмом

1.1 Понятие о ВС с многоуровневым параллелизмом

1.1.1 Модель коллектива вычислителей

При проектировании вычислительных средств обработки информации возможно два пути следования [65–68, 77, 104]:

- построение электронных вычислительных машин (ЭВМ), моделирующих процесс выполнения алгоритма одиночным человеком-вычислителем;
- создание вычислительных систем, моделирующих процесс выполнения алгоритма коллективом вычислителей.

ЭВМ представляет собой аппаратно-программный комплекс, построенный на модели вычислителя. *Модель вычислителя* сформулирована в [68] и наиболее полно по отношению к вычислительным машинам отражена в [66]. Для этой модели характерны фиксированная структура, неоднородность связей и функциональных элементов. Развитие ЭВМ движется по пути увеличения числа арифметико-логических устройств (АЛУ), способных одновременно выполнять команды (параллелизм уровня команд, Instruction Level Parallelism – ILP). На ряду с наращиванием степени параллелизма уровня команд в архитектуре ЭВМ развиваются возможности по ускорению вычислений как для одного АЛУ, так и для всех имеющихся, например, внеочередное выполнение команд (Out-of-order execution), предсказание условных переходов и др. Однако, использование модели вычислителя для построения высокопроизводительных вычислительных средств ограничено теоретическими и техническими пределами скорости выполнения операций (возможностями элементной базы и фундаментальными физическими законами) [104].

Под *коллективом вычислителей* подразумевается совокупность вычислительных машин, программно-аппаратурным способом настраиваемая на

решение общей задачи. В основу коллектива вычислителей положены три основополагающих архитектурных принципа:

- *параллелизм* (parallelism, concurrency) при обработке информации;
- *программируемость структуры* (programmability, adoptability) – настраиваемости структуры сети связей между вычислителями, достигаемая программными средствами;
- *однородность конструкции* (homogeneity), однородность вычислителей и структуры.

Модель коллектива вычислителей базируется на диалектическом отрицании принципов, лежащих в основе модели вычислителя. Параллелизм при обработке информации предполагает наличие алгоритма решения задач с помощью параллельных программ. Параллельная программа – это совокупность взаимодействующих вычислительных процессов, выполняющих одновременную (параллельную) работу над выделенными им данными.

Принцип программируемости структуры заключается в том, чтобы в коллективе вычислителей была заложена возможность автоматической (программной) настройки проблемно-ориентированных (виртуальных) конфигураций и их перенастройки в процессе функционирования с целью обеспечения адекватности структурам и параметрам решаемых задач и достижения эффективности при заданных условиях эксплуатации [66, 104, 105].

Конструктивная однородность модели коллектива вычислителей заключается в формировании его из совокупности одинаковых вычислителей, регулярно соединенных между собой.

В отличие от одиночного вычислителя реализующие данные принципы вычислительные средства обладают теоретически неограниченной производительностью, так как могут безгранично наращивать число вычислителей. Кроме того, по сравнению с ЭВМ построенное на модели коллектива вычислительное средство способно обеспечить заданный уровень надежности и живучести, т.е. возможностью функционировать при отказах элементов, а также допускает простой способ наращивания производительности [104].

Вычислительное средство, основанное на модели коллектива вычислителей, называется *вычислительной системой* (ВС) [66, 104].

Под *распределенной ВС* понимается совокупность взаимосвязанных и одновременно функционирующих унифицированных ЭВМ, которая способна не только реализовать (параллельный) процесс решения сложной задачи,

но и в процессе работы автоматически настраиваться и перестраиваться с целью достижения адекватности между своей структурно-функциональной организацией и структурой и характеристиками решаемой задачи [104, 105].

1.1.2 Классификация ВС

Согласно классификации, предложенной в 1966 году М.Дж. Флинном, архитектуры вычислительных средств делятся на четыре класса: SISD (Single Instruction stream / Single Data stream), SIMD (Single Instruction stream / Multiple Data stream), MISD (Multiple Instruction stream / Single Data stream), MIMD (Multiple Instruction stream / Multiple Data stream). В основе классификации лежит разделение архитектур вычислительных средств по количеству обрабатываемых ими потоков команд и данных.

К архитектурам класса SISD относится ЭВМ. Под потоком команд понимается любая их последовательность, поступающая для исполнения вычислительным средством (ЭВМ или процессором, в случае SISD-архитектуры). При выполнении команд потока требуются операнды (данные), следовательно, поток команд «порождает» поток данных [104].

Архитектуры MISD, SIMD, MIMD относятся к вычислительным системам. В этих архитектурах имеет место «множественность» потоков или (и) команд, или (и) данных. Множественность характеризуется количеством одновременно реализуемых потоков команд или (и) данных. Основные типы ВС приведены ниже.

Многопроцессорные ВС с общей памятью – системы с MIMD-архитектурой, которые состоят из множества процессоров и разделяемой (возможно секционированной) памяти; взаимодействие между процессорами и памятью осуществляется через коммутатор (общую шину и т.п.), а между процессорами – через память. К таким системам относятся машины на базе многоядерных процессоров, а также графические процессоры (например, GPU NVIDIA, AMD) и сопроцессоры семейства Intel Xeon Phi.

Распределенные ВС – мультипроцессорные ВС с MIMD-архитектурой, в которых нет единого ресурса (общей памяти). Основные компоненты

распределенной ВС (такие, как коммутатор, устройство управления, арифметико-логическое устройство или процессор, память) допускают представление в виде композиции из одинаковых элементов (локальных коммутаторов и устройств управления, локальных процессоров и модулей памяти) [66]. Это широкий класс систем, представителями которого являются вычислительные кластеры и проприетарные системы с распределенной памятью (например, Cray XK7 и IBM BlueGene/Q).

Вычислительные системы с программируемой структурой полностью основываются на модели коллектива вычислителей и являются композицией взаимосвязанных элементарных машин. Каждая ЭМ в своем составе обязательно имеет локальный коммутатор (ЛК), процессор и память, а также может иметь внешние устройства. Локальная память ЭМ предназначается для хранения и части данных, и, главное, ветви параллельной программы. Архитектура ВС с программируемой структурой относится к типу MIMD. Такие ВС по своим потенциальным архитектурным возможностям не уступают ни одному из перечисленных выше классов систем. Концепция вычислительных систем с программируемой структурой была сформулирована в Сибирском отделении АН СССР, первая система («Минск-222») была построена в 1965–1966 гг. [105].

Современные ВС не реализуют архитектуру одного конкретного класса (SIMD, MISD, MIMD), а являются мультиархитектурными. Функциональная структура таких систем имеет иерархическую организацию, а каждый уровень иерархии может относиться к любому из классов архитектур, либо также быть мультиархитектурным. Таким образом, иерархическая структура современных ВС порождает многоуровневый параллелизм. Эффективное программирование таких ВС требует применения широкого спектра технологий.

1.1.3 ВС с многоуровневым параллелизмом

Распределенные ВС имеют иерархическую структуру и обладают многоуровневым параллелизмом: параллелизм процессов на уровне вычислительных узлов, параллелизм потоков на уровне процессорных ядер,

параллелизм команд на уровне АЛУ и параллелизм данных на уровне векторных АЛУ. На уровне вычислительных узлов выполняется множество процессов, которые взаимодействуют между собой посредством передачи сообщений. Разработка параллельных программ на данном уровне ведется в моделях параллельного программирования, ориентированных на отсутствие общей памяти, например, модель передачи сообщений (Message Passing) либо модель разделенного глобального адресного пространства (Partitioned Global Address Space – PGAS).

Современные процессоры вычислительных систем являются многоядерными и предоставляют параллелизм на уровне вычислительных ядер. На этом уровне программная реализация параллельных алгоритмов ведется в модели многопоточного программирования. Взаимодействие потоков происходит через общую память.

Процессорные ядра реализуют суперскалярную архитектуру с конвейерным принципом выполнения команд [22, 28, 45, 76, 93]. В таких ядрах содержится множество одновременно функционирующих АЛУ, реализующих параллелизм команд. Кроме параллелизма уровня команд вычислительные ядра современных процессоров реализуют параллелизм уровня данных векторными АЛУ. Большинство процессоров предоставляют расширение архитектуры набора команд для использования векторных АЛУ (Intel: Streaming SIMD Extensions – SSE, SSE2, SSE3, SSE4, Advanced Vector Extensions – AVX, AVX2, AVX512; AMD: 3DNow!; ARM: Advanced SIMD – NEON; IBM: AltiVec). Высокая производительность процессора недостижима без реализации его архитектурой различных возможностей по ускорению вычислений [22, 28, 45], например, концепции внеочередного выполнения команд (Out-of-order execution), переименовывания регистров, обнаружения программных циклов, предсказания условных переходов и многого другого.

На Рисунке 1.1 представлены уровни параллелизма, реализуемые современными ВС, на примере системы Sunway Taihulight, которая занимает 1 позицию в списке высокопроизводительных ВС TOP500 (редакция от 21 июня 2017 года).

На пути к достижению эксафлопной производительности ВС происходит наращивание степени параллелизма каждого уровня. Для эффективного использования их возможностей необходима разработка инструментария

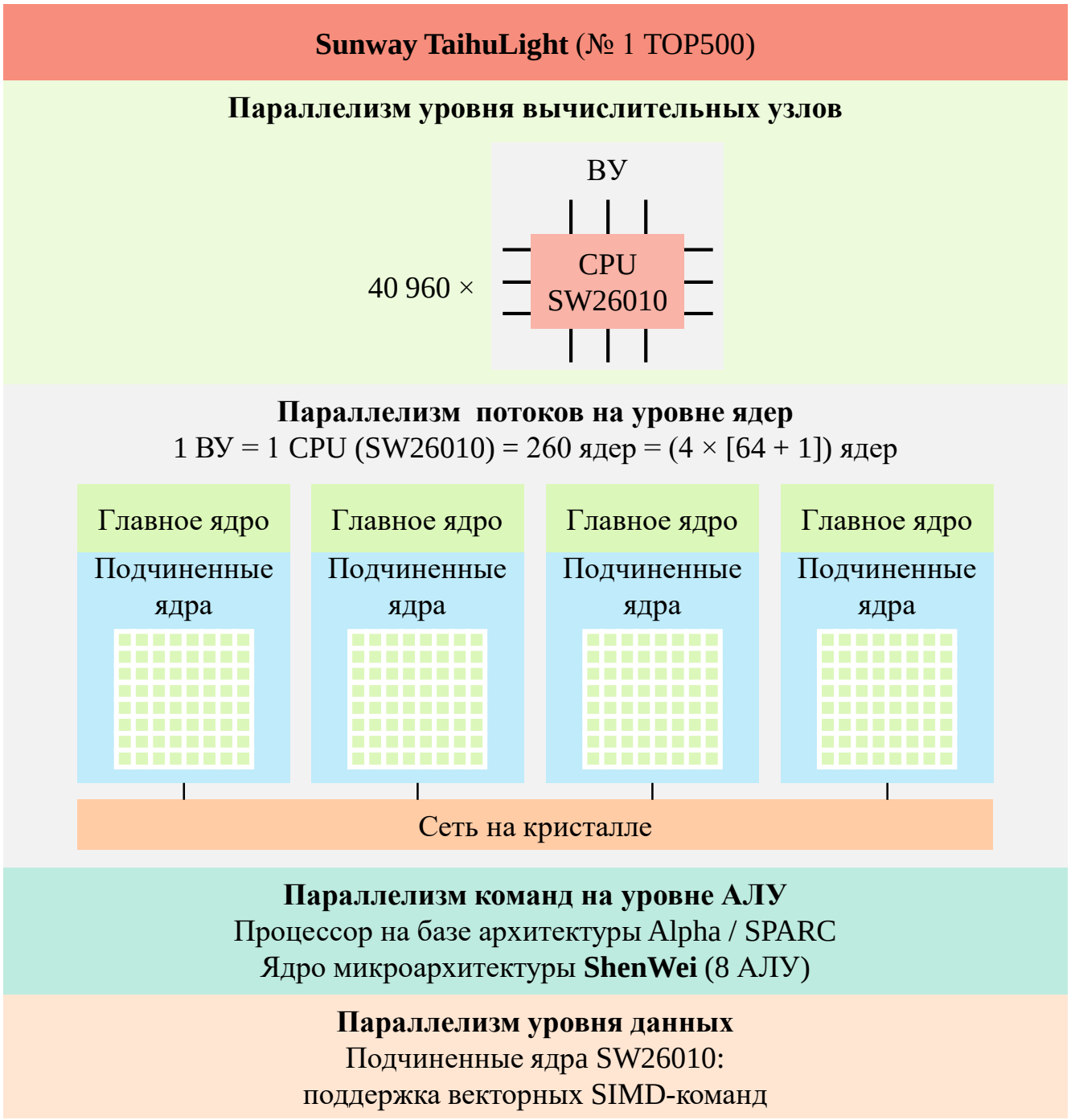


Рисунок 1.1 – Многоуровневый параллелизм ВС Sunway TaihuLight

архитектурно-ориентированной оптимизации выполнения параллельных программ.

1.2 Параллельные алгоритмы

Параллельный алгоритм (parallel algorithm) является фундаментальным понятием в теории вычислительных систем [62, 64, 66, 74, 91, 92, 94, 96–98, 103, 104, 106, 107].

Параллельный алгоритм – это описание процесса обработки информации, ориентированное на реализацию в коллективе вычислителей [104]. Такой алгоритм, в отличие от последовательного, предполагает одновременное выполнение нескольких операций в пределах одного шага вычислений и, как последовательный алгоритм, сохраняет зависимость последующих этапов от результатов предыдущих.

Параллельный алгоритм решения задачи составляет основу параллельной программы, которая, в свою очередь, влияет на алгоритм функционирования коллектива вычислителей. Запись параллельного алгоритма на языке программирования, доступном коллективу вычислителей, называют *параллельной программой*, а сам язык – параллельным. Параллельные алгоритмы и программы следует разрабатывать для тех задач, которые недоступны для решения на средствах, основанных на модели вычислителя. Эти задачи принято называть сложными или трудоемкими.

Качество параллельного алгоритма (или его эффективность) определяется методикой распараллеливания сложных задач. Выделяют два основных подхода к распараллеливанию задач: локальное и глобальное (крупноблочное) распараллеливание. Первый подход ориентирован на расщепление алгоритма решения сложной задачи на предельно простые блоки (операции или операторы) и требует выделения для каждого этапа вычислений максимально возможного количества одновременно выполняемых блоков. Он не приводит к параллельным алгоритмам, эффективно реализуемым коллективом вычислителей. Так как процесс такого распараллеливания весьма трудоемок, а получаемые параллельные алгоритмы характеризуются не только структурной неоднородностью, но и существенно разными объемами

операций на различных этапах вычислений. Последнее является серьезным препятствием на пути (автоматизации) распараллеливания и обеспечения эффективной эксплуатации ресурсов коллектива вычислителей. Локальное распараллеливание позволяет оценить предельные возможности коллектива вычислителей при решении сложных задач, получить предельные оценки по распараллеливанию сложных задач.

Второй подход ориентирован на разбиение сложной задачи на крупные блоки-подзадачи, между которыми существует слабая связность. Тогда в алгоритмах, построенных на основе *крупноблочного распараллеливания*, операции обмена между подзадачами будут составлять незначительную часть по сравнению с общим числом операций в каждой подзадаче. Такие подзадачи называют *ветвями параллельного алгоритма*, а соответствующие им программы – *ветвями параллельной программы* или *процессы параллельной программы*.

Структура обменов информацией между ветвями параллельной программы характеризуется *информационным графом*. Информационный граф параллельной программы (task graph) – это конечный граф, вершинам которого соответствуют параллельные ветви, а ребрам – обмены информацией между ветвями.

Разработка параллельного алгоритма ведется в рамках определенной *модели параллельных вычислений* (LogP, LogGP, Hockney и др.), которая в абстрактной форме отражает архитектуру целевой ВС.

Для разработки параллельных программ создано большое количество инструментальных средств, ориентированных на задействование параллелизма различных уровней системы. На уровне вычислительных узлов ВС с распределенной памятью разработка параллельных программ ведется преимущественно при помощи стандарта MPI (Message Passing Interface) и семейства библиотек SHMEM. Стандарт MPI по праву заслуживает место «ассемблера параллельных вычислений», однако, с его использованием связано возникновение ряда сложностей и неудобств, в первую очередь необходимость явного обращения к функциям коммуникационных операций и синхронизация ветвей MPI-программы. Поэтому для упрощения процесса разработки параллельных программ активно ведется развитие высокопродуктивных средств параллельного программирования, в частности,

языков, реализующих модель разделенного глобального адресного пространства (Partitioned Global Address Space – PGAS). К классу PGAS относятся такие языки, как IBM X10, Cray Chapel, Unified Parallel C, Coarray Fortran, ParJava, DVM.

Каждый вычислительный узел (ВУ) распределенной ВС по отдельности представляет собой многопроцессорную ВС с общей памятью (многопроцессорные SMP/NUMA-узлы). На этом уровне в основном применяются средства многопоточного программирования: POSIX Threads, OpenMP, Cilk/Cilk++, Intel Threading Building Blocks и др. Для создания параллельных программ для графических процессоров и решений на базе ПЛИС применяются специализированные языки и стандарты: NVIDIA CUDA, OpenCL, OpenACC, OpenMP 4.x, COLAMO [75].

На уровне вычислительных ядер процессора для задействования параллелизма команд и данных применяются автоматические (на этапе компиляции), полуавтоматические (при помощи специальных директив компилятора) и ручные техники оптимизации, а именно, векторизация кода, планирование команд и конвейеризация циклов.

Учитывая многоуровневую архитектуру современных ВС, создание эффективных параллельных программ невозможно без наличия инструментария (математических моделей, алгоритмов и программного обеспечения) архитектурно-ориентированной оптимизации выполнения параллельных программ на распределенных вычислительных системах.

1.3 Модели параллельных вычислений

Модели параллельных вычислений (*model of parallel computation*) небходимы для анализа эффективности параллельных алгоритмов и представляют собой математическую модель, которая описывает:

- архитектуру ВС;
- набор базовых операций управления параллельными процессами;
- методы оценки временных затрат на организацию параллельных вычислений.

Широкое распространение получили модель Hockney [49] и семейство моделей LogP (LogGP, PLogP, LogGPS, LogPC, LogfP) [25, 33, 36].

Модель Hockney [49] предполагает, что время отправки сообщения размером m между двумя узлами составляет $\alpha + \beta m$, где α – это задержка для каждого сообщения (латентность, latency, startup time), а β – время передачи одного байта сообщения или обратная величина пропускной способности сети. Стоит отметить, что перегрузка каналов связи не может быть промоделирована с помощью этой модели.

Модель LogP [25] описывает сеть в терминах верхней границы латентности (latency, delay) передачи сообщения одного процессора другому – L, накладных расходов (overhead) или промежутка времени, в течение которого процессор занят передачей или приемом сообщения – o, минимального интервала между сообщениями – g и количества процессоров – P, вовлеченных в коммуникационное взаимодействие. Время отправки сообщения между двумя процессорами в соответствии с моделью LogP равно $L+2o$. LogP предполагает, что только небольшие сообщения постоянного размера передаются между процессорами.

LogGP [36] представляет собой расширение модели LogP и в дополнение к ней допускает большие сообщения, вводя параметр G – побайтный интервал. Модель LogGP определяет время отправки сообщения размером m между двумя процессорами как $L + 2o + (m - 1)G$. В обеих моделях отправитель может инициировать новое сообщение по прошествии времени g.

Модель PLogP [33] является расширением модели LogP. Модель PLogP определена в терминах задержки – L, накладных расходов отправителя и получателя $o_s(m)$ и $o_r(m)$ соответственно, интервала между сообщениями $g(m)$ и количества процессоров – P, вовлеченных в коммуникационное взаимодействие. В этой модели накладные расходы отправителя и получателя и интервал между сообщениями зависят от размера сообщения. Понятие задержки и интервала в модели PlogP немного отличается от модели LogP/LogGP. Задержка в модели PLogP включает в себя все сопутствующие факторы, такие как копирование данных в и из сетевых интерфейсов, в дополнение ко времени передачи сообщения. Интервал в модели PLogP определен как минимальный временной интервал между последовательными передачами или приемами сообщения, подразумевая, что во все моменты времени $g(m) \geqslant o_s(m)$ и $g(m) \geqslant o_r(m)$. Время отправки сообщения размером

m между двумя процессорами в модели PLogP равно $L + g(m)$. Если $g(m)$ – линейная функция от размера m сообщения, и L исключает задержки отправителя, то модель PLogP идентична модели LogGP, которая различает накладные расходы отправителя и получателя.

1.4 Выводы

1. Современные вычислительные системы являются мультиархитектурными и реализуют многоуровневый параллелизм.
2. Для организации эффективного функционирования ВС и достижения близкой к пиковой производительности требуется максимально использовать возможности каждого уровня параллелизма, что недостижимо без инструментальных средств создания параллельных программ, учитывающих многоуровневый параллелизм ВС.
3. Актуальной задачей является разработка средств архитектурно-ориентированной оптимизации выполнения параллельных программ для вычислительных систем с многоуровневым параллелизмом.

Глава 2. Модель разделенного глобального адресного пространства

2.1 Описание модели PGAS

Модель разделенного глобального адресного пространства (partitioned global address space – PGAS) – это модель параллельного программирования, реализующая общее адресное пространство над распределенной памятью ВС [4, 15, 30, 32, 43, 51, 58, 59]. В модели PGAS доступ из любой элементарной машины (ЭМ) к глобальному адресному пространству осуществляется как к локальной памяти при помощи высокоуровневых операторов присваивания «=». То есть, модель PGAS позволяет создавать программы для распределенных ВС, используя семантику обращения к памяти из ВС с общей памятью. Достоинство данной модели в том, что, в отличие от модели передачи сообщений, в ней отсутствуют коммуникационные операции. Программа, разработанная в модели PGAS, называется *PGAS-программой*, а язык программирования, реализующий эту модель, – *PGAS-языком*.

На Рисунке 2.1 представлена модель разделенного глобального адресного пространства и ее отображение на распределенную ВС из n ЭМ. Ключевым понятием в модели PGAS является *область* (в терминах языка IBM X10 – *Place*), которая представляет абстракцию ЭМ с разделяемой памятью на базе многоядерного процессора. В рамках областей выполняются потоки (в терминах языка IBM X10 – *Activity*). Данные, над которыми потоки выполняют операции, могут храниться либо в памяти текущей области (*локальной памяти*), либо в памяти другой области (*удаленной памяти*). Память множества областей формирует разделенное глобальное адресное пространство. Доступ ко всем данным в глобальном адресном пространстве осуществляется одинаково без явного обращения в PGAS-программе к коммуникационным операциям вне зависимости от того, где они хранятся: в локальной памяти или удаленной. Каждая область закреплена за определенной ЭМ. Локальная память области является физической памятью ЭМ, а ее потоки выполняются процессорными ядрами соответствующей ЭМ.

Реализация модели PGAS определяется PGAS-языком, который предоставляет конструкции для работы с областями (Place) и распределенными

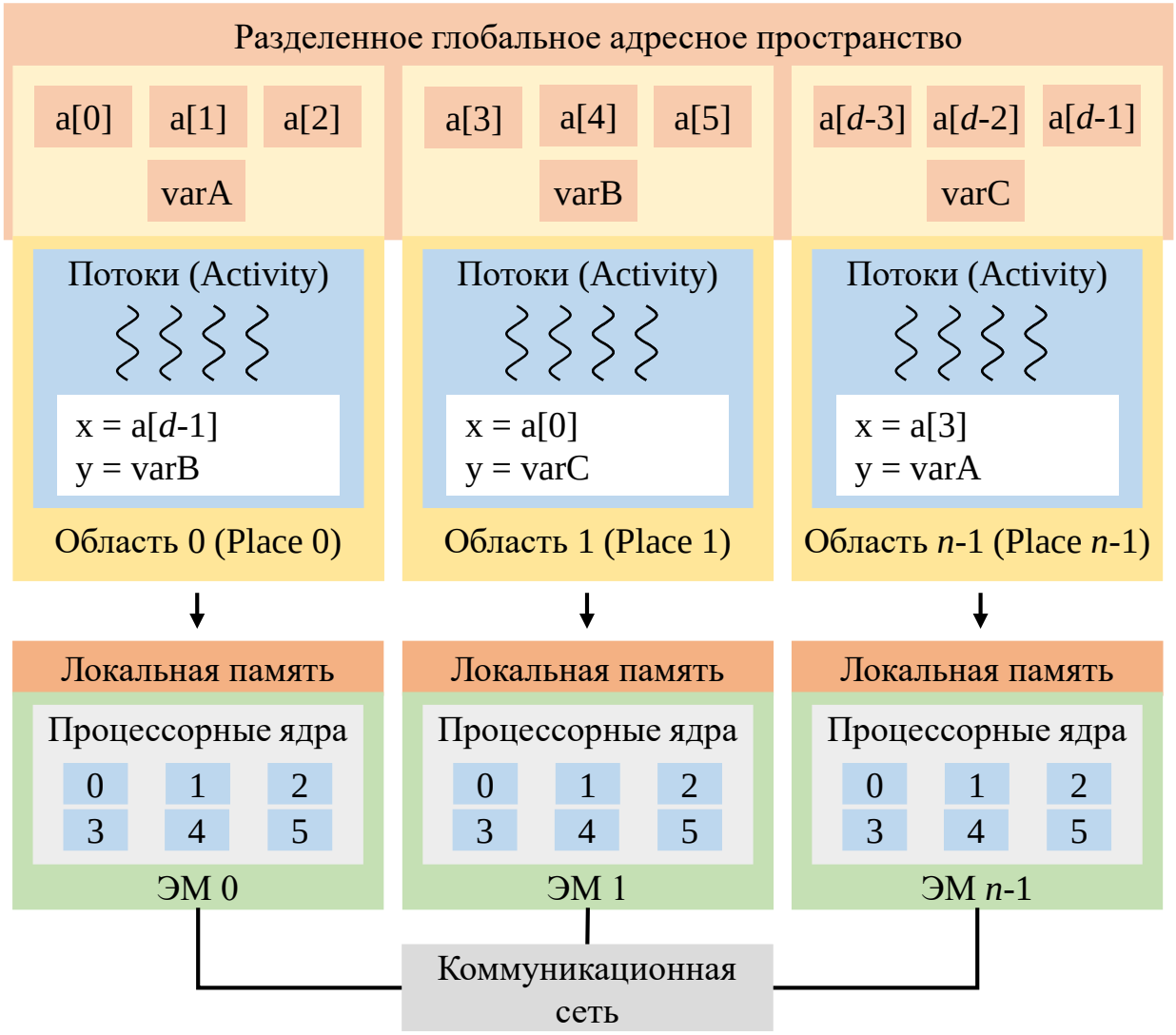


Рисунок 2.1 – Отображение модели PGAS на распределенную ВС

по ним данными. Организация разделенного глобального адресного пространства на распределенной по ЭМ памяти осуществляется компилятором и runtime-системой языка. В случае обращения потока одной области к данным, хранимым в удаленной памяти, runtime-система языка организует их доставку при помощи неявного выполнения двусторонних (*дифференцированных*, *point-to-point communications*), односторонних (*one-sided communications*) или коллективных (*collective communications*) коммуникационных операций.

Современные PGAS-языки условно можно разделить на 2 поколения [4, 30, 58]. Языки первого поколения создавались, чтобы объединить преимущества модели $SPMD$ (*SPMD – Single Program, Multiple Data* (*единая программа, множество данных*) и ВС с общей памятью. Одним из таких активно развивающихся языков является $Unified\ Parallel\ C$

(UPC) [30]. В Листинге 2.1 показан пример реализации алгоритма умножения матрицы на вектор на языке UPC. Ключевое слово $shared$ используется при объявлении массивов a, b и c для того, чтобы сделать их распределенными. Иначе говоря, элементы этих массивов будут блочно храниться в распределенной памяти ВС. Обращение к элементам массива не требует от программиста использования коммуникационных операций для их доставки из удаленной памяти. Эти действия выполняет runtime-система языка, обращения к которой добавляются компилятором.

Листинг 2.1: Пример умножения матрицы на вектор на языке UPC

```
1  shared int a[N][N];
2  shared int b[N],
3             c[N];
4  ...
5  void main (void) {
6      int i, j;
7      upc_forall( i = 0 ; i < N; i++; i) {
8          c[i] = 0;
9          for ( j= 0 ; j < N; j++)
10             c[i] += a[i][j]*b[j];
11     }
12 }
```

Потребность в современных, удобных и высокоуровневых средствах создания масштабируемых параллельных программ для ВС с распределенной памятью привела к появлению PGAS-языков второго поколения [4, 59]. Эти языки удовлетворяют трем основным требованиям:

1. Распределенная память ВС представлена в виде единого глобального адресного пространства;
2. Коммуникационные операции используются только runtime-системой языка;
3. Наличие главного потока управления.

К PGAS-языкам второго поколения относятся следующие языки: IBM X10 [59], Cray Chapel [4], HPF [51], Sun Fortress [23] и др. Отличительной особенностью PGAS-языков второго поколения от первого является то, что они не ограничены использованием модели $SPMD$. Вместо этого, такие языки позволяют параллельную обработку данных за счет выполнения

каждой ЭМ независимого потока управления. Такой подход характерен для многопроцессорных ВС с общей памятью.

Выполнение PGAS-программы, разработанной на языке второго поколения, начинается одним потоком управления нулевой области (Place 0). Эта область называется *главной областью*, а ЭМ, за которой закреплена нулевая область, – *главной ЭМ*. Остальные области – подчиненные, а соответствующие им ЭМ – *подчиненные ЭМ*. Для выполнения параллельной обработки данных в PGAS-языках второго поколения имеются конструкции порождения новых потоков управления как на текущей ЭМ, так и на удаленных ЭМ. Эти конструкции позволяют выделять участки кода для выполнения в новом потоке на текущей (в IBM X10 конструкция $async$) или на удаленной ЭМ (в IBM X10 конструкция at). Конструкция, в которой выделенный участок кода требуется выполнить на подчиненных ЭМ (удаленных ЭМ), называется *конструкцией передачи управления подчиненным ЭМ*.

Так же как и в PGAS-языках первого поколения, в языках второго поколения предусмотрена возможность прозрачной работы с распределенными массивами. В Листинге 2.2 приведен фрагмент программы умножения матрицы на вектор на языке IBM X10. В этом примере массив a блоками хранится в распределенной памяти ВС. Массивы b и c созданы в памяти главной ЭМ. При помощи конструкции at осуществляется передача потока управления подчиненной ЭМ, в памяти которой хранится i-ая строка матрицы a. Подчиненная ЭМ выполняет произведение i-ой строки матрицы a и вектора b. Результат произведения сохраняется в массиве c при помощи глобальной ссылки c_ref.

В модели разделенного глобального адресного пространства отсутствуют коммуникационные операции. При создании PGAS-программ программист использует только языковые конструкции, которые преобразуются PGAS-компилятором в последовательность вызовов функций информационных обменов.

Компиляторы основной части языков программирования семейства PGAS являются *транспилерами* (*source-to-source compiler*) или трансляторами, преобразующими код программы, написанной на одном языке, в аналогичный код, но на другом языке, как правило, C/C++ или Java [31, 59]. На Рисунке 2.2 представлен общий процесс компиляции от PGAS-программ до исполняемого кода или интерпретируемого байт-кода. В общем

Листинг 2.2: Фрагмент программы на языке IBM X10 умножения матрицы на вектор

```
public static def main (Rail[String]) {
    // Распределенный массив
    val a = new DistArray_Block_2[Int](N, N);
    // Локальный массив
    val b = new Array[Int](N);
    val c = new Array[Int](N);
    // Глобальная ссылка на массив <<c>>
    val c_ref = GlobalRef(c);

    for (i in 0..(N-1)) {
    /* Передача управления подчиненной ЭМ
       и запуск на ней нового потока <<async>> */
        at (a.place(i,0)) async {
            var sum:Int = 0;
            for (j in 0..(N-1)) {
                sum += a(i,j)*b(j);
            }
            at (c_ref) {
               c_ref()(i) += sum;
            }
        }
    }
}
```

случае процесс компиляции PGAS-программы содержит 2 этапа. На первом этапе выполняется трансляция PGAS-программы в промежуточный язык. В процессе трансляции компилятор преобразует языковые конструкции в последовательность вызовов функций runtime-системы, которые содержат обращения к коммуникационным операциям. В результате генерируется новая *P-программа* в модели передачи сообщений на другом промежуточном языке программирования, как правило, C/C++ или Java. *P-программа* – это семантически эквивалентное отображение PGAS-программы, разработанной в модели разделенного глобального адресного пространства, на модель передачи сообщений. Трансляция PGAS-программы в P-программу должна обеспечить эффективное использование информационных обменов, так как последние могут привести к существенным накладным расходам во время

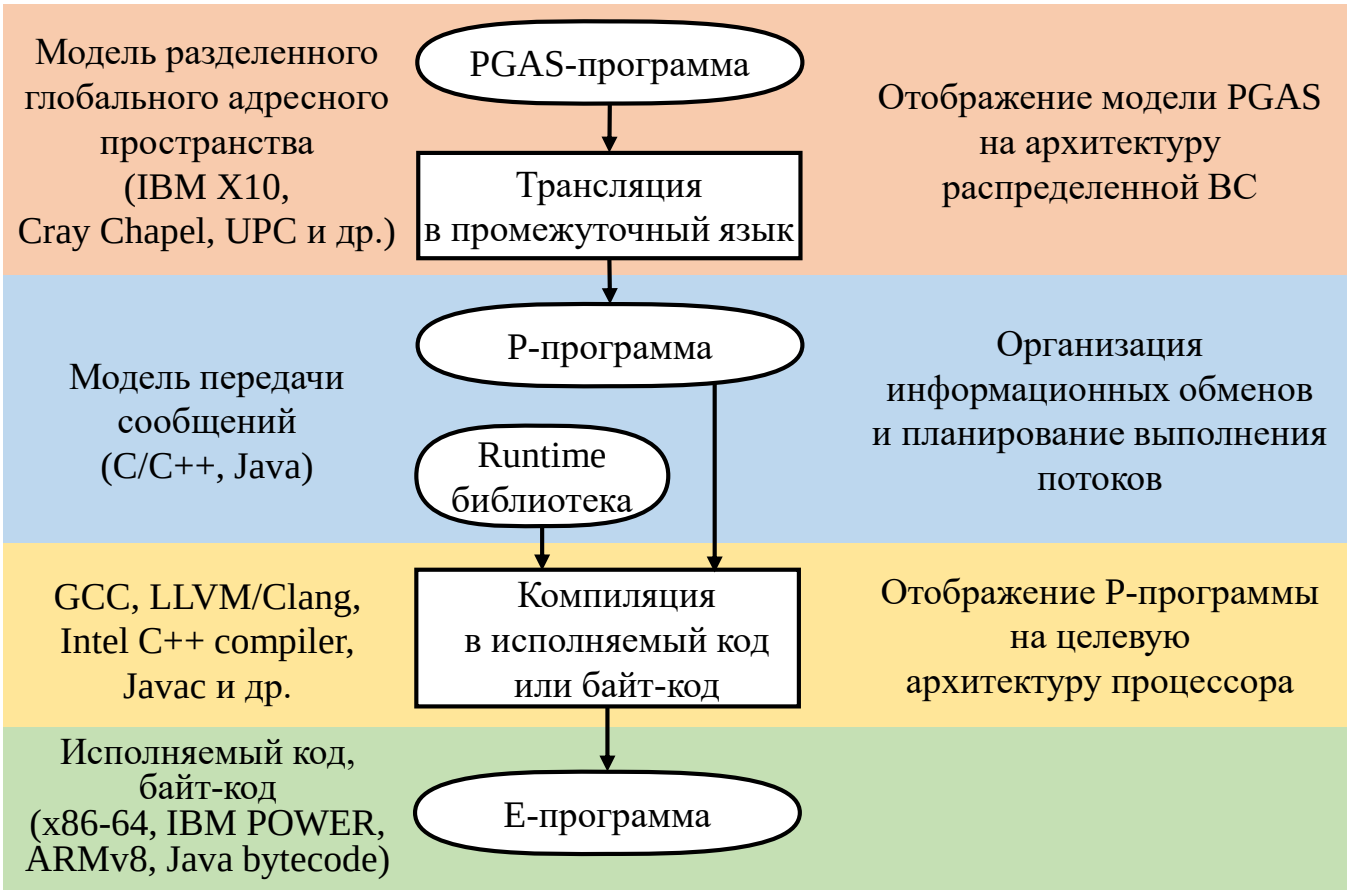


Рисунок 2.2 – Двухэтапный процесс компиляции PGAS-программы

выполнения. По этой причине на данном этапе активно развиваются методы компиляторной оптимизации коммуникационных операций.

На втором этапе выполняется отображение сгенерированной P-программы на целевую архитектуру процессора ЭМ. Данный процесс принято называть *пост-компиляцией*. Для повышения эффективности использования архитектурных возможностей целевого процессора во время пост-компиляции применяются методы оптимизации и автоматической векторизации кода. В результате генерируется *E-программа*, которая будет запущена на распределенной ВС. E-программа представлена исполняемым кодом или интерпретируемым байт-кодом. В роли пост-компилятора могут выступать промышленные компиляторы GCC, Intel C++ compiler, LLVM/Clang, Javac, PGI, PathScale EKO Compiler Suite и др.

На Рисунке 2.3 представлен общий стек программного обеспечения, необходимый для реализации модели PGAS, от стандартной PGAS-библиотеки до драйвера сетевого устройства. Основными компонентами модели разделенного глобального адресного пространства являются PGAS-компилятор, преобразующий P-программу в E-программу, и runtime-система,

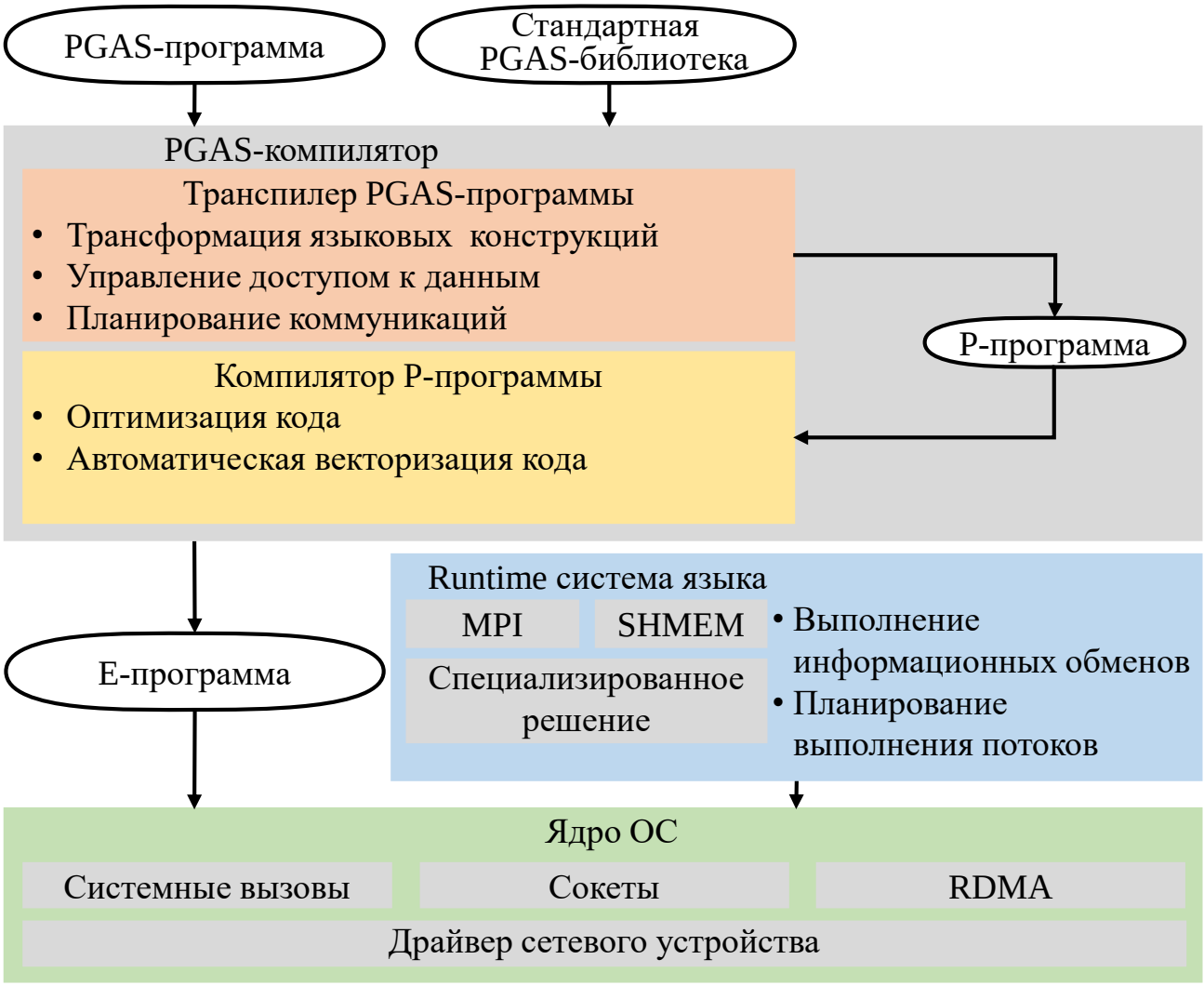


Рисунок 2.3 – Стек программного обеспечения модели PGAS

выполняющая коммуникационные операции. Реализация runtime-системы может быть основана на библиотеках стандарта MPI (MPICH2, Open MPI, MVAPICH2, Cray MPI, IBM MPI, Intel MPI и др.) и семейства SHMEM (SGI-SHMEM, MP-SHMEM, LC-SHMEM, Q-SHMEM, HP SHMEM, IBM SHMEM, OpenSHMEM и др.), либо может быть специализированное решение.

2.2 Процесс выполнения PGAS-программ в модели передачи сообщений

В этом разделе рассматривается процесс запуска скомпилированной PGAS-программы, разработанной на PGAS-языке второго поколения, и ее выполнение в модели передачи сообщений. Языковые PGAS-конструкции

позволяют представить распределенную память ВС в виде единого глобального адресного пространства. В таких языках отсутствует возможность осуществлять явным образом коммуникационные операции. Все информационные обмены между ЭМ выполняются неявно runtime-системой языка. Компилятор языков семейства PGAS транслирует языковые конструкции в последовательность вызовов функций runtime-библиотеки, которые выполняют информационные обмены между ЭМ.

Рисунок 2.4 демонстрирует процесс запуска скомпилированной PGAS-программы на ВС с распределенной памятью. E-программа, полученная в результате компиляции, копируется на каждую ЭМ и запускается. После запуска на каждой ЭМ создается область. На главной ЭМ в рамках области запускается основной поток, который начинает выполнять пользовательский код.

Процесс выполнения PGAS-программы на ВС с распределенной памятью состоит из фаз выполнения вычислений и обращения к функциям runtime-системы для управления параллельными процессами и передачи сообщений. Функционирование runtime-системы PGAS-языков второго поколения подобно организации вычислений в модели акторов [1]. Акторами в данном случае выступают ЭМ, среди которых один актор выделен главным (главная ЭМ или главная область). Информационные обмены осуществляются при помощи функций $Send/Receive$, первый аргумент в которых – номер принимающей/отправляющей ЭМ, а второй – отправляемое сообщение. Структура сообщения состоит из полей $type$, $code$ и $data$. Поле $type$ содержит информацию о типе сообщения. Поле $code$ хранит код, который был отправлен для выполнения на текущей ЭМ. Данные, требуемые для выполнения этого кода, хранятся в поле $data$.

Код и используемые данные передаются в *сериализованном* виде, т.е. в виде последовательности байт. В зависимости от реализации runtime-системы языка сериализация может быть выполнена различными способами, например, в случае объектно-ориентированных PGAS-языков код и данные сообщения формируют языковой объект (C++-, Java-объект), который передается в виде последовательности байт. В процедурных PGAS-языках передаваемые участки кода могут быть вынесены в отдельные функции, каждой из которых назначается уникальный идентификатор. В этом случае для сериализации кода потребуется найти уникальный идентификатор

соответствующей функции, который будет записан в поле *code* передаваемого сообщения. Так как на главной и подчиненных ЭМ выполняется одна и та же E-программа, принимающая ЭМ сможет найти по идентификатору функцию и вызвать ее.

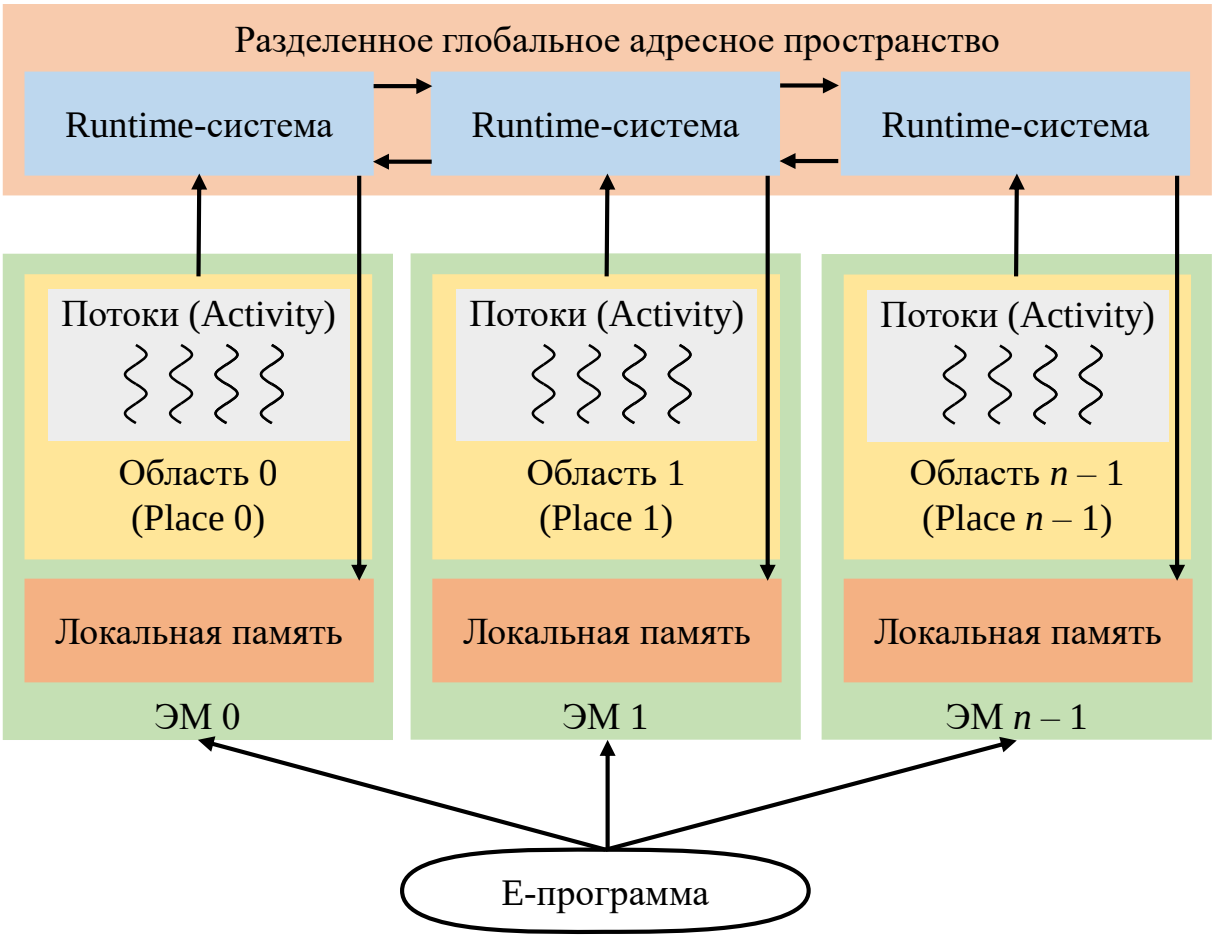


Рисунок 2.4 – Запуск и выполнение скомпилированной PGAS-программы

В Алгоритме 1 представлена логика работы runtime-системы на главной ЭМ абстрактного PGAS-языка. После запуска PGAS-программы главная ЭМ выполняет следующие основные действия:

1. Рассылает всем подчиненным ЭМ сообщение с типом $InitMsg$, требующее выполнить инициализацию локальных данных;
2. Выполняет пользовательский код. Пользовательский код может содержать вызовы коммуникационных функций, необходимые для создания распределенных массивов, а также для организации передачи потока управления подчиненным ЭМ. В этом случае сообщение имеет тип $UserCodeMsg$;

Алгоритм 1 Алгоритм работы runtime-системы главной ЭМ

Input: n – Количество ЭМ

```
1:  function MasterWorkload(n)
2:      msg.type ← InitMsg
3:      for i in 1..(n − 1) do
4:          Send(i, msg)
5:      done
6:      ExecuteUserProgram()          ▷ Выполнение пользовательского кода
7:      msg.type ← FinalizeMsg
8:      for i in 1..(n − 1) do
9:          Send(i, msg)
10:     done
11: end function
```

3. После завершения выполнения пользовательского кода рассылает всем подчиненным ЭМ сообщение с типом $FinalizeMsg$, сигнализирующее о завершении выполнения PGAS-программы.

Алгоритм 2 демонстрирует логику работы runtime-системы подчиненных ЭМ. Каждая подчиненная ЭМ хранит очередь q принятых сообщений. После запуска PGAS-программы на распределенной ВС подчиненные ЭМ ожидают сообщения инициализации с типом $InitMsg$ от главной ЭМ. Получив данное сообщение, ЭМ выполняют инициализацию локальных данных, необходимых для работы runtime-системы. Вновь пришедшие сообщения от главной ЭМ или от подчиненных помещаются в очередь q. Подчиненная ЭМ извлекает из очереди сообщение и при помощи функции $ExecuteMsgCode$ выполняет его код, содержащий команды тела конструкции передачи потока управления или команды для создания части распределенного массива в локальной памяти текущей ЭМ. Для выполнения кода сообщения функция $ExecuteMsgCode$ сперва должна десериализовать поля $code$ и $data$. Алгоритм выполняется до тех пор, пока не будет извлечено сообщение с типом $FinalizeMsg$, служащее сигналом завершения выполнения PGAS-программы.

Алгоритм 2 Алгоритм работы runtime-системы подчиненных ЭМ

Input: n – Количество ЭМ

```
1:  function SlaveWorkload(n)
2:      while true do
3:          msg ← QueueGetMsg(q)
4:          if msg.type == InitMsg then
5:              Initialize(n)
6:              continue
7:          end if
8:          if msg.type == UserCodeMsg then
9:              ExecuteMsgCode(msg)
10:             continue
11:         end if
12:         if msg.type == FinalizeMsg then
13:             Finalize(n)
14:             Exit()
15:         end if
16:     end while
17: end function
```

2.3 Конструкция передачи потока управления подчиненным ЭМ

В PGAS-языках второго поколения, таких как IBM X10 и Cray Chapel, присутствует конструкция передачи потока управления от главной ЭМ к подчиненной. В языке IBM X10 реализована конструкция at, в Cray Chapel – конструкция on. Такие конструкции позволяют выделять в программе участок кода, который требуется выполнить на указанной ЭМ. Этот участок кода принято называть *телом конструкции*, и именно он в сериализованном виде помещается в поле $code$ сообщения, отправляемого runtime-системой языка.

Листинг 2.3 демонстрирует пример использования конструкции at передачи потока управления подчиненным ЭМ на языке IBM X10. В этом примере осуществляется вызов функции $ComputeSum$ суммирования элементов массива a на подчиненной ЭМ (удаленной ЭМ). Параметр $world.next(here)$ конструкции at указывает ЭМ, которой будет передано

Листинг 2.3: Пример использования конструкции at на языке $IBM\ X10$

```
1 public static def main (Rail[String]) {
2   val a = [1,2,3];
3   val world = Place.places();
4
5   at(world.next(here)) {
6       ComputeSum(a);
7   }
8 }
```

управление. Тело конструкции содержит код, выполняемый указанной ЭМ. В указанном выше примере тело конструкции содержит вызов функции $ComputeSum$, которой в качестве аргумента передается массив a.

Для выполнения тела конструкции передачи потока управления подчиненной ЭМ необходимо скопировать в память ЭМ код конструкции (функцию $ComputeSum$) и используемые данные (массив a). Для этого компилятор PGAS-языка преобразует конструкцию at следующим образом:

1. Выделяет тело конструкции в отдельную фунцию;
2. Создает сообщение, передаваемое подчиненной ЭМ, которое в сериализованном виде содержит функцию – тело конструкции at и используемые данные;
3. Вставляет вызовы коммуникационных функций для отправки созданного сообщения подчиненной ЭМ.

Листинг 2.4: Пример трансформации конструкции at PGAS-компилятором

```
1 def BodyAt(a:Array[Long]) {
2   ComputeSum(a);
3 }
4 ...
5 public static def main (Rail[String]) {
6   val a = [1,2,3];
7   val world = Place.places();
8
9   msg.code = Serialize(BodyAt);
10  msg.data = Serialize(a);
11  id = world.next(here);
12  Send(id, msg);
13 }
```

В Листинге 2.4 показан пример $X10$-подобного промежуточного представления, полученного после преобразования конструкции at PGAS-компилятором программы, приведенной в Листинге 2.3. В результате формируется функция $BodyAt$, содержащая код тела конструкции at, вставлены вызовы функций сериализации $Serialize$ и функция $Send$ для отправки созданного сообщения msg подчиненной ЭМ с номером id.

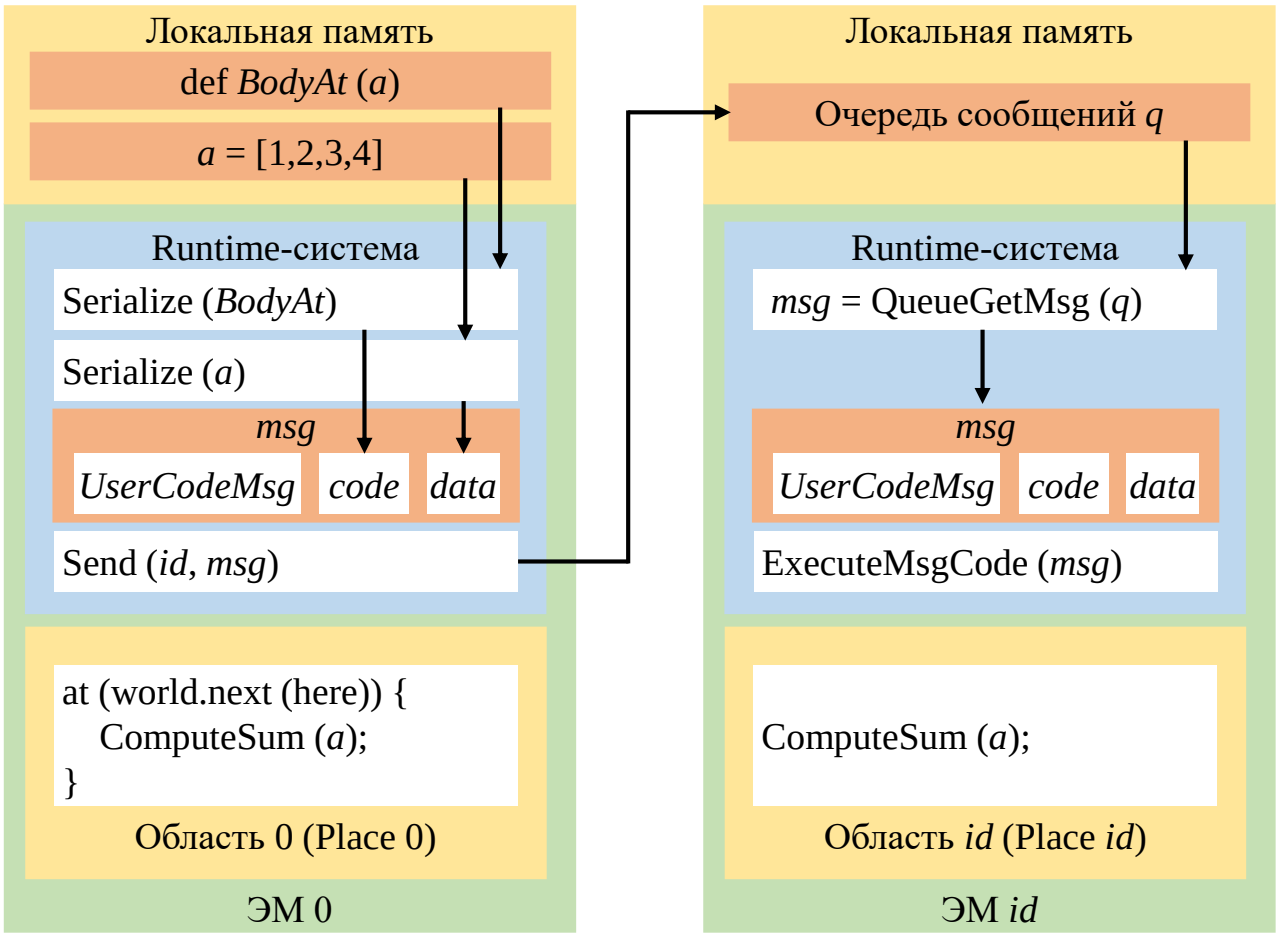


Рисунок 2.5 – Выполнение конструкции передачи потока управления подчиненной ЭМ

На Рисунке 2.5 изображена интерпретация выполнения конструкции передачи потока управления подчиненным ЭМ (Листинг 2.4) в модели передачи сообщений. Во время выполнения PGAS-программы runtime-система языка при помощи функции `Serialize` сериализует код тела конструкции `at`, который на этапе компиляции преобразован в функцию `BodyAt`. Массив `a`, хранимый в памяти главной ЭМ, также сериализуется в последовательность байт для отправки подчиненной ЭМ с номером `id`. Результаты сериализации сохраняются в сообщении `msg` в полях `code` и `data`. После

завершения формирования сообщения осуществляется его отправка ЭМ с номером `id` при помощи функции `Send`.

Принятые сообщения подчиненной ЭМ с номером `id` сохраняются в очереди `q`. Runtime-система подчиненной ЭМ извлекает из очереди сообщение `msg` и при помощи функции `ExecuteMsgCode` выполняет код сообщения, предварительно выполнив его десериализацию. В данном случае ЭМ с номером `id` выполнит отправленную ей функцию `BodyAt` над массивом `a`.

2.4 Конструкция циклической передачи потока управления подчиненным ЭМ

Одним из часто используемых шаблонов в программах, разработанных на PGAS-языках второго поколения, является циклическая конструкция передачи потока управления подчиненным ЭМ. В этой конструкции присутствует цикл из r итераций. На каждой итерации осуществляется передача потока управления подчиненным ЭМ из множества M при помощи конструкции `at`. В Листинге 2.5 представлен пример такого шаблона на языке IBM X10. В этом примере функция `f` осуществляет доступ к элементам массива `a`.

Наличие таких шаблонов в коде PGAS-программ может послужить источником накладных расходов на коммуникационные операции, необходимые для копирования в память подчиненных ЭМ, используемых в теле конструкции данных. Ключевую роль в организации эффективного выполнения конструкций циклической передачи потока управления играет PGAS-компилятор. Во время трансляции PGAS-програмы в P-программу компилятор определяет схему информационных обменов между главной и подчиненными ЭМ.

Актуальной является задача создания алгоритмов организации эффективного выполнения конструкций циклической передачи потока управления подчиненным ЭМ.

Листинг 2.5: Доступ подчиненных ЭМ к массиву a на языке $IBM\ X10$

```
val a = new Array[Long](d);
val M = Place.places();
...
for (i in 0..(r - 1)) {
    for (j in M) {
        at (j) {
            f(a);
        }
    }
}
```

2.5 Задача трансформации конструкций циклической передачи потока управления подчиненным ЭМ

Постановка задачи. Имеется параллельная PGAS-программа, которая будет выполняться на распределенной ВС из n ЭМ. В программе присутствует конструкция циклической передачи потока управления подчиненным ЭМ. Цикл конструкции содержит r итераций, на каждой из которых осуществляется передача управления подчиненным ЭМ из множества M при помощи конструкции `at`. Всего на каждой итерации управление передается $m = |M|$ элементарным машинам.

Находящаяся внутри цикла конструкция передачи управления подчиненным ЭМ удовлетворяет следующим требованиям:

1. В теле конструкции выполняются операции над элементами l массивов a_0,...a_{l-1};
2. Тело конструкции представлено функциями $f_0(a_0), ..., f_{l-1}(a_{l-1})$, где функция $f_i(a_i)$ содержит операции, выполняемые над элементами массива a_i;
3. Массивы $a_0, ..., a_{l-1}$ хранятся в памяти главной ЭМ и не требуют согласования состояния с массивами, скопированными в память подчиненных ЭМ.

В Листинге 2.6 приведен пример конструкции циклической передачи потока управления подчиненным ЭМ, удовлетворяющей приведенным

Листинг 2.6: Пример конструкции, удовлетворяющей введенным требованиям на X10-подобном PGAS-языке

```
1  val a_0 = new Array[Long](s_0);
2  ...
3  val a_{l-1} = new Array[Long](s_{l-1});
4
5  /* Начало циклической конструкции в модели PGAS */
6  for (i in 0,..,r-1) {
7      for (j in M) {
8          at (j) {
9              f_0(a_0);
10             ...
11             f_{l-1}(a_{l-1});
12         }
13     }
14 }
15 /* Конец конструкции */
```

требованиям. Для ее выполнения PGAS-компилятор должен отобразить высокоуровневую PGAS-конструкцию на модель передачи сообщений. Иными словами, на этапе компиляции необходимо трансформировать цикл из r итераций, а также добавить вызовы функции runtime-системы языка для организации информационных обменов.

Требуется. Разработать алгоритм преобразования циклических конструкций передачи потока управления ЭМ из модели PGAS в модель передачи сообщений.

На вход алгоритма поступает промежуточное представление кода высокоуровневой конструкции циклической передачи потока управления подчиненным ЭМ в модели PGAS, представленное абстрактным синтаксическим деревом (АСД). Во время компиляции алгоритм трансформирует цикл, преобразует конструкцию передачи потока управления в последовательность вызовов функций runtime-библиотеки для копирования функций $f_0,...,f_{l-1}$ и доставки массивов $a_0,...,a_{l-1}$ в память подчиненных ЭМ. Таким образом, результатом работы алгоритма является модифицированная P-программа в модели передачи сообщений.

2.6 Алгоритмы трансформации конструкций циклической передачи потока управления подчиненным ЭМ

Во время передачи потока управления подчиненным ЭМ runtime-система при помощи информационных обменов выполняет копирование требуемых данных в память удаленной ЭМ. Обращения к функциям runtime-системы для выполнения коммуникационных операций добавляются PGAS-компилятором во время преобразования PGAS-программы в P-программу модели передачи сообщений.

Для трансформации конструкций циклической передачи потока управления подчиненным ЭМ в модель передачи сообщений компиляторы PGAS-языков второго поколения широко используют алгоритм *By-Iterative Copying* (*BIC*) [59], который генерирует обращения к функциям коммуникационных операций на каждой итерации цикла. Этот алгоритм не требует сложного анализа промежуточного представления тела конструкции на этапе компиляции программы, так как используемые массивы передаются подчиненным ЭМ из множества M на каждой итерации $i = \{0..r-1\}$. Достоинством данного алгоритма является универсальность его использования, т.е. он применим к конструкциям всех возможных видов. Однако, существенный недостаток этого алгоритма заключается в том, что избыточное копирование всех элементов используемых массивов $a_0,...,a_{l-1}$ на каждой итерации приведет к существенным накладным расходам во время выполнения PGAS-программы.

Для сокращения накладных расходов на избыточное копирование элементов массивов в память подчиненных ЭМ используется алгоритм *Scalar Replacement* [5, 7]. Во время трансформации циклической конструкции передачи потока управления алгоритм добавляет вызовы коммуникационных операций для отправки только используемых элементов массивов $a_0,...,a_{l-1}$, а не всех целиком, как в случае использования алгоритма *BIC*. Подобная трансформация кода циклической конструкции требует дополнительных действий на этапе статического анализа промежуточного представления PGAS-программы, так как необходимо определить индексы используемых элементов. Недостатком этого алгоритма является ограниченность его применения, а именно, алгоритм не применим в случае, если на этапе статического анализа промежуточного представления PGAS-программы не представляется

возможным определить значения индексов используемых элементов массивов $a_0,...,a_{l-1}$.

Сократить накладные расходы на избыточное копирование массивов $a_0,...,a_{l-1}$ в память подчиненных ЭМ, но при этом избежать недостатки алгоритма *Scalar Replacement* позволяет предложенный автором алгоритм опережающего копирования *Array Preload* [35, 69, 78, 80–83, 90]. Данный алгоритм выполняет преобразование циклической конструкции, после которого копирование массивов $a_0,...,a_{l-1}$ в удаленную память подчиненных ЭМ происходит только один раз перед итерациями цикла. Так как массивы копируются целиком, а не только используемые элементы, алгоритм может быть применим в тех случаях, когда на этапе статического анализа кода PGAS-программы невозможно определить значения индексов используемых элементов. Недостатком алгоритма *Array Preload* является необходимость на этапе компиляции производить существенную трансформацию конструкции циклической передачи управления подчиненной ЭМ, так как необходимо добавить пролог цикла и изменить обращения к массивам $a_0,...,a_{l-1}$ в функциях $f_0,...,f_{l-1}$.

2.6.1 Алгоритм *By-Iterative Copying* трансформации циклических конструкций произвольной формы

В компиляторах PGAS-языков второго поколения, таких как IBM X10 [59] и Cray Chapel [4], для трансформации конструкций циклической передачи потока управления подчиненным ЭМ большое распространение получил алгоритм *BIC*, так как область его применимости не ограничена формой конструкции. Этот алгоритм преобразует цикл так, что во время его выполнения на каждой итерации происходят информационные обмены для доставки требуемых данных в память удаленной ЭМ. Логика выполнения P-программы, полученной после трансформации PGAS-программы (Листинг 2.6) алгоритмом *BIC*, приведена в Алгоритме 3. В результате трансформации тело конструкции at преобразовано в функцию $BodyAt$, а в тело цикла добавлен код для создания сообщения msg и его отправки ЭМ с номером $j \in M$.

Алгоритм 3 Алгоритм выполнения циклической конструкции передачи потока управления подчиненным ЭМ в P-программе, полученной алгоритмом *ByIterative Copying*

```
1:  function BodyAt((a_0,...,a_{l-1}))
2:      f_0(a_0)
3:      ...
4:      f_{l-1}(a_{l-1})
5:  end function
6:  ...
7:  function Main()
8:      ...
9:      ▷ Циклическая конструкция P-программы в модели передачи сообщений
10:     for i in {0,..,r − 1} do
11:         for j in M do
12:             msg.type ← UserCodeMsg
13:             msg.data ← Serialize((a_0,...,a_{l-1}))
14:             msg.code ← Serialize(BodyAt)
15:             Send(j, msg)
16:         done
17:     done
18:     ▷ Конец конструкции
19:     ...
20: end function
```

Алгоритм 4 содержит логику работу *By-iterative Copying*. На вход алгоритма поступает абстрактное синтаксическое дерево C (Рисунок 2.6) конструкции циклической передачи потока управления подчиненным ЭМ из множества M. На выходе формируется модифицированное АСД C' (Рисунок 2.8) конструкции, а также АСД P (Рисунок 2.7), отображающее функцию BodyAt. Алгоритм состоит из 7 основных шагов:

1. Поиск в АСД C поддерева B, которому сопоставлено тело внутреннего цикла, на итерациях которого осуществляется передача потока управления подчиненным ЭМ при помощи конструкции at. Поиск выполняет функция LookupInnerLoopBody.

Алгоритм 4 Алгоритм *By-iterative Copying* трансформации конструкций произвольной формы

Input: C – АСД конструкции циклической передачи потока управления подчиненным ЭМ;

Output: C' – модифицированное АСД конструкции циклической передачи потока управления подчиненным ЭМ;
P – АСД функции $BodyAt$, представляющей тело конструкции at;

1: **function** DEFAULTTRANSFORMATION(C, C', P)
2: $B \leftarrow$ LOOKUPINNERLOOPBODY(C)
3: $r \leftarrow$ LOOKUPIDREMOTENODE(C)
4: $(a_0,...,a_{l-1}) \leftarrow$ LOOKUPARRAYS(B)
5: $(f_0,...,f_{l-1}) \leftarrow$ LOOKUPOPERATORS($B, (a_0,...,a_{l-1})$)
6: $P \leftarrow$ BUILDTREEBODYAT($(f_0,...,f_{l-1}), (a_0,...,a_{l-1}), B$)
7: $B' \leftarrow$ BUILDASTSENDMSG($B, (f_0,...,f_{l-1}), (a_0,...,a_{l-1}), P, r$)
8: **end function**

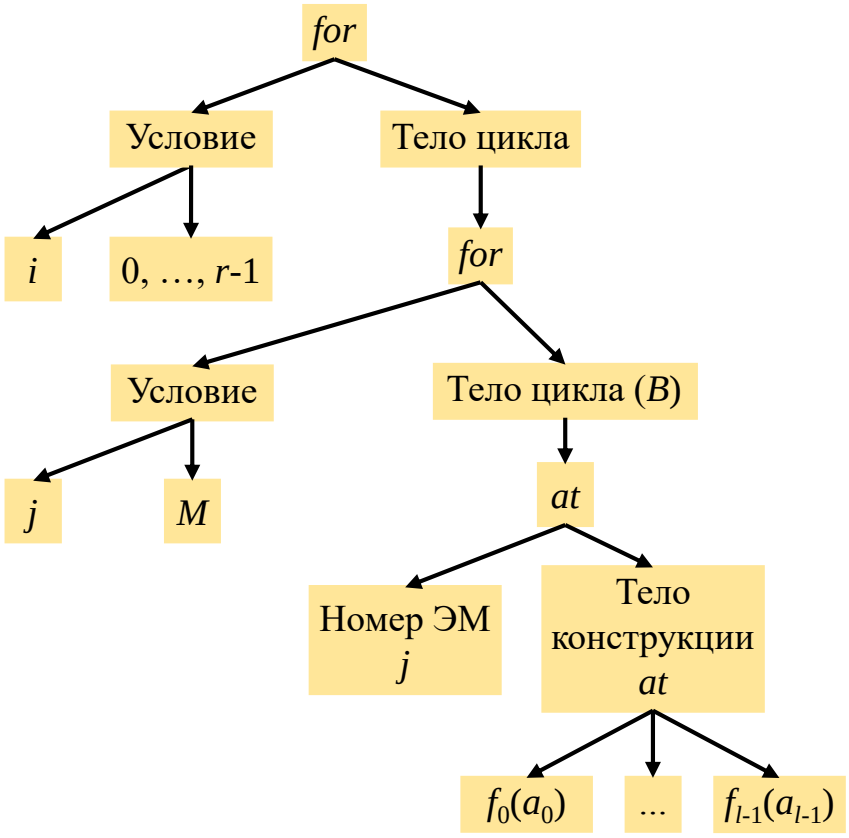


Рисунок 2.6 – Фрагмент АСД, отображающего циклическую конструкцию передачи потока управления подчиненным ЭМ

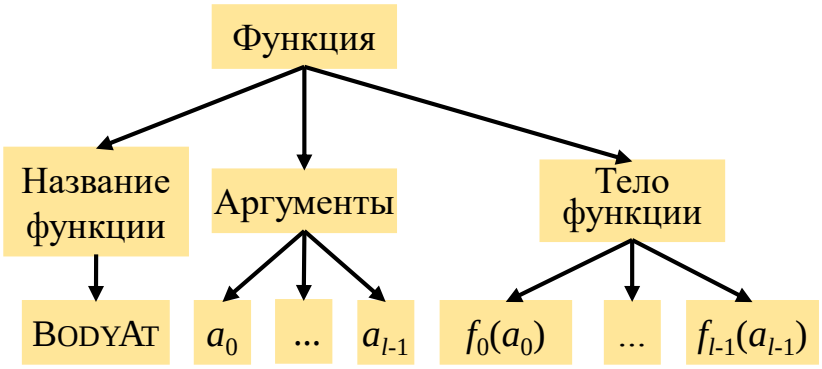


Рисунок 2.7 – Фрагмент АСД, которому соответствует функция *BodyAt*, сформированная алгоритмом *By-iterative Copying*

2. Поиск в АСД C узла r, которому сопоставлен номер удаленной ЭМ $j \in M$. Поиск осуществляется при помощи функции LookupIdRemoteNode.
3. Поиск в АСД B и формирование последовательности узлов $(a_0,...,a_{l-1})$, которые отображают массивы, чьи элементы используются внутри тела конструкции at. Для этого используется функция LookupArrays, принимающая в качестве входного аргумента корень АСД B тела внутреннего цикла.
4. Поиск в АСД B и формирование последовательности узлов $(f_0,...,f_{l-1})$, которым соответствуют функции, содержащие операции работы с элементами массивов $a_0,...,a_{l-1}$. Поиск выполняется при помощи функции LookupOperators, принимающей на вход корень АСД B и список его узлов $(a_0,...,a_{l-1})$ АСД.
5. Построение АСД P, отображающего функцию BodyAt. Для построения дерева используется функция BuildTreeBodyAt. На Рисунке 2.7 показан пример результирующего на данном шаге АСД.
6. Формирование модифицированного АСД B', которое соответствует телу цикла P-программы в модели передачи сообщений. Модифицированное АСД содержит узлы, которым сопоставлен код для формирования сообщения msg и отправки его ЭМ с номером j. Эти действия выполняются функцией BuildASTSendMsg.
7. Замена в АСД C поддерева B тела внутреннего цикла PGAS-программы на поддерево B' преобразованного тела этого же цикла P-программы. Замена производится при помощи функции ReplaceAST. В результате формируется модифицированное АСД

C' конструкции в модели передачи сообщений для P-программы. На Рисунке 2.8 приведено результирующее АСД C' после всех преобразований алгоритма.

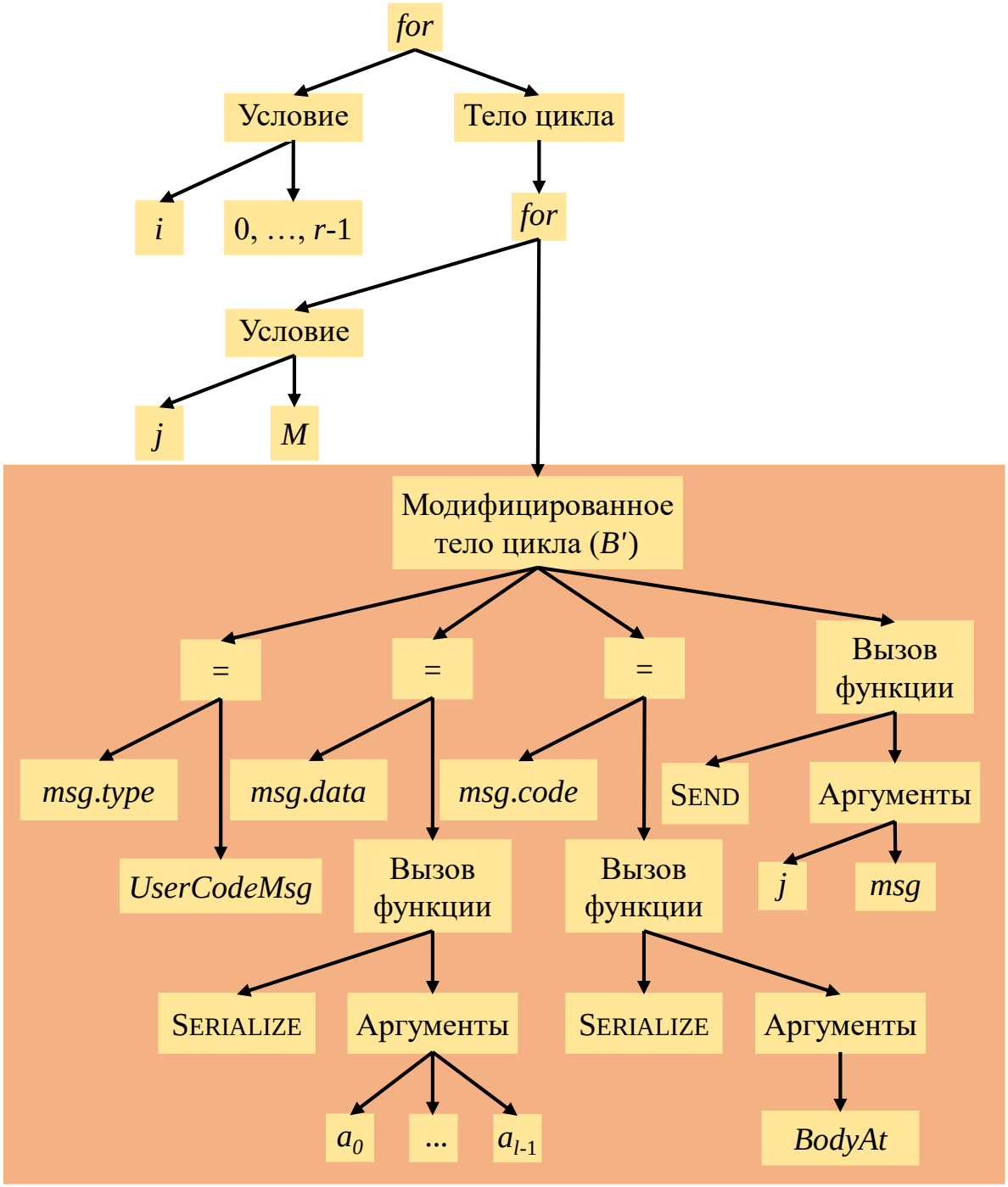


Рисунок 2.8 – Фрагмент АСД конструкции циклической передачи потока управления подчиненным ЭМ, полученный после трансформации алгоритмом *By-iterative Copying*

2.6.2 Алгоритм *Scalar Replacement* копирования отдельных элементов массивов

Параллельные PGAS-программы могут содержать конструкции циклической передачи управления подчиненным ЭМ, в которых используются элементы массивов с заранее известными значениями индексов, либо их можно определить на этапе компиляции. Для сокращения накладных расходов на передачу используемых данных в таких программах компиляторы для PGAS-языков второго поколения (IBM X10, Cray Chapel), реализуют алгоритм *Scalar Replacement* [5, 7]. Этот алгоритм преобразует конструкции циклической передачи управления подчиненным ЭМ PGAS-программы при ее отображении на модель передачи сообщений таким образом, чтобы в память удаленных ЭМ копировались только используемые элементы массивов $a_0,...,a_{l-1}$, а не все, как в случае использования *By-iterative Copying*.

Алгоритм *Scalar Replacement* подразумевает, что на этапе компиляции возможно определить последовательность $T = (t_0,...,t_{q-1})$ используемых элементов массивов $a_0,...,a_{l-1}$, где $t_i \in a_j, 0 \leqslant j \leqslant l-1$ и последовательность $(g_0,...,g_{q-1})$ операций над ними. В алгоритме 5 представлена логика работы P-программы, которая является отображением PGAS-программы из Листинга 2.6 на модель передачи сообщений после преобразований *Scalar Replacement*. В результате преобразований все используемые элементы массивов, собранные в последовательности T, будут на каждой итерации i копироваться в память удаленных ЭМ с номером j. Тело конструкции `at`, представленное функцией BodyAt, также подверглось трансформации, а именно, после преобразований оно содержит не функции $f_0,...,f_{l-1}$ операций над массивами $a_0,...,a_{l-1}$, а функции $g_0,...,g_{q-1}$ операций над элементами из последовательности T.

Алгоритм 6 содержит логику работы *Scalar Replacement*. На вход алгоритма поступает абстрактное синтаксическое дерево C (Рисунок 2.6) конструкции циклической передачи потока управления подчиненным ЭМ из множества M. На выходе формируется модифицированное алгоритмом АСД C' (Рисунок 2.11) конструкции, а также АСД P (Рисунок 2.10), представляющее функцию BodyAt. Алгоритм состоит из 7 основных шагов:

Алгоритм 5 Алгоритм выполнения циклической конструкции передачи потока управления подчиненным ЭМ в Р-программе, полученной алгоритмом *Scalar Replacement*

```
1: function BodyAt(T)
2:     g_0(t_0)
3:     ...
4:     g_{q-1}(t_{q-1})
5: end function
6: ...
7: function Main()
8:     ...
9:     ▷ Циклическая конструкция Р-программы в модели передачи сообщений
10:    for i in {0,..,r − 1} do
11:        for j in M do
12:            msg.type ← UserCodeMsg
13:            msg.data ← Serialize(T)
14:            msg.code ← Serialize(BodyAt)
15:            Send(j, msg)
16:        done
17:    done
18:    ▷ Конец конструкции
19:    ...
20: end function
```

1. Поиск в АСД C поддерева B, которому соответствует тело внутреннего цикла;
2. Поиск в АСД B узла r, соответствующего номерам подчиненных ЭМ, которым конструкция at передаст управление во время выполнения;
3. Поиск в АСД B поддерева A, содержащего узлы, формирующие тело конструкции at. На Рисунке 2.9 изображен пример такого АСД;
4. Поиск в теле конструкции at используемых элементов массивов a_1,...,a_{l-1} и формирование последовательности T из найденных элементов;

5. Поиск в теле конструкции at операторов, выполняющих действия над элементами массивов из последовательности T, и формирование из них последовательности $(g_0,...,g_{q-1})$;
6. Построение АСД P, которое представляет функцию $BodyAt$, содержащую трансформированное тело конструкции at. В результате преобразований функции $f_0,...,f_{l-1}$ с операциями над массивами $a_0,...,a_{l-1}$ заменяются на операторы $(g_0,...,g_{q-1})$ с операндами из последовательности T;
7. Построение модифицированного АСД B', соответствующего циклической конструкции, содержащей код создания и отправки сообщения msg.

Алгоритм 6 Алгоритм *Scalar Replacement* трансформации конструкций с обращением к определенным элементам массивов

Input: C – АСД конструкции циклической передачи потока управления подчиненным ЭМ;

Output: C' – модифицированное АСД конструкции циклической передачи потока управления подчиненным ЭМ;
P – АСД функции $BodyAt$, представляющей тело конструкции at;

1: **function** ScalarReplacementTransformation(C, C', P)
2: $B \leftarrow$ LookupInnerLoopBody(C)
3: $r \leftarrow$ LookupIdRemoteNode(C)
4: $A \leftarrow$ LookupAtBody(B)
5: $T \leftarrow$ LookupUsedArraysElements(A)
6: $(g_0,...,g_{q-1}) \leftarrow$ LookupOperators(B, T)
7: $(a_0,...,a_{l-1}) \leftarrow$ LookupArrays(B)
8: $(f_0,...,f_{l-1}) \leftarrow$ LookupOperators($B, (a_0,...,a_{l-1})$)
9: $P \leftarrow$ BuildTreeBodyAt($(f_0,...,f_{l-1}), (a_0,...,a_{l-1}), B$)
10: $B' \leftarrow$ BuildASTSendMsg($B, (f_0,...,f_{l-1}), (a_0,...,a_{l-1}), P, r$)
11: **end function**

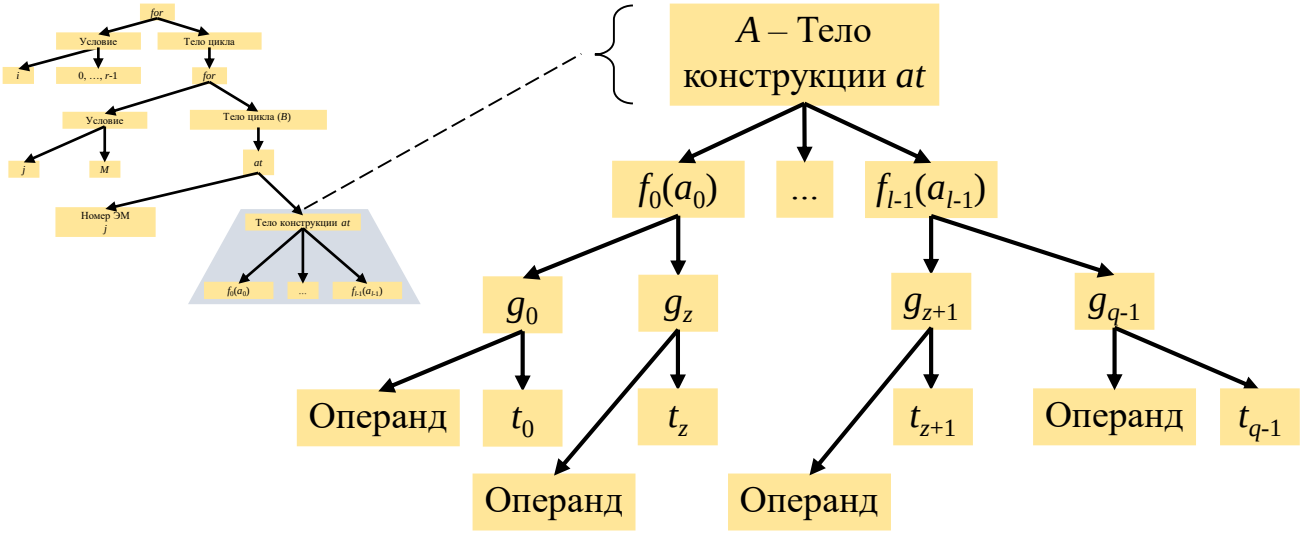


Рисунок 2.9 – Фрагмент АСД циклической передачи потока управления подчиненным ЭМ с развернутым телом конструкции `at`

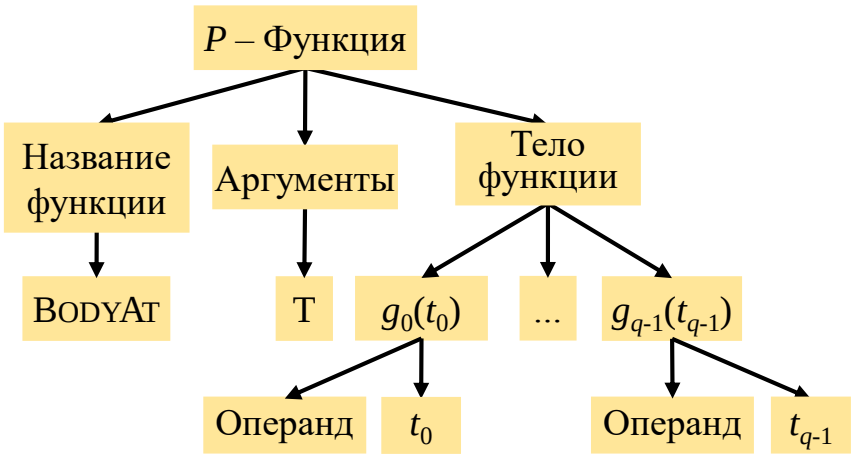


Рисунок 2.10 – Фрагмент АСД, которому соответствует функция BodyAt, сформированная алгоритмом *Scalar Replacement*

2.6.3 Алгоритм *Array Preload* опережающего копирования массивов

Идея алгоритма *Array Preload* заключается в преобразовании конструкции циклической передачи потока управления подчиненным ЭМ таким образом, чтобы при выполнении PGAS-программы копирование используемых данных в память удаленных ЭМ осуществлялось только один раз перед циклом, а не на каждой итерации i [35, 69]. Такой подход принято называть *опережающим копированием массивов* (*preemptive copying*).

Опережающее копирование массивов в удаленную память подчиненных ЭМ выполняется прологом цикла. Основные преобразования, выполняемые алгоритмом *Array Preload*, направлены на формирование пролога цикла для

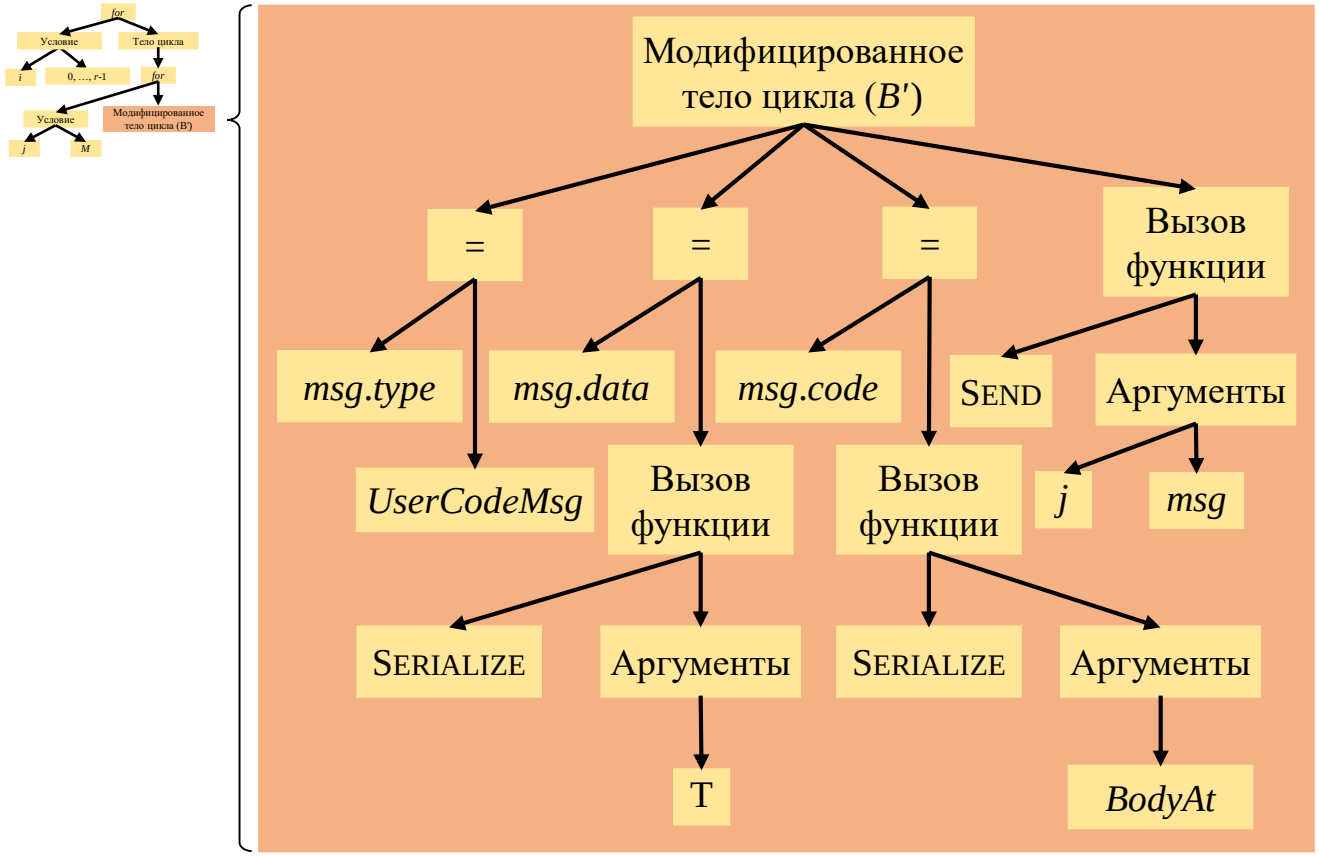


Рисунок 2.11 – Фрагмент АСД конструкции циклической передачи потока управления подчиненным ЭМ, полученный после трансформации алгоритмом *Scalar Replacement*

отправки подчиненным ЭМ используемых массивов и создания их локальных копий в памяти ЭМ, а также на преобразование тела конструкции передачи управления. В ходе преобразований тела конструкции обращения к массиву, расположенному в памяти главной ЭМ, заменяются на обращения к его локальной копии, созданной в памяти подчиненных ЭМ.

Алгоритм 8 содержит логику работу Р-программы, полученной преобразованиями *Array Preload*. В результате применения алгоритма была добавлена функция PROLOGUEBODY, создающая локальные копии массивов, пролог цикла, выполняющий отправку сообщений удаленным ЭМ, и модифицирована функция *BodyAt*, в которой обращения к массивам $a_0,...,a_{l-1}$ заменены на обращения к их локальным копиям $a'_{l-1},...,a'_{l-1}$. Таким образом, отправка массивов $a_0,...,a_{l-1}$ ЭМ из множества M на каждой итерации i не требуется.

Алгоритм 7 содержит основные этапы работы *Array Preload*. На вход алгоритма подается АСД C конструкции циклической передачи потока управления подчиненным ЭМ из множества M (Рисунок 2.6). На выходе алгоритм

формирует АСД C' (Рисунок 2.15) конструкции, АСД P (Рисунок 2.14) функции $BodyAt$ и АСД H, представляющее функцию для создания локальных копий массивов в памяти подчиненных ЭМ. Алгоритм состоит из 9 шагов, первые 4 шага выполняют те же действия, что и алгоритм *By-Iterative Copying*. Оставшиеся 5 шагов выполняют:

1. Построение АСД H функции PROLOGUEBODY (Рисунок 2.12), выполняющей создание локальных копий массивов $a_0,...,a_{l-1}$ в памяти подчиненных ЭМ;
2. Поиск последовательности массивов $(a'_{l-1},...,a'_{l-1})$, созданных в локальной памяти подчиненных ЭМ;
3. Построение АСД S (Рисунок 2.13), соответствующего прологу цикла, который выполняет отправку сообщения подчиненным ЭМ для создания в их памяти локальных копий используемых массивов;
4. Построение АСД P функции BODYAT (Рисунок 2.14), соответствующей телу конструкции передачи управления подчиненным ЭМ;
5. Преобразование АСД B циклической конструкции передачи управления подчиненным ЭМ в АСД B' (Рисунок 2.15), содержащее узлы для отправки сообщения.

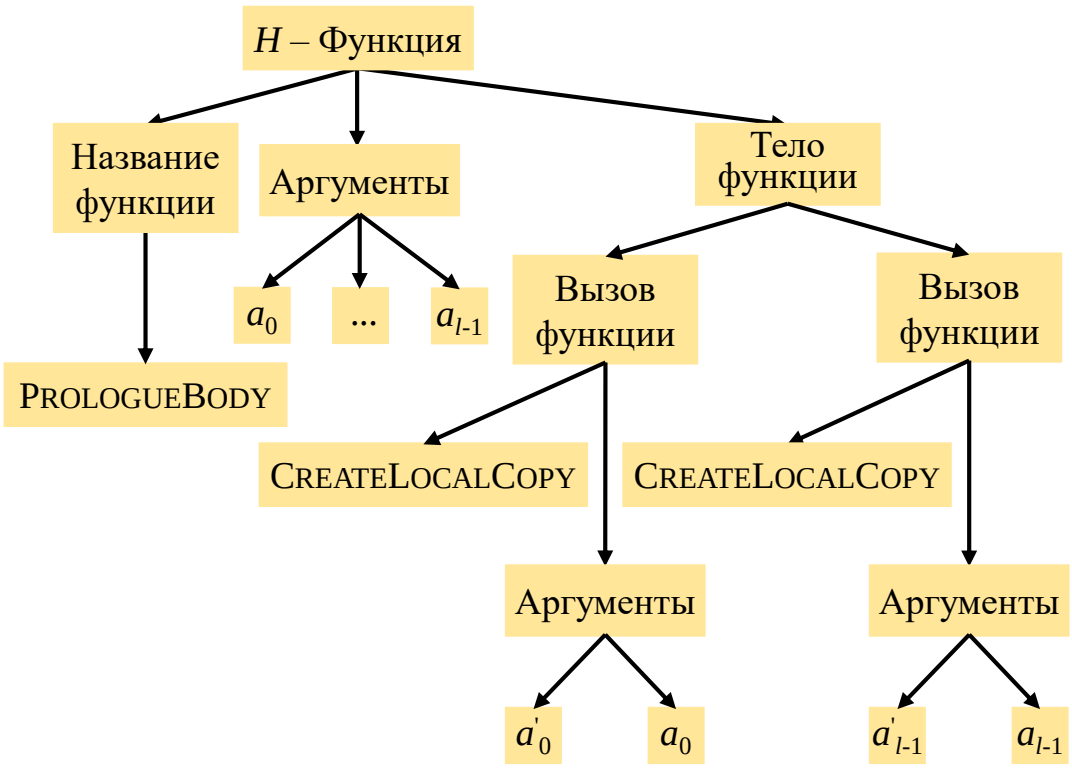


Рисунок 2.12 – Фрагмент АСД H функции $PrologueBody$ создания локальных копий массивов $a_0,...,a_{l-1}$ в памяти удаленных ЭМ

Алгоритм 7 Алгоритм *Array Preload* трансформации конструкций для выполнения опережающего копирования массивов

Input: C – АСД конструкции циклической
передачи потока управления подчиненным ЭМ;

Output: C' – модифицированное АСД конструкции циклической
передачи потока управления подчиненным ЭМ;
P – АСД функции $BodyAt$, представляющей тело конструкции at;
H – АСД функции $PrologueBody$ создания локальных копий
используемых массивов;

1: **function** ARRAYPRELOADTRANSFORMATION(C, C', P)
2: $B \leftarrow$ LOOKUPINNERLOOPBODY(C)
3: $r \leftarrow$ LOOKUPIDREMOTENODE(C)
4: $(a_0,...,a_{l-1}) \leftarrow$ LOOKUPARRAYS(B)
5: $(f_0,...,f_{l-1}) \leftarrow$ LOOKUPOPERATORS($B, (a_0,...,a_{l-1})$)
6: $H \leftarrow$ BUILDPROLOGUEBODY($(a_0,...,a_{l-1})$)
7: $(a'_0,...,a'_{l-1}) \leftarrow$ GETLOCALARRAYS(H)
8: $S \leftarrow$ BUILDLOOPPROLOGUE($(a_0,...,a_{l-1}),H$)
9: $P \leftarrow$ BUILDTREEBODYAT($(f_0,...,f_{l-1}), (a'_0,...,a'_{l-1})$)
10: $B' \leftarrow$ BUILDASTSENDMSG($B, (f_0,...,f_{l-1}), NULL, P, r$)
11: **end function**

Алгоритм 8 Алгоритм выполнения циклической конструкции передачи потока управления подчиненным ЭМ в Р-программе, полученной алгоритмом *Array Preload*

```
1: function PrologueBody((a_0,...,a_{l-1}))
2:     CreateLocalCopy(a'_0, a_0)
3:     ...
4:     CreateLocalCopy(a'_{l-1}, a_{l-1})
5: end function
6: function BodyAt()
7:     f_0(a'_0)
8:     ...
9:     f_{l-1}(a'_{l-1})
10: end function
11: ...
12: function Main()
13:     ...
14:     ▷ Циклическая конструкция Р-программы в модели передачи сообщений
15:     ▷ Пролог цикла
16:     for j in M do
17:         msg.type ← UserCodeMsg
18:         msg.data ← Serialize((a_0,...,a_{l-1}))
19:         msg.code ← Serialize(PrologueBody)
20:         Send(j, msg)
21:     done
22:     for i in {0,..,r − 1} do
23:         for j in M do
24:             msg.type ← UserCodeMsg
25:             msg.data ←
26:             msg.code ← Serialize(BodyAt)
27:             Send(j, msg)
28:         done
29:     done
30:     ▷ Конец конструкции
31:     ...
32: end function
```

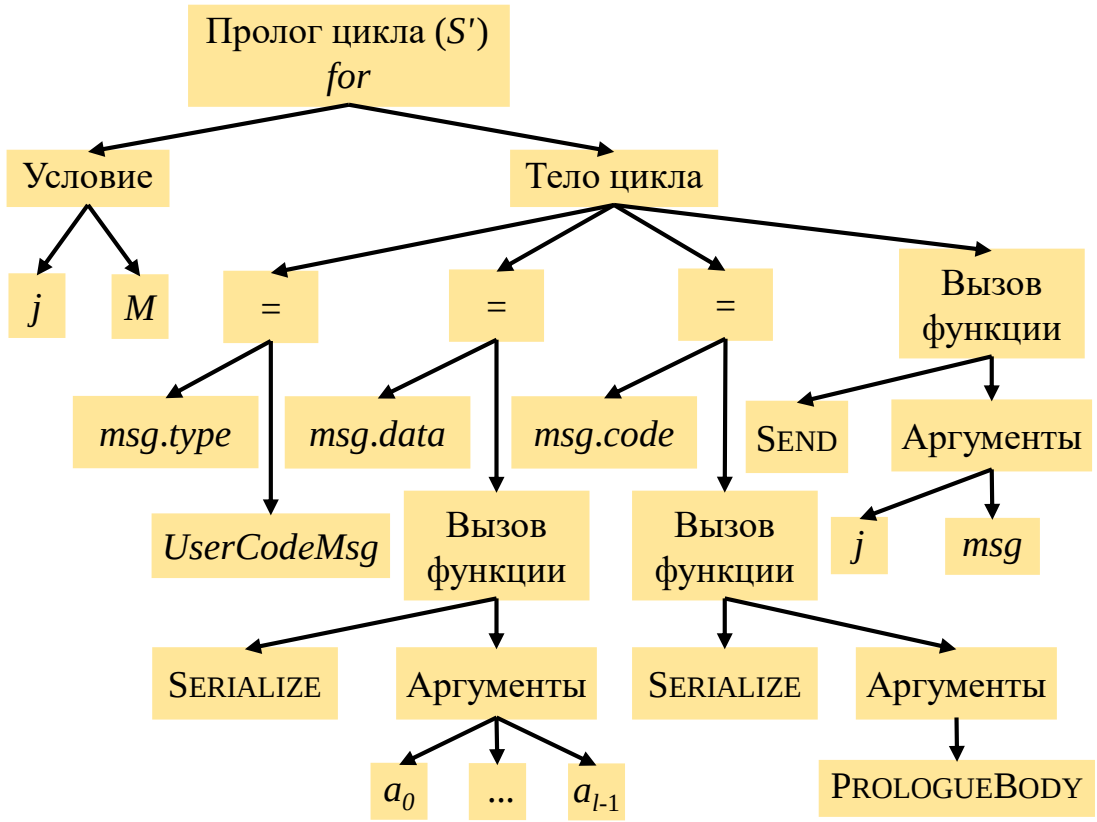


Рисунок 2.13 – Фрагмент АСД S, соответствующего прологу цикла для опережающего копирования массивов $a_0,...,a_{l-1}$ в память удаленных ЭМ

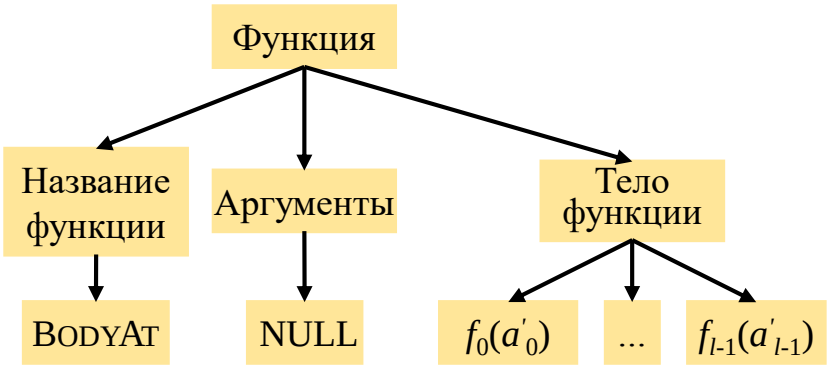


Рисунок 2.14 – Фрагмент АСД, которому соответствует функция BODYAT, сформированная алгоритмом *Array Preload*

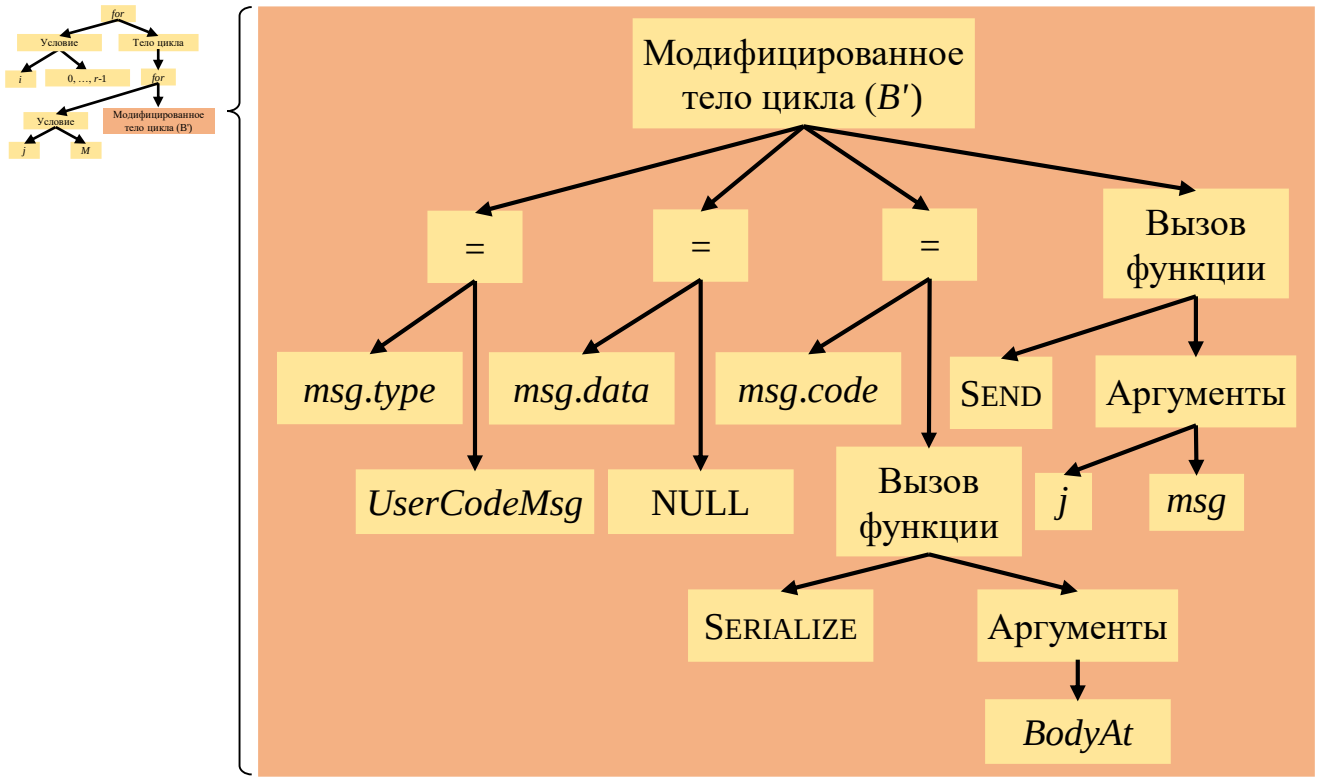


Рисунок 2.15 – Фрагмент АСД конструкции циклической передачи потока управления подчиненным ЭМ, полученный после трансформации алгоритмом *Array Preload*

2.7 Теоретический анализ эффективности алгоритмов

Описанные выше алгоритмы трансформации конструкций циклической передачи потока управления подчиненным ЭМ из множества M преобразуют высокоуровневые конструкции PGAS-программы в последовательность вызовов функций runtime-системы P-программы. В этом разделе выполнен анализ эффективности полученного в результате применения описанных алгоритмов кода P-программы для доставки используемых массивов подчиненным ЭМ. Кроме этого, для каждого алгоритма построены оценки времени выполнения в моделях Hockney [49], LogP [25] и LogGP [36].

Алгоритм *By-Iterative Copying* преобразует конструкцию циклической передачи потока управления подчиненным ЭМ из множества M таким образом, что при выполнении P-программы на каждой итерации конструкции каждой подчиненной ЭМ будут передаваться все используемые массивы a_0,...,a_{l-1}. В этом случае главной ЭМ на каждой итерации потребуется отправить V байт:

$$V = m \cdot \sum_{i=0}^{l-1} s_i \cdot b,$$

где b – размер одного элемента массива в байтах, а s_i – количество элементов в массиве a_i.

Эффективность преобразований, выполняемых алгоритмом *By-Iterative Copying*, определяется временем, затраченным на доставку используемых массивов в память подчиненных ЭМ при выполнении P-программы. Пространственно-временная диаграмма информационных обменов в модели LogP и LogGP при выполнении P-программы представлена на Рисунках 2.16, 2.17. Время t выполнения конструкции в модели LogP, LogGP и Hockney (содержательный смысл параметров моделей см. в Разделе 1.3) будет равно:

$$t_{LogP} = g \cdot m \cdot r - g + L + 2 \cdot o;$$

$$t_{LogGP} = ((\sum_{i=0}^{l-1} s_i \cdot b - 1) \cdot G + g) \cdot m \cdot r - g + L + 2 \cdot o =$$

$$= ((\frac{V}{m} - 1) \cdot G + g) \cdot m \cdot r - g + L + 2 \cdot o;$$

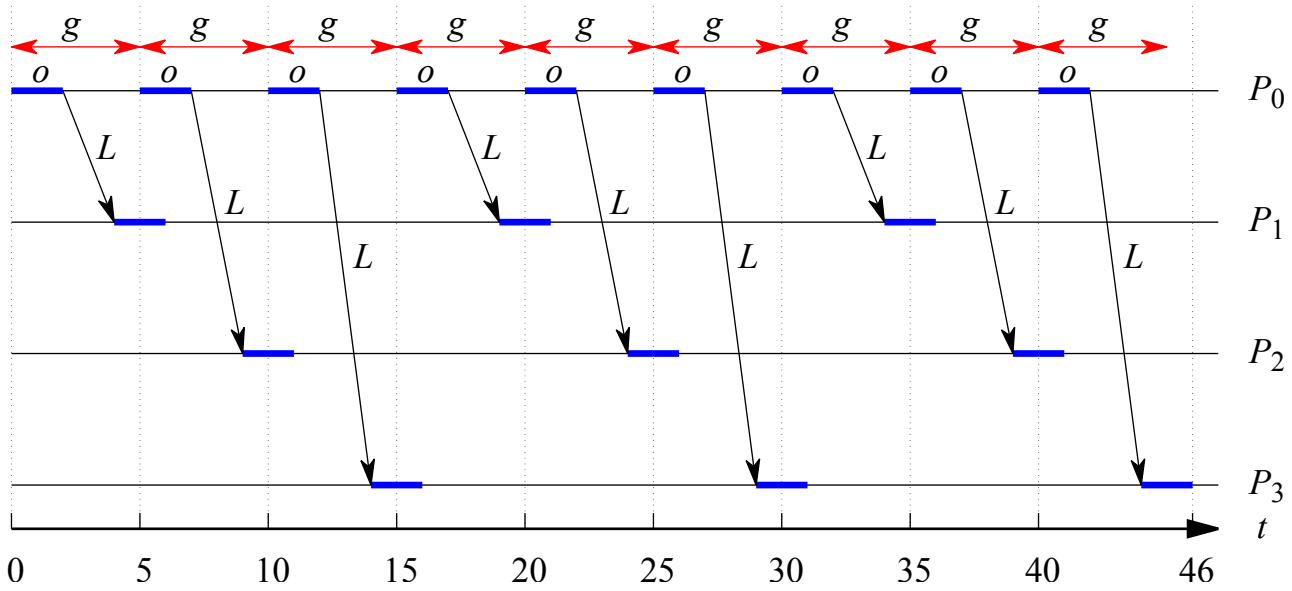


Рисунок 2.16 – Пространственно-временная диаграмма информационных обменов в модели LogP при выполнении циклической конструкции передачи потока управления подчиненным ЭМ, преобразованной алгоритмом *By-Iterative Copying* ($P = 4$)

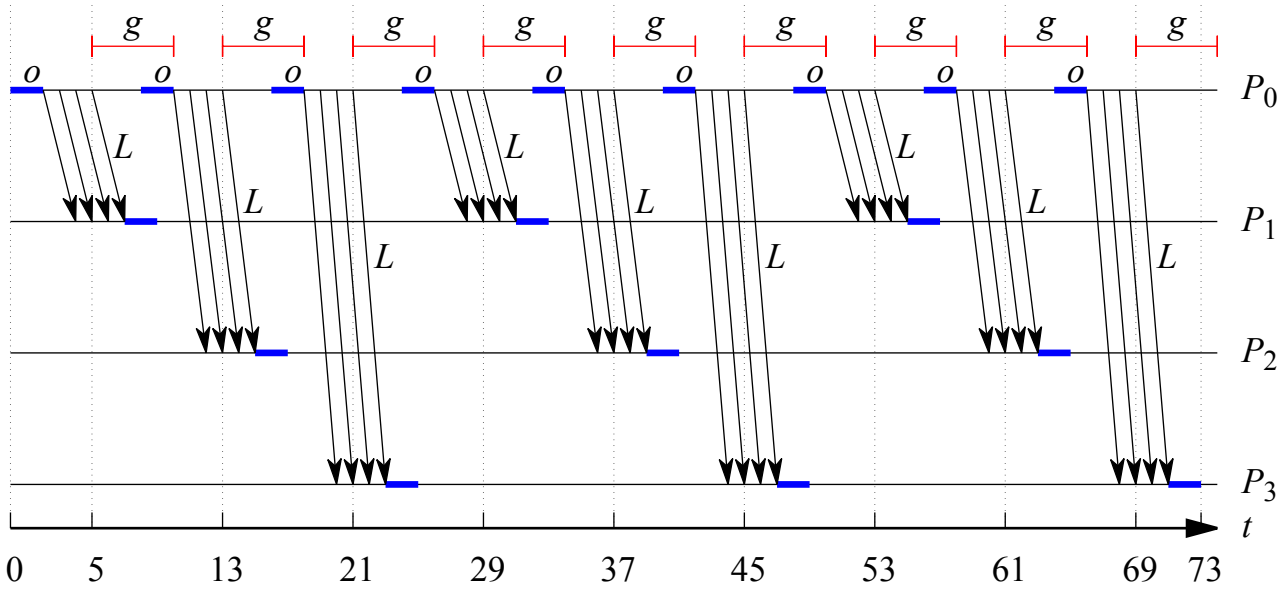


Рисунок 2.17 – Пространственно-временная диаграмма информационных обменов в модели LogGP при выполнении циклической конструкции передачи потока управления подчиненным ЭМ, преобразованной алгоритмом *By-Iterative Copying* ($P = 4$)

$$t_{Hockney} = r \cdot m \cdot (\alpha + \beta \cdot \sum_{i=0}^{l-1} s_i \cdot b) = r \cdot m \cdot (\alpha + \beta \cdot \frac{V}{m}).$$

Алгоритм *Scalar Replacement* выполняет преобразование конструкции циклической передачи потока управления подчиненным ЭМ таким образом,

чтобы во время ее выполнения в Р-программе главная ЭМ на каждой итерации отправляла только используемые элементы массивов. В этом случае главной ЭМ на каждой итерации потребуется отправить V байт:

$$V = m \cdot q \cdot b,$$

где b – размер одного элемента массива в байтах. А время t выполнения конструкции в моделях LogP, LogGP и Hockney будет равно:

$$t_{LogP} = g \cdot m \cdot r - g + L + 2 \cdot o;$$

$$t_{LogGP} = ((q \cdot b - 1) \cdot G + g) \cdot m \cdot r - g + L + 2 \cdot o =$$
$$= ((\frac{V}{m} - 1) \cdot G + g) \cdot m \cdot r - g + L + 2 \cdot o;$$
$$t_{Hockney} = r \cdot m \cdot (\alpha + \beta \cdot q \cdot b) =$$
$$= r \cdot m \cdot (\alpha + \beta \cdot \frac{V}{m}).$$

При выполнении циклических конструкций передачи потока управления подчиненным ЭМ в Р-программе, формируемой с применением алгоритма *Array Preload*, копирование массивов a_0,...,a_{l-1} в удаленную память ЭМ происходит только один раз перед итерациями в прологе цикла. Поэтому главная ЭМ на каждой итерации отправляет V байт:

$$V = m \cdot \sum_{i=0}^{l-1} s_i \cdot b,$$

где b – размер одного элемента массива в байтах. Время t выполнения пролога цикла и самой конструкции в моделях LogP, LogGP и Hockney будет иметь следующий вид:

$$t_{LogP} = g \cdot m - g + L + 2 \cdot o;$$

$$t_{LogGP} = (((\sum_{i=0}^{l-1} s_i \cdot b) - 1) \cdot G + g) \cdot m - g + L + 2 \cdot o =$$
$$= ((\frac{V}{m} - 1) \cdot G + g) \cdot m - g + L + 2 \cdot o;$$
$$t_{Hockney} = m \cdot (\alpha + \beta \cdot \sum_{i=0}^{l-1} s_i \cdot b) =$$

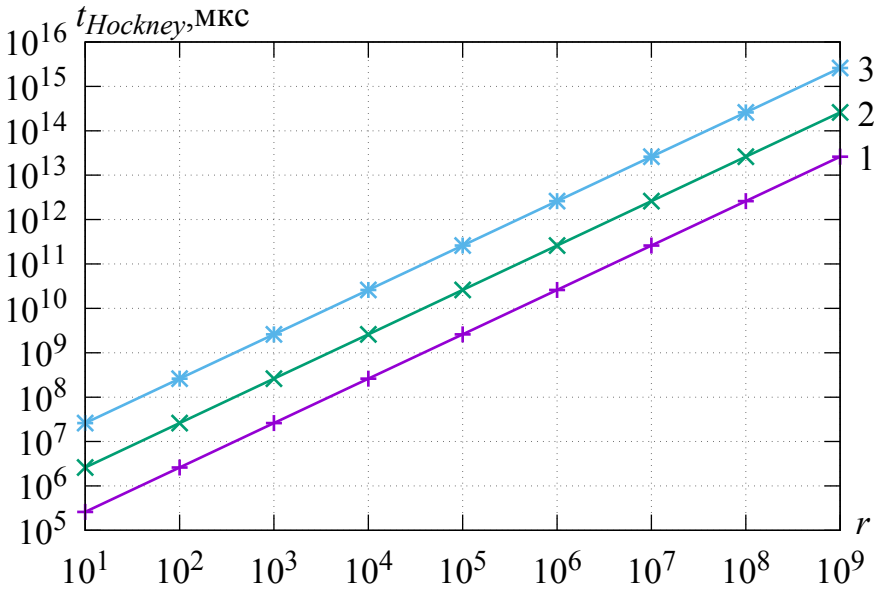
$$= m \cdot (\alpha + \beta \cdot \frac{V}{m}).$$

Выполнен анализ эффективности алгоритмов *By-Iterative Copying*, *Scalar Replacement* и *Array Preload* преобразования циклических конструкций передачи потока управления подчиненным ЭМ в модели Hockney. В процессе анализа варьировались такие параметры как m – количество подчиненных ЭМ, r – число итераций в циклической конструкции, s – размер используемого подчиненными ЭМ массива. Параметры, описывающие ВС в модели Hockney: α – задержка при передачи сообщения и β – время передачи одного байта сообщения, были выбраны, как для коммуникационной технологии Infiniband QDR ($\alpha = 1$ мкс, $\beta = 2 \cdot 10^{-4}$ мкс). Предполагалось, что параллельная PGAS-программа содержит одну циклическую конструкцию передачи потока, в теле которой подчиненные ЭМ использовали только 1 элемент массива с типом `double` ($b = 8$ байт). Ниже представлены результаты анализа.

На Рисунках 2.18, 2.19, 2.20 показана зависимость времени выполнения циклической конструкции в P-программе, полученной алгоритмами *By-Iterative Copying*, *Scalar Replacement* и *Array Preload* соответственно, от числа r итераций при различных значениях m – количества подчиненных ЭМ. Алгоритмы *By-Iterative Copying* и *Scalar Replacement* демонстрируют линейную зависимость времени выполнения конструкции от числа r итераций в цикле, в то время как алгоритм *Array Preload* – константную.

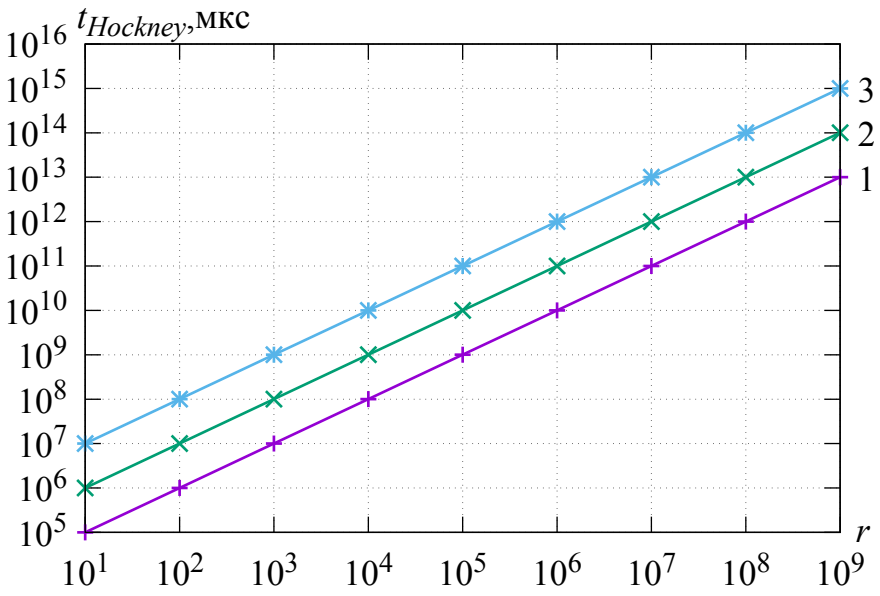
На Рисунках 2.21, 2.22 изображены графики, демонстрирующие зависимость времени выполнения циклической конструкции передачи потока управления в P-программе от числа m подчиненных ЭМ, которым передается управление при различных значениях количества итераций r в цикле. Все три алгоритма показали линейную зависимость времени выполнения трансформированной конструкции от количества m подчиненных ЭМ, которым передается управление. На Рисунке 2.22 видно, что код, генерируемый алгоритмом *Array Preload*, обладает меньшим временем выполнения, чем код, полученный алгоритмами *By-Iterative Copying* и *Scalar Replacement*.

На Рисунках 2.23, 2.24, 2.25 показана зависимость времени выполнения кода циклической конструкции передачи потока управления подчиненным ЭМ в P-программе, полученной разными алгоритмами трансформации, от размера s массива, доступ к которому осуществляется подчиненными ЭМ.



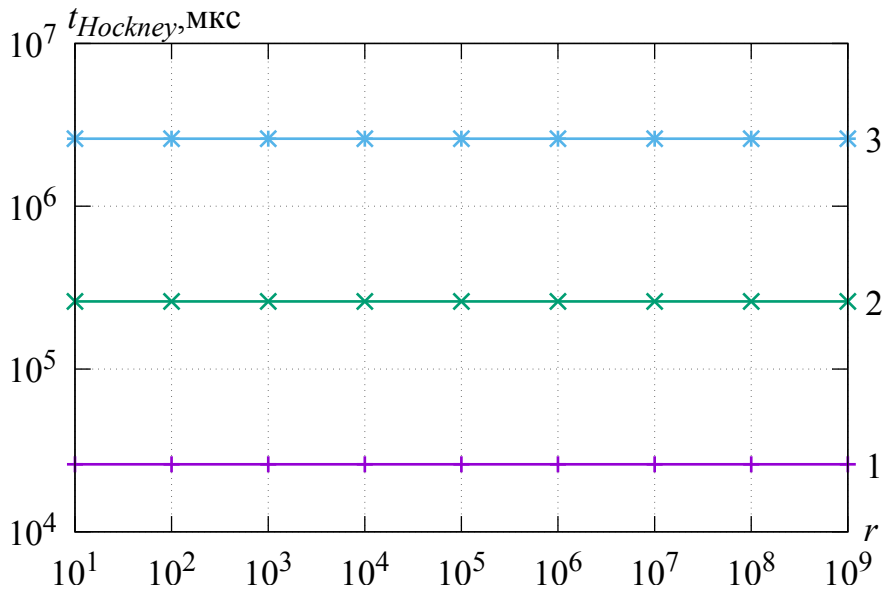
1 – $m = 10^4$; 2 – $m = 10^5$; 3 – $m = 10^6$;

Рисунок 2.18 – Зависимость времени выполнения копирования массивов в память подчиненных ЭМ при выполнении Р-программы, полученной алгоритмом *By-Iterative Copying*, от числа r итераций в циклической конструкции передачи потока управления ($\alpha = 1$ мкс, $\beta = 2 \cdot 10^{-4}$ мкс, $b = 8$ байт)



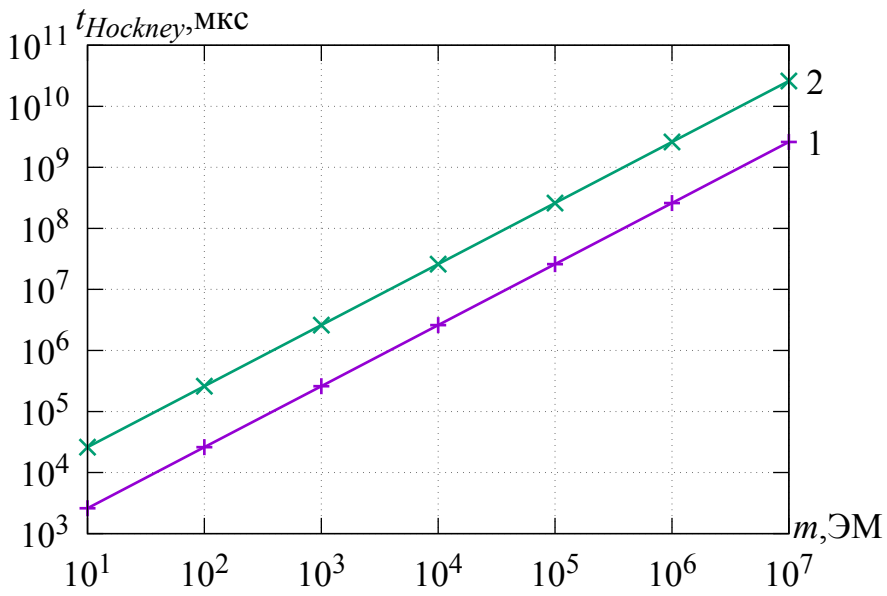
1 – $m = 10^4$; 2 – $m = 10^5$; 3 – $m = 10^6$;

Рисунок 2.19 – Зависимость времени выполнения копирования массивов в память подчиненных ЭМ при выполнении Р-программы, полученной алгоритмом *Scalar Replacement*, от числа r итераций в циклической конструкции передачи потока управления ($\alpha = 1$ мкс, $\beta = 2 \cdot 10^{-4}$ мкс, $b = 8$ байт)



1 – $m = 10^4$; 2 – $m = 10^5$; 3 – $m = 10^6$;

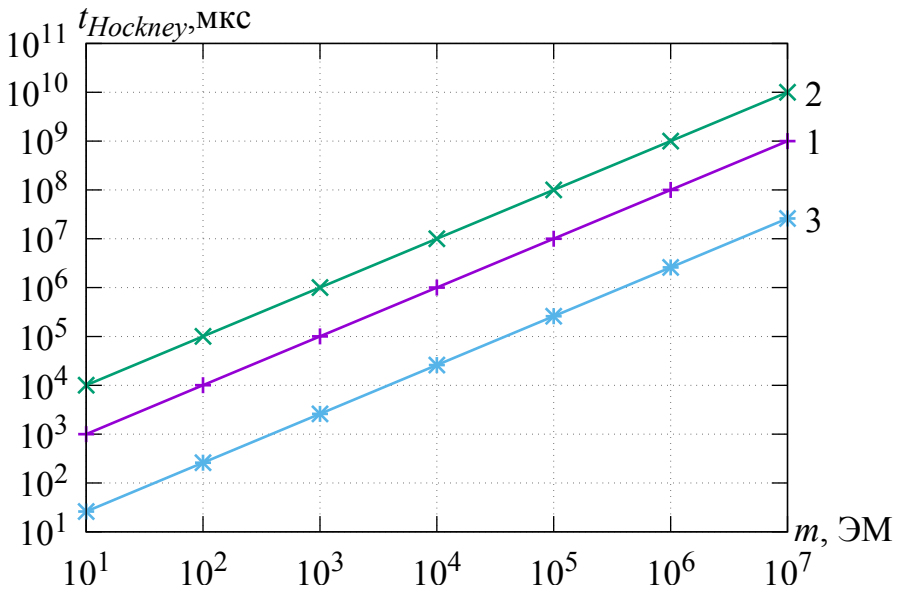
Рисунок 2.20 – Зависимость времени выполнения копирования массивов в память подчиненных ЭМ при выполнении Р-программы, полученной алгоритмом *Array Preload*, от числа r итераций в циклической конструкции передачи потока управления ($\alpha = 1$ мкс, $\beta = 2 \cdot 10^{-4}$ мкс, $b = 8$ байт)



1 – $r = 10^2$; 2 – $r = 10^3$;

Рисунок 2.21 – Зависимость времени выполнения копирования массивов в память подчиненных ЭМ при выполнении Р-программы, полученной алгоритмом *By-Iterative Copying*, от количества m подчиненных ЭМ, которым передается управление ($\alpha = 1$ мкс, $\beta = 2 \cdot 10^{-4}$ мкс, $b = 8$ байт)

На Рисунке 2.24 видно, что алгоритм *Scalar Replacement* позволяет сгенерировать код для циклической конструкции в Р-программе, время выполнения



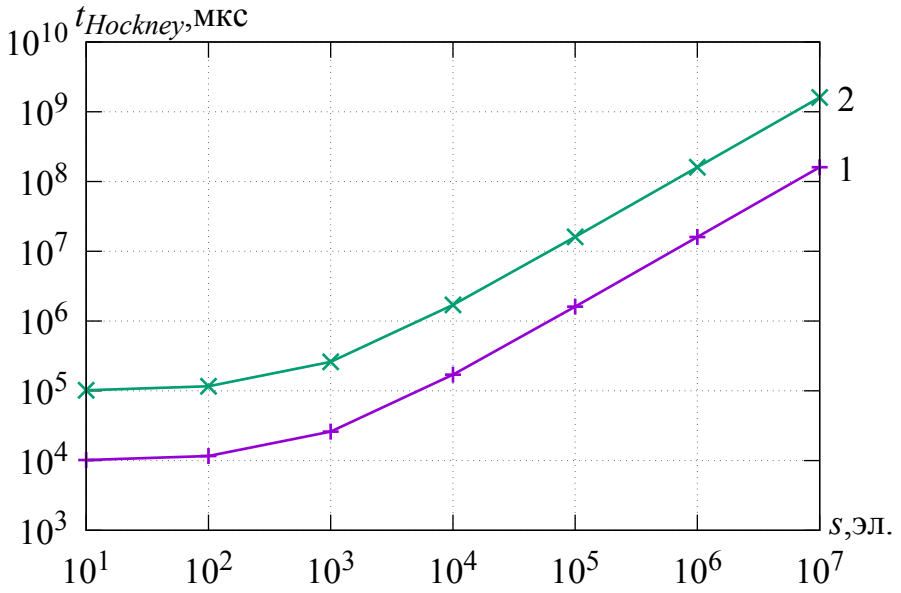
1 – *Scalar Replacement* $r = 10^2$; 2 – *Scalar Replacement* $r = 10^3$;
3 – *Array Preload* $r = 10^3$

Рисунок 2.22 – Зависимость времени выполнения копирования массивов в память подчиненных ЭМ при выполнении Р-программы, полученной алгоритмами *Scalar Replacement* – 1,2 и *Array Preload* – 3, от количества m подчиненных ЭМ, которым передается управление ($\alpha = 1$ мкс, $\beta = 2 \cdot 10^{-4}$ мкс, $b = 8$ байт)

которого не зависит от размера s используемого подчиненными ЭМ массива. Однако, по Рисунку 2.25 можно установить, что в случае, если массив, элементы которого используются подчиненными ЭМ, имеет размер $10^1 \leqslant s \leqslant 10^6$, то рекомендуется использовать алгоритм *Array Preload*. Так как в этом случае, время выполнения кода, полученного алгоритмом *Array Preload*, меньше, чем кода, полученного алгоритмам *Scalar Replacement* и *By-Iterative Copying*.

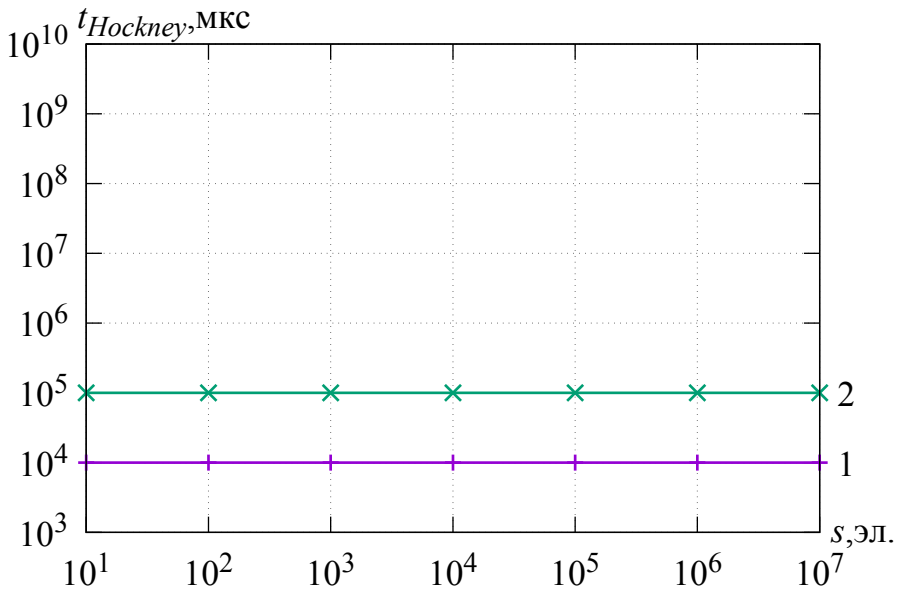
2.8 Экспериментальное исследование эффективности алгоритмов

Экспериментальное исследование алгоритмов проводилось на вычислительных кластерах Jet (ЭМ на базе двух четырехъядерных процессоров Intel Xeon E5420, соединенных сетью Gigabit Ethernet) и Oak (ЭМ на базе двух четырехъядерных процессоров Intel Xeon E5420, соединенных сетью



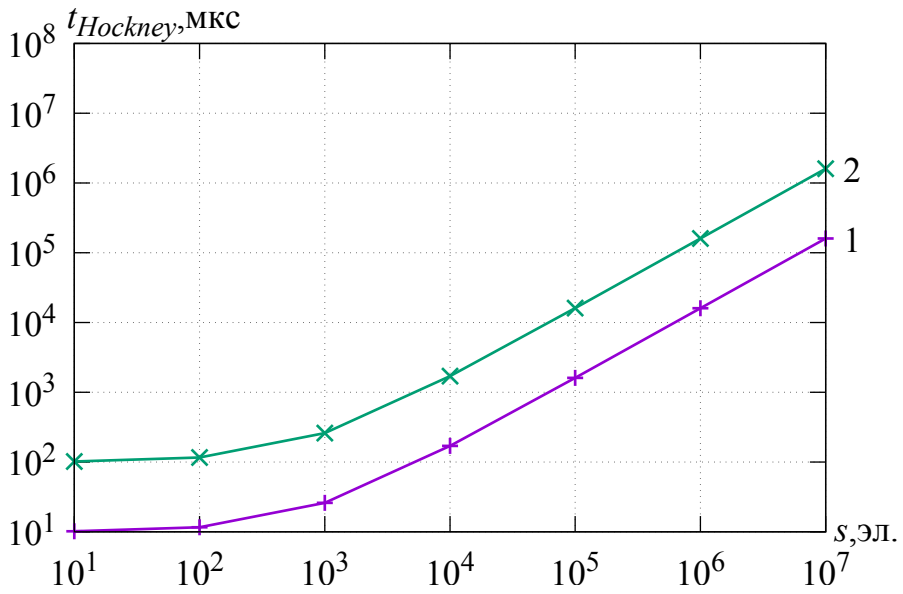
1 – $m = 10^1$; 2 – $m = 10^2$;

Рисунок 2.23 – Зависимость времени выполнения копирования массивов в память подчиненных ЭМ при выполнении Р-программы, полученной алгоритмом *By-Iterative Copying*, от размера s массива, доступ к которому осуществляется подчиненными ЭМ ($\alpha = 1$ мкс, $\beta = 2 \cdot 10^{-4}$ мкс, $b = 8$ байт, $r = 10^3$ итераций)



1 – $m = 10^1$; 2 – $m = 10^2$;

Рисунок 2.24 – Зависимость времени выполнения копирования массивов в память подчиненных ЭМ при выполнении Р-программы, полученной алгоритмом *Scalar Replacement*, от размера s массива, доступ к которому осуществляется подчиненными ЭМ ($\alpha = 1$ мкс, $\beta = 2 \cdot 10^{-4}$ мкс, $b = 8$ байт, $r = 10^3$ итераций)



1 – $m = 10^1$; 2 – $m = 10^2$;

Рисунок 2.25 – Зависимость времени выполнения копирования массивов в память подчиненных ЭМ при выполнении Р-программы, полученной алгоритмом *Array Preload*, от размера s массива, доступ к которому осуществляется подчиненными ЭМ ($\alpha = 1$ мкс, $\beta = 2 \cdot 10^{-4}$ мкс, $b = 8$ байт, $r = 10^3$ итераций)

InfiniBand QDR) Центра параллельных вычислительных технологий СибГУТИ и Института физики полупроводников им. А.В. Ржанова СО РАН. Предложенный алгоритм *Array Preload* программно реализован для языка IBM X10 [100].

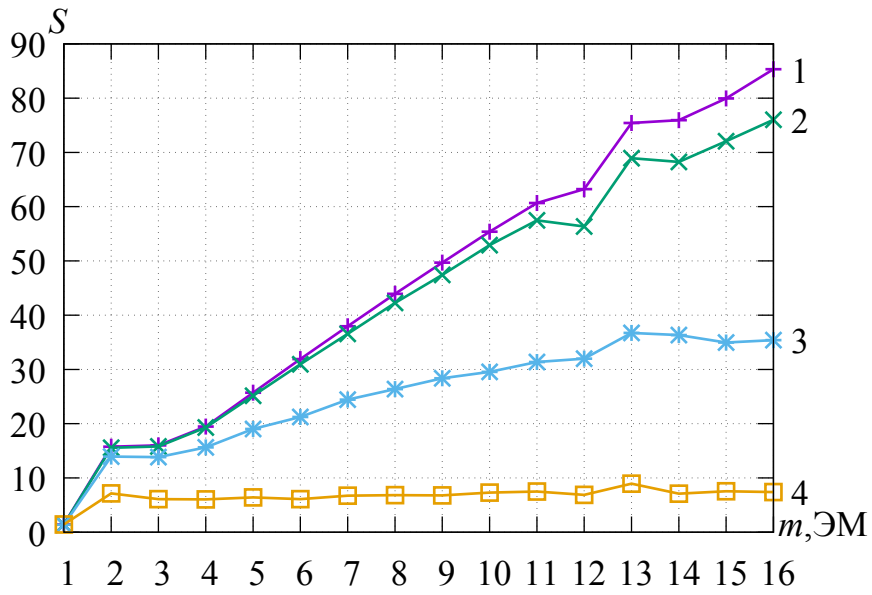
Для оценки эффективности алгоритмов *By-Iterative Copying*, *Scalar Replacement* и *Array Preload* использовался синтетический тест – циклический доступ к одному элементу массива типа `double` ($b = 8$), расположенному в памяти главной ЭМ. В этом тесте присутствовала одна циклическая конструкция передачи потока управления подчиненным ЭМ из множества M, в теле которой выполнялось обращение к одному элементу массива типа `double` ($b = 8$), хранимого в памяти главной ЭМ. Длина s массива в памяти главной ЭМ, число r итераций цикла, а также количество $m = |M|$ подчиненных ЭМ варьировалось в ходе экспериментов. Компилятор IBM X10 был собран с библиотеками MPICH2 3.0.4 (Jet) и MVAPICH2 2.0 (Oak).

На Рисунках 2.26, 2.28 2.27, 2.29 показана зависимость коэффициента ускорения выполнения синтетического теста на ВС с сетью связи Gigabit

Ethernet, трансформированного алгоритмами *Scalar Replacement* и *Array Preload* от числа m подчиненных ЭМ при различных значениях размера массива s и числа итераций r.

На Рисунках 2.30, 2.31 представлены графики зависимости коэффициента ускорения выполнения тестовой программы на ВС с сетью связи InfiniBand QDR, полученной с применением алгоритма *Array Preload*, от числа m подчиненных ЭМ при варьирующихся значениях размера массива s (Рисунок 2.31) и числа итераций r (Рисунок 2.30).

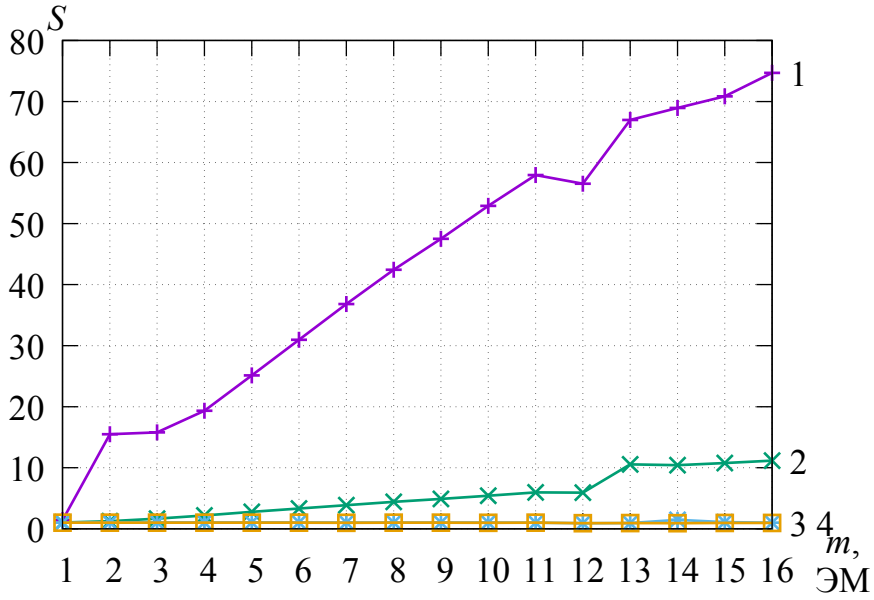
Алгоритм *Array Preload*, по сравнению со стандартным алгоритмом, позволяет сократить время выполнения циклического доступа ко всем элементам массивов в 1,2–2,5 раза на кластерных ВС с сетями связи Gigabit Ethernet и InfiniBand QDR.



1 – $s = 40000$, $r = 40000$; 2 – $s = 40000$, $r = 4000$;
3 – $s = 40000$, $r = 400$; 4 – $s = 40000$, $r = 40$;

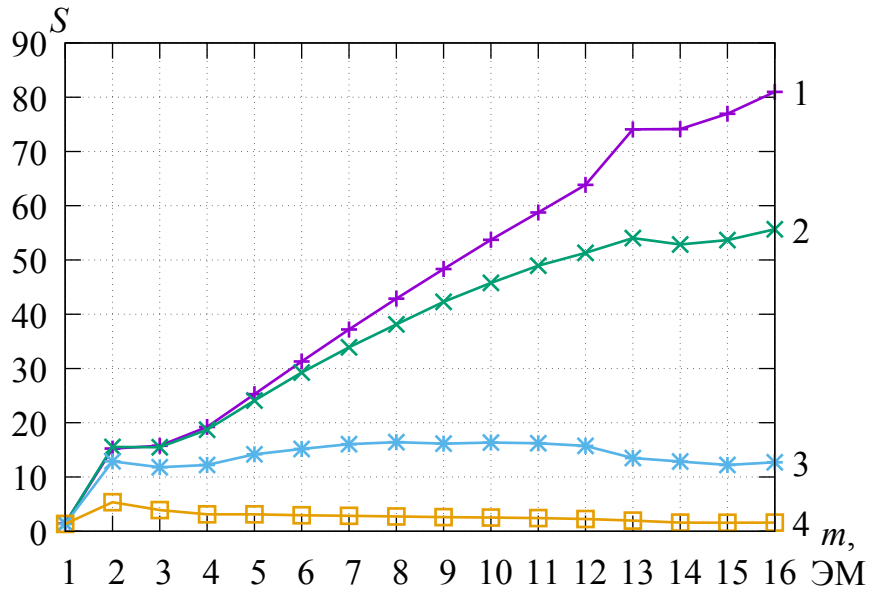
Рисунок 2.26 – Ускорение тестовой программы на языке IBM X10, скомпилированной с применением алгоритма *Scalar Replacement* (кластер Jet)

На Рисунках 2.30, 2.31 показаны графики зависимости значения коэффициента ускорения выполнения X10-теста от количества m ЭМ после применения алгоритма *Array Preload* на кластере Oak с сетью связи InfiniBand QDR. В ходе экспериментов варьировалось количество r итераций в циклической конструкции и размер s массива, к элементам которого подчиненные ЭМ осуществляли доступ. В общем случае ускорение зависит от



1 – $s = 40000$, $r = 4000$; 2 – $s = 4000$, $r = 4000$;
3 – $s = 400$, $r = 4000$; 4 – $s = 40$, $r = 4000$;

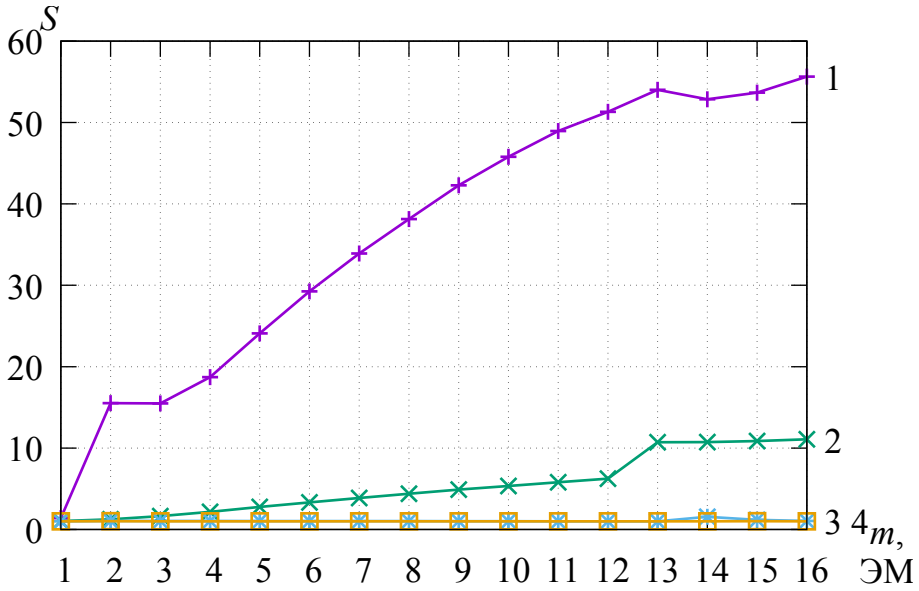
Рисунок 2.27 – Ускорение тестовой программы на языке IBM X10, скомпилированной с применением алгоритма *Scalar Replacement* (кластер Jet)



1 – $s = 40000$, $r = 40000$; 2 – $s = 40000$, $r = 4000$;
3 – $s = 40000$, $r = 400$; 4 – $s = 40000$, $r = 40$;

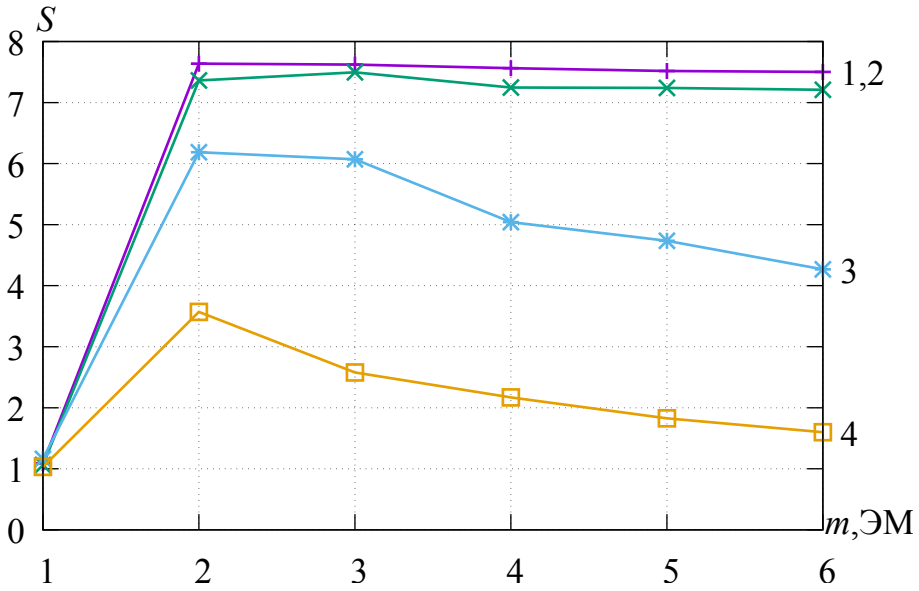
Рисунок 2.28 – Ускорение тестовой программы на языке IBM X10, скомпилированной с применением алгоритма *Array Preload* (кластер Jet)

производительности коммуникационной сети, количества подчиненных ЭМ, размера массива, количества итераций в цикле (в теле которого организован циклический доступ).



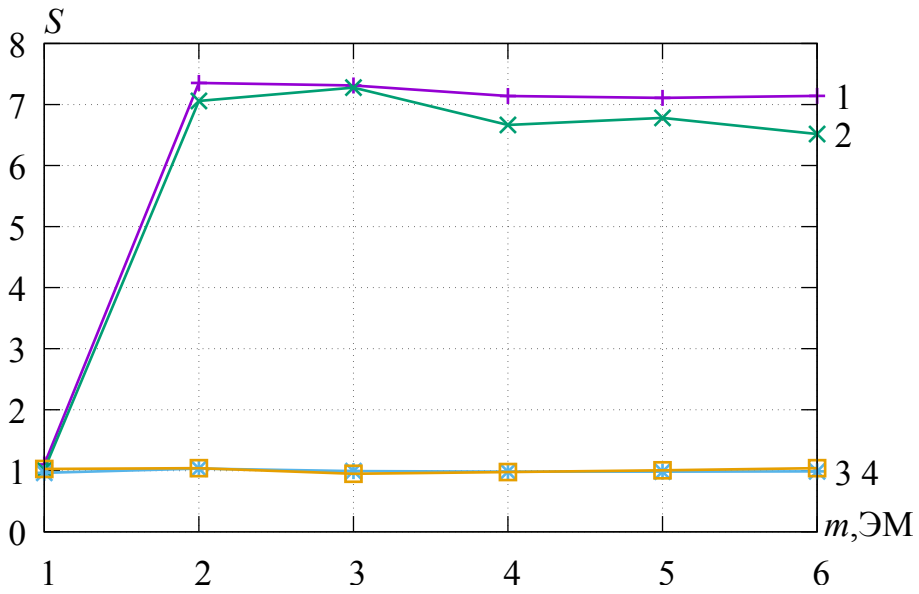
1 – $s = 40000$, $r = 4000$; 2 – $s = 4000$, $r = 4000$;
3 – $s = 400$, $r = 4000$; 4 – $s = 40$, $r = 4000$;

Рисунок 2.29 – Ускорение тестовой программы на языке IBM X10, скомпилированной с применением алгоритма *Array Preload* (кластер Jet)



1 – $s = 40000$, $r = 40000$; 2 – $s = 40000$, $r = 4000$;
3 – $s = 40000$, $r = 400$; 4 – $s = 40000$, $r = 40$;

Рисунок 2.30 – Ускорение тестовой программы на языке IBM X10, скомпилированной с применением алгоритма *Array Preload* (кластер Oak)



1 – $s = 40000$, $r = 4000$; 2 – $s = 4000$, $r = 4000$;
3 – $s = 400$, $r = 4000$; 4 – $s = 40$, $r = 4000$;

Рисунок 2.31 – Ускорение тестовой программы на языке IBM X10, скомпилированной с применением алгоритма *Array Preload* (кластер Oak)

2.9 Выводы

1. Предложен алгоритм *Array Preload* трансформации конструкций циклической передачи потока управления подчиненным элементарным машинам, сокращающий время выполнения программ за счет опережающего копирования информационных массивов. В отличие от известных, разработанный метод применим к PGAS-программам, в которых на этапе компиляции неизвестно множество используемых элементов массивов.
2. В моделях параллельных вычислений LogP, LogGP и Hockney построены оценки эффективности выполнения формируемого алгоритмом *Array Preload* кода, показывающие отсутствие функциональной зависимости времени его выполнения от количества итераций циклов.
3. Выполнена реализация алгоритма в виде расширения компилятора языка IBM X10. По сравнению со стандартным алгоритмом предложенный позволяет на кластерных ВС с сетями связи Gigabit Ethernet

и InfiniBand QDR сократить время выполнения циклического доступа к элементам массивов в 1.2–2.5 раз.

Глава 3. Оптимизация выполнения программ на многопроцессорных ВС с общей памятью

В данной главе рассматриваются методы оптимизации выполнения параллельных программ на многопроцессорных ВС с общей памятью. Рассматриваемые методы ориентированы на повышение эффективности синхронизации потоков параллельных программ и оптимизацию выполнения циклических конструкций, в частности, конвейеризацию и векторизацию циклов. Описан алгоритм оптимизации обнаружения конфликтов при выполнении синхронизации потоков с помощью программной транзакционной памяти. Алгоритм основан на предварительном профилировании параллельной программы. Для профилирования предложен инструментарий, позволяющий получать информацию о динамических свойствах транзакционных секций программы.

3.1 Обзор методов синхронизации на ВС с общей памятью

3.1.1 Понятие состояния гонки за данными

При выполнении многопоточной программы на ресурсах многопроцессорной ВС с общей памятью может возникнуть *состояние гонки за данными* (*data race*) – ситуация, при которой множество потоков пытаются одновременно получить доступ к разделяемому ресурсу, к одной области памяти. Возникновение состояния гонки за данными является результатом ошибки при разработке параллельного алгоритма и приводит к некорректной работе программы [24].

Пусть имеются два потока A и B, которые одновременно выполняют код C_A и C_B. В коде выполняется операция чтения над ячейками памяти из множества $R(A)$ – для потока A и $R(B)$ – для потока B. Со считанными из ячеек памяти значениями выполняются арифметические и логические операции, результат которых при помощи операции записи сохраняется во

множество ячеек памяти $W(A)$ – для потока A и $W(B)$ – для потока B. Таким образом, множество $\{R(A) \cup W(A)\}$ есть множество используемых потоком A ячеек памяти в коде C_A, а множество $\{R(B) \cup W(B)\}$ – множество используемых потоком B ячеек памяти в коде C_B.

В этом случае *состояние гонки за данными* возникает, если не выполняется хотя бы одно из условий Бернстайна [18]

1. $R(A) \cap W(B) = \varnothing$
2. $W(A) \cap R(B) = \varnothing$
3. $W(A) \cap W(B) = \varnothing$

Большинство современных языков программирования имеют возможность создания *потоков* (*threads*), тем самым реализуя модель многопоточного программирования. В случае, если два или более потоков одновременно будут обращаться к данным, хранимым в одной и той же ячейке памяти, и как минимум один из них будет выполнять операцию записи, возникает состояние гонки за данными. В Листинге 3.1 представлен фрагмент OpenMP-программы на языке C, в которой возникает состояние гонки за данными. Этот фрагмент кода содержит распараллеленный при помощи OpenMP-директив цикл из `n` итераций. В теле цикла накапливается сумма `summ` значений `a`, возвращаемых функцией `func(i)`. Так как доступ к общей переменной `summ` выполняется одновременно несколькими потоками, то выполнение данного кода приведет к возникновению состояния гонки за данными.

Листинг 3.1: Фрагмент OpenMP-программы, приводящей к возникновению состояния гонки за данными

```
1 int summ = 0;
2 int i;
3 int n = omp_get_thread_num() * 1000;
4
5 #pragma omp parallel for shared(summ)
6 for (i = 0; i < n; i++) {
7     int a = func(i);
8     summ += a;
9 }
```

Возникновение состояний гонки за данными является одним из негативных явлений многопоточной модели параллельного программирования.

Для их предотвращения используются следующие подходы к созданию потокобезопасных программ:

- синхронизация потоков при помощи операций блокировок [12];
- использование не блокирующих алгоритмов и структур данных [9];
- использование транзакционной памяти [52, 99].

Кроме исследований в области разработки средств создания потокобезопасных программ, в мире активно ведутся работы по разработке средств автоматического обнаружения гонок [6, 14].

3.1.2 Синхронизация потоков при помощи операций блокировок

Для предотвращения возникновения состояния гонок за данными необходимо синхронизировать доступ потоков к общей области памяти. Один из активно используемых подходов к синхронизации потоков заключается в использовании операций блокировок [24]. Этот подход позволяет создавать в коде программы *критические секции* – участки кода, выполнение которых возможно только одним потоком в каждый момент времени. Последовательное выполнение критических секций достигается при помощи механизма *взаимного исключения* (*mutual exclusion*).

В основе механизма взаимного исключения лежат две операции `lock` и `unlock`, выполняемые над специальным объектом – *мьютексом* (*mutex*). Операция `lock` выполняется при входе в критическую секцию и помечает мьютекс как «*занят*», а операция `unlock` – при выходе и помечает мьютекс как «*свободен*». Операцию `lock` принято называть захватом мьютекса, а операцию `unlock` – освобождением.

Пусть имеется поток A и поток B, а также логические переменные F и S, сигнализирующие о выполнении критической секции C. Если поток A выполняет критическую секцию C, то логическая переменная F принимает значение *ИСТИНА*, а если критическую секцию выполняет поток B, то значение *ИСТИНА* принимает логическая переменная S. Механизм взаимного исключения в любой момент времени для критической секции C обеспечивает выполнение следующего условия:

$$\neg(A \wedge B) = \textit{ИСТИНА} \tag{3.1}$$

Листинг 3.2: Фрагмент программы, использующей механизм взаимного исключения для создания критической секции

```
1  int summ = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  /* Код, выполняемый множеством потоков. */
4  void *thread_body(void *)
5  {
6      int i;
7      ...
8      for (i = begin; i < end; i++) {
9          int a = func(i);
10         pthread_mutex_lock(&m)
11         summ += a;
12         pthread_mutex_unlock(&m)
13     }
14     return NULL;
15 }
```

В Листинге 3.2 приведен пример создания критической секции при помощи механизма взаимного исключения. В этом примере функция `thread_body` выполняется множеством потоков. Каждый поток выполняет цикл со своим пространством итераций, в теле которого накапливается сумма возвращаемого значения функции `func`. Доступ к разделяемой переменной `summ` осуществляется внутри критической секции, организованной механизмом взаимного исключения при помощи мьютекса `m`. При входе в критическую секцию, для захвата мьютекса, поток должен выполнить функцию `pthread_mutex_lock`, а при выходе из критической секции, для освобождения мьютекса, – `pthread_mutex_unlock`. Эти функции предоставляются библиотекой стандарта POSIX *Pthreads*, реализующей операции по созданию и управлению потоками.

Операции `lock` и `unlock` должны обеспечивать выполнение условия 3.1 и реализуются на основе атомарной операции *сравнение с обменом* (*compare and swap – CAS*). В Алгоритме 9 приведена логика выполнения операции CAS, выполняющей запись значения третьего аргумента в первый аргумент в случае, если значение второго аргумента и первого равны. В противном случае во второй аргумент записывается значение первого

аргумента. Атомарная операция CAS предоставляется архитектурой набора команд большинства процессоров, например, архитектуры IA-32/Intel 64 реализуют команду сравнение с обменом – `cmpxchg`, которая совместно с префиксом `lock` (`lock cmpxchg`) выполняется атомарно.

Алгоритм 9 Алгоритм операции `CAS`

```
1: function CAS(val, expected, new)
2:     if val == expected then
3:         val ← new
4:         return true
5:     end if
6:     expected ← val
7:     return false
8: end function
```

На эффективность выполнения критических секций значимое влияние оказывает алгоритм реализации операции `lock`. Наиболее распространенными являются: алгоритм циклической блокировки (*spinlock*), алгоритм с использованием фьютекса (*futex*) и адаптивный алгоритм. В частности, вышеперечисленные алгоритмы используются для реализации мьютекса в библиотеке стандарта POSIX *Pthreads*, являющейся частью стандартной библиотеки языка C *glibc*.

Алгоритм 10 Алгоритм реализации операций над мьютексом методом циклической блокировкой *spinlock*

```
1: function LOCK(m)
2:     expected ← 0
3:     while CAS(m, expected, 1) ≠ true do
4:         expected ← 0
5:         NOP()
6:     end while
7: end function
8: function UNLOCK(m)
9:     m ← 0
10: end function
```

Алгоритм 10 содержит псевдокод функции `lock`, реализованной алгоритмом циклической блокировки мьютекса и псевдокод фунции `unlock`. При выполнении операции `lock` данным алгоритмом поток в цикле пытается захватить мьютекс `m`, установив его в значение, равное 1, при помощи атомарной операции CAS. Для освобождения мьютекса (операция `unlock`) его значение необходимо установить в 0. Активное ожидание освобождения мьютекса является главным недостатком данного алгоритма, так как в случае, если мьютекс защищает объемную критическую секцию, процессорное ядро, на котором выполняется ожидающий поток, будет выполнять пустую работу (команда `NOP` – *No operation*). Кроме того, частое использование префикса `lock` в операции CAS, а также частое обращение к разделяемой области памяти, в которой хранится мьютекс, могут привести к снижению эффективности параллельной программы.

Алгоритм 11 Алгоритм реализации операций над мьютексом с использованием фьютекса (*futex*)

```
1:  function LOCK(m)
2:      expected ← 0
3:      while CAS(m, expected, 1) ≠ true do
4:          expected ← 0
5:          FUTEX_WAIT(m, expected)
6:      end while
7:  end function
8:  function UNLOCK(m)
9:      m ← 0
10:     FUTEX_WAKE(m, 1)
11: end function
```

Алгоритм 11 содержит операции захвата мьютекса с использованием фьютекса и его освобождения. При использовании этого алгоритма поток выполняет одну попытку захватить мьютекс при помощи операции `CAS`, если ему это не удается, то поток переводится в спящее состояние при помощи функции `FUTEX_WAIT`. Функция `FUTEX_WAIT`, принимающая два аргумента: мьютекс и его ожидаемое значение, вытесняет текущий поток из очереди планирования планировщика в ядре ОС и переводит его в спящее состояние. Пробуждение потока и его постановка в очередь планирования происходит

только после того, как передаваемый первым аргументом мьютекс не примет значение, передаваемое вторым аргументом, в данном случае 0 – значение, при котором мьютекс `m` свободен. Операция `unlock` освобождения мьютекса устанавливает его значение равным 0 и вызывает функцию `FUTEX_WAKE` для пробуждения спящих потоков, если такие имеются. Функция `FUTEX_WAKE` принимает два аргумента: мьютекс, изменение которого ожидают спящие потоки, и количество пробуждаемых потоков. В данном случае пробуждается 1 поток. Этот алгоритм рекомендуется использовать в программах с большими критическими секциями. В случае небольшой критической секции этот алгоритм может привести к существенным накладным расходам на выполнение системных вызовов `FUTEX_WAIT` и `FUTEX_WAKE`.

Алгоритм 12 Адаптивный алгоритм реализации операций на мьютексом

```
1:  function LOCK(m)
2:      i ← 0
3:      expected ← 0
4:      while CAS(m, expected, 1) ≠ true do
5:          expected ← 0
6:          i ← i + 1
7:          if i == attempts then
8:              FUTEX_WAIT(m, expected)
9:              i ← 0
10:         end if
11:         NOP()
12:     end while
13: end function
14: function UNLOCK(m)
15:     m ← 0
16:     FUTEX_WAKE(m, 1)
17: end function
```

Алгоритм 12 содержит псевдокод адаптивного алгоритма операций захвата мьютекса `lock` и освобождения `unlock` мьютекса. Отличительной особенностью данного алгоритма от предыдущего (Алгоритм 11) является то, что выполняющий захват мьютекса поток переходит в спящее состояние не сразу после первой неудавшейся попытки выполнить операцию `CAS`, а по

истечении `attempts` попыток. Выбор оптимального значения для параметра `attempts` осуществляется на основании динамических характеристик программы и свойств критических секций. Для этого могут использоваться методы оптимизации по результатам предварительного профилирования (*profile guided optimization – PGO*), а также различные методы статического анализа кода на этапе компиляции программы. Данная тема представляет интерес и нуждается в дополнительном исследовании.

3.1.3 Неблокирующие алгоритмы и структуры данных

Неблокирующая синхронизация выполнения потоков параллельной программы является одним из способов организации потокобезопасной работы с разделяемыми структурами данных, такими как массивы, списки, хеш-таблицы, деревья и др. при помощи алгоритмов, свободных от блокировок.

Алгоритмы, свободные от блокировок (*lock-free algorithms*) – это потокобезопасные алгоритмы, на каждом шаге которых гарантируется успешное выполнение хотя бы одного потока из множества выполняемых этот алгоритм [9]. Таким образом, алгоритм содержащий критическую секцию, выполнение которой возможно только после успешного захвата мьютекса, не является свободным от блокировок. Так как в случае, если захвативший мьютекс поток по каким-то причинам не сможет выполнить критическую секцию, например, будет вытеснен планировщиком операционной системы, ни один другой поток не сможет ее выполнить. В алгоритмах без блокировок такая ситуация исключена.

Алгоритм 13 Lock-free алгоритм операции `PUSH` помещения элемента в стек

```
1: function PUSH(S, x)
2:     node ← LinkedListCreateNode(x)
3:     do
4:         head ← S.head
5:         node.next ← head
6:     while CAS(S.head, head, node) ≠ true
7: end function
```

Пусть имеется потокобезопасный стек S на основе двусвязного списка. В этот стек несколько потоков одновременно пытаются поместить элемент x при помощи операции `PUSH`, которая реализована свободным от блокировок алгоритмом. В Алгоритме 13 представлен псевдокод такой операции. Тогда каждый поток выполняет цикл `do-while` до тех пор, пока операция `CAS` не завершится успешно. Если операция `CAS` завершилась неуспешно, и поток повторяет выполнение цикла снова, то это значит, что какой-то другой поток успешно выполнил операцию `PUSH`. Таким образом, гарантируется успешное выполнение операции хотя бы одним потоком, что и является обязательным для алгоритмов, свободных от блокировок.

Алгоритмы и структуры данных, свободные от блокировок являются актуальной темой в области исследования перспективных подходов к созданию потокобезопасных масштабируемых программ, над которой ведется активная работа [9, 24]. Однако, недостатками данного способа является высокая сложность разработки и отладки параллельных программ и ограниченность его применения. Несмотря на то, что неблокирующие алгоритмы и структуры данных полностью избавляют от возможности возникновения *взаимной блокировки* (*deadlock*), они не избавляют от возможности появления *активной блокировки* (*livelock*).

3.1.4 Транзакционная память

Транзакционная память (*Transactional memory*) – это один из примитивов синхронизации потоков, позволяющий предотвратить возникновение состояния гонок за данными без использования блокировок [52, 99]. В отличие от примитивов синхронизации на основе блокировок, транзакционная память позволяет защитить *область памяти программы* от конкурентного доступа множества потоков, а не участок кода от одновременного выполнения. Использование транзакционной памяти существенно упрощает процесс разработки потокобезопасных программ и позволяет избежать ошибок синхронизации. Отсутствие операций блокировок хорошо сказывается на масштабируемости и эффективности параллельных программ. Существуют как программные реализации транзакционной памяти (*software transactional*

memory – STM): LazySTM, TinySTM, GCC TM, DTMC, RSTM, STMX, STM Monad, так и аппаратные реализации в процессорах (*hardware transactional memory – HTM*): Intel TSX, AMD ASF, Oracle Rock, IBM POWER8, IBM PowerPC A2.

Транзакционная память предоставляет языковые конструкции или API, позволяющие выделять в программе участки кода, в которых осуществляется защита совместно используемых областей памяти. Такие участки кода принято называть *транзакционными секциями* (*transactional section*), а их выполнение осуществляется без блокирования потоков. В случае, если в целевом процессоре не реализована аппаратная транзакционная память, то задача по контролю за корректностью выполнения транзакций ложится на среду выполнения (runtime). Если же поддержка HTM реализована, то корректность выполнения (например, отсутствие состояний гонок за данными) обеспечивается процессором, чаще всего на уровне работы кэш-памяти при помощи модифицированного протокола когерентности кэшей – *TMESI* [26]. Если во время выполнения транзакции одним потоком произошло изменение защищаемой области памяти другими потоками, то транзакция считается некорректной. В этом случае при помощи аппаратных возможностей процессора либо runtime-системы выполнение транзакции прерывается. Все модифицированные в рамках нее области памяти восстанавливаются в исходное состояние, и поток начинает выполнять ее заново.

3.2 Программная транзакционная память

3.2.1 Основные понятия STM

Программная транзакционная память предоставляет операции начала транзакции – `BeginTransaction`, ее принудительного завершения – `AbortTransaction` и конца транзакции – `CommitTrasnaction`, позволяющие создавать в коде программы транзакционные секции. Для их выполнения runtime-система языка создает транзакции. *Транзакция*

(*transaction*) – это конечная последовательность операций транзакционного чтения/записи памяти. Операция *транзакционного чтения* выполняет копирование содержимого указанного участка общей памяти в соответствующий участок локальной памяти потока. *Транзакционная запись* копирует содержимое указанного участка локальной памяти в соответствующий участок общей памяти, доступной всем потокам.

Инструкции транзакций выполняются потоками параллельно (конкурентно). После завершения выполнения транзакция может быть либо *зафиксирована* (commit), либо *отменена* (cancel). Фиксация транзакции подразумевает, что все сделанные в рамках нее изменения памяти становятся необратимыми. При отмене транзакции ее выполнение прерывается, а состояние всех модифицированных областей памяти восстанавливается в исходное с последующим перезапуском транзакции (*откат транзакции*, rollback).

Отмена транзакции происходит в случае обнаружения *конфликта* – ситуации, при которой два или более потока обращаются к одному и тому же участку памяти, и как минимум один из них выполняет операцию записи.

Для разрешения конфликта разработаны различные подходы, например, можно приостановить на некоторое время или отменить одну из конфликтующих транзакций.

Листинг 3.3: Добавление пары $(key, value)$ в хеш-таблицу h

```
1  /* Совместно используемая хеш-таблица */
2  hashtable_t *h;
3  /* Код потоков */
4  void *thread_start(void *arg) {
5      struct data *d = (struct data *)arg;
6      prepareData(d);
7      /* Транзакционная секция */
8      __transaction_atomic {
9          /* Добавление элемента в хеш-таблицу */
10         struct data *d = (struct data *)arg;
11         hashtable_insert(h, d);
12     }
13     saveData(d);
14     return NULL;
15 }
```

В Листинге 3.3 представлен пример создания транзакционной секции, в теле которой выполняется добавление элемента в хеш-таблицу множеством потоков. После выполнения тела транзакционной секции каждый поток приступит к выполнению кода, следующего за ней, в случае отсутствия конфликтов. В противном случае поток повторно будет выполнять транзакцию до тех пор, пока она не будет успешно зафиксирована.

3.2.2 Реализация программной транзакционной памяти

Реализация программной транзакционной памяти требует изменения трех основных составляющих:

- стандарта языка программирования;
- компилятора;
- runtime-библиотеки.

Поддержка STM в стандарте языка программирования требует наличия ключевых слов для создания транзакционных секций в коде, а также наличия описания семантики использования транзакционной памяти в параллельных программах. Поэтому международным комитетом ISO по стандартизации языка C++, в рамках рабочей группы WG21, ведутся работы по внедрению транзакционной памяти в стандарт языка. На сегодняшний день предложен черновой вариант спецификации поддержки транзакционной памяти в C++ [24, 38]. Она предоставляет ключевые слова `__transaction_atomic`, `__transaction_relaxed` для создания транзакционных секций, а также `__transaction_cancel` для принудительной отмены транзакции. Эти конструкции также возможно использовать и в программах на языке C.

Компилятор, поддерживающий языковые конструкции для создания транзакционных секций, принято называть *STM-компилятором*. STM-компилятор преобразует языковые конструкции транзакционной памяти в последовательность вызовов функций runtime-библиотеки или в специальные ассемблерные инструкции, если целевой процессор поддерживает аппаратную транзакционную память. Для языков программирования C/C++ поддержка STM реализована в таких компиляторах как: GCC, начиная с версии 4.7, Intel C++ Compiler и др.

Runtime-библиотека STM отслеживает попытки одновременного доступа к одной и той же области памяти, тем самым осуществляя обнаружение конфликтов. Кроме этого, runtime-библиотека организует использование возможностей аппаратной транзакционной памяти, если целевой процессор ее поддерживает. В компиляторе GCC реализована библиотека `libitm`, которая содержит несколько алгоритмов работы STM, в том числе и алгоритм, основанный на использовании аппаратной транзакционной памяти. Для обеспечения архитектурной и платформенной переносимости программ, использующих STM, компания Intel разработала спецификацию ABI (*aplication binary interface*, *бинарный интерфейс приложений*) для STM-компиляторов и runtime-систем – Intel TM ABI. Компилятор GCC и библиотека `libitm` реализуют этот ABI.

3.2.3 Алгоритмы работы программной транзакционной памяти

Алгоритм работы программной транзакционной памяти реализуется runtime-системой и определяет:

- способ хранения информации о состоянии защищаемых областей памяти;
- политику обновления объектов в памяти;
- стратегию обнаружения конфликтов;
- метод разрешения конфликтов.

Для обнаружения конфликтов runtime-система языка отслеживает попытки одновременного доступа к одной и той же области памяти. Это реализуется путем поддержки информации о состоянии защищаемых регионов памяти. Возможны два уровня гранулярности контролируемых областей: *уровень программных объектов* (*object-based STM*) и *уровень слов памяти* (*word-based STM*).

Уровень программных объектов подразумевает поддержку runtime-системой метаданных о состоянии каждого объекта программы. Например, объектов в C++-программе.

Для реализации уровня слов памяти в простейшем случае требуется каждый байт линейного адресного пространства процесса сопровождать

метаданными, что является практически невозможным. Вместо этого линейное адресное пространство процесса разбивается на фиксированные блоки, каждый из которых сопровождается метаданными о состоянии (подход, подобный прямому отображению физических адресов на кэш-память процессора) [47, 55]. Это приводит к тому, что множеству областей памяти соответствуют одни метаданные, что является источником возникновения ложных конфликтов. *Ложный конфликт* (*false conflict*) – это ситуация, при которой два или более потока во время выполнения транзакции обращаются к разным участкам линейного адресного пространства, но отображаемым на одни и те же метаданные. Поэтому runtime-система воспринимает такую ситуацию как конфликт (data race), хотя на самом деле таковой отсутствует. Ложные конфликты существенно снижают эффективность параллельных STM-программ. Поэтому остро стоит задача разработки алгоритмов обнаружения и сокращения числа ложных конфликтов в реализациях STM.

Политика обновления объектов в памяти определяет, когда изменения объектов в рамках транзакции будут записаны в память. Распространение получили две основные политики – ленивая и ранняя. *Ленивая* политика обновления объектов в памяти (lazy version management) откладывает все операции с объектами до момента фиксации транзакции. Все операции записываются в специальном журнале (redo log), который при фиксации используется для отложенного выполнения операций. Очевидно, что это замедляет операцию фиксации, но существенно упрощает процедуры ее отмены и восстановления. Примером реализаций ТП, использующих данную политику, являются RSTM-LLT [48] и RSTM-RingSW [54, 61].

Ранняя политика обновления (eager version management) предполагает, что все изменения объектов сразу записываются в память. В журнале отката (undo log) фиксируются все выполненные операции с памятью. Он используется для восстановления оригинального состояния модифицируемых участков памяти в случае возникновения конфликта. Эта политика характеризуется быстрым выполнением операции фиксации транзакции, но медленным выполнением процедуры ее отмены. Примерами реализаций, использующих раннюю политику обновления данных, являются GCC (libitm), TinySTM [55], LSA-STM [47], Log-TM [37], RSTM [48] и др.

Момент времени, когда инициируется алгоритм обнаружения конфликта, определяется *стратегией обнаружения конфликтов*. При *отложенной*

стратегии (lazy conflict detection) алгоритм обнаружения конфликтов запускается на этапе фиксации транзакции [54]. Недостатком этой стратегии является то, что временной интервал между возникновением конфликта и его обнаружением может быть достаточно большим. Эта стратегия используется в RSTM-LLT [48] и RSTM-RingSW [48, 54, 61].

Пессимистичная стратегия обнаружения конфликтов (eager conflict detection) запускает алгоритм их обнаружения при каждой операции обращения к памяти. Такой подход позволяет избежать недостатков отложенной стратегии, но может привести к значительным накладным расходам, а также, в некоторых случаях, может привести к увеличению числа откатов транзакций. Стратегия реализована в TinySTM [55], LSA-STM [47] и TL2 [11]. В компиляторе GCC (libitm) реализован комбинированный подход к обнаружению конфликтов – отложенная стратегия используется совместно с пессимистической.

3.3 Задача о предотвращении возникновения ложных конфликтов

3.3.1 Определение ложных конфликтов

Для обнаружения конфликтных операций требуется отслеживать изменения состояния используемых областей памяти. Информация о состоянии может соответствовать областям памяти различной степени гранулярности. Выбор гранулярности обнаружения конфликтов – один из ключевых моментов при реализации программной транзакционной памяти.

На сегодняшний день используются два уровня гранулярности: *уровень программных объектов* (object-based STM) и *уровень слов памяти* (word-based STM). Уровень программных объектов подразумевает отображение объектов модели памяти языка (объекты C++, Java, Scala) на метаданные runtime-библиотеки. При использовании уровня слов памяти осуществляется отображение блоков линейного адресного пространства процесса на метаданные. Метаданные хранятся в таблице, каждая строка которой соответствует

объекту программы или области линейного адресного пространства процесса. В строке содержатся номер транзакции, выполняющей операцию чтения /записи памяти; номер версии отображаемых данных; их состояние и др. Модификация метаданных выполняется runtime-системой с помощью атомарных операций процессора.

Реализация программной транзакционной памяти в компиляторе GCC использует уровень слов памяти, в которых размер блока по умолчанию равен 16 байт.

На Рисунке 3.1 представлен пример организации метаданных транзакционной памяти с использованием уровня слов памяти (GCC 4.7+). Линейное адресное пространство процесса фиксированными блоками циклически отображается на строки таблицы, подобно кэшу прямого отображения. Выполнение операции записи приведет к изменению поля «состояние» соответствующей строки таблицы на «заблокировано». Доступ к области линейного адресного пространства, у которой соответствующая строка таблицы помечена как «заблокировано», приводит к конфликту.

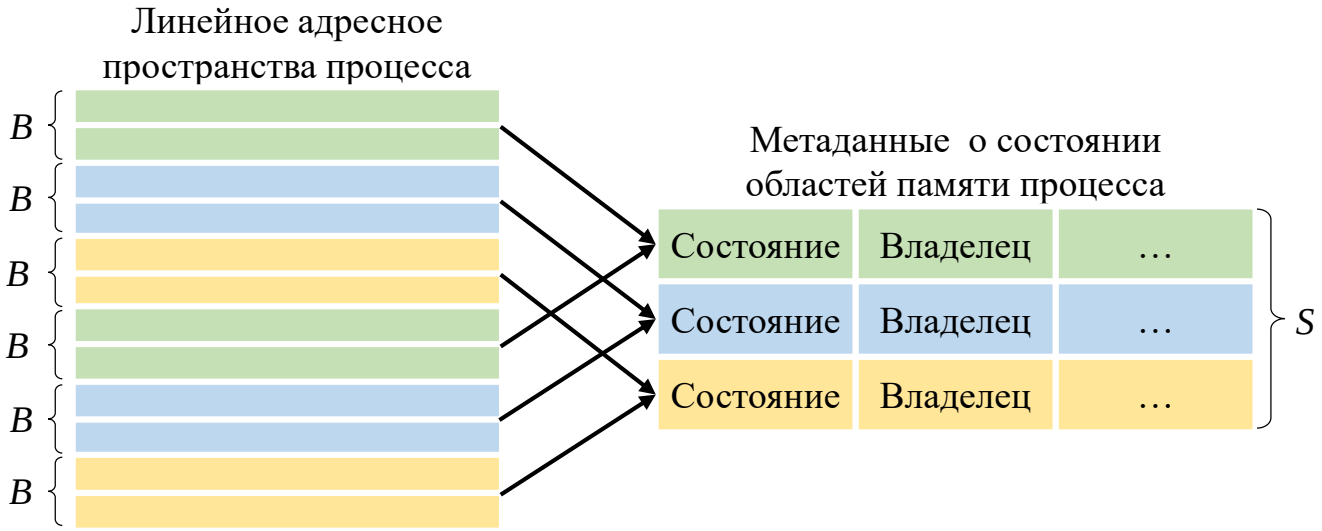


Рисунок 3.1 – Таблица с метаданными GCC 4.7+ (word-based STM): $B = 16$, $S = 2^{19}$

Основными параметрами транзакционной памяти с использованием уровня слов памяти являются число S строк таблицы и количество B адресов линейного адресного пространства, отображаемых на одну строку таблицы. От выбора этих параметров зависит число ложных конфликтов – ситуаций, аналогичных ситуации ложного разделения данных при работе кэша процессора. В текущей реализации GCC (4.7-5.1) эти параметры фиксированы [19].

При отображении блоков линейного адресного пространства процесса на метаданные runtime-библиотеки возникают коллизии. Это неизбежно, так как размер таблицы метаданных гораздо меньше размера линейного адресного пространства процесса. Коллизии приводят к возникновению ложных конфликтов. *Ложный конфликт* – ситуация, при которой два или более потока во время выполнения транзакции обращаются к разным участкам линейного адресного пространства, но сопровождаемым одними и теми же метаданными о состоянии, и как минимум один поток выполняет операцию записи. Таким образом, ложный конфликт – это конфликт, который происходит не на уровне данных программы, а на уровне метаданных runtime-библиотеки.

Возникновение ложных конфликтов приводит к откату транзакций так же, как и возникновение обычных конфликтов, несмотря на то, что состояние гонки за данными не возникает, что влечет за собой увеличение времени выполнения STM-программ. Сократив число ложных конфликтов, можно существенно уменьшить время выполнения программы.

На Рисунке 3.2 показан пример возникновения ложного конфликта в результате коллизии отображения линейного адресного пространства на строку таблицы. Поток 1 при выполнении операции записи над областью памяти с адресом $A1$ захватывает соответствующую строку таблицы. Выполнение операции чтения над областью памяти с адресом $A2$ потоком 2 приводит к возникновению конфликта, несмотря на то, что операции чтения и записи выполняются над различными адресами. Последнее обусловлено тем, что 1 и 2 отображены на одну строку таблицы метаданных.

3.3.2 Предотвращение возникновения ложных конфликтов методом реорганизации таблицы метаданных

В работе [60] для минимизации числа ложных конфликтов предлагается использовать вместо таблицы с прямой адресацией (как в GCC 4.7+), в которой индексом является часть линейного адреса, хеш-таблицу, коллизии в которой разрешаются методом цепочек. В случае отображения нескольких

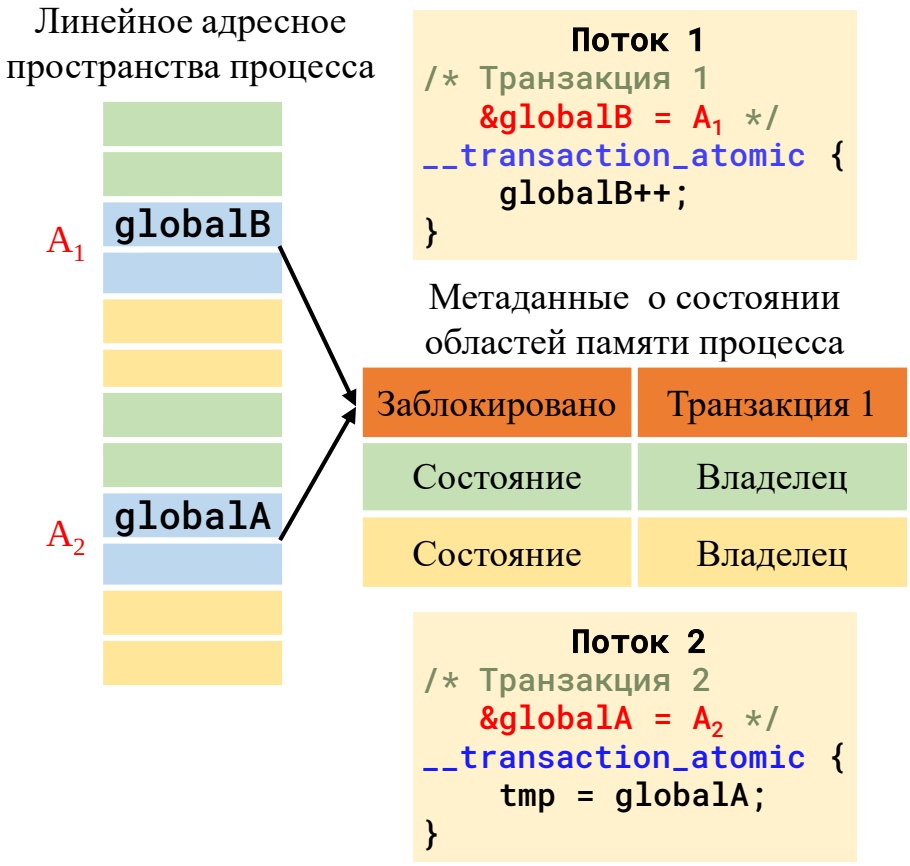


Рисунок 3.2 – Пример возникновения ложного конфликта при выполнении двух транзакций (GCC 4.7+)

адресов на одну запись таблицы каждый адрес добавляется в список и помечается тэгом для идентификации (Рисунок 3.3). Такой подход позволяет избежать ложных конфликтов, однако накладные расходы на синхронизацию доступа к метаданным существенно возрастают, так как значительно увеличивается количество атомарных операций **сравнение с обменом** (compare and swap – CAS).

3.4 Сокращение возникновения ложных конфликтов по результатам предварительного профилирования

Автором предложен метод, позволяющий сократить число ложных конфликтов в STM-программах [34, 70–72, 84–89]. Предполагается, что метаданные организованы в виде таблицы с прямой адресацией. Суть метода

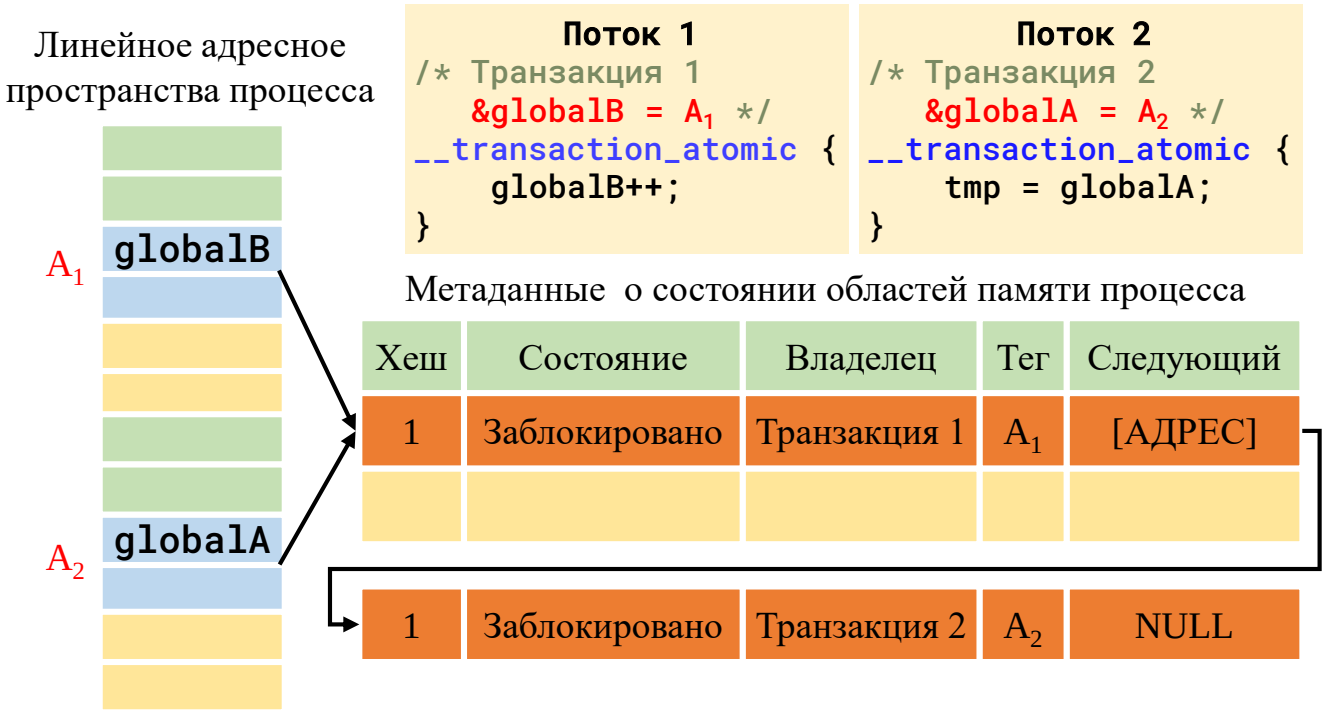


Рисунок 3.3 – Хеш-таблица для хранения метаданных без ложных конфликтов

заключается в автоматической настройке параметров S и B таблицы под динамические характеристики конкретной STM-программы. Метод включает три этапа.

Этап 1. Внедрение функций библиотеки профилирования в транзакционные секции. На первом этапе выполняется компиляция C/C++ STM-программы с использованием разработанного модуля анализа транзакционных секций и внедрения вызова функций библиотеки профилирования (модуль расширения GCC). В ходе статического анализа транзакционных секций STM-программ выполняется внедрение кода для регистрации обращений к функциям Intel TM ABI (`_ITM_beginTransaction`, `_ITM_comitTransaction`, `_ITM_LU4`, `_ITM_WU4` и др.). Детали реализации модуля описаны ниже.

Этап 2. Профилирование программы. На данном этапе выполняется запуск STM-программы в режиме профилирования. Профилировщик регистрирует все операции чтения/записи памяти в транзакциях. В результате формируется протокол (trace), содержащий информацию о ходе выполнения транзакционных секций:

– адрес и размер области памяти, над которой выполняется операция;
– временная метка (timestamp) начала выполнения операции.

Этап 3. Настройка параметров таблицы. По протоколу определяется средний размер W читаемой/записываемой области памяти во время выполнения транзакций. По значению W подбираются субоптимальные параметры B и S таблицы, с которыми STM-программа компилируется.

Алгоритм 14 Алгоритм выбора значений параметров B и S (Этап 3)

```
1:  function STMOptimizeParams(T, B, S)
2:      W ← ProcessTrace(T)
3:      if W == 1 then
4:          B ← 2^4
5:          S ← 2^18
6:      else if W == 4 then
7:          B ← 2^6
8:          S ← 2^19
9:      else if W == 8 then
10:         B ← 2^7
11:         S ← 2^20
12:     else if W == 64 then
13:         B ← 2^8
14:         S ← 2^21
15:     end if
16:     CompileProgram(S, B)
17: end function
```

В Алгоритме 14 представлен псевдокод алгоритма STMOptimizeParams, реализующий этап 3. Эксперименты с тестовыми STM-программами из пакета STAMP [50] (6 типов STM-программ) позволили сформулировать эвристические правила для подбора параметров B и S по значению W, которое определяется в результате анализа протокола trace (функция ProcessTrace). Функция CompileProgram запускает процесс компиляции с новыми параметрами реализации STM. Значение параметра S целесообразно выбирать из множества $\{2^{18}, 2^{19}, 2^{20}, 2^{21}\}$. Значение параметра B выбирается следующим образом:

- если $W = 1$ байт, то $B = 2^4$ байт;
- если $W = 4$ байт, то $B = 2^6$ байт;

– если $W = 8$ байт, то $B = 2^7$ байт;
– если $W >= 64$ байт, то $B = 2^8$ байт.

3.5 Программный инструментарий для сокращения ложных конфликтов

Автором разработан программный инструментарий (STM false conflict optimizer) для оптимизации ложных конфликтов, возникающих при выполнении параллельных программ с транзакционной памятью [101]. Инструментарий позволяет выполнять профилирование STM-программ. Информация, полученная в результате профилирования, предоставляет достаточно сведений о динамических характеристиках транзакционных секций для того, чтобы ответить на вопрос: «Фиксации каких транзакций или операции над какими данными приводят к отмене других транзакций?». Кроме этого, разработанное программное средство позволяет определить субоптимальные значения параметров реализации runtime-системы ТП, а именно, число строк таблицы метаданных о состоянии областей памяти и количество адресов линейного адресного пространства, отображаемых на одну строку таблицы.

3.5.1 Функциональная структура пакета

Программный инструментарий состоит из трех основных компонентов (Рисунок 3.4):

– модуль внедрения функций библиотеки профилирования в код транзакционных секций ($tm\_prof\_analyzer$);
– библиотека профилирования параллельной программы с транзакционной памятью ($libitm\_prof$);
– модуль анализа протокола выполнения транзакционных секций, установки значений параметров реализации ($tm\_proto\_analyzer$).

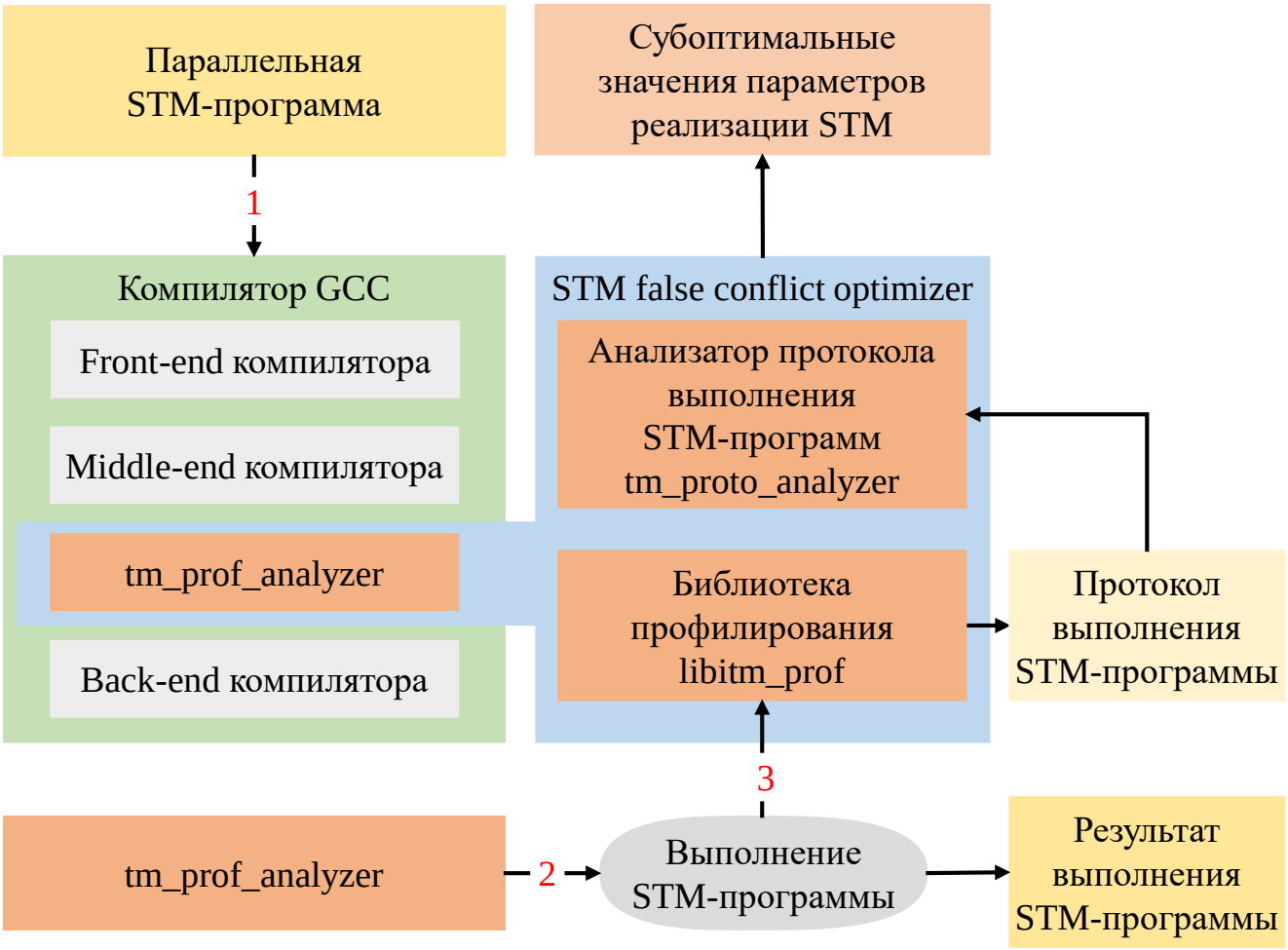


Рисунок 3.4 – Функциональная структура разработанного пакета; 1 – компиляция STM-программы; 2 – запуск STM-программы под управлением профилировщика; 3 – обращение к функциям профилировщика

3.5.2 Внедрение функций профилировщика

STM-компилятор осуществляет трансляцию транзакционных секций в последовательность вызовов функций runtime-системы поддержки транзакционной памяти [39]. Компания Intel предложила спецификацию ABI для runtime-систем поддержки транзакционной памяти – Intel TM ABI [29]. Компилятор GCC (библиотека `libitm`) реализует этот интерфейс, начиная с версии 4.7. В Листинге 3.4 представлен пример трансляции компилятором GCC транзакционной секции в обращения к функциям Intel TM ABI.

Листинг 3.4: Трансляция транзакционной секции компилятором GCC; код вверху – исходная транзакционная секция; код внизу – промежуточное представление трансформированной транзакционной секции

```
/*Исходная Транзакционная секция*/
int a, b;
...
__transaction_atomic {
    if (a == 0)
        b = 1;
    else
        a = 0;
}
...
```

```
/*Трансформированная транзакционная секция*/
     ...
     state = _ITM_beginTransaction()
<L1>:
     if (state & a_abortTransaction)
         goto <L3>;
     else
         goto <L2>;
<L2>:
     if (_ITM_LU4(&a) == 0)
         _ITM_WU4(&b, 1);
     else
         _ITM_WU4(&a, 0);
     _ITM_commitTransaction();
<L3>:
     ...
```

В общем случае последовательность выполнения транзакции следующая:

1. Создание транзакции (вызов `_ITM_beginTransaction`) и анализ ее состояния. Если состояние транзакции содержит флаг принудительной отмены, то выполнение продолжается с метки <L3>, т.е. осуществляется выход из транзакции, иначе выполнение тела транзакции начинается с метки <L2>;
2. Выполнение транзакции. Если выполняется принудительная отмена транзакции, то в состоянии устанавливается флаг принудительной отмены (`a_abortTransaction`) и управление передается метке <L1>;
3. Попытка фиксации транзакции (вызов `_ITM_commitTransaction`). В случае возникновения конфликта транзакция отменяется, в состояние транзакции записывается причина отмены, и выполнение транзакции повторяется, начиная с метки <L1>.

Разработанный модуль `tm_prof_analyzer` внедрения функций библиотеки профилирования выполнен в виде встраиваемого модуля компилятора GCC. Программист компилирует STM-программу с ключом `-fplugin=tm_prof_analyzer.so`. Модуль внедрения выполняет анализ промежуточного представления `GIMPLE` транзакционных секций и добавляет функции регистрации обращений к функциям Intel TM ABI: регистрация начала транзакции и ее фиксации, транзакционное чтение/запись областей памяти.

В Листинге 3.5 представлен пример внедрения вызовов функций библиотеки профилирования в транзакционную секцию. Функции с префиксом `tm_prof_` выполняют регистрацию событий.

Во время выполнения STM-программы под управлением профилировщика (`libitm_prof`) функции регистрации обращений к интерфейсам Intel TM ABI заносят в протокол адреса и размер областей памяти, над которыми выполняются операции, а также время начала выполнения операций. После завершения выполнения STM-программы формируется протокол выполнения программы, на основе которого модуль анализа (`tm_proto_analyzer`) осуществляет выбор субоптимальных параметров таблицы метаданных транзакционной памяти.

Листинг 3.5: Встраивание в транзакционную секцию функций библиотеки профилирования; код вверху – исходная транзакционная секция; код внизу – промежуточное представление трансформированной транзакционной секции

```
/*Исходная транзакционная секция*/
int a, b;
...
__transaction_atomic {
    if (a == 0)
        b = 1;
    else
        a = 0;
}
...
```

```
/*Трансформированная транзакционная секция*/
     ...
     state = _ITM_beginTransaction()
     tm_prof_begin(state);
<L1>:
     if (state & a_abortTransaction)
         goto <L3>;
     else
         goto <L2>;
<L2>:
     tm_prof_operation(sizeof(a));
     if (_ITM_LU4(&a) == 0) {
         tm_prof_operation(sizeof(b));
         _ITM_WU4(&b, 1);
     } else {
         tm_prof_operation(sizeof(a));
         _ITM_WU4(&a, 0);
     }
     _ITM_commitTransaction();
     tm_prof_commit();
<L3>:
     ...
```

3.6 Экспериментальное исследование метода оптимизации обнаружения конфликтов

Экспериментальное исследование проводилось на ВС, оснащенной двумя четырехъядерными процессорами Intel Xeon E5420. В данных процессорах отсутствует поддержка аппаратной транзакционной памяти (Intel TSX). В качестве тестовых программ использовались многопоточные STM-программы из пакета STAMP [11, 54, 61]. Число потоков варьировалось от 1 до 8. Тесты собирались компилятором GCC 5.1.1. Операционная система – GNU/Linux Fedora 21 x86_64.

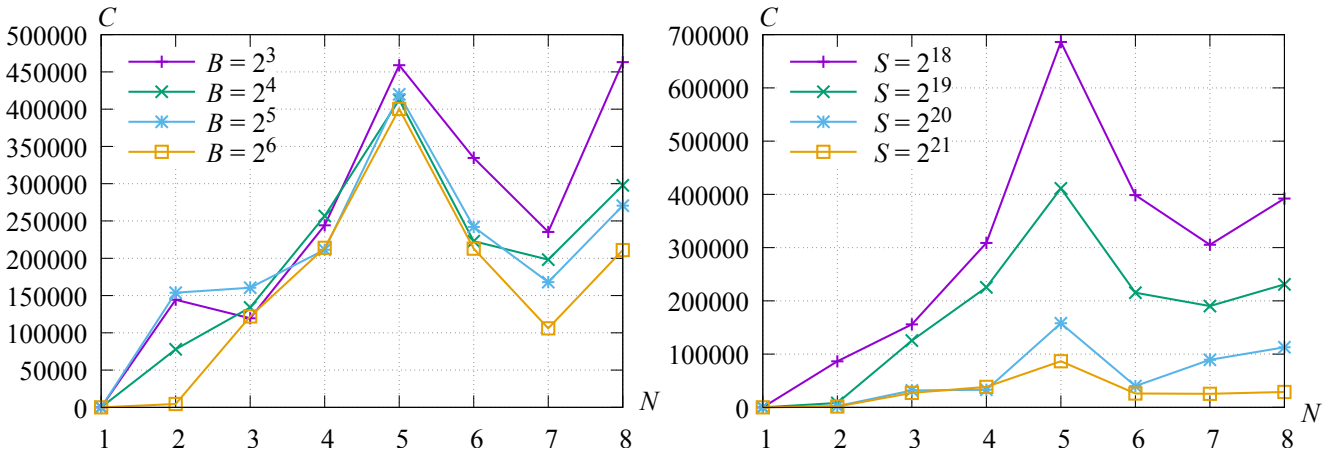
В рамках экспериментов измерялись значения двух показателей:

– время t выполнения STM-программы;

– количество C ложных конфликтов в программе.

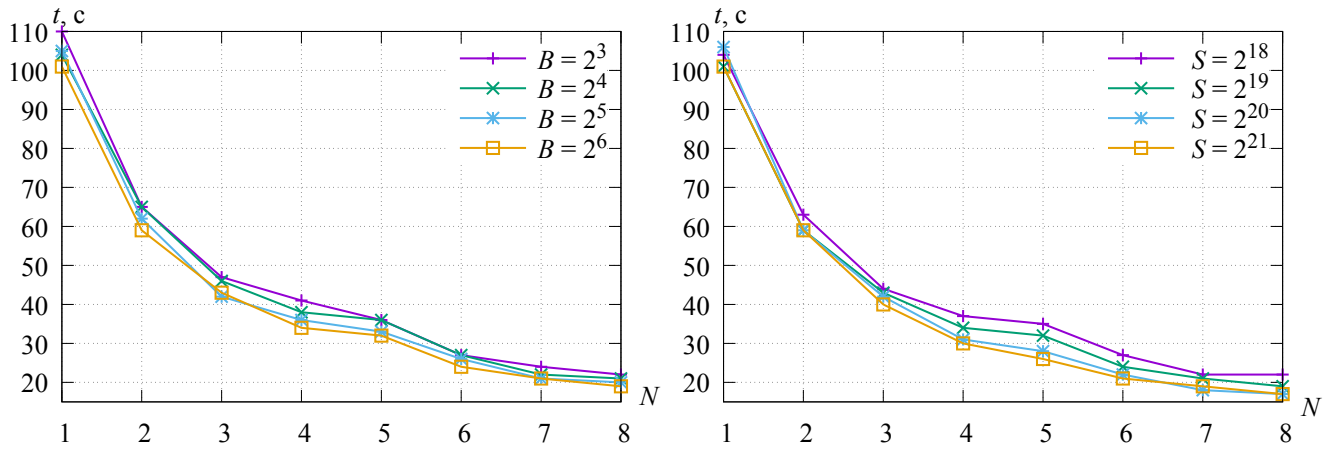
На Рисунках 3.5а, 3.5б, 3.6а, 3.6б показана зависимость количества C ложных конфликтов и времени t выполнения теста от числа потоков при различных значениях параметров B и S. Результаты приведены для программы genome из пакета STAMP. В ней порядка 10 транзакционных секций, реализующих операции над хеш-таблицей и связными списками. Видно, что увеличение значений параметров S и B приводит к уменьшению числа возможных коллизий (ложных конфликтов), возникающих при отображении адресов линейного адресного пространства процесса на записи таблицы. При размере таблицы 2^{21} записей, на каждую из которых отображается 2^6 адресов линейного адресного пространства, достигается минимум времени выполнения теста genome, а также минимум числа ложных конфликтов.

Время выполнения теста genome удалось сократить в среднем на 20% за счет минимизации числа ложных конфликтов.



(а) $S = 2^{19}$ (б) $B = 2^6$

Рисунок 3.5 – Число C ложных конфликтов



(а) $S = 2^{19}$ (б) $B = 2^6$

Рисунок 3.6 – Время t выполнения теста

3.7 Оптимизация выполнения циклов

3.7.1 Архитектурные возможности ускорения вычислений

Для эффективного использования ресурсов многоядерного процессора современные компиляторы применяют машинно-независимые техники оптимизации программ [42]. Такие техники либо полностью игнорируют ресурсы процессора, либо представляют их моделью, которая не учитывает архитектурные возможности [40]. Существует три основных типа машинно-независимых техник оптимизации: автоматическая векторизация

кода, конвейеризация циклов и планирование команд [2, 3, 41, 42]. Однако, чтобы повысить эффективность выполнения вычислений на современных процессорах, требуется учитывать архитектурные и микроархитектурные возможности [20–22, 53].

Параллелизм команд на уровне АЛУ ядра и микроархитектурные возможности ускорения вычислений. Современные процессоры являются многоядерными и, как правило, обладают суперскалярной архитектурой с конвейерным принципом выполнения команд [45, 76, 93]. Основная идея конвейерной архитектуры заключается в разбиении выполнения команд процессора на несколько простых стадий. Следующая команда начнет свое выполнение после завершения выполнения нескольких стадий текущей команды, а не после ее окончательного выполнения.

Суперскалярная архитектура подразумевает наличие в вычислительном ядре процессора нескольких АЛУ, способных одновременно выполнять команды (*параллелизм на уровне команд*, *instruction level parallelism*).

Способ организации архитектуры набора команд в процессоре принято называть микроархитектурой. Микроархитектура современных процессоров является суперскалярной и реализует принцип конвейерного выполнения команд. На Рисунке 3.7 представлена функциональная структура Skylake микроархитектуры вычислительного ядра, которая используется в процессорах Intel Core 6-го поколения [27, 28]. Функциональную структуру вычислительного ядра с микроархитектурой Skylake можно условно разделить на 2 логических блока: конвейер верхнего уровня (Frontend) и суперскалярный конвейер нижнего уровня (Backend) [28]. Конвейер верхнего уровня выбирает CISC-команды из L1 кэша команд, при помощи устройства декодирования (Legacy Decode Pipeline) преобразует в микрокоманды (RISC-команды) и помещает в очередь декодированных микрокоманд (Instruction Decode Queue – IDQ). Пропускная способность устройства декодирования составляет 5 команд, а емкость очереди – 64 микрокоманды.

Суперскалярный конвейер нижнего уровня состоит из функциональных устройств (Port 0-7), которые реализуют выполнение арифметических и логических операций, загрузки данных в регистры и сохранение значений регистров в память, управляющие и системные команды. Функциональные устройства также организованы по принципу конвейерной обработки данных.

Наличие нескольких функциональных устройств позволяет выполнять микрокоманды, не зависящие по данным, одновременно (параллелизм на уровне команд). Микроархитектура Skylake может выдавать до 8 микрокоманд за такт.

Функциональным устройствам выдаются только готовые для выполнения микрокоманды. Микрокоманда считается готовой для выполнения, если ее операнды загружены из памяти в регистры. Если такие микрокоманды отсутствуют, то конвейер свободных функциональных устройств простаивает и ничего не выполняет. Для максимального задействования всех имеющихся функциональных устройств в конвейере вычислительного ядра имеется динамический планировщик, реализующий парадигму внеочередного выполнения микрокоманд (Out-of-order execution). Данная парадигма позволяет сократить время простоя конвейеров функциональных устройств и в большей степени задействовать микроархитектурные возможности для ускорения вычислений.

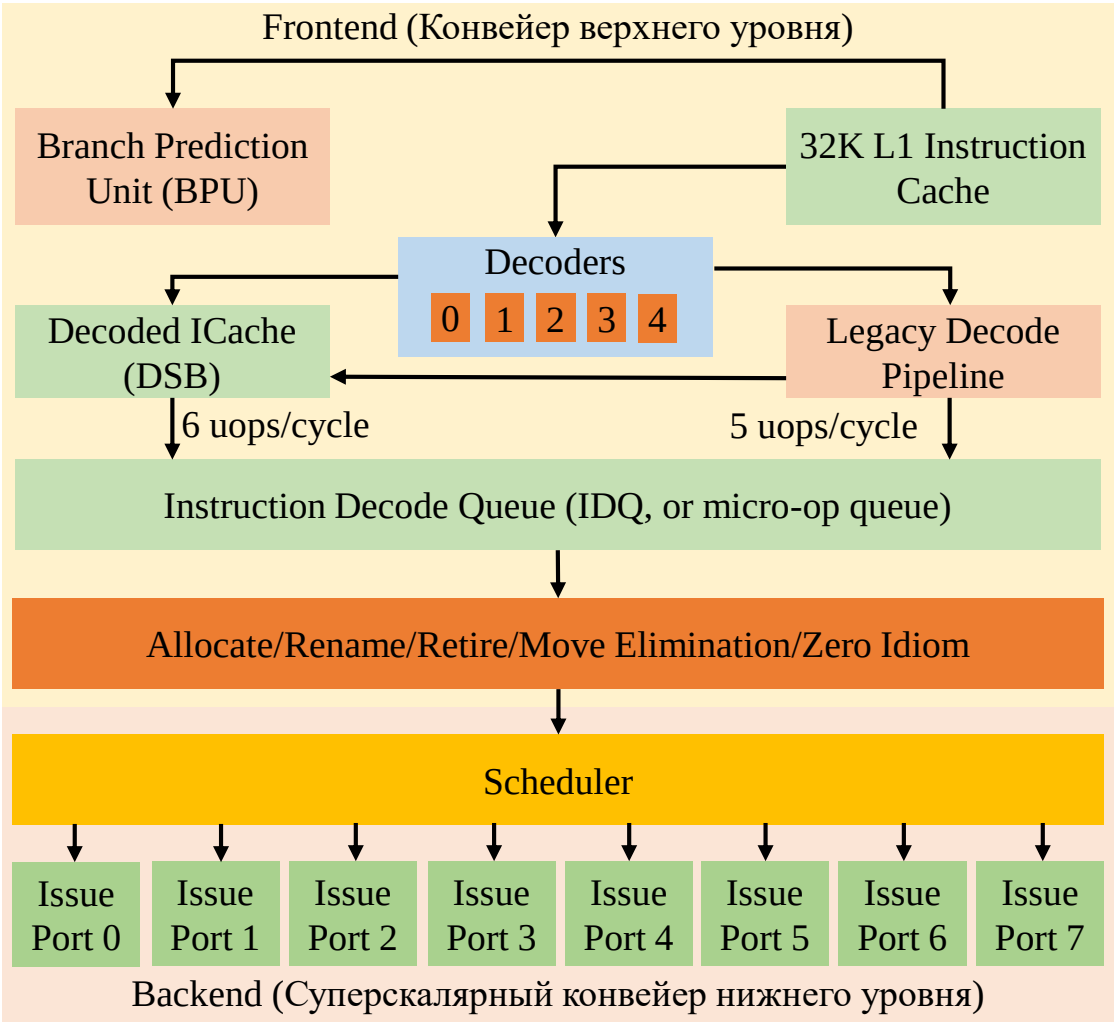


Рисунок 3.7 – Функциональная структура микроархитектуры Skylake

Динамический планировщик выбирает готовые микрокоманды из очереди и направляет свободным функциональным устройствам для выполнения. Алгоритм выбора планировщиком готовых микрокоманд должен не только максимальным образом задействовать параллелизм на уровне команд и использовать микроархитектурные возможности для ускорения вычислений, но и сохранить семантику выполняемого кода. Динамические планировщики многих современных процессоров реализуют алгоритм Томасуло [56].

Параллелизм данных на уровне векторных АЛУ вычислительного ядра. Наряду с параллелизмом уровня команд вычислительные ядра современных процессоров реализуют параллелизм уровня данных благодаря наличию векторных АЛУ. Наборы команд практически всех архитектур современных процессоров включают поддержку векторных команд: Intel SSE/AVX/AVX-512, ARM NEON SIMD, IBM AltiVec, MIPS MSA. Процессоры, реализующие поддержку таких команд, содержат одно или несколько параллельно работающих векторных АЛУ и совокупность векторных регистров. В отличие от векторных систем 1990-х годов, современные процессоры поддерживают выполнение операций с векторами относительно небольшой длины (64 – 512 бит), предварительно загруженными из оперативной памяти в векторные регистры (класс векторных систем «регистр-регистр»).

Основная сфера применения векторных инструкций – сокращение времени работы с одномерными массивами. При векторизации происходит трансформация выполнения итераций обработки массивов данных в векторные инструкции, выполняющиеся одновременно над несколькими экземплярами данных. Как правило, ускорение, достигаемое при использовании векторных инструкций, в первую очередь определяется количеством элементов массива, помещающихся в векторный регистр. Например, каждый из 16 векторных регистров AVX имеет ширину 256 бит, что позволяет загружать в них 16 элементов типа `short int` (16 бит), 8 элементов типа `int` или `float` (32 бита) и 4 элемента типа `double` (64 бита). Соответственно, при использовании AVX ожидаемое ускорение выполнения операций с массивами типа `short int` – 16 раз, `int` и `float` – 8 раз, `double` – 4 раза.

Процессоры Intel Xeon Phi поддерживают набор векторных инструкций AVX-512 и содержат 32 векторных регистра шириной 512 бит. Каждое

ядро процессора с микроархитектурой Knights Corner содержит одно векторное АЛУ шириной 512 бит, а ядра процессора с микроархитектурой Knights Landing – два АЛУ.

Достижение максимального ускорения при векторной обработке требует учета микроархитектурных параметров процессоров. Например, таких как выравнивание на заданную границу начальных адресов массивов, загружаемых в векторные регистры (32 байта для AVX и 64 байта для AVX-512). А также смешанное использование SSE- и AVX-инструкций. В этом случае при переходе от выполнения команд одного векторного расширения к другому процессор сохраняет (при переходе от AVX к SSE) или восстанавливает (в противоположном случае) старшие 128 бит векторных регистров YMM (AVX-SSE transition penalties) [44].

Причиной дополнительного ускорения может являться параллельное выполнение векторных инструкций несколькими векторными АЛУ. Таким образом, эффективно векторизованная версия программы в меньшей степени загружает ряд подсистем суперскалярного конвейера процессора [57].

Разработчикам прикладных программ доступны следующие способы использования векторных инструкций: ассемблерные вставки; интринсики (intrinsics); SIMD-директивы компиляторов, стандартов OpenMP, OpenACC; языковые расширения и библиотеки; автоматическая векторизация компилятором. В данной работе внимание уделено последнему подходу. Автоматическая векторизация циклов компиляторами является одной из наиболее значимых методик их оптимизации. Такой способ векторизации не требует значительной модификации прикладных программ и обеспечивает их переносимость на уровне исходного кода между разными архитектурами процессоров.

3.7.2 Инструментарий анализа эффективности использования функциональных устройств вычислительного ядра

Архитектурно-ориентированная оптимизация программ требует использования средств анализа эффективности использования микроархитектурных

возможностей для ускорения вычислений. В ходе выполнения всей программы или определенного участка такие средства должны собирать такую информацию как количество декодированных команд, количество микрокоманд, выданных функциональным устройствам для выполнения, количество процессорных тактов, необходимых для выполнения выделенного участка кода, количество процессорных тактов, в течение которых простаивал конвейер функциональных устройств, количество условных переходов и др.

Современные процессоры реализуют технологии, позволяющие выполнять глубокий анализ эффективности кода программы. В частности, в архитектуру процессоров фирмы Intel внедрены специальные счетчики мониторинга производительности, реализована технология Intel Processor Trace для трассировки выполнения кода программ [27, 28].

В процессе профилирования и анализа эффективности программ очень важно уменьшить влияние ОС на выполнение кода. Полностью исключить активность ОС невозможно, так как неизбежно происходят прерывания и исключения, обработка системных вызовов, работа планировщика процессов и многое другое. Активность ОС, которую принято называть *шумом ОС*, затрудняет анализ эффективности кода программ.

Для сокращения влияния шума ОС на анализируемый код разработан инструментарий профилирования программ [79, 102]. На Рисунке 3.8 приведена функциональная структура разработанного профилировщика. Предложенный инструментарий позволяет получить значения счетчиков производительности при выполнении заданного участка для конкретной микроархитектуры. На текущий момент реализована поддержка микроархитектуры Ivy Bridge.

Профилировщик состоит из модуля ядра Linux и runtime-библиотеки. Модуль ядра Linux запускает механизм отслеживания интересующих событий производительности при установке определенных значений в специальные моделезависимые регистры (Model-Specific Registers, MSR). Выбор событий производительности осуществляется на этапе инициализации профилировщика, а их список для конкретных микроархитектур представлен в [27, 28].

Runtime-библиотека предоставляет удобный интерфейс для выделения анализируемого участка кода. После выполнения программы результаты профилирования могут быть представлены в виде таблицы.

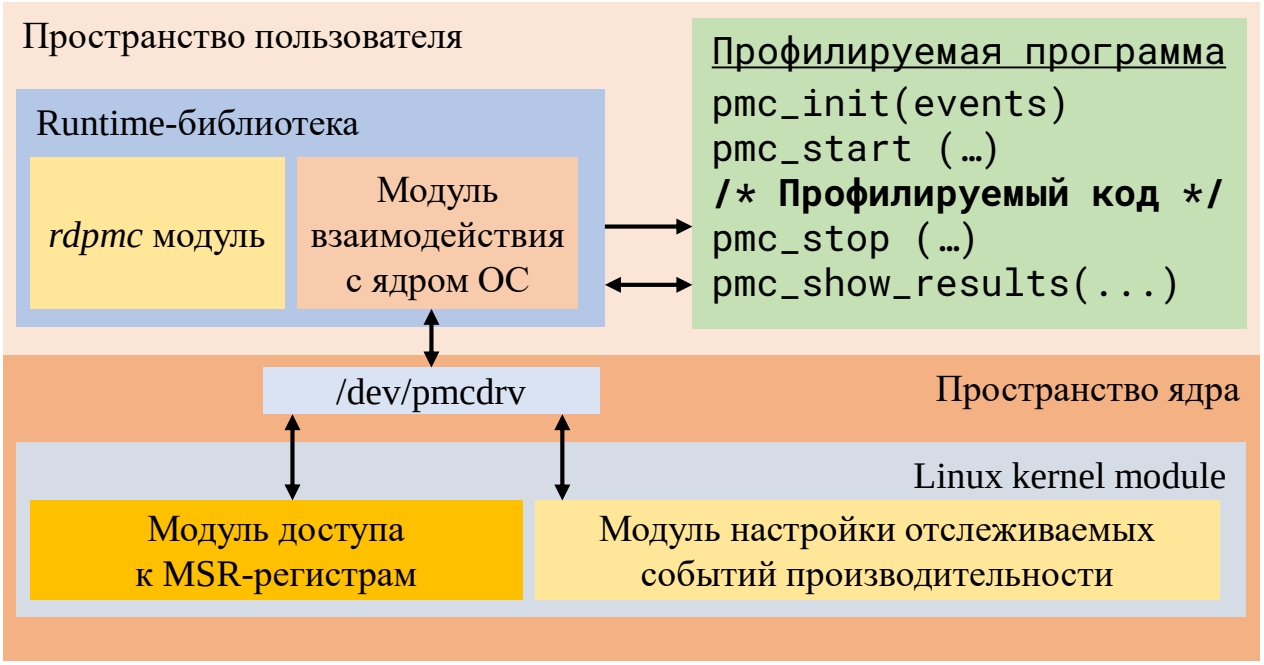


Рисунок 3.8 – Функциональная структура инструментария профилирования

3.7.3 Анализ эффективности подсистем автоматической векторизации циклов в открытых компиляторах

Для оценки эффективности подсистем векторизации в компиляторах GCC C/C++ и LLVM/Clang в работе использовался пакет Extended Test Suite for Vectorizing Compilers (ETSVC) [17], содержащий основные классы циклов, встречающихся в научных приложениях на языке C. Исходная версия пакета была разработана в конце 1980-х годов группой Дж. Донгарры и содержала 122 цикла на языке Fortran для оценки эффективности компиляторов векторных ВС [8, 10]. В 2011 году группа Д. Падуа транслировала пакет TSVC на язык программирования C и дополнила его новыми циклами [16]. Расширенная версия пакета содержит 151 цикл. Циклы разделены на категории: анализ зависимостей по данным (36 циклов), анализ потока управления и трансформация циклов (52 цикла), распознавание идиоматических конструкций (редукции, рекуррентности и т.п., 27 циклов), полнота понимания языка программирования (23 цикла). Кроме этого, в набор включены 13 контрольных циклов – тривиальные циклы, с векторизацией которых должен справиться каждый векторизующий компилятор.

Циклы оперируют с одномерными и двумерными глобальными массивами, начальные адреса которых выравнены на заданную границу (по умолчанию 16 байт). Одномерные массивы содержат $125 \cdot 1024/$`sizeof(TYPE)` элементов заданного типа `TYPE`, а двумерные – 256 элементов по каждому измерению.

Каждый цикл размещен в отдельной функции. Перед выполнением цикла в функции `init` выполняется инициализация массивов значениями, характерными для данного теста. Внешний цикл используется для увеличения времени выполнения теста (формирования статистики). Вызов пустой функций `dummy` предотвращает нежелательную оптимизацию внешнего цикла (трансформацию и вынесение внутреннего цикла за пределы внешнего, как инвариантного по отношению к внешнему). После выполнения циклов происходит вычисление и вывод на экран контрольной суммы элементов результирующего массива.

Эксперименты проводились на системе, представляющей собой сервер на базе двух процессоров Intel Xeon E5-2620 v4 (архитектура Intel 64, микроархитектура Broadwell, 8 ядер, Hyper-Threading включен, поддержка набора векторных инструкций AVX 2.0), 64 Гбайта оперативной памяти DDR4, операционная система GNU/Linux CentOS 7.3 x86-64 (ядро linux 3.10.0-514.2.2.el7).

Анализировалась работа следующих открытых компиляторов: GCC C/C++ 6.3.0, LLVM/Clang 3.9.1 [95]. Компиляция векторизованной версии пакета ETSVC выполнялась с опциями, указанными в Таблице 1 (столбец 2). Для генерации скалярной версии теста опции оптимизации сохранялись, но отключался векторизатор компилятора (столбец 3, Таблица 1).

Таблица 1: Опции, используемые при компиляции

Компилятор	Опции компиляции	Отключение векторизатора
GCC C/C++ 6.3.0	-O3 -ffast-math -fivopts -march=native -fopt-info-vec -fopt-info-vec-missed -fno-tree-vectorize	-fno-tree-vectorize
LLVM/Clang 3.9.1	-O3 -ffast-math -fvectorize -Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize	-fno-vectorize

Глобальные массивы в пакете ETSVC были выравнены на границу 32 байта. Эксперименты выполнены для массивов с элементами типов `double`, `float`, `int` и `short int`. На Рисунке 3.9 представлены результаты для данных типа `double`. Для GCC C/C++ общее количество векторизованных циклов составляет 79. При этом 34 из них были векторизованы только им. LLVM/Clang векторизовал 51 цикл, 6 из которых смог векторизовать только он. Количество невекторизованных циклов ни одним из компиляторов составило 66. Результаты векторизации для массивов с элементами типов `float`, `int` и `short int` аналогичны `double` для обоих компиляторов.

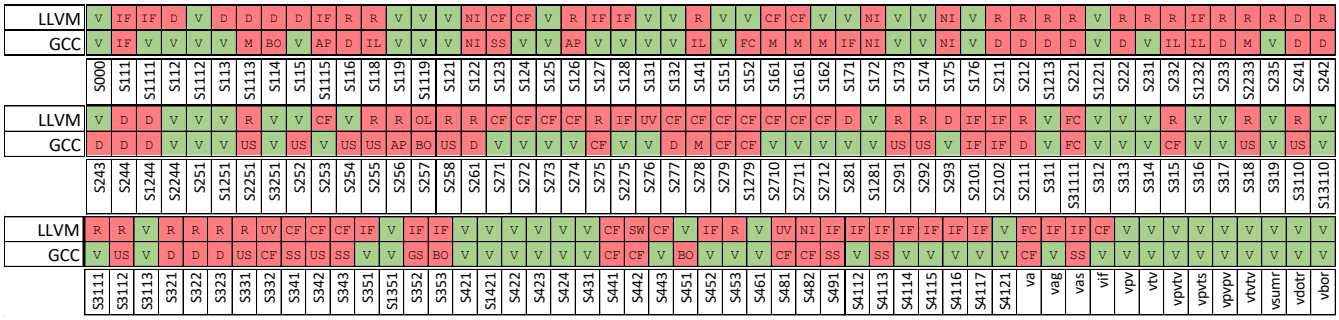


Рисунок 3.9 – Результаты векторизации циклов (архитектура Intel 64, тип данных `double`)

Таблица 2 содержит расшифровку сокращенных обозначений в результатах векторизации на Рисунке 3.9. В категории «Анализ зависимостей по данным» для типа данных `double` не были векторизованы ни одним из компиляторов 14 циклов. Проблемными в этой категории оказались циклы, содержащие линейные зависимости (рекуррентности 1-го порядка), непоследовательный доступ к элементам массива, индуктивные переменные в сочетании с условными и безусловными (`goto`) переходами внутри цикла, вложенностью циклов и переменными значениями нижней и(или) верхней границы цикла и(или) шага выполнения итераций. В последнем случае на этапе статической компиляции ни один из компиляторов не может принять однозначного решения о наличии зависимости по данным и принимает пессимистическое решение о том, что зависимость существует.

В категории «Анализ потока управления и трансформация циклов» сложными для векторизации оказались 29 циклов, требующие выполнения следующих преобразований: расщепление тела цикла, перестановка циклов, расщепление вершин в графе зависимостей по данным (для устранения

контура в графе и как следствие исключения выходных зависимостей и антизависимостей в цикле [63]) и растягивание скаляров и массивов. Среди причин неудач компиляторов: зависимость значений переменных-счетчиков итераций вложенных циклов друг от друга; линейные зависимости в теле цикла (рекуррентности 1-го порядка); условные и безусловные переходы в теле цикла; охватывающие (wraparound) переменные.

Следующие идиоматические конструкции из категории «Распознавание идиоматических конструкций» оказались проблемными при векторизации 15 циклов: рекуррентности 1-го и 2-го порядков, поиск элемента в массиве, свертка цикла и редукция с вызовом функции. Причина невекторизации циклов, содержащих рекуррентные отношения, заключается в наличии линейной зависимости по данным. В цикле, осуществлявшем поиск первого элемента в массиве, удовлетворяющего заданному условию, проблема возникла из-за прерывания вычислений в цикле безусловным переходом `goto`.

Над циклами, для которых была выполнена раскрутка (unrolling) вручную, компиляторы выполняют операцию свертки (rerolling) прежде, чем приступить к векторизации [13, 46]. Исследуемые компиляторы приняли решение, что векторизация таких циклов возможна, но будет неэффективной. Причиной этого является использование косвенной адресации при обращении к элементам массива: `X[Y[i]]`, где `X` – одномерный массив типа `float`, `Y` – указатель на целочисленный одномерный массив, `i` – переменная-счетчик итераций цикла.

Еще одна идиоматическая конструкция, вызвавшая проблемы с векторизацией – редукция, а именно нахождение суммы элементов одномерного массива. Здесь причиной невекторизации является наличие вызовов функции `test`, вычисляющей сумму 4-х элементов, начиная с того, который был ей передан в качестве аргумента.

Категория «Полнота понимания языка программирования» содержит 6 невекторизованных ни одним из компиляторов циклов. Проблемы в циклах: прерывание вычислений (вызов функции `exit` или `break`), использование оператора `switch`, условные переходы и косвенная адресация при доступе к элементам массивов. Векторизаторы, реализованные в компиляторах, не смогли выполнить анализ потока управления.

Среди контрольных циклов не были векторизованы ни одним из компиляторов 2, в которых реализовано поэлементное копирование двух одномерных массивов с использованием косвенной адресации.

Таблица 2: Сокращенные обозначения результатов векторизации

V	Цикл векторизован полностью
RV	Остаток цикла (remainder) не векторизован
IF	Векторизация возможна, но не эффективна
D	Зависимость по данным препятствует векторизации (предполагаемая линейная или нелинейная зависимость по данным в цикле)
M	Сгенерировано несколько версий цикла, из которых в процессе выполнения программы будет выбрана невекторизованная (multiversioning)
BO	Неподходящая операция или неподдерживаемая форма границы цикла (например, при использовании функций `sinf` и `cosf`)
AP	Сложный шаблон доступа к элементам массива (например, величина шага по индексу больше 1)
R	Значение, которое не может быть идентифицировано как результат редукции, используется вне цикла (наличие индуктивных переменных)
IL	Переменная-счетчик внутреннего цикла не является инвариантом (например, переменная-счетчик внутреннего цикла зависит от переменной-счетчика внешнего цикла)
NI	Невозможно вычислить количество итераций (нижняя и(или) верхняя граница цикла заданы аргументами функции)
CF	Невозможно определить направление потока управления (условные переходы внутри цикла)
SS	Цикл не подходит для векторной записи по несмежным адресам (scatter store, например, в случае упаковки двумерного массива в одномерный)
ME	Цикл с несколькими выходами невозможно векторизовать (наличие `break` или `exit` внутри цикла)
FC	Цикл содержит вызовы функций или данные, которые невозможно проанализировать
OL	Значение не может быть использовано за пределами цикла (растягивание скаляров или использование одномерных и двумерных массивов в одном цикле)
UV	Векторизатор не может понять поток управления в цикле (условные переходы внутри цикла)
SW	Наличие оператора `switch` в цикле
US	Неподдерживаемое использование в выражении (растягивание скаляров, распознавание охватывающих переменных)
GS	В базовом блоке нет сгруппированных операций записи (развернутое скалярное произведение)

Максимальное ускорение, полученное при векторизации компилятором GCC C/C++, составило 4.06, 8.1, 12.01 и 24.48 для типов `double`, `float`, `int` и `short int`, соответственно. Компилятором LLVM/Clang получены следующие значения максимального ускорения: 5.12 (`double`),

10.22 (`float`), 4.55 (`int`) и 14.57 (`short int`). Ускорение измерялось как отношение времени выполнения скалярного кода к времени выполнения векторизованного. При этом учитывались только значения ускорения, большие 1.15. Как показал анализ, значения максимального ускорения соответствуют циклам, выполняющим операции редукции (сумма, произведение, поиск минимального или максимального элемента) над элементами одномерных массивов всех типов данных. Эти циклы относятся в ETSVC к категории «Распознавание идиоматических конструкций».

3.7.4 Экспериментальное исследование возможностей микроархитектурной оптимизации кода

Анализ микроархитектурных возможностей для ускорения вычислений выполнен на примере оптимизации выполнения функции SAXPY. Функция SAXPY выполняет сумму двух векторов со скалярным произведением константы и одного из слагаемых. Экспериментальное исследование выполнено на целевом процессоре Intel Core i5 – 3320M с микроархитектурой Ivy Bridge.

Функциональная структура микроархитекутуры Ivy Bridge [28] показана на Рисунке 3.10. Ivy Bridge содержит 6 параллельных функциональных устройств (ФУ), что позволяет отправлять на выполнение до 6 микрокоманд за один такт. 2 из 6 ФУ осуществляют загрузку данных из памяти в регистр, и 1 ФУ выполняет сохранение значения регистра в память. Оставшиеся 4 ФУ реализуют арифметические и логические операции.

Конвейер верхнего уровня микроархитектуры Ivy Bridge способен декодировать до 4 команд за такт. Декодированные команды помещаются в кэш декодированных команд емкостью 1536 микрокоманд.

Для максимально эффективного задействования всех имеющихся ФУ имеется динамический планировщик. Он выбирает готовые для выполнения микрокоманды из очереди декодированных. Очередь декодированных команд может хранить до 28 микрокоманд.

В Листинге 3.6 приведен пример реализации скалярной версии функции SAXPY на языке C и ее скомпилированный код на языке Assembler.

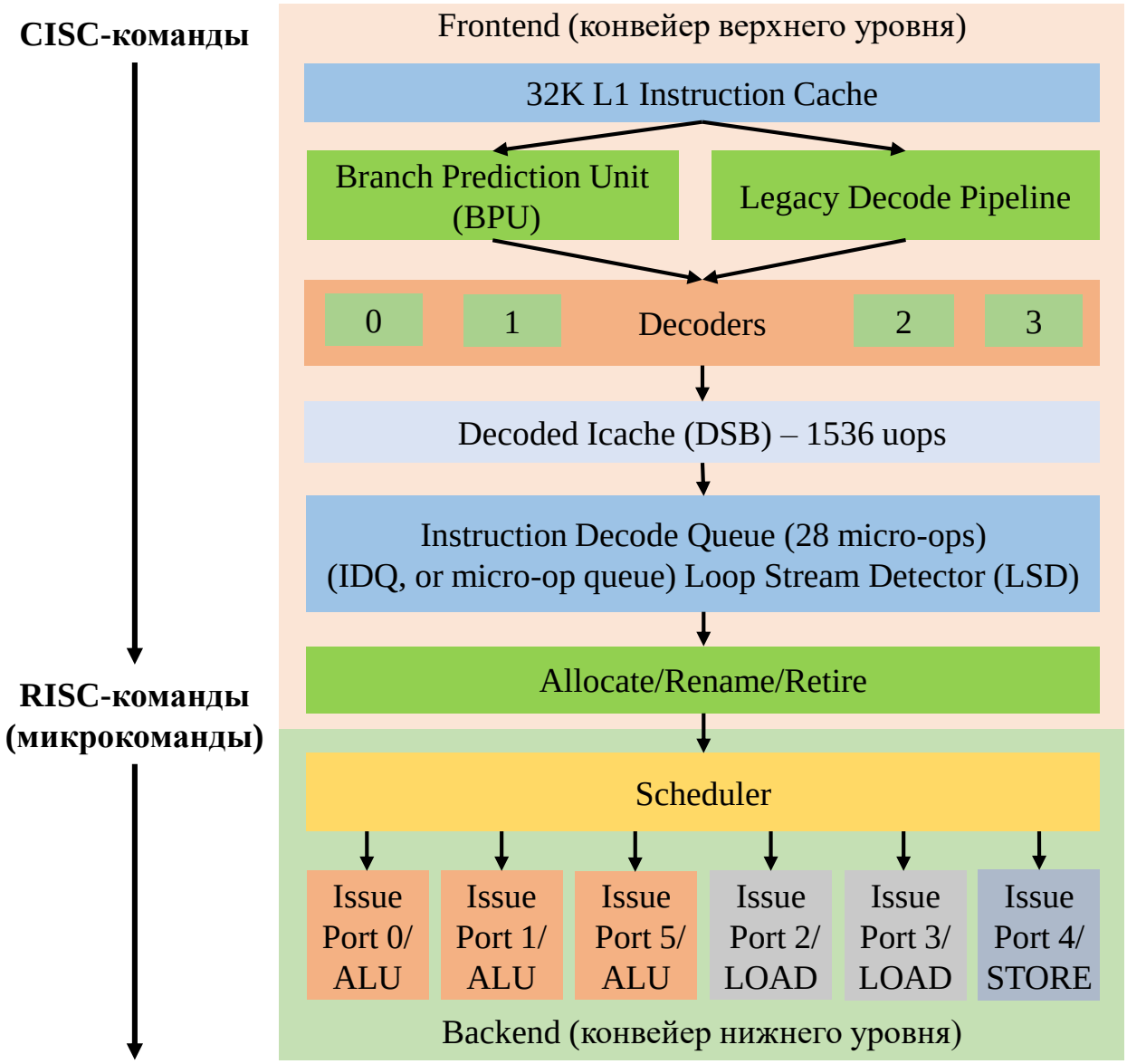


Рисунок 3.10 – Функциональная структура микроархитектуры Ivy Bridge

Таблица 3: Значения счетчиков производительности для скалярной версии функции SAXPY ($y[i] = a \cdot x[i] + y[i]; n = 100000$)

TSC	Core Cyc.	Ref. Cyc.	Instruct.	Uops	Uop p0	Uop p1	Uop p4
170494	203314	170534	700007	800010	125395	101371	100004
170512	203335	170560	700007	800010	125402	101484	100004
170515	203333	170560	700007	800010	125425	101400	100004
170518	203357	170534	700007	800010	122958	101328	100004
170533	203358	170560	700007	800010	125117	101504	100004

Данная реализация на каждой итерации выполняет 2 команды загрузки данных из памяти в регистр, 1 команду сохранения значения регистра в память,

Листинг 3.6: Скалярная версия реализации функции SAXPY

```
    /* C code */
void saxpy(float *x, float *y, float a, int n)
{
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```

```
    /* Assembler code */
L1:
    movss  (%rdi,%rax,4),%xmm1
    mulss   %xmm0,%xmm1
    addss    (%rsi,%rax,4),%xmm1
    movss   %xmm1,(%rsi,%rax,4)
    add       $0x1,%rax
    cmp      %eax,%edx
    jg          L1
```

по 1 команде умножения и сложения для вычисления SAXPY. Для организации цикла на каждой итерации требуется выполнить 1 команду сложения для увеличения счетчика на 1, 1 команду сравнения содержимого регистров и 1 команду условного перехода. В Таблице 3 представлены значения счетчиков производительности процессора, полученные при выполнении кода из Листинга 3.6.

Таблица 4: Значения счетчиков производительности для векторной версии функции SAXPY ($y[i] = a \cdot x[i] + y[i]; n = 100000$)

TSC	Core Cyc.	Ref. Cyc.	Instruct.	Uops	Uop p0	Uop p1	Uop p4
81634	97439	81718	137510	200014	58446	28220	25004
81648	97436	81744	137510	200014	58963	27868	25004
81654	97448	81744	137510	200014	57789	28651	25004
81657	97452	81744	137510	200014	25005	27764	25004
81660	97451	81770	137510	200014	25008	28199	25004

Цикл из Листинга 3.6 легко векторизуется. В Листинге 3.7 представлена реализация векторизованной версии функции SAXPY при помощи набора инструкций SSE. Команды сложения и умножения, а также команды загрузки

Листинг 3.7: Векторная версия реализации функции SAXPY

```
  /* C vectorized code */
__m128 *xx = (__m128 *)x;
__m128 *yy = (__m128 *)y;

int k = n / 4;
__m128 aa = _mm_set1_ps(a);
for (int i = 0; i < k; i++) {
    __m128 z = _mm_mul_ps(aa, xx[i]);
    yy[i] = _mm_add_ps(z, yy[i]);
}
```

```
 /* Assembler vectorized code */
L1:
   movaps (%rdi,%rax,1),%xmm1
   mulps  %xmm0,%xmm1
   addps  (%rsi,%rax,1),%xmm1
   movaps %xmm1,(%rsi,%rax,1)
   add    $0x10,%rax
   cmp    %rax,%rdx
   jne    L1
```

данных из памяти в регистр и сохранения значения регистров в память выполняются с регистрами шириною 128 бит. При условии последовательного выполнения команд теоретически может быть достигнуто максимальное ускорение до 4 раз от использования набора команд `SSE`. Однако, на практике, после векторизации функции удалось достигнуть ускорения приблизительно в 2 раза. В Таблице 4 приведены значения счетчиков производительности процессора, полученные при выполнении векторизованной функции SAXPY.

Анализ микроархитекты Ivy Bridge показал, что команды `mulps` и `addps` могут выполняться разными ФУ параллельно. Однако, использование параллелизма уровня команд в случае вычисления функции SAXPY невозможно, так как присутствует зависимость по данным между командами `mulps` и `addps`.

Конвейеризация цикла `for` функции SAXPY, представленной в Листинге 3.7, позволит избавиться от зависимости по данным между командами `mulps` и `addps`. Листинг 3.8 содержит код конвейеризированной версии цикла. На каждой итерации команда `mulps` выполняется над данными i-ой

Листинг 3.8: Конвейеризация векторной версии реализации функции SAXPY

```
1  /* Assembler of pipelined and vectorized code */
2     movaps (%rdi), %xmm2
3     movaps 16(%rdi), %xmm1
4     movaps (%rsi), %xmm3
5     mulps %xmm0, %xmm2
6  .L3:
7     mulps %xmm0, %xmm1
8     addps %xmm2, %xmm3
9     movaps %xmm3, (%rsi, %rax)
10    movaps %xmm1, %xmm2
11    movaps 16(%rsi, %rax), %xmm3
12    movaps 32(%rdi, %rax), %xmm1
13
14    addq    $16, %rax
15    cmpq    %rdx, %rax
16    jne .L3
17
18    mulps %xmm0, %xmm1
19    addps %xmm2, %xmm3
20    movaps %xmm3, (%rsi, %rax)
```

Таблица 5: Значения счетчиков производительности для конвейеризированной векторной версии функции SAXPY ($y[i] = a \cdot x[i] + y[i]; n = 100000$)

TSC	Core Cyc.	Ref. Cyc.	Instruct.	Uops	Uop p0	Uop p1	Uop p4
77192	98012	77220	225009	225061	25452	29478	25009
77194	98014	77220	225009	225061	25330	28895	25006
77194	98015	77246	225009	225061	25346	29046	25008
77197	98014	77220	225009	225061	25221	28405	25006
77200	98018	77246	225009	225061	25256	28364	25009

итерации цикла, а команд `addps` и сохранение результата – над данными $i - 1$ итерации. Данное преобразование цикла не нарушает оригинальную семантику программы, так как отсутствует зависимость по данным между итерациями цикла. Однако, требуется наличие пролога и эпилога цикла для

Листинг 3.9: Конвейеризация векторной версии реализации функции SAXPY с раскрученным циклом на 2 итерации

```
/* Assembler of unrolled vectorized code  */
    /* Prologue loop */
    movaps (%rdi), %xmm2
    movaps 16(%rdi), %xmm1
    movaps (%rsi), %xmm3
    mulps %xmm0, %xmm2
.L3:
    mulps %xmm0, %xmm1
    addps %xmm2, %xmm3

    movaps 16(%rsi, %rax), %xmm4
    movaps 32(%rdi, %rax), %xmm2
    movaps %xmm3, (%rsi, %rax)
    addps %xmm1, %xmm4
    mulps %xmm0, %xmm2

    movaps 48(%rdi, %rax), %xmm1
    movaps 32(%rsi, %rax), %xmm3
    movaps %xmm4, 16(%rsi, %rax)

    addq    $32, %rax
    cmpq    %rdx, %rax
    jne .L3
    /* Epilogue loop */
    mulps %xmm0, %xmm1
    addps %xmm2, %xmm3
    movaps %xmm3, (%rsi, %rax)
    movaps 16(%rsi, %rax), %xmm4
    addps %xmm1, %xmm4
    movaps %xmm4, 16(%rsi, %rax)
```

подготовки операндов нулевой итерации и сохранения результата последней итерации.

В Таблице 5 приведены значения счетчиков производительности процессора, полученные во время выполнения реализации из Листинга 3.8. После конвейеризации цикла время выполнения программы удалось сократить на 6% по сравнению с векторной версией SAXPY, приведенной в Листинге 3.7.

Таблица 6: Значения счетчиков производительности для конвейеризированной векторной версии функции SAXPY с раскрученным циклом на 2 итерации ($y[i] = a \cdot x[i] + y[i]; n = 100000$)

TSC	Core Cyc.	Ref. Cyc.	Instruct.	Uops	Uop p0	Uop p1	Uop p4
73014	92707	73060	162508	175012	25007	27359	25003
73016	92704	73060	162508	175012	25006	27401	25004
73019	92705	73034	162508	175012	46635	27571	25004
73019	92711	73060	162508	175012	47005	27345	25004
73022	92721	73086	162508	175012	25010	27588	25004

В реализации SAXPY из Листинга 3.8 присутствует одно избыточное перемещение значения из регистра `xmm1` в `xmm2` (Листинг 3.8 строка 10). Раскрутка цикла на 2 итерации позволит избежать этого перемещения. Результат трансформации цикла приведен в Листинге 3.9. Данное преобразование позволило сократить время выполнения функции SAXPY на 11% по сравнению с конвейеризированной версией Листинга 3.8. Таблица 6 содержит значения счетчиков производительности процессора для данной версии.

Таблица 7: Значения счетчиков производительности для AVX версии функции SAXPY ($y[i] = a \cdot x[i] + y[i]; n = 100000$)

TSC	Core Cyc.	Ref. Cyc.	Instruct.	Uops	Uop p0	Uop p1	Uop p4
81339	97016	81406	75016	100022	33490	15376	12504
81345	97026	81406	75016	100022	33467	15357	12504
81355	97041	81406	75016	100022	33462	15597	12504
81366	97046	81406	75016	100022	33552	15421	12504
81403	97106	81458	75016	100022	32424	15554	12504

Кроме набора инструкций `SSE`, микроархитектура Ivy Bridge поддерживает расширение архитектуры набора команд `AVX`, которое позволяет выполнять векторные команды над регистрами шириною 256 бит, что вдвое больше ширины регистров `SSE`. Теоретическое максимальное ускорение от использования набора команд `AVX` составляет 2 раза относительно использования набора инструкций `SSE`. В Листинге 3.10 приведена реализация

AVX версии функции SAXPY. Использование регистров большей ширины и команд, способных выполнять над ними арифметические операции, не принесло ожидаемого ускорения, так как в данном примере основным фактором, влияющим на ускорение, является загрузка данных из памяти в регистр и сохранение значения регистра в память. Микроархитектура Ivy Bridge содержит 2 ФУ для загрузки данных из памяти в регистр. Пропускная способность каждого ФУ 128 бит за цикл, что делает неэффективным использование 256 битных регистров и команд над ними. В Таблице 7 представлены значения счетчиков производительности процессора после выполнения AVX версии функции SAXPY.

Листинг 3.10: AVX версия реализации функции SAXPY

```
/* C vectorized AVX code */
__m256 *xx = (__m256 *)x;
__m256 *yy = (__m256 *)y;

int k = n / 8;
__m256 aa = _mm256_set1_ps(a);
for (int i = 0; i < k; i++) {
    __m256 z = _mm256_mul_ps(aa, xx[i]);
    yy[i] = _mm256_add_ps(z, yy[i]);
}

for (int i = k * 8; i < n; i++)
    y[i] = a * x[i] + y[i];
```

```
/* Assembler vectorized AVX code */
    lea     -0x1(%r8),%ecx
    xor     %eax,%eax
    add     $0x1,%rcx
    shl     $0x5,%rcx
    nopl    0x0(%rax)
L2:
    vmulps  (%rdi,%rax,1),%ymm2,%ymm1
    vaddps  (%rsi,%rax,1),%ymm1,%ymm1
    vmovaps %ymm1,(%rsi,%rax,1)
    add     $0x20,%rax
    cmp     %rax,%rcx
    jne     L2
L1:
    shl     $0x3,%r8d
    cmp     %r8d,%edx
    movslq  %r8d,%rax
    jle     L3
    nopl    0x0(%rax)
L4:
    vmulss  (%rdi,%rax,4),%xmm0,%xmm1
    vaddss  (%rsi,%rax,4),%xmm1,%xmm1
    vmovss  %xmm1,(%rsi,%rax,4)
    add     $0x1,%rax
    cmp     %eax,%edx
    jg      L4
L3:
    vzeroupper
```

3.7.5 Экспериментальное исследование эффективности векторизации алгоритма умножения матриц

Распространенным шаблоном вычислений в библиотеках линейной алгебры является гнездо из трех циклов ($i = 0...M-1; j = 0...N-1; k = 0...K-1$), в котором выполняются арифметические операции над элементами двумерных массивов. Здесь i, j, k – индуктивные переменные, являющиеся счетчиками циклов. Такое гнездо характеризуется наличием инструкций обработки элементов массивов только в самом внутреннем вложенном цикле и пространством итераций, образующим прямоугольный параллелепипед с размерностями M, N, K.

Типичным представителем гнезда из трех циклов является алгоритм умножения матриц GEMM, выполняющий вычисления вида: $A = \alpha \cdot B \cdot C + \beta \cdot A$, где A, B и C – двумерные массивы с размерностями $M \times N$, $M \times K$ и $K \times M$, соответственно, а α и β – скалярные коэффициенты. Для алгоритма умножения матриц по определению $\alpha = \beta = 1$. Если $M = N = K, A$, то B и C – квадратные матрицы. Частным случаем GEMM является алгоритм DGEMM, оперирующий матрицами, элементами которых являются числа с плавающей запятой двойной точности (тип данных `double`).

Алгоритм умножения матриц DGEMM можно реализовать в виде двух последовательных версий (см. Рисунок 3.11). В первой версии циклы выполняются в порядке $i \rightarrow j \rightarrow k$, а во второй – в порядке $i \rightarrow k \rightarrow j$. На Рисунке 3.11 порядок следования циклов представлен в виде кортежей из трех индуктивных переменных $< i, j, k >$ и $< i, k, j >$. Каждая из этих версий может быть векторизована тремя способами: 1) только по самому внутреннему вложенному циклу; 2) по среднему вложенному циклу и 3) по обоим этим циклам. На Рисунке 3.11 в кортежах индуктивная переменная векторизуемого цикла помечена символом «$\rightarrow$». Шаг S выполнения итераций для векторизуемых циклов равен количеству элементов типа данных `double`, помещающихся в векторный регистр целевой архитектуры вычислительной системы. Например, для архитектуры с короткими векторными регистрами, поддерживающей набор векторных инструкций Intel AVX, $S = 4$.

Проведено экспериментальное исследование эффективности предложенных версий алгоритма умножения матриц. Исследование проводилось на

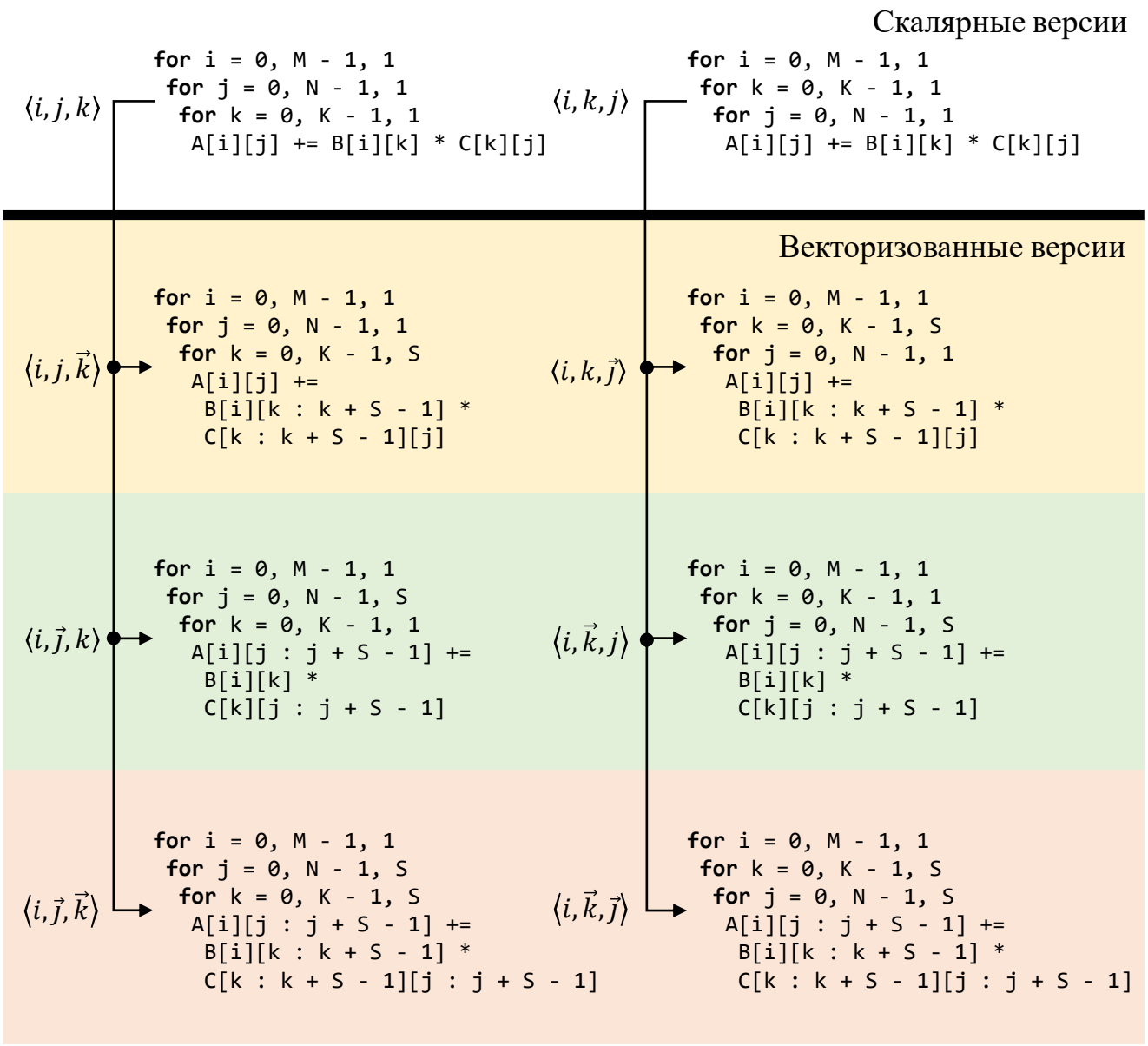
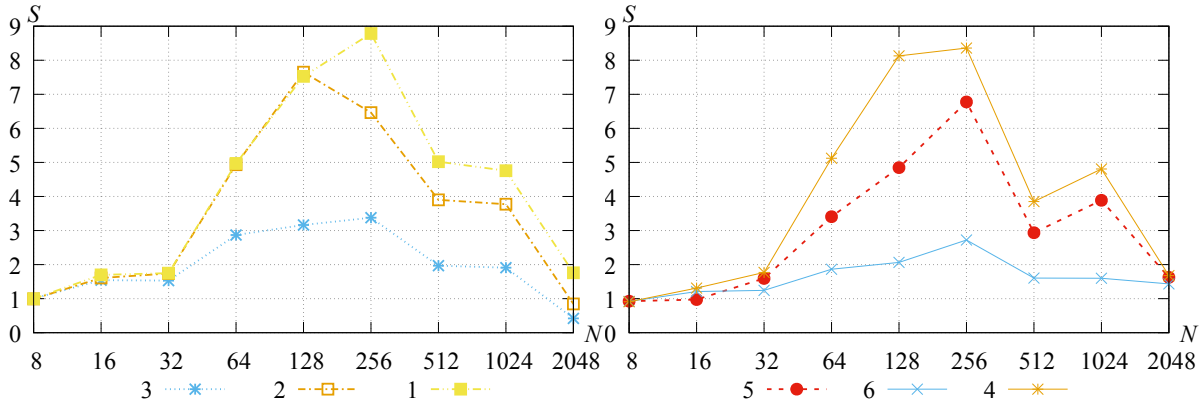


Рисунок 3.11 – Скалярные и векторизованные версии алгоритма умножения матриц DGEMM по определению для двумерных массивов $A[M][N]$, $B[M][K]$ и $C[K][N]$ (S – количество элементов массивов, помещающихся в векторный регистр)

двух ВС с общей памятью. Первая ВС укомплектована двумя процессорами Intel Xeon E5-2620 v3 (микроархитектура Haswell), а вторая – двумя процессорами Intel Xeon E5-2620 v4 (микроархитектура Broadwell). Обе ВС имеют 64 Гбайта ОЗУ.

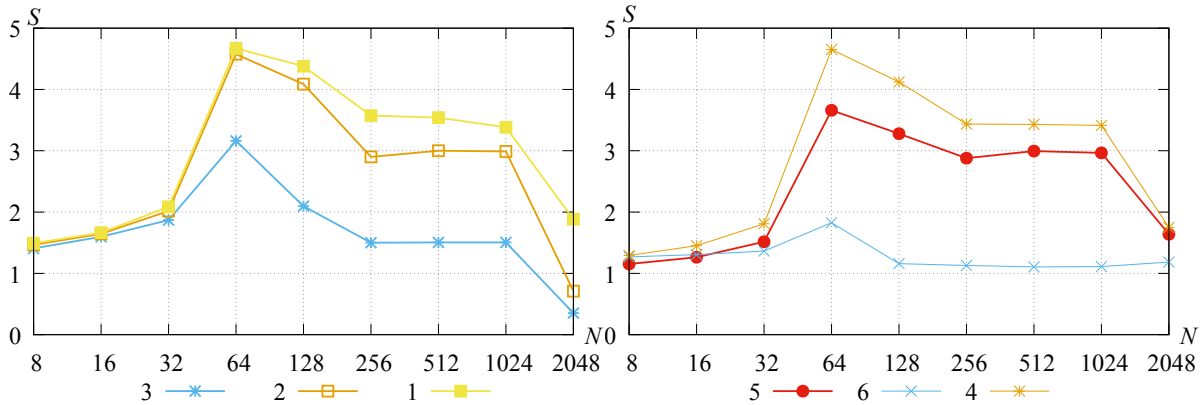
Для сокращения влияния сторонних факторов на выполнение тестов в ходе экспериментов учитывались особенности NUMA-архитектуры. Запуск процесса, выполняющего тест, производился на том же самом NUMA-узле, на котором происходило выделение памяти.



(а) Версия $< i,j,k >$ (б) Версия $< i,k,j >$

1 – $< i, \vec{j}, \vec{k} >$ с непоследовательным доступом к элементам матрицы C;
2 – $< i, \vec{j}, \vec{k} >$ с использованием команды `vbroadcastsd`; 3 – $< i, \vec{j}, k >$;
4 – $< i, \vec{k}, \vec{j} >$ с использованием команды `vbroadcastsd`; 5 – $< i, k, \vec{j} >$;
6 – $< i, \vec{k}, \vec{j} >$ с непоследовательным доступом к элементам матрицы C;

Рисунок 3.12 – Зависимость ускорения выполнения теста на микроархитектуре Broadwell от размерности матриц



(а) Версия $< i,j,k >$ (б) Версия $< i,k,j >$

1 – $< i, \vec{j}, \vec{k} >$ с непоследовательным доступом к элементам матрицы C;
2 – $< i, \vec{j}, \vec{k} >$ с использованием команды `vbroadcastsd`; 3 – $< i, \vec{j}, k >$;
4 – $< i, \vec{k}, \vec{j} >$ с использованием команды `vbroadcastsd`; 5 – $< i, k, \vec{j} >$;
6 – $< i, \vec{k}, \vec{j} >$ с непоследовательным доступом к элементам матрицы C;

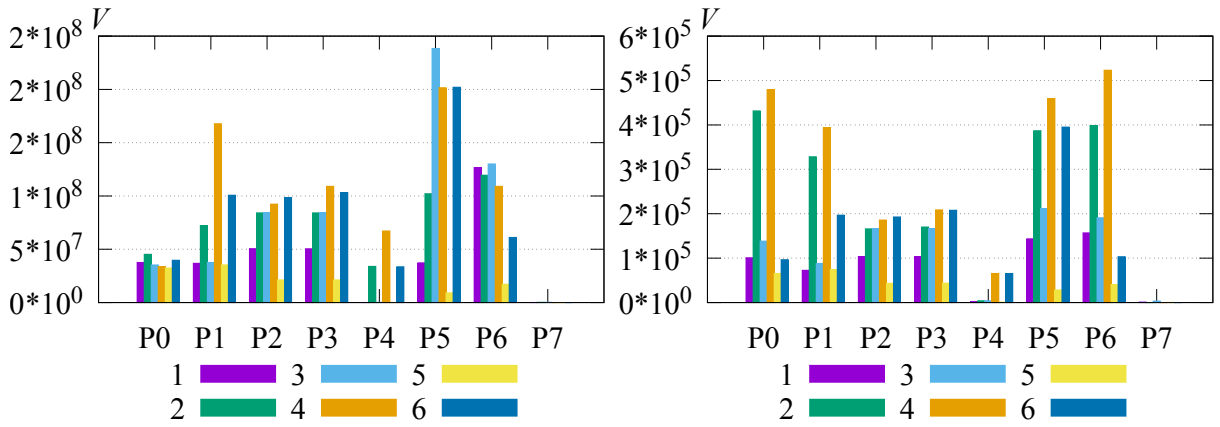
Рисунок 3.13 – Зависимость ускорение выполнения теста на микроархитектуре Haswell от размерности матриц

Таблица 8: Значения счетчиков производительности для микроархитектуры Broadwell ($N = 256$)

Версия DGEMM	**Ускорение**	**L1d-load-misses**	**LLC-load-misses**	**mem-loads**	**mem-stores**	**cycles**	**instructions**	**CPI**	**branch-instructions**	**branch-misses**
$< i, j, k >$: 1 (Рисунок 3.12а)	8.79	2115816	0	5308466	16396	9850108	14059112	0.7	1065249	267
$< i, j, k >$: 2 (Рисунок 3.12а)	6.46	2143096	0	6324281	16396	13456733	12682856	1.06	532769	275
$< i, j, k >$: 3 (Рисунок 3.12а)	3.38	4263691	3456	12615758	32780	25802907	37914476	0.68	4210981	16901
$< i, k, j >$: 4 (Рисунок 3.12б)	8.36	2118641	0	5308466	1048588	10355924	13780585	0.75	1065249	270
$< i, k, j >$: 5 (Рисунок 3.12б)	6.77	2098595	1826	8454194	4194316	12790121	33949289	0.38	4260129	520
$< i, k, j >$: 6 (Рисунок 3.12б)	2.72	2178667	185	20987997	4194317	32114941	74548847	0.43	5259559	16657

Таблица 9: Значения счетчиков производительности для микроархитектуры Haswell ($N = 64$)

Версия DGEMM	**Ускорение**	**L1d-load-misses**	**LLC-load-misses**	**mem-loads**	**mem-stores**	**cycles**	**instructions**	**CPI**	**branch-instructions**	**branch-misses**
$< i, j, k >$: 1 (Рисунок 3.13а)	4.67	8684	0	86038	1034	93663	240100	0.39	17501	72
$< i, j, k >$: 2 (Рисунок 3.13а)	4.57	9884	0	100368	1034	97662	203236	0.48	8797	71
$< i, j, k >$: 3 (Рисунок 3.13а)	3.16	19336	0	198678	2060	213945	600612	0.36	66653	1095
$< i, k, j >$: 4 (Рисунок 3.13б)	4.65	8730	0	86038	16394	91136	222693	0.41	17501	72
$< i, k, j >$: 5 (Рисунок 3.13б)	3.66	9643	0	135190	65546	144023	549349	0.26	69725	72
$< i, k, j >$: 6 (Рисунок 3.13б)	1.83	9282	0	328728	65547	483118	1169893	0.41	83037	78



(а) Broadwell ($N = 256$) (б) Haswell ($N = 64$)

1 – $< i, \vec{j}, \vec{k} >$ с непоследовательным доступом к элементам матрицы C;
2 – $< i, \vec{j}, \vec{k} >$ с использованием команды `vbroadcastsd`; 3 – $< i, \vec{j}, k >$;
4 – $< i, \vec{k}, \vec{j} >$ с использованием команды `vbroadcastsd`; 5 – $< i, k, \vec{j} >$;
6 – $< i, \vec{k}, \vec{j} >$ с непоследовательным доступом к элементам матрицы C;
Рисунок 3.14 – Диаграмма распределения микрокоманд по портам (ФУ)

Графики зависимости ускорения различных реализаций теста DGEMM от количества строк и столбцов N используемых матриц приведены на Рисунке 3.13 для микроархитектуры Haswell и на Рисунке 3.12 для Broadwell. В работе [73] представлены реализации приведенных версий алгоритма DGEMM. Наибольшее ускорение получено версией 1 (Рисунки 3.12а, 3.13а), достигнув максимального значения при размере массивов 64×64 элемента типа `double` для микроархитектуры Haswell ($S = 4.67$), и при 256×256 элементов для Broadwell ($S = 8.79$). В этой версии выполнена векторизация циклов j и k. Отличительной особенностью данной реализации является работа с транспонированной матрицей C. Это позволило уменьшить количество кэш-промахов при выполнении загрузки данных из L1 кэша (8684 кэш-промахов при $N = 64$, микроархитектура Haswell, и 2115816 кэш-промахов при $N = 256$, микроархитектура Broadwell), а также сократить число операций загрузки и сохранения данных из/в память (86038/1034 load/store операций при $N = 64$, микроархитектура Haswell, и 5308466/16396 load/store операций при $N = 256$, микроархитектура Broadwell). Детальный отчет, содержащий значения счетчиков производительности основных версий алгоритма умножения матриц, приведен в Таблице 9 для микроархитектуры Haswell и в Таблице 8 для Broadwell. Количество V микрокоманд, назначенных на ФУ P0-P7 (порты выдачи микрокоманд) вычислительного ядра для

микроархитектур Broadwell и Haswell, показано на Рисунках 3.14а и 3.14б соответственно.

3.8 Выводы

1. Предложен метод сокращения числа ложных конфликтов в многопоточных программах на базе программной транзакционной памяти. В основе метода лежит подбор (суб)оптимальных параметров таблиц обнаружения конфликтов в реализации транзакционной памяти по результатам предварительного профилирования целевой программы.
2. Выполнена программная реализация метода сокращения числа ложных конфликтов в расширении компилятора GCC. Использование предложенного метода позволяет сократить время выполнения параллельных программ в среднем на 20%, что экспериментально показано на тестовых программах из пакета STAMP.
3. Для известных алгоритмов автоматической векторизации циклов в открытых компиляторах GCC и LLVM/Clang выявлены классы трудно векторизуемых циклов из тестового набора ETSVC. Установлено, что на архитектуре Intel 64 известные алгоритмы способны векторизовать от 34% до 52% циклов пакета ETSVC. Построенное подмножество циклов составляет базисный набор для анализа эффективности ядер автовекторизаторов оптимизирующих компиляторов для векторных процессоров класса «регистр-регистр».
4. Создан инструментарий анализа эффективности использования микроархитектурных возможностей ядер суперскалярных процессоров ВС. В отличие от известных пакетов, предложенные программные средства позволяют анализировать загрузку суперскалярного конвейера архитектуры Intel 64 потоком инструкций с точностью до нескольких команд ассемблера.
5. Выполнено экспериментальное исследование архитектурных возможностей ускорения вычислений на многопроцессорных ВС с общей памятью. Используя возможности параллелизма данных на уровне векторных АЛУ, было достигнуто ускорение от 4.67 до 8.79 раз.

В ходе экспериментов показано, что использование только микроархитектурных возможностей вычислительного ядра процессора позволяет сократить время выполнение параллельной программы на 11%.

Глава 4. Мультикластерная ВС

Глава посвящена развитию мультикластерной ВС с иерархическим параллелизмом. В конфигурировании ВС автор принимал активное участие. Выполнено описание стека программного и аппаратного обеспечения подсистем ВС.

4.1 Архитектура ВС

Центром параллельных вычислительных технологий федерального государственного бюджетного образовательного учреждения высшего образования «Сибирский государственный университет телекоммуникаций и информатики» (СибГУТИ) создана и развивается мультикластерная ВС с многоуровневым параллелизмом. Автор принимал активное участие в разработке и конфигурировании каждой подсистемы ВС. Основное назначение данной ВС – исследование архитектуры и тестирование и отладка инструментария параллельного мультипрограммирования, моделирование (отказоустойчивых) распределенных вычислительных технологий и подготовка квалифицированных специалистов в области распределенных и параллельных вычислительных технологий.

Конфигурация мультикластерной ВС представлена на Рисунке 4.1. В ее состав входят кластер Jet и кластер Oak. Кластер Jet укомплектован 18 двухпроцессорными SMP-узлами на базе Intel Xeon E5420 (микроархитектура Intel Core). В кластере имеются 2 коммуникационные сети: вычислительная сеть и служебная сеть. Коммуникационные сети построены на базе технологии Gigabit Ethernet и изолированы друг от друга. Пиковая производительность кластера – 1,44 TFLOPS.

Кластер Oak состоит из 3 подсистем:

1. Подсистема из 6 двухпроцессорных NUMA-узлов на базе Intel Xeon E5620 (микроархитектура Westmere). Взаимодействие ВУ при

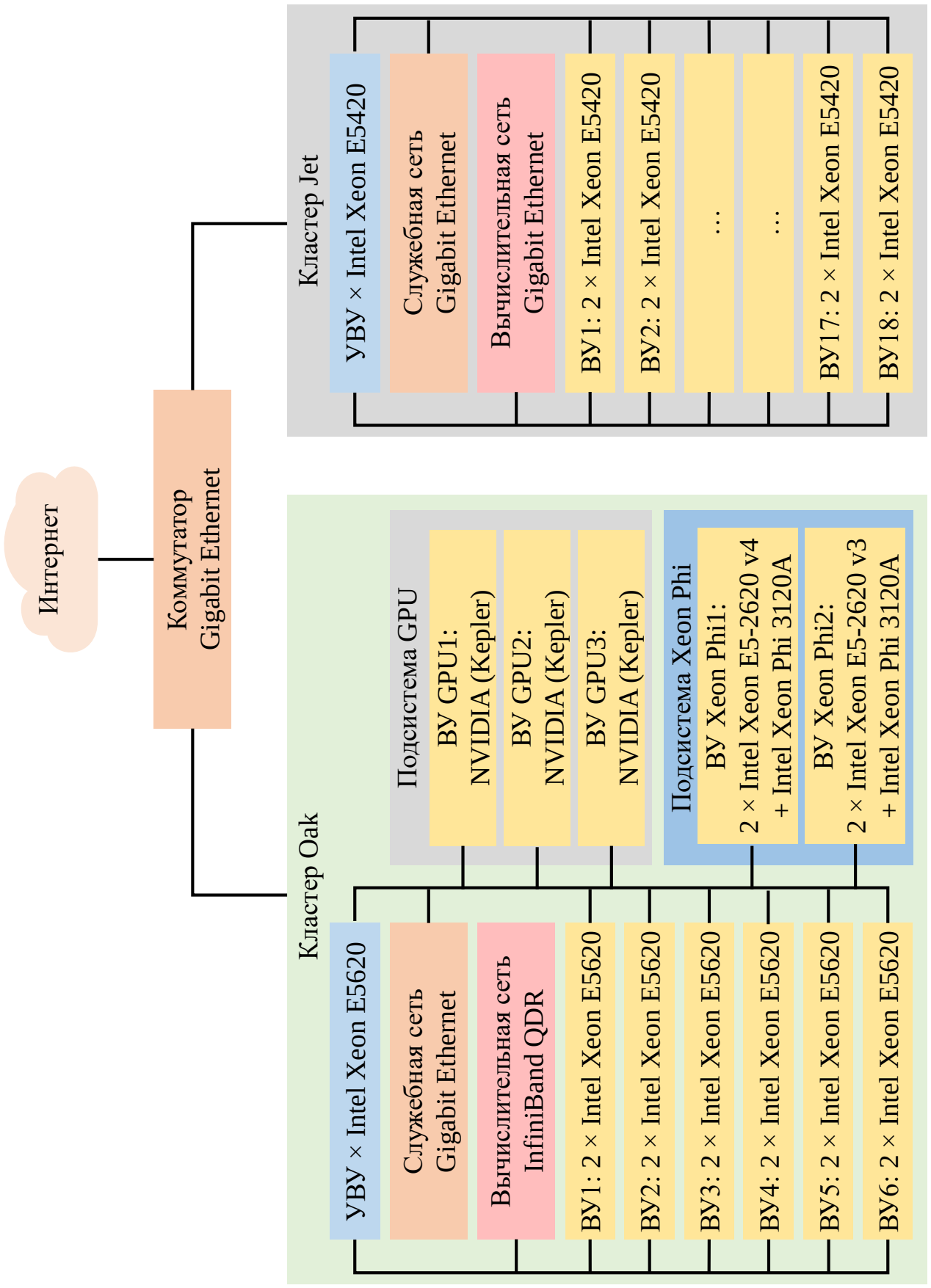


Рисунок 4.1 – Конфигурация мультикластерной ВС

выполнении параллельных программ осуществляется через вычислительную сеть, построенную по технологии InfiniBand QDR с пропускной способность 40 Гбит/с.

2. Подсистема графических ускорителей, оснащенная 3 графическими процессорами NVIDIA (микроархитектура Kepler), содержащими 1536 вычислительных ядер.
3. Подсистема из 2 двухпроцессорных NUMA-узлов на базе Intel Xeon E5-2620 (микроархитектуры Broadwell и Haswell), каждый из которых оснащен сопроцессором Intel Xeon Phi (поколение Knights Corner), содержащим 56 вычислительных ядер (4 $\times$ 56 аппаратных потока).

Для взаимодействия узлов подсистем в кластере Oak настроена служебная сеть на базе технологии Gigabit Ethernet. Пиковая производительность кластера – 460 GFLOPS.

4.2 Программное обеспечение мультикластерной ВС

4.2.1 Стандартное программное обеспечение

На кластере Jet используется дистрибутив ОС GNU/Linux Fedora, а на кластере Oak – CentOS. Управление ресурсами ВС осуществляется при помощи систем пакетной обработки заданий TORQUE на кластере Jet и SLURM – на кластере Oak.

На Рисунке 4.2 представлен стек программного обеспечения для организации функционирования мультикластерной ВС с иерархическим параллелизмом. Стек содержит в себе основные компоненты системного и прикладного программного обеспечения. Алгоритмы и программный инструментарий, предложенные автором, внедрены или задействуют каждый уровень стека от стандарта языков программирования до модулей ядра ОС Linux.

Для разработки параллельных программ на мультикластерной ВС имеются:

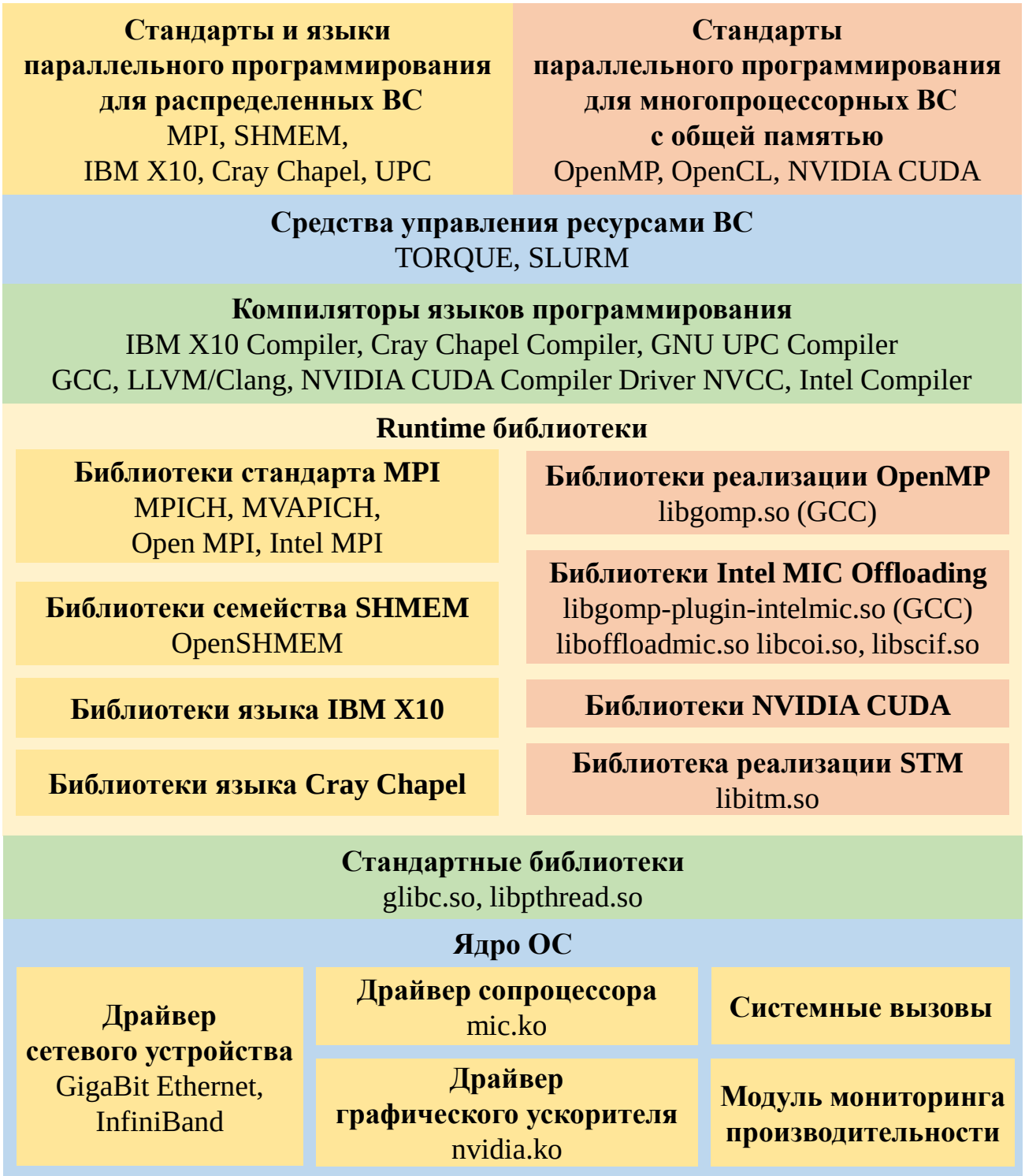


Рисунок 4.2 – Стек программного обеспечения для организации функционирования мультикластерной ВС

– стандарты многопоточного программирования: OpenMP, OpenCL;
– технологии программирования графических ускорителей: NVIDIA CUDA, стандарты OpenCL, OpenACC;
– средства многопоточного программирования сопроцессоров Intel Xeon Phi: компилятор Intel, кросс-компилятор GCC;

– библиотеки стандарта MPI: MPICH, MVAPICH, Open MPI, Intel MPI;
– компиляторы языков семейства PGAS: Cray Chapel, IBM X10.

4.2.2 Средства создания PGAS-программ

Алгоритм *Array Prelaod* трансформации циклических конструкций передачи потока управления подчиненным ЭМ, представленный в Главе 2, программно реализован в компиляторе языка IBM X10. На Рисунке 4.3 представлена функциональная структура компилятора.

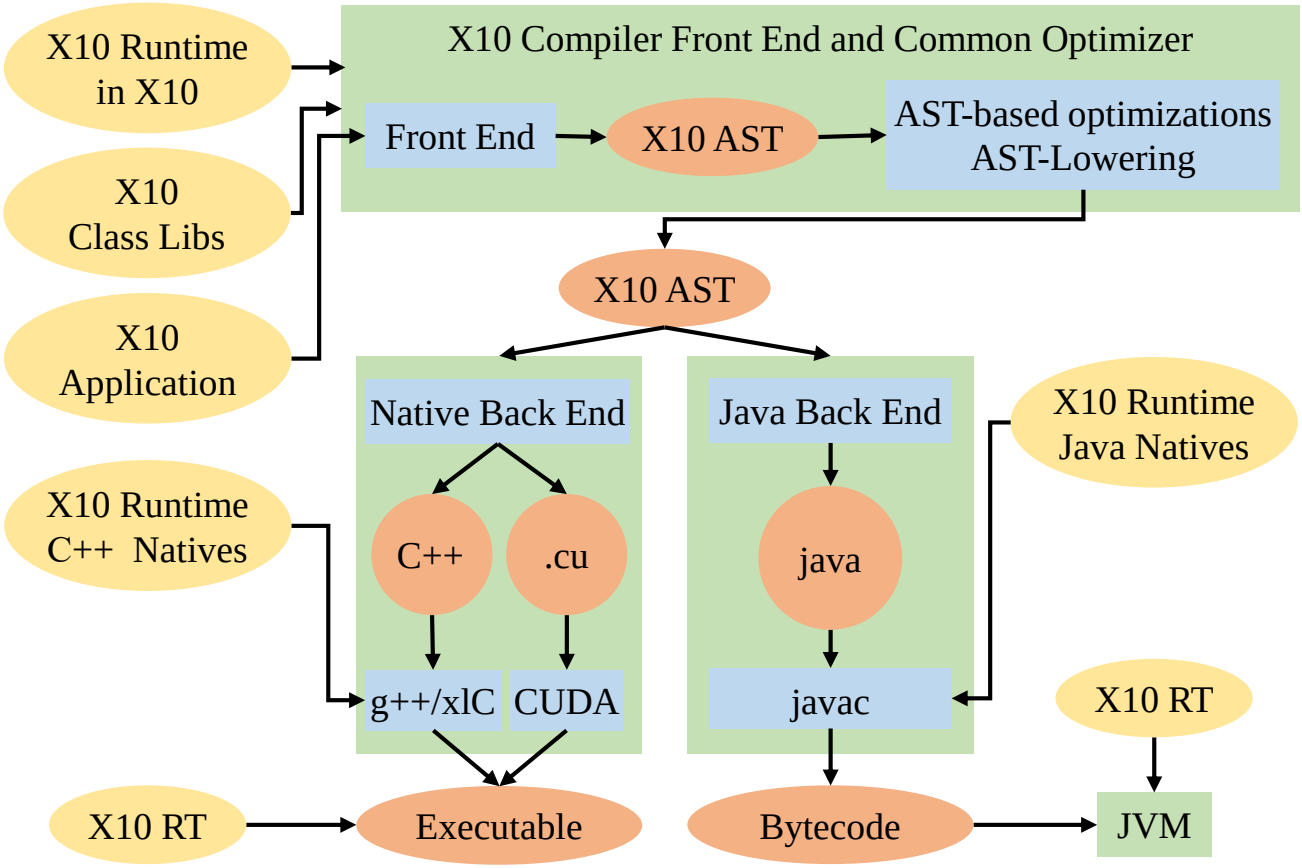


Рисунок 4.3 – Функциональная структура компилятора языка IBM X10

Компилятор языка IBM X10 является транспилером (source-to-source компилятором), транслирующим программу, написанную на языке IBM X10 в программу на языке C++ или Java. Компилятор содержит в себе:

- стандартные библиотеки языка IBM X10, содержащие реализацию стандартных алгоритмов и структур данных, например, реализации списков, хеш-таблиц, сортировок и др.;
- библиотеки на языке IBM X10, представляющие операции над основными понятиями модели PGAS в реализации IBM X10, например, области (Place), потоки (Activity) и др.;
- библиотеки на языках C/C++ или Java организующие работу runtime-системы языка, например, обращение к коммуникационным операциям;
- Модуль оптимизации и трансляции X10-программы в промежуточный язык C++ или java (X10 Frontend compiler);
- Модуль генерации исполняемого кода или jvm байт-кода (X10 Backend compiler).

В процессе компиляции IBM X10 программы Frontend компилятора генерирует абстрактное синтаксическое дерево программы, после этого выполняет над ним преобразования по упрощению и оптимизации АСД. Оптимизационные преобразования реализованы в виде проходов компилятора по АСД. Предложенный алгоритм *Array Preload* реализован в виде отдельного прохода в компиляторе IBM X10.

После всех преобразований Frontend компилятора по полученной версии оптимизированного АСД генерирует P-программу на языке C++ или Java и передает управление модулю генерации исполняемого кода или JVM байт-кода (Backend компилятора).

Backend компилятора IBM X10 выполняет архитектурно-ориентированную оптимизацию P-программы. На данном этапе могут быть применимы алгоритмы и подходы к оптимизации выполнения программ, предложенные автором в Главе 3. На выходе Backend компилятора генерируется исполняемый код программы или JVM байт-код. Выбор способа Backend компиляции (Native Backend или Java Backend) осуществляется программистом.

4.3 Выводы

1. Описана функциональная структура мультикластерной ВС с иерархическим параллелизмом, в конфигурировании которой автор принимал активное участие.
2. Описан стек программного обеспечения для организации функционирования мультикластерной ВС.
3. Выполнено развитие программно-аппаратной конфигурации мультикластерной ВС. В состав системы введены сегменты на базе вычислительных узлов с сопроцессорами Intel Xeon Phi и графическими процессорами NVIDIA. Программный инструментарий системы расширен разработанными автором пакетами оптимизации использования многоуровневого параллелизма ВС в параллельных программах.

Заключение

В диссертации предложены архитектурно-ориентированные методы и алгоритмы организации функционирования ВС и оптимизации выполнения параллельных программ на них.

1. Для ВС с многоуровневым параллелизмом разработаны средства оптимизации циклического доступа к информационным массивам в параллельных программах на базе модели разделенного глобального адресного пространства.

1.1. Предложен алгоритм *Array Preload* трансформации конструкций циклической передачи потока управления подчиненным элементарным машинам, сокращающий время выполнения программ за счет опережающего копирования информационных массивов. В отличие от известных, разработанный метод применим к PGAS-программам, в которых на этапе компиляции неизвестно множество используемых элементов массивов.

1.2. В моделях параллельных вычислений LogP, LogGP и Hockney построены оценки эффективности выполнения формируемого алгоритмом *Array Preload* кода, показывающие отсутствие функциональной зависимости времени его выполнения от количества итераций циклов.

1.3. Выполнена реализация алгоритма в виде расширения компилятора языка IBM X10. По сравнению со стандартным алгоритмом предложенный позволяет на кластерных ВС с сетями связи Gigabit Ethernet и InfiniBand QDR сократить время выполнения циклического доступа к элементам массивов в 1.2–2.5 раз.

2. Для многопроцессорных вычислительных узлов с общей памятью разработаны и исследованы методы оптимизации выполнения многопоточных программ.

2.1. Предложен метод сокращения числа ложных конфликтов в многопоточных программах на базе программной транзакционной памяти. В основе метода лежит подбор (суб)оптимальных параметров таблиц обнаружения конфликтов в реализации транзакционной памяти по результатам предварительного профилирования целевой программы.

2.2. Выполнена программная реализация метода сокращения числа ложных конфликтов в расширении компилятора GCC. Использование предложенного метода позволяет сократить время выполнения параллельных программ в среднем на 20%, что экспериментально показано на тестовых программах из пакета STAMP.

2.3. Для известных алгоритмов автоматической векторизации циклов в открытых компиляторах GCC и LLVM/Clang выявлены классы трудно векторизуемых циклов из тестового набора ETSVC. Установлено, что на архитектуре Intel 64 известные алгоритмы способны векторизовать от 34% до 52% циклов пакета ETSVC. Построенное подмножество циклов составляет базисный набор для анализа эффективности ядер автовекторизаторов оптимизирующих компиляторов для векторных процессоров класса «регистр-регистр».

2.4. Создан инструментарий анализа эффективности использования микроархитектурных возможностей ядер суперскалярных процессоров ВС. В отличие от известных пакетов, предложенные программные средства позволяют анализировать загрузку суперскалярного конвейера архитектуры Intel 64 потоком инструкций с точностью до нескольких команд ассемблера.

3. Выполнено развитие программно-аппаратной конфигурации мультикластерной ВС. В состав системы введены сегменты на базе вычислительных узлов с сопроцессорами Intel Xeon Phi и графическими процессорами NVIDIA. Программный инструментарий системы расширен разработанными автором пакетами оптимизации использования многоуровневого параллелизма ВС в параллельных программах.

Рекомендации и перспективы дальнейшей разработки темы:

1. Разработка методов организации отказоустойчивого (живучего) мультипрограммного функционирования большемасштабных мультиархитектурных ВС при выполнении PGAS-программ.

2. Развитие оценочных моделей для расстановки приоритетов (полу)автоматического совместного использования различных форм параллелизма в программах: передача сообщений, многопоточность, векторизация кода.

Список сокращений и условных обозначений

АЛУ арифметико-логическое устройство.
ВС вычислительная система.
ОС операционная система.
ПТП программная транзакционная память.
ПЛИС программируемая логическая интегральная схема.
ПО программное обеспечение.
ФУ функциональное устройство.
ЭВМ электронная вычислительная машина.
ЭМ элементарная машина.
HTM Hardware Transactional Memory.
MPI Message Passing Interface.
PGAS Partitioned Global Address Space
STM Software Transactional Memory.
SIMD Single Instruction Multiple Data.

Список литературы

1. Agha, Gul. Actors: A Model of Concurrent Computation in Distributed Systems [Text] / Gul Agha. — Cambridge, MA, USA : MIT Press, 1986. — ISBN: 0-262-01092-5.

2. Aho, Alfred V. Principles of Compiler Design (Addison-Wesley Series in Computer Science and Information Processing) [Text] / Alfred V. Aho, Jeffrey D. Ullman. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1977. — ISBN: 0201000229.

3. Avetisyan, Arutyun. GCC instruction scheduler and software pipelining on the Itanium platform [Text] / Arutyun Avetisyan, Andrey Belevantsev, Dmitry Melnik. — 2017. — 11. — URL: http://rogue.colorado.edu/EPIC7/avetisyan.pdf.

4. Callahan, D. The Cascade High Productivity Language [Text] / D. Callahan, B.L. Chamberlain, H.P. Zima // International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004). — [S. l. : s. n.], 2004. — P. 52–60.

5. Chen, W. Communication Optimizations for Fine-grained UPC Applications [Text] / W. Chen, C. Iancu, K. Yelick // 14th International Conference on Parallel Architectures and Compilation Techniques (PACT). — [S. l. : s. n.], 2005. — P. 267–278.

6. Choi, Jong-Deok. Race Frontier: Reproducing Data Races in Parallel-program Debugging [Text] / Jong-Deok Choi, Sang Lyul Min // SIGPLAN Not. — 1991. — apr. — Vol. 26, no. 7. — P. 145–154.

7. Communication Optimizations for Distirbuted-Memory X10 Programs [Text] / R. Barik, J. Zhao, D. Grove [et al.] // IEEE International Parallel and Distributed Processing Symposium. — [S. l. : s. n.], 2011. — P. 1–13.

8. D., Callahan. Vectorizing Compilers: A Test Suite and Results [Text] / Callahan D., Dongarra J., Levine D. // Proc. of the ACM/IEEE Conf. on Supercomputing. — [S. l. : s. n.], 1988. — P. 98–105.

9. D., Hendler. A scalable lock-free stack algorithm [Text] / Hendler D., Shavit N., Yerushalmi L. // Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures. SPAA '04. — [S. l. : s. n.], 2004. — P. 206–215.

10. D., Levine. A Comparative Study of Automatic Vectorizing Compilers [Text] / Levine D., Callahan D., Dongarra J. // Journal of Parallel Computing. — Vol. 17. — [S. l. : s. n.], 1991. — P. 1223–1244.

11. Dice, Dave. Transactional locking II [Text] / Dave Dice, Ori Shalev, Nir Shavit // In DISC '06: Proc. 20th International Symposium on Distributed Computing, September 2006. — Vol. 4167. — [S. l. : s. n.], 2006. — P. 194–208.

12. Dijkstra, E. W. Solution of a Problem in Concurrent Programming Control [Text] / E. W. Dijkstra // Commun. ACM. — 1965. — sep. — Vol. 8, no. 9. — P. 569–.

13. E., Rohou. Vectorization Technology To Improve Interpreter Performance [Text] / Rohou E., Williams K., Yuste D. // ACM Trans. on Architecture and Code Optimization. — Vol. 9(4). — [S. l. : s. n.], 2013. — P. 26:1–26:22.

14. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs [Text] / Stefan Savage, Michael Burrows, Greg Nelson [et al.] // SIGOPS Oper. Syst. Rev. — 1997. — oct. — Vol. 31, no. 5. — P. 27–37.

15. An Evaluation of Global Address Space Languages: Co-array Fortran and Unified Parallel C [Text] / Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey [et al.] // Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — PPoPP '05. — New York, NY, USA : ACM, 2005. — P. 36–47.

16. An Evaluation of Vectorizing Compilers [Text] / Maleki S., Gao Ya., Garzaran M.J. [et al.] // Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques. — [S. l. : s. n.], 2011. — P. 372–382.

17. Extended Test Suite for Vectorizing Compilers [Electronic resource]. — [S. l. : s. n.]. — URL: http://polaris.cs.uiuc.edu/~maleki1/TSVC.tar.gz.

18. Feautrier, Paul. Bernstein's Conditions [Text] / Paul Feautrier // Encyclopedia of Parallel Computing / Ed. by David Padua. — Boston, MA : Springer US, 2011. — P. 130–134. — ISBN: 978-0-387-09766-4.

19. Felber, Pascal. Transactional locking II [Text] / Pascal Felber, Christof Fetzer, Torvald Riegel // Dynamic performance tuning of word-based software transactional memory. PPOPP 2008. — [S. l. : s. n.], 2008. — P. 237–246.

20. Fog, Agner. — Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs [Text], 2010. — http://www.agner.org/optimize/instruction_tables.pdf.

21. Fog, Agner. — Optimizing subroutines in assembly language: An optimization guide for x86 platforms [Text], 2010. — http://agner.org/optimize/optimizing_assembly.pdf.

22. Fog, Agner. — The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers [Text], 2010. — http://www.agner.org/optimize/microarchitecture.pdf.

23. Fortress (Sun HPCS Language) [Text] / Guy L. Steele, Eric Allen, David Chase [et al.] // Encyclopedia of Parallel Computing / Ed. by David Padua. — Boston, MA : Springer US, 2011. — P. 718–735.

24. Herlihy, M. The Art of Multiprocessor Programming [Text] / M. Herlihy, N. Shavit. // Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. — [S. l. : s. n.], 2008. — P. 508.

25. Hoefler, T. LogGP in Theory and Practice - An In-depth Analysis of Modern Interconnection Networks and Benchmarking Methods for Collective Operations [Text] / T. Hoefler, T. Schneider, A. Lumsdaine // Elsevier Journal of Simulation Modelling Practice and Theory. — 2009. — Vol. 17, no. 9. — P. 1511–1521.

26. An Integrated Hardware-software Approach to Flexible Transactional Memory [Text] / Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain [et al.] // SIGARCH Comput. Archit. News. — 2007. — jun. — Vol. 35, no. 2. — P. 104–115.

27. Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual [Text] / Intel Corporation. — [S. l. : s. n.], 2015. — September.

28. Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual [Text] / Intel Corporation. — [S. l. : s. n.], 2016. — January.

29. Intel Corporation. Intel Transactional Memory Compiler and Runtime Application Binary Interface. Revision: 1.0.1 [Electronic resource]. — [S. l. : s. n.], 2008. — URL: https://software.intel.com/sites/default/files/m/5/a/2/a/f/8097-Intel_TM_ABI_1_0_1.pdf.

30. Introduction to UPC and Language Specification [Text] : Rep. : CCS-TR-99-157 / George Mason University ; Executor: W. Carlson, J.M. Draper, D.E. Culler [et al.] : 1999. — MAY. — URL: http://upc.gwu.edu/.

31. Kennedy, K. Optimizing compilers for modern architectures: a dependence-based approach [Text] / K. Kennedy, R. Allen. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc, 2002.

32. Kennedy, Ken. The Rise and Fall of High Performance Fortran: An Historical Object Lesson [Text] / Ken Kennedy, Charles Koelbel, Hans Zima // Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages. — HOPL III. — New York, NY, USA : ACM, 2007. — P. 7–1–7–22.

33. Kielmann, Thilo. Fast Measurement of LogP Parameters for Message Passing Platforms [Text] / Thilo Kielmann, Henri E. Bal, Kees Verstoep // Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing. — London, UK, UK : Springer-Verlag, 2000. — P. 1176–1183.

34. Kulagin, I. Optimization of Conflict Detection in Parallel Programs with Transactional Memory [Text] / I. Kulagin, M. Kurnosov // Proc. of 10th Annual International Scientific Conference on Parallel Computing Technologies (PCT-2016). — Vol. 1576. — [S. l. : s. n.], 2016. — P. 582–594.

35. Kulagin, I. Heuristic Algorithms for Optimizing Communications in Parallel PGAS-programs [Text] / I. Kulagin, A. Paznikov, M. Kurnosov // Proc.

of the 13th International Conference on Parallel Computing Technologies (LNCS). — Vol. 9251. — [S. l. : s. n.], 2015. — P. 405–409.

36. LogGP: Incorporating Long Messages into the LogP Model — One Step Closer Towards a Realistic Model for Parallel Computation [Text] / Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, Chris Scheiman // Technical report, University of California at Santa Barbara. — 1995.

37. LogTM: Log-based transactional memory [Text] / Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan [et al.] // In HPCA '06: Proc. 12th International Symposium on High-Performance Computer Architecture, February 2006. — [S. l. : s. n.], 2006. — P. 254–265.

38. Luchango, Victor. Transactional memory for C++ [Electronic resource]. — [S. l. : s. n.]. — URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3718.pdf.

39. M., Olszewski. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory [Text] / Olszewski M., Cutler J., Steffan J. G. // Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques. PACT '07. — [S. l. : s. n.], 2007. — P. 365–375.

40. Makarov, Vladimir N. The finite state automaton based pipeline hazard recognizer and instruction scheduler [Text] / Vladimir N. Makarov, Red Hat // in GCC. Proc. of GCC Summit. — [S. l. : s. n.], 2003.

41. Moon, Soo-Mook. Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining [Text] / Soo-Mook Moon, Kemal Ebcioğlu // ACM Trans. Program. Lang. Syst. — 1997. — nov. — Vol. 19, no. 6. — P. 853–898.

42. Muchnick, Steven S. Advanced Compiler Design and Implementation [Text] / Steven S. Muchnick. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1997. — ISBN: 1-55860-320-4.

43. Numrich, Robert W. Co-array Fortran for Parallel Programming [Text] / Robert W. Numrich, John Reid // SIGPLAN Fortran Forum. — 1998. — aug. — Vol. 17, no. 2. — P. 1–31.

44. P., Konsor. Avoiding AVX-SSE Transition Penalties [Electronic resource]. — [S. l. : s. n.]. — URL: https://software.intel.com/en-us/articles/avoiding-avx-sse-transition-penalties.

45. Patterson, David A. Computer Organization and Design: The Hardware/-Software Interface [Text] / David A. Patterson, John L. Hennessy. — 3rd edition. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2007. — ISBN: 0123706068, 9780123706065.

46. R.C., Metzger. Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization [Text] / Metzger R.C., Wen Zh. — [S. l. : s. n.], 2000.

47. Riegel, Torvald. A Lazy Snapshot Algorithm with Eager Validation [Text] / Torvald Riegel, Pascal Felber, Christof Fetzer // 20th International Symposium on Distributed Computing (DISC). — [S. l. : s. n.], 2006.

48. Rochester Software Transactional Memory Runtime. Project web site [Electronic resource]. — [S. l. : s. n.]. — URL: www.cs.rochester.edu/research/synchronization/rstm/.

49. Roger W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2 [Text] / Roger W. Hockney // Parallel Computing. — 1994. — Vol. 20, no. 3. — P. 389–398.

50. STAMP: Stanford Transactional Applications for Multi-Processing [Text] / Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, K. Olukotun // 2008 IEEE International Symposium on Workload Characterization. — [S. l. : s. n.], 2008. — Sept. — P. 35–46.

51. Schreiber, Robert. An Introduction to HPF [Text] / Robert Schreiber // The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications. — [S. l.] : Springer-Verlag, 1996. — P. 27–44.

52. Shavit, N. Software Transactional Memory [Text] / N. Shavit, D. Touitou. // In PODC'95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, New York, NY, USA, Aug. 1995. — [S. l. : s. n.], 1995. — P. 204–213.

53. Software Pipelining [Text] / Vicki H. Allan, Reese B. Jones, Randall M. Lee, Stephen J. Allan // ACM Comput. Surv. — 1995. — sep. — Vol. 27, no. 3. — P. 367–432.

54. Spear, Michael F. RingSTM: scalable transactions with a single atomic instruction [Text] / Michael F. Spear, Maged M. Michael, Christoph von Praun // In SPAA '08: Proc. 20th Annual Symposium on Parallelism in Algorithms and Architectures, June 2008. — [S. l. : s. n.], 2008. — P. 275–284.

55. Time-based Software Transactional Memory [Text] / Pascal Felber, Christof Fetzer, Patrick Marlier, Torvald Riegel // IEEE Transactions on Parallel and Distributed Systems. — Vol. 21(12). — [S. l. : s. n.], 2010. — P. 1793–1807.

56. Tomasulo, R. M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units [Text] / R. M. Tomasulo // IBM J. Res. Dev. — 1967. — jan. — Vol. 11, no. 1. — P. 25–33.

57. Vector Parallelism in JavaScript: Language and Compiler Support for SIMD [Text] / Jibaja I., Jensen P., Hu N. [et al.] // Proc. of the Int. Conf. on Parallel Architecture and Compilation Techniques. — [S. l. : s. n.], 2015. — P. 407–418.

58. Vijay, Saraswat. X10: Concurrent Programming for Modern Architectures [Text] / Saraswat Vijay // Proceedings of the 5th Asian Conference on Programming Languages and Systems. — APLAS'07. — Berlin, Heidelberg : Springer-Verlag, 2007. — P. 1–1.

59. X10: An Object-oriented approach to non-uniform Clustered Computing [Text] / P. Charles, C. Donawa, K. Ebcioglu [et al.] // 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2005). — [S. l. : s. n.], 2005. — P. 519–538.

60. Zilles, Craig. Implications of false conflict rate trends for robust software transactional memory [Text] / Craig Zilles, Ravi Rajwar // In IISWC '07: Proc. 2007 IEEE. — [S. l. : s. n.], 2007. — P. 237–246.

61. A comprehensive strategy for contention management in software transactional memory [Text] / Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, Michael L. Scott. // In PPoPP '09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2009. — [S. l. : s. n.], 2009. — P. 141–150.

62. Бурцев, В.С. Параллелизм вычислительных процессов и развитие архитектур суперЭВМ [Текст] / В.С. Бурцев. — М. : ИВВС РАН, 1997.

63. Векторизация программ: теория, методы, реализация [Текст] // Сб. статей: Пер. с англ. и нем. М.: Мир. — [Б. м. : б. и.], 1991. — С. 275.

64. Воеводин, В.В. Параллельные вычисления [Текст] / В.В. Воеводин, Вл.В. Воеводин. — СПб. : БХВ-Петербург, 2002.

65. Дмитриев, Ю. К. Вычислительные системы из мини-ЭВМ [Текст] / Ю. К. Дмитриев, В. Г. Хорошевский. — М. : Радио и связь, 1982.

66. Евреинов, Э.В. Однородные вычислительные системы [Текст] / Э.В. Евреинов, В.Г. Хорошевский. — Новосибирск. : Наука, 1978.

67. Евреинов, Э. В. О возможности построения вычислительных систем высокой производительности [Текст] / Э. В. Евреинов, Ю. Г. Косарев. — Новосибирск : СО АН СССР, 1962.

68. Евреинов, Э. В. Однородные универсальные вычислительные системы высокой производительности [Текст] / Э. В. Евреинов, Ю. Г. Косарев. — Новосибирск. : Наука, 1966.

69. И.И., Кулагин. Эвристические алгоритмы оптимизации информационных обменов в параллельных PGAS-программах [Текст] / Кулагин И.И., Пазников А.А., Курносов М.Г. // Вестник СибГУТИ. — 2014. — № 3. — С. 52–66.

70. И.И., Кулагин. Инструментация и оптимизация выполнения транзакционных секций многопоточных программ [Текст] / Кулагин И.И., Курносов М.Г. // Труды Института системного программирования РАН. — 2015. — Т. 27, № 6. — С. 135–119.

71. И.И., Кулагин. О спекулятивном выполнении критических секций на вычислительных системах с общей памятью [Текст] / Кулагин И.И., Курносов М.Г. // Известия южного федерального университета. Технические науки. — 2016. — № 11. — С. 54–64.

72. И.И., Кулагин. Оптимизация обнаружения конфликтов в параллельных программах с транзакционной памятью [Текст] / Кулагин И.И., Курносов М.Г. // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. — 2016. — Т. 5, № 4. — С. 46–60.

73. "Исходный код разработанных тестов DGEMM"[Электронный ресурс]. — [Б. м. : б. и.]. — URL: https://github.com/Ikulagin/vect_bench.

74. Каляев, И.А. Реконфигурируемые мультиконвейерные вычислительные структуры [Текст] / И.А. Каляев. — Ростов-на-Дону : ЮНЦ РАН, 2003.

75. Каляев, А.В. Модульно-наращиваемые многопроцессорные системы со структурно-процедурной организацией вычислений [Текст] / А.В. Каляев, И.И Левин. — М. : Янус-К, 2003.

76. Ким, А. К. Микропроцессоры и вычислительные комплексы семейства «Эльбрус» [Текст] / А. К. Ким, В. И. Перекатов, С. Г Ермаков. — [Б. м.] : СПб.: Питер, 2013. — URL: http://www.mcst.ru/files/511cea/886487/1a8f40/000000/book_elbrus.pdf.

77. Корнеев, В.В. Архитектура вычислительных систем с программируемой структурой [Текст] / В.В. Корнеев. — Новосибирск. : Наука, 1985.

78. Кулагин, И.И. Повышение эффективности циклического доступа к удаленным массивам в программах на языке IBM X10 [Текст] / И.И. Кулагин // Сборник статей Всероссийской научно-практической конференции: Многоядерные процессоры, параллельное программирование, ПЛИС, системы обработки сигналов. – Изд-во Алт. ун-та. — [Б. м. : б. и.], 2015. — С. 129–136.

79. Кулагин, И.И. Подход к архитектурно-ориентированной оптимизации программ для архитектуры Intel 64 [Текст] / И.И. Кулагин // Материалы Российской научно-технической конференции «Обработка

информации и математическое моделирование» – Новосибирск: СибГУТИ. — [Б. м. : б. и.], 2017. — С. 300–310.

80. Кулагин, И.И. Исследование эффективности доступа к распределенным массивам в программах на языке IBM X10 [Текст] / И.И. Кулагин, М.Г. Курносов // Материалы Российской научно-технической конференции «Обработка информационных сигналов и математическое моделирование». – Новосибирск. — [Б. м. : б. и.], 2014. — С. 77–79.

81. Кулагин, И.И. Методы оптимизации передачи массивов в параллельных программах на языке IBM X10 [Текст] / И.И. Кулагин, М.Г. Курносов // Материалы докладов 10-ой Российской конференции с международным участием «Новые информационные технологии в исследование сложных структур» (ICAM-2014). – Томск: НТЛ. — [Б. м. : б. и.], 2014. — С. 5–6.

82. Кулагин, И.И. Оптимизация доступа к удаленным массивам в программах на языке IBM X10 [Текст] / И.И. Кулагин, М.Г. Курносов // Материалы 52-й Международной научной студенческой конференции МНСК-2014: Информационные технологии / Новосиб. гос. ун-т. – Новосибирск. — [Б. м. : б. и.], 2014. — С. 168.

83. Кулагин, И.И. Оптимизация информационных обменов в параллельных PGAS-программах [Текст] / И.И. Кулагин, М.Г. Курносов // Материалы 3-й Всероссийской научно-технической конференции «Суперкомпьютерные технологии» (СКТ-2014). — [Б. м. : б. и.], 2014. — С. 158–162.

84. Кулагин, И.И. Алгоритмы оптимизации ложных конфликтов в параллельных программах на базе транзакционной памяти [Текст] / И.И. Кулагин, М.Г. Курносов // Материалы международной конференции «Актуальные проблемы вычислительной и прикладной математики» (АПВПМ-2015, АМСА-2015), Новосибирск. — [Б. м. : б. и.], 2015. — С. 416–422.

85. Кулагин, И.И. Анализ обнаружения ложных конфликтов в приложениях с программной транзакционной памятью [Текст] / И.И. Кулагин, М.Г. Курносов // Материалы Российской научно-технической конференции «Обработка информации и математическое моделирование», 2015. – Новосибирск: СибГУТИ. — [Б. м. : б. и.], 2015. — С. 335–337.

86. Кулагин, И.И. Подход к сокращению ложных конфликтов в параллельных программах на базе транзакционной памяти [Текст] / И.И. Кулагин, М.Г. Курносов // Тезисы докладов Сибирской конференции по параллельным и высокопроизводительным вычислениям (СКПВВ-2015), Томск. — [Б. м. : б. и.], 2015. — С. 48.

87. Кулагин, И.И. О подходах к реализации программной транзакционной памяти [Текст] / И.И. Кулагин, М.Г. Курносов // Материалы Российской научно-технической конференции «Обработка информации и математическое моделирование», 2016. – Новосибирск: СибГУТИ. — [Б. м. : б. и.], 2016. — С. 331–338.

88. Кулагин, И.И. О спекулятивном выполнении критических секций на вычислительных системах с общей памятью [Текст] / И.И. Кулагин, М.Г. Курносов // Материалы Всероссийской научно-технической конференции «Суперкомпьютерные технологии» (СКТ-2016). — [Б. м. : б. и.], 2016. — С. 170–174.

89. Кулагин, И.И. Оптимизация обнаружения конфликтов в параллельных программах с транзакционной памятью [Текст] / И.И. Кулагин, М.Г. Курносов // Параллельные вычислительные технологии (ПаВТ'2016): труды международной научной конференции (28 марта – 1 апреля 2016 г., г. Архангельск). Челябинск: Издательский центр ЮУрГУ. — [Б. м. : б. и.], 2016. — С. 582–594.

90. Курносов, М.Г. Оптимизация выполнения MPI-программ по результатам их профилирования [Текст] / М.Г. Курносов, И.И. Кулагин // Материалы Российской научно-технической конференции «Обработка информационных сигналов и математическое моделирование». Новосибирск. — [Б. м. : б. и.], 2012. — С. 159–160.

91. Левин, В.К. Высокопроизводительные мультипроцессорные системы [Текст] / В.К. Левин // Информационные технологии и вычислительные системы. — 1995. — № 1. — С. 12–21.

92. Малышкин, В.Э. Параллельное программирование мультикомпьютеров [Текст] / В.Э. Малышкин, В.Д. Корнеев. — Новосибирск : НГТУ, 2006.

93. Микропроцессорные системы : Учебное пособие для вузов [Текст] / Александров Е.К., Грушвицкий Р.И., Куприянов М.С. [и др.]. — [Б. м.] : Под общ. ред. Д. В. Пузанкова. – СПб. : Политехника, 2012. — URL: http://www.studentlibrary.ru/book/ISBN5732505164.html.

94. Миренков, Н.Н. Параллельное программирование для многомодульных вычислительных систем [Текст] / Н.Н. Миренков. — М. : Радио и связь, 1989.

95. Молдованова, О.В. Векторизация циклов в открытых компиляторах для архитектур с короткими векторными регистрами [Текст] / О.В. Молдованова, И.И. Кулагин, М.Г. Курносов // Труды 13-й Международной школы-семинара "Проблемы оптимизации сложных систем"в рамках международной конференции IEEE SIBIRCON 2017, Новосибирск, 18-22 сент. 2017 г. — [Б. м. : б. и.], 2017. — С. 70–78. — URL: http://conf.nsc.ru/opcs2017/ru/proceedings.

96. Панфилов, И.В. Вычислительные системы [Текст] / И.В. Панфилов, А.М. Половко. — М. : Советское радио, 1980.

97. Поспелов, Д.А. Введение в теорию вычислительных систем [Текст] / Д.А. Поспелов. — М. : Советское радио, 1972.

98. Прангишвили, И.В. Параллельные вычислительные системы с общим управлением [Текст] / И.В. Прангишвили, С.Я. Виленкин, И.Л. Медведев. — М. : Энергопромиздат, 1983.

99. С.Д., Кузнецов. Транзакционная память [Электронный ресурс]. — [Б. м. : б. и.], 2004. — URL: www.citforum.ru/programming/digest/transactional_memory/.

100. Свид. 2015619554 Российская Федерация. Свидетельство о государственной регистрации программы для ЭВМ. Программа компиляторной оптимизации доступа к удаленным массивам в программах на языке IBM X10 / Кулагин И.И., Курносов М.Г. Заявитель и патентообладатель ФГОБУ ВПО «СибГУТИ». Заявл. 14.07.2015, опубл. 08.09.2015. [Электронный ресурс]. — [Б. м. : б. и.].

101. Свидетельство 2016660098 РФ. Программа оптимизации выполнения транзакционных секций в параллельных программах для вычислительных систем с общей памятью : свидетельство об официальной регистрации программы для ЭВМ / И.И. Кулагин, М.Г. Курносов; заявитель и патентообладатель СибГУТИ; заявл. 25.07.2016, опубл. 06.09.2016. [Электронный ресурс]. — [Б. м. : б. и.].

102. Свидетельство 2017660065 РФ. Программа детального анализа производительности выполнения кода на архитектуре Intel 64 : свидетельство об официальной регистрации программы для ЭВМ / И.И. Кулагин, М.Г. Курносов; заявитель и патентообладатель СибГУТИ; заявл. 17.07.2017, опубл. 14.09.2017 [Электронный ресурс]. — [Б. м. : б. и.].

103. Топорков, В.В. Модели распределенных вычислений [Текст] / В.В. Топорков. — М. : ФИЗМАТЛИТ, 2004.

104. Хорошевский, В.Г. Архитектура вычислительных систем [Текст] / В.Г. Хорошевский. — М. : МГТУ им. Н. Э. Баумана, 2008.

105. Хорошевский, В.Г. Распределенные вычислительные системы с программируемой структурой [Текст] / В.Г. Хорошевский // Вестник СибГУТИ. — 2010. — № 2. — С. 3–41.

106. Языки и параллельные ЭВМ : сб. ст. [Текст] / Под ред. А. А. Самарский. — М. : Наука, 1990.

107. Яненко, Н. Н. Параллельные вычисления в задачах математической физики на вычислительных системах с программируемой структурой [Текст] / Н. Н. Яненко, В.Г. Хорошевский, А.Д. Рычков // Электронное моделирование. — 1984. — Т. 6, № 1. — С. 3–8.

Список иллюстраций

Список таблиц

Приложения

Приложение А

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2015619554

Программа компиляторной оптимизации доступа к удаленным массивам в программах на языке IBM X10

Правообладатель: ***Федеральное государственное образовательное бюджетное учреждение высшего профессионального образования «Сибирский государственный университет телекоммуникаций и информатики» (RU)***

Авторы: ***Кулагин Иван Иванович (RU),***
Курносов Михаил Георгиевич (RU)

Заявка № **2015616424**
Дата поступления **14 июля 2015 г.**
Дата государственной регистрации
в Реестре программ для ЭВМ ***08 сентября 2015 г.***

Заместитель руководителя Федеральной службы по интеллектуальной собственности

Л.Л. Кирий

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2016660098

Программа оптимизации выполнения транзакционных секций в параллельных программах для вычислительных систем с общей памятью

Правообладатель: ***Федеральное государственное бюджетное образовательное учреждение высшего образования «Сибирский государственный университет телекоммуникаций и информатики» (RU)***

Авторы: ***Кулагин Иван Иванович (RU),***
Курносов Михаил Георгиевич (RU)

Заявка № **2016618071**
Дата поступления **25 июля 2016 г.**
Дата государственной регистрации
в Реестре программ для ЭВМ ***06 сентября 2016 г.***

Руководитель Федеральной службы
по интеллектуальной собственности

Г.П. Ивлиев

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2017660065

«Программа детального анализа производительности выполнения кода на архитектуре Intel 64»

Правообладатель: ***Федеральное государственное бюджетное образовательное учреждение высшего образования «Сибирский государственный университет телекоммуникаций и информатики» (RU)***

Авторы: ***Кулагин Иван Иванович (RU),***
Курносов Михаил Георгиевич (RU)

Заявка № **2017617478**
Дата поступления **17 июля 2017 г.**
Дата государственной регистрации
в Реестре программ для ЭВМ ***14 сентября 2017 г.***

Руководитель Федеральной службы
по интеллектуальной собственности

Г.П. Ивлиев

Приложение Б

СПРАВКА
об использовании результатов диссертационной работы
Кулагина Ивана Ивановича «Средства архитектурно-ориентированной оптимизации выполнения параллельных программ для вычислительных систем с многоуровневым параллелизмом», представленной на соискание ученой степени кандидата технических наук по специальности 05.13.15 – «Вычислительные машины, комплексы и компьютерные сети»

Справка дана Кулагину И.И. в том, что результаты его диссертационной работы «Средства архитектурно-ориентированной оптимизации выполнения параллельных программ для вычислительных систем с многоуровневым параллелизмом», представленной на соискание ученой степени кандидата технических наук по специальности 05.13.15 – «Вычислительные машины, комплексы и компьютерные сети», использованы при выполнении научно-исследовательских работ на базе федерального государственного бюджетного образовательного учреждения высшего образования «Сибирский государственный университет телекоммуникаций и информатики» (СибГУТИ):

- проект РФФИ 15-07-00653 «Разработка моделей, методов и программного обеспечения оптимизации выполнения параллельных программ на языках семейства PGAS» (2015-2017 гг., руководитель проекта – д.т.н., доцент Курносов М.Г.);
- проект РФФИ 15-37-20113 «Модели, алгоритмы и программное обеспечение оптимизации функционирования иерархических мультиархитектурных вычислительных систем» (2015-2016 гг., руководитель проекта – д.т.н., доцент Курносов М.Г.);

а именно:

- методы оптимизации циклического доступа к распределенным массивам в параллельных PGAS-программах;
- методы и алгоритмы сокращения числа ложных конфликтов в реализациях программной транзакционной памяти на мультиархитектурных вычислительных системах.

Проректор
по научной работе СибГУТИ — Е.Р. Трубехин

УТВЕРЖДАЮ
Ректор СибГУТИ
_____ В.Г. Беленький
«24» 10 2017 г.

АКТ
о внедрении результатов диссертационной работы Кулагина Ивана Ивановича «Средства архитектурно-ориентированной оптимизации выполнения параллельных программ для вычислительных систем с многоуровневым параллелизмом», представленной на соискание ученой степени кандидата технических наук по специальности 05.13.15 – «Вычислительные машины, комплексы и компьютерные сети», в учебный процесс федерального государственного бюджетного образовательного учреждения высшего образования «Сибирский государственный университет телекоммуникаций и информатики» (СибГУТИ)

Мы, нижеподписавшиеся, заместитель заведующего Кафедрой вычислительных систем к.т.н. доцент О.В. Молдованова, заместитель декана Факультета информатики и вычислительной техники к.т.н. доцент В.И. Агульник, составили настоящий акт о том, что результаты диссертационной работы И.И. Кулагина «Средства архитектурно-ориентированной оптимизации выполнения параллельных программ для вычислительных систем с многоуровневым параллелизмом» используются в учебном процессе СибГУТИ, а именно: теоретические результаты по оптимизации циклического доступа к распределенным массивам, а также методы сокращения числа ложных конфликтов в реализациях программной транзакционной памяти используются в курсе лекций по дисциплине «Теория функционирования распределенных вычислительных систем» (уровень бакалавриата), результаты по анализу методов автоматической векторизации циклов используются в курсе «Параллельное программирование» (уровень магистратуры).

Заместитель заведующего
Кафедрой вычислительных систем
к.т.н. доцент О.В. Молдованова

Заместитель декана
Факультета информатики и вычислительной техники
доцент В.И. Агульник