Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

Л.В. Городняя

ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

Часть 4

Параллельное программирование

Препринт
175

Новосибирск 2015

Препринт является четвёртой частью серии «Парадигмы программирования», посвященной исследованию парадигм программирования. Представлены результаты анализа особенностей языков параллельного программирования. Содержание препринта представляет интерес для системных программистов, студентов и аспирантов, специализирующихся в области системного и теоретического программирования, и для всех тех, кто интересуется проблемами современной информатики, программирования и информационных технологий.

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

L.V. Gorodnyaya

PROGRAMING PARADIGMS

Part 4
Parallel Programming

Preprint
175

Novosibirsk 2015

The work describes research and specification of basic paradigms of programming. The author analyzes and compares special features of parallel programming languages. A functional model of comparative description of implementation semantics of basic paradigms is proposed. The author proposes a scheme of describing and defining paradigm features of programming languages.

ВВЕДЕНИЕ

Прогресс производства аппаратуры намного опередил развитие технологий программирования. Параллельные архитектуры теперь позволяют принципиально повышать производительность программ решения многих задач [3, 6, 9]. Методы автоматического распараллеливания программ способны обеспечить значительное ускорение вычислений, явно сводимых к комплексу независимых процессов обработки элементов векторов, но отступают перед программами более широкого класса. Перспектива выражения языковыми средствами параллелизма на уровне постановки новой задачи приторможена проблемами оптимизирующей верифицирующей компиляции, обеспечивающей аккуратный выбор эффективной и надежной схемы параллелизма. Кроме того, трудоёмкость повторного программирования и отладки параллельных алгоритмов обуславливает рассмотрение готовых последовательных программ в качестве эталонов при оценке результатов параллельных вычислений, что сдерживает развитие языков параллельного программирования и методов их реализации.

В настоящее время осознана актуальность формирования парадигмы параллельного программирования, вызванная расширением и развитием системы базовых понятий, необходимых для рациональной разработки систем управления процессами на современной аппаратуре [2, 8]. Парадигмы в программировании характеризуются стилем мышления при решении задачи, системой используемых понятий и особенностями их реализации. Можно констатировать, что стиль мышления и система понятий для параллельного программирования уже сложились в процессе эволюции языков программирования, но типовая их поддержка в системах программирования ещё не сформировалась [35]. По-видимому, кроме верифицирующей компиляции, это направление потребует создания трансформационной технологии конструирования параллельных программ из имеющихся, уже отлаженных, обычных однопроцессорных программ.

«Ничем не скроешь фундаментальную трудность параллелизма» – так прозвучало заявление Мартина Одерски (Martin Odersky) в его приглашенном докладе на 20-й Международной конференции «Конструирование компиляторов», состоявшейся в марте 2011 года в Германии под председательством Дженса Кнопа (Jens Knoop) [40]. Несмотря на многочисленные призывы и конкурсы, пока не появилось общих идей по новому поколению языков и систем программирования (ЯСП), в которых параллелизм имел

бы самостоятельное значение, а не рассматривался бы (по понятным причинам) как пристройка к традиционному программированию.

14. ПРОСТРАНСТВО РЕШЕНИЙ

Рост интереса к параллельному программированию в наши дни связан с переходом к массовому производству многоядерных архитектур. Ознакомление с понятиями и явлениями параллельного программирования в данном материале нацелено на профилактику прочного привыкания к принципам традиционного последовательного программирования. Такая профилактика необходима для специализации в области разработки распределенных информационных систем, а также приложений для суперкомпьютеров, многопроцессорных конфигураций и графических процессоров.

Современное развитие ЯСП на практике ориентировано на решение задач параллельного программирования. Как правило, новые ЯСП включают в себя библиотечные модули, обеспечивающие организацию процессов, или подъязыки, допускающие многопоточное программирование. Это, увы, не исключает реальную практику ручного распараллеливания ранее отлаженных обычных программ, приведения их к виду, удобному для применения производственных систем поддержки параллельных вычислений. Значительная часть таких работ носит технический характер и заключается в систематической реорганизации структур данных, изменении статуса переменных и включении в программу аннотаций, сообщающих компилятору об информационно-логических взаимосвязях. Существенным ограничением результата ручного распараллеливания является не только опасность повторной отладки алгоритма, но и его жесткая зависимость от характеристик целевой архитектуры.

Разнообразие моделей параллельных вычислений и расширение спектра доступной архитектуры следует рассматривать как вызов разработчикам ЯСП, способных решать проблему создания методов компиляции многопоточных программ для многопроцессорных конфигураций. Язык должен допускать представление любых моделей параллелизма, проявляемого на уровне решаемой задачи или реализуемого с помощью реальной архитектуры. Причем такое представление требует лаконичных форм и конструктивных построений, гарантирующих сохранение свойств программ при их реорганизации. Не менее важна расширяемость языка по мере развития средств и методов параллельных вычислений, темп которого превышает скорость осознания специалистами их возможностей.

Рассматривая задачу формализации языков параллельного программирования как путь к решению проблемы адаптации программ к различным особенностям используемых многопроцессорных комплексов и многоядерных процессоров, приходится видеть, что решение этой проблемы требует разработки новых методов компиляции программ, особенно масштабируемой кодогенерации, и развития средств и методов ясного описания семантики языков параллельного программирования.

14.1. Параллельные алгоритмы

Обзор решений, встречающихся при организации параллельных процессов для реализации параллельных алгоритмов, даёт некоторое представление о разнообразии многопроцессорных архитектур и систем:

1. Процессоров может быть много, и они могут обладать разными системами команд.
2. Процессоры могут работать автономно или сообща выполнять общую работу.
3. Отдельные процессоры могут функционировать в холостую, не выполняя полезной работы.
4. Процессоры имеют свою локальную память и могут работать в общей памяти.
5. Распределение действий по процессорам может быть предписано программой, а может осуществляться супервизором, поддерживающим определенную модель вычислений.

Такое пространство допускает естественное развитие в направлении моделирования асинхронных процессов, позволяет уяснить, что существуют контексты исполнения программ, от которых зависит выполнимость команд. При анализе результатов программ обнаруживаются разные категории ошибок управления и выделяются действия, невыполнимые, несмотря на то, что все образующие их команды выполнимы. Проявляется роль частичного упорядочения действий в обеспечении работоспособности программ средствами управления вычислениями. Обнаруживается неимперативность последовательного исполнения действий при организации параллельных процессов.

Переход к параллельным алгоритмам влечёт пересмотр содержания многих понятий и введение новых терминов, отражающих разного рода явления и эффекты, не имевшие особого значения для обычных последова-

тельных алгоритмов. Прошли времена, когда параллельное программирование называли не алгоритмическим. Теперь термин «параллельный алгоритм» связан с пониманием, что алгоритм может выполняться как одним, так и группой исполнителей. Такое понимание резко расширяет пространство решений задач, меняет подходы к реализации решений, использующих параллелизм, и в некоторой мере сказывается собственно на постановке задач и планировании жизненного цикла программ решения задач, ориентированных на использование параллелизма.

Прежде всего, следует прояснить следующие вопросы, связанные с формулировкой постановки задачи:

I. Насколько изменится трудоёмкость жизненного цикла программы решения задачи с помощью параллельного алгоритма?
II. В какой мере при постановке задачи следует учитывать модель параллелизма?
III. Как обосновать и измерить выигрыш от разработки параллельного алгоритма?
IV. Насколько изменяется постановка задачи при переходе к параллельным алгоритмам?
V. Что даёт парадигма параллельного программирования на уровне разработки параллельного алгоритма?
VI. Какими средствами представляются разрабатываемые параллельные алгоритмы решения задачи на этапе, предшествующем разработке программы?

Ответы на вопрос I

- Длительность жизненного цикла программы зависит от уровня изученности пространства допустимых решений задачи. Поэтому возникает дополнительная работа – изучение спектра возможных алгоритмических решений, не замеченных при последовательном программировании.
- При наличии разработанного последовательного алгоритма, от отлаженной программной реализации которого не стоит отказываться, возникает дополнительная работа по разбиению представленных временных отношений на принципиальные и несущественные и установлению функциональной эквивалентности результата.
- Темп обновления аппаратуры обуславливает для практически значимых задач высокую вероятность дополнительной работы по

адаптации алгоритма к новым техническим решениям, предвидеть конкретику которых не всегда удаётся.

Ответы на вопрос II

- Выбор модели параллелизма на уровне постановки задачи дает возможность снизить трудозатраты на реализацию первой версии долгоживущей программы.
- Соответствие модели параллелизма возможностям доступной многопроцессорной архитектуры может утратить актуальность, что приводит к необходимости повторного пересмотра постановки задачи, разработки алгоритма и программирования.
- Возможно определение достаточно общей модели параллелизма, допускающей пошаговую конкретизацию, отражающую специфику доступной аппаратуры.

Ответы на вопрос III

- Существуют средства измерения пространственно-временных характеристик производительности программ.
- Возможен аналитический расчет или логический вывод других эксплуатационных характеристик программы, таких как надежность, безопасность, справедливость и т.п.
- Достигли производственного уровня эксперименты по верификации программ на моделях, способные показывать уязвимые места в готовых программах и спецификациях.

Ответы на вопрос IV

- Можно, не изменяя формулировку задачи вида «По заданным данным и известной, алгоритмически вычислимой, функции получить её результат», изменить трактовку, подразумевая «По заданным данным и известной, вычисляемой с помощью параллельного алгоритма, функции получить её результат».
- Возможно упоминание в постановке задачи типа целевой многопроцессорной архитектуры.
- Более естественно в постановку задачи или методы решения включить её декомпозицию на независимые подзадачи, часть которых определяют временные отношения и взаимодействия между другими подзадачами, что позволяет стратифицировать проблемы управления вычислениями и собственно вычислений.

Ответы на вопрос V

- Парадигма параллельного программирования предлагает разработчику параллельного алгоритма стиль мышления и подход к декомпозиции постановки задачи на фрагменты, приспособленные к параллельному программированию на доступных многопроцессорных архитектурах.
- Языки и системы параллельного программирования предоставляют средства и методы, поддерживающие эффективную организацию параллельных вычислений на доступном оборудовании, включая и однопроцессорные конфигурации.
- Мультипарадигматические языки программирования поддерживают доступ к общим библиотекам параллельного программирования, что снимает зависимость реализации параллельных алгоритмов от первичного выбора изобразительных средств.

Ответы на вопрос VI

- Существует представительное множество языков спецификации и верификации моделей, приспособленных представлять параллельные алгоритмы и проверять их свойства.
- Декларативные языки программирования и языки сверх высокого уровня позволяют представлять и отлаживать прототипы программ параллельных вычислений.
- Возможен путь формирования технологии, подобной решениям в универсальном языке моделирования UML, – поддержки расширяемого ряда канонических диаграмм, компактно представляющих необходимый уровень детализации параллельного алгоритма.

За полвека традиционного последовательного программирования отлажено колоссальное количество программ, аккумулированных в системы программирования и стандартные библиотеки. Изменение постановок задач, уже имеющих готовые отлаженные программные решения, ради учёта допустимого параллелизма чревато повторным программированием и, что гораздо более трудоёмко, повторной отладкой.

Основной аргумент – целесообразность учета естественного параллелизма на уровне постановки задачи, утрачиваемого при решении задачи посредством обычных алгоритмов. Число языков параллельного программирования, удобных для реализации параллельных алгоритмов год от года

растёт, хотя и их применение решает не все проблемы организации параллельных вычислений.

Итак, параллельный алгоритм может быть реализован по частям на множестве различных устройств с последующим объединением полученных результатов и получением целевого результата. Возникают чисто практические вопросы:

A. Каким образом в определении алгоритма выделены части, выполняемые отдельными устройствами?
B. Обязана ли реализация алгоритма использовать в точности представленный в его определении набор устройств?
C. Можно ли последовательный алгоритм рассматривать как параллельный, исполняемый на одном устройстве?

Ответы на вопрос A

- Встречаются разные средства явной разметки и автоматического распределения действий алгоритма по устройствам.
- Алгоритм может быть построен как структура из отдельных алгоритмов, независимо выполняемых каждый на своем устройстве или семействе устройств.
- Выполняемые на разных устройствах алгоритмы могут взаимодействовать, и на них могут быть наложены временные отношения и связи по обмену данными.

Ответы на вопрос B

- Реализация алгоритма может быть выполнена как на меньшем, так и на большем числе устройств.
- Устройства могут быть равноправными и обладать одинаковыми возможностями.
- Среди устройств может быть специализация, определяющая выбор выполняемых частей алгоритма независимо от определённой алгоритмом разметки.

Ответы на вопрос C

- Изменяется понимание собственно последовательности действий. Если для последовательных алгоритмов можно было интуитивно полагать, что очередное действие выполняется немедленно вслед за предыдущим, то для параллельного алгоритма в случае его выполнения на меньшем числе устройств следует допускать

возможность, что между соседними действиями может быть исполнено действие из смежной части.

- Последовательный алгоритм можно распараллеливать, т. е. разбивать на автономные части, чтобы задействовать доступную группу процессоров.
- Параллельный алгоритм обычно можно реализовать на одном процессоре, причем существует целое семейство допустимых реализаций.

Следующая обойма вопросов касается категории «время» и связана с проблемами синхронизации:

D. Могут ли части параллельного алгоритма обладать своим независимым или централизованным отсчетом времени?
E. Может ли синхронизация частей алгоритма противоречить его информационным связям и логике управления?
F. Можно ли синхронизацию частей алгоритма рассматривать как частный случай асинхронности?

Ответы на вопрос D

- Части параллельного алгоритма могут обладать как своим собственным, так и общим централизованным временем.
- Часть параллельного алгоритма может согласовывать свое время со временем других частей.
- При согласовании времени возможны вставки специальных фиктивных действий, называемых паузами.

Ответы на вопрос E

- Синхронизация частей алгоритма может противоречить его информационным связям и логике управления, что рассматривается как ошибка исполнения.
- Исключение синхронизационных противоречий отчасти обеспечивается неимперативным стилем реализации алгоритма, допускающим преобразования программ.
- Информационные связи и логику управления можно понимать как условия готовности действий, независящие от синхронизационных отношений.

Ответы на вопрос F

- При последовательной реализации параллельных алгоритмов не выполнима физическая одновременность.
- На семействе устройств возможна относительная одновременность действий из разных частей, понимаемая как существование двух различных моментов времени: когда ни одно из них не начинало выполняться, и когда все они уже завершили выполнение. Существует, точнее, не исключена, и физическая одновременность: когда каждое действие выполняется.
- Синхронизация частей асинхронного алгоритма, не нарушающая условия готовности его частей и действий, может рассматриваться как частный случай асинхронности.

Особые сложности параллелизма вызывают вопросы доступа к памяти:

G. Каким образом взаимодействующие части параллельного алгоритма обмениваются данными?
H. Могут ли части параллельного алгоритма изменять состояние общей памяти и памяти других частей?
I. Может ли часть параллельного алгоритма воспрепятствовать использованию своей памяти другими частями?

Ответы на вопрос G

- Может быть определено соседство частей или действий, способных к сверхбыстрому обмену данными по особым каналам.
- Возможен обмен данными по известным параметрам доступа, не противоречащим дисциплине работы с памятью.
- Могут быть заданы специальные механизмы обмена через общую память и сообщения.

Ответы на вопрос H

- Части параллельного алгоритма могут изменять состояние общей памяти или памяти других частей.
- Изменение состояния общей памяти или чужой памяти может требовать специальных действий и условий готовности таких действий.
- Может существовать особая дисциплина функционирования изменяемой памяти, при которой доступ к ней происходит автоматически, без явного представления в алгоритме.

Ответы на вопрос I

- Часть алгоритма может не обладать своей памятью вообще.
- Локальная память части алгоритма может быть защищена от стороннего доступа.
- Возможна организация доступа к общей памяти с распределением прав на её использования частями алгоритма.

Кроме того, части алгоритма могут быть определены в разных моделях вычислений и над разными структурами данных. Оценка результата разработки параллельного алгоритма, кроме оценки сложности вычислений и объёма данных, осложнена целесообразностью оценивать выполнение разного рода трудно формализуемых критериев, часть которых, однако, поддаётся современным средствам верификации на моделях.

И всё же, сколь ни сложен мир параллелизма, программистам предстоит его понять и освоить!

В начале 1970-х годов В. Е. Котов и А. С. Нариньяни предложили весьма общую модель параллельных вычислений, согласно которой параллельные процессы выстраиваются из действий, связанных с так называемыми условиями готовности, определяющими возможность выполнения действия [17]. Интересно отметить появление новых архитектур, обладающих полным набором команд с условным исполнением.

Парадигмы программирования в форме языков и систем параллельного программирования представляют научное знание о потенциале информационных технологий (ИТ). В момент создания языки программирования отражают исходное знание специалистов относительно области приложения ИТ. Практика разработки и применения систем программирования конкретизирует и уточняет такое знание в форме отлаженных программ с комплектами данных для них и прецедентами успешного их применения. Развитие парадигм программирования отражает практичность языковых понятий и реализационных структур, используемых при создании сложных систем, включая высокопроизводительные и распределённые системы [22, 32].

Исследования в области информатики в настоящее время претерпевают изменение системы понятий, используемой в практических ИТ. В этом процессе расширяется пространство решений сложных задач, модернизируются методы развития информационных систем (ИС) на основе компьютерных сетей и многопроцессорных комплексов [35].

Интересно рассмотреть перспективы развития парадигм программирования, обусловленных изменением условий эксплуатации современных

информационных систем, особенно связанных с повсеместным распространением сетевых технологий, меняющих критерии оценки качества программ и методы обеспечения надежности и производительности программирования. Две основные линии такого развития – разработка распределенных информационных систем (РИС) и компонентное программирование (КП).

При практичном решении проблемы разработки языков параллельного программирования и подходов к их реализации в центре внимания вопросы выбора лаконичных форм представления программ, обеспечения конструктивных построений, гарантирующих сохранение правильности программ при их реорганизации, и поддержки расширяемости языка по мере развития средств и методов параллельного программирования.

Описание парадигмы параллельного программирования отражает прагматические различия в условиях реализации и применения изобразительных средств, используемых в жизненном цикле программ. Парадигмы параллельного программирования занимают нишу, связанную с реализацией программ выполнения вычислений на многопроцессорных системах для организации высокопроизводительных вычислений. Эта ниша обременена резким повышением трудоемкости отладки программ, вызванной комбинаторикой выполнения фрагментов асинхронных процессов.

При дальнейшем изложении в качестве базовой модели организации параллельных вычислений предложена модель подготовки многопоточных программ с явными средствами синхронизации действий, поддерживающая неявное распределение процессоров и методику представления параллельных алгоритмов с помощью такой модели.

Ознакомление с аппаратом сетей, приспособленных к достаточно тонкой детализации семантики параллелизма, приводит к алгебре процессов, иерархическим сетям, уровням определений и различной конкретизации известных понятий. Решение проблемы синхронизации независимых сетей в терминах синхросетей допускает использование представления сетевых и распределенных систем как основу при выборе единого сетевого представления семантики программ, процессов, языков и систем программирования [6, 17, 21].

Анализ обзора средств и методов представления программ и процессов в форме определения языков и изображения эквивалентных им сетевых граф-схем с целью демонстрации возможностей декомпозиции параллельных программ и реорганизации процессов их выполнения, включая распараллеливание, даёт основания для вывода, что полное представление об

асинхронных процессах, их эффективности и проблемах организации дают работы по сетям Петри [21].

Формализм сетей Петри представляет собой определение двудольного графа, вершины которого символизируют действия и сигналы. Дуги графа задают схему передачи сигналов при срабатывании действий. Различают классы сетей Петри с разными системами правил функционирования. Теоретически исследовано большое число классов, свойства которых гарантируют полезные для организации параллельных процессов характеристики, такие как выразительная сила, живучесть, безопасность, справедливость и др. Прогресс в области исследования свойств сетевых моделей послужил основой создания средств верификации параллельных и распределенных программ на моделях.

Правила функционирования сетей, в свою очередь, также могут быть заданы в форме сетей, что открывает путь к унификации средств исследования программ и процессов, позволяет решать вопросы организации межсетевых взаимодействий и наложения диагностических реакций. При таком рассмотрении формализма сетей удается достаточно точно моделировать известные аспекты аппаратной реализации параллельных вычислений, включая запуск очередных действий без ожидания завершения предшественников, размазывание значений по смежным потокам, организацию транзакций, сравнение с эталонами и др.

Варьирование правил функционирования сетей допускает как асинхронную, так и синхронную организацию срабатывания действий, включая дозирование нагрузки и специализацию процессоров и распределение действий по потокам выполнения. Использование иерархических, многоуровневых, структурированных и расширяемых сетей обеспечивает моделирование практически любых, накопленных в языках программирования, техник программирования и представления структур данных.

14.2. Практичные системы программирования

Реальная практика параллельного программирования сконцентрирована на моделях параллелизма для матричных вычислений. Идеальный параллелизм сводится к генерации параллельных итераций с локальными обменами данных при неограниченном числе процессоров над одноуровневой общей памятью. Критерий успеха основан на оценке ускорения вычислений и производительности систем в рамках программ над системами типов, допускающими статический контроль.

При измерении производительности суперкомпьютеров и в экспериментах с распараллеливанием программ активно используются задачи научных расчётов, преимущественно реализующих алгоритмы векторной обработки данных, удобно распараллеливаемых с помощью MPI. Практические задачи современного параллельного программирования обычно выглядят как приведение больших отлаженных ранее программ на Си или Фортране к форме, дающей выигрыш от распараллеливания с помощью штатных средств, включаемых в доступные системы программирования. В целом такая работа сводится к следующим видам работы:

- Разметка участков программы, допускающих автоматическое распараллеливание.
- Анализ участков и причин, препятствующих распараллеливанию программы.
- Выбор участков программы, допускающих их техническое приведение к форме, пригодной для распараллеливания.
- Изобретение рецептов полуручного преобразования текста программы с целью расширения возможностей распараллеливания.
- Приведение текста программы к предельно распараллеливаемой форме.
- Прогон распараллеленной версии программы для оценки выигрыша от параллелизма.
- Частичное перепрограммирование и отладка фрагментов программы для исключения или смягчения эффектов, препятствующих достижению нужных характеристик производительности.
- Установление частичной функциональной эквивалентности исходной программы и результирующей её версии.

Обычно компилятор поддерживает оптимизации, обеспечивающие устранение неиспользуемого кода, чистку циклов, слияние общих подвыражений, перенос участков повторяемости для обеспечения однородности распараллеливаемых ветвей, раскрутку или разбиение цикла, втягивание константных вычислений, уменьшение силы операций, удаление копий агрегатных конструкций и др.

Рассматривается зависимость ускорения вычислений от числа процессоров и объема общей и распределенной памяти. Выполняется систематическая замена рекурсии на циклы. Предпочитается однородное пространство процессоров, общая память, быстрые обмены, соседство, гарантирующие улучшение производительности систем для высокопроизводительных вычислений.

Заметно влияние дисциплины работы с памятью на характеристики параллельных процессов. Используется защищенная и размазанная память. Различны решения, принятые в разных языках программирования, по работе с многоуровневой и разнородной памятью (доступ, побочный эффект, реплики, дубли и копии). Обработка транзакций становится одной из типовых семантик работы с памятью в языках программирования.

Предлагаются средства и методы типизации управления процессами, удобными для подготовки программ, ориентированных на исполнение с помощью Open MP или MPI. Идеи языков программирования по сетевому представлению типов управления и дисциплины работы с памятью ещё не получили удобной системной поддержки.

Компонентно-ориентированная разработка ПО может следовать единому сетевому определению семантики языков программирования и процесса разработки программ.

Некоторую поддержку работ по подготовке программ, использующих параллельные процессы, обеспечивает созданная фирмой Microsoft технология .NET, позволяющая применять общие системные библиотеки в программах на разных языках программирования [33].

Лидирующие в области поддержки параллелизма фирмы Intel и Microsoft предлагают разработчикам высокопроизводительных вычислений новые решения, направленные на преобразования программ в процессе их создания и распараллеливания [27]:

- помощник по параллельному программированию (Parallel Advisor XE);
- компиляторы-отладчики и библиотеки (Parallel Composer XE2011);
- анализатор потоков и памяти (Parallel Inspector XE);
- профилировщик производительности (Parallel Amplifier XE).

Помощник по параллельному программированию анализирует исходный текст программы на языках Fortran или C до ее компиляции и отмечает узкие места, препятствующие распараллеливанию.

Анализатор потоков и памяти исследует полученный в результате компиляции объектный код и выявляет неудачное распределение ресурсов, снижающее производительность программы.

Профилировщик производительности позволяет наблюдать экспериментальную проверку достигнутой производительности программы.

Технология применения всех этих инструментов предполагает, что программист по ходу дела многократно принимает решения об изменении исходного кода программы и вручную включает в него необходимые прагмы,

указывающие компилятору допустимость оптимизирующих преобразований, что делает работу компилятора по распараллеливанию программы проще и надежнее. Возникает проблема автоматизации преобразования программ с целью приведения их к виду, удобному для распараллеливания компилятором и настройки объектного кода на конкретную многопроцессорную конфигурацию. Часть проблем решает схема подготовки параллельных программ с использованием библиотек преобразований исходного и объектного кода.

При демонстрации новых средств поддержки параллельного программирования представители фирм-разработчиков компиляторов показывают, насколько просто выполняется каждое такое преобразование в отдельной позиции программы. Проблемой является то, что таких позиций в нужной программе может быть не десятки и сотни, а тысячи и десятки тысяч. Отыскать их все без специальных инструментов уже проблематично. Кроме того, при выполнении таких работ возникает необходимость повторной разработки структур данных и схем управления вычислениями, а также обоснованного выбора средств реализации программного конструктива.

Так, например, возникает необходимость устранения или преобразования ряда механизмов работы с глобальными переменными, общими блоками данных, глубокой вложенности процедур, перехода от рекурсивных вызовов к итерациям со статическим управлением циклом и другое. Кроме того, возникает целый спектр аналитических выкладок по установлению фактических областей видимости имён, выбор между общей памятью и памятью конкретного потока. Выполнение таких работ вручную связано с риском порождения типичных для параллельных программ ошибок, таких как гонки памяти, конфликт доступа, взаимоблокировка и т. п.

Реорганизация векторов для нужд эффективного распараллеливания может идти разными путями: разделение на чётные-нечётные элементы, разбиение на половины или четверти, распределение по частям, требующим разной интенсивности вычислений и т.п.

Иногда такие проблемы можно решить на уровне макропреобразований текста программы, выполняемых препроцессорами, – надо лишь разработать систему подходящих конкретной программе макроопределений. Такие системы обычно не обладают универсальностью: они отражают индивидуальный стиль программирования, учитывают особенности мнемоники и неявные границы схем управления вычисления. В результате создаётся многопоточная версия программы, строго говоря, не являющаяся эквивалентом исходной программы, а лишь эквивалентно строящая её основные результаты.

Особый круг проблем связан с навыками учёта особенностей многоуровневой памяти в многопроцессорных системах. Обычное последовательное программирование такие проблемы может не замечать, полагаясь на решения компилятора, располагающего статической информацией о типах используемых данных и способного при необходимости выполнить оптимизирующие преобразования программы.

Следует отметить, что использование языков параллельного программирования в качестве языка представления исходной программы не гарантирует её приспособленность к удачному распараллеливанию.

При анализе пригодности программы к распараллеливанию анализируются следующие потенциальные зоны риска:

- объявление и использование глобальных переменных;
- области видимости переменных и констант;
- определения распараллеливаемых процедур, содержащие внутренние вызовы других процедур;
- рекурсивные определения;
- заголовки циклов, управляющие кратностью выполнения распараллеливаемого тела цикла;
- ветвления и переключатели, дающие развилки потоков;
- обработчики исключений;
- реакции на события и сообщения;
- позиции возможного обмена данными между потоками;
- тиражирование данных в общей памяти и в файлах;
- списки параметров функций, возможно преобразуемые в данные в общей памяти;
- декомпозиция управления циклом и реорганизация соответствующих структур данных;
- границы участков изменения значений переменных;
- переменные для значений промежуточных вычислений;
- многократно выполняемые идентичные вычисления – избыточные;
- инкрементные переменные, динамику изменения которых желательно сохранять для пост-анализа;
- позиции диагностической аранжировки для пост-анализа;
- фактически не задействованные переменные и константные или не используемые вычисления;
- параметризация зависимостей частей многопоточной программы от номера потока.

Вытекающие из такого анализа преобразования программы могут быть выполнены в стиле символьной обработки данных с помощью техники лексического и синтаксического анализа, регулярных выражений и макропреобразований. Чисто технические трудности связаны с разнообразием типов преобразований. Практики придерживаются традиционной методики отладки, т. е. разделяют процесс преобразований на шаги, на каждом из которых выполняется ровно один тип преобразований, после которого выполняется прогон программы на имеющихся тестовых данных с целью удостоверения сохранения частичной функциональной эквивалентности.

Таким образом, техника приведения программ к распараллеливаемой форме меняет стиль применения систем программирования, требует программистской работы на уровне грани между разбором программы и кодогенерацией, а также обустройства библиотек преобразований программы и её оптимизирующих преобразований.

Важно учесть изменение критериев применимости оптимизирующих преобразований. Если устранение избыточных вычислений в последовательной программе обычно оценивается как улучшение, то в параллельной программе оно может оказаться пессимизацией из-за появления лишних взаимосвязей между потоками и доступа к общей памяти.

Следует особо принять во внимание, что, хотя основная трудоёмкость жизненного цикла программ связана с тестированием и отладкой программ, именно это направление за полвека профессионального программирования практически не получило должного прогресса. Если первый барьер к успеху в параллельном программировании обусловлен последовательным стилем мышления, то второй очевидно зависит от до сих пор не преодолённой трудоёмкости отладки. Основные результаты здесь получены в форме парадигм функционального, структурного и объектно-ориентированного программирования и технологий поддержки непрерывной работоспособности разрабатываемой программы, смягчающих сложность тестирования, развития и версифицирования программного продукта.

Представляется важным принять во внимание новые технологии программирования, такие как экстремальное программирование (XP) и функционально-ориентированное проектирование (FDD), поддерживающие разработку программ по нисходящей и восходящей методике [31].

Интересно вспомнить, что среди экстракодов БЭСМ-6 был экстракод отладки, что позволяло программисту выстраивать свои сценарии отладки и тестирования на уровне кода программы.

Отладочные средства современных визуальных оболочек систем программирования, как и в ранние времена последовательного программирования, предоставляют для отладки параллельных программ средства выполнения в пошаговом режиме, отслеживания значений переменных и установления точек приостановок, для работы с которыми программист должен заранее подготовить комплекты тестов для всех участков программы и контроля корректности значений внутренних переменных. Возможно указание условий, при которых отладчик произведёт приостановку программы и обеспечит воспроизводимость повторного прогона программы.

Специфику параллельного программирования отражает лишь возможность переключения потоков, обнаружения совместно используемых (разделяемых) данных, реентерабельных процедур, вызываемых повторно из другого потока до их завершения, обнаружение тупиков и потерянных сигналов.

Для целей отладки выполняется компиляция специальной отладочной версии программы, что говорит о необходимости дополнительной отладки программы уже без отладочного сервиса. Впрочем, известны средства фоновой отладки, обеспечивающее слежение за отлаживаемой программой из независимого смежного процесса уровня ОС.

Лишь в последние годы в работах ИСП РАН наметилось более серьёзное отношение к проблемам отладки параллельных программ, что представлено в форме появления теорий для обработки информации, подверженной искажениям, и методики сравнительной отладки программ, в которой задействовано понятие эталонной программы и схемы распределённого отладчика.

14.3. Экспериментальные верификаторы

Часть трудностей планирования пространства поиска для практических задач преодолевается методами символьной обработки и классическими методами эвристического поиска в пространствах состояний, которые традиционно иллюстрируются на известных логических головоломках, конструированием инвариантов для показа прогресса. В целом современные системы верификации предоставляют возможность анализировать свойства достижимости, безопасности, живости, справедливости и другие, для которых удаётся представить подходящий комплект предикатов [29].

При многослойном описании семантики языка параллельного программирования наполнение программ может развиваться независимо от схем управления вычислениями, а схемы можно реорганизовывать без дополни-

тельной отладки наполнения. Выбор целевой архитектуры и зависящей от нее схемы управления может быть обусловлен как архитектурой, так и критериями эффективность (компактность программы, объем памяти, скорость обработки данных). Абстрактные схемы управления вычислениями определяются независимо от наполнения узлов схемы. Они играют роль макетов или моделей программ и работают подобно макросам (открытая подстановка), но с контролем соответствия параметров объявленным синтаксическим типам фрагментов. В новых языках программирования, поддерживающих параллелизм, таких как императивный C# и функциональный F#, имеется возможность манипулировать структурами данных, представляющими внутренний код программы.

Выделение схемного уровня упрощает включение в схему разработки программ механизмов верификации (подобие модели или соответствие аксиомам), а на их основе возможна проверка программ на правдоподобие, логический вывод свойств, выполнение индуктивных и дедуктивных построений. Кроме того, техника отладки программ обогащается возможностью привлечения протоколов ранее выполненных вычислений и приведения программ к нормальным формам, удобным для сведения к базовым/стандартным моделям параллельных вычислений.

При теоретико-экспериментальном исследовании параллельных программ нередко используют ручное создание эквивалентов на языках спецификаций, таких как Promela для Spin-анализатора. Promela – язык верификации программ с помощью моделей, обеспечивающий методы абстрагирования распределенных систем. Он рассчитан на проверку отдельных аспектов поведения процессов. Полная верификация складывается из серии шагов конструирования все более точных Promela-моделей программы. Каждая модель может быть верифицирована Spin-анализатором с учетом разнотипных допущений относительно контекста исполнения программы. Единожды установленная корректность программы распространяется на все ее последующие модели. Promela-модель строится из процессов, каналов сообщений и переменных. Процессы – это глобальные объекты для представления независимых составляющих распределенной системы. Каналы сообщений и переменные могут быть глобальными или локализованными внутри процесса. Процессы задают поведение, а каналы сообщений и переменные определяют контекст исполнения программы. Значения переменных типизированы и могут быть агрегированы в вектора и структуры. Поведение процесса задается как схема управления изменением состояний контекста исполнения программы. Для Spin-анализатора Promela-модель

сопровождается специальными ltl[1]-формулами, специфицирующими условия корректности верифицируемой программы. Такая техника верификации достаточна для обнаружения малоочевидных ошибок времени выполнения, наличия тупиков, оттеснения выполнимых действий, а также нарушения требований справедливости, корректной завершаемости, причинно-следственного и темпорального порядка и других формализуемых условий. Переход от практики ручного моделирования к формальному выводу Promela-моделей и/или логических формул может стать важным звеном обеспечения надежности параллельного программирования.

Такие модели используют вычислимые выражения ряда временных логик с учётом путей или состояний, перебор всех или указанных траекторий в программе, метод поиска контрпримеров, механизмы задания бесконечных последовательностей, конструирования контрольных автоматов по спецификации программы, функционирующих параллельно с исходной программой, конструирование гипотез об ошибках.

Например, система Spin на базе языка Promela предоставляет достаточно богатые и удобные средства для выражения требований к корректности параллельных программ. Известны прецеденты выявления с помощью таких систем весьма тонких, ускользнувших от обычных средств отладки и тестирования, ошибок в достаточно сложных программах. Но применение таких средств не внедрено в регламент систем программирования. Программист должен вручную написать спецификацию, представляющую процессы, каналы взаимодействия процессов и переменные простых реализационных типов данных. Процессы представляются моделями переходов от одного состояния к другому, и по комплекту таких моделей выстраивается полная функциональная модель программы. При описании системы параллельных процессов возникает понятие «тип процесса», определяющее тип поведения взаимодействующих действий. Существенным ограничением системы является допуск лишь переменных с конечной областью определения, включая одномерные вектора. Это соответствует специфике языков низкого уровня, отладка программ на которых наиболее сложна. Языки параллельного программирования, за редким исключением, представляют собой объединение высоко уровневых средств обработки структур данных с низкоуровневыми средствами управления вычислениями, что дает основания для практики включения средств верификации в сценарий кодогенерации параллельных программ с одной стороны и распространения техники верификации на более сложные структуры данных с другой стороны.

[1] Linear time temporal logic formulae

Важно преодолеть отсутствие связи практики параллельного программирования с результатами исследований в области верифицирующей компиляции и типизации схем управления параллельными процессами, отражающих специфику требований, предъявляемых к высокопроизводительным программам. Между тем, методы и техника верификации выходят на уровень практичного применения: известны прецеденты успешного применения таких систем как Promela, Spin и др. в области проверки протоколов взаимодействия процессов. Основная трудность – практика ручного перепрограммирования параллельных алгоритмов. Имеются свидетельства, что ошибки в спецификациях обнаруживаются намного чаще, чем ошибки в программах. Вывод – еще не сложилась общая парадигма параллельного программирования, увязывающая в единую картину средства и методы создания параллельных программ и их усовершенствования по мере технического прогресса.

15. МОДЕЛИ ПАРАЛЛЕЛИЗМА В ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

История языков программирования накопила целый ряд примеров лаконичных форм представления программ, начиная с умолчаний, неявных циклов и операторов ввода-вывода в языке Fortran. С точностью до реализационной прагматики, при разработке языков параллельного программирования можно унаследовать языковые конструкции и механизмы из привычных парадигм программирования и зарекомендовавших себя языков параллельного программирования.

Прежде всего, это алгебраические механизмы распространения функций и операций относительно структур данных, предложенные в первом языке параллельного программирования APL. Дальнейшее упрощение изобразительных средств управления параллелизмом дает предложенный в языке Sisal подход к неявному распараллеливанию циклов на основе построения пространства итераций по пространству обрабатываемых данных.

Ещё острее становятся проблемы разработки, отладки, тестирования и верификации параллельных программ по сложности весьма превосходящие обычное программирование. Именно здесь сосредоточена исследовательская активность современного языкотворчества и системного программирования, поиск средств и методов интеграции удачного опыта параллельного программирования и успешного обучения методам параллельных вычислений. Акцент на анализ подходящих решений в области разработки

переносимых программ, средств визуализации процессов, типизации управления и дисциплины доступа к памяти.

Рассмотрим развитие моделей параллелизма в языках высокого уровня. В этом плане заслуживают внимания средства и методы представления программ организации параллельных процессов, исторически сложившиеся в современных языках программирования, и резервы уточнения семантики языков высокого уровня с целью повышения эффективности и надежности использования новых возможностей аппаратуры и информационных технологий. Рассмотрение моделей параллелизма, встроенных в наиболее известные языки программирования, такие как языки APL, Algol-68, Setl, Ada и др. [4,23,26,44], дополнено и небольшим экскурсом в отечественные разработки в области языков параллельного программирования (БАРС, Поляр, Норма, mpC и др.) [17,20].

Практически в мире параллелизма все базовые понятия программирования претерпевают изменение или расширение (программа, ветвление, цикл, событие, память, результат). Появляется ряд специфических для параллельного программирования понятий (процессор, поток, ожидание, длительность, фильтр, барьер). Программы становятся многопоточными, циклы – параллельными, память обретает копии и реплики, события происходят одновременно в разных синхронизуемых процессах, вычисление результата может не означать завершение процесса. Такая ревизия понятий влияет не только на стиль программирования, но и изменяет характер компиляции на этапе генерации кода программы. Возрастает роль техники использования многократно используемых компонент схемного уровня, соответствующего средствам типизации управления процессами.

Существуют версии ряда стандартных языков императивного программирования, приспособленные к выражению взаимодействия последовательных процессов в предположении, что в каждый момент времени существует лишь один процесс. При таком подходе в программе выделяются критические интервалы, учет которых полезен при распараллеливании программ. Многие традиционные языки программирования приспособлены к выражению локального параллелизма с помощью специальных расширений или библиотечных функций, обеспечивающих выделение участков с независимыми действиями, пригодными для распараллеливания компилятором.

Большие надежды связаны со строго функциональным подходом к спецификации параллельных программ и типов данных в языке с предпочтением так называемой «ленивой» схемы вычислений. Противопоставление достоинств и недостатков ленивых и энергичных методов вычислений от-

части смягчается концепцией «монад» в строго функциональном языке программирования Haskell. Особенности определения семантики языковых конструкций по-прежнему не отражают решение проблем обеспечения удобочитаемости программ и их отладки. Табулирование сложных вычислений, называемое «мемоизация», становится популярным практичным инструментом снижения сложности вычислений.

В настоящее время существуют сотни функциональных языков программирования, ориентированных на разные классы задач параллельного программирования. Языки функционального программирования обогатились типовыми средствами практически всех известных подходов к представлению программ и организации вычислительного эксперимента и информационных процессов. Обеспечена организация параллельных процессов. Возможна визуализация данных и программ. Имеются средства стандартного и объектно-ориентированного программирования. Поддержано управление компиляцией и конструирование компиляторов. На сегодняшний день утратили актуальность опасения относительно ресурсно-эксплуатационных трудностей функционального программирования. Функциональное программирования вносит свой вклад техникой рекурсивных определений и отображений, параллелизмом обработки аргументов функций, ленивых вычислений, а также сведением понятия «условия» в ветвлениях к понятиям «страж» или «образец».

В семействе практичных языков функционального программирования заметное место занимают языки организации распределенных и параллельных вычислений. Практики с большой похвалой отзываются о языке функционального программирования *Erlang* фирмы Ericsson.

Объектно-ориентированное программирование активизировало, сделало обыденной перегрузку операций и функций.

Более специфичные и эффективные формы выражений возникают при обработке мультизначений с помощью фильтров и проекций в новых мульти-парадигматических и учебных языках программирования.

Fortran – сопрограммы

Средства представления параллельных вычислений доступны, начиная с первых языков высокого уровня. Языки Fortran II и Fortran IV [16] были достаточно универсальны для представления программ организации параллельных процессов. Механизм сопрограмм, допускающий многовходовые (Entry) процедуры, позволял представлять программы взаимодействующих процессов и декомпозировать программу на управляющую и вычисляющую части, выглядящие как независимые компоненты программы, но это

не привело к практике параллельного программирования и постепенно сопрограммы превратились модули, обеспечивающие представление иерархии функций подобно иерархии классов в ООП. Синтаксически средства параллельного программирования выглядят в современных Fortran-программах как разметка текста ключевыми словами и вызовы библиотечных функций.

Lisp – отложенные вычисления

Унификация представления данных и программ позволила передачу параметров по заранее вычисленному значению или указателю пополнить возможностью организовывать передачу представления выражений, вычисление которых можно выполнять по мере необходимости. Такая возможность привела к концепции «ленивых» вычислений и развитию методов динамической оптимизации процессов в отличие от статической оптимизации программ [40,42].

APL – векторные операции

Более удачной оказалась идея поэлементной обработки однородных структур данных в языке APL, особенно с появлением векторных архитектур Cray [23]. В 1962 году был предложен интересный механизм реализации многомерных векторов, приспособленный к расширению и распараллеливанию обработки данных. Сложные данные представляются как пара из последовательности скаляров и паспорта, согласно которому эта последовательность структурируется. Такое решение позволяет любое определение функции над скалярами автоматически распространять на произвольные структуры данных из однотипных скаляров. И в настоящее время для большинства специализированных языков параллельного программирования типично, что сложные построения факторизуются с учетом особенностей структуры данных так, что выделяются несложные отображающие функции, «просачиваемые» по структуре данных с помощью функций более высокого порядка – функционалов. В результате можно независимо варьировать структуры данных, функционалы, методы сборки полного результата и набор отображаемых множеств. Целенаправленно выделяются конвейерные процессы, приспособленные к минимизации хранения промежуточных результатов. С 1980-ых годов эта идея наследуется многими языками, поддерживающими параллелизм и в наше время фактически является стандартной.

Algol-68 – критические участки и семафоры

К концу 1960-х годов сложилось значительное разнообразие теоретических моделей параллелизма, при исследовании которых проявилась проблема надежности параллельных вычислений, выразившаяся в неожиданном различии поведения императивной последовательности действий в зависимости от включаемых в нее фрагментов «независимых» процессов, нарушающих императивность. Для профилактики таких эффектов в семантику языка Algol-68 включается идея непрерывно исполняемых критических участков и представления их защиты в терминах семафоров [26].

Unix и JCL – управление процессами

Независимо идеи явного порождения процессов и организации их взаимодействия через каналы возникают в языках управления заданиями и процессами в операционных системах [43].

Setl – множества и кванторы

Другой подход к надежности программирования предложен в языке теоретико-множественного программирования Setl, ориентированном на активизацию интуиции грамотных математиков при разработке спецификаций программ в терминах преобразования множеств, естественно подразумевающих возможность параллельной обработки элементов множества, причем в реализационно-независимом стиле [44]. Наследование решений из универсальных языков сверх высокого уровня, таких как Setl, абстрагирование данных и процессов в которых приспособлено к гибкому и строгому структурированию, удобно для культивирования доказательных построений в практике параллельного программирования. В этом плане представляет особый интерес эксперимент по развитию теоретико-множественной семантики языка Setl, в котором весьма общее построение формул с кванторами над множествами погружено в обычную схему последовательного управления процессами. Реализация языка Setl характеризуется богатым полиморфизмом. Для представления множеств используется около двадцати разных структур данных, выбор которых осуществляется системой программирования в зависимости от динамики операций над множествами. В результате программируемые функции слабо зависят от реализационной структуры данных. В практике управления процессами используется понимание команд как позиций независимого порождения процессов. Такое понимание естественно согласуется с идеями теории

множеств о независимости элементов множеств и может служить основой архитектуро-независимой семантики языка программирования.

SQL - Транзакционная память и нормальные формы

Много более заметная проблема параллельной обработки данных независимыми операторами обнаружилась в практике применения общих баз данных, приведшей к выделению языка запросов (SQL) и концепции транзакционной памяти, теперь рассматриваемой как перспективная основа семантики языков параллельного программирования [10].

Occam – взаимодействие «процесс-канал»

Середина 1970-х годов характеризуется кризисом технологии программирования, выход из которого тогда виделся в массовом переходе к параллельному программированию. Активные исследования разрешимых классов параллельных схем программ показали ряд неудобных, снижающих эффективность распараллеливания, конструкций, таких как ветвления. Э. Дейкстра опубликовал решение этой проблемы в форме защищенных команд, которая нашла свое место в определении языка Occam,предоставляющем для транспьютерного программирования модель взаимодействия CSP процесс-канал [28]. В эти же годы популяризируются идеи структурного программирования, нацеленные на снижение сложности отладки программ, близкие идеям функционального программирования, которое теперь рассматривается как один из универсальных методов представления удобно распараллеливаемых программ [11].

Ada – «рандеву»

В проект языка Ada предпочли включить механизм «рандеву», сводящий представление взаимодействия процессов к рассредоточенному обмену сообщениями, подобному сигналам в оборудовании и модели CCS, что можно рассматривать как приаппаратное низкоуровневое средство, несколько диссонирующее с высоким уровнем языка [4].

БАРС – сетевое управление

В нашей стране разработаны языки БАРС и Поляр с разными концепциями сетевого управления процессами и представления дисциплины доступа к памяти [17,20]. Программирование на уникальном по уровню средств управления процессами языке БАРС нацелено на обеспечение высокопроизводительных вычислений и организацию асинхронных параллельных процессов. При создании языка БАРС в качестве базового ЯВУ

был привлечен популярный язык Pascal, в 1970-е годы перераставший из учебного в производственный язык системного программирования. При сохранении основных принципов семантики вычислений были существенно обобщены средства структуризации данных на основе понятия «мультимножество», приспособленного к именованию элементов структур данных и учета кратности их использования. Работа с именованной памятью (Name-oriented) дополнена возможностью задавать дисциплину доступа к элементам памяти. Идеи более ранних языков параллельного программирования были развиты и обогащены в языке БАРС в трех направлениях:

1) в качестве базовой структуры данных были выбраны мультимножества (размеченные множества с кратностью элементов);

2) описание элементов памяти сопровождается предписанием дисциплины доступа к памяти;

3) средства управления асинхронными процессами включали механизм сетей Петри, координирующих работу независимо созданных функциональных фрагментов.

Процедуры в таком языке приспособлены к варьированию дисциплины доступа к данным и схемы управления процессами обработки данных. Сети Петри позволяют независимые описания процессов связывать в терминах разметки. Узлы с одинаковой разметкой срабатывают одновременно. Процесс обработки данных рассматривается как распределенная система, находящаяся под сетевым управлением. Узлы такой системы могут сработать в зависимости от условий готовности разной природы: доступность ресурсов, сигналы монитора, внутри сетевые отношения, иерархия сетей, правила функционирования разносортных подсетей. Вычисления, как и в языках APL и Sisal, распространяются со скаляров на сложные структуры. Радикальное продвижение в повышении уровня программирования, предложенное в языке БАРС, заключается в переносе механизма типизации данных на проблему типизации схем управления.

Sisal – однократные присваивания и пространства итерирования

Авторы строго типизированного функционального языка параллельного программирования Sisal создали интересный прецедент по расширению и уточнению системы понятий программирования для нужд представления и реализации масштабируемых параллельных вычислений [39]. Программа в этом языке строится из участков с однократными присваиваниями, удобными для оптимизирующих преобразований, включая распараллеливание. В структуре цикла выделены позиции для формирования пространства параллельных итераций, фильтрации или сборки параллельно полученных

результатов и обработки потоков данных при развитой системе работы с векторами, включая методику распространения скалярных операций на структуры данных [13,39].

Произошедшее в конце 1970-х годов отвлечение внимания от кризиса технологии программирования на задачу освоения микропроцессоров не остановило поиск языковых решений для представления программ, обладающих параллелизмом. Появился функциональный язык параллельного программирования Sisal, позволяющий формировать пространства итераций для эффективного распараллеливания циклов компилятором. Название языка функционального программирования Sisal расшифровывается как “Streams and Iterations in a Single Assignment Language”. Система вычислений в языке Sisal использует понятие «мультизначение», позволяющее подобно языку APL распространять скалярные действия на данные любой структуры, а их обработку осуществлять на многопроцессорных конфигурациях. Отображение мультизначения рассматривается как обработка его элементов на независимых процессорах. Результаты отображения могут повергнуться свертке или фильтрации. Sisal-программа представляет собой набор функций, допускающих частичное применение, т.е. вычисление при неполном наборе аргументов. В таком случае по исходному определению функции строятся его проекции, зависящие от остальных аргументов, что позволяет оперативно использовать эффекты смешанных вычислений и определять специальные оптимизации программ, связанные с разнообразием используемых конструкций и реализационных вариантов параллельных вычислений. Основное продвижение по технике программирования в языке Sisal – развитие структуры циклов для их реализации на параллельных процессорах. Введено понятие «пространство итераций» и предложена специальная конструкция для фильтрации мультизначений, получаемых при совмещенном исполнении итераций и участков повторяемости. Формирование мультизначения управляется представлением пространства итераций и учетом зависимостей между одноуровневыми итерациями. Работа с именованной памятью (Name-oriented) освобождена от проблемы побочных эффектов методом локализации участков с однократными присваиваниями – SSA-форм, что делает программы удобными для преобразований, оптимизации и компиляции, включая распараллеливание и масштабирование.

Oberon – обучение управлению процессами

1980-е годы знаменует переход к сетевой обработке данных и признанию потенциала ООП при организации информационных бизнес-процессов. Появляются Oberon, Eiffel, SmallTalk-80, C++, Erlang, Perl и

другие языки, отчасти компенсирующие недостаток базовых средств и методов реализации массово используемых императивных языков программирования в новых условиях. В наши дни Вирт позиционирует язык Oberon как кандидат на включение в школьную программу информатики вместо языка Паскаль [5].

Python и Ruby – разработка распределённых систем

Общий прогресс в эксплуатационных характеристиках оборудования с 1990-х годов резко расширил возможности сборки информационных систем из готовых компонентов и сделал доступными свободно распространяемые программные инструменты конструирования систем программирования, как правило, поддерживающие организацию параллельных процессов, если не собственно на уровне языка, то на уровне библиотечных компонент. Мультипарадигматические языки Python и Ruby показывают хорошие результаты в программировании сетевых процессов для многопроцессорных комплексов и привлекают большое число сторонников. Язык Python зарекомендовал себя как удобное средство разработки распределенных систем и сетевого программирования [19].

Haskell – «ленивые» вычисления и мемоизация

Кроме того, существуют сотни функциональных языков программирования, ориентированных на разные классы задач параллельного программирования. Языки функционального программирования обогатились типовыми средствами практически всех известных подходов к представлению программ и организации вычислительного эксперимента и информационных процессов. Обеспечена организация параллельных процессов. Возможна визуализация данных и программ. Имеются средства стандартного и объектно-ориентированного программирования. Поддержано управление компиляцией и конструирование компиляторов. Методы функционального проектирования и программирования обеспечивают технику представления и отладки функциональных моделей, спецификации и верификации программ, исследования их свойств и экспериментального сравнения моделируемых параллельных процессов с моделями и прототипами. Функциональный подход исторически является основой для исследования средств и методов программирования, прототипирования и декомпозиции программируемых систем и развития современных методов параллельного и многоязыкового программирования. Разработан чисто функциональный язык Haskell, предлагающий эффективную модель «ленивых» вычислений с ме-

моизацией промежуточных результатов по принципу полузабытых методов «математического динамического программирования» [45].

Норма – параллельные вычисления

Система поддержки параллельного программирования для решения задач вычислительной математики.

MPI и Open MP – средства распараллеливания программ

Появляются популярные системы MPI и Open MP, обеспечивающие эффективность параллельного программирования в рамках языков Fortran и C [27]. В MPI взаимодействие процессов обеспечивается через посылку сообщений между процессорами. Open MP предоставляет процессам возможность использовать разные виды памяти, включая быструю общую память, что при удачном ее распределении позволяет достигать высокой эффективности.

mpC – синхронизация семейств процессоров

Практически в мире параллелизма все базовые понятия программирования претерпели изменение или расширение (программа, ветвление, цикл, событие, память, результат). Появился ряд специфических для параллельного программирования понятий (процессор, поток, ожидание, длительность, фильтр, барьер). Программы стали многопоточными, циклы – параллельными, память обретает копии и реплики, события происходят одновременно в разных синхронизуемых процессах, вычисление результата может не означать завершение процесса. Такая ревизия понятий влияет не только на стиль программирования, но и изменяет характер компиляции на этапе генерации кода программы. Возрастает роль техники использования многократно используемых компонент схемного уровня, соответствующего средствам типизации управления процессами. Так например, язык mpC предлагает более детальный учет механизмов взаимодействия параллельных процессов в терминах барьеров и специальных категорий переменных, обладающих особым, аппаратно реализуемым поведением.

Java, C#, Scala, F# - библиотеки управления синхронизацией

В новых системах программирования для языков Java, C#, Scala, F# и т.д. существенно повысилась результативность системных решений в области работы с памятью, компиляции, манипулирования комплектами функций и классами объектов, выделение которых по существу обусловлено результатами теоретических работ в области системного статического

анализа, основу которых все в большей мере составляет функциональный подход [33,36,37]. Получили значительное развитие методы декомпозиции программ в рамках объектно-, субъектно- и аспектно-ориентированных подходов к определению систем программирования на базе многократно используемых компонент. Для современных областей программирования и проектирования характерна интеграция средств и методов из разных парадигм, что может привести к профессиональной консолидации программистского корпуса.

Изучаются средства организации параллельных процессов средствами языка функционального программирования F# с библиотеками .Net. На этом языке рассматривается возможность реорганизация программ, что поддержано в языке специальными средствами типа Quotation – доступ к внутреннему представлению программ в виде структуры данных. Анализируется вклад типизации данных в построение эффективных программ. Использование модифицируемой и защищенной памяти. Прикладные аспекты работы с информацией, отражающей специфику области приложения программ (measure).

C# дает возможность встраивать функциональные построения в контекст привычных императивных программ. Рассматривается стыковка программ на новых языках с производственными системами, разрабатываемыми на базе библиотек .Net. Возможность оперирования деревом разбора программы и ее исполнимым кодом. Вопросы защиты программ и данных. Управление дисциплиной доступа к памяти и представление запросов к базам данных.

CUDA – многопроцессорные видеоплаты

Появление технологии CUDA, объединившей в графических ускорителях достоинства этих ранее сложившихся подходов, выводит параллельное программирование в ранг массово доступных методов создания программных систем благодаря преодолению стоимостного барьера. Следует ожидать, что развитие парадигмы параллельного программирования приведет к улучшению средств поддержки полного жизненного цикла программ, включая активное использование методов верификации взаимодействия процессов, автоматизацию приведения обычных программ к эффективно распараллеливаемой форме, обеспечение мобильности параллельных программ относительно параллельных архитектур, а следовательно и к повышению эффективности и надежности программ, приспособленных к многократному использованию типизированных решений особо важных наукоемких задач. На повестке дня – разработка методов архитектурно неза-

висимой кодогенерации масштабируемых параллельных программ, легко настраиваемых на особенности используемых вычислительных комплексов. Так, например, новый язык программирования OpenCL обеспечивает параллелизм на уровне инструкций и на уровне данных для различных графических и центральных процессоров.

16. ЯЗЫКИ СВЕРХВЫСОКОГО УРОВНЯ

Подготовка программ на базе языков сверхвысокого уровня (ЯСВУ) нацелена на длительный срок жизни запрограммированных решений особо важных и сложных задач. Удлинение жизненного цикла достигается представлением обобщенных решений с определенной степенью свободы по отношению к полным пространствам допустимых смежных компонент, реализованных ранее или планируемых на будущее. Реализационное сужение семейства процессов, допускаемых семантикой

ЯСВУ, противоречит его целям или концепциям по следующим прагматическим мотивам:

- высокий уровень абстрагирования программируемых решений;
- решаются задачи, зависящие от непредсказуемых внешних факторов;
- базовые средства и/или алгоритмы вычислений используют параллелизм;
- актуальны прагматические требования к темпу и производительности вычислений;
- эксплуатируются динамически реконфигурируемые многопроцессорные комплексы.

Обычно создатели нового ЯСВУ используют в качестве исходного материала один или несколько базовых ЯВУ и встраивают в них изобретаемые средства и методы. От базовых ЯВУ наследуются парадигмы удовлетворительного решения сопутствующих задач. В таких случаях парадигматическая характеристика ЯСВУ может формулироваться относительно базовых ЯВУ, хотя внешнее синтаксическое сходство языковых конструкций иногда скрывает совсем другую семантику.

Для ЯСВУ характерно применение регулярных, математически ясных и корректных, абстрактных структур, при обработке которых возможны преобразования данных и программ, использование подобий и доказательных построений. Все это призвано гарантировать высокую производительность

вычислений, надежность процесса разработки программ и длительность их жизненного цикла. Типичны алгебраические спецификации, теоретико-множественные построения, параллелизм, модели процессов разработки программ. Изобретаются специальные системные средства, повышающие емкость представлений, их общность и масштабируемость. Естественный резерв производительности компьютеров – параллельные процессы. Их организация требует контроля и детального учета временных отношений и неимперативного стиля управления действиями. Суперкомпьютеры, поддерживающие высокопроизводительные вычисления, нуждаются в особой технике системного программирования, которая еще не сложилась, хотя уже имеется опыт эффективного решения особо важных задач.

Результативен графово-сетевой подход к представлению систем и процессов для параллельных архитектур, получивший выражение в специализированных языках параллельного программирования и суперкомпиляторах, приспособленных для отображения абстрактных структур управления процессами уровня задач на конкретную пространственную структуру процессоров доступного оборудования. Появляются языковые средства поддержки полного жизненного цикла программ, совмещенного с процессом изучения класса решаемых задач. Для простоты изучения ЯСВУ могут выглядеть как расширение привычных ЯВУ, возможно со средствами низкоуровневого управления процессами. Характерна ортогональность семантических систем, варьирование методов реализации и обобщение схем управления процессами, включая разработку необычных/непривычных средств.

Таким образом можно как бы «просачивать» определения функций над простыми данными, распределять их по структурам данных и тем самым распространять простые функции на сложные данные подобно матричной арифметике. Похожие построения предлагаются Бэкусом в его программной статье о функциональном стиле программирования и в языке APL, ориентированном на обработку матриц [38].

Существует ряд языков функционального программирования [25], требующих или допускающих спецификацию объектов, что, кроме дисциплины программирования, дает средства для корректной работы с пакетами, сопряжения с модулями на других языках, оптимизирующих преобразований, распараллеливания и верификации программ (Sisal, ML, Haskell, Scala и др.) [36,39,45].

Здесь рассмотрены лишь ключевые идеи ряда ЯСВУ, ориентированных на параллельное программирование, без описания деталей реализационной семантики. При переходе к компьютерным языкам предстоит рассмотреть

языки разработки, тестирования, отладки, спецификации и верификации программ, СУ-конструирование, средства и методы оптимизации программ и компонентного программирования, а также ряд новых технологий разработки распределенных информационных систем.

Как правило языки параллельного программирования включают в себя средства, характерные для разных парадигм. Это определяет целесообразность трансформационного подхода к накоплению правильности программных решений при разработке и модернизации параллельных программ на разных языках в рамках общей системы программирования. Развитие ЯСП в настоящее время ориентировано на решение задач на основе общих библиотечных модулей, обеспечивающих эффективную организацию процессов, или подъязыков, допускающих многопоточное программирование. Это, увы, не исключает реальную практику ручного распараллеливания ранее отлаженных обычных программ, приведения их к виду, удобному для применения производственных систем поддержки параллельных вычислений. Значительная часть таких работ носит технический характер и заключается в систематической реорганизации структур данных, изменении статуса переменных и включении в программу аннотаций, сообщающих компилятору об информационно-логических взаимосвязях. Существенным ограничением результата ручного распараллеливания является не только опасность повторной отладки алгоритма, но и его избыточная зависимость от характеристик целевой архитектуры.

Для преодоления такой зависимости на уровне операционной семантики языка параллельного программирования предлагается придерживаться следующих вспомогательных определений:

Схема определяет варианты управления действиями в многопоточной программе. При выделении схем можно с помощью параметров выразить зависимость потоков от событий, действий, различных процессоров и изменяемых переменных. Формально схема работает как макроопределение – это открытая подстановка, выполняющая ту же работу, что препроцессоры во многих ЯСП. Разница в том, что схема включается в программу на этапе синтаксического анализа, ее параметры обязаны синтаксически соответствовать понятиям языка и реализация схемы может быть при компиляции заменена ее более эффективным библиотечным эквивалентом. Схемы могут быть рекурсивны или содержать циклы. (На уровне языка высокого уровня пользователь сможет сам объявлять реализационные эквиваленты схем. Такая возможность может быть востребована при распараллеливании последовательных программ.)

Программа – это синхросеть потоков, выполняющихся в общей памяти – заданном контексте, возможно с выделением значимых событий – барьеров или синхронизаторов. Невыделенные барьеры не влияют на выполнение потоков. Одноименность барьеров символизирует одновременность прохождения потоков через них – это событие, общее для всех потоков, включающих в себя значимый барьер Программа правильна, если существует контекст, в котором синхронизация потоков корректна, т.е. не возникает недостижимости синхронизатора в одном из потоков.

Фрагмент – это укрупненное действие, объект управления при определении потоков, схем и программ, основная единица, задающая логику обработки данных в терминах параллелизма, ветвлений, многократности исполнения и распределения работ по процессорам. При работе с переменными фрагмент ограничен требованием однократности присваивания. Сложные фрагменты программы строятся с помощью схем из более простых и из базовых действий, которые задают точки и методы информационной обработки, а также определяют информационные взаимодействия фрагментов программы друг с другом через общую память. Для базовых действий известны критерии их выполнимости, при нарушении которых действие считается не выполнявшимся.

Поток строится из фрагментов, возможно синхронизованных по одноименным барьерам с другими потоками. Должен существовать контекст, в котором выполнимы все действия потока, выполняемого автономно. При формировании потока выбирается схема управления фрагментами, точнее порядок и способ выполнения действий, возможно со счётчиком для удобства отладки. Для работы с рекурсией и бесконечными циклами предполагается использование ограничения времени выполнения потока.

Слой – это схема, объединяющая действия, порядок выполнения которых не задан, что рассматривается как параллельное выполнение. При необходимости действия можно упорядочить или частично синхронизовать с помощью последовательности барьеров в рамках более общей синхросети.

Линия – это схема, задающая порядок выполнения действий как порядок их вхождения, т.е. последовательно, подобно линейному участку в обычной программе.

Обход – это схема, которая дает возможность определять ветвящиеся процессы с помощью условий выбора, вероятности срабатывания действий или произвольного выбора. Условие считается вычисленным лишь, если оно истинно**.** При невыполненном условии недостижимы барьеры, расположенные внутри невыбранного фрагмента.

Включение – это схема, обеспечивающая многократное использование общих фрагментов потока или схемы, задаваемых непосредственно схемой или с подстановкой имен барьеров, фрагментов, переменных и процессоров, а также вызовы функций, включаемых в динамике.

Назначение – это схема, позволяющая явно распределять работу многопоточной программы по процессорам.

Итерация обеспечивает многократность выполнения фрагмента. Можно представлять неограниченную итерацию в расчете, что ее ограничат смежные потоки с помощью синхронизации, счётчика или ограничения длительности.

Вычисление выражения – это простейшее действие. Любой процессор содержит память для текущего результата его работы. Возможна реорганизация структур данных, при которой перемещаются их элементы из левой в правую часть с учетом определения структуры данных правой части и фильтра, выделяющего элементы из левой части. Результат – пара из полученной структуры и остатка преобразуемой. Критерий выполнимости – наличие данных, достаточных для выполнения реорганизации. Например, нельзя в вектор разместить объекты из пустой очереди. Действие «подождет», когда в очереди появятся данные.

Контекст – это память для общедоступных данных многопоточной программы. Она может быть неоднородной по времени доступа. Контекст задан как структура данных, связывающая имена со значениями констант и переменных, определениями функций или схем управления. Контекст можно уточнять заданием новых связей. Все имена уникальны, т.е. нет локализации по блокам, хотя процессоры имеют свою защищенную память. Фактически связывания имен – это присваивания, но с сохранением старых значений, которые можно достать при необходимости специальными функциями.

Многопоточная программа – сеть потоков, выполняющихся в общей памяти – заданном контексте, возможно с выделением значимых событий – барьеров или сихронизаторов. Невыделенные барьеры не влияют на выполнение потоков. Одноименность барьеров символизирует одновременность прохождения потоков через них – это событие, общее для всех потоков, включающих в себя значимый барьер. Программа правильна, если существует контекст, в котором синхронизация потоков корректна, т.е. не возникает недостижимости синхронизатора в одном из потоков.

Многопроцессорный комплекс состоит из конечного числа процессоров, обладающих своими системами команд, включающими общие команды управления процессами и их взаимодействиями.

Последовательность вычисления не обязана совпадать с последовательностью размещения вычисленных значений. Основные структуры данных – это очереди, элементы которых доступны последовательно и могут быть размещены рассредоточено, а новые элементы добавляются в хвост или голову очереди, и вектора с обычным индексированием элементов, размещенных в соседних регистрах, допускающих произвольный доступ.

При таких допущениях потоки могут разрабатываться на базе разных языков и отлаживаться на разных процессорах в расчёте на предстоящую интеграцию многопоточной программы с использованием конверторов с разных исходных языков на окончательный язык.

Проблема разработки языков и систем параллельного программирования для поддержки параллельных вычислений непременно включает автоматизацию распараллеливания и верификации программ и настраиваемой кодогенерации на типовые многопроцессорные комплексы.

При компиляции нужна типизация схем управления процессами, преобразования сети потоков и анализ информационных связей с целью верификации и вывода оптимального размещения потоков по процессорам. Кодогенерация должна допускать оперативный учет смены используемого многопроцессорного комплекса.

Важный акцент – параллелизм имеет самостоятельное значение, а не является пристройкой к традиционному программированию.

Появление нового поколения языков программирования, таких как C# и F#, показывает общую тенденцию включения средств динамической обработки кода программ, что позволяет при реализации параллельных алгоритмов решать задачи верификации и оптимизация программ, включая их распараллеливание. Следует отметить, что кроме собственно правильно-

сти, понимаемой как соответствие аргументов результатам, параллельные программы должны отвечать достаточно сложным критериям, таким как корректность синхронизации процессов, надежность, живучесть, справедливость и др., не отслеживаемые обычной схемой компиляции программ. В дополнение к понятию «ошибка» возникает не менее важное направление анализа разного рода тормозящих эффектов, снижающих производительность программ при их формальной корректности.[29] Конструктивные методы параллельного программирования желательно нацеливать на обеспечение нужных свойств по построению, основных на определении расширяемых и трансформируемых схем.

Трансформационное программирование обеспечивает сведение сети разносортных потоков к однородной системе потоков, однозначно отображаемых на заданный комплекс процессоров – размещение потоков по процессорам или назначение процессоров для выполнения потоков. Применительно к высокопроизводительному программированию это дает формальные методы представления иерархии данных в многоязыковых системах программирования. Такие методы обеспечивают функциональную унификацию реализуемых понятий, включая абстрагирование данных и процессов в рамках единой методики представления параллельных программ на основе выделения базовых функций и структурирования схем управления вычислениями с помощью специально разрабатываемых абстрактных форм и конкретных шаблонов.

Функциональное программирование вносит свой вклад техникой рекурсивных определений и отображений, параллелизмом обработки аргументов функций, асинхронностью ленивых вычислений, а также сведением понятия «условия» в ветвлениях к понятиям «страж» или «образец». Использование методов функционального программирования радикально снижает трудоемкость отладки программных компонент благодаря предпочтению достаточно универсальных определений и динамическому контролю корректности вычислений. В центре внимания – выделение базовых функций, реализуемых в наиболее общей форме. Благодаря этому снижается комбинаторная сложность тестирования комплексов из отлаженных компонент, что обеспечивает полноту и надежность параллельного программирования.

Чтобы избежать повторной компиляции при отладочном согласовании выбора схемы управления с целевой архитектурой и формировании внутреннего представления многопоточных программ обычно используются методы смешанных вычислений и техника макрогенерации. Практичность таких методов обуславливается вполне конкретными, не редко диктуемы-

ми аппаратурой, требованиями к фрагментам наполнения, связанными с целесообразностью достижения конечности отладки фрагментов, результаты которой можно показать на регулярно генерируемом тесте. (Целостность действий, однократность присваиваний, одномерные вектора, функции без рекурсии, циклы со статически определенной кратностью, потоки действий, выполняемых по готовности данных – как арифметические выражения.)

16.1. Параллельное программирование. APL

В 1962 году К.Е. Айверсон, автор первого языка параллельного программирования APL, самим названием «A Programming Language» подчеркнул тот факт, что настоящее программирование – это параллельное программирование. Нередко переход от последовательного к параллельному алгоритму, подобный переходу от планиметрии к стереометрии, сопровождается радикальным пересмотром решений как относительно структур данных, так и относительно методов управления вычислениями. Поэтому спецификация абстрактных схем управления должна быть приспособлена к согласованной реорганизации дисциплины доступа к структурам данных и выбора методов управления вычислениями.

Параллельное программирование на языке APL можно рассматривать как укрупнение единиц обработки − распространение методов стандартного программирования со скалярных значений на векторы произвольной размерности [22]. Каждая команда может локально изменять не просто значение, а целый вектор. Синтаксис языка APL создан независимо, без оглядки на традиции представления программ и алгоритмов. Реализационная семантика языка APL сложилась как композиция из основных семантических систем вычислений, структуризации данных и управления парадигмы стандартного программирования, распространенных с обработки скаляров на обработку однородных векторов произвольной размерности. Основное отличие – организация памяти. Память функционирует в стиле без побочных эффектов (value-oriented), характерном для парадигмы функционального программирования. Спецификой реализации памяти для языка APL является использование паспортов данных, хранимых независимо от собственно блоков данных, и допускающих программную обработку.

Текст переводится в код, обрабатывающий вектора с паспортами.

Используются следующие виды команд:

- создание вектора процессоров,
- нумерация К процессоров,

– засылка константы на все процессоры,
– рассылка элементов вектора по процессорам,
– загрузка команды/функции на все процессоры,
– размещение контекста на всех процессорах.

СП = (Текст → {Код | Адрес}): Пам [Переменная] → Пам

Таблица 1

Конкретизация понятий в языке APL

Понятие	APL
Атом (Элементарное)	Имя, |скаляр|
Структура	Вектор
Переменная	Ид → Адр (Знач), []
Значение	Адрес <Паспорт, последовательность>
Выражение	Унарные, Бинарные, C := A + B
Действие/Операция	Арифметика, Выборки, = обработка паспортов
Условие/истина	0
Функция/процедура	
Аргумент	Локальные переменные на стеке
Вызов Фн/подпр	
Определение Фн/подпр	
Идентификатор	Адрес

Фрагмент	*Пояснение*
▼ r ← SUM v; sp	Объявлена новая операция
[1] vsp ← (0 <, v)/, v	Внешняя переменная
[2] sp ← +/ vsp	Локальная переменная
[3] r ← sp ÷ +/ 0<, v	Формирование выдаваемого результата
[4] ▼	

Пример 1. Определение операции

16.2. Теоретико-множественное программирование. Setl

Декларативное программирование на теоретико-множественном языке Setl сводит информационную обработку к автоматизации создания (развертывания) структур данных, над которыми все представляющие программу выражения становятся истинными высказываниями [44]. При создании языка Setl в качестве базового ЯВУ был применен язык Fortran, конструкции которого определили вид семантических систем вычислений и управления процессами, с тем изменением, что основная структура данных − не векторы, а множества, причем предельно приближенные к математической традиции классического понимания множеств. Семантика эффективной работы с памятью на уровне указателей (pointer-oriented) сопряжена с весьма абстрактной логической схемой выбора структур данных при размещении значений в памяти в зависимости от результатов анализа программы и контекста ее исполнения с использованием перебора вариантов, что сближает реализационную семантику работы с памятью с парадигмой логического программирования. Параллелизм программируется как обработка элементов множеств.

Реализационные варианты СД в языке Setl для множеств предусматривают различия в методах хранения и обработки кортежей, отображений, формирователей множеств над базовыми множествами, хэш-таблиц с разными расстановочными функциями, не полностью вычисленных множеств. При этом обеспечивается профилактика потерь памяти. Программы обработки множеств не зависят от разных методов их представления в памяти. Выразительная сила языка достаточна для его применения в качестве теоретико-множественной спецификации программ.

SETL – язык сверхвысокого уровня, представляет собой попытку активного использования теоретико-множественных понятий в практике программирования.

Согласно концепции этого языка, понятие «функция» обладает двойственной природой. Функция может быть представлена в алгоритмическом стиле – определением процедуры, выполнение которой сопоставляет результат допустимому аргументу. Но столь же правомерно представление функции в виде графика, отображающего аргументы в результаты. Оба представления могут существовать одновременно – это всего лишь две реализации одной функции. Графическое понимание функции включает в себя и табличную реализацию подобно математическим таблицам Брадиса. Кроме того график функции не обязан быть линией – это может быть фигура произвольных очертаний. Следовательно, аргументу может соответст-

вовать множество результатов, лежащих на пересечении вертикали с этой фигурой – графиком функции. При такой трактовке нет ничего удивительного в постепенном накоплении или построении графика функции. Можно задать небольшое множество точек графика, а потом постепенно его пополнять. По замыслу Дж.Шварца, автора языка ***SETL***, такая методика может выполнять роль оптимизации особо сложных вычислений.

Более формальный макет может быть построен из спецификаций функций в виде типовых выражений, задающих описание типов аргументов и результатов. Такой макет может работать как "заглушка" для нереализованных компонентов. Вместо них может работать универсальная функция, проверяющая соответствие фактических аргументов предписанному типу данных и вырабатывающая в качестве результата произвольное данное, соответствующее описанию результата. Этот механизм будет более эффективен в паре с простым макетом из тестов, если результат выбирать из коллекции тестов.

Используются следующие виды команд:

- табличные и табулируемые функции,
- pointer-oriented представление данных,
- выбор произвольного элемента,
- циклы – пространства итераций,
- формирователи множеству,
- каноническое представление и 17 структур данных,
- кванторы: for each, any=kill, every, the only,
- таблицы решений.

Таблица 2

Конкретизация понятий в языке Setl

Понятие	Setl
Атом (Элементарное)	Имя, Число
Структура	{;} {|,,,} [,,] <,,,>
Переменная	Имя
Значение	{} скаляр
Выражение	Формирователь, Кванторы, Операторы

Действие/Операция	= + элемент и теор-множ
Условие/истина	
Функция/ Подпрограмма	Подпрограмма, Таблица
Аргумент	На стеке
Вызов Фн/подпр	
Определение Фн/подпр	Перечисление, Как Fortran, Перегрузка (приоритеты)
Идентификатор	

Фрагмент	*Пояснение*
IF C1, 'T T' C2, 'T F' THEN (S1; 'X ' S2; ' X')	Шкалы истинности условий Переходы в зависимости от истинности условий

Пример 2. «if c1 then if c2 then s1 else s2» с помощью таблицы условий на языке SEL (версия Setl)

16.3. Высокопроизводительное программирование. БАРС

Программирование на языке БАРС нацелено на обеспечение высокопроизводительных вычислений и организацию асинхронных параллельных процессов [17]. При создании языка БАРС в качестве базового ЯВУ был применен популярный язык Pascal, в те годы перераставший из учебного в производственный язык системного программирования. При сохранении основных принципов семантики вычислений были существенно обобщены средства структуризации данных на основе понятия «комплекс», приспособленного к именованию элементов структур данных и учета кратности их использования. Работа с именованной памятью (Name-oriented) дополнена возможностью задавать дисциплину доступа к элементам памяти в терминах их создания-уничтожения и обработки.

Радикальное продвижение в повышении уровня программирования, предложенное в языке БАРС, заключается в переносе идеи типизации данных на проблему типизации схем управления. Процесс обработки данных рассматривается как распределенная система, находящаяся под сетевым управлением. Узлы такой системы могут сработать в зависимости от усло-

вий готовности разной природы: доступность ресурсов, сигналы монитора, внутри сетевые отношения, иерархия сетей, правила функционирования разносортных подсетей. Вычисления как и в языке APL распространяются со скаляров на сложные структуры.

В целом определение языка БАРС представлено как семейство основных подъязыков, соответствующих основным семантическим системам[2], обеспечивающим программирование схем управления в виде сетей, объявление дисциплины доступа к памяти, наложение на процессы вычисления произвольных схем управления и дисциплины доступа к памяти, что позволяет рассматривать БАРС как язык представления распределенных информационных систем.

Основные подъязыки:

- дисциплина доступа к памяти,
- управление вычислениями – синхросети с барьерами,
- просачивание операций над векторами,
- выражения над комплексами.

Таблица 3

Конкретизация понятий в языке БАРС

Понятие	БАРС
Атом (Элементарное)	Имя, скаляр
Структура	комплекс
Переменная	
Значение	
Выражение	алгебра
Действие/Операция	Над скалярами и комплексами, Отношения над сетями
Условие/истина	
Функция/ подпрограмма / модуль	Схема управления, Дисциплина доступа
Аргумент	
Вызов Фн/подпр	синхронизация
Определение Фн/подпр	Именованы переход, наполненный определением
Идентификатор	Имя перехода

[2] Понятие «основные семантические системы» и их состав были предложены В.Е. Котовым при разработке языка БАРС [74].

Фрагмент	*Пояснение*
Модуль УРАВНЯТЬ Если ≠ (↑AO, ↑BO, ↑CO) Вход A = AO, B = BO, C = CO Выход AO = A, BO = B, CO = C … определение .. … строение … Конец	Увязка внешних и внутренних устройств

Пример 3. Обмен данными.

16.4. Параллельное функциональное программирование. Sisal

Здесь мы рассмотрим один из довольно известных – функциональный язык параллельного программирования ***SISAL*** [13,39].

Название языка расшифровывается как “Streams and Iterations in a Single Assignment Language”, сам он представляет собой дальнейшее развития языка *VAL*, известного в середине 70-х годов. Среди целей разработки языка *SISAL* следует отметить наиболее характерные, связанные с функциональным стилем программирования:

- создание универсального функционального языка;
- разработка техники оптимизации для высокоэффективных параллельных программ;
- достижение эффективности исполнения, сравнимой с императивными языками типа Fortran и C;
- внедрение функционального стиля программирования для больших научных программ.

Эти цели создателей языка *SISAL* подтверждают, что функциональные языки способствуют разработке корректных параллельных программ. Одна из причин заключается в том, что функциональные программы свободны от побочных эффектов и ошибок, зависящих от реального времени. Это существенно снижает сложность отладки. Результаты переносимы на разные архитектуры, операционные системы или инструментальное окружение. В отличие от императивных языков, функциональные языки уменьшают нагрузку на кодирование, в них проще анализировать информационные потоки и схемы управления. Легко создать функциональную программу, которая безусловно является параллельной, если ее можно писать, освободившись от большинства сложностей параллельного программирования, связанных с выражением частичных отношений порядка между отдельными операциями уровня аппаратуры. Пользователь языка *SISAL* по-

лучает возможность сконцентрироваться на конструировании алгоритмов и раз работке программ в терминах крупноблочных и регулярно организованных построений, опираясь на естественный параллелизм уровня постановки задачи.

Параллельное программирование на языке Sisal опирается на парадигму функционального программирования [38]. Но замысел языка нацелен на создание конкуренции вечно живому языку Fortran и, кроме того, в качестве базового языка был использован язык VAL, в свою очередь многое унаследовавший от языка Pascal. От языка Fortran унаследован ряд идей по обработке и представлению векторов.

Система вычислений в языке Sisal использует понятие «мультизначение», позволяющее подобно языку APL распространять скалярные действия на данные любой структуры, а их обработку осуществлять на многопроцессорных конфигурациях. Работа с именованной памятью (Name-oriented) освобождена от проблем с побочными эффектами с помощью локализации участков с однократными присваиваниями – SSA-форм, что делает программы удобными для преобразований, оптимизации и компиляции, включая распараллеливание и масштабирование. Отображение значений при информационной обработке рассматривается как исполняемое на многопроцессорных конфигурациях. Результаты отображения могут повергнуться свертке или фильтрации. Формирование конфигурации управляется представлением пространства итераций и учетом зависимостей между одноуровневыми итерациями. Программа строится из участков с однократными присваиваниями, что упрощает технику оптимизационных и распараллеливающих оптимизаций.

Основное продвижение по технике программирования – развитие структуры циклов для их реализации на параллельных архитектурах. Введено понятие «пространство итераций» и предложена специальная конструкция для фильтрации мультизначений, получаемых при совмещенном исполнении итераций и участков повторяемости.

Основные виды команд:

- обработка потоков (очередь =стек= список);
- контроль однократности присваиваний (SSA-формы);
- дополнение цикла участком «returns» для оформления значения распараллеленного цикла;
- формирование пространство итераций;
- канальный обмен между итерациями;
- операции по упаковке, свёртке или фильтрации серийных значений.

Таблица 4

Конкретизация понятий в универсальном языке программирования Sisal

Понятие	Sisal
Атом (Элементарное)	Имя, скаляр
Структура	Потоки множества
Переменная	
Значение	Мультизначения - арности
Выражение	SSA
Действие/Операция	Просачивания Потоки фильтры
Условие/истина	
Функция/ Подпрограмма/ фильтр	Пространство итераций
Аргумент	Локалы Мультизначение после цикла для фильтра
Вызов Фн/подпр	
Определение Фн/подпр	
Идентификатор	Адрес значения, Вектор, Имя тега, Тип и др.

Фрагмент	*Пояснение*

For
Approx := 1.0;
Sign := 1.0;
Denom := 1.0;
i := 1
 	
while i <= Cycles do Sign := -Sign;
Denom := Denom + 2.0;
Approx := Approx + Sign / Denom;
i := i + 1
 	
returns Approx * 4.0
end for
	% инициирование цикла
	
	% предусловие завершения цикла
	% однократные
	% присваивания
	% образуют
	% тело цикла
	
	% выбор и вычисление результата цикла

Пример 4. Вычисление числа π (пи)

Фрагмент	*Пояснение*
for i in [1..Cycles/2] do val := 1.0/real(4*i-3) – 1.0/real(4*i-1); returns sum(val) end for * 4.0	% пространство параллельно исполнимых итераций % тело цикла, для каждого i исполняется независимо % выбор и свертка результатов всех итераций цикла % вычисление результата выражения

Пример 5. Это выражение также вычисляет число π (пи). Это выражение вычисляет сумму всех вычисленных значений `val` и умножает результат на `4.0`

Фрагмент	*Пояснение*

for i in [1..2] dot j in [3..4] do
returns product (i+j)
end for
	% для пар индексов [1,3] и [2,4]
	% произведение сумм
	% = 24

Пример 6. В for-выражениях операции `dot` и `cross` могут порождать пары индексов при формировании пространства итерирования

Фрагмент	*Пояснение*

for i in [1..2] cross j in [3..4] do
returns product (i+j)
end for
	% для пар [1,3], [1,4], [2,3] и [2,4]
	% произведение сумм
	% = 600

Пример 7. В for-выражениях операции `dot` и `cross` могут порождать пары индексов при формировании пространства итерирования

Фрагмент	*Пояснение*

for
I := 1
while I < S do
K := I;
I := old I + 2;
J := K + I;
returns product(I+J)
end for
	% значение из предыдущей итерации

Пример 8. Итеративное for-выражение с обменом данными между итерациями

Как это свойственно языкам функционального программирования, *SISAL* язык математически правильный – функции отображают аргументы в результаты без побочных эффектов, и программа строится как выражение, вырабатывающее значение. Наиболее интересна форма параллельного цикла. Она выделяет три части: *for* - генератор пространства итераций*, do* - тело цикла и *returns* - формирователь возвращаемых значений.

SISAL-программа представляет собой набор функций, допускающих частичное применение, т.е. вычисление при неполном наборе аргументов. В таком случае по исходному определению функции строятся его проекции, зависящие от остальных аргументов, что позволяет оперативно использовать эффекты смешанных вычислений и определять специальные оптимизации программ, связанные с разнообразием используемых конструкций и реализационных вариантов *параллельных вычислений*.

Фрагмент	*Пояснение*
function Sum (N); result (+ (sqw (1 .. N)));	% Сумма квадратов

Пример 9. Сумма квадратов

Для ЯСВУ характерна яркая специфика, связанная с поиском новых средств и методов программирования. Такая специфика может стать основой новой парадигмы.

Ряд ЯВУ, такие как Fortran, Lisp, Algol, Apl, Pascal, используются как базовые при создании новых ЯВУ и ЯСВУ, что позволяет формулировать относительные парадигматические характеристики в лаконичной и легко воспринимаемой форме.

17. МНОГОПОТОЧНОСТЬ И МНОГОПРОЦЕССНОСТЬ

Результаты анализа сложившейся практики системной поддержки конструктивных построений, гарантирующих правильность программ при их реорганизации, показывают, что как правило, все сводится к комбинаторике многократно используемых компонент, таких как структуры данных, функции, модули, классы объектов, что снижает сложность отладки программ. Характерна факторизация программ исходя из определения структур данных и проявления подобия программ обработки данных форматам структур данных. Языки и системы функционального программирования допускает

факторизацию более общего вида. Они позволяют выделять такие компоненты как схемы управления, что дает возможность типизации управления, действий, моделей вычисления, реорганизации памяти, и т. д. При параллельном программировании возникает потребность в такой факторизации относительно схем управления и дисциплины доступа к памяти.

Тем не менее, в докладах на конференциях по методам компиляции пока представляют отдельные, ранее сложившиеся средства реализации языков и систем программирования, лишь адаптированные к первичному переносу в мир параллельного программирования. В их числе – «компиляция на лету», выделение чисто функциональных подмножеств и форм с однократным присваиванием, обратимая компиляция и интерпретация, транзакционная память, анализ достижимости действий, преобразования циклов над большими массивами и т. д.

Кроме того, отметим проблему расширяемости языков и систем программирования по мере развития средств и методов параллельных вычислений, обусловленную высоким темпом прогресса в области элементной базы и информационных технологий в целом [16,24].

Основные понятия языка программирования – схемы управления программой, образующие программу действия, вычисления и данные – при переходе к параллельному программированию претерпевают изменения, и возникает необходимость в дополнительных понятиях.

Основные методы представления вычислений связаны с использованием неявных циклов, позволяющих избежать выписывания однотипных схем над стандартными структурами данных типа многомерных векторов. Так, например, результат операции над скалярами может быть распространен на произвольные однородные структуры данных. Список операций, допускающих распространение, определен реализацией. Обычно это арифметические операции (+ - * /).

Из бинарных отношений можно конструировать фильтры. Результат фильтрации исчезает из аргумента – он переносится в другую структуру данных или сохраняется как значение. Структура из фильтров дает структуру из результатов их применения к одному и тому же аргументу.

Результативность вычислений можно повышать с помощью специально устроенных данных – так называемых «рецептов» – это отложенное исполнение фрагмента программы, его замыкание, т. е. пара из действия и контекста. Барьеры в рецептах не действуют.

17.1. Трансформационная семантика

Трансформационная семантика обеспечивает сведение конструкций языка программирования к его базовым средствам, что позволяет упростить операционную семантику, а также выбрать реализационное ядро системы программирования при его экспериментальной раскрутке. Похожая техника применяется при сведении грамматик языка к форме, удобной для автоматизации построения анализатора, и при оптимизации программ [14].

Направление преобразований программ обычно связано с определенными критериями применимости и оптимальности, учитывающими результаты анализа логических и информационных связей. При организации параллельных процессов такие критерии обладают спецификой, отражающей особенности эксплуатации многоядерных архитектур. В частности, возрастает роль учета скоростей доступа к разнородной памяти и статического планирования загрузки процессоров наряду с обеспечением обратимости обработки данных и динамического управления производительностью вычислений. Поддержка такой семантики вычислений выходит за границы традиционных решений по реализации языков высокого уровня. Другой пример – локализация многократных вычислений, обычно дающая оптимизацию, в многопоточной программе может дать потери времени на ожидание обмена данными со смежными потоками.

Задача трансформационной семантики – сведение программы к нормализованной форме, удобной для интерпретации программ или генерации масштабируемого исполнимого кода [1, 41]. В случае многопоточных программ преобразование сети потоков может быть нацелено на сведение к однородной системе потоков, однозначно отображаемых на заданный комплекс процессоров – размещение потоков по процессам или назначение процессоров для выполнения потоков.

Такую схему можно выразить формулой:

[(b0: ; b1: ; … bK:), p1!(b0: d0i; …), … pN!(b0: d0j; ...)],

где i и j обозначают принадлежность потоку.

, – параллельное или одновременное исполнение,

; – последовательное исполнение,

! – назначение процессора,

b0:, b1:, … – барьеры,

p1, pN, … – процессоры,

d0i, d0j, … – действия,
(b0:; b1:; … bK:) – шкала событий/барьеров,
pM!(b0: d0i; …) – программа (последовательность) действий для процессора pM.

В такой, как бы «причесанной», форме все потоки начинаются с барьеров, и общая шкала событий упорядочена так, что последовательность событий потока ей не противоречит. Можно считать, что процессоры включаются сами. Шкала событий содержит списки ожидающих потоков. Действия, выполняемые процессорами, соотнесены с их исходными потоками.

Достаточно простые преобразования сети потоков позволяют варьировать схемы потоков и многие конструкции языка программирования сводить к взаимодействию простых потоков:

(A;B) ↔ (a:A; b:B) [3]– Расстановка – стирание барьеров.

(a:A; b:B) ↔ [[a:A, b:B], (a: ; b:)] – вынесение последовательного управления в отдельный поток – восстановление последовательности действий..

(a: A; b: B) ↔ [(a: A; b:), (b: B)] – разрез последовательности – перенос «хвоста», если нет локальной информационной зависимости между A и B.

(a: A; b: B) ↔ [a: A, b: B] – разбиение последовательности на потоки – сплющивание линии в слой, если A и B информационно не связаны.

[a: A, b: B] ↔ { (a: A; b: B) | (b: B; a: A)} – слияние потоков в одну последовательность – вытягивание слоя в линию. Выбор варианта требует учета последовательности барьеров в других потоках и общей шкале событий. Критерий оптимальности – объем выполнимых вычислений.

((a: A; b: B) ; c: C) ↔ (a: A; (b: B ; c: C)) ↔ (a: A; b: B ; c: C)

[[a: A, b: B] , c: C] ↔ [a: A, [b: B , c: C]] ↔ [a: A, b: B, c: C]

([A,B];C) ↔ [(A;C), (B;C)] – исключение слияний – слияние совпадающих продолжений.[4]

[(A;B),C]↔ ([A,C]; [B,C]) – распределение параллельного потока – слияние совпадающих потоков

[[a: A], [a: B]]↔ [a: [A, B]] – варьирование числа одновременных потоков.

Обратимость преобразований и чувствительность их результата к информационным связям между фрагментами программы требуют формали-

[3] Здесь и далее в формулах малыми латинскими буквами обозначены барьеры, а большими – действия.

[4] потеря эффективности и влияние информационной связности потока могут быть устранены реализацией в стиле так называемых «рецептов», принятых при организации «ленивых» вычислений (lazy evaluation).

зации критериев применимости трансформаций и выбора подходящего варианта. Можно констатировать, что выяснение информационной связанности действий B и A сводится к проверке существования контекста, в котором различны результаты программы C при изменении порядка вычислений B и A.

$$[(A; B), C] \neq [(B; A), C]$$

Для динамического анализа хода вычислений можно предложить операцию «Выполнялось?», реализуемую не как базовое средство, а сведением к пометке барьером позиции сразу вслед за проверяемым действием.

((? Выполнялось X); Y) ↔ [[X ; x:] , x: Y] – проверка, выполнялось ли действие X. Y выполнится лишь если X выполнялось

Аналогично из числа базовых средств можно вывести ветвления, циклы и вызовы функций, реализуя их средствами синхронизации потоков:

(if A then B) ↔ [(A;b:);(b:B)] – сведение обхода к синхронизации и обратно.

При отладке формируется ряд контекстов, на которых демонстрируются отдельные свойства фрагментов, из которых собирается полная программа. Это контексты для отдельных потоков, для пар синхронизованных потоков, для интегрированной из потоков программы, а кроме того контексты для удостоверения наличия-отсутствия информационных связей между фрагментами.

Возможны пользовательские преобразования схем управления процессами, что позволит не только минимизировать «ручную» оранжировку распараллеливаемых программ, но и даст основу для формирования библиотек преобразования схем программ. Теоретически такие преобразования следует сопровождать доказательствами частичной эквивалентности для понимания границ их применимости.

На этом фоне задача трансформационной семантики языка параллельного программирования – сведение программы к нормализованной форме, удобной для интерпретации программ, их распараллеливания и генерации настраиваемого объектного кода. Похожая техника применяется при сведении грамматик языка к форме, удобной для автоматизации построения анализатора, и при оптимизации программ [41].

Направление преобразований программ обычно связано с определенными критериями применимости и оптимальности, учитывающими результаты анализа логических и информационных связей. При организации параллельных процессов такие критерии обладают спецификой, отражающей

особенности эксплуатации многоядерных архитектур. В частности, возрастает роль учета времени доступа к разнородной памяти и статического планирования загрузки процессоров наряду с обеспечением обратимости обработки данных и динамического управления производительностью вычислений. Поддержка такой семантики вычислений выходит за границы традиционных решений по реализации языков высокого уровня.

При декомпозиции трансформационной семантики языка параллельного программирования возникает коллекция нормализованных функциональных моделей отдельных аспектов управления процессами. Нормализация моделей направлена на ограничение сложности их анализа, приспособленность к интеграции с другими моделями, поддержку быстрого прототипирования многопоточных программ, верификацию различных свойств программных систем и их факторизацию в процессе разработки и усовершенствования. Такие механизмы могут быть включены в систему параллельного программирования.

Необходимость согласования большого числа разноплановых факторов приводит к многослойному описанию семантики языков параллельного программирования (ЯПП), в котором разделены уровни абстрактных схем управления вычислениями и наполнения схем конкретными вычислениями. Чтобы избежать повторной компиляции при согласовании выбора схемы управления с целевой архитектурой, при формировании внутреннего представления многопоточных программ обычно используются методы смешанных вычислений и техника макрогенерации. Практичность таких методов обуславливается вполне конкретными, не редко диктуемыми аппаратурой, требованиями к фрагментам наполнения, связанными с целесообразностью достижения конечности отладки фрагментов (целостность действий, однократность присваиваний, одномерные вектора, функции без рекурсии, циклы со статически определенной кратностью, потоки действий, выполняемых по готовности данных – как арифметические выражения) [12].

17.2. Абстрактный комплекс

Операционная семантика языка программирования обычно базируется на определении абстрактной машины, которая в случае многопроцессорных конфигураций естественно становится конструкцией из абстрактных процессоров – абстрактным комплексом (АК). В основном определение команд абстрактной машины наследует решения SECD-машины, предложенные Лэндиным и описанные в книге Хэндерсона [34]. Для языка парал-

лельного программирования такой комплекс можно определить следующим образом:

АМ = <Регистры, [АП-1, … АП-n], СК>

АП-i = <Пам-i, СК-i>

Регистры = < стек результатов,
структура общего контекста,
синхро-сеть потоков/процессоров (активных и пассивных),
очередь барьеров>

СК – система команд, включающая в себя универсальные для всех процессоров команды, типичные для языков управления заданиями

Появляется пересмотр ряда понятий, а именно переход от АМ к абстрактному комплексу (АК):

АК = <P, R, B, D> , где

P – вектор процессоров,

R – список внешних результатов вычислений,

B – таблица синхронизованных потоков, ждущих достижения барьера другими потоками,

D – набор пассивных потоков.

Элементы P - вектора процессоров содержат номер активного потока, текущий результат его выполнения и собственно список его предстоящих действий (фрагментов).

P = <i, s, c>, где

i – уникальный номер потока, выполняемого на процессоре P,

s – стек текущих результатов i-го потока,

c – список действий i-го потока.

Определена частичная функция T(i), задающая ограничение времени выполнения потока P(i).

Общая система команд АК поддерживает выполнение следующих действий:

LOAD – загрузка произвольного пассивного потока

F-LOAD – загрузка заданного пассивного потока.

BAR – статическая синхронизация потоков по барьерам.

IF – фильтр по заданному предикату.

WHEN – ожидание истинности предиката.

FORK – развилка.

JOIN – слияние ветвей.

SEND – динамическая синхронизация активных потоков в стиле «рандеву».

WAIT – ожидание сообщения.

MASSAGE – получение сообщения.

KILL – принудительное отключение потока с диагностическим сообщением.

STOP – плановое завершение работы потока с формированием внешнего результата.

Нормальное завершение многопоточной программы происходит при исчерпании регистров B и D и плановом завершении всех активных потоков.

Кроме того, общее завершение работы происходит при невозможности завершить работу каких-либо потоков регистра B - взаимоблокировки.

Для простоты учебной модели здесь не рассматривается учет разнообразия категорий систем команд отдельных процессоров и видов используемой памяти с различной дисциплиной функционирования. Ради удобочитаемости общие команды АК представлены в виде системы переходов:

[i s c, ...] R B D → [i' s' c', …] R' B' D'

Изначально многопоточная программа размещена в регистре пассивных потоков и каждый процессор содержит команду LOAD.

[- - (LOAD . cc), ...] R B (i s c . D) → [i s (c . cc), …] R B D [5]

[- - (LOAD . c1), - - (LOAD . c2), …, - - (LOAD . cK)] R B [p1,p2, …, pL] → [(p1 . c1), (p2 . c2), …, (pK . cK)] R B [pK+1, …, pL] при K < L

[- - (LOAD . c1), - - (LOAD . c2), …, - - (LOAD . cK)] R B [p1,p2, …, pL] → [(p1 . c1), (p2 . c2), …, (pL. cL), - - (LOAD . cL+1), ..., - - (LOAD . cK)] R B – при K > L

Плановое завершение потока – становится общедоступным внешний результат работы потока, процессор освобождается и готов к загрузке нового потока:

[i s (stop . c), ...] R B D → [- - (LOAD . c), …] ((i s) . R) B D

Исчерпание действий потока – появляется диагностическое сообщение об опустошении потока:

[i s - , ...] R B D → [- - LOAD, …] ((i s «is_empty») . R) M+ B D

[5] Символ «-» означает пустой регистр.

Принудительное отключение потока – диагностическое сообщение о прерывании потока, при необходимости доступен его результат:
[i s (kill . c), ...] R B D → [- - (LOAD . c), …] ((i s «is_killed») . R) B D

Превышение допустимого времени активной работы потока – доступны параметры потока на случай предоставления ему дополнительного времени:
[i (T(i)=0 . s) c, ...] R B D → [- - (LOAD . c), …] ((i s «is_time_out») . R) B D

Синхронизация потоков по барьерам

Если достигнутый потоком барьер имеет вхождение в другие активные или пассивные потоки, то его процессор освобождается, а поток размещается в таблице потоков, ожидающих синхронизации по барьерам.
[i s (BAR b . c), ...] R B D → [- - LOAD, …] R B+(b . <i s c>) D при вхождении b другой поток.

Если достигнутый потоком барьер не имеет вхождения ни в один другой активный или пассивный поток, то текущий поток продолжает работу, а потоки, ждущие срабатывания этого барьера, переходят в набор пассивных потоков.
[i si (BAR b . c), ...] R (…, (b . <j sj cj>), ...) D → [i {si sj} c, …] R B (<j {si sj} cj>. D) барьер достигнут.

При отсутствии ожидающих данный барьер потоков работает текущий барьер.
[i s (BAR b . c), ...] R B D → [i s c, …] R B D

Фильтрация по заданному условию.
[i (1 . s) (IF . c), ...] R B D → [i s c, …] R B D
[i (0 . s) (IF . c), ...] R B D → [- - (LOAD . c), …] (i s . R) B D

Цикл по ожиданию заданного условия
[i (1 . s) (WHEN cc . c), ...] R B D → [i s c, …] R B D
[i (0 . s) (WHEN cc . c), ...] R B D → [- - (LOAD . c), …] R B D + <i s (cc WHEN cc . c)>

Разветвление

[i s (FORK {c1 c2} . c), - - LOAD, ...] R B D → [i s (c1 BAR g . c), h s (c2 BAR g JOIN i), …] R B D

[i s1 c, h s2 (JOIN i), ...] R B D → [i {s1 s2} c, - - LOAD, …] R B D

При наличии свободного процессора на него загружается ответвление, синхронизованное с первым потоком, где g -уникальное имя барьера, h – уникальное имя ответвлённого потока. Иначе ответвление размещается как пассивный поток.

[i s (FORK {c1 c2} . c), ...] R B D → [i s (c1 BAR g . c), …] R B (h s (c2 BAR g JOIN I) . D)

Загрузка заданного потока.

[i s (F-LOAD j . c), - - LOAD, ...] R B (j sj cj . D) → [i s (g . c), j sj (cj . g), …] R B D

При отсутствии свободного процессора текущий поток переходит в набор пассивных потоков.

[i s (F-LOAD j . c), ...] R B (j sj cj . D) → [j sj (cj . g), …] R B (D + i s (g . c))

Динамическая синхронизация потоков в стиле «рандеву» с помощью сообщений активным потокам:

[i s (SEND j A . c), …, j sj cj, ...] R B D →
[i s c, …, j sj (MESSAGE i A . cj), …] R B D

При наличии активного потока, которому передается сообщение, этот активный поток предваряется действием, извещающим о получении сообщения A и его передачей.

[j sj ((MESSAGE i A WAIT i . cj'), …] R B D →
[j sj (A . cj'), …] R B D

Если получатель сообщения содержит ожидание сообщения от данного потока, то сообщение A размещается в начале потока. Оно может быть командой.

[j sj (MESSAGE i A cj1 . cj'), …] R B D →
[j sj (cj1 MESSAGE i A . cj'), …] R B D

В противном случае действие, извещающее о получении сообщения, и голова получателя меняются местами.

Естественно, на уровне абстрактной машины поддержаны обычные бинарные операции над данными и дополнительные операции локального управления процессами, возникающие при реализации системы программирования. Определение синхронизации циклов и рекурсий может быть представлено в терминах индексируемых барьеров. Динамическая синхронизация в стиле ООП несложно выражается с помощью дополнительного регистра для реализации канала обмена сообщениями.

Наполнение многопоточной программы может развиваться независимо от схем управления вычислениями в отдельных потоках, а схемы можно реорганизовывать без дополнительной отладки наполнения. Они играют роль макетов или моделей программ и работают подобно макросам (открытая подстановка), но с контролем соответствия параметров объявленным синтаксическим типам фрагментов. В новых языках программирования, поддерживающих параллелизм, таких как императивный C# и функциональный F#, имеется возможность манипулировать структурами данных, представляющими внутренний код программы.

Выделение схемного уровня упрощает включение в схему разработки программ механизмов верификации (подобие модели или соответствие аксиомам), а на их основе возможна проверка программ на правдоподобие, логический вывод свойств, выполнение индуктивных и дедуктивных построений. Кроме того, техника отладки программ обогащается возможностью привлечения протоколов ранее выполненных вычислений и приведения программ к нормальным формам, удобным для сведения к базовым/стандартным моделям параллельных вычислений.

18. РЕАЛИЗАЦИОННАЯ ПРАГМАТИКА

Выделение в схеме компиляции параллельных программ уровня трансформационной семантики позволяет пересмотреть формат абстрактной машины. Вместо базового исполнителя команд, определяемых как переходы от одного состояния к другому, становится естественным использовать абстрактный макроассемблер, допускающий настройку на конкретную конфигурацию при исполнении программы – настраиваемую кодогенерацию.

18.1. Настраиваемая кодогенерация

Похожий механизм был предложен в форме символьного макроассемблера в качестве выходного языка системы БЕТА. В середине 1960-ых годов была создана система программирования АЛЬФА, входным языком которой был ранее разработанный под руководством А. П. Ершова АЛЬФА-язык. Система представляла собой многофазный оптимизирующий компилятор (АЛЬФА-транслятор) со входного языка в код М-20. В системе использовался ряд промежуточных, внутренних языков, разработанных специально для удобства анализа и отдельных преобразований программ. Вскоре возник замысел системы БЕТА, начинавшийся также с разработки входного языка (БЕТА-язык), но в условиях появления заметного числа новых серийных ЭВМ. Это привело к идее создания символьного макроассемблера в качестве выходного языка будущей системы БЕТА. Такой макроассемблер, получивший название языка СИГМА, был реализован Г. Г. Степановым. Язык СИГМА концептуально близок понятию абстрактной машины (АМ) Венской методики с некоторыми существенными отличиями. АМ определяется как языково-ориентированный машинно-независимый семантический базис, реализация которого выполняется отдельно для каждой машины в предположении, что трудоемкость реализации невелика. Язык СИГМА создан как языково-независимое представление программ, допускающее автоматизированную настройку на конкретную машину.

18.2. Память

На уровне первичного элементарного программирования мало заметна принципиальная разница между хранением данных в оперативной и внешней памяти. Особенности работы с внешней памятью много детальнее изучены в практике организации реляционных баз данных (БД) и применения языка запросов к базам данных SQL [10].

Характерной особенностью хранимых в БД является их соответствие некоторым физическим параметрам реальных объектов. Такие параметры относительно одного объекта можно представлять в виде строки таблицы. Существуют системы управления базами данных (СУБД), обеспечивающие создание, долговременной хранение, использование и уточнение таких таблиц.

В использовании конкретных таблиц может быть заинтересовано множество пользователей для разных сфер применения. Хранение таблиц может быть организовано на базе различных устройств и файловых систем и

доступ к ним может реализовываться с помощью различных дисциплин коммуникации. Для обеспечения эффективности использования аппаратуры и необходимого для практичности пользовательского интерфейса разработаны определённые системные решения, что суммарно приводит к трём уровням проектирования архитектуры БД.

Уровни архитектуры:

1) внутренний (организация хранения);
2) внешний (логика использования);
3) промежуточный (концепции системы).

Важным достижением реализационного похода в этом направлении является понятие нормальных форм (НФ), дающее разработчику БД чёткое руководство по прогнозу эффективности принятых им решений. Число таких форм слегка превышает десяток. Соответствие нормальной форме определяет выбор схемы реализации БД, компактность и структуру хранения данных во внешней памяти и скорость доступа к хранимым данным. Причём во многих случаях повышение компактности хранения сопровождается повышением скорости доступа, опровергая расхожее утверждение, что выигрыш в памяти влечёт потери в скорости и наоборот. Нормальные формы образуют упорядоченную последовательность – очередная наследует свойства предыдущей.

18.3. Нормальные формы

1НФ – элементы таблиц являются атомарными значениями, т.е. не предполагается использование их частей;

2НФ – все неключевые атрибуты зависят от первичного ключа, что позволяет добиться четкого расслоения связей;

3НФ – отсутствуют функциональные транзитивные зависимости атрибутов, что допускает разбиение таблиц на отдельные проекции;

4НФ – все многозначные зависимости сведены к одному атрибуту, через который может быть введена декомпозиция таблицы, что позволяет их привести к формату однородных структур данных;

5НФ – многоколонные таблицы сводятся к набору более простых проекций без потерь информации, что дает возможность реализации индексных файлов для организации быстрого доступа.

Любая долговременная и многопользовательская работа сопряжена с проблемой сохранения информационной целостности данных. В этом плане важно понятие «транзакция», означающее совокупность действий, вы-

полнение которых не допускает прерываний. В случае прерывания отменяются все частичные результаты транзакции. Такое понятие сейчас играет важную роль в параллельном программировании.

Созданный в начале 1970-х годов язык SQL фактически стал стандартным языком доступа к базам данных. Новые языки программирования включают его механизмы в качестве библиотечных модулей или интерфейсов для разработки приложений, связанных с использованием БД.

При всей простоте языка, сводящего работу с БД к описанию маршрута доступа к полям записей, хранимых в известных файлах, и команд манипулирования ими разными категориями пользователей, реализационно он оказался достаточно сложным из-за повышенных требований к надежности и безопасности обработки информации в многопроцессорных и сетевых неоднородных конфигурациях в многопользовательском режиме. Часть проблем решается распределением прав доступа между разными категориями пользователей, механизмами авторизации доступа, резервного хранения данных и поддержки журнала воздействий на БД. Эксплуатация важных БД обычно сопровождается разработкой специальных механизмов стратегического дублирования и восстановления хранимых данных.

Инструментарий БД активно используется в современных клиент-серверных системах при организации обработки больших и динамичных структур данных, используемых в географически распределённой работе, что используется и при организации процесса разработки программ.

Подобно тому, как объявление нормальных форм БД освободило разработчиков СУБД от обязанности поддерживать эффективное хранение бездумно организованных данных, определение нормальных форма программ, нацеленных на параллельные вычисления, может дать разработчикам систем программирования подход к накоплению средств и методов организации грамотно и корректно сконструированных программ, допускающих выполнение параллельных процессов на многопроцессорных конфигурациях разной природы.

18.4. Распределенные программные системы

Появление сетевых телекоммуникаций и развитие Интернета резко пошатнуло программистский постулат о неизменности работающей программы и в защищённости обстановки её функционирования. В дополнение к сравнительно ясным критериям правильности, эффективности и компактности программы стали всё большее значение иметь противоречивые критерии надежности и безопасности программ, влекущие ряд специальных,

не всегда очевидных, технических требований к процессу разработки распределенных программных систем, функционирующих на многопроцессорных комплексах [22, 32].

Надежность:

- совместное использование ресурсов разными программами;
- открытость и расширяемость программы, включая добавление новых функций;
- параллельные, возможно взаимодействующие, процессы выполнения подзадач;
- масштабируемость, допускающая наращивание технических мощностей;
- отказоустойчивость, включая сценарии частичного функционирования при локальных сбоях оборудования;
- дозируемая прозрачность, дающая принципиальный доступ к данным без вмешательства в физическое распределение ресурсов.

Безопасность:

- управление производительностью на основе анализа параметров сети;
- защита необходимых коммуникаций от стороннего доступа;
- работоспособность используемых или разрабатываемых сервисов;
- выбор правильной архитектуры.

Обеспечение таких критериев при разработке программ обеспечивается разного рода компонентными технологиями, аккумулирующими отлаженные решения, требующие повышенного профессионализма. Наиболее известные из них – это COM/DCOM, Corba и .Net, которые предоставляют разработчикам программ подобие многоразовых полуфабрикатов для быстрого изготовления программных продуктов.

Успех ООП в практике разработки бизнес-приложений программ привёл к переносу концепций ООП на технологию разработки компиляторов при реализации ЯП, воплощенный в проекте **.Net.**

- библиотека классов для реализации ЯП;
- рабочая среда разработки компилятора;
- промежуточный язык для создания динамической поддержки объектного кода;
- пространства имён для встраивания программы в разные обстановки;

– совместное хранение компонентов и улучшений;
– использование потенциала библиотек, созданных на разных ЯП;
– XML как один из внутренних языков;
– механизмы частичной и динамической компиляции;
– поддержка процесса сборки программ;
– апробация на многих ЯП и разных парадигмах;
– взаимодействие с COM/DCOM, Windows, Web и Intrnet-сервисами.

При экспериментах с **.**Net-технологией выполнена реализация многих известных языков программирования и специально разработаны новые мульти-парадигматические ЯП, такие как C#, F#, позволяющих совмещать преимущества разных парадигм программирования. Кроме того, преодолён стереотип одноязыкового программирования, т.к. обеспечен доступ к библиотекам, созданным на других языках.

Появляется вероятность снижения трудоёмкости отладки совмещением интерпретаторов и компиляторов в рамках одного инструментального окружение, как это было в реализации Lisp 1.5 [42], что отчасти смягчит расходы на повторную компиляцию при локальных изменениях ОО-программ.

ЗАКЛЮЧЕНИЕ

Путь к автоматизации параллельного программирования лежит через создание системы конструирования и отладки языково-зависимых библиотек преобразований программ, компиляция которых поддержана использованием верифицированных программных компонент и настраиваемой на конкретное оборудование кодогенерацией.

Следующий шаг – разработка языка высокого уровня параллельного программирования, приспособленного к ознакомлению студентов с разнообразием моделей вычислений и с методами верификации программ, без которых надежность параллельного программирования весьма проблематична. Для нужд этого шага требуется строгая формализация семантики в трансформационном стиле. Представленный в препринте подход к выбору механизмов для системы параллельного программирования предназначен для поддержки экспериментов по разработке новых языков, ориентированных на учебно-исследовательские проекты в области создания распределенных информационных систем.

Полноценное решение проблем параллельного программирования требует создания более специализированного инструментария, некоторые ме-

ханизмы реализации которого могут быть изучены в форме экспериментальной разработки учебного языка параллельного программирования.

СПИСОК ЛИТЕРАТУРЫ

1. Айлиф Дж. Принципы построения базовой машины. − М.: Мир, 1973. − 119 с.
2. Андреева Т.А., Ануреев И.С., Бодин Е.В., Городняя Л.В., Марчук А.Г., Мурзин Ф.А., Шилов Н.В.. Компьютерные языки как форма и средство представления, порождения и анализа научных и профессиональных знаний. // Труды XV Всероссийской научно-методической конференции "Телематика-2008" − Санкт-Петербург, 2008. − с. 77-78.
3. Андрианов А.Н., Задыхайло И.Б., Мямлин А.Н., Поддерюгина Н.В., Поздняков Л.А. тенденции вразвитии супер-ЭВМ. Внешнее окружение и языковые проблемы. // Прикладная информатика. Вып 2 (7) Сб.статей под ред. В.М. Савинкова. − М.: Финансы и статистика, 1984. − с. 187-199.
4. Вегнер П. Программирование на языке Ада. − М.: Мир. 1983. − 239 с.
5. Вирт Н. От Модулы к Оберону. − // Системная информатика. Вып 1. Проблемы современого программирования. - Новосибирск: Наука. Сиб. отд-ние, 1991. − с. 63-75
6. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления.– СПб.: БХВ-Петербург, 2002. – 608 с.
7. Городняя Л.В. Вычислительные сети для описания базового языка. − В сб.: Параллельные вычислительные и программные системы. − Новосибирск, 1981. − с. 135-151.
8. Городняя Л.В. Парадигмы параллельного программирования в университетских образовательных программах и специализации //Всероссийская научная конференция "Научный сервис в сети Интернет: решение больших задач − Новороссийск-Москва, 2008. − с. 180-184.
9. Городняя Л.В., Евстигнеев В.А., Касьянов В.Н. Вопросы эффективного использования параллельных ЭВМ. // Математические модели и численные методы механики сплошных сред. − Новосибирск, 1996. − с. 221-222.
10. Грабер М. Введение в SQL. − М.: Лори, 1996. − 377 с.
11. Дейкстра Э. Дисциплина программирования. − М.: Мир, 1978. − 275 с.
12. Евстигнеев В.А. VLIW-машины: развитие архитектуры и принципов построения программного обеспечения // Системная информатика. Вып 4. Методы теоретического и системного программирования. − Новосибирск: Наука. Сиб. изд. фирма, 1995. − с. 304-333.
13. Евстигнеев В.А., Городняя Л.В., Густокашина Ю.В. Язык функционального программирования SISAL, // Интеллектуализация и качество программного обеспечения. − Новосибирск, 1994. − с. 21-42.
14. Ершов А.П. Смешанные вычисления: потенциальные приложения и проблемы исследования. // Тезисы докладов и сообщений. Всесоюзная конференция "Ме-

тоды математической логики в проблемах искусственного интеллекта и систематическое программирование", ч.2. − Вильнюс, 1980. – с. 26-55
15. Катаев Н.А.Статический анализ последовательных программ в системе автоматизированного распараллеливания САПФОР# / Новосибирск, ПАВТ 2012.
16. Катцан Г. Язык Фортран 77. – М.: Мир. 1982. – 208 с.
17. Котов В.Е. МАРС: архитектуры и языки для реализации параллелизма. // Системная информатика. Вып 1. Проблемы современого программирования. – Новосибирск: Наука. Сиб. отд-ние, 1991. – с.174-194.
18. Лавров С.С. Расширяемость языков. Подходы и практика. // Прикладная информатика. 1984. Вып. 2. – 17-23.
19. Лейнингем И. Освой самостоятельно Python. – М. Вильямс. – 444 с.
20. Лельчук Т.И., Марчук А.Г. Язык программирования Поляр: описание, использование, реализация. – Новосибирск, 1986. – 94 с.
21. Ломазова И.А. Вложенные сети Петри. – М.: Научный мир. – 207 с.
22. Лорин Г. Распределенные вычислительные системы. – М. Радио и связь, 1984. – 293 с.
23. Магариу Н.А. Язык программирования АПЛ. – М.: Радио и связь. – 96 с.
24. Пентковский В.М., Синдеев Б.П. Использование расширяемого языка высокого уровня для определения специализированных языков управления системами. // Системная информатика. Вып 4. Методы теоретического и системного программирования. – Новосибирск: Наука. Сиб. изд. фирма, 1995. – с. 93-101
25. Пеппер П., Экснер Ю., Зюдхольд М. Функциональный поход к разработке программ с развитым параллелизмом с. 334-360 Системная информатика. Вып 4. Методы теоретического и системного программирования. – Новосибирск: Наука. Сиб. изд. фирма, 1995. – 361 с.
26. Пересмотренное сообщение об АЛГОЛЕ 68 / Под ред. А. П. Ершова. – М.: Мир, 1979
27. Рогожин К. http://www.swconf.ru/Novosibirsk/ Материалы фирм Intel и Microsoft, представленные на семинаре для разработчиков
28. Руководство по языку Оккам. – Новосибирск, 1987. – 75 с.
29. Савенков К. Верификация программ на моделях. / Курс ВМК МГУ имени М.В.Ломоносова. http://savenkov.lvk.cs.msu.su/mc/lect02.pdf
30. Сошников Д. В. Функциональное программирование на языке F#. – М.: ДМК Пресс, 2011
31. Стивен Р. Палмер, Джон М.Фелсинг. Практическое руководство по функционально-ориентированной разработке ПО. – М.: Вильямс, 2002. – 299 с.
32. Таненбаум Э., ван Стеен М. Распределенные системы. Принципы и парадигмы. – СПб.: Питер, 2003. – 877 с.
33. Уоткинс Д., Хаммонд М., Эйбрамз Б. – Программирвоание на платформе .Net. – М. Вильямс, 2003. – С. 367.
34. Хендерсон П. Функциональное программирование. – М.: Мир, 1983. – 349 с.

35. Хоар Ч. Взаимодействующие последовательные процессы. – М.: Мир, 1989 – 264 с.
36. Хорстман К. Scala для нетерпеливых. – ДМК пресс, 2013. – 408 с. – 300 экз. – ISBN 978-5-94074-920-2, 978-0-321-77409-5.
37. Кей С. Хорстманн Java SE 8. Вводный курс = Java SE 8 for the Really Impatient. – М.: «Вильямс», 2014. – 208 с. – ISBN 978-5-8459-1900-7.
38. Backus J. Can programming be liberated from the von Neumann style? A functional stile and its algebra of programs. – Commun. ACM 21, 8, 1978, – p.613-641.
39. Cann D. C. SISAL 1.2: A Brief Introduction and tutorial. Preprint UCRL-MA-110620. Lawrence Livermore National Lab., Livermore – California, May, 1992. – 128 p.
40. Jens Knoop Compiler Construction. 20th International Conference, CC 2011. Held as Part of the Joint European Conferences on Theory and |Practice of Software, Lecture Notes in Computer Sciences, 6601. ETAPS 2011 Saarbrcken, Germany, March 26 – April 3, 2011. Springer. 330 p.
41. Lucas P., Lauer P., Stigleitner H. Method and Notation for the Formal Definition of Programming Languges. IBM Laboratory – Venna, TR 25.087, 1968.
42. McCarthy J. LISP 1.5 Programming Mannual. – The MIT Press., Cambridge, 1963. – 106p.
43. Ritchie D.M., Tompson K. The UNIX Time-Sharing System – Bell System Technical Journal, v.57, N 6, 1978. – pp. 1905-1929.
44. Schwartz, Jacob T., "Set Theory as a Language for Program Specification and Programming". – Courant Institute of Mathematical Sciences, New York University, 1970.
45. http://haskell.org/aboutHaskell.html. Материалы по языку Haskell.

Приложение

Обозначение	*Расшифровка*
АК	Абстрактный комплекс
БД	Базы данных
ИС	Информационная система
ИТ	Информационные технологии
КП	Компонентное программирование
НФ	Нормализованная форма
ООП	Объектно-ориентированное программирование
ОС	Операционная семантика
РИС	Распределённая информационная система
СД	Структуры данных
СП	Система программирования
СУ	Синтаксическое управление
ТП	Технология программирования
ФП	Функциональное программирование
ЭП	Эксплуатационная прагматика
ЯВУ	Язык высокого уровня
ЯП	Язык программирования
ЯСП	Язык и система программирования
ЯСВУ	Язык сверх высокого уровня
ЯПП	Язык параллельного программирования
CSP	
CCS	
FDD	Функционально-ориентированное проектирование
SSA	Однократное присваивание
XP	Экстремальное программирование

СОДЕРЖАНИЕ

[6] Части 1–3 и 5 – отдельные препринты.

Л.В. Городняя

ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

Часть 4

Параллельное программирование

Препринт
175

Рукопись поступила в редакцию 18.02.2015
Редактор Т. М. Бульонкова
Рецензент Ф.А. Мурзин

Подписано в печать 19.03.2015
Формат бумаги 60 × 84 1/16 Объем 4.2 уч.-изд.л., 4.63 п.л.
Тираж 60 экз.

Типография Оригинал-2, г. Бердск, ул. Олега Кошевого, 6, оф. 2
тел./факс: 8 (383) 328-32-38, (38341) 2-12-42, сот.: 8 913 987 77 67