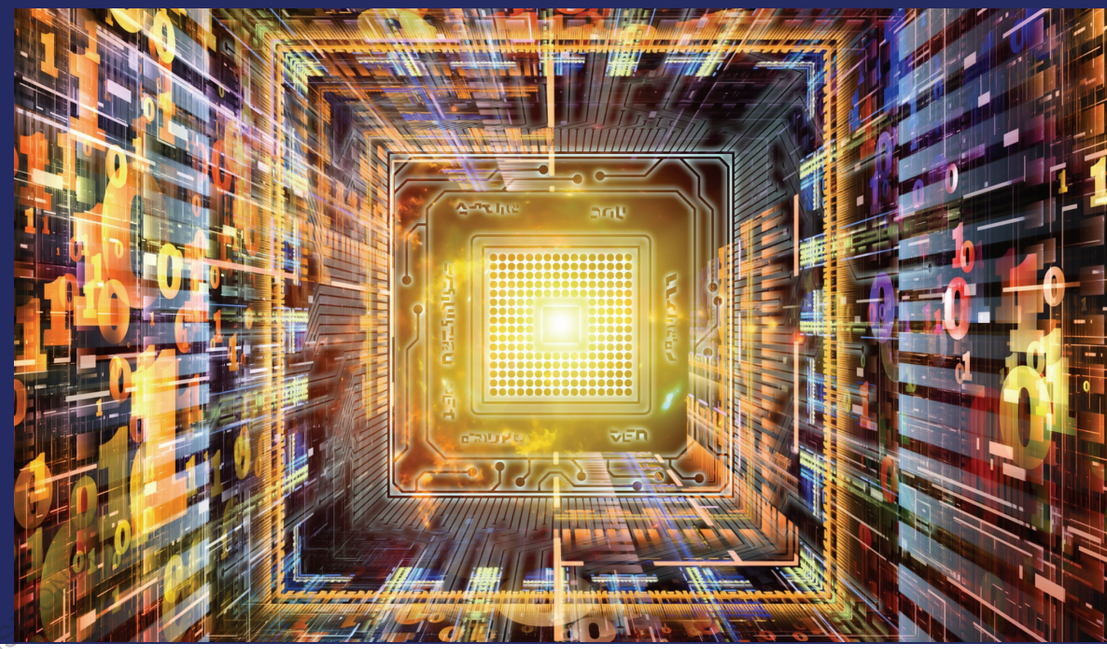
Описывается вторая версия языка классического и параллельного программирования Planning C. Язык базируется на концепции параллельных топологий, состоящих из процедур/функций с планированием повторного входа (ППВ/ФППВ), а также параллельной работы таких единичных ППВ/ФППВ. Во второй версии введены анонимные топологии и ППВ/ФППВ, построенные как расширение языка. Предложена алгебра топологий. Предложена концепция расширения языка на базе сканирующих макросов в связке с многократно и коллективно согласуемыми дедуктивными макромодулями. Введено понятие интеллектуальной мемоизации, позволяющей иногда предсказывать (с помощью нейронных сетей, МГУА и линейных экстраполяторов) отсутствующие в кэше значения. Введено понятие предицирующих каналов передачи, позволяющих иногда вычислять очередные значения, не дожидаясь их приема, что способно существенно повысить степень асинхронности программы и ускорить ее параллельное исполнение. На базе предицирующих каналов, функционирующих в условиях частично транзакционной памяти, предложена концепция сверхоптимистичных вычислений, позволившая существенно ускорить параллельную реализацию метода обратного распространения ошибки для нейронных сетей.

Пекунов Владимир Викторович родился 21 января 1977 в г.Заполярный, Мурманской обл. Учился в средней школе №4. С отличием закончил Ивановский государственный энергетический университет (1999). Доктор технических наук (2010). Член-корреспондент РАЕ (2019). В настоящее время работает в сфере новых языков параллельного программирования и моделирования.

9 786204 733258

Язык программирования Planning C v2.0

Владимир Пекунов



Язык параллельного и классического программирования Planning C v2.0

LAP LAMBERT Academic Publishing

Владимир Пекунов

Владимир Пекунов

Язык параллельного и классического программирования Planning C v2.0

Владимир Пекунов

Язык параллельного и классического программирования Planning C v2.0

LAP LAMBERT Academic Publishing RU

Imprint

Cover image: www.ingimage.com

Publisher:
LAP LAMBERT Academic Publishing
is a trademark of
Dodo Books Indian Ocean Ltd., member of the OmniScriptum S.R.L Publishing group
str. A.Russo 15, of. 61, Chisinau-2068, Republic of Moldova Europe
Printed at: see last page
ISBN: 978-620-4-73325-8

Оглавление

Введение

Невзирая на огромное количество существующих языков параллельного программирования (как специализированных, так и расширений классических языков), задача простого, адекватного и эффективного представления параллельных алгоритмов до сих пор остается актуальной. Проблемы существующих средств описания параллельных алгоритмов заметны, например, при решении на гибридных вычислительных системах сложных многостадийных задач, сочетающих переборные алгоритмы с непереборными. Примером таких задач могут быть некоторые комплексные методы из области искусственного интеллекта, такие как обучение нейронных сетей с применением различных вариаций генетического случайного поиска [28] и метода обратного распространения ошибки. Это интересное направление, весьма актуальное в связи с активным применением в настоящее время интеллектуальных алгоритмов на базе нейронных сетей для распознавания образов, принятия решений, нестандартной обработки изображений.

Упомянутые выше многостадийные (переборные или непереборные) задачи часто требуют организации параллельных процессов в соответствии с типовыми схемами «вектор», «конвейер» и «портфель задач». Соответственно, применяемый для их реализации язык параллельного программирования должен иметь эффективные и компактные способы реализации таких схем. Логично также было бы ожидать наличия в языке, помимо указанных специализированных шаблонов параллелизма, некоторых обобщенных средств описания различных, в том числе комбинированных, вычислительных топологий, которые могут иметь место при решении комплексных проблем, включающих рассматриваемые нами задачи в качестве подзадач. При этом язык, видимо, должен содержать дополнительные средства для обмена данными и синхронизации процессов. Учитывая, что задачи могут решаться на системах различной архитектуры, некоторые из таких средств должны присутствовать как минимум в универсальной и, желательно, в специальной (оптимизированной) формах.

Как будет показано далее, в обзоре некоторых, наиболее популярных и развитых языков параллельного программирования (Haskell, Java,

Go, Cilk и других) и расширений классических непараллельных языков (MPI, OpenMP, OpenCL, OpenAcc, TBB, DVM и других), вышеуказанные требования в них выполняются лишь отчасти, а в полном объеме не присутствуют ни в одном. Поэтому актуальна разработка нового языка параллельного программирования. Весьма интересным представляется также ввод в такой язык некоторых специальных стратегий, предполагающих более глубокое применение интеллектуальных логических и предицирующих технологий, которые позволили бы более эффективно (как в смысле затрат времени на разработку, так и в смысле времени исполнения конечной полученной программы) решать различные задачи.

Сформулируем более подробно некоторые базовые требования к средствам описания параллельных алгоритмов в разрабатываемом языке:

а) легкость описания, выражающаяся, прежде всего, в ограниченном количестве требуемых для распараллеливания дополнительных программных конструкций;

б) адекватность описания — применяемые средства распараллеливания не должны коренным образом изменять структуру изначального непараллельного алгоритма;

в) эффективность описания, состоящая в балансе между двумя основными стратегиями распараллеливания (построить кроссплатформенный код и построить наиболее вычислительно эффективный код), приводящая к необходимости наличия двух наборов средств: архитектурно-независимых и специализированных;

г) безопасность программирования — преимущественное применение средств, которые принципиально не создают некоторых типичных проблем (например, тупиков); четкая формализация некоторых аспектов параллельных алгоритмов, которая позволяет легко выявить некорректные ситуации; наличие высокоуровневых четко формализованных средств описания таких аспектов (например, на базе логики предикатов и некоторой достаточно элементарной алгебры);

д) адаптируемость к задаче/программисту – возможность оперативного расширения языка новыми, возможно более высокоуровневыми и удобными конструкциями, в том числе предметно-ориентированными.

Первому и второму требованию достаточно хорошо отвечает такая технология программирования, как применение процедур с планирова-

нием повторного входа. Ранее автором было показано, что данная технология позволяет кратко и весьма эффективно реализовать (как в параллельной, так и в последовательной форме) достаточно популярные алгоритмы, обычно решаемые с применением стека, дека и очереди, а также многие редукционные и стадийные алгоритмы, распараллеливаемые в соответствии с парадигмами вектора, конвейера и «портфеля задач» [36]. В данной работе будет показано, что такая технология позволяет эффективно описывать как переборные, так и непереборные задачи, а также, при внесении незначительных изменений в базовую концепцию, предоставить минимальные (необходимые и достаточные) для описания произвольных алгоритмов (в том числе использующих списки и массивы) средства.

Далее будут предложены дополнительные средства распараллеливания, в полной мере отвечающие трем последним из вышеуказанных пяти требований. Для повышения безопасности параллельного программирования ограничимся тремя основными направлениями: а) введем в язык четкие формальные (императивные и/или декларативные) средства описания виртуальных топологий, однозначно указывающие возможные направления передачи данных и диагностирующие некорректные направления; б) предложим высокоуровневую алгебраическую нотацию сокращенной записи топологий, которая также потенциально может снизить количество возможных ошибок при описании топологий; в) введем дополнительные средства из области функционального программирования, существенно упрощающие реализацию некоторых алгоритмов, повышающие читаемость и локальность кода.

Что же касается эффективности описания параллельных алгоритмов, то (для ее повышения) в качестве *архитектурно-независимых средств* распараллеливания целесообразно определить базовые средства, технически уже позволяющие записать произвольный параллельный алгоритм: а) средства для работы с виртуальной/реальной общей памятью, включая декларацию соответствующих программных объектов и применение семафоров (с помощью которых можно реализовать, в частности, блокировки и критические секции) и барьеров, б) средства для работы в парадигме, реализующей идеи передачи сообщений, а именно — каналы и виртуальные топологии. Что же касается *специализированных средств*, то в их число будут входить частные версии архитектурно-

независимых средств для различных вычислительных систем, а также элементы программирования с применением транзакционной памяти и критических секций (для систем с общей памятью), а также векторизации исполнения плана процедур с планированием повторного входа (для векторных расширителей, функционирующих, например, в соответствии с идеологией SIMT, таких как многоядерные графические видеокарты).

Адаптируемость к задаче/программисту может быть обеспечена побочным применением специальных формализмов, которые первоначально разрабатывались для облегчения разработки и ввода в язык анонимных вычислительных топологий. Такие формализмы позволяют легко выделить вводимую в язык новую конструкцию (произвольного вида и назначения, не только топологию) и ее элементы в тексте программы (с помощью какой-либо простой, стандартной технологии, например, модифицированных регулярных выражений), после чего вставляют в программу замещающие конструкцию фрагменты (здесь может быть использована технология генерации кода по типу примененной в языке PHP). Интересным применением такой технологии может быть самомодификация программы под задачу, которая может производиться некими решающими (например, логическими) метаалгоритмами, анализирующими (например, с помощью тех же модифицированных регулярных выражений) и согласующими различные части программы путем ввода и/или модификации ее фрагментов.

Возможности разработанного языка программирования будут проиллюстрированы на серии примеров (программ и/или алгоритмов), в числе которых алгоритмы обучения нейронных сетей, некоторые из которых могут быть особенно эффективно реализованы средствами Planning C. Обучение нейронной сети является задачей многомерной глобальной оптимизации, которая решается детерминированными или стохастическими методами [9]. В настоящей работе одним из примеров является фрагмент реализации стохастического метода (вариации генетического случайного поиска [ГСП]) и еще более подробно рассмотрен пример параллельной реализации одного из детерминированных методов – метода обратного распространения ошибки. Первый метод обладает рядом свойств (стадийность, наличие переборных элементов и алгоритмов подсчета элементов), позволяющих построить его эффективную параллельную реализацию на Planning C. В частности, для поддержки

стадийности можно использовать конвейер, перебор однотипных элементов реализуется, например, векторизацией исполнения плана процедуры с планированием повторного входа (узла конвейера), а алгоритмы параллельного подсчета уместно реализовать с применением транзакционной памяти. Второй метод может быть дополнительно эффективно распараллелен с помощью предлагаемой в данной работе технологии сверхоптимистичных вычислений.

Итак, *целью данной работы* является повышение эффективности программирования параллельного решения широкого круга вычислительно трудозатратных задач (многостадийных, векторных, переборных, комбинированных) на системах разной архитектуры, в том числе задач обучения глубоких нейронных сетей.

С этой целью ставятся *три основные задачи*:

1. Разработать новый язык параллельного программирования, достаточно адекватный и эффективный для решения указанных задач, основанный на идее применения процедур с планированием повторного входа.

2. Разработать некоторые новые приемы программирования, адаптированные к новому языку для наиболее эффективной алгоритмизации.

3. Предложить новую схему дополнительного распараллеливания метода обратного распространения ошибки, эффективно реализуемую на разработанном языке.

При разработке нового языка программирования воспользуемся элементами теории распараллеливания вычислений, теории формальных грамматик, теории объектно-событийных моделей [20] и основными принципами структурного и, в некоторой степени, объектно-ориентированного программирования. Также применим элементы теорий нейронных сетей и методов группового учета аргументов (МГУА).

По мнению автора, *научная новизна данной работы* заключается в следующих пунктах:

1. Предложен новый язык классического и параллельного программирования Planning C (расширение языка C/C++), отличающийся от аналогов наличием ряда конструкций, существенно упрощающих программирование ряда алгоритмов на базе массивов, линейных списков, стека, дека и очереди за счет применения идеи процедур с планированием повторного входа. Реализация данной идеи позволила создать набор

минимально необходимых средств алгоритмизации, доказаны их полнота и эквивалентность классическим основным средствам структурного программирования в современных алгоритмических языках. Основные конструкции допускают реализацию как в традиционной, так и в анонимной формах, что существенно повышает читаемость программы.

2. Предложены базовые конструкции распараллеливания в Planning C, охватывающие основные виды параллелизма зависимых и независимых подзадач на базе процедур с планированием повторного входа. В отличие от аналогов, данные конструкции позволяют четко и непротиворечиво описывать наиболее общие случаи масштабируемых виртуальных топологий, а также кратко и эффективно описывать некоторые широко употребимые частные случаи (вектор, конвейер, «портфель задач»), контролируя корректность передач данных. Применение многих базовых средств языка дает полную уверенность в отсутствии программных тупиков. Введены наборы как архитектурно-независимых (упрощающих программирование для разных архитектур), так и специализированных (позволяющих получить более эффективный код) классических средств распараллеливания для систем с общей или разделяемой памятью при опциональном наличии векторных OpenCL-расширителей.

3. Предложен декларативный подход к описанию масштабируемых виртуальных топологий на базе специальных синтаксических конструкций — макромодулей, использующих конструирующие Prolog-элементы. В отличие от аналогов, данный подход позволяет четко и непротиворечиво описывать формальные правила построения топологий, что потенциально способно снизить количество ошибок их описания. Предложенные дедуктивные макромодули обладают важным для практики побочным применением — позволяют реализовать порождающее программирование, с помощью которого, например, можно создавать настраивающиеся на особенности конкретной решаемой задачи программы. Макромодули испытаны на ряде тестовых задач построения классических топологий, показана адекватность как идеи, так и ее реализации.

4. Впервые предложен новый вид технологических средств предикции для использовании в параллельном программировании – предикционно-решающие каналы. Предложены два вида каналов – авторегрессионные точечные и линейные (явные или неявные) коллективного

решения. Разработаны соответствующие классы. Каналы отличаются от известных аналогов возможностью контролируемой предикции еще не принятых значений на основании анализа предыдущих переданных значений, что позволяет в некоторых случаях повысить асинхронность программного кода и, тем самым, поднять эффективность распараллеливания.

Кроме того применение линейных каналов коллективного решения позволяет осуществлять скрытые переходы от явных разностных схем к неявным, отличающимся большей устойчивостью счета, а также от последовательных алгоритмов счета к параллельным.

5. Введено понятие частично транзакционной памяти, отличающейся от общепринятой транзакционной памяти наличием нетранзакционных переменных и особым механизмом функционирования авторегрессионных точечных каналов в условиях согласования с откатами. Предложены соответствующие синтаксические конструкции и алгоритмы. В отличие от аналогов, применение каналов в режиме частично транзакционной памяти позволяет реализовать новый режим параллельной работы – сверхоптимистичные вычисления.

6. Предложены языковые средства и алгоритмы, поддерживающие мемоизацию процедур/функций. Впервые предложена концепция интерполирующей мемоизации, позволяющей предсказать вещественный результат мемоизации даже в случае отсутствия текущего набора аргументов в кэше. При этом используется нейронная сеть прямого распространения, линейный экстраполятор или МГУА. Сформулированы алгоритмы, позволяющие контролировать точность предикции и гибко регулировать интервалы доверия ее результатам. Показано, что для *некоторых задач из области вычислительной математики*, интерполирующая мемоизация с предиктором способна давать ускорение в 1,32÷8,8 раза. Построением доказано, что возможно применение предикторов интерполирующей мемоизации в качестве встроенных в язык обучаемых нейросетей, линейных экстраполяторов и МГУА-полиномов.

7. Сформулированы новые средства, позволяющие не только вводить в язык новые вычислительные топологии, но и строить иные произвольные расширения языка (новые конструкции, элементы аспектно-ориентированного и метапрограммирования) с применением групп регулярно-логических выражений (объединенных в сканирующие макросы –

сканеры, индентифицирующие требуемые языковые фрагменты) и, связанных с ними, многократно согласуемых дедуктивных макромодулей. Приведен синтаксис новых конструкций, доказана их алгоритмическая реализуемость на языках высокого уровня. В отличие от аналогов, данные конструкции отличаются крайней простотой и низкими затратами на разработку, поскольку не требуют кропотливой работы по внедрению синтаксиса разрабатываемых расширений в грамматику исходного языка. Это позволяет применять их, например, для быстрого прототипирования новых языковых конструкций. Сформулирован ряд идей по возможности интеллектуализации обработки программ с применением связок «сканеры-макромодули», в частности, интеллектуальной самомодификации программ и верификации реализованных алгоритмов.

8. Предложена новая общая схема вспомогательного распараллеливания циклов с зависимыми витками и внутренними зависимостями в теле, которая может быть применена при реализации некоторых математических алгоритмов, допускающих расчет с определенными погрешностями, в частности, *при обучении глубоких нейронных сетей*. Исследовано применение данной схемы с разбиением линейного фрагмента программы (тела цикла) более чем на две стадии. Предложенная схема (*сверхоптимистичные вычисления*) основана на совместном применении предсказывающих каналов и частично транзакционной памяти (такое решение является новым), опробована на задачах обучения нейронной сети методом обратного распространения ошибки, моделирования распространения тепла в тонком стержне, моделирования динамики заряженных частиц в электростатической линзе. Во всех случаях дополнительная погрешность составила не более 3,5%. При использовании в качестве основной, в задаче о распространении тепла схема распараллеливания показала максимальное ускорение около 4,5 на 8 ядрах. При использовании в качестве дополнительной, схема распараллеливания показала увеличение ускорения на 7÷46% для различных задач. Дополнительно реализована схема автоматического выбора разделения ядер системы по блокам, обрабатывающим различные стадии цикла, позволившая существенно увеличить ускорение (до 1,7 раз) по сравнению с механическим разделением множества ядер на равные части.

9. Предложена новая простая высокоуровневая алгебраическая нотация сокращенной записи виртуальных топологий, потенциально более

понятная, логичная и компактная (используются многие основные принципы классической алгебры) в сравнении с классическими алгоритмическими средствами описания топологий, ограничивающимися перечислением дуг [46, 51]. Применение предложенной нотации потенциально снизит количество ошибок при описании виртуальных топологий малой и средней сложности. Показана адекватность нотации для описания ряда нестандартных топологий.

Практическая ценность данной работы состоит в разработке простого кроссплатформенного транслятора с открытым кодом для предложенного языка программирования Planning C 2.0. Транслятор (доступен по ссылке: http://www.pekunov.byethost31.com/Progs.htm#Reenterable) написан на языке Free Pascal. Он позволяет оперативно применить предложенные в данной работе идеи программирования на практике, повысив эффективность решения ряда практически важных задач, таких как обучение глубоких нейронных сетей, на системах с общей или разделяемой памятью с возможностью использования ресурсов векторных OpenCL-совместимых расширителей (таких как современные многоядерные видеокарты).

Глава 1. Язык программирования Planning C: общая концепция и базовые средства описания непараллельных алгоритмов

Целью данной главы является повышение адекватности (компактности и эффективности) и безопасности программного описания различных непараллельных алгоритмов, подразумевающих применение стека, дека или очереди, а также возможное распространение данного подхода на программирование произвольных алгоритмов (циклических, ветвящихся, рекурсивных, использующих массивы и/или списки). Для достижения данной цели поставим следующие *задачи*: а) предложить новый, компактный и эффективный подход к описанию алгоритмов на базе стека, дека и очереди; б) предложить элементы подхода к описанию произвольных алгоритмов, доказав эквивалентность новых средств описания классическим средствам структурного программирования (управления выполнением, декларации и использования данных).

1.1. Основные рассматриваемые алгоритмы

Одним из традиционных приемов программирования является сведение задачи к решению серии схожих подзадач меньшей размерности. Во многих алгоритмах, основанных на таком подходе, для определения последовательности решения подзадач применяются типовые линейные структуры данных «дек», «очередь» и другие. Назовем некоторые алгоритмы:

а) поиска в ширину в дереве [25];

б) нумерации узлов дерева таким образом, чтобы в пределах каждого уровня нумерация была последовательной;

в) нахождения в графе цепочки сетевых подграфов;

г) поиска кратчайшего пути между электронными деталями на плате с применением волнового алгоритма Ли;

д) нумерации узлов дерева по тройкам «родитель-потомки».

Первые четыре задачи обычно решаются с помощью типовой структуры данных «очередь». Из множества возможных вариантов тако-

го решения проанализируем наиболее значимые. В простейшем случае возможен явный ввод дополнительной переменной — очереди (реализованной с применением шаблонов STL [14]). Более перспективным решением может быть разработка соответствующего паттерна проектирования [3]. Оба подхода требуют ввода и контроля целого комплекса новых классов и явного ввода переменных, что чревато возникновением элементарных синтаксических и мелких логических (отсутствие декларации, отсутствие инициализации) ошибок программирования, негативно отражается на плотности и скорости исполнения кода. Пятая задача обычно решается с помощью структуры «дек» (при обходе дерева по тройкам узлы ставятся в конец очереди, а при перенумерации узлов внутри троек — в начало).

Предлагается новый, более компактный, очевидный и перспективный для параллельного решения подход к представлению вышеупомянутого класса алгоритмов, сводимых к порождению серии схожих подзадач с планированием последовательности их решения в соответствии с одной из стратегий: очереди, стека, дека. Подход основан на оформлении алгоритма в виде специальной процедуры, к которой неявно присоединена универсальная линейная структура данных, что позволяет избежать указанных выше проблем.

Развитие данного подхода представляет как *теоретический* [возможно ли программирование *произвольных алгоритмов* (циклических, ветвящихся, рекурсивных, использующих массивы и/или списки) и какие минимальные средства для этого требуются], так и *практический интерес* [поскольку обеспечивается потенциальное сокращение объема и повышение качества проектирования кода (за счет его лучшей структурированности и минимальности некоторых конструкций)].

1.2. Процедуры с планированием повторного входа

Предлагается ввести в алгоритмические языки высокого уровня (в частности, в языки C/C++) формализм процедуры с планированием повторного входа. Такая процедура будет отличаться от обычной наличием плана исполнения, элементами которого (этапами) являются векторы значений параметров для очередного вызова процедуры.

Структурирование и специальная разметка кода в соответствии с таким подходом выносят на поверхность скрытый параллелизм задачи, позволяя учесть и наличие зависимостей в данных. В дальнейшем будет показано, что указанные особенности позволяют эффективно применить к работе процедуры идеологию распараллеливания «портфель задач». Особо будут рассмотрены случаи задач, решение которых производится за несколько стадий (таких как поиск минимального и максимального элементов в дереве) или подразумевает работу с данными, компоненты которых рассчитываются независимо, что позволяет реализовать конвейерную или векторную обработку соответственно, используя цепь (группу) процедур, запущенных в параллельном режиме.

Рассмотрим два варианта реализации плана: динамический и статический.

1.2.1. Динамический план

В данном случае любой явный (в том числе рекурсивный) вызов процедуры с повторным входом создает *новый план*, который сначала содержит один элемент (начальный этап) — вектор значений параметров, указанных при вызове процедуры. В ходе исполнения процедура может включить в план один или несколько дополнительных этапов с указанием соответствующих значений параметров процедуры. Включение в план может производиться в соответствии с одной из основных стратегий: «очередь», что соответствует вставке (планированию) в конец плана, или «стек», что соответствует вставке в начало плана[1]. Потребность в тех или иных операциях вставки определяется реализуемым алгоритмом. Аналогичным образом, можно предусмотреть операции явного извлечения этапов, соответственно, как из начала, так и из конца плана, что может быть полезно, например, при возникновении ситуации, требующей *частичной отмены* ранее спланированных этапов. При выходе из процедуры производится проверка плана: если план пуст, то осуществляется возврат в вызывающую программу, в противном случае из начала плана извлекается очередной

[1] План исполнения может быть реализован в виде очереди или стека, либо их комбинации — дека (буфера с двумя направлениями, например, кольцевого). Выбор реализации осуществляется компилятором в зависимости от операций с планом, задействованных в алгоритме.

этап, соответствующие ему значения параметров помещаются в параметры процедуры, и производится повторный вход (переход в начало процедуры).

1.2.2. Статический план

Если процедура имеет *статический (постоянный) план*, то его исполнение может быть остановлено (с выходом из процедуры) и возобновлено при следующем входе в процедуру. Возможны две модификации статического плана: а) *глобальный* (единый на всех уровнях рекурсивного вызова), б) *локальный* (индивидуальный на каждом уровне рекурсии). Есть два способа входа в процедуру:

- обычный (характерный для динамического плана), со включением нового этапа в начало плана независимо от наличия или отсутствия в нем других этапов;

- с возобновлением исполнения, при котором значения параметров, указанные при вызове, игнорируются. Параметры процедуры получают значения, соответствующие первому этапу в плане. Если план пуст, то возникает ошибочная ситуация, которую можно индицировать, например, генерацией исключения.

Применение статического плана напоминает программирование с использованием сопрограмм с той разницей, что для рассматриваемых нами процедур точкой входа *всегда* является начало процедуры. Возможна *разработка простых или рекурсивных генераторов* (разновидностей сопрограмм [10], реализованных, например, в языке Python [29]) на базе глобального или локального статического плана. Простые генераторы позволят компактно реализовать, например, стек и очередь для сложных типов данных без явного ввода дополнительных структур. Но в первую очередь статический план необходим при реализации алгоритмов, формирующих промежуточные результаты, требующие внешней обработки, по завершении которой уже запланированные работы должны быть продолжены. Сюда относятся схемы поиска с возобновлением, в частности, поиск кратчайших путей от одной исходной точки до нескольких конечных по волновому алгоритму Ли, где промежуточным результатом будет путь до очередной точки.

Весьма интересна идея генерирования и отчуждения/копирования статического плана с возможным его последующим использованием в качестве отдельной сложной структуры данных (стека/дека/очереди или массива/списка). Этот момент будет подробно рассмотрен нами далее.

1.2.3. Передача параметров. Редукция

Опишем особенности *основных правил передачи параметров* в процедуры с повторным входом. Параметры, передаваемые по значению, могут быть использованы тем же образом и по тем же правилам, что и при вызове обычных процедур. Изменения затрагивают лишь параметры, передаваемые по ссылке, позволяющие возвращать в вызывающую программу некоторое значение. Предлагается ввести возможность *редукции*[1] для таких параметров, обозначаемую наличием специального модификатора с указанием коммутативной бинарной операции/функции редукции. Результатом редукции является результат применения указанной операции/функции к множеству значений данного параметра, полученных в ходе выполнения различных этапов плана. При отсутствии необходимости в редукции предлагается организовать «тунелирование по умолчанию» последних значений таких параметров между последовательными вызовами процедуры. В некоторых случаях может иметь смысл разрыв такой цепи передачи (*отсечение*) путем явного указания значений таких параметров на очередном этапе вызова. Сформулируем простые правила:

- если при планировании этапа было затребовано отсечение, то при входе в процедуру, соответствующем данному этапу, параметр, передаваемый по ссылке, получит значение, явно указанное при планировании этапа;

- если при планировании этапа отсечения не предполагалось, то при соответствующем входе в процедуру, параметр, передаваемый по ссылке, будет иметь значение, полученное при завершении предшествующего этапа (после очередного выхода из процедуры);

[1] Редукция будет работать и при параллельном (по этапам плана) исполнении процедуры в соответствии с идеологией «портфеля задач».

- при возврате в вызывающую программу фактический параметр, переданный по ссылке, будет иметь значение, полученное на последнем этапе исполнения процедуры с повторным входом.

1.2.4. Зависимые подзадачи. Расширение рекурсивных типов данных

Отличительной особенностью рекурсивных типов данных является легкость построения стратегии их обхода, что объясняется возможностью определения общего правила для единичного элемента данных, которое может быть рекурсивно последовательно применено ко всем элементам. В данном пункте речь пойдет о рекурсивных типах с зависимостями обработки по данным. Легко видеть, что для таких типов также можно определить некоторые общие правила определения зависимости. Их выявление позволяет резко упростить программирование, в особенности эффективное параллельное программирование, требующее учета зависимостей в данных для качественного распределения задач по процессорам вычислительной системы.

В качестве примера назовем такую задачу как поиск наилучшего хода в комбинационных играх, сводящуюся к построению минимаксного дерева, в котором для выбора оптимального хода в некоем узле необходимо иметь уже рассчитанные варианты ходов для дочерних узлов. Это задача, которая может быть весьма трудоемкой при большой глубине анализа и, следовательно, весьма привлекательной для распараллеливания. Классическое решение, например, с использованием OpenMP 3.0, подразумевает организацию рекурсии с явным формированием портфеля текущих задач (используется директива task).

Такое решение получается несколько непрозрачным (с точки зрения распараллеливания) и требующим явной синхронизации, хотя, возможно и достаточно компактным с точки зрения количества операторов. Отметим, что *целью рекурсии* является разрешение зависимостей по данным, обеспечивающее корректный порядок обработки узлов «снизу-вверх». Более логичным было бы решение, при котором зависимости по данным обозначались бы декларативно, формировался бы общий список подзадач – узлов дерева, в котором система бы автоматически находила

задачи, для которых уже выполнены предусловия запуска: листовые или такие, у которых уже исполнены все дочерние узлы.

Список может формироваться в виде плана исполнения процедуры с повторным входом, где *каждому этапу плана соответствует единственная задача-узел*. Таким образом, целесообразно оформить предполагаемое нововведение как расширение для таких процедур. Это, как будет показано далее, позволит повысить гибкость решения, предлагая даже два варианта (полностью декларативный и расширенный частично декларативный) обозначения зависимостей по данным. В расширенном варианте, в частности, появляется возможность автоматизации проверок групповых зависимостей по значениям элементов полей-контейнеров данных.

1.2.4.1. Полностью декларативное обозначение зависимостей

Такое обозначение может быть выполнено для рекурсивного типа. Целесообразно ввести такую разметку этого типа, по которой однозначно восстановимо общее для всех элементов данных этого типа правило проверки возможности их запуска на обработку. При этом необходимо неким специальным образом обозначить:

а) поля-ссылки на предусловия, то есть на элементы данных того же типа, которые должны быть обработаны до текущего;

б) поле-состояние, хранящее отметку о том, был ли обработан текущий элемент;

в) значение пустой ссылки, помогающее определить независимые элементы данных, не ссылающиеся на какие-либо другие;

г) значения, определяющие обработанное (пассивное) и необработанное (активное) состояния.

Учитывая возможность неоднократного ввода разметки для решения разных задач с одним базовым типом данных, целесообразно не только разрешить разметку непосредственно в typedef-определениях типов, но и ввести typedef-подобный оператор *markupdef* переобозначения разметки, который, в отличие от typedef не подразумевает ввод нового типа данных. Тогда синтаксические инновации, касающиеся структур и уний, выглядят следующим образом:

ввод_или_разметка_рекурсивного_типа = («**typedef**» | «**markupdef**»)
[пробелы] определение [пробелы] имя_типа [пробелы] «**;**»
определение = («**struct**» | «**union**») [пробелы] [хвостовое_имя] [пробелы]
«**{**» [пробелы] {декларация_поля} [пробелы] «**}**»
декларация_поля = [[пробелы] разметка_поля] [пробелы]
тип [пробелы] имя [пробелы] «**;**»
разметка_поля = ссылка_на_предусловие | состояние
ссылка_на_предусловие = «**pre_id**»
состояние = «**state**» [пробелы] «(»[пробелы] значение_"пассивно"
[пробелы] «**,**» [пробелы] значение_"активно" [пробелы] «)»

Очевидно, что разметка поля определяет его назначение. Особо заметим, что типом поля, размеченного как ссылка на предусловие, *обязан* быть указатель на размечаемый тип.

Вышеуказанные дополнения к синтаксису позволяют определить три из четырех категорий элементов, подлежащих обозначению, а именно (а), (б) и (г). Элемент категории (в), то есть значение пустой ссылки, указывается в заголовке процедуры с повторным входом, обрабатывающей соответствующую сложную структуру, элементы которой относятся к размечаемому рекурсивному типу данных. Способ указания будет продемонстрирован далее.

1.2.4.2. Расширенное частично декларативное обозначение зависимостей

Данный вид обозначения не предполагает разметку типа. Соответственно, от значительной части декларативных утверждений приходится переходить к декларативно-императивным, которые *явно* работают с полями рекурсивного типа данных и *явно* определяются программистом, что дает большую гибкость составления кода. Синтаксис будет подробно рассмотрен далее, пока что определим *семантику* обозначений зависимостей по данным.

В частности, от декларации поля состояния и его значений переходим к указанию *операций установления set* и *reset*, соответственно, активного и пассивного значений состояния, которые могут представлять собой как элементарные присваивания вида «по-

ле_состояния = значение» так и вызовы каких-либо функций, устанавливающих необходимые значения.

Синтаксическая проблема состоит в том, что такого рода операции, также как и иные указания в данной форме обозначения зависимостей включены в заголовок процедуры с повторным входом, то есть декларируются однократно, хотя и могут применяться как к самому текущему обрабатываемому элементу данных, так и к элементам данных, доступным по ссылкам из него. Поэтому возникает необходимость некоей унифицированной ссылки на элемент данных, относительно которого операции и иные указания рассматриваются. Проблема решается введением ключевого слова *id*, которое и является такой ссылкой-подстановкой. Соответственно, операции set и reset, например, могут выглядеть так:

```
set(id->Work = 1)
reset(id->Work = 0)
```

Ссылки на предусловия обработки по-прежнему формулируются декларативно, в виде списка ссылок на соответствующие поля в утверждении *depends*. Например:

```
depends(id->Left, id->Right)
```


Отметим существование еще одной проблемы, которая эффективно решается только в данном способе обозначения зависимостей по данным. Полностью декларативное обозначение (в нашем варианте) задачу не решает. Проблема состоит, например, в том, что элемент рекурсивного типа данных может ссылаться на иные элементы не непосредственно через индивидуальные поля, а через контейнер (массив или STL-контейнеры list, vector). В таком случае утверждение depends в общем случае не может включать простое перечисление полей, а требует определения некоторой общей схемы перечисления, которой, как и в традиционном программировании может быть классический цикл. Для этого вводится механизм *энумераторов* (перечислителей) — программных префиксов перед параметризованной спецификацией поля, указывающих способ перечисления его элементов. Помимо простого перечисления энумераторы используются в групповых операциях, результат которых определяется значениями элементов группы.

Энумератор указывается в квадратных скобках перед перечисляемой конструкцией и обычно представляет собой заголовок цикла, со-

ответственно перечисляемая конструкция ссылается на счетчик цикла. Например, если ссылки на дочерние элементы хранятся в структуре Childs класса vector, то утверждение depends может выглядеть следующим образом:

```
depends ([for (int i=0; i<id->Childs.size(); i++)] id->Childs[i])
```

Заключительным элементом разметки является указание выражения, значение которого будет определять, *может ли текущая задача быть отправлена на обработку*[1]. В самом деле, если в предыдущем способе такая проверка осуществлялась автоматически, опираясь на декларации полей состояния и ссылок на предусловия, то здесь такая информация априорно отсутствует. Поэтому и появляется необходимость в соответствующем условном выражении, обозначенном классическим ключевым словом *if*. Например, для бинарного дерева проверка возможности обработки, если она в родительском узле зависит от факта обработанности дочерних, звучит «если текущий узел листовой или дочерние узлы уже обработаны» и обозначается, соответственно, так:

```
if( (!id->Left || id->Left->Work) && (!id->Right || id->Right->Work) )
```

К выражению if также может быть применен энумератор, например:

```
if( [for (int i=0; i<id->Childs.size(); i++)] (!id->Childs[i] ||
                                    id->Childs[i]->Work) )
```

В этом случае мы неявно имеем дело с групповой операцией «И».

Значение пустой ссылки указывается в заголовке процедуры с повторным входом также как и в полностью декларативном варианте обозначения зависимостей.

1.2.5. Цепи процедур с планированием повторного входа

Рассмотрим алгоритмы, имеющие несколько последовательно зависимых (стадийных) или полностью независимых (параллельных) сегментов решения, каждый из которых предполагает генерацию и испол-

[1] Проверка условия происходит, если выполнены условия предобработки для всех элементов, перечисленных в утверждении depends. Таким образом, реализуется двухэтапная проверка, которая, возможно, позволит оптимизировать процесс определения текущих задач, предназначенных для обработки, хотя и является определенного рода «синтаксическим сахаром».

нение серии подзадач. В случае стадийных алгоритмов набор подзадач одного сегмента тем или иным образом определяет множество подзадач в другом сегменте. Стадийную форму могут иметь, например, некоторые алгоритмы работы с двоичным деревом, включающие его обход и выполнение дополнительных операций. Назовем алгоритмы: а) поиска минимального и максимального элементов, б) нахождения по отдельности сумм правых и левых элементов. К алгоритмам с независимыми сегментами решения отнесем задачи обработки множества данных, каждый элемент которого включает несколько самостоятельных компонентов, например, векторов и матриц.

Пусть каждый сегмент решения реализуется отдельной процедурой с повторным входом. В случае стадийного решения этапы плана для очередного сегмента формируются (в том или ином заданном порядке) при исполнении плана предыдущего сегмента. Тогда все вышеуказанные алгоритмы, независимо от наличия зависимостей между сегментами, достаточно естественно могут быть реализованы *цепью процедур* с планированием повторного входа, в которой *каждая процедура реализует один из сегментов решения и может являться генератором плана для следующей процедуры в цепи*.

Для первой процедуры в цепи присутствие начального этапа в плане, инициирующего генерацию последующих этапов, является обязательным. Для прочих процедур такое условие отсутствует, поскольку возможна оперативная генерация начальных этапов предшествующей в цепи процедурой.

1.3. Базовый синтаксис

Пусть первоначально возможность планирования повторного входа будет существовать лишь для процедур[1] (точнее, void-функций в терминах C/C++). Предлагается ввести в язык несколько новых ключевых

[1] В дальнейшем это требование будет снято, поскольку в некоторых случаях требуется (содержательно или для поддержки минимальных средств программирования, что будет продемонстрировано далее), чтобы блок программы с планированием повторного входа возвращал значение.

слов и функций, а также расширить трактовку служебного символа «!» и ключевых слов **static** и **continue**.

1.3.1. Декларация и вызов

Полный заголовок процедуры с повторным входом имеет формат:

полный_заголовок = самостоятельная_процедура | элемент_цепи
самостоятельная_процедура = «**reenterable**» [ограничение] пробелы [«**static**» пробелы [(«**local**»|«**global**») пробелы]] заголовок [предикаты]
элемент_цепи = «**chain**» [ограничение] пробелы заголовок [[пробелы] интерфейс_элемента_цепи] [предикаты]
интерфейс_элемента_цепи = «**throw**» [пробелы] «(» [параметры] [пробелы] «)»
заголовок = имя [пробелы] «(» [параметры] «)»
ограничение = «**[**» максимальное_количество_этапов «**]**»
параметры = параметр {«**,**» параметр}
параметр = [пробелы] [идентификация_рекурсивного_типа | редукция] декларация_параметра
идентификация_рекурсивного_типа = «**id**» [пробелы] «(» [пробелы] значение_пустой_ссылки [пробелы] «)»
редукция = «**reduction**» «(» (знак_операции | имя_функции) «)»
предикаты = [пробелы] предикат {[пробелы] предикат}
предикат = операция_активизации | операция_деактивизации | зависимости_по_данным | проверка_активности
операция_активизации = «**set**» [пробелы] «(» операторы «)»
операция_деактивизации = «**reset**» [пробелы] «(» операторы «)»
операторы = присваивание | иные_операторы
присваивание = «**id**» («**.**» | «**->**») обозначение_поля [пробелы] «**=**» [пробелы] выражение
зависимости_по_данным = «**depends**» [пробелы] «(» поля «)»
поля = [пробелы] ([энумератор] «**id**» («**.**» | «**->**») обозначение_поля) | (поле_структуры {«**,**» [пробелы] «**id**» («**.**» | «**->**») обозначение_поля})
энумератор = операторы_C++
проверка_активности = «**if**» [пробелы] «(» [энумератор] логическое_выражение_относительно_id «)»

Любое из ключевых слов **reenterable** или **chain** пока свидетельствует об отсутствии возвращаемого результата (процедура) и о возможности повторного входа. Наличие **static** (или **static global**) означает применение глобального статического плана, **static local** — локального статического, отсутствие **static** — динамического. Если максимальное количество этапов не указано, то план может иметь произвольный размер. Ключевое слово **chain** подразумевает, что текущая процедура обладает динамическим планом и может являться элементом цепи. *Интерфейсом процедуры* назовем список формальных параметров, которым должна обладать следующая в цепи процедура. Возможна явная спецификация интерфейса после ключевого слова **throw**. При ее отсутствии считается, что интерфейс процедуры совпадает с ее списком формальных параметров. Наличие модификатора **reduction** с указанием знака бинарной операции или имени бинарной функции перед декларацией параметра детерминирует необходимость редукции его значений с применением данной операции/функции. Важно отметить, что в Planning C редукционному параметру начальное значение автоматически не присваивается, такая инициализация должна быть выполнена в программе *в явной форме*.

Обычный вызов процедуры с повторным входом записывается аналогично вызову любой процедуры (void-функции) C/C++. Чтобы более гибко распределять ресурсы многопроцессорной системы, особенно при работе с цепью процедур, введена возможность указания количества процессоров (переменной или константы), на котором будет производиться исполнение (по умолчанию равного единице):

вызов **=** имя_процедуры **«(»** [аргументы] **«)»** [**«/»** количество_процессоров]

Вызов с возобновлением исполнения отличается наличием ключевого слова **continue** перед именем процедуры. Способ вызова внутри процедуры идентифицируется функцией **plan_after_continue()**, которая возвращает **false** при обычном вызове и **true** при возобновлении.

1.3.2. Управление планом

Включение нового этапа в конец плана исполнения реализуется обращением к функции **plan_last**, которая существует в пределах теку-

щей процедуры с повторным входом и имеет эквивалентный список формальных параметров. Значения, указанные в списке параметров при вызове **plan_last**, станут значениями параметров процедуры на этапе, спланированном при данном вызове, с вышеуказанными оговорками относительно параметров, передаваемых по ссылке.

Включение нового этапа в начало плана реализуется обращением к функции **plan_first** с теми же областью видимости, параметрами и правилами их передачи, что и при вызове **plan_last**.

Обычно извлечение этапов из плана осуществляется автоматически (в соответствии с базовой логикой функционирования процедуры с повторным входом). Тем не менее, предусмотрена возможность оперативного извлечения этапа из начала (вызовом функции **plan_get_first**) или из конца (вызовом функции **plan_get_last**) плана. Каждая из этих функций имеет список параметров, эквивалентный списку параметров текущей процедуры с планированием повторного входа с одним существенным отличием – *все параметры передаются по ссылке* (имеют неявно присутствующий модификатор **&**). В результате вызова функции **plan_get_first/plan_get_last** вектор элементов соответствующего (начального или конечного) этапа плана помещается в ее параметры, а сам этап извлекается из плана.

Директива **clear_plan** очищает план исполнения в пределах текущей процедуры. Такая возможность может быть востребована в алгоритмах с досрочным завершением запланированных работ (например, при работе с волновым алгоритмом Ли прекращается анализ оставшихся элементов «волны», если кратчайший путь найден).

Системная переменная **plan_empty** принимает истинное значение, если план пуст.

Прекращение исполнения плана реализуется директивой **plan_stop**. Ее исполнение не приводит к немедленному выходу из процедуры, но сигнализирует, что после завершения текущего этапа плана должен произойти возврат в вызывающую программу. При возврате динамический план очищается, а статический — сохраняется, что дает возможность повторного входа в процедуру с возобновлением исполнения плана (с применением ключевого слова **continue**).

Если при планировании этапа необходим разрыв (отсечение) цепи передачи параметра по ссылке, то предлагается указывать символ «**!**»

непосредственно после значения соответствующего фактического параметра при обращении к функциям **plan_first** и **plan_last**.

1.3.3. Запуск цепи

В однопроцессорном варианте процедуры цепи запускаются последовательно, причем каждая из них работает вплоть до исчерпания собственного плана исполнения. Источником этапов плана может быть как сама процедура, так и предшествующая ей в цепи процедура. Чтобы обеспечить запуск цепи, ее первая процедура должна быть выполнена не менее одного раза, следовательно, всегда должна иметь *начальный этап* в плане исполнения. Остальным процедурам такой этап необходим лишь в некоторых случаях, например, для *общей инициализации*. Потребность в такой инициализации определяется первым параметром конструкции запуска цепи процедур с повторным входом:

«plan_chain» «(» общая_инициализация **«,»** описатель_цепи **«)»**

общая_инициализация = логическое_выражение

Формат описателя цепи предусматривает два случая, когда цепь включает последовательность вызовов: а) произвольных процедур; б) одной и той же процедуры. Во втором случае указывается формат вызова процедуры и длина (количество элементов) цепи.

описатель_цепи = (вызов {(**«,»** | **«->»**) вызов}) | (длина_цепи **«,»** вызов)

Все процедуры цепи не только должны быть декларированы с ключевым словом **chain**, но и являться совместимыми — интерфейс любой процедуры должен совпадать по количеству и типам элементов со списком формальных параметров следующей в цепи процедуры. Параметры, указанные в вызовах процедур, определяют содержание начальных (инициализационных) этапов плана исполнения соответствующих процедур, если такие этапы предусмотрены логикой запуска цепи. *Инициализационный этап всегда выполняется процедурой первым*, независимо от того, были ли уже поступления в план от предыдущей в цепи процедуры. Это гарантирует корректность инициализации процедур цепи.

Помещение нового этапа в начало плана следующей по цепи процедуры осуществляется обращением к функции **throw_first**, которая су-

ществует в пределах текущей процедуры и имеет список формальных параметров, эквивалентный интерфейсу данной процедуры. Включение нового этапа в конец плана следующей процедуры реализуется обращением к функции **throw_last** с теми же областью видимости и параметрами.

Необходимо сказать несколько слов о решении проблемы идентификации стадии конвейера[1]. Специальная функция **throw_stage()** возвращает порядковый номер стадии, которые нумеруются с нуля до N-1, где N — общее число стадий. Число N можно определить, получив результат функции **throw_num_stages**().

1.4. Реализация некоторых алгоритмов

Ранее перечислялись некоторые алгоритмы, которые могут быть реализованы с помощью очереди и дека. Более компактно, естественно и надежно эти алгоритмы могут быть записаны с применением процедур с планированием повторного входа (вставка в конец плана эквивалентна применению очереди, вставка как в начало, так и в конец эквивалентна использованию дека). Повышение надежности программирования (снижение количества потенциальных ошибок) объясняется устранением необходимости в явном вводе управляющей структуры (стека, дека, очереди), подразумевающем: а) декларацию вспомогательных типов и переменных и б) организацию вставки и циклического извлечения элементов.

Рассмотрим последовательную нумерацию узлов двоичного дерева по уровням сверху вниз и слева направо. Введем вспомогательную структуру, описывающую узел:

```
typedef struct TreeTag {
  int Data;
  struct TreeTag * Left;
  struct TreeTag * Right;
} TreeNode;
```

Пусть переменная Root — указатель на корневой элемент дерева. Полученный код имеет объем в 1,5÷2 раза меньше в сравнении с традиционной реализацией данного алгоритма с применением очереди.

---

[1] Эта проблема решается одинаково как при последовательном, так и при параллельном запуске цепи.

```
int Number;
reenterable EnumerateByLevel(TreeNode * Cur) {
 Cur->Data = Number++;
 if (Cur->Left) plan_last(Cur->Left);
 if (Cur->Right) plan_last(Cur->Right);
}
/* Вызов: Number = 1; EnumerateByLevel(Root); */
```

Рассмотрим алгоритм нумерации узлов дерева по тройкам. Воспользуемся туннелированием параметра по ссылке Number:

```
reenterable EnumerateByFamilies(TreeNode * Cur,
   char Level, int &Number) {
 Cur->Data = Number++;
 if (Level) {
  if (Cur->Left) plan_last(Cur->Left,0,Number);
  if (Cur->Right) plan_last(Cur->Right,0,Number);
 }
 else {
  if (Cur->Right) plan_first(Cur->Right,1,Number);
  if (Cur->Left) plan_first(Cur->Left,1,Number);
 }
}
/* Вызов:
 int Number = 1;
 EnumerateByFamilies(Root,0,Number); */
```


Проиллюстрируем применение статического плана, реализовав очередь[1], каждый элемент которой включает два значения (целочисленное и вещественное). При обычном обращении к процедуре queue значение помещается в очередь, а при вызове с возобновлением — извлекается.

```
reenterable static queue(int &Int, double &Dbl) {
 if (!plan_after_continue()) plan_last(Int,Dbl);
 plan_stop;
}
/* Помещение в очередь пары (A,B): queue(A,B); */
/* Извлечение пары (A,B): continue queue(A,B); */
```

Такая реализация несколько проще и потенциально дает более компактный и эффективный код по сравнению с шаблонами STL, так как не требует декларации (явной и/или неявной) новых классов и работы с объектами. Теоретически быстрее осуществляется и компиляция, поскольку не производятся разбор и интерпретация шаблонов с выводом

[1] Аналогично можно организовать стек, заменив в реализации plan_last на plan_first

классов. Полученная реализация также компактнее и по сравнению с организацией очереди как простого массива структур.

Для иллюстрации целесообразности применения цепей процедур рассмотрим алгоритм последовательной нумерации узлов двоичного дерева по уровням снизу вверх и справа налево, который реализуется в два этапа: планирования обхода элементов сверху вниз и слева направо в процедуре Rev1 с последующим формированием эквивалентного плана, но с этапами, идущими в обратном порядке, для процедуры последовательной нумерации Rev2.

```
chain Rev1(TreeNode* Cur) throw(TreeNode* Cur, int &Number)
{
 if (Cur->Right) plan_last(Cur->Right);
 if (Cur->Left) plan_last(Cur->Left);
 throw_first(Cur,1);
}
chain Rev2(TreeNode * Cur, int &Number) {
 Cur->Data = Number++;
}
/* Вызов (Root — указатель на корневой элемент дерева):
 int Number = 1;
 plan_chain(0,Rev1(Root),Rev2(Root,Number));
*/
```


1.5. Предлагаемые расширения

Перечисленные выше семантико-синтаксические конструкции были предложены автором еще в работе [20]. В языке Planning C присутствует ряд новых дополнительных конструкций, некоторые из которых (имеющих отношение к классическому последовательному программированию) и будут обсуждаться в данном пункте.

1.5.1. Функции с планированием повторного входа

Прежде всего, введем возможность реализации не только процедур, но и *функций с планированием повторного входа*. При этом заметим, что для цепи процедур понятие возвращаемого значения является, по нашему мнению, значительной условностью. Поэтому сказанное да-

лее относится только к одиночным процедурам, декларируемым с применением ключевого слова **reenterable**. Соответственно, расширяется синтаксис полного заголовка:

полный_заголовок = самостоятельный_блок | элемент_цепи
самостоятельный_блок = «**reenterable**» [ограничение] пробелы [«**static**» пробелы [(«**local**»|«**global**») пробелы]] [тип_значения пробелы] заголовок [предикаты]

Элемент тип_значения может представлять любой тип, в том числе **void** (что равносильно отсутствию типа). Возврат значения из функции с повторным входом осуществляется также, как и в обычной функции C/C++. Необходимо лишь заметить, что в вызывающую программу всегда будет возвращаться значение, указанное при обработке *последнего этапа функции*.

1.5.2. Расширенная трактовка плана: отчуждение плана. Внешнее заполнение/чтение

Как уже было сказано выше, функция или процедура с планированием повторного входа четко связана с неявно присутствующей линейной структурой данных [стек, дек или очередь, которые, в действительности реализуются либо через массив (*при указании ограничения* на максимальный размер плана в заголовке), либо через связный список (если *ограничение не указано*)]. Однако такая неявность не всегда удобна, особенно при необходимости работать с планом в целом, например, при единовременном *формировании всего плана извне*, а также при *импорте плана*, заполненного внутри процедуры/функции. Это, теоретически, возможно для случая *статического глобального плана*, который действительно является некоей глобальной переменной и, следовательно, отчуждаем от процедуры/функции (что невозможно для случая динамического плана и затруднительно для статического локального плана, который, в реализации, является *списком планов*).

В связи с вышесказанным, имеет смысл разрешить: а) непосредственный доступ к плану по имени процедуры/функции, б) декларацию переменных, имеющих те же типы, что и план и/или его элементы, в) возможность различных присваиваний с участием плана (и,

возможно, его элементов) и вышеуказанных переменных. Для реализации первой возможности вводится специальная ссылка на план процедуры/функции:

ссылка_на_план = «*****» имя_процедуры_или_функции

Вторая возможность может быть покрыта путем использования всего двух ключевых слов:

тип_плана = «**plan_type**» «(» имя_процедуры_или_функции «)»

тип_элемента_плана = «**plan_item_type**» «(»
имя_процедуры_или_функции «)»

Третья возможность частично реализуется стандартными возможностями C/C++ (возможны прямые присваивания между планом/элементом плана и переменными, имеющими соответствующие типы), однако для *поэлементной работы с планом* введен некоторый дополнительный синтаксис:

вставка_элемента_в_начало_плана = элемент «**>>**» план

вставка_элемента_в_конец_плана = план «**<<**» элемент

извлечение_элемента_из_начала_плана = элемент «**<<**» план

извлечение_элемента_из_конца_плана = план «**>>**» элемент

значение_эдемента_из_начала_плана = «**++**» план

значение_элемента_из_конца_плана = план «**++**»

значение_элемента_из_позиции_плана[1] = план «[» позиция «]»

план = переменная_типа_план | ссылка_на_план

элемент = переменная_типа_элемент_плана

позиция = числовое_выражение

При выполнении операций поэлементного доступа к плану или к переменной, имеющей тип плана, может потребоваться информация о текущем количестве элементов в соответствующей структуре, для этого используется специальная функция:

число_элементов = «**_size**» «(» план «)»

Важные замечания о сути введенных неявных типов:

а) тип «план» является *массивом* (размер которого при инициализации равен нулю), если соответствующая процедура/функция имела ограничение на количество элементов плана, указанное в ее заголовке, в

[1] Имеет смысл *только для плана процедуры/функции, являющегося массивом*, то есть соответствующего процедуре/функции с ограничением размера плана в заголовке.

противном случае тип «план» представляет *список*, доступ к элементам которого через «[]»-синтаксис невозможен;

б) тип «элемент_плана» является *структурой*, поля которой имеют типы и имена, соответствующие параметрам процедуры/функции. Доступ к этим полям выполняется по классическим правилам C/C++.

Приведем фрагмент программы, в котором декларация процедуры с планированием повторного входа введена исключительно для *неявной декларации типа плана и типа его элемента* (сама процедура даже не вызывается). В примере создается план-список, каждый элемент которого хранит одно целое число, далее он заполняется десятью целыми числами, которые выводятся на экран.

```
reenterable static External(int A) { }
int main() {
 plan_item_type(External) S;
 for (int i = 0; i < 10; i++) {
     S.A = i;
     *External<<S;
 }
 for (int i = 0; i < 10; i++) {
     S<<*External;
     cout<<S.A<<" ";
 }
 cout<<endl;
 return 0;
}
```


Тот же фрагмент может быть записан и для плана-массива.

```
reenterable[100] static External(int A) { }
int main() {
 plan_item_type(External) S;
 for (int i = 0; i < 10; i++) {
     S.A = i;
     *External<<S;
 }
 for (int i = 0; i < _size(*External); i++) {
     cout<<*External[i].A<<" ";
 }
 cout<<endl;
 return 0;
}
```

Мы обсудили возможность поэлементных внешних заполнения и чтения плана. Формально, этих возможностей уже достаточно. Однако для повышения удобства целесообразно ввести дополнительный синтак-

сис, который, в частности, позволял бы работать с планом в целом, используя массивы и размножаемые значения одиночных переменных и констант. Это позволило бы, например, напрямую создать план из массива/массивов/переменных/констант или прочитать план в массив/массивы/переменные.

При таком подходе план может рассматриваться как *таблица*, каждая *колонка* которой представляет набор значений одного из параметров соответствующей процедуры/функции (колонки следуют в том же порядке, что и параметры). Колонку можно сопоставить массиву, простой переменной, константе или пустому знаку.

При *заполнении плана*:

а) из массива — колонка заполняется от начала к концу (сверху вниз), элементы читаются последовательно (число строк равняется числу элементов самого длинного из читаемых массивов; если длина массива меньше числа строк, то в оставшиеся строки помещается значение последнего элемента);

б) из простой переменной или константы — значение заносится во все элементы колонки;

в) из пустого знака — колонка не заполняется.

При *чтении плана*:

а) в массив — элементы заполняются последовательно, колонка читается от начала к концу (сверху вниз);

б) в простую переменную — помещается значение из соответствующего элемента *первой* строки (заполненного плана);

в) в пустой знак — никаких операций не происходит.

Теперь определим *синтаксис*:

заполнение_плана = «*****» имя_процедуры_или_функции «**<<[**» заполнители «**];**»

заполнители = заполнитель {«**,**» [пробелы] заполнитель}

заполнитель = переменная | константа | пустой_знак | массив

пустой_знак = «**_**»

массив = идентификатор «**[]**» {«**[]**»}

чтение_плана = «*****» имя_процедуры_или_функции «**>>[**» читатели «**];**»

читатели = читатель {«**,**» [пробелы] читатель}

читатель = переменная | пустой_знак | массив

Здесь необходимо обратить особое внимание на тот факт, что количество указанных знаков «[]» при идентификаторе массива определяет размер читаемого/заполняемого единичного элемента. Если число таких знаков равно числу измерений массива, то единичным элементом является элемент массива. Если число знаков «[]» меньше числа измерений, то единичным элементом является более крупный блок массива (по обычным правилам C/C++, которые применяются при развертке лишь части индексов многомерного массива). При этом размер читаемого/записываемого элемента транслятор определяет с помощью вызова **sizeof**, причем в каждую пару «[]» неявно ставится ноль: «[0]». Таким образом, размер элемента ссылки-массива «A[][]» равен «**sizeof(A[0][0])**».

Еще одно важное замечание: *копирование данных в вышеуказанных операциях выполняется побайтно*.

Приведем небольшой пример, в котором заполняется массив **array**, который через план процедуры **External**, используя введенный синтаксис, копируется в массив **array2**. Содержимое плана и массива **array2** выводится на экран.


```
reenterable static External(int A, double X, int B) {
  cout<<"("<<A<<" _ "<<B<<") ";
}
int main() {
 int array[10];
 for (int i = 0; i < 10; i++)
     array[i] = i;
 *External<<[array[],_,1];
 int array2[10] = {0};
 *External>>[array2[]];
 continue External(-1, 0.0, -1); /* Вызываем External, минуя этап инициализации (указанные здесь параметры пропускаются). В следующем пункте данной работы будет рассмотрен иной, более краткий формат вызова процедуры/функции без инициализации */
 cout<<endl;
 for (int i = 0; i < 10; i++)
     cout<<array2[i]<<" ";
 cout<<endl;
 return 0;
}
```

В результате работы данного примера на экран выводятся две строки:
(0 _ 1) (1 _ 1) (2 _ 1) (3 _ 1) (4 _ 1) (5 _ 1) (6 _ 1) (7 _ 1) (8 _ 1) (9 _ 1)
0 1 2 3 4 5 6 7 8 9

1.5.3. Расширенный синтаксис вызова

В последнем примере предыдущего пункта для вызова процедуры с заполненным планом использовался оператор **continue**. Это несколько громоздкая форма записи, для которой, учитывая очевидную полезность работы с отчуждаемым (заполняемым извне) планом, семантически целесообразно ввести эквивалентную, более краткую форму:

вызов_без_инициализации = имя_процедуры_или_функции «([» фиктивные_параметры «])»

Приведем небольшой пример, в котором значения из заполненного извне плана выводит сама процедура. При этом инициализационный этап не нужен, что позволяет с успехом применить предложенный в данном пункте синтаксис вызова.

```
reenterable[100] static External(int A) {
 cout<<A<<" ";
}
int main() {
 plan_item_type(External) S;
 for (int i = 0; i < 10; i++) {
     S.A = i;
     *External<<S;
 }
 External([-1]);
 cout<<endl;
}
```


1.6. Предельные средства алгоритмизации. Связь с элементарной расширенной машиной Тьюринга

Как уже было декларировано выше, отчуждаемый план вполне может заменить основные структуры (стек, дек, очередь, список и массив), состоящие из сложных многокомпонентных элементов. Таким об-

разом, во многих случаях можно избежать: а) применения оператора **typedef** для декларации типов элементов-записей (**struct**) и собственно вышеперечисленных линейных структур; б) декларации соответствующих переменных. Это достаточно интересный факт, который побуждает рассмотреть, помимо декларационных, алгоритмические возможности процедур/функций с планированием повторного входа (ФППВ/ПППВ). Данный вопрос имеет существенное *теоретическое значение*.

Теорема об алгоритмизации на ПППВ (ТА1). Все основные управляющие конструкты (процедуры и функции, циклы и ветвление) в алгоритмическом языке могут быть реализованы с помощью ПППВ, если язык поддерживает сокращенное вычисление логических выражений.

Доказательство. ПППВ по определению способны представлять обычные *процедуры*, которые просто не используют конструкции для работы с планом исполнения, содержащий в таком случае единственный элемент — вектор значений параметров, переданных в процедуру при ее вызове. В данной работе введены ФППВ, дя которых разрешена спецификация типа возвращаемого значения и его указание в операторе возврата (значением такой ФППВ является значение, возвращенное ею после обработки последнего этапа плана). По аналогии с ПППВ, с помощью ФППВ можно представить *обычные функции*.

Ветвление вида «если <условие> то А иначе В» при наличии сокращенного вычисления логических выражений записывается (в синтаксисе C++ и дополнительных конструкций, предложенных в работе [20] и в настоящей работе) следующим образом:

```
reenterable bool procA(<параметры A>)
{ A; return true; }
reenterable bool procB(<параметры B>)
{ B; return true; }
…
<условие> && procA(<параметры A>) || procB(<параметры B>);
```

Здесь применен эффект краткого вычисления логических выражений.

Пусть *конструкции планирования в начало и конец плана исполнения являются функциями*, возвращающими истинное значение. Тогда основной *цикл* вида «пока <условие> повторять А» запишется так:

```
reenterable bool procA(<параметры A>)
{ A; return true; }
reenterable while_loop(<параметры>)
```

```
{  <условие> && plan_last(<параметры>) && procA(<параметры
A>); }
…
while_loop(<параметры>);
```

Известно, что с помощью такого цикла с предусловием можно реализовать *любые циклы*. Теорема доказана.

Следствие. Учитывая сказанное в п.1.5.2, ФППВ/ПППВ являются *достаточными предельными средствами программирования*, с помощью которых можно выразить декларацию элементарных, многоэлементных и линейных структур данных, описать переменные этих типов, реализовать произвольные алгоритмы обработки данных.

Теорема об адекватном предельном вычислителе ТАПВ1. Наиболее адекватным предельным вычислителем, способным реализовать ПППВ/ФППВ, является элементарная расширенная машина Тьюринга [15].

Доказательство. Элементарная расширенная машина Тьюринга (ЭлРМТ) является расширением классической машины Тьюринга (МТ), которая уже способна реализовать произвольный алгоритм. При этом ЭлРМТ обладает *планом исполнения*, может ставить новые состояния в начало или конец плана, исполняет план, извлекая из его начала новое состояние при каждой промежуточной остановке машины. Этот минимальный механизм аналогичен логике работы ПППВ/ФППВ, следовательно ЭлРМТ является более адекватным вычислителем для ФППВ/ПППВ, нежели классическая машина Тьюринга, сохраняя предельные свойства последней.

Теорема о реализуемости ТРБ1. Базовые алгоритмические средства Planning C (ПППВ/ФППВ) реализуемы: можно построить транслятор, генерирующий соответствующий код на произвольном алгоритмически полном языке.

Доказательство. Согласно ТАПВ1, ПППВ/ФППВ могут быть реализованы на ЭлРМТ (являющейся надмножеством МТ), то есть существуют алгоритмы их реализации, разрешимые по Тьюрингу, Следовательно, такие алгоритмы могут быть реализованы на любом алгоритмическом Тьюринг-полном языке. Для этого достаточно построить транслятор, генерирующий соответствующий код, представляющий указанные алгоритмы. Данная задача тривиальна, поскольку в предельном случае можно однозначно генерировать код для ЭлРМТ по известному

графу переходов (алгоритму), а в практически важных случаях — генерировать код на алгоритмически полном языке высокого уровня, например, по известной блок-схеме. Доказано.

В данном пункте нам осталось только привести некоторые примеры программирования с использованием сформулированных выше предельных средств. Следующий фрагмент кода вводит ППВП **Array**, являющуюся генератором массива целых чисел (размер массива задается значением переменной **ArrayN**). Используется эквивалент цикла.

```
int ArrayN = 0;
reenterable[200] static Array(int Data)
{
 ArrayN-- > 0 && plan_first(Data) && plan_last(Data) ||
plan_stop;
}
```

Функция **GenArray(N)** является «оберткой», вызывающей **Array** для генерации нового массива. Используется эквивалент ветвления с телом в лямбда-функции.

```
plan_type(Array) GenArray(int N) {
 plan_type(Array) Empty;
 ArrayN = N;
 *Array = Empty;
 auto callArray = [&] ()->bool {
   Array(0);
   return true;
 };
 N > 0 && callArray();
 return *Array;
}
```

Функция **DelArrayItem(A, index)** удаляет из массива **A** элемент с индексом **index**. Используется вспомогательная ФППВ **_DelArrayItem**, выполняющая реальное удаление с помощью эквивалента цикла.

```
reenterable int _DelArrayItem(plan_type(Array) & A, int in-
dex) {
  A[index] = A[index+1];
  index < _size(A)-2 && plan_last(A, index+1);
  return 1;
}
void DelArrayItem(plan_type(Array) & A, int index) {
  index < _size(A)-1 && _DelArrayItem(A, index);
  plan_item_type(Array) dummy;
  A>>dummy;
}
```

И еще одна функция, выводящая сгенерированный массив **A** на экран, начиная с позиции **j**. Здесь в решении участвует лямбда-функция (по соображениям повышения читаемости кода), представляющая тело прямой ветви.

```
reenterable int OutArray(plan_type(Array) &A, int j) {
 auto if_body = [&] ()->int {
   cout<<A[j].Data<<" ";
   plan_first(A, j+1);
   return 1;
 };
 j < _size(A) && if_body();
 return 1;
}
```

Выводы к первой главе

В данной главе предложены новые формализмы и синтаксические средства, предназначенные, преимущественно, для описания различных (в том числе многостадийных) алгоритмов, подразумевающих применение стека, дека или очереди. Синтаксис подробно описан. Показано, что новый подход дает более компактную и очевидную реализацию перечисленных выше алгоритмов, в том числе с применением редукционных операций над данными, реализуемыми бинарными коммутативными операциями и функциями. Возможна разработка аналогов простых и рекурсивных генераторов. Рассмотрена технология, позволяющая автоматизировать проверку зависимостей по данным при обработке (в том числе параллельной) сложных рекурсивных структур (сетей, деревьев) путем либо декларативной разметки соответствующего типа либо определения специальных декларативно-императивных правил его обработки.

Предложенные средства позволяют также *неявно декларировать сложные типы (стек, дек, очередь, список, массив), новые переменные* этих типов и их *обработчики* (ПППВ/ФППВ), реализовать *произвольные (ветвящиеся, циклические, рекурсивные) алгоритмы*.

Предложенные средства эквивалентны по выразительным возможностям средствам структурного программирования. Адекватным предельным вычислителем, способным реализовать ПППВ/ФППВ, является элементарная расширенная машина Тьюринга [15].

Глава 2. Распараллеливание вычислений в Planning C

Целью данной главы является повышение адекватности, эффективности и безопасности описания параллельных алгоритмов, базирующихся на ряде как классических так и нестандартных паттернов параллелизма, основываясь на введенном в предыдущей главе формализме процедур/функций с планированием повторного входа и цепей таких процедур.

Для достижения данной цели поставим следующие *задачи*: а) провести обзор некоторых современных языков, расширений и интерфейсов параллельного программирования, б) определить основные тенденции формирования набора различных средств распараллеливания и потребности в них, в) определить паттерны параллелизма, реализация которых естественным образом укладывается в базовую концепцию ППП В и требует ввода лишь незначительных новых синтаксических средств; г) определить возможности применения данных средств (также с минимальными изменениями) для различных основных архитектур вычислителей; д) обобщить, насколько это возможно, разработанные средства для случаев произвольных паттернов параллелизма; е) предложить дополнительные универсальные и специализированные (преимущественно архитектурно-зависимые) средства, которые в совокупности с основными покроют определенные нами базовые потребности в средствах распараллеливания; ж) определить эффективность реализации некоторых типовых алгоритмов с применением разработанных средств.

2.1. Обзор современных языков, расширений и интерфейсов параллельного программирования

Первой задачей, которую необходимо решить, является определение набора необходимых и достаточных средств параллельного программирования. Применим нисходящий (путем классификации, от общего к частному) метод, который подкрепим соответствующим обзором существующих языков, расширений и интерфейсов параллельного про-

граммирования. В основу классификации может быть положен один из *двух основных принципов*: по виду параллелизма или по основных архитектурам вычислителей [5, 6].

По основным архитектурам и вытекающим из них возможностям реализации определим следующую классификацию:

1. Системы с общей памятью.

1.1. Универсальные SMP-системы (SIMD, MIMD).

1.1.1. В реализациях с классическими переменными.

1.1.1.1. По требованиям исключительности доступа к коду и/или данным.

1.1.1.1.1. Средства, обеспечивающие исключительный доступ (семафоры, блокировки, мониторы, критические секции, атомарные операции и редукция).

1.1.1.1.2. Средства, использующие транзакционную модель памяти (транзакционные блоки и процедуры/функции).

1.1.1.2. Средства синхронизации (барьеры, ожидания, условные переменные).

1.1.1.3. Средства порождения подзадач/потоков.

1.1.2. В реализациях с «ленивыми» переменными (средства декларации таких переменных).

1.2. Векторные расширители (SIMT).

1.2.1. Специфические средства (планирование и запуск пула потоков, некоторые векторные операции).

1.2.2. Средства, применяемые для универсальных SMP-систем (барьеры, атомарные операции и др.)

2. Системы с разделяемой памятью.

2.1. Организация виртуальной общей памяти (используется то или иное подмножество средств для работы в SMP-системах).

2.2. Средства порождения процессов.

2.3. Средства передачи сообщений.

2.3.1. По требованиям синхронности/асинхронности.

2.3.1.1. Синхронные коммуникации.

2.3.1.2. Асинхронные коммуникации.

2.3.2. По требованиям к блокировке.

2.3.2.1. Блокирующие коммуникации.

2.3.2.2. Неблокирующие коммуникации.

2.3.3. По количеству участников.

2.3.3.1. Коммуникации «точка-точка».

2.3.3.2. Коллективные коммуникации.

3. Гибридные системы.

3.1. Многоуровневый подход (на каждом уровне архитектуры используются средства, типичные для данной архитектуры).

3.2. Виртуализация (выбирается набор средств, характерный для одной архитектуры, на уровнях с иной архитектурой он реализуется путем виртуализации).

По видам параллелизма[1] *и вытекающим из них дополнительным условиям* можно определить следующую классификацию:

1. Параллелизм по процессам.

1.1. Средства порождения процессов.

1.2. Шаблонные взаимодействия процессов (определяемые топологиями).

1.2.1. Средства организации/взаимодействия в топологии «вектор»

1.2.2. Средства организации/взаимодействия в топологии «конвейер»

1.2.3. Средства организации/взаимодействия в топологии «портфель задач».

1.2.4. Средства организации/взаимодействия в топологиях «звезда», «клика», «гиперкуб» и прочих топологиях.

1.3. Универсальные взаимодействия процессов.

1.3.1. Средства, характерные для SMP-систем (см. предыдущую классификацию).

1.3.2. Средства передачи сообщений (см. предыдущую классификацию).

1.3.3. Специфические средства, определяемые особенностями языка или архитектуры системы.

2. Параллелизм по данным (реализуется либо декларативно, например, разметкой, либо императивно, например, с помощью распараллеленных циклов).

[1] Параллелизм по инструкциям включен в качестве одного из подпунктов.

2.1. Средства работы с линейными данными (сюда относятся не только высокоуровневые средства, но и векторные команды современных процессоров, например, MMX, SSE, AVX).

2.2. Средства работы с многомерными данными.

2.3. Средства работы с рекурсивными данными.

Перейдем теперь к рассмотрению некоторых, наиболее характерных и часто применяемых (в настоящее время) языков, расширений и интерфейсов параллельного программирования, особо отмечая, какие из вышеупомянутых средств параллельного программирования в них используются.

1. MPI-3 [5, 34, 51]. Содержит широкий набор синхронных и асинхронных, блокирующих и неблокирующих средств обмена данными (коллективных и по схеме «точка-точка»). Есть средства «одностороннего» обмена данными через «окна» — удаленный доступ к памяти в другом процессе. Присутствуют средства порождения процессов, с которыми поддерживается взаимодействие через интеркоммуникатор (что не вполне удобно). Кроме того, есть возможность связи и с процессами, не относящимся к порожденным текущей программой (по технологии, схожей со стандартными свойствами обмена данными по протоколу TCP/IP). Есть поддержка виртуальных топологий, в том числе универсальный конструктор графа-топологии (не вполне удобный для прочтения и описания). Данные средства поддерживаются для программирования как в общей, так и в разделяемой памяти.

2. C$ [1]. Ориентирован на работу с GPU. Поддерживает явный параллелизм по элементам векторно-матричных данных. Есть неявные ленивые вычисления. Язык реализуется интерпретатором (динамически определяются параллельные ветви, видимо, отправляемые на GPU). Код программы не вполне очевиден для прочтения. Примечательно, что синтаксис цикла часто отсутствует — используется умолчание о переборе по возможным значениям индексов.

3. Delphi XE7 [59]. Используется библиотека стандартных классов и шаблонов (PPL), в которую требуемые для параллельного исполнения фрагменты кода передаются в теле сторонних (анонимных или обычных) процедур/функций. Имеются средства порождения параллельных задач (с возможностями выполнения атомарных операций, ожиданий), средства организации непополняемого параллельного цикла, средства запуска

в параллель функции с «ленивым результатом»: классы TTask и TParallel с методом For, интерфейс IFuture. В-целом, возможности Delphi схожи с возможностями OpenMP 3.0 (с действием директив task, for, atomic), но несколько беднее.

4. Cilk [36]. Императивный язык на базе C. Используется явное указание параллелизма со вставкой нескольких ключевых слов (при их игнорировании получаем корректно работающую непараллельную программу). Основан на порождении параллельно работающих cilk-подзадач с возможностью ожидания их завершения. Содержит средства для работы с блокировками, а также inlet-процедуры, гарантирующие атомарность работы с переменными (возможно реализуются критическими секциями).

5. Cilk Plus. Расширение Cilk, в который введены параллельное исполнение циклов, специальная нотация для векторной обработки массивов и атомарные (потокобезопасные) переменные.

6. Go [60]. Схож с Cilk в плане подхода к порождению подзадач (реализуется единственным ключевым словом go). Включает каналы (синхронные и буферизованные асинхронные, одно- и двунаправленные) для взаимодействия параллельных процессов. Имеет функцию выбора готового к приему данных канала. Есть библиотека функций, включающих реализацию условных переменных, блокировок и ожидания завершения группы процессов.

7. Haskell [50]. Содержит достаточно большое количество средств программирования для систем разных архитектур. В основе языка лежит концепция «ленивых» переменных и возможности явного указания порождаемых гранул параллелизма. Допускается параллелизм по данным (определяется с помощью разметки), как по линейным, так и по рекурсивным, путем указания общего кода, исполняемого для каждого элемента. На схожем принципе (применения общего кода для элементов структурированных данных) основано использование возможностей векторных GPU-расширителей. Есть средства работы с транзакционной памятью. Дополнительно имеются возможности явного порождения и взаимодействия (с помощью атомарных переменных, блокировок, синхронных каналов) потоков. Поддерживается распараллеливание для систем с разделяемой памятью (используется порождение процессов и передача сообщений).

8. Erlang [44]. Это декларативный язык, поддерживающий порождение легковесных процессов, асинхронно обменивающихся сообщениями. Ориентирован на работу как на SMP-системах, так и на системах с разделяемой памятью. Допускается группировка (связывание) процессов. Процессы работают в разделенных областях памяти, доступ к которым осуществляется только через соответствующие процессы. При этом, фактически, реализуется парадигма параллельного программирования с виртуально разделенной памятью, не требующая ни блокировок, ни семафоров.

9. Java [36]. Используются потоки. Имеются синхронизированные методы и блоки, обеспечивающие исключительный доступ к данным (заменяют блокировки и критические секции). Кроме того, имеется поддержка сигналов и механизма ожидания завершения потоков. Значительное количество специфических средств программирования содержится в библиотеке: семафоры, барьеры, синхронный обмен объектами, атомарные операции, очереди с многопоточной обработкой, «портфель задач», «портфель» рекурсивно порождаемых задач.

10. OpenMP [5, 13]. Представляет собой, преимущественно, средства разметки кода и небольшую библиотеку функций. Имеются средства распараллеливания циклов, организации параллельных секций, порождения подзадач. Используется многопоточная модель, поддерживаются атомарные операции, блокировка, редукция, вложенный параллелизм. При удалении директив разметки программа обычно работает как однопроцессорная, без изменения алгоритма и результатов.

11. MC# [24]. Поддерживает асинхронный запуск функции/процедуры на многоядерной и/или кластерной системе, а также на векторном GPU-расширителе. Для взаимодействия и синхронизации запущенных процессов используются асинхронные каналы (с очередью), связываемые с функциями-обработчиками. Обработчики реализуют чтение из канала. Связка обработчика с несколькими каналами, срабатывающая только после прихода данных на все связанные каналы, может использоваться для синхронизации. Запуск в отдельном потоке, на отдельной машине кластерной системы или на GPU специфицируется с помощью специальных ключевых слов. Функции, исполняемые на GPU, пишутся в соответствии с идеологией CUDA, для их запуска, помимо специального ключевого слова, используется еще несколько инициали-

зирующих функций. Аргументы GPU-функций неявно копируются в память расширителя и обратно (это весьма компактное и прозрачное решение для программиста (в отличие, например, от OpenAcc), схожий подход будет применен в настоящей работе).

12. Concurrent C [36, 43]. Программа состоит из одного или нескольких процессов (оформленных в стиле функций). Процессы порождаются динамически, как подзадачи. Взаимодействие процессов поддерживается посредством синхронных и асинхронных транзакций. Инициирование транзакции схоже с вызовом процедуры/функции. Присутствует специальный оператор выбора активной транзакции из указанного множества. Встроенные средства блокировки при работе с общей памятью отсутствуют. Программы транслируются на стандартный C.

13. F# [41]. Основан на идее применения «ленивых переменных». Поддерживается параллельный запуск асинхронных процессов, поддерживаются блокировки. Есть специальный класс Mailboxprocessor, предназначенный для обработки пополняемой извне (в параллельном режиме) очереди сообщений. Может использоваться .NET-библиотека для программирования в общей памяти (содержит реализации семафора, барьера, потоков, ожидания, атомарных операций и иных базовых средств).

14. T-система [5]. Построена на принципе использования неготовых («ленивых») переменных с возможностью асинхронного запуска функций программы в параллельном режиме, при этом в стандартный язык вводятся всего два новых ключевых слова — tfun для обозначения асинхронной функции и tval для спецификации неготового значения. При удалении или игнорировании этих ключевых слов программа работает по тому же алгоритму, но в непараллельном режиме.

15. Windows API. Операционная система Windows поддерживает многопроцессное/многопоточное программирование и содержит все необходимые средства для организации параллельной обработки как на SMP-системах, так и на кластерных системах. Взаимодействие процессов может осуществляться через общую память (есть блокировки, семафоры, критические секции, события и другие средства), путем передачи сообщений, по сетевым протоколам, с помощью технологий COM/DCOM [27]. Чтобы не повторяться, просто отметим, что многие из

упомянутых средств присутствуют и в иных современных многозадачных операционных системах, например, в Linux/Unix.

16. DVM [11]. Поддерживает работу на SMP-системах, кластерах, векторных GPU-расширителях и гибридных системах. Параллелизм описывается разметкой с помощью специальных директив. Переменные подразделяются на распределенные и локальные. Поддерживается редукция. Параллелизм по данным реализуется путем распределения витков цикла по процессорам. Параллелизм задач обеспечивается распределением данных и вычислений по секциям массива процессоров. Вычисления вне параллельных циклов исполняются на тех процессорах, на которых присутствует переменная, значение которой изменяется в результате выполнения операции. Возможна как синхронная, так и асинхронная работа с удаленными данными. Интересно отметить, что асинхронная обработка основана на динамическом сборе информации о ячейках, значения которых необходимо будет предварительно подгрузить при последующем выполнении того же блока кода. Дополнительно заметим, что DVM накладывает ряд ограничений на базовый язык и несколько сложен в части описания распределения данных по процессорам.

17. Unified Parallel C [40]. Обеспечивает параллельную обработку данных в системах с общей или разделяемой памятью. В целом соответствует SPMD (Single Program Multiple Data)-модели параллельных вычислений, причем используется явная спецификация параллелизма. Язык основан на идее секционирования общей памяти, секции логически распределены между потоками. Возможно применение барьеров и блокировок, также может указываться модель контроля целостности данных (strict (атомарный подход) или relaxed (без контроля)). Данные, описанные с модификатором shared, автоматически распределяются по потокам и могут быть обработаны в параллельном режиме. Они доступны или по индексу (если это массив), или по указателю. Любопытна внутренняя конструкция указателя (при таком подходе) — он содержит номер потока, базовый адрес и позицию относительно блока. Язык поддерживает распараллеливание циклов с независимыми итерациями.

18. OpenCL [52]. Предназначен для программирования преимущественно векторных вычислений на GPU-расширителе или на обычном универсальном процессоре, имеющим команды векторной обработки (например, MMX, SSE). Содержит библиотеку функций поддержки за-

пуска программ на соответствующем векторном устройстве и обмена данными с ним, а также упрощенный диалект языка C, на котором и пишется программа для расширителя. Такие программы взаимодействуют через общую память, причем возможно применение барьеров, атомарных и векторных операций. В целом, используется SIMT (Single Program Multiple Threads)-модель. Расширитель взаимодействует с основной программой, исполняющейся на центральном процессоре, через виртуальную общую память или через специфические функции копирования данных. Взаимодействие между kernel-программами на расширителе возможно через специальные объекты pipe, что обеспечивает поддержку параллелизма по процессам (задачам).

OpenCL не вполне удобен в программировании, поскольку процедуры копирования данных на расширитель и запуска векторной kernel-программы с получением результатов требуют вызова множества специальных функций. Тем не менее, отметим, что важным плюсом OpenCL является универсальность предложенного подхода, в отличие, например, от технологии CUDA, нередко используемой для программирования параллельных вычислений на многоядерных видеокартах nVidia.

19. Ada [36, 54, 58]. Реализован как для систем с общей памятью, так и для систем с разделяемой памятью. Поддерживает параллелизм циклов, защищенные типы (экземпляр такого типа эквивалентен монитору) и синхронную передачу сообщений по принципу рандеву. Параллельные модули описываются в формате задач, синтаксически схожих с процедурами. Присутствует специальный оператор выбора (на принимающей стороне) активного рандеву. Доступ к общим переменным незащищенного типа не блокируется, для этого необходима явная реализация блокировок со стороны программиста, например на базе вышеупомянутых рандеву. При исполнении программы в системе с разделяемой памятью доступ к памяти, физически относящейся к иной машине кластера, выполняется системой и прозрачен для программиста.

20. Threading Building Blocks (TBB) [46]. Это библиотека шаблонов классов и функций, реализующих различные паттерны параллелизма, абстрагированные от деталей архитектуры системы. При этом операции трактуются как задачи, динамически распределяемые по ядрам системы. Библиотека содержит блокировки, условные переменные, атомарные операции, параллельные алгоритмы (цикл, конвейер, векторная

обработка, редукция и другие), а также средства описания параллелизма в виде графа зависимостей элементов (по данным), которые автоматически запускаются на параллельное исполнение в соответствии лишь с теми ограничениями, которые наложены зависимостями. Кроме того, имеются средства порождения задач и ожидания их завершения. Следует отметить, что данный набор возможностей достаточно интересен, но, к сожалению, синтаксис результирующей программы несколько сложен для понимания и, вероятно, не вполне удобен для разработки.

21. OpenAcc [61]. Предназначен для программирования параллельных вычислений на многоядерных процессорах и векторных GPU-расширителях. Как и OpenMP, построен на директивах разметки, сопровождаемых вызовами библиотечных функций, исполняемых на центральном процессоре. Поддерживает три уровня параллелизма: по группам, по участникам группы, а также векторный параллелизм на уровне команд. Возможен параллельный запуск фрагментов кода, параллельное исполнение циклов, весьма интересна возможность автоматического определения в коде параллельных kernel-фрагментов. Поддерживается редукция. Имеются директивы для копирования данных между центральным процессором и расширителем. Допускаются атомарные операции. Содержит средства разметки, позволяющие в явном виде указать функции, которые компилируются как для центрального процессора, так и для расширителя — это достаточно разумный и практичный подход, который получит развитие и в данной работе.

Подводя итоги данного краткого обзора, следует отметить наличие двух основных (и несколько противоположных) тенденций:

а) многие языки/расширения (Ada, Erlang, C$, Cilk, OpenMP, OpenCL, MC# и др.) стремятся *минимизировать набор используемых средств распараллеливания*, что, видимо, объясняется тенденциями к универсализации таких средств для различных архитектур и к максимальному упрощению распараллеливания (иногда сводимому к некоторой разметке кода, удаление которой часто не меняет алгоритм исполнения программы, лишь переводя его в непараллельный режим);

б) ряд языков/расширений (Haskell, Java, MPI, TBB), напротив, стремится к *максимизации средств поддержки параллельного программирования*, предлагая *специфические решения* как для различных классов алгоритмов/топологий, так и, иногда, для различных архитектур вы-

числительных систем, что объясняется, видимо, тенденциями к максимально возможному повышению эффективности работы конечной программы (путем применения конкретных, оптимизированных под конкретные случаи средств) и, иногда, к повышению легкости применения таких средств (особенно для программистов, использующих в своей основной работе иные языки), которые нередко являются достаточно стандартными.

Отметим, что для первой тенденции достаточно характерен подход, когда базовый набор средств распараллеливания в значительной степени диктуется особенностями языка (Erlang, C$). Также заметим, что некоторые языки/расширения занимают, скорее, нейтральную позицию между вышеупомянутыми основными тенденциями (Delphi, Go).

Предположим, что в нашем случае наиболее адекватным решением станет *композиция двух вышеупомянутых подходов*, при которой разрабатываемый язык должен:

а) покрывать все основные потребности, определяемые набором базовых возможных архитектур и видов параллелизма;

б) содержать некий минимальный набор базовых и достаточно универсальных средств распараллеливания (в значительной степени диктуемых особенностями выбранной парадигмы программирования с применением процедур с планированием повторного входа), обеспечивающих достаточные удобство, минимальность, безопасность и кроссплатформенность решения на базе ППП/ФППВ для основных рассматриваемых нами одно- или многостадийных алгоритмов, предполагающих сведение задачи к серии подзадач, решаемых с применением массивов, стека, дека или очереди;

в) содержать наборы незначительно специализированных, алгоритмо- и архитектурно-ориентированных средств (библиотек функций и дополнительных ключевых слов), повышающих удобство и эффективность реализации алгоритмов. В частности, имеет смысл ввести специальные средства для работы с виртуальной общей памятью на кластере, специализировать семафоры и каналы, создав частные (и, как следствие, более эффективные) их версии для SMP-систем и кластерных систем, добавить некоторые средства исключительно для SMP-систем (блокировки, которые для таких систем несколько более легковесны в сравне-

нии с семафорами, и критические секции, не имеющие смысла для кластера).

Дополнительно отметим (также по результатам обзора), что рассмотренные нами языки и расширения практически не предоставляют достаточно удобных и эффективных средств организации произвольных виртуальных топологий (средства MPI не удобны, средства TBB несколько сложны в применении) и, как ни странно, в большинстве своем (за исключением TBB) не содержат таких средств даже для такой простой и достаточно часто применяемой топологии как конвейер. При этом топологии, несомненно, являются достаточно ценным формализмом, поскольку позволяют компилятору или динамической среде исполнения иметь априорную информацию о предполагаемых направлениях обменов и, на ее основе, оптимизировать передачи данных. Поэтому, поставим задачу разработки достаточно простых, мощных и эффективных средств описания произвольных и некоторых простых специфических виртуальных топологий (на базе процедур с планированием повторного входа) для создаваемого языка. Вероятно, эти средства, использующие множество ПППВ (на кластерной или SMP-системе), в сочетании со средствами, работающими с одной ПППВ (на SMP-системе и/или на векторном расширителе), и с некоторыми функциями синхронизации, сформируют декларированный нами выше минимальный набор базовых (и достаточно универсальных) средств распараллеливания.

2.2. Классические паттерны параллелизма в системах с общей или разделяемой памятью

В данном разделе будет рассмотрена реализация (на разрабатываемом языке) некоторых базовых шаблонов параллелизма, которые наиболее компактно и адекватно описываются с помощью введенного нами формализма ПППВ/ФППВ и цепей ПППВ. Особый интерес представляют возможности реализации паттернов «вектор», «конвейер» и «портфель задач», которые, вероятно, будут востребованы при программировании решения многостадийных переборных и неперебoрных задач обучения глубоких нейронных сетей прямого распространения [23] с применением предлагаемых в данной работе подходов. Первичное же

изложение методологии программирования с применением данных патернов мы построим на базе более простых и общеупотребимых алгоритмов.

2.2.1. Случай единственной процедуры. «Портфель задач»

Если рассматривать планирование очередного этапа исполнения процедуры как планирование новой, относительно независимой подзадачи, которая может быть исполнена на отдельном процессоре/ядре, то применение процедур с повторным входом может быть одним из вполне оправданных подходов к распараллеливанию алгоритма. Это особенно простой вариант распараллеливания для алгоритмов, реализация которых с применением процедур с планированием повторного входа выгодна изначально.

В эту группу входят, в частности, алгоритмы групповой обработки элементов, включенных в сложную нелинейную структуру данных (дерево, сеть), например, поиска минимального элемента дерева или суммирования таких элементов. Для эффективной обработки линейных структур данных (массивов) в параллельных языках программирования и параллельных расширениях стандартных языков обычно существуют стандартные средства, которые для повышения эффективности даже могут предусматривать балансировку загрузки процессоров системы, например, конструкция **parallel for** в расширении OpenMP [5] имеет специальные режимы **dynamic** и **guided**, регулирующие загрузку процессоров (методология «портфеля задач» [36]). Для параллельной обработки нелинейных структур данных стандартных средств пока нет, хотя ряд задач может быть реализован с применением конструкции по типу **task**[1] (см., например, [38]) стандарта OpenMP 3.0, которая также позволяет балансировать загрузку процессоров в режиме **untied**.

Полноценным решением может быть «линеаризация» задачи обработки нелинейных структур, если для реализации алгоритма воспользоваться формализмом процедур с повторным входом, предполагающим исполнение линейного плана этапов обработки. Задача сведется к адекватно-

[1] Стандарт OpenMP 3.0, вводящий данную конструкцию, дан в спецификации по ссылке http://openmp.org/mp-documents/spec30.pdf

му распараллеливанию процесса исполнения плана работ, что может быть реализовано внутренними средствами системы с применением стандартных приемов параллельной обработки линейных структур данных с балансировкой, например, той же идеи «портфеля задач». Разумеется, в таком случае необходимо четко выделять фрагменты плана, в пределах которого возможно одновременное исполнение работ.

Отметим, что для некоторых алгоритмов применение процедур с планированием повторного входа не дает существенного эффекта при работе с одноядерными системами[1], но позволяет эффективно распараллелить расчет в многопроцессорных/многоядерных системах. Например, применение *повторного входа со вставкой в начало плана* при определенных условиях может быть эквивалентно программированию с применением рекурсии с точки зрения организации последовательности вызовов. В системах с общей памятью замена рекурсии на планирование в начало (итерацию) может быть *одной из стратегий*, позволяющих с минимальными затратами (или даже автоматически) распараллелить алгоритм. Наиболее эффективна такая стратегия для алгоритмов с полностью независимыми по данным ветвями дерева рекурсивных вызовов, например, для поиска оптимального хода в позиционных играх. Решение будет не менее простым, чем, например, полученное с применением концепции Т-параллелизма [5] или схожего с ним подхода [4] на базе мелкогранулярного параллелизма асинхронно исполняемых функций.

2.2.1.1. Базовые языковые средства

Планирование этапов обработки в общем случае, безусловно, подразумевает наличие детерминированной последовательности исполнения этих этапов, которая для случая одноядерной системы определяется естественной последовательностью просмотра плана — от начала к концу. В случае многопроцессорной системы необходимо, придерживаясь, в целом, той же последовательности, обеспечить параллельность исполнения, по меньшей мере, некоторых этапов.

Предлагается ввести понятие *группы этапов* и представить план в виде последовательности непересекающихся групп. Этапы группы могут

[1] Имеется в виду однопроцессорный компьютер, процессор с одним ядром

выполняться либо последовательно, либо параллельно. Всякая очередная группа по умолчанию исполняется в последовательном режиме, начиная с первого включенного в нее этапа. В процессе последовательного исполнения группы возможно *однократное переключение* в режим параллельного исполнения (обратное переключение недопустимо). *В параллельном режиме* группа рассматривается как динамически пополняемый «портфель задач»: любые этапы, входящие в группу (в том числе и новые, попадающие в группу в результате планирования при исполнении уже включенных в группу этапов), могут исполняться параллельно и в любой последовательности на любом из процессоров системы.

Необходимо учесть, что в параллельном режиме обеспечить «тунелирование по умолчанию» и корректность отсечения значений нередукционных параметров, передаваемых по ссылке, иногда невозможно. Такие параметры будут доступны для одновременной модификации всем процессорам на любом этапе и, следовательно, могут рассматриваться как глобальные переменные, для корректной работы с которыми необходимо использовать семафоры или иные средства монопольного доступа.

План делится на группы с помощью *маркеров* — специальных пометок в плане, которые могут разделять два любых этапа, а также присутствовать до первого и/или за последним этапом. Группой является:

а) весь план, если он не содержит маркеров;

б) участок плана между двумя маркерами;

в) участок, начинающийся с первого этапа и заканчивающийся этапом перед ближайшим справа маркером;

г) участок, заканчивающийся на последнем этапе плана и начинающийся этапом после ближайшего слева маркера.

Позиция плана, соответствующая концу произвольной группы (маркеру или концу плана), фактически является *точкой барьерной синхронизации*. Перед началом работы (при явном вызове процедуры с повторным входом) план включает единственный начальный этап, не содержит маркеров и, следовательно, является группой.

Предлагается ввести новые ключевые слова (**plan_group_first**, **plan_group_last**, **plan_group_parallelize**), представляющие, соответственно *три новые базовые операции* для работы с планом:

– вставку маркера в начало плана;

– вставку маркера в конец плана;

– переключение из последовательного режима исполнения группы в параллельный.

Также предлагается ввести четыре функции:

а) **plan_processor_id()**, возвращающую логический номер процессора (ядра) в контексте текущей параллельной конструкции, на котором исполняется этап плана;

б) **plan_processors()**, возвращающую количество процессоров (ядер), задействованных в обработке текущей параллельной конструкции;

в) **plan_locking()**, блокирующую вышеуказанные базовые операции для работы с планом;

г) **plan_unlocking()**, снимающую блокировку операций с планом.

Две последние функции необходимо применять при явной работе с планом ППППВ/ФППВ (например, при его внешнем заполнении/чтении, см. п.1.5.2) на каком-либо этапе для предотвращения его модификации иными этапами той же ППППВ/ФППВ или иными ППППВ/ФППВ, если работа идет в параллельном режиме.

При работе с одноядерной системой новые ключевые слова игнорируются, функция **plan_processor_id()** всегда возвращает номер единственного процессора (ноль), функция **plan_processors()** всегда возвращает единицу, функции блокировки/деблокировки игнорируются. Такая реализация следует основным принципам построения современных параллельных расширений языков программирования (OpenMP, DVM [11], HPF/HPC [36]). Поэтому легко предложить варианты интеграции средств, реализующих новые операции для работы с планом, в состав таких расширений.

Возможен, например, вариант с модификацией стандарта OpenMP: достаточно ввести новую директиву (синтаксис представлен в видоизмененной расширенной нотации Бэкуса-Наура без указания кавычек для терминальных элементов), с помощью которой можно реализовать все три базовые операции

«#pragma» пробелы «omp» пробелы [«parallel» пробелы] «plan» пробелы («first» | «last» | «parallelize»)

2.2.1.2. Параллельная реализация некоторых алгоритмов

Приведем простой и малозатратный вариант параллельной реализации алгоритма поиска максимального элемента в дереве с корнем **Root**. Узлы дерева представляют собой элементы данных типа **TreeNode**, константа **nProcs** — количество ядер в системе. Параллельный поиск[1] представлен процедурой с повторным входом **_TreeMax**, функция **TreeMax** (с параметром — указателем на корень дерева) собирает частные результаты поиска по всем процессорам и возвращает конечный результат:

```
int MaxResult[nProcs];
reenterable _TreeMax(TreeNode * Cur) {
 int thread_id = plan_processor_id();
 if (Cur==Root) plan_group_parallelize;
 if (Cur->Left) plan_last(Cur->Left);
 if (Cur->Right) plan_last(Cur->Right);
 if (MaxResult[thread_id]<Cur->Data)
    MaxResult[thread_id] = Cur->Data;
}
int TreeMax(TreeNode * Root) {
 int Result;
 memset(MaxResult,0,sizeof(MaxResult));
 _TreeMax(Root);
 Result = MaxResult[0];
 for (int i=1; i<nProcs; i++)
  if (MaxResult[i]>Result) Result = MaxResult[i];
 return Result;
}
```


Заметим, что вышеприведенную программу также можно написать с использованием редукционного параметра, передаваемого по ссылке.

```
reenterable _TreeMax(TreeNode * Cur, reduction(max) int
&Max) {
 if (Cur==Root) plan_group_parallelize;
 if (Cur->Left) plan_last(Cur->Left,Max);
```

[1] Реализация алгоритма с применением многих параллельных расширений C/C++ потребовала бы введения дополнительной переменной (очереди обработки или массива ссылок на узлы дерева) и цикла обработки, итерации которого распределялись бы между процессорами. Естественнее выглядят реализация поиска с применением T-параллелизма и OpenMP 3.0, но такой подход может быть менее эффективным, учитывая внутренние трудозатраты на организацию контроля готовности значений переменных в T-системе и на порождение/синхронизацию подзадач в OpenMP 3.0.

```
 if (Cur->Right) plan_last(Cur->Right,Max);
    Max = Cur->Data;
}
int TreeMax(TreeNode * Root) {
 int Result;
 _TreeMax(Root,Result);
 return Result;
}
```

Реализация того же алгоритма с применением OpenMP 3.0 выглядит следующим образом:

```
int _TreeMax(TreeNode * Cur) {
  int L = -1E300;
  int R = -1E300;
  int C = Cur->Data;
  if (Cur->Left)
     #pragma omp task shared(L) untied
     L = _TreeMax(Cur->Left);
  if (Cur->Right)
     #pragma omp task shared(R) untied
     R = _TreeMax(Cur->Right);
  #pragma omp taskwait
  return max(C,max(L,R));
}
int TreeMax(TreeNode * Root) {
  int Result;
  #pragma omp parallel
  {
   #pragma omp single
   Result = _TreeMax(Root);
  }
  return Result;
}
```


Легко видеть, что реализация с применением процедур с повторным входом выигрывает по компактности у OpenMP-реализации (11 операторов против 13-ти и 1 директива распараллеливания[1] против 5-ти). При этом *параллелизм варианта с применением процедур с планированием более прозрачен*, поскольку относится к линейному плану и не требует поиска ветвей, которые могут исполняться в параллель. То же самое можно сказать и о целом ряде иных, рассматриваемых далее примеров.

[1] Здесь и далее plan_group_parallelize считается директивой, а не оператором.

Также представляет интерес параллельная реализация сортировки массива целых чисел методом Шелла. Метод предполагает последовательное выполнение стадий, на каждой из которых решается (в параллельном режиме) серия однотипных подзадач сортировки фрагментов массива. Выделение стадий реализуется группировкой этапов плана, каждому из которых (за исключением первого, на котором формируется план) ставится в соответствие подзадача сортировки.

```
reenterable Shell(char Plan, int * Arr, int N, int incr) {
 static int Incr[100] = {1};
 int i,j;
 plan_group_parallelize;
 if (Plan) { /* Генерируем общий план исполнения */
   for (i=0; Incr[i]<N; Incr[i+1] = 3*Incr[i]+1, i++);
   while (i--) {
    int NLists = N%Incr[i] ? N%Incr[i] : Incr[i];
    int * B = Arr;
    plan_group_last; /* Отмечаем начало новой стадии */
    for (j=0; j<NLists; j++)
      plan_last(0,B++,N--,Incr[i]);  /*  Создание  подзадачи
*/
   }
 }
 else /* Решение подзадачи */
   /* Сортировка фрагмента массива Arr[0,incr,2*incr,…] */
   InternalSort(Arr,N,incr);
}
/* Вызов: Shell(1,массив,число_элементов,0); */
```

Продемонстрируем также возможность достаточно эффективного *распараллеливания циклических алгоритмов с заранее неизвестным количеством независимых друг от друга итераций*. Предположим, что необходимо просуммировать результаты вычисления функции f(i), причем i = 1, 2, … Суммирование ведется, пока монотонно убывающая функция g(i) > ε, предполагается, что трудоемкость вычисления f(i) много больше трудоемкости расчета g(i).

```
double Sums[nProcs] = {0};
reenterable SumF(int i) {
 if (i==1)
    plan_group_parallelize;
 if (g(i)>Eps) {
     plan_last(i+1); /* Инициируем запуск следующего этапа
*/
     Sums[plan_processor_id()] += f(i);
```

```
 }
}
/* Вызов:
  double Sum = 0.0;
  SumF(1);
  for (int i=0; i<nProcs; i++)
    Sum += Sums[i];
*/
```

Процедура с повторным входом **SumF** формирует массив частичных сумм Sums, который остается лишь просуммировать. Для сравнения далее приведен код решения той же задачи с применением OpenMP 3.0. Общие объемы кода (9 операторов при решении с OpenMP и 10 операторов при использовании процедур с планированием повторного входа) имеют практически равные значения, тогда как количество директив распараллеливания OpenMP (4 директивы) существенно превышают аналогичный показатель (1 директива) процедур с планированием.

```
double Sums[nProcs] = {0};
double SumF() {
 double Result = 0.0;
 #pragma omp parallel
 {
 #pragma omp single
   {
    for (int i=1; g(i)>Eps; i++)
      #pragma omp task untied
      Sums[omp_get_thread_num()]+=f(i);
    #pragma omp taskwait
   }
 }
 for (int i=0; i<NP; i++)
     Result += Sums[i];
 return Result;
}
/* Вызов:
  double Sum = SumF();
*/
```

Отметим, что еще более простым может быть решение с применением редукционного параметра в процедуре с планированием:

```
reenterable SumF(int i, reduction(+) double & Sum) {
 if (i==1)
    plan_group_parallelize;
 if (g(i)>Eps) {
     plan_last(i+1,Sum);
```

```
      Sum = f(i);
 } else Sum = 0.0;
}
/* Вызов:
  double Sum = 0.0;
  SumF(1, Sum);
*/
```

Получено более компактное решение в сравнении с OpenMP 3.0: 8 операторов против 9-ти и 1 директива против 4-х.

2.2.1.3. Связь с параллельной расширенной машиной Тьюринга.

Теорема об адекватном параллельном предельном вычислителе ТАПВ2. Наиболее адекватным предельным вычислителем, способным реализовать работающую в режиме «портфель задач» ППППВ/ФППВ, является параллельная расширенная машина Тьюринга ПарРМТ [15].

Доказательство. ППППВ/ФППВ, работающая в непараллельном режиме, согласно ТАПВ1, имеет в качестве адекватного предельного вычислителя элементарную расширенную машину Тьюринга (ЭлРМТ). Существенным дополнением к алгоритму работы базовой ППППВ/ФППВ является возможность запуска исполнения плана в параллельном режиме. Соответственно, если обеспечить возможность запуска параллельного исполнения этапов плана ЭлРМТ, можно получить наиболее адекватный параллельный предельный вычислитель. Согласно [15], такая возможность обеспечивается модификацией ЭлРМТ — ПарРМТ. Следовательно, ПарРМТ является требуемым вычислителем. Доказано.

Теорема о реализуемости ТРБ2. Базовые параллельные алгоритмические средства Planning C (ППППВ/ФППВ, работающие в параллельном режиме) реализуемы: можно построить транслятор, генерирующий соответствующий код на произвольном алгоритмически полном языке.

Доказательство. Доказывается аналогично ТРБ1.

2.2.1.4. Векторизация исполнения плана процедуры с применением многоядерных OpenCL-расширителей

Выше была рассмотрена возможность запуска параллельной обработки группы этапов плана для ФППВ/ПППВ на многоядерных/многопроцессорных системах. Представляется достаточно перспективной возможность запуска такой обработки на векторных расширителях (многоядерные видеокарты или основные процессоры с векторными командами). Однако при этом необходимо учесть некоторые важные моменты:

а) необходимо четко определить (как для повышения структурированности программы, так и для упрощения процесса трансляции), какие фрагменты программы будут исполняться только основными процессорами, какие — расширителем/расширителями и какие — и основными процессорами и расширителями;

б) необходимо четко, однозначно и, по возможности, просто решить проблему копирования данных на расширители и с расширителей;

в) следует ввести средства идентификации расширителей и получения информации о них;

г) необходимы простые средства запуска (с адекватной настройкой расширителя) параллельной обработки группы этапов плана;

д) поскольку планирование пакета потоков для расширителя обычно выполняется статически (до запуска), представляется целесообразным *запретить модификации плана ПППВ/ФППВ в ходе его обработки на расширителе*.

Поставленное нами выше пожелание о возможности адекватного исполнения ПППВ/ФППВ как на графических процессорах, так и на основных процессорах с векторными командами приводит нас к достаточно очевидной идее применения OpenCL-совместимого программирования для таких процедур/функций (при этом OpenCL, вероятно, будет использоваться транслятором разрабатываемого языка на уровне реализации).

Для повышения читаемости программы и упрощения трансляции предлагается ввести специальную директиву, разрешающую векторное исполнение ФППВ/ПППВ:

«**#pragma**» пробелы «**plan**» пробелы «**vectorized**»

Теперь рассмотрим вопрос о маркировке фрагментов. Следуя логике применения схожих синтаксических средств при решении одной задачи, введем еще одну маркировочную директиву:

«**#pragma**» пробелы «**plan**» пробелы («**common**» | «**gpu**») («**begin**» | «**end**»)

В зависимости от указания **gpu** или **common** маркируется, соответственно начало (при указании **begin**) или конец (при указании **end**) блока, исполняемого только на расширителе или общего для основных процессоров и расширителя. Кроме того, необходимо предусмотреть возможность определения (на уровне препроцессора) режима, в котором в настоящий момент компилируется общий блок — для этого, *при компиляции для расширителя* определяется специальный символ **__GPU__**.

Для решения проблемы идентификации расширителей (их может быть несколько) предлагается использовать их строковые OpenCL-обозначения. В случае, если несколько расширителей имеют одинаковое обозначение, пусть транслятор автоматически добавляет к нему номер такого расширителя. Однако следует предусмотреть и возможность некоей общей индексации расширителей, например, по классической схеме C/C++: от нуля до уменьшенного на единицу общего числа расширителей. Соответственно, имеет смысл ввести *функцию*

get_device_id(индекс),

которая возвращает строковый идентификатор расширителя по его однозначному индексу (в системной таблице) или специальное значение NULL, если указанный индекс выходит за границу таблицы расширителей (это позволит неявно определять *общее количество расширителей*).

Программе может потребоваться дополнительная информация о конкретном расширителе, которую можно получить с помощью специальных функций, в которые передается фрагмент строкового идентификатора расширителя (достаточный для его уникальной идентификации) или NULL (идентифицирует наиболее производительный из имеющихся расширителей):

- **vector_max_size**(фрагмент_имени_расширителя_или_NULL) — возвращает максимально возможное количество потоков (максимальный размер группы потоков для мультипроцессора, умноженный на число мультипроцессоров) в задаче (векторизованном плане);

- **vector_max_units**(фрагмент_имени_расширителя_или_NULL) — возвращает количество мультипроцессоров на указанном расширителе;

Запуск векторного исполнения этапов плана осуществляется с помощью директивы

«**plan_group_vectorize**» «(« фрагмент_имени_расширителя_или_NULL «)»

Эта директива *запускается непосредственно из тела ППППВ/ФППВ*. Ее исполнение подразумевает следующие действия:

а) извлечение из плана *максимально возможного количества этапов, входящих в текущую группу* (но не превышающего как размера группы, так и значения, возвращенного функцией **vector_max_size**);

б) компиляцию (если она еще не была осуществлена) тела соответствующей ППППВ/ФППВ для расширителя, в идентификаторе которого встречается указанный в параметре фрагмент (или на наиболее производительном расширителе, если вместо фрагмента идентификатора расширителя указан NULL);

в) копирование спланированных данных на расширитель;

г) запуск параллельного (векторного) исполнения извлеченных этапов с помощью ППППВ/ФППВ на расширителе, при этом *каждому этапу соответствует один поток* и *все попытки изменения плана в период векторной обработки игнорируются*;

д) ожидание завершения кода на расширителе;

е) обратное копирование данных с расширителя в основную память.

При таком подходе может быть рекомендован вариант организации ППППВ/ФППВ, при котором в число параметров вводится флаг инициализации. При вызове ППППВ/ФППВ из основной программы этот флаг истинен, а при планировании новых этапов внутри процедуры/функции целесообразно спланировать его ложным. Тогда можно легко разделить ППППВ/ФППВ с помощью ветвления на две части: инициализационную, в которой одним из основных процессоров планируется серия этапов для последующего векторного исполнения и вызывается директива **plan_group_vectorize**, и параллельную, которая обрабатывает соответствующие этапы в векторном режиме на расширителе. Шаблон работы с ППППВ будет выглядеть следующим образом:

```
reenterable proc(bool init, …) {
  if (init) {
```

```
      … plan_first(false, …);
      … plan_last(false, …);
      Plan_group_vectorize(…);
   } else {
      /* Параллельная обработка этапов плана */
   }
}
/* Вызов: proc(true, …); */
```

ФППВ/ПППВ, предназначенная для работы на расширителе, может использовать следующие основные функции:

- **plan_vector_id()**, которая возвращает номер текущего потока на расширителе;

- **plan_vector_size()**, сообщающую общее количество потоков;

- **barrier(способ_обновления_памяти)**, определяющую точку барьерной синхронизации с указанным способом обеспечения когерентности видимых в разных потоках значений ячеек памяти.

Важным вопросом является *копирование данных из плана на расширитель и обратно*. Для параметров, не являющихся указателями, эта проблема решается однозначно и автоматически — копирование данных идет строго в одном направлении (из основной памяти в память расширителя), количества передаваемых байт строго равны размерам параметров, определяемых с помощью встроенной функции **sizeof**. Если же параметр является указателем, то в этом случае автоматически определить объем передаваемых данных в C/C++ классическими средствами невозможно, поэтому данная проблема перекладывается на программиста: вводятся специальные атрибуты-модификаторы для таких параметров, которые указываются в заголовке соответствующей ППВ/ФППВ и определяют размер копируемой информации (единицей является элемент данных).

Модификатор **_global(N)** определяет, что текущий параметр хранит **N** элементов по указанному адресу, имеет одно и то же значение для всех этапов плана и копируется на расширитель однократно.

Модификатор **_local(N)** определяет, что каждый текущий параметр также хранит **N** элементов, которые, однако, могут иметь разные значения, хранящиеся в виде одномерного массива (значения в нем следуют подряд, группами по **N** элементов) в основной памяти и *пересылаются для каждого этапа плана индивидуально в обоих направлениях*: на расширитель (до запуска на нем процедуры/функции) и с расширителя

(после завершения процедуры/функции на расширителе). Параметры с таким модификатором, следовательно, могут использоваться *для получения результатов с расширителя*.

Важно отметить, что параметры с модификатором **_local(N)**, вообще говоря, могут иметь варьирующийся размер данных **N** для различных этапов плана. Для решения этой проблемы вводится возможность динамического определения текущего количества элементов **N** через спланированное значение одного из прочих параметров этапа плана: в таком случае вместо конкретного N указывается ссылка вида:

«**__planned__**» «**.**» имя_параметра плана

Например, при указании модификатора **_local(__planned__.K)** для некоего параметра-указателя **P**, в ходе обмене данными с расширителем для какого-либо этапа читается значение параметра **K**, спланированное для этого этапа, которое и определяет количество копируемых элементов из текущей позиции массива данных по адресу **P**.

Приведем полный *пример небольшой программы, выполняющей простые вычисления на векторном расширителе и иллюстрирующей обмен данными с ним*.


```
#pragma plan vectorized
#include <iostream>
using namespace std;

#pragma plan common begin
#define N 5
#define threads 100
#pragma plan common end

#pragma plan gpu begin
#define addition 0.01
#pragma plan gpu end

float MUL = 3.14;
float * _OUT = NULL;

reenterable void proc(bool init, int k, _global(1) float *
mul,    _global(threads)    int    *    sizes,    int    n,
_local(__planned__.n) float * out) {
  int i;
  if (init) {
     for (i = 0; i < threads; i++) {
         plan_last(false, i, mul, sizes, sizes[i], out);
```

```
            out += sizes[i];
        }
        plan_group_vectorize(NULL);
    } else
        for (i = 0; i < n; i++) {
            *out = k*(*mul);
#ifdef __GPU__
            *out += addition;
#endif
            out++;
        }
}

int main() {
    int * sizes = new int[threads];
    int NN = 0;
    for (int i = 0; i < threads; i++) {
        sizes[i] = 1 + i % 2;
        NN += sizes[i];
    }
    _OUT = new float[NN];
    cout<<"MAX group size = "<<vector_max_size(NULL)<<endl;
    proc(true, 0, &MUL, sizes, 0, _OUT);
    for (int i = 0; i < NN; i++)
        cout<<_OUT[i]<<" ";
    cout<<endl;
    delete[] _OUT;
    delete[] sizes;
    return 0;
}
```


Осталось рассмотреть еще несколько специальных функций. Поскольку компиляция программы для расширителя выполняется внутри вызова директивы **plan_group_vectorize**, необходима функция, которая позволяла бы определить точное время, потраченное системой исключительно на последний сеанс компиляции кода для указанного расширителя (это может быть необходимо, например, в целях профилировки). Такой функцией является

last_compile_time(фрагмент_имени_расширителя_или_NULL)

Необходимо обратить особое внимание, что после вызова данной функции время компиляции сбрасывается в ноль.

При работе с OpenCL (который в данной работе является нижним слоем поддержки параллелизма на уровне векторных расширителей)

важным фактором является *разбиение множества потоков на группы* — в частности, функция **barrier** работает исключительно в пределах текущей группы. Оптимальное разбиение может являться достаточно сложным вопросом (с одной стороны, выгоден такой размер группы, который максимально полезно нагрузит мультипроцессор, с другой стороны, для поддержания такого размера может не хватить ресурсов, например, регистров), решение которого целесообразно возложить на компилятор, однако не исключается возможность предложить ему, например, *ограничение на максимальное количество потоков в группе*. Система будет пытаться выполнить это ограничение, если это будет возможно, в противном случае число потоков в локальной группе определится инфраструктурой OpenCL. Соответственно, введены две функции:

- **set_max_group_threads(N)**, которая предлагает системе ограничение на максимальное количество потоков **N** в группе;

- **get_max_group_threads()**, которая возвращает текущее ограничение. По умолчанию это ограничение равно 128.

2.2.2. Вектор и конвейер как частные случаи виртуальных топологий

Выше нами был введен формализм цепей процедур с повторным входом. Заметим, что такие цепи не только являются интуитивно понятным способом записи некоторых алгоритмов, но и позволяют явно специфицировать потенциальное наличие векторного или конвейерного параллелизма[1], согласно УЦП1 [20]. Это создает предпосылки для распараллеливания преимущественно в декларативном (иногда и чисто интуитивном) стиле, путем простого указания соответствующих директив, в отличие от более трудозатратного императивного стиля, требующего явной алгоритмизации порождения и взаимодействия параллельных процессов. При этом будем придерживаться той же предпосылки, что и ранее: наличие базовых директив распараллеливания в большинстве случаев не должно менять алгоритм работы программы.

[1] При условии, что такой параллелизм существует. Невозможно распараллелить алгоритмы, в которых обработка сегментов фактически является последовательной, то есть начать обработку сегмента невозможно ранее, чем будет завершена обработка предыдущего по цепи сегмента.

2.2.2.1. Базовые языковые средства

Для реализации *векторного параллелизма* логичным решением является ввод механизма параллельного запуска для процедур, формирующих цепь с общей инициализацией. *Конвейерный параллелизм* реализуется тем же механизмом, причем конвейерная передача данных фактически реализуется путем вставки дополнительных этапов в начало (**throw_first**) или конец (**throw_last**) плана следующей в цепи процедуры. В отличие от реализации векторного параллелизма первая процедура цепи обязательно выполняется хотя бы один раз, запуская конвейер, для прочих элементов цепи инициализация опциональна.

Задача сводится к вводу конструкции параллельного запуска процедур, входящих в цепь. Предлагается использовать тот же синтаксис, что и для обычного запуска цепи, с той разницей, что вместо ключевого слова **plan_chain** следует применять слово **plan_parallel_chain**. Любая из процедур цепи может исполняться как на одном так и на нескольких процессорах (внутреннее распараллеливание по типу «портфеля задач»), достаточно указать соответствующее их количество при записи вызова процедуры в цепи, а также может запустить иную цепь. Таким образом реализуется *вложенный параллелизм*.

Исполнение цепи процедур заканчивается, когда исчерпываются планы *всех* процедур, включенных в цепь. Подчеркнем, что в отличие от однопроцессорного варианта, где любая (за исключением первой) процедура начинала работу только после того как ее исходный план был сформирован предыдущей по цепи процедурой, в параллельном варианте любая из процедур включается в работу сразу же, как только в ее плане появится по меньшей мере один этап работ.

2.2.2.2. Запуск на кластере: общие правила

По умолчанию предполагается, что все узлы цепи (или иной виртуальной топологии, как будет показано далее) процедур будут исполняться на одной многопроцессорной/многоядерной машине. Если же предполагается запуск цепи на кластере, то выполняются следующие действия:

а) в произвольном месте программы (чаще в начале) указывается специальная разрешительная директива

#pragma plan clustered

б) перед директивой **plan_parallel_chain** указывается префикс

clustered(ссылка_на_массив_числовых_идентификаторов_или_NULL)

Если в качестве параметра префикса передан NULL, то запуск производится в стандартном одномашинном режиме (это *позволяет выбирать режим запуска динамически*). Если же передана ссылка на массив чисел, то он воспринимается как массив числовых идентификаторов узлов кластера, на которых запускается цепь. Каждый такой идентификатор, по аналогии с MPI, представляет собой число в диапазоне от 0 до C-1, где C — общее количество узлов. Для *работы с такими кластерными идентификаторами* вводятся *две новые функции*:

- **plan_cluster_size()** — возвращает число узлов,
- **plan_cluster_id()** — возвращает идентификатор текущего узла.

2.2.2.3. Каналы: одномашинные и кластерные

Очевидно, что для реализации более сложных алгоритмов параллельной обработки необходимы дополнительные средства обмена данными между процедурами, исполняемыми на различных процессорах (в разных потоках) или на различных узлах кластера. Не отрицая возможности использования традиционных средств синхронизации потоков (семафоров, блокировок и т.п.) для организации обменов данными через общую память, отметим потребность в наличии более простых и естественных средств обмена, в роли которых могут выступать *типизированные каналы передачи данных*, аналогичные по функциональности введенным в языке MC# [24]. С точки зрения программирования канал **P<T>** для данных, имеющих тип **T**, описывается двумя объектами **in<T>(P)** и **out<T>(P)**, представляющими, соответственно его вход и выход.

Классы для этих объектов генерируются автоматически, с помощью специальной конструкции

«**funnel**» «(» вид «**,**» тип_элемента_данных «)»

при работе в общей памяти или

«**cfunnel**» «(» вид «**,**» тип_элемента_данных «)»

для работы на кластере где

вид = «**in**» | «**out**»

причем **in** указывается для принимающей стороны, **out** — для посылающей стороны.

Возможны *два основных способа создания канала*: с использованием ссылки (только для канала между процессорами в общей памяти) и с использованием имени[1] (для каналов любого вида). *Создание канала с применением ссылки (являющееся более быстрым, но менее удобным способом)* требует, чтобы процедура, исполняемая на одном процессоре, создала один из «концов» канала и передала указатель на него процедуре, исполняемой на любом ином процессоре, которая создает ответный «конец». При этом первый «конец» создается конструктором без параметров (или с единственным параметром — начальным размером буфера, измеряемым в элементах данных), а для создания второго используется конструктор, принимающий ссылку. Например:

```
funnel(out, double) * put_obj = new funnel(out, double)();
…
funnel(in, double) * get_obj = new funnel(in,
double)(put_obj);
```


Создание канала с использованием имени (очевидно, менее быстрый, но более надежный и универсальный способ) не накладывает ограничений на очередность создания объектов-«концов». Они создаются на разных процессорах (или на разных машинах, соответственно) с единственным параметром конструктора — одинаковым именем создаваемого канала. Разумеется, *у различных каналов* имена должны быть *уникальны*. Например:

```
cfunnel(out, double) * put_obj = new cfunnel(out,
double)("CNL");
…
cfunnel(in, double) * get_obj = new cfunnel(in,
double)("CNL");
```

Оба вышеуказанных способа, несомненно, предполагают (в ходе основной работы) внутренние проверки на *наличие у канала обоих созданных объектов-«концов»*. Однако иногда может возникнуть необхо-

[1] Это единственный доступный способ связывания «концов» канала при работе на различных узлах кластера.

димость выполнить такую проверку явно, для этого у каждого из объектов-«концов» существует функция-метод **ready()**, возвращающая истину, если ответный объект-«конец» уже создан, иначе возвращающая ложное значение.

Чтение и запись в канал являются *блокирующими*, причем операция записи в канал — *асинхронная (пока не заполнен буфер передачи)*. Для помещения в канал одного элемента данных можно присвоить соответствующее значение объекту-концу,

out-объект «**=**» посылаемое_значение

а для чтения одного элемента можно прочитать значение с помощью конструкции вида

«*» in-объект

Для работы с более, чем одним элементом данных вызываются методы **get** или **put** соответствующего объекта:

out-объект.**put**(указатель_на_буфер, число_отправляемых_байт)

in-объект.**get**(указатель_на_буфер, число_принимаемых_байт)

Чтобы избежать внутренних ожиданий поступления/отправки данных, вместо которых можно было бы загрузить процессор полезной работой, целесообразно ввести возможность *проверки буфера приема/передачи на пустоту*. Введем для этого метод **empty()**, наличествующий у обоих объектов-«концов», возвращающий истину, если канал не содержит еще не отправленных/полученных данных, иначе возвращающий ложное значение.

Приведем простейший (исключительно иллюстративный) пример применения канала (с созданием по имени) на кластере из двух узлов:

```
#pragma plan clustered
#include <iostream>
using namespace std;
chain A() {
  for (int i = 0; i < 100; i++) {
      cfunnel(out, int) * V = new cfunnel(out, int)("CNL");
      *V = i;
      delete V;
  }
}
chain B() {
  for (int i = 0; i < 100; i++) {
      cfunnel(in, int) * V   = new cfunnel(in, int)("CNL");
      cout<<**V<<" ";
```

```
      delete V;
  }
}
int main() {
  int IDs[2] = {0,1};
  clustered(IDs) plan_parallel_chain(1, A(), B());
  return 0;
}
```

Достаточно *содержательным* примером, демонстрирующим целесообразность применения каналов, является стадийный алгоритм [32] интегрирования одномерного уравнения теплопроводности [2, 7] методом скалярной прогонки с предварительным расчетом одного из граничных значений по схеме Головичева. Формируется цепь процедур с повторным входом, работающая в конвейерном режиме, каждый i-й элемент ($i = \overline{0, N-1}$, где N — длина цепи) которой решает уравнение в одном из участков расчетной области, определяя значения температуры T в (k – i)-й временной точке, где k — номер итерации. Передача актуальных значений T_{k-i} от i-й процедуры (i + 1)-й процедуре осуществляется по конвейеру, а передача значений, необходимых для предварительного расчета, в обратном направлении осуществляется посредством канала. Этот пример ориентирован на работу в общей памяти, приведем (ввиду его большого объема) только фрагмент с ППВ:


```
const double tau = 0.1;
const double D   = 0.001;
const double h   = 0.1;
const double hh  = h*h;
const double LeftT  = 300.0;
const double RightT = 350.0;
const double InitT  = 273.0;
const int N = 4*100000+2;
const double EndTime = tau*1000;
/* Создаются в вызывающей программе */
double * Therm = NULL;
double * _L[MAX_PROCS] = {NULL};
double * _M[MAX_PROCS] = {NULL};
/* Заполняются в вызывающей программе */
int FirstIdx[MAX_PROCS]; /* Индексы начальных элементов в
Therm */
int LastIdx[MAX_PROCS]; /* Индексы последних элементов в
Therm */
```

```
chain ThermalEquation(int NProcs, double Time, fun-
nel(in,double) * outLEFT, funnel(out,double) * inLEFT)
{
 int stage = throw_stage();
 int NN = LastIdx[stage]-FirstIdx[stage]+1;
 static double Q1 = tau*D/hh;
 double * L = _L[stage];
 double * M = _M[stage];
 double Left_1, Left_0;
 int i,j,k;
 funnel(out,double) * putLEFT = new fun-
nel(out,double)(outLEFT);
 funnel(in,double) * getLEFT = new fun-
nel(in,double)(inLEFT);
 funnel(out,double) * putRIGHT = new funnel(out,double)();
 funnel(in,double) * getRIGHT = new funnel(in,double)();
 Time+=tau;
 if (stage==0 && Time<EndTime)
plan_last(NProcs,Time,NULL,NULL);
 if (stage<NProcs-1)
throw_last(NProcs,Time,getRIGHT,putRIGHT);
 L[0]=0;
 if (stage>0) {
    double Z1 = Q1;
    double Z2 = Q1;
    double Z3 = 1+2*Q1;
    *putLEFT = Therm[FirstIdx[stage]+1];
    Left_0 = **getLEFT;
    Left_1 = **getLEFT;
    M[0] =
(Left_0+Z1*Left_1+Z2*Therm[FirstIdx[stage]+1])/Z3;
 } else
    M[0] = LeftT;

 for (i=1, j=FirstIdx[stage]+1; i<NN-1; i++, j++) {
   double Z1 = Q1;
   double Z2 = Q1;
   double Z3 = 1+2*Q1;
   double V  = Z3-Z1*L[i-1];
   L[i] = Z2/V;
   M[i] = (Therm[j]+Z1*M[i-1])/V;
 }
 if (stage<NProcs-1)
    Therm[LastIdx[stage]] = **getRIGHT;
 else
```

```
    Therm[LastIdx[stage]] = RightT;

 for (i=NN-1, j=LastIdx[stage], k=0; i>=1; i--, j--, k++)
     {
      Therm[j-1] = Therm[j]*L[i-1]+M[i-1];
      if (k<2 && stage<NProcs-1) *putRIGHT = Therm[j-1];
     }
}
```

2.2.2.4. Примеры реализации алгоритмов

Рассмотрим алгоритм поиска минимального и максимального элементов в двоичном дереве. Функция **TreeMinMax** запускает цепь. На первой стадии конвейера (**_TreeMax**) выполняется обход дерева с *параллельным поиском максимума*, на второй стадии (**_TreeMin**) — *параллельный поиск минимума*. Поиск реализован с применением *редукции*, результаты помещаются во вторые параметры процедур-стадий.

Исполнение на каждой стадии дополнительно распараллелено на **PerStageNP** процессорах, таким образом всего используется 2·**PerStageNP** процессоров. Реализация такого алгоритма на OpenMP была бы менее прозрачной[1], требующей большего количества директив распараллеливания и явного обмена данными через общую память с блокировкой или с применением атомарных операций.

```
chain _TreeMax(TreeNode * Cur, reduction(max) int & Max)  {
 int Min;
 throw_last(Cur,Min); /* Планируем следующую стадию */
 /* Распараллеливаем исполнение стадии */
 if (Cur==Root) plan_group_parallelize;
 /* Планируем обход дерева */
 if (Cur->Left) plan_last(Cur->Left,Max);
 if (Cur->Right) plan_last(Cur->Right,Max);
 Max = Cur->Data;
}
chain _TreeMin(TreeNode * Cur, reduction(min) int & Min) {
 if (Cur==Root) plan_group_parallelize;
 Min = Cur->Data;
}
```

[1] Реализация конвейера с применением OpenMP будет продемонстрирована на более простом примере.

```
/* Функция поиска минимума и максимума */
int TreeMinMax(int PerStageNP, int & Min) {
 int Max = -0x7FFFFFFF;
 Min = 0x7FFFFFFF;
/* Запускаем параллельное исполнение цепи из двух стадий,
на каждой стадии задействуем PerStageNP процессоров */
plan_parallel_chain(0,_TreeMax(Root,Max)/PerStageNP,_TreeMi
n(Root,Min)/PerStageNP);
return Max;
}
```

Для сравнения приведем реализацию *неполного аналога* данного алгоритма, не сохраняющего значение поля **Data**, с применением OpenMP 3.0.

```
void _FindMinMaxTree(TreeNode * Cur, char FindMin) {
 if (Cur->Left)
    #pragma omp task untied
    _FindMinMaxTree(Cur->Left,!FindMin);
 if (Cur->Right)
    #pragma omp task untied
    _FindMinMaxTree(Cur->Right,!FindMin);
 #pragma omp taskwait
 if (Cur->Left && Cur->Right)
  if   (FindMin)   Cur->Data=min(Cur->Left->Data,Cur->Right-
>Data);
  else Cur->Data = max(Cur->Left->Data,Cur->Right->Data);
 else if (Cur->Left) Cur->Data = Cur->Left->Data;
 else if (Cur->Right) Cur->Data = Cur->Right->Data;
}
void FindMinMaxTree(TreeNode * Root, char FindMin) {
  #pragma omp parallel
  {
   #pragma omp single
   {
    _FindMinMaxTree(Root,FindMin);
   }
  }
}
```

Несколько проигрывая по общему количеству операторов (17 против 15-ти) вариант с процедурами с планированием повторного входа выигрывает у программы, написанной с применением OpenMP, по количеству директив распараллеливания (2 против 5-ти). При этом OpenMP-вариант имеет *меньшую степень параллелизма*, поскольку *не реализует*

конвейерную обработку данных, а ограничивается раздачей «портфеля задач».

Теперь рассмотрим простой пример: расчет на конвейере выражения вида R = A·X+B, где R, A, X, B — вектора чисел, для набора из пяти таких векторов. Считается, что экземпляры каждого вектора расположены в двумерных массивах, которые также (после построчного разложения) рассматриваются как одномерные. Первая стадия конвейера вычисляет произведение, вторая — сумму. Вариант с использованием процедур с повторным входом:

```
chain Stage0(int N, double * V1, double * V2, double * V3,
double * R) throw(double * V1, double MUL, double * R) {
 for (int i=0; i<N; i++, V1+=VEC_SIZE, R+=VEC_SIZE) {
    double MUL = 0.0;
    for (int j=0; j<VEC_SIZE; j++)
        MUL += (*V2++)*(*V3++);
    throw_last(V1,MUL,R);
   }
}
chain Stage1(double * V1, double MUL, double * R) {
 for (int j=0; j<VEC_SIZE; j++)
     R[j] = V1[j]+MUL;
}
/* Вызов
plan_parallel_chain(0, Stage0(5, (double * ) X1, (double *
) X2, (double*)X3, (double *)R), Stage1(NULL,0,NULL));
*/
```


Далее приведен вариант с применением OpenMP. Синхронизация потоков осуществляется путем ожидания (используется функция **Yield**, передающая на время ожидания управление иным потокам) требуемого значения разделяемой атомарной переменной **NLOCK**:

```
int NLOCK = 0;
 #pragma omp parallel num_threads(2) shared(NLOCK)
  switch (omp_get_thread_num()) {
    case 0:
      for (int i=0; i<5; i++)
        {
         R[i][0] = 0.0;
         for (int j=0; j<VEC_SIZE; j++)
             R[i][0] += X2[i][j]*X3[i][j];
         #pragma omp atomic
           NLOCK++;
         #pragma omp flush(NLOCK)
```

```
        }
      break;
    case 1:
      for (int i=0; i<5; i++)
        {
         while (NLOCK<i) Yield();
         for (int j=VEC_SIZE-1; j>=0; j--)
             R[i][j] = X1[i][j]+R[i][0];
        }
  }
```

Легко видеть, что вариант на OpenMP проигрывает как по общему числу операторов[1] (13 против 10-ти) так и по количеству директив распараллеливания (4 против 0). Этот факт хорошо демонстрирует превосходство предложенного нами формализма при описании конвейерных вычислений.

В заключение приведем пример реализации *векторного параллелизма* на задаче о покомпонентном делении вектора X1 на вектор X2:

```
chain Div(double * V1, double * V2) {
 int N = throw_stage();
 V1[N] /= V2[N];
}
/* Вызов
plan_parallel_chain(1,VEC_SIZE,Div(X1,X2));
*/
```

Для сравнения рассмотрим тот же пример на OpenMP:

```
#pragma omp parallel for num_threads(VEC_SIZE)
for (int i = 0; i<VEC_SIZE; i++)
  V1[i] /= V2[i];
```

За счет экономии оператора на декларации вспомогательной переменной — номера элемента, которая в качестве счетчика цикла внесена в заголовок цикла, реализация на OpenMP оказалась короче на два оператора, но потребовала наличия директивы распараллеливания, отсутствующей в реализации на процедуре с повторным входом.

2.2.2.4.1. Параллельная обработка рекурсивных типов данных с зависимыми подзадачами

Рассмотрим решение задачи расчета минимаксного дерева. Приведем три варианта программы. Вариант с полностью декларативным обозначением зависимостей:

[1] Оператор Yield() был отнесен к директивам распараллеливания.

```
markupdef struct TreeTag { /* Разметка типа */
  DataItem Data;
  pre_id struct TreeTag * Left;
  pre_id struct TreeTag * Right;
  state(0,1)   char Work;
} TreeNode;
chain GenMinMaxTree(TreeNode * Cur, char FindMin) {
 if (Cur==Root) plan_group_parallelize;
 throw_first(Cur,FindMin);   /*  Возможно  и  применение
throw_last, поскольку порядок обработки на следующей стадии
определяется автоматически */
 if (Cur->Left)
    plan_last(Cur->Left,!FindMin);
 if (Cur->Right)
    plan_last(Cur->Right,!FindMin);
}
chain  FindMinMaxTree(id(NULL)  struct  TreeTag  *  Cur,  char
FindMin) {
 if (Cur==NULL) plan_group_parallelize;
 else if (Cur->Left && Cur->Right)
  if   (FindMin)   Cur->Data=min(Cur->Left->Data,Cur->Right-
>Data);
  else Cur->Data=max(Cur->Left->Data,Cur->Right->Data);
 else if (Cur->Left) Cur->Data = Cur->Left->Data;
 else if (Cur->Right) Cur->Data = Cur->Right->Data;
}
/* Вызов
plan_parallel_chain(1,GenMinMaxTree(Root,0)/(nProcs/2),
FindMinMaxTree(NULL,0)/(nProcs/2));
*/
```


Необходимо пояснить, почему в данном случае был использован конвейер. Проблема состоит в том, что обход дерева начинается сверху вниз, а реальная обработка данных вершин в соответствии с зависимостями по данным — снизу вверх. Поэтому, если попытаться реализовать алгоритм в одну стадию, руководствуясь правилами, данными введенными нами механизмами обозначения зависимостей по данным, сталкиваемся с противоречием, которое состоит в том, что невозможно выполнить обход дерева сверху вниз, поскольку при таком порядке обработки этапы плана, соответствующие верхним узлам дерева, просто не могут выполниться, сгенерировав этапы, соответствующие нижележащим узлам, до этих узлов. В данном случае решением является двухстадийный алгоритм, где на первой стадии производится обход дерева с генерацией

плана этапов работ для следующей стадии, на которой идет обработка с учетом зависимостей по данным (снизу вверх по дереву)[1].

Вариант с расширенным частично декларативным обозначением зависимостей отличается от предыдущего лишь отсутствием разметки типа и модифицированным заголовком процедуры — второго звена конвейера:

```
chain FindMinMaxTree(id(NULL) TreeNode * Cur, char FindMin)
 if((!id->Left||id->Left->Work)&&(!id->Right||id->Right-
>Work))
 set(id->Work=1)
 reset(id->Work=0)
 depends(id->Left,id->Right)
```

Очевидно, что второй вариант более гибок, но и является более громоздким. В заключение рассмотрим решение той же задачи с использованием OpenMP 3.0:

```
void _FindMinMaxTree(TreeNode * Cur, char FindMin) {
 if (Cur->Left)
    #pragma omp task untied
    _FindMinMaxTree(Cur->Left,!FindMin);
 if (Cur->Right)
    #pragma omp task untied
    _FindMinMaxTree(Cur->Right,!FindMin);
 #pragma omp taskwait
 if (Cur->Left && Cur->Right)
  if   (FindMin)   Cur->Data=min(Cur->Left->Data,Cur->Right-
>Data);
  else Cur->Data=max(Cur->Left->Data,Cur->Right->Data);
 else if (Cur->Left) Cur->Data = Cur->Left->Data;
 else if (Cur->Right) Cur->Data = Cur->Right->Data;
}
void FindMinMaxTree(TreeNode * Root, char FindMin) {
  #pragma omp parallel
  {
```


[1] Еще одним решением (на одной процедуре с планированием повторного входа) могло бы стать разделение плана этапов работ на две группы с помощью маркера таким образом, чтобы планирование обхода производилось бы в первой группе без учета зависимостей по данным, а обработка — во второй группе с учетом таких зависимостей. Однако это потребовало бы модификации формализма ППП, а именно ввода понятия групп плана с учетом зависимостей и без их учета, что сводилось бы, например, ко вводу дополнительного типа маркера (отсутствие учета зависимостей по данным) и соответствующих директив **plan_group_free_first**, **plan_group_free_last**. В соответствии с принципом бритвы Оккама в данной работе было принято решение не прибегать к такому приему, ограничившись конвейерным вариантом.

```
    #pragma omp single
    {
     _FindMinMaxTree(Root,FindMin);
    }
   }
}
```

В данном случае решение на процедурах с планированием повторного входа при большей декларативности содержит и большее количество операторов (18 против 15-ти при использовании OpenMP) при равном количестве директив (5 штук), связанных с распараллеливанием.

2.3. Виртуальные топологии: произвольные схемы обмена данными

Как было показано выше, с помощью ППВ можно адекватно построить вычислительные топологии «вектор» и «конвейер», рассматривая процедуры как узлы топологии и применяя конструкции вида **throw_first/throw_last** для передачи данных. На базе этого подхода можно строить и *более сложные топологии*, если ввести возможность удобной уникальной идентификации произвольного (в рамках описанных связей) направления передачи, задействовав под его указание, например, *один из параметров* **throw**-конструкции. На передающей стороне в этот параметр будет передаваться идентификатор приемника, а на принимающей стороне он будет содержать идентификатор передатчика. В таком случае останется только ввести достаточно удобные и мощные средства описания набора узлов топологии и связей между ними, разрешив как их *простое списковое перечисление*, так и *автоматизированную генерацию по некоторым, например, декларативным правилам*. Это в значительной степени решит, в частности, проблему *масштабируемости топологий*.

Следует отметить, что произвольная виртуальная топология в общем случае, видимо, будет отличаться от рассмотренных нами ранее топологий тем, что для нее далеко не всегда возможно обеспечить непараллельный режим работы (последовательную обработку данных), например, если имеют место циклические связи.

Поэтому, для рассматриваемых нами далее произвольных виртуальных топологий *постулируем работу исключительно в параллельном режиме*.

2.3.1. Базовые языковые средства

В соответствии с вышесказанным, необходимо ввести некоторый базовый (списковый) синтаксис описания произвольной топологии и детализировать семантико-синтаксические особенности идентификации направлений передачи. При этом общая идеология работы произвольной топологии практически не отличается от идеологии работы параллельной цепи процедур, появляется лишь возможность указания направлений передачи этапов в план некоторой процедуры-узла. Все функции, работавшие с цепями, будут работать и с произвольными топологиями.

Важной *особенностью* работы произвольной топологии является *необходимость явного завершения ее работы* с применением специальной функции **plan_topology_quit()**, которая может быть вызвана на любом узле/узлах топологии. Ее появление оправдывается стремлением к минимизации обменов данными между элементами топологии (исключается необходимость сообщать другим элементам о своем «локальном» периодическом окончании/возобновлении работы, что требуется для реализации алгоритма согласованного выхода), особенно при работе в кластерном варианте.

Базовый синтаксис описания топологии включает явный список ее связей (узлы топологии при этом описываются *неявно*), при этом возможно перечисление как элементарных связей между двумя узлами, так и линейных последовательностей связей. Такие элементы перечисления, для упрощения, имеют практически такой же формат, что и цепи ППВ, используя ключевое слово **plan_parallel_chain**. Есть лишь несколько *существенных отличий*:

а) предполагается, что каждая ППВ топологии всегда имеет инициализационный этап (это наиболее общий случай), поэтому начальный логический параметр из указаний **plan_parallel_chain** исключен:

б) поскольку одна и та же ППВ может одновременно представлять несколько узлов, введена дополнительная *индексация* (начиная с

единицы, причем *единица подразумевается по умолчанию*) различных узлов, *описываемых одной ППППВ*, индекс может указываться в квадратных скобках непосредственно после идентификатора ППППВ;

в) указание количества повторений одной ППППВ в рамках **plan_parallel_chain** не допускается, поскольку несколько затрудняет указание уникальных индексов ППППВ в таком случае, провоцируя различные логические ошибки;

г) поскольку для успешной работы топологии необходимо, прежде всего, убедиться в отсутствии в ней незапланированных (внесенных по ошибке) циклов, предполагается, что *указание любого цикла* всегда представляет собой такой перечень связей, по меньшей мере одна связь которого специфицирована с применением описателя **plan_parallel_reverse** вместо **plan_parallel_chain**. Предполагается, что *по прямым связям* (**plan_parallel_chain**) граф топологии должен представлять собой *сеть или группу сетей*. Этот факт *всегда* проверяется средой исполнения Planning C-программ, при наличии цикла по прямым связям генерируется ошибка исполнения.

Дадим формальное описание *базового синтаксиса топологии*, предполагая, что все описываемые элементы могут разделяться *пробелами*:

топология = [clustered_префикс] «**plan_topology**» «**{**»
(описание_линии | описание_реверса) «**;**»
{ (описание_линии | описание_реверса) «**;**»}
«**}**» [«**/**» максимальное_число_ядер] «**;**»

описание_линии = «**plan_parallel_chain**» «(» последовательность_ППППВ «)»

описание_реверса = «**plan_parallel_reverse**» «(» ППППВ_узел («**->**» | «**,**») ППППВ_узел «)»

последовательность_ППППВ = ППППВ_узел {(«**->**» | «**,**») ППППВ_узел}

ППППВ_узел = идентификатор_процедуры [«**[**» индекс «**]**»] «(»
[инициализационные_параметры] «)»

индекс = числовая_константа

Отметим, что возможен запуск топологии как на кластере так и в одномашинном варианте. Для этого используются те же средства (префикс **clustered** с параметром), что и для запуска цепи. Еще одной интересной особенностью топологий является возможность их *вложенности*,

то есть на произвольной ППВ, запущенной в одной топологии, можно стартовать дополнительную топологию. При этом взаимодействие между такими топологиями можно организовать, например, *с применением каналов*.

Детализируем *идентификацию направлений передачи*. Используется следующее правило: если узел-ППВ А имеет единственную выходную связь в узел-ППВ В, то передача этапа в план В осуществляется точно также, как и для простых цепей процедур. Если же А имеет более одной выходной связи, то каждый узел-ППВ из числа выходных должен иметь в интерфейсе специальный параметр предопределенного типа **input_proc**. Тогда справедливы следующие утверждения:

а) передающая ППВ может указать в таком параметре *полный идентификатор приемника* (значение типа **input_proc**) в формате

имя_ППВ-приемника [«[» индекс_ППВ «]»]

где индекс по умолчанию равен единице;

б) в качестве пустого значения типа **input_proc** может указываться специальная константа **empty_proc**;

в) принимающая ППВ может прочитать из такого параметра полный идентификатор передатчика и разобрать его на две составляющие (адрес ППВ и ее индекс) с помощью функций:

-

input_addr(полный_идентификатор_передатчика_типа_input_proc**)**, возвращающей адрес передающей ППВ, который можно сравнить с явным адресом произвольной ППВ, определяемый с помощью конструкции **&ППВ**;

-

input_index(полный_идентификатор_передатчика_типа_input_proc**)**, возвращающей индекс передающей ППВ;

г) любая ППВ может получить свой собственный индекс, вызвав функцию **plan_linear_num()**.

Предполагается, что среда исполнения программы динамически определяет, является ли указанное направление передачи данных разрешенным в рамках текущей топологии и, если направление к таковым не относится, генерирует ошибку исполнения.

Учитывая, что предложенные нами средства позволяют описывать достаточно сложные топологии (которые, как будет показано далее мо-

гут даже генерироваться по определенным декларативным правилам), актуальным представляется ввод *функции получения оперативной информации о полных идентификаторах (типа input_proc) входных или выходных ППП по отношению к некоей опорной ППП*. Для этого введена функция

plan_neighbours(флаг_направления**,** полный_идентификатор_опорной_ППП**,**
адрес_переменной_количества**,** адрес_массива_идентификаторов**)**

Первый параметр (флаг_направления) должен иметь истинное значение при извлечении информации о входных связях, ложное — о выходных. Третий параметр указывает на целочисленную переменную, принимающую количество связей, а в четвертый параметр передается указатель на начало массива (из элементов типа **input_proc**), который будет заполнен требуемой информацией о полных идентификаторах связанных ППП.

Приведем пример небольшой программы, в которой создается топология из трех узлов: управляющего A и двух рабочих B и C. Рабочие узлы объединены в цикл, по которому с помощью конструкций throw передается увеличивающееся число. Запуск циклического обмена и информирование о необходимости прекращения работы топологии осуществляется управляющей ППП. Поскольку все узлы имеют только одну выходную связь, указание **input_proc**-параметров в **throw**-конструкциях не требуется[1]. Отметим, что данная программа может использоваться для сравнения производительности разных внутренних реализаций основных механизмов Planning C (чем больше выводимое программой на экран число, тем выше производительность).

```
#include <iostream>
using namespace std;
bool stop;
chain A(bool init) throw(bool init, int N) {
  stop = false;
  throw_first(false, 1);
  Sleep(2000); /* Задержка на 2 секунды */
  stop = true;
}
chain B(bool init, int N) throw(bool init, bool _stop, int
N) {
```

[1] Адресация такого рода будет рассмотрена далее, на более сложных примерах.

```
  if (!init) {
     if (stop) throw_first(false, true, N);
     else throw_first(false, false, N+1);
  }
}
chain C(bool init, bool _stop, int N) throw(bool init, int
N) {
  if (!init) {
     if (_stop) {
        cout<<N;
        plan_topology_quit();
     } else throw_first(false, N+1);
  }
}
int main() {
  plan_topology {
    plan_parallel_chain(A(true)->B(true,0)-
>C(true,false,0));
    plan_parallel_reverse(C(true,false,0)->B(true,0));
  }/3;
  return 0;
}
```


2.3.2. Макромодули: декларативное описание и масштабирование топологий

Основным недостатком представленного выше «спискового» описания топологии является его полная статичность и, как следствие, *немасштабируемость*. На практике более часто встречаются задачи, для которых определен *общий принцип установления связей топологии*, а число задействованных в такой топологии узлов определяется *динамически*, либо на этапе исполнения, либо на этапе компиляции, предшествующей исполнению. В первом случае топология обычно формируется *императивно*, путем исполнения фрагмента основной программы, вызывающей некие специальные функции, устанавливающие связи между всеми имеющимися узлами. Такой вариант, реализованный, например, в стандарте MPI и в библиотеке TBB, в некоторых нетривиальных случаях (например, при реализации топологии «гиперкуб») может отличаться достаточно высокой алгоритмической сложностью. Во втором случае

возможен *декларативный* подход[1], при котором для построения связей топологии используются некие логические правила.

Выберем второй подход, как более наглядный и, в ряде случаев, потенциально более мощный. Возможно, наиболее простым (для программиста) вариантом будет прямая генерация рассмотренного выше «спискового» описания топологии *логическими правилами*. Это естественным образом приводит нас к идее *генерации*, по меньшей мере, дескриптивных фрагментов основной программы в ходе прямого применения логических правил, записанных, например, *на языке Prolog*, имеющего бесплатную и свободно распространяемую реализацию GNU-Prolog. Соответственно, нам необходимо предложить синтаксические средства, которые позволяли бы описывать топологию как совокупность статических и динамических (генерируемых на одной из стадий компиляции в ходе применения предикатов GNU-Prolog) элементов. Это в полной мере *решит проблему масштабирования топологий* и, возможно, снизит вероятность внесения ошибок в их описания.

Введем понятие *дедуктивного макромодуля*, который оформлен в виде специального программного блока и имеет параметры, в зависимости от которых им генерируется фрагмент программного кода. Соответствующий код будет вставлен в программу в точке обращения к макромодулю, в котором будут указаны конкретные значения его параметров. Предполагается, что макромодуль будет генерировать код на этапе компиляции, точнее, на стадии препроцессинговой обработки. Соответственно, это накладывает определенные *ограничения на возможные значения его параметров* — это должны быть выражения, которые можно вычислить на этапе препроцессинга: предположим, что это выражения, содержащие исключительно именованные и неименованные константы, в том числе те, которые формируются в результате классических макроподстановок C/C++.

Предлагается следующий *синтаксис декларации дедуктивного макромодуля* (все элементы в описании могут разделяться пробелами):

«**#**» «**def_module**» «(» префиксная_строка «)» имя_топологии «(»

---

[1] Строго говоря, он возможен и в первом случае. Тогда исполняемая программа должна либо интегрировать интерпретатор логических правил, либо иметь его в виде внешнего приложения/библиотеки. Соответственно, это может быть достаточно громоздким и/или неудобным решением.

[имя_параметра] {«**,**» имя_параметра} «)» «**{**«
(предикат | цель | произвольный_Planning_C_код)
{(предикат | цель | произвольный_Planning_C_код }
«**}**» «**;**»
предикат = «@» имя_предиката [«(» переменные_предиката «)»] [«**:-**»
GNU_PROLOG_выражение] «**.**»
цель = «@» «**goal**» «**:-**» GNU_PROLOG_выражение «**.**»
префиксная_строка = «**plan_topology**»

Здесь произвольный_Planning_C_код должен представлять собой фрагмент синтаксически корректного языкового выражения, не содержащего символа «@». В рассматриваемом случае топологии это может быть описатель ее статического элемента (описание_линии или описание_реверса, см. выше). GNU_PROLOG_выражение может содержать вызовы любых предикатов GNU Prolog, в том числе генерирующих консольный вывод — результаты этого вывода и будут использоваться в качестве *сгенерированных фрагментов кода*. В большинстве случаев вывод будет генерироваться предикатом ***write*** *и/или специальными библиотечными предикатами, создающими строку-описатель элементарной (между двумя узлами-ППП В) связи*:

- двухместным предикатом **plan_parallel_chain(A, B)**, генерирующим строку-описатель прямой связи вида «plan_parallel_chain(A->B);»;

- двухместным предикатом **plan_parallel_reverse(A, B)**, генерирующим строку-описатель реверсной связи вида «plan_parallel_reverse(A->B);».

Обращение к макромодулю-топологии имеет формат:

имя_топологии «(» [значение_параметра] {«**,**» значение_параметра} «)»
[«**/**» число_ядер] «**;**»

В точке обращении к макромодулю компилятором выполняются следующие действия:

а) вычисляются все параметры обращения;

б) значения параметров подставляются в текст модуля вместо соответствующих лексем - имен параметров;

в) из текста модуля исключаются все предикаты, из которых формируется текст логической GNU Prolog-программы;

г) фрагмент модуля, содержащий какую-либо из целей, заменяется результатом доказательства этой цели (то есть блоком выведенных на консоль в ходе доказательства строк) в контексте сформированной логической GNU Prolog-программы;

д) в программу на Planning C вместо обращения к макромодулю вставляется код, содержащий префиксную строку и результирующий текст модуля, обрамленный фигурными скобками.

Необходимо детализировать возможные *типы параметров*. Каждый *параметр* (после выполнения всех макроподстановок и подстановок значений констант, определенных в программе с помощью ключевого слова **const**) должен быть *константным выражением, содержащим только неименованные константы*. Такое выражение может быть числом/числовым выражением, или строкой (заключенной в апострофы), или списком, который может содержать числа, строки и другие списки. Числовые выражения вычисляются, применительно к результирующим значениям действуют следующие простые правила:

- целые числа так и считаются целыми;

- близкие к нулю вещественные константы считаются целочисленными нулями;

- близкие к целым вещественные значения считаются соответствующими целыми (с округлением);

- прочие значения считаются вещественными.

Развернутый в константное выражение *параметр распознается по следующим правилам*:

а) если он начинается с «[», то это *список*, который передается в макромодуль без изменения вплоть до «]» *с учетом сбалансированности по вложенным парам квадратных скобок;*

б) если он начинается с «'», то это строка, которая передается без изменения вплоть до закрывающего апострофа «'» с учетом наличия в строке возможных пар апострофов, представляющих апостроф, являющийся одним из символов строки;

в) иначе делается попытка распознать параметр как число/числовое выражение.

В качестве примера приведем текст макромодуля, генерирующего *топологию «гиперкуб»* с **NDims** измерений, в каждом из которых **NN** элементов, причем каждый элемент является процедурой A, для которой

указывается инициализационный список параметров **Args**. Заметим, что хотя текст модуля достаточно объемен, реализация соответствующей топологии средствами какого-либо императивного языка высокого уровня, вероятно, была бы еще более сложной и непрозрачной, что связано с неочевидностью реализации k-вложенного цикла, где величина k заранее неизвестна.

```
#def_module(plan_topology) hypercube(NDims,NN,Args) {
   @getidx(_,[],_,1):-!.
   @getidx(N,[H|T],Mul,IND):-Mul1 is Mul*N,
getidx(N,T,Mul1, IBase), IND is IBase+(H-1)*Mul.

   @try_link(L,HB,H,R,I,N):-
     H>0, N1 is N+1, H<N1, !,
     append(L,[H|R],I1),
     getidx(N,I,1,IND),
     number_atom(IND,AIND), atom_concat('A[',AIND,A1),
atom_concat(A1,']',A2), atom_concat(A2,Args,A3),
     getidx(N,I1,1,IND1),
     number_atom(IND1,AIND1), atom_concat('A[',AIND1,A4),
atom_concat(A4,']',A5), atom_concat(A5,Args,A6),
     (
      HB<H -> plan_parallel_chain(A3,A6);
plan_parallel_reverse(A3,A6)
     ),
     write(' ').
   @try_link(_,_,_,_,_,_):-!.

   @make_links(I,N):-
     append(L,[H|R],I),
     HM is H-1,
     HP is H+1,
     try_link(L,H,HM,R,I,N), try_link(L,H,HP,R,I,N),
     fail.
   @make_links(_,_):-!.

   @hyper(0,I,_,N):-!, make_links(I,N), !.
   @hyper(_,_,0,_):-!.
   @hyper(DIM,INDS,I,N):-
     append(INDS,[I],NEXTINDS),
     DIM1 is DIM-1,
     hyper(DIM1,NEXTINDS,N,N),
     I1 is I-1,
     hyper(DIM,INDS,I1,N).
```

```
   @goal:-hyper(NDims,[],NN,NN).
};
/* Пример фрагмента программы с обращением к макромодулю
для генерации куба 9×9×9: */
chain A(input_proc Src, int Dims, int N, int K) { … }
int main() {
  const int Dims = 3;
  const int N = 3;

  hypercube(Dims, N, '(empty_proc, Dims, N, 0)');
  return 0;
}
```

Рассмотрим еще один более простой, но полный *пример установки топологии «труба» из **N** элементов и работы с ней*. По «трубе», сначала вправо (до конца), а затем влево (также до конца) передается число **9**, которое после завершения всех пересылок выводится на экран.

```
#include <iostream>
using namespace std;
chain A(bool init, input_proc Src, int N, int K) throw(bool
init, input_proc Src, int N, int K) {
  if (init) {
     if (plan_linear_num() == 1)
        throw_first(false, A[plan_linear_num()+1], N, 9);
  } else {
     if (input_index(Src) < plan_linear_num()) {
        if (plan_linear_num() < N) {
           throw_first(false, A[plan_linear_num()+1], N,
K);
        } else
           throw_first(false, A[plan_linear_num()-1], N,
K);
     } else {
        if (plan_linear_num() == 1) {
           cout<<K<<endl;
           plan_topology_quit();
        } else
           throw_first(false, A[plan_linear_num()-1], N,
K);
     }
  }
}
#def_module(plan_topology) tube(N) {
  @body(NN) :-
    N1 is NN-1,
```

```
    for(I, 1, N1),
      number_atom(I, IND1),
      atom_concat('A[',IND1,A1),
atom_concat(A1,'](true,empty_proc,N,0)',A2),
      I1 is I+1,
      number_atom(I1, IND2),
      atom_concat('A[',IND2,A3),
atom_concat(A3,'](true,empty_proc,N,0)',A4),
      plan_parallel_chain(A2,A4),
      plan_parallel_reverse(A4,A2),
    fail.
  @body(_):-!.
  @goal :- body(N).
};
int main() {
  const int N = 5;
  tube(N)/N;
  return 0;
}
```


2.3.3. Сторонние эффекты применения макромодулей: порождающее программирование

Как уже упоминалось выше, макромодуль-топология может содержать произвольный статический Planning C-код, в который встраиваются динамически генерируемые целевыми предикатами фрагменты кода, которые в общем случае также могут быть произвольными. Поэтому уже возможно применение макромодулей для параметризованной генерации синтаксических конструкций, включающих префиксованный блок (П-блок) в фигурных скобках: деклараций структур, классов, уний, функций.

Однако весьма интересной представляется идея генерировать с помощью макромодулей *любой Planning C-код*, реализовав, таким образом, *полноценное порождающее программирование*, которое может быть применено для адаптивного и компактного решения задач тонкой настройки программы на различные входные параметры, например, для адаптивного выбора наиболее подходящих шаблонов оптимального распараллеливания в зависимости от характеристик конкретной вычислительной системы (например, используя подход, данный в работе [17]).

Учитывая, что механизм порождения, в нашем случае, базируется на схеме логического вывода, *могут решаться и еще более сложные, интеллектуальные задачи*, например, оперативной генерации решающего задачу алгоритма.

Еще раз приведем общую схему порожденного макромодулем фрагмента:

префиксная_строка «{» сгенерированный_макромодулем_код «}».

Очевидно, что если ввести синтаксические средства, позволяющие убрать префиксную строку (возможность ее изменения заложена в макромодуль изначально) и обрамляющие скобки, то задача порождения принципиально произвольного Planning C-кода будет решена. При этом мы сохраним полную совместимость с введенными выше базовыми средствами описания виртуальных топологий. Соответственно, определим *два специальных предиката, управляющих порождением кода*:

- **prefix_off**, выключающий префиксную строку,
- **brackets_off**, выключающий обрамляющие фигурные скобки.

Эти предикаты имеют глобальный для всего макромодуля эффект, соответственно они могут быть вызваны в любом из предикатов/целей модуля.

Формат о*бращения к макромодулю общего вида*:

имя_модуля «(« [значение_параметра] {«,» значение_параметра} «)»

Передача в макромодуль общего вида произвольного однострочного параметра, являющегося законченным Planning C-оператором (в том числе вызовом функции или лямбда-функции), позволяет, например, генерировать вложенные циклы, для которых указанный параметр представляет собой тело цикла. Возможен и иной подход, при котором макромодуль генерирует только серию заголовков цикла — в этом случае используется П-блок с опущенными префиксом и скобками. Далее приведен пример, содержащий соответствующий макромодуль **big_loop**, который вызывается для генерации тройного вложенного цикла (i = 0..2, j = 0..3, k = 0..4).

```
#include <iostream>
using namespace std;
#def_module() big_loop(Vars, Lows, Highs) {
  @goal:-brackets_off.
  @loop([],[],[]):-!.
```

```
  @loop([V|VT],[L|LT],[H|HT]):-
    write('for (int
'),write(V),write('='),write(L),write(';'),

write(V),write('<='),write(H),write(';'),
                   write(V),write('++) '),
    loop(VT,LT,HT).
  @goal:-loop(Vars,Lows,Highs).
};
int main() {
  big_loop(['i','j','k'],[0,0,0],[2,3,4])
    cout<<"1";
  cout<<endl;
  return 0;
}
```

Еще один пример иллюстрирует применение макромодуля **tree_node**, который генерирует декларацию типа элемента n-арного дерева ($n \geq 2$).

```
#def_module(struct) tree_node(ID, Type, Name, Arity) {
  @goal:-brackets_off.
  @goal:-write(ID). {
    @goal:-write(Type),write(' '),write(Name),write(';').
    @refs(2):-write('struct '),write(ID),write(' * Left;'),
              write('struct '),write(ID),write(' * Right;'),
              !.
    @refs(N):-write('struct '),write(ID),write(' *
Children['),write(N),write('];'),!.
    @goal:-refs(Arity).
  }
};
tree_node('node2', 'int', 'Data', 2);
tree_node('node4', 'int', 'Data', 4);
int main() {
  struct node2 BinLeaf;
  struct node4 QuadLeaf;
  BinLeaf.Data = 1;
  BinLeaf.Left = NULL;
  BinLeaf.Right = NULL;
  QuadLeaf.Data = 2;
  for (int i = 0; i < 4; i++)
      QuadLeaf.Children[i] = NULL;
  return 0;
}
```

Следует отметить, что определение макромодуля может содержать обращения к иным макромодулям. Таким образом, реализованы вложенные макромодули, с помощью которых можно (в некоторых случаях) сократить общий объем модулей.

2.4. Классические средства параллельного программирования

В данном пункте будут кратко представлены классические средства параллельного программирования, встречающиеся (в том или ином составе) в большинстве соответствующих языков и расширений.

2.4.1. Общие (в том числе незначительно специализированные) средства для систем с различной архитектурой

Как уже упоминалось выше, цепи процедур и виртуальные топологии могут работать (в зависимости от наличия префикса **clustered**) как в системах с общей памятью, так и на кластерных системах. Актуальна задача унификации алгоритмов соответствующих ППП путем применения семантически одинаковых (и лишь незначительно отличающихся по синтаксису) средств поддержки параллельного программирования, в качестве которых целесообразно выбрать *классические средства для работы в общей памяти*: атомарные глобальные данные, семафор (являющийся более общим случаем блокировки) и барьер.

Атомарные глобальные данные для случая SMP-систем представлены ячейками оперативной памяти, для случая же кластерных систем введен специальный объект класса **cvar**(тип_элемента), представляющий в общем случае массив из N элементов указанного типа (параметр N указывается в вызове конструктора). В частном случае, когда N = 1, для объекта класса **cvar** определены операторы присваивания (=, +=, -=), инкремента (++) и декремента (--), извлечения данных (*). В общем случае (N > 1) предполагается загрузка полного массива данных в объект методом

put(указатель_на_буфер, размер_данных_в_байтах)

и извлечения их из объекта методом

get(указатель_на_буфер, размер_данных_в_байтах).

Поэлементная работа с массивом данных **cvar** малоэффективна в условиях кластерной реализации и, по этой причине, не реализована.

Необходимо особо подчеркнуть, что, поскольку любой блок атомарных глобальных данных един для всех узлов кластера, операции его *создания* (вызов конструктора) и *уничтожения* (явный или неявный вызов деструктора) *являются групповыми* и должны *синхронно выполняться на всех узлах*.

Семафор представлен специальным объектом типа **plan_sem_t** для SMP-систем или **plan_csem_t** для кластеров. Сразу подчеркнем, что, как и в случае глобальных данных, *кластерный семафор* требует *синхронной инициализации/деинициализации на всех узлах кластера*.

Для *создания SMP-семафора* декларируется соответствующая переменная, после чего вызывается функция инициализации

plan_sem_init(указатель_на_семафор, начальное_значение_счетчика_семафора).

Для *создания кластерного семафора* используется аналогичная методика, но вызывается функция **plan_csem_init**.

Декремент счетчика семафора (с возможным ожиданием, если счетчик равен нулю) выполняется функциями **plan_sem_wait/plan_csem_wait**, в которые передается указатель на соответствующий семафор. *Инкремент счетчика* реализован функциями **plan_sem_release/plan_csem_release**, в которые также передается указатель на семафор.

Деинициализация семафора выполняется функциями **plan_sem_destroy/ plan_csem_destroy**, в которые также передается указатель на семафор.

Применение *барьера*, в отличие от атомарных глобальных данных и семафора не требует явной работы с переменными разных типов, поэтому для барьеров удалось разработать *абсолютно идентичные для разных архитектур средства*. Единственный вопрос, который требует проработки — определение областей действия барьера. Данный вопрос может быть решен вводом понятия *группы узлов топологии*.

Пусть по умолчанию все узлы топологии объединены в группу с идентификатором **topology**. Можно ввести *функцию регистрации узла в группе*

plan_register(идентификатор_группы**),**

которая включает узел, вызвавший функцию, в группу с указанным идентификатором (если группы не существовало, то она автоматически будет создана). Тогда для создания группы достаточно вызвать вышеуказанную функцию на всех требуемых узлах. Предполагается, что вызовы этой функции не синхронизируются, поэтому, для обеспечения полной уверенности в том, что группа создана полностью и соответствующие регистрационные сообщения дошли до всех узлов топологии, следует обеспечить внешнюю синхронизацию узлов, например, вызвав барьер по всей топологии.

Функция установки барьера имеет вид:

plan_registered_barrier(идентификатор_группы**)**

Заметим, что различные запущенные топологии не могут иметь кросс-групп, по соображениям упрощения языка и повышения безопасности программирования. Соответственно, *синхронизация различных топологий с помощью одного барьера невозможна*.

Приведем пример, в котором на кластере из **N** узлов запускается топология «клика», в рамках которой выполняется передача данных по схеме «каждый-каждому». Количество вызовов узлов-ПППВ регистрируется в атомарной глобальной переменной **DATA**. Чтобы дождаться завершения подсчета **DATA**, узлы синхронизируются с помощью барьера, после чего один из узлов выводит накопленное значение **DATA** на экран.

```
#pragma plan clustered
#include <iostream>
using namespace std;
cvar(int) * DATA = NULL;
int Counter;
chain A(input_proc Src, int N) {
  int id = plan_linear_num()-1;
  if (Src == empty_proc) {
     (*DATA)++;
     Counter = 1;
     for (int i = 0; i < N; i++)
         if (i != id)
            throw_first(A[i+1], N);
```

```
  } else {
     (*DATA)++;
     Counter++;
     if (Counter == N) {
        plan_registered_barrier(topology);
        if (id == N-1) {
           cout<<**DATA<<endl;
           plan_topology_quit();
        }
     }
  }
}
#def_module(plan_topology) clique(N) {
  @inner_loop(_,0):-!.
  @inner_loop(I,J) :-
    (I==J ->
      true;
      (
       number_atom(I,IND1),
       atom_concat('A[',IND1,B1),
atom_concat(B1,'](empty_proc,N)',B2),
       number_atom(J,IND2),
       atom_concat('A[',IND2,B3),
atom_concat(B3,'](empty_proc,N)',B4),
       (I<J ->
        plan_parallel_chain(B2,B4);
        plan_parallel_reverse(B2,B4)
       ),
       plan_parallel_reverse(B4,B2)
      )
    ),
    J1 is J-1,
    inner_loop(I,J1).
  @outer_loop(0):-!.
  @outer_loop(I) :-
    inner_loop(I, N),
    I1 is I-1,
    outer_loop(I1).
  @goal :- outer_loop(N).
};
int main() {
  const int N = 5;
  DATA = new cvar(int)(1);
  if (plan_cluster_id() == 0)
     *DATA = 0;
```

```
  int IDS[N];
  for (int i = 0; i < N; i++)
      IDS[i] = i;
  clustered(IDS) clique(N)/N;
  delete DATA;
  return 0;
}
```

2.4.2. Специфические средства для систем с общей памятью

В данном пункте будут описаны средства, которые по тем или иным причинам неэффективны для систем с разделяемой памятью, но могут, тем не менее, повысить эффективность реализации программ для систем с общей памятью.

2.4.2.1. Классические средства

В качестве классических средств целесообразно выбрать такие достаточно часто используемые и еще не включенные нами в состав языка формализмы как *критическая секция и блокировка*[1]. Предлагается реализовать их в эквивалентном OpenMP по семантике стиле.

Критическая секция описывается конструкцией

«**plan_critical**» «(» идентификатор_секции «)» «{» [операторы] «}»

При этом считается, что блоки, для которых указан единый идентификатор, относятся к одной и той же критической секции.

Блокировки представлены переменными типа **plan_lock_t**, Для работы с ними, по аналогии с OpenMP, предлагаются четыре функции, каждая из которых принимает единственный параметр — указатель на блокировку: инициализации **plan_init_lock**, деинициализации **plan_destroy_lock**, установки блокировки **plan_set_lock**, снятия блокировки **plan_unset_lock**.

[1] Блокировка легко эмулируется уже используемым нами семафором, но в данном случае предполагается, что специализированная реализация блокировки может быть существенно более эффективной в случае SMP-систем, поэтому ее применение вполне оправдано.

Ввиду общеизвестности данных средств, примеры их использования здесь не приводятся.

2.4.2.2. Неклассическое параллельное программирование: транзакционная память

Транзакционная память [33] является сравнительно новым средством параллельного программирования и, с теоретической точки зрения, позволяет существенно его упростить за счет устранения необходимости в явных синхронизациях процессов (с помощью барьеров) и обеспечении монопольного доступа (с применением семафоров, блокировок, критических секций и прочих подобных средств). Однако следует заметить, что программирование с транзакционной памятью по определению дает во многих случаях менее эффективные программы, поскольку подразумевает многократное (избыточное) выполнение пересогласуемых (откатываемых) секций кода. Тем не менее, это достаточно интересное средство, особенно удобное для исследователей, не являющихся профессиональными программистами.

Не предлагая каких-либо существенных семантических нововведений в данной области, ограничимся, преимущественно, изложением *синтаксиса* вводимых конструкций. В Planning C, как и в GNU C/C++, транзакционным блоком может являться или процедура (в нашем случае это ФППВ/ППППВ) или локальный фрагмент программного кода. *Локальный блок* описывается следующим образом

«**plan_atomize**» «{» [операторы] «}»

и трактуется стандартно, как транзакция, результат выполнения которой либо принимается, либо, в случае возникновения конфликтов, откатывается с повторным исполнением блока. В случае, когда транзакционный режим может быть включен для ППППВ/ФППВ, справедлива следующая теорема:

Теорема о применении транзакционной памяти (ТА2). Допустимо исполнение группы этапов плана параллельно, рассматривая запуск ППППВ/ФППВ для каждого этапа группы как транзакцию. Достаточно ввести директиву запуска группы этапов плана в параллель, в транзакционном режиме [33] и постулировать неизменность плана (как

текущей ППППВ, так и следующей по цепи ППППВ) в рамках каждой такой транзакции.

Доказательство. Введем требуемую директиву **plan_group_atomize**, запускающую группу этапов плана в параллель, считая всю ППППВ транзакционным блоком. Если разрешены модификации плана (собственного или следующей по цепи ППППВ), то любая ППППВ, работающая более чем с одним этапом почти обречена на постоянный перезапуск транзакций для согласования изменений плана. Это приводит к фактически последовательному исполнению транзакций. Поэтому целесообразно запретить модификацию планов при запуске группы этапов плана в параллельно-транзакционном режиме, как это уже сделано нами, например, при запуске векторной обработки группы этапов плана на векторном OpenCL-расширителе. Доказано.

Классическим примером применения транзакционной памяти является построение распределения значений массива **FS** из **P** элементов ($\forall i : FS[i] \in [MIN; MAX]$) по **NDIV** равновеликим интервалам. Индексы значений, относящихся к k-му интервалу, накапливаются в массивах **idx**[k], а количество соответствующих значений накапливается в массиве **nn**. Таким образом, мы получаем в **nn** гистограмму с детализацией в **idx**.


```
reenterable void histo(bool init, float gap, float MIN, int
i, float * FS, int ** idx, int * nn) {
  if (init) {
     for (int j = 0; j < P; j++)
         plan_last(false, gap, MIN, j, FS+j, idx, nn);
     plan_group_atomize;
  } else {
     int k = (int)((*FS-MIN)/gap);
     if (k < 0) k = 0;
     if (k >= NDIV) k = NDIV-1;
     idx[k][nn[k]++] = i;
  }
}
/* Вызов:
float gap = (MAX-MIN)/NDIV;
histo(true, gap, MIN, 0, FS, idx, nn);
*/
```

2.5. Замеры эффективности распараллеливания

Основной интерес представляют наиболее часто встречающиеся случаи многоядерных систем с общей памятью, возможно при наличии одной-двух многоядерных видеокарт. По этой причине, а также в связи с отсутствием у автора возможности (на момент написания) замерить характеристики распараллеливания на полноценной кластерной системе, ограничимся именно вышеупомянутыми случаями.

2.5.1. Многоядерные системы с общей памятью

Замеры были проведены для реализаций вышеприведенных и некоторых иных алгоритмов с единственной модификацией: чтобы на практике проиллюстрировать эффект от распараллеливания, задачи были намеренно усложнены — вместо элементов данных типа «целое число» использовались элементы типа «вектор вещественных чисел» из 1000 и 2500 компонентов. Замеры проводились на двухпроцессорной 8-ядерной системе 2×Xeon 2,33 ГГц. По результатам вычислялись ускорение $S = S(N_П)$ и эффективность распараллеливания $Q = Q(N_П)$ в зависимости от количества ядер $N_П$:

$$S(N_П) = \frac{t_1}{t_{N_П}};$$

$$Q(N_П) = \frac{S(N_П)}{N_П},$$

где t_1 — время исполнения на одном ядре; $t_{Nп}$ — время исполнения на $N_П$ ядрах. Соответствующие графики показаны на рис. 2.1: сортировка Шелла — (а) и (б); суммирование элементов дерева — (в) и (г); нахождение минимального и максимального элементов дерева (конвейерный алгоритм) — (д) и (е); интегрирование одномерного уравнения теплопроводности (конвейерный алгоритм [32] с применением каналов для обменов данными в направлении, обратном направлению конвейера) — (ж) и (з). В сортировке Шелла использовался массив из 2^{13} элементов, столько же узлов содержали деревья.

Как и следовало ожидать, в большинстве случаев (рис. 2.1, а, б, г, д, е) характеристики оказались лучше при более высокой вычислительной нагрузке на один этап плана работ, поскольку затраты на распараллеливание не меняются, а на вычисления растут, соответственно растет доля распараллеливаемых вычислений в законе Амдала [5, 6]. Интересным исключением является случай суммирования элементов дерева, показанный на рис. 2.1, в, где пик ускорения выше для вектора из 1000 элементов в случае использования трех ядер. Возможно, именно в этом случае более эффективно был задействован кэш процессоров [12, 35], что и дало повышение производительности.

Заметим, что средняя эффективность распараллеливания для традиционных алгоритмов программирования не очень высока и плавно падает с ростом количества задействованных ядер. Пик чаще приходился на 5÷6 процессоров. Необходимо отметить, что данные результаты были получены на экспериментальной версии транслятора-препроцессора[1] для Planning C, которая фактически генерирует код, распараллеленный с применением OpenMP/OpenCL/MPI. Более эффективная реализация с применением низкоуровневого API операционной системы, вероятно, позволила бы получить более высокие результаты.

Интересны результаты, показанные при параллельном численном решении уравнения теплопроводности (см. рис. 2.1, ж, з). При использовании до 6 процессоров эффективность распараллеливания $Q(N_П)$ близка к 100-процентной с первым небольшим пиком суперлинейного ускорения, где $Q(N_П)$ превышает 100%, объясняющегося более производительной работой ядер за счет повышения эффективности использования кэша: общий объем данных, будучи поделен на количество задействованных ядер, полностью поместился в кэшах ядер, что повысило их производительность до пиковой.

Второй пик $Q(N_П)$, соответствующий на первый взгляд нереальным значениям 160÷170%, объясняется изменившимися свойствами решаемой задачи [17, 22] при увеличении количества ядер. В решаемой задаче используется распараллеливание по пространству (геометрический параллелизм [5]) с предварительным вычислением значений на одном из двух стыков расчетных блоков по схеме Головичева [37, с.47].

[1] Доступен по ссылке: http://www.pekunov.byethost31.com/Progs.htm#Reenterable

Увеличение числа ядер дает увеличение числа стыков и, соответственно, увеличение фактов расчета по схеме Головичева, дающей менее точные результаты [31] с фактическим «замедлением» по времени [26, с.99], значения которых, однако, обеспечили более эффективную работу алгоритма интегрирования, возможно даже на уровне машинной арифметики. Поэтому скорректированные значения эффективности распараллеливания (приведенные к средней эффективности работы алгоритма интегрирования) на втором пике, скорее всего соответствуют значениям около 100%.

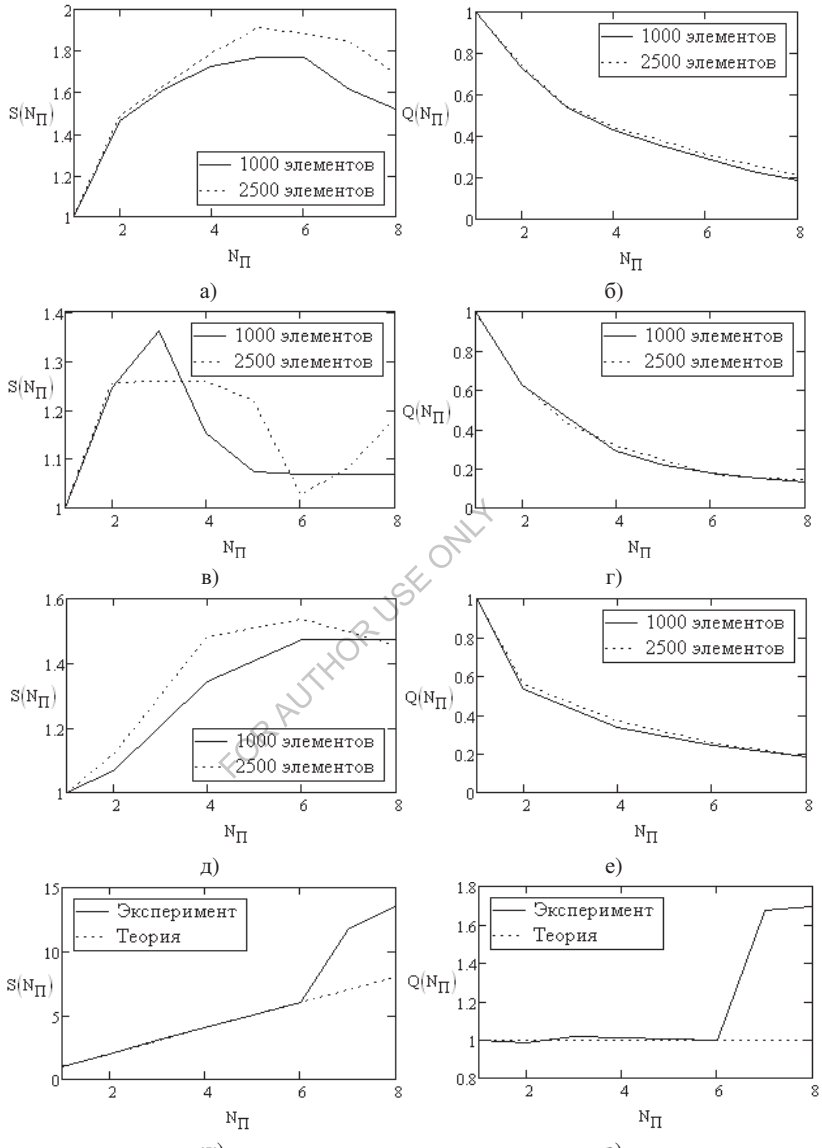


Рис. 2.1. Ускорение (а), (в), (д), (ж) и эффективность распараллеливания (б), (г), (е), (з) при различном количестве ядер (различные алгоритмы)

2.5.2. Процессоры с векторными инструкциями и видеокарты

Замеры проводились на программе обучения глубокой нейронной сети (структуры 6×5×1, с одним нейроном с повторным входом [23] в первом и во втором слое, остальные нейроны этих слоев имели активационную функцию «экспоненциальная сигмоида», третий слой — линейный) прямого распространения, аппроксимирующей зависимость коэффициента преломления воды от длины волны, с помощью предложенной в работе [23] модификации метода генетического случайного поиска (исходный вариант метода разработан С.Г.Сидоровым [28], предложившим ввести элементы генетического алгоритма для скрещивания результатов лучших из случайных проб) в рамках переборно-секционированного подхода, с дополнительными итерациями градиентного метода. Код (его фрагмент приведен в конце данного пункта), написанный первоначально в стиле параллелизма «портфель задач» на базе ПППВ, был незначительно модернизирован для векторного исполнения на многоядерных расширителях, в роли которых выступали две машины:

а) стандартная 16-ядерная (HyperThreading-ядра) система платформы Google's Compute Engine с многоядерными процессорами Xeon 2,6 ГГц с векторными SSE-инструкциями (выполнение четырех операций с данными одинарной точности за такт), что приблизительно эквивалентно наличию 64 потоковых процессоров (использовались драйверы Intel OpenCL);

б) система, на которой задействовались, преимущественно, одно ядро центрального процессора и две видеокарты NVidia Tesla K40 (0,88 ГГц, 2880 потоковых процессоров, драйверы nVidia).

На каждой итерации алгоритма обучения для скрещивания (с попыткой градиентного улучшения) отбиралось *не более 64 точек*, то есть пакет потоков, отправляемый на расширитель, содержал не более 4096 элементов.

Среднее время выполнения итерации на 16-ядерной системе составило 24,4 с, на системе с видеокартами — 4,79 с. По таким данным, *не имея длительности выполнения программы на одном потоке*, подсчитать ускорение и эффективность невозможно, но можно, по меньшей мере, определить, во сколько раз k эффективность распараллеливания

Q_{gpu} на системе с видеокартами меньше эффективности Q_{cpu} на 16-ядерной системе. Пусть количество операций для выполнения одной итерации не зависит от архитектуры расширителя и равняется $N_{оп}$. Тогда, если обозначить частоты процессоров первой и второй систем как f_{cpu} и f_{gpu}, а среднее время выполнения итерации как t_{64}^{cpu} и t_{5760}^{gpu}, соответственно, справедливы следующие выкладки:

$$t_1^{cpu} = \frac{N_{оп}}{f_{cpu}};$$

$$Q_{cpu} = \frac{t_1^{cpu}}{64 \cdot t_{64}^{cpu}};$$

$$t_1^{gpu} = \frac{N_{оп}}{f_{gpu}} = \frac{t_1^{cpu} \cdot f_{cpu}}{f_{gpu}};$$

$$Q_{gpu} = \frac{t_1^{gpu}}{5760 \cdot t_{5760}^{gpu}} = \frac{t_1^{cpu} \cdot f_{cpu}}{5760 \cdot t_{5760}^{gpu} \cdot f_{gpu}} = k \cdot Q_{cpu} = k \cdot \frac{t_1^{cpu}}{64 \cdot t_{64}^{cpu}};$$

$$k = \frac{64 \cdot t_{64}^{cpu} \cdot f_{cpu}}{5760 \cdot t_{5760}^{gpu} \cdot f_{gpu}} = \frac{64 \cdot 24{,}4 \cdot 2{,}6}{5760 \cdot 4{,}79 \cdot 0.88} \approx 0{,}17.$$

Итак, мы получили, что эффективность распараллеливания на системе с многоядерными видеокартами оказалась *приблизительно в шесть раз меньше* эффективности системы с 16 ядрами центрального процессора. При сравнительном ускорении в

$$\frac{24{,}4}{4{,}79} \approx 5{,}09$$

раза это достаточно *приемлемый результат*, учитывая неполную загрузку потоковых процессоров видеокарт (не более 4096 потоков из 5760 возможных) и существенную латентность их доступа к глобальной памяти.

Для справки приведем фрагмент использованного кода программы обучения глубокой нейронной сети в версии с применением векторных расширителей. Это ПППВ, выполняющая *третью стадию алгоритма* (скрещивание проб с дополнительным градиентным поиском). Данная ПППВ запускается как цепь-вектор, количество элементов которой со-

ответствует числу доступных векторных расширителей. Необходимые пояснения параметрам ПППВ даны в комментариях.

```
chain tune(
 bool init, /* Признак инициализационной стадии */
 int gn, /* Количество точек-родителей для скрещивания */
 int DIM, /* Количество координат для каждой точки */
 int _nmax, /* Максимальное число итераций градиентным ме-
тодом */
 int nPairs, /* Число обучающих пар */
 _global(nPairs) float * XP, /* Входные значения пар */
 _global(nPairs) float * YP, /* Выходные значения пар */
 _global(DIM) char * DimFlags, /* Флаги для каждой коорди-
наты: разрешена ли ее модификация при обучении */
 int base_offs, int base_size0, int base_size, /* Параметры
разделения пакета из gn*gn задач градиентного поиска по по-
токам векторного расширителя */
 _local(DIM*__planned__.base_size) float * OutX, /* Выход-
ные значения координат «улучшенных» поиском точек-потомков
*/
 _local(__planned__.base_size) float * OutF, /* Значения
целевой функции в «улучшенных» точках-потомках */
 _global(DIM*gn) float * XS /* Координаты точек-родителей
*/
) {
  int gn2 = gn*gn;
  if (init) {
    int stages = throw_num_stages();
    int stage = throw_stage();
    const char * device_id = get_device_id(stage);
    int max_work_size = vector_max_size(device_id);
    int divgn2 = gn2/stages;
    int modgn2 = gn2%stages;
    int _gn2 = divgn2 + (stage < modgn2 ? 1 : 0);
    int nn = min(_gn2, max_work_size);
    int nbase_size0 = _gn2/nn;
    int nbase_rest = _gn2%nn;
    int offs = stage*divgn2 + (stage < modgn2 ? stage :
modgn2);
    int i;
    for (i = 0; i < nn; i++) {
        int nbase_size = (i < nbase_rest ? nbase_size0+1 :
nbase_size0);
```

```
        plan_last(false, gn, DIM, _nmax, nPairs, XP, YP,
DimFlags, offs, nbase_size0, nbase_size, OutX+offs*DIM,
OutF+offs, XS);
        offs += nbase_size;
    }
    plan_group_vectorize(device_id);
  } else {
     int id = plan_vector_id();
     int i, j, k;
     barrier(0);
     for (j = 0; j < base_size; j++) {
         int _j = base_offs + j;
     int p1 = _j/gn;
     int p2 = _j%gn;
     int nmax = _nmax;
     float x[MAX_DIM], x1[MAX_DIM];
     float grad[MAX_DIM];
     float f, f1, fx;
     for (i = 0; i < DIM; i++) {
          x1[i] = 0.5f*(XS[i + p1*DIM]+XS[i + p2*DIM]);
          if (j < base_size0) barrier(0);
     }
     f1 = F(j < base_size0, 0, nPairs, XP, YP, x1);
     bool stopped = false;
     do {
          if (!stopped) {
             for (i = 0; i < DIM; i++)
                x[i] = x1[i];
             f = f1;
          }
          fx = F(j < base_size0, stopped ? &fx : 0, nPairs,
XP, YP, x);
          for (i = 0; i < DIM; i++) {
               float X1[MAX_DIM];
               if (!stopped) {
                  for (k = 0; k < DIM; k++)
                       X1[k] = x[k];
                  X1[i] += delta;
               }
               if (DimFlags[i])
                  grad[i] = (F(j < base_size0, stopped ?
&grad[i] : 0, nPairs, XP, YP, X1) - fx)/delta;
               else
                  grad[i] = 0.0f;
               if (j < base_size0)
```

```
                barrier(0);
            }
            if (!stopped)
                for (i = 0; i < DIM; i++)
                    x1[i] = x[i] - alpha*grad[i];
                f1 = F(j < base_size0, stopped ? &f1 : 0,
nPairs, XP, YP, x1);
                stopped = f1 >= f;
        } while (--nmax);
        for (i = 0; i < DIM; i++) {
            OutX[i+j*DIM] = x[i];
            if (j < base_size0) barrier(0);
        }
        OutF[j] = f;
        if (j < base_size0) barrier(0);
       }
       barrier(0);
    }
}
```


Выводы ко второй главе

Проведен обзор некоторых современных языков, расширений и интерфейсов параллельного программирования. Определены основные требования к средствам распараллеливания для разрабатываемого языка. Предложены простые средства распараллеливания процесса решения задач на кластерных системах, системах с общей памятью, векторных расширителях и гибридных системах с использованием функций и процедур с повторным входом.

В простейшем случае параллелизация решения сводится к адекватному внутреннему распараллеливанию исполнения плана работ («портфель задач») с балансировкой, которое реализуется системой программирования. Это позволяет «линеаризовать» задачи параллельной обработки нелинейных структур данных и выделить скрытый параллелизм некоторых рекурсивных алгоритмов, отчасти заменив рекурсию планированием. Данная схема распараллеливания легко векторизуется для решения на OpenCL-расширителях. Показано, что наиболее адекватным предельным параллельным вычислителем для ППП/ФППВ при

использовании «портфеля задач» является параллельная расширенная машина Тьюринга.

Предложен механизм цепей ППВ, работающих как на кластерных, так и на SMP-системах, позволяющий не только достаточно просто записывать некоторые алгоритмы, сводимые к решению серии подзадач с последовательной (стадийной) или параллельной (векторной) обработкой данных, но и естественно реализовать для них конвейерный и векторный виды параллелизма. Обмен данными может выполняться посредством прямого планирования, с применением физической или виртуальной общей памяти или с помощью каналов.

Введены виртуальные топологии, позволяющие описывать произвольные статические и динамические (масштабируемые) паттерны параллелизма. Масштабируемость обеспечивается применением дедуктивных макромодулей, использующих параметризованные логические правила, в ходе интерпретации которых порождается описание топологии[1]. Незначительная модификация таких макромодулей обеспечила полную поддержку порождающего программирования. Применение логических правил в таком контексте позволяет решать достаточно интеллектуальные задачи, например, генерировать описания топологий или фрагменты решающих задачу алгоритмов по высокоуровневой формальной постановке проблемы.

Введены некоторые дополнительные архитектурно-независимые (семафоры, барьеры, виртуальная общая память) и специализированные (блокировки, критические секции, транзакционная память) средства параллельного программирования. Приведены результаты экспериментальных замеров эффективности распараллеливания в системах с общей памятью и на векторных расширителях, демонстрирующие качество программ, написанных с применением разработанного языка Planning C.

[1] Применение четких формальных логических правил (по сравнению с классическим алгоритмическим или прямым перечислительным описанием связей топологии) потенциально способно снизить количество ошибок описания топологии

Глава 3. Новые программные конструкции в Planning C 2.0

Содержание предыдущих глав относилось, преимущественно, *к первой версии Planning C*. Версия 2.0 включает ряд важных дополнений, введенных для более глубокой реализации цели данной работы – повышения эффективности параллельного программирования вычислительно трудозатратных задач, которое вполне может быть осуществлено, помимо всего прочего, за счет привлечения подходов из области искусственного интеллекта, подразумевающих прогнозирование неизвестных данных с помощью интерполяции и экстраполяции. По мнению автора, решение таких задач потребует не только ввода интерполяционно-экстраполяционных функций/классов, которые обычно реализуются в составе дополнительных библиотек, но и, хотя бы частичный, перевод таких функций на уровень языковых конструкций. Такой перевод должен быть достаточно естественным, поэтому логичным представляется ввод конструкций такого рода для неких внутренних (скрытых) целей программы, например, при реализации табличной мемоизации процедур и функций, при которой можно пытаться предсказать отклик на комбинацию параметров, которой еще нет в таблице. При этом явное применение получаемых таким образом интерполяторов и экстраполяторов будет логически вторичным вариантом. Что же касается ввода классических/явных интерполяционно-экстраполяционных средств, то они также могут быть вполне уместны в программировании, например, для предсказания еще не полученных данных с целью повышения асинхронности и, соответственно, снижения общего времени выполнения.

Далее, заметим, что программирование достаточно сложных задач нередко требует взаимного согласования различных фрагментов программы (относящихся, например, к единому множеству концептов) с целью выбора наиболее эффективного варианта решения. Это приводит к идее применения элементов метапрограммирования в более явной форме, чем это реализуется в C++ в настоящее время. В предыдущих главах уже упоминались дедуктивные макромодули, в связи с вышеизложенным имеет смысл расширить их применение возможностью кол-

лективного многократного согласования, доведя до логического завершения идею интеллектуального метапрограммирования.

Нельзя также не отметить современную тенденцию к активному применению функционального программирования, в связи с чем представляется целесообразным введение в Planning C анонимных вариантов ППП/ФППВ и топологий.

Идея обеспечения возможности оперативного ввода в язык новых топологий, описываемых с применением новых удобочитаемых конструкций, была первоначально реализована автором с помощью специального механизма, который далее был обобщен на случай ввода в язык новых *произвольных* конструкций. Данный механизм может быть особенно ценным при реализации каких-либо специальных (например, предметно-ориентированных) расширений языка с целью повышения скорости и эффективности программирования. Его возможности также могут быть ценными и для оперативной разработки прототипов новых универсальных расширений языка. В частности, именно с помощью данного, первоначально предназначенного исключительно для расширения набора топологий, механизма в Planning C были введены средства, поддерживающие не только стандартные топологии и анонимные ППП/ФППВ/топологии, но и интеллектуальную мемоизацию, а также некоторые дополнительные возможности по автоматическому распараллеливанию (подробное рассмотрение которых выходит за рамки данной работы). Кроме того, разработанные средства вполне могут быть применены, например, для реализации аспектно-ориентированного программирования.

Итак, *целью данной главы* является дополнительное повышение скорости и эффективности разработки программного обеспечения, по возможности в соответствии с современными тенденциями. Для достижения этой цели поставим задачей разработку новых языковых средств Planning C, реализующих:

а) интеллектуальную мемоизацию с возможностью применения полученных языковых конструкций для решения задач интерполирования/прогнозирования;

б) средства обмена данными с явным применением прогнозирования еще не полученных значений;

в) средства компактного описания и применения стандартных вычислительных топологий;

г) анонимные ППП/ФППВ/топологии;

д) многократное согласование дедуктивных модулей;

е) средства оперативного расширения языка;

ж) различные средства, необходимые для поддержки некоторых новых приемов программирования, которые будут представлены в следующей главе.

3.1. Предикционно-решающие каналы (авторегрессионные модели и персептроны)

Как уже упоминалось выше, целесообразна разработка таких средств обмена, которые позволяли бы предсказывать еще не полученные данные с целью повышения асинхронности и, соответственно, снижения общего времени выполнения. Предикция должна быть достаточно надежной, при этом необходимо иметь достаточное качество прогнозов при небольшой вычислительной трудоемкости. Это приводит к идее применения простых и быстрых (пусть даже и менее совершенных) алгоритмов, которые были бы, по возможности, «прозрачны» для программиста (например, были бы совмещены со стандартными средствами обменов данными в многопроцессорной среде), привычны с точки зрения интерфейса и гибки в применении. В частности, достаточно естественной представляется инкапсуляция средств предикции в стандартные каналы передачи данных[1]. Схожий подход применен в работе [45], но лишь в отношении простейшей предикции некоторых служебных данных, поступающих в маршрутизатор из компьютерных сетей в ответ на запрос.

Особый интерес также представляет задача поиска таких программных формализмов и стратегий вычисления с применением предицирующих каналов, которые делали бы возможным или упрощали еще более глубокое распараллеливание некоторых алгоритмов, имеющее

[1] Такая же инкапсуляция может быть выполнена, например, для стандартных каналов языков Go [60] и OCaml [53].

сложности в реализации стандартными средствами. Такого рода стратегии будут рассмотрены в следующей главе.

3.1.1. Предикционно-решающие каналы

Введем понятие предикционно-решающего канала межпроцессной или внутрипроцессной передачи данных, который может работать в трех основных режимах, определяемых конкретной вызываемой функцией приема данных и ее параметрами:

а) *классический* режим асинхронной передачи данных, при котором принимающая сторона в обязательном порядке дожидается данных, переданных отправляющей стороной;

б) *предикционный* режим, при котором принимающая сторона пытается спрогнозировать переданные ей, но еще не полученные данные, основываясь на накопленной истории их изменения в ходе ранее осуществленных приемов-передач. Если накопленная история недостаточна, то канал осуществляет прием данных в классическом режиме;

в) *режим с таймаутом*, при котором в принимающую функцию передается значение допустимого времени ожидания. Если данные приходят за меньшее время, чем указано, то они принимаются в классическом режиме. Если же за указанное время данные не поступили, то принимающая функция переводится в предикционный режим.

Важно заметить, что вполне допустима работа канала в случае, если *передающая и принимающая стороны совпадают*. При этом процесс засылает серию данных в канал и получает из него результат прогнозирования следующих значений для этой серии данных, то есть, в данном случае предикционно-решающий канал является *локальным прогностером*.

Предложим *две основные разновидности* предикционно-решающих каналов:

1. *Авторегрессионные точечные каналы*, выполняющие экстраполяцию значений конкретной переменной в некоторой точке, основываясь на истории значений, поступивших ранее в канал в данной точке. В параллельном программировании ряда задач моделирования процессов в сплошных средах такие каналы могут использоваться для устранения

ожиданий при обменах данными на стыках блоков расчетной области (при использовании геометрического параллелизма), путем предсказания принимаемых значений переменных. Возможно даже частичное периодическое исключение обменов вообще (с последующим приемом актуальных данных и коррекцией экстраполяторов канала), в таких случаях работа идет с предсказанными данными, а устранение обменов значительно уменьшает время работы программы. Обычно такая задача решается *явным* применением экстраполяторов или дополнительных разностных схем, поэтому применение авторегрессионных каналов существенно упрощает программу, соответственно, снижается вероятность внесения в нее ошибок.

Также возможно, например, точечное предсказание различных характеристик вычислительного процесса, например, предполагаемого времени работы численного алгоритма в конкретных узлах, для выбора варианта алгоритма или для эффективного локально статического распределения нагрузки (узлов) по процессорам, что особенно актуально для задач моделирования процессов аэро- или гидродинамики, осложненных химической кинетикой (см. [17]).

2. *Линейные каналы коллективного решения (явные и неявные)*, выполняющие экстраполяцию массива значений некоторой переменной, основываясь на предыдущих значениях этой же переменной, а также, возможно, актуальных значений переменных из иных каналов (зависимость канала от иных каналов определяется при его создании). Если новые (прогнозируемые) значения в канале зависят только от предыдущих значений в некотором подмножестве элементов массива, то используются простые линейные соотношения и канал является *явным*. Если же новые значения в произвольном элементе массива в канале зависят также от новых значений в иных элементах этого же канала, то для расчета новых значений решается система линейных уравнений и канал является *неявным*.

Очевидно, что, при адекватном применении, каналы коллективного решения, фактически, могут выполнять аппроксимацию явных и/или неявных разностных схем, что определяет два основных варианта их использования:

а) для решения (более корректного с точки зрения численных методов по сравнению с точечными каналами) тех же задач предварительно

ного расчета значений в узлах расчетной сетки на стыках блоков расчетной области, которые в общем случае, несомненно, зависят и от «соседних» узлов и от значений других переменных в текущем и «соседних» узлах;

б) для решения достаточно неожиданной задачи *простого и эффективного перехода (в простых случаях) от явных разностных схем к неявным*, что обычно требует применения специальных математических подходов (например, схем расщепления по физическим процессам или по пространственным переменным) и достаточно громоздкого программирования. В самом деле, достаточно организовать некоторое количество итераций вычислительного процесса по явной схеме (при соблюдении условий вычислительной устойчивости) с передачей получаемых данных в неявный канал коллективного решения, который, фактически, аппроксимирует по этим данным неявную схему. Далее организуется цикл отправки «пустых» данных в этот канал с приемом из него данных в строгом *предикционном режиме* – это, фактически, будут результаты дальнейшего эквивалентного счета *по неявной разностной схеме*, отличающейся заметно большей вычислительной устойчивостью. Заметим, что такой же подход может использоваться и для обратного перехода – от неявной разностной схемы к явной. Также отметим, что подобного рода переход от расчета по исходной разностной схеме, реализованной в программе, к расчету по разностной схеме, аппроксимированной каналом, может, фактически, быть *одним из вариантов автоматического распараллеливания расчета*, если исходная программа не является параллельной, а канал реализует счет в параллельном режиме. Необходимо лишь предусмотреть *возможность масштабирования построенного каналом предиктора*, путем его быстрого одношагового пересчета при изменении каких-либо параметров схемы, например, величины шага по времени. Это можно осуществить путем обучения нескольких вспомогательных каналов (при разном значении параметра), на основании данных которых основной канал подбирает, например, простые линейные зависимости коэффициентов предиктора от параметра. Так можно не только переходить от явных схем к распараллеленным неявным, но и увеличивать при этом шаг интегрирования по времени, что дает дополнительное ускорение счета.

3.1.2. Авторегрессионные точечные каналы

Авторегрессионный точечный канал выполняет или обычный блокирующий прием или экстраполяцию очередных значений V_i некоторой переменной в точках i = 1…N, где N – число точек. Для экстраполяции используется история значений (величины $H_{i,j}$, поступившие ранее в канал в данной точке, j = 1…P, где P – глубина истории). Экстраполяция реализуется линейным соотношением:

$$V_i = K_{i,0} + \sum_{k=1}^{S_i} K_{i,k} H_{i,P-k+1} ; S_i \le P,$$

где $K_{i,k}$ – коэффициенты i-го экстраполятора, S_i – порядок i-го экстраполятора. Очевидно, что такой экстраполятор эквивалентен линейному персептрону из одного нейрона со смещением $K_{i,0}$.

Обучение i-го экстраполятора (персептрона) выполняется с помощью метода наименьших квадратов, путем минимизации функционала

$$F(K_i) = \sum_{j=S_i}^{P-1} \left(K_{i,0} + \sum_{k=1}^{S_i} K_{i,k} H_{i,j-k+1} - H_{i,j+1} \right)^2 \to \min ,$$

которая, в данном линейном случае, сводится к решению системы линейных алгебраических уравнений, возникающих из условий равенства первых частных производных нулю:

$$\frac{\partial F(K_i)}{\partial K_{i,r}} = 0 ; r = 0 \ldots S_i.$$

Решение системы может осуществляться, например, методом Гаусса-Зейделя, или, если он не сходится – методом LU-разложения.

Порядок экстраполятора S_i обычно выбирается из условия минимума $F(K_i)$, однако если для всех S_i не выполняется условие $F(K_i) \le \varepsilon$, где ε — допустимая погрешность экстраполяции, то выбирается $S_i = 0$. Величины N, P, ε задаются при инициализации канала.

3.1.2.1. Языковые средства поддержки

Авторегрессионные точечные каналы представлены шаблонными классами **funneled_predictor_in**<тип_элемента> и **fun-**

neled_predictor_out<тип_элемента>, представляющими, соответственно, входной и выходной концы канала. Для этих каналов, как и для обычных, определены *три конструктора*:

а) для создания первого конца нового канала:

funneled_predictor_in(int N, int P = 5, double rel_eps = 0.001),

funneled_predictor_out(int N, int P = 5, double rel_eps = 0.001),

где **N** – число элементов данных (для каждого из которых обучается свой предиктор), **P** – глубина истории для прогноза, **rel_eps** – относительная точность, которую должен обеспечивать режим предикции;

б) для создания ответного конца нового канала:

funneled_predictor_in(int P, void * _Ref),

funneled_predictor_out(int P, void * _Ref),

где **N** – число элементов данных (для каждого из которых обучается свой предиктор), **_Ref** – ссылка на уже созданный противоположный конец канала;

в) для создания концов канала, идентифицируемого по уникальному имени **Name**:

funneled_predictor_in(const char * Name, int N = 1, int P = 5, double rel_eps = 0.001),

funneled_predictor_out(const char * Name, int N = 1, int P = 5, double rel_eps = 0.001),

где **N** – число элементов данных (для каждого из которых обучается свой предиктор), **P** – глубина истории для прогноза, **rel_eps** – относительная точность, которую должен обеспечивать режим предикции.

Например, если создается локальный (соединяющий процесс с самим собой) одноточечный канал (P = 4, ε = 0,01) для приема-передачи вещественного числа, то его полная инициализация, включающая создание приемного и передающего коннекторов выглядит так:

```
funneled_predictor_out<double> * out = new
     funneled_predictor_out<double>("FUNNEL", 1, 4, 0.01);
funneled_predictor_in<double> * in = new
     funneled_predictor_in<double>("FUNNEL", 1, 4, 0.01);
```

Класс **funneled_predictor_out**, с точки зрения программного интерфейса, полностью эквивалентен выходному концу обычного канала, он также имеет метод **put**, с теми же параметрами, выполняющий от-

правку данных в канал (с ожиданием его освобождения, если в канале были непринятые данные).

Класс **funneled_predictor_in**, как и в случае обычного канала, имеет эквивалентный метод **get** с теми же параметрами, выполняющий безусловный прием данных. Однако при этом он также содержит метод

bool get_timed(ссылка_на_буфер, таймаут_в_миллисекундах),

который ожидает прием данных в течение времени, не превышающего указанного таймаута. Если данные успели прийти, то они копируются в указанный буфер, а метод возвращает ложное значение. Если данные прийти не успели и уже был обучен предиктор, то выполняется прогнозирование ожидаемых данных (на базе собранной истории), в результате чего прогнозные значения копируются в буфер, а метод возвращает истинное значение (после такого прогнозирования далее будет необходимо все же либо принять поступающие данные методом **get** или **get_and_correct**, либо сбросить их методом **cancel_and_push**). Если же предиктор обучить не удалось, то метод ожидает приема данных, не обращая внимания на указанную величину таймаута, и возвращает ложное значение. Заметим, что если для такого канала требуется получить данные *строго с привлечением предиктора*, то достаточно просто указать отрицательную величину таймаута.

Поскольку, хотя бы иногда необходимо *проверять валидность предиктора*, класс **funneled_predictor_in** дополнительно содержит метод

get_and_correct(ссылка_на_буфер),

который (в обязательном порядке) ожидает приема данных в указанный буфер, затем сравнивает их с прогнозными (чтобы они присутствовали, необходимо вызывать данный метод после метода **get_timed**, причем только в том случае, если он вернул истинное значение) и, если расхождение превышает требуемую точность, переобучает предиктор.

В случае, если нет оснований сомневаться в предикторе и прием с помощью **get_timed** успешно завершен (с предикцией), может потребоваться элементарный сброс поступающего в канал внешнего значения с сопутствующим помещением в историю спрогнозированных значений. Это возможно с помощью метода

cancel_and_push(ссылка_на_буфер_со_спрогнозированными_значениями).

Соответственно, прием в предикционном режиме (с последующей коррекцией предиктора) может выполняться в таком случае кодом, подобным следующему:

```
if (in->get_timed((void *) &val, -1.0)) { // Предикционный
режим. Если заменить параметр «-1.0» положительным значением
таймаута, то реализуется режим приема с таймаутом.
      double buf = 0.0;
      in->get_and_correct((void *) &buf); // Классический режим
приема с коррекцией предиктора
      cout << buf << "[Предсказано = " << val << "] ";
} else
      cout << val << " "; // Если предиктор не был вычислен (это
возможно в режиме приема с таймаутом)
```

Дополнительно отметим, что существуют кластерные версии вышеупомянутых классов-концов авторегрессионного точечного канала. Соответственно, они именуются как

cfunneled_predictor_in<тип_элемента>,

cfunneled_predictor_out<тип_элемента>.

В целом, они имеют такую же логику работу и отличаются, по очевидным причинам, только реализацией и наличием *лишь одного конструктора, принимающего уникальное имя создаваемого канала*.

Далее приведем небольшой демонстрационный пример применения таких классов в программе, одна часть которой передает данные второй части, которая пытается принять их с таймаутом и предикцией.

```
#pragma plan clustered

#include <stdlib.h>
#include <omp.h>
#include <time.h>

const int N = 26;

chain sender() {
      double seq[N] = { 1.0, 2.0, 3.0, 4.0, 3.0, 2.0, 1.0, 2.0,
3.0, 4.0, 3.0, 1.0, 2.0, 3.0, 5.0, 1.0, 2.0, 1.0, 2.0, 1.0, 2.0,
1.0, 2.0, 1.0, 2.0, 1.0 };
      cfunneled_predictor_out<double> * out = new
cfunneled_predictor_out<double>("FUNNEL", 1, 4, 0.01);
      while (!out->ready());
      for (int i = 0; i < N; i++) {
```

```
          out->put((void *) &seq[i], sizeof(double));
          Sleep(20);
      }
      delete out;
}

chain receiver() {
      double seq1[N] = { 0 };
           cfunneled_predictor_in<double> * in = new
cfunneled_predictor_in<double>("FUNNEL", 1, 4, 0.01);
           while (!in->ready());
           for (int i = 0; i < N; i++) {
               if (in->get_timed((void *) &seq1[i], rand() %
30)) {
                  double buf = 0.0;
                  in->get_and_correct((void *) &buf);
                  cout << buf << "[predict=" << seq1[i] << "] ";
             } else
                cout << seq1[i] << " ";
        }
          delete in;
      plan_topology_quit();
}

int main() {
      srand((unsigned int) time(NULL));

      int IDS[2] = {0,1};

      clustered(IDS)
      plan_topology {
           plan_parallel_chain(sender(),receiver());
      };

      return 0;
}
```


3.1.3. Линейные каналы коллективного решения

Линейный канал коллективного решения выполняет или обычный блокирующий прием или экстраполяцию очередных значений V_i некоторой переменной в точках i = 1…N, где N – число точек. Для экстраполяции используются:

а) история значений искомой величины (значения $H_{i,j}$, поступившие ранее в канал, j = 1…P, где P – глубина истории);

б) история значений величин, поступивших в каналы, связанные с текущим (значения $R_{i,j,m}$, m = 1…M, где M – число каналов, связанных с текущим).

Как уже упоминалось выше, канал коллективного решения может быть явным или неявным. Кроме того, он может (по желанию исследователя-программиста) иметь или не иметь свободный член B_i.

Экстраполяция *в явном канале со свободным членом* B_i реализуется линейным соотношением:

$$V_i = B_i + \sum_{k=1}^{N} A_{i,k} H_{k,P} + \sum_{m=1}^{M} C_{i,m} R_{i,P,m} ;$$

где $A_{i,k}$ и $C_{i,m}$ – коэффициенты i-го экстраполятора. Очевидно, что расчет такого экстраполятора эквивалентен расчету однослойного линейного персептрона с N нейронами, имеющими смещение B_i.

Необходимо сразу отметить, что в матрице C ненулевыми являются только элементы, соответствующие значимым (для расчета в каждой i-й точке) связанным каналам. Значимость выявляется обычным корреляционным анализом (связанный канал k значим для расчета в i-й точке, если модуль коэффициента корреляции между $R_{i,j,k}$ и $H_{i,j}$ превышает некоторую наперед заданную величину Δ).

Обучение i-го явного экстраполятора (персептрона) выполняется с помощью метода наименьших квадратов, путем минимизации функционала

$$F(A_i, B_i, C_i) = \sum_{j=1}^{P-1} \left(B_i + \sum_{k=1}^{N} A_{i,k} H_{k,j} + \sum_{m=1}^{M} C_{i,m} R_{i,j+1,m} - H_{i,j+1} \right)^2 \rightarrow \min ,$$

которая сводится к решению системы линейных алгебраических уравнений, возникающих из условий равенства первых частных производных по неизвестным коэффициентам нулю.

Экстраполяция *в неявном канале со свободным членом* B_i реализуется решением системы из N линейных уравнений относительно V:

$$H_{i,P} = B_i + \sum_{k=1}^{N} A_{i,k} V_k + \sum_{m=1}^{M} C_{i,m} R_{i,P,m} ; \; i = 1…N,$$

где A и C – матрицы коэффициентов коллективного экстраполятора. Очевидно, что расчет такого экстраполятора эквивалентен решению *за-*

дачи, обратной расчету однослойного линейного персептрона с N нейронами, имеющими смещение B_i (здесь требуется определить такие значения входов персептрона, при которых наблюдаются указанные значения выходов).

Обучение неявного экстраполятора (персептрона) выполняется последовательно для каждого из значений i, с помощью метода наименьших квадратов, путем минимизации функционала

$$F(A_i, B_i, C_i) = \sum_{j=1}^{P-1} \left(B_i + \sum_{k=1}^{N} A_{i,k} H_{k,j+1} + \sum_{m=1}^{M} C_{i,m} R_{i,j+1,m} - H_{i,j} \right)^2 \to \min ,$$

которая сводится к решению системы линейных алгебраических уравнений, возникающих из условий равенства первых частных производных по неизвестным коэффициентам нулю.

Экстраполяция в каналах без свободного члена также реализуется линейными соотношениями при $B_i = 0$. Поиск соответствующих экстраполяторов производится по тому же принципу, что и для каналов со свободным членом.

Далее заметим, что независимо от явности/неявности канала, количество ненулевых членов p в A_i обычно выбирается из условия минимума $F(A_i,B_i,C_i)$, однако если для всех p не выполняется условие $F(A_i,B_i,C_i) \le \varepsilon$, где ε – допустимая погрешность экстраполяции, то выбирается p = 0, в этом случае в расчете участвуют только B_i и C_i.

Величины N, ε, признаки «явный/неявный» и «со свободным членом/без него» задаются при инициализации канала.

3.1.3.1. Оптимизация поиска предиктора в каналах коллективного решения

В настоящее время в каналах коллективного решения для упрощения процесса расчета предиктора предполагается, что матрицы A являются диагональными с числом диагоналей W[1]. *Для сокращения времени поиска предиктора* вполне допускается явное указание максимального числа (должно быть нечетным) диагоналей W при создании концевых

[1] В более общем случае, для определения ненулевых элементов A_i, несомненно, необходим анализ корреляции между значением величины в i-й точке и значениями в соседних точках.

коннекторов канала (в конструкторах). Это позволяет существенно упростить определение ненулевых элементов A путем последовательных попыток определения предиктора от максимально возможного W до его нулевого значения. Среди найденных вариантов предиктора выбирается тот, который дает минимальную (в смысле метода наименьших квадратов) погрешность.

3.1.3.2. Масштабирование предикторов линейных каналов

Масштабирование построенного каналом предиктора осуществляется путем его быстрого одношагового пересчета в зависимости от некоторого внешнего параметра k (например, величины шага по времени). Для подготовки масштабирования (в некотором канале X) необходимы следующие шаги:

а) произвести серию передач данных (полученных при некотором фиксированном значении $k = k_1$) через канал X;

б) ввести вспомогательный канал Z и произвести через него серию передач данных (полученных при некотором фиксированном значении $k = k_2$);

в) произвести вызов метода **parametrize** для «среды» канала X, передав в него k_1, k_2 и «среду» канала Z.

После этого канал X становится параметризованным, скорректировать предполагаемое значение предиктора (для значения параметра k_3) можно, вызвав метод **use_parameter**(k_3) для среды канала «X».

Параметризация канала X по значениям из предикторов каналов X и Z осуществляется путем поиска индивидуальных линейных зависимостей коэффициентов предиктора от параметра k. Например, пусть каналы X и Z имеют свободные члены. Тогда имеет место соотношение:

$$B_i^{X,масшт}(k) = \alpha_{B_i}^X + \beta_{B_i}^X k,$$

коэффициенты которого определяются путем решения системы:

$$\begin{cases} B_i^{X,исх}(k_1) = \alpha_{B_i}^X + \beta_{B_i}^X k_1 \\ B_i^{Z,исх}(k_2) = \alpha_{B_i}^X + \beta_{B_i}^X k_2 \end{cases}$$

где индексы «масшт» и «исх» относятся, соответственно, к масштабированному каналу X и к немасштабированным (исходным) данным кана-

лов X и Z. Если система не имеет единственного решения,то можно принять

$$\alpha_{B_i}^{X} = B_i^{X,\text{исх}}(k_1);\ \beta_{B_i}^{X} = 0.$$

Аналогичным образом параметризуются не только свободные члены, но и все остальные коэффициенты предикторов.

Как уже упоминалось выше, параметризация позволяет, например, увеличивать шаг интегрирования по времени для аппроксимированных неявных разностных схем, что дает дополнительное ускорение счета. В приложении П2 приведен пример программы на Planning C на линейных каналах коллективного решения, осуществляющей переход от явной схемы (для численного интегрирования нестационарного уравнения теплопроводности) к аппроксимированной неявной с параметризованным шагом по времени.

3.1.3.3. Языковые средства поддержки

Линейные каналы коллективного решения представлены шаблонными классами

funneled_perceptron_in<тип_элемента>,
funneled_perceptron_out<тип_элемента>,

представляющими, соответственно, входной и выходной концы канала[1]. Для этих каналов определены *два конструктора*:

а) для создания концов канала, идентифицируемого по уникальному имени **Name**:

funneled_perceptron_in(bool _explicit, bool hasB, const vector<_funnel<тип_элемента> *> &refs, const char * Name, int N = 1, double rel_eps = 0.001, int PATTERN = -1),
funneled_perceptron_out(bool _explicit, bool hasB, const vector<_funnel<тип_элемента> *> &refs, const char * Name, int N = 1, double rel_eps = 0.001, int PATTERN = -1),

где **_explicit** определяет, является ли канал явным, **hasB** – имеет ли канал свободный член, **refs** – вектор ссылок на «среды» связанных каналов (их можно получить методом **getRef()** из любых концов связанных каналов),

[1] В настоящее время существуют только в версии с общей памятью, кластерная версия пока не реализована.

N – число элементов передаваемых данных, **rel_eps** – относительная точность, которую должен обеспечивать режим предикции, **PATTERN** – максимальное число (должно быть нечетным; игнорируется, если отрицательно) диагоналей W при создании концевых коннекторов канала (см. п.3.1.3.1);

б) для создания ответного конца нового канала:

funneled_perceptron_in(bool _explicit, bool hasB, const vector<_funnel<Type> *> &refs, int N, void * _Ref, int PATTERN = -1),
funneled_perceptron_out(bool _explicit, bool hasB, const vector<_funnel<Type> *> &refs, int N, void * _Ref, int PATTERN = -1),

где параметры имеют то же назначение, что и в конструкторах из пункта (а). Единственный новый параметр – **_Ref**, являющийся ссылкой на уже созданный конец канала.

Дополнительно следует подчеркнуть, что, при создании канала, *вектор ссылок **refs** заполняется лишь на одном из концов канала*, в противном случае возникнет дублирование ссылок.

Например, если создается явный канал (зависимый от канала inb) без свободного члена, $\varepsilon = 10^{-5}$, для приема-передачи N вещественных чисел, то его полная инициализация, включающая создание приемного и передающего коннекторов на передающей стороне выглядит так:


```
vector<_funnel<double> *> refs;
refs.push_back(inb->getRef());
funneled_perceptron_out<double> * outpi = new
     funneled_perceptron_out<double>(true, false, refs,
"iPFUNNEL", N, 1E-5);
```

На принимающей стороне:

```
vector<_funnel<double> *> refs;
funneled_perceptron_in<double> * inpi = new
     funneled_perceptron_in<double>(true, false, refs,
"iPFUNNEL", N, 1E-5);
```

Класс **funneled_perceptron_out**, с точки зрения программного интерфейса, полностью эквивалентен выходному концу **funneled_predictor_out**, он также имеет метод **put**, с теми же параметрами, выполняющий отправку данных в канал (с ожиданием его освобождения, если в канале были непринятые данные).

Класс **funneled_perceptron_in** интерфейсно эквивалентен (см. п.3.1.2.1) классу **funneled_predictor_in**, он также имеет методы **get**, **get_timed**, **get_and_correct**, **cancel_and_push**, имеющими то же назначение.

Прием в предикционном режиме с последующей коррекцией предиктора выполняется по той же схеме, что и для точечного канала (см. выше).

Продолжим приведенный выше пример для принимающего конца канала inpi, при этом предполагаем, что передача-прием по связанному каналу inb уже произведены. Прием в предикционном режиме (в массив seqpp) без коррекции предиктора выполняется по схеме:

```
if (inpi->get_timed((void *) seqpp, -1.0)) { // Предикционный
режим. Если заменить параметр «-1.0» положительным значением
таймаута, то реализуется режим приема с таймаутом.
     inpi->cancel_and_push((void *) seqpp); // Классический
режим приема – принятое значение сбрасывается, вместо него в
историю помещаются значения, только что вычисленные предиктором
 } else
     cout << "NO PREDICTION!" << endl;
```


3.1.4. Некоторые сведения об апробации

В работе [16] автором приведены достаточно детальные сведения о решении задачи численного интегрирования одномерного нестационарного уравнения теплопроводности с применением каналов вышеописанных видов. Было показано существенное повышение ускорения (в 1,4÷6,5 раза на восьмиядерной 16-потоковой вычислительной системе проекта Google’s Compute Engine) для параллельной моделирующей программы (неявный метод Эйлера) при переходе от классической схемы с обычными каналами и предвычислением с помощью дополнительной разностной схемы Головичева, к схеме с применением авторегрессионных точечных каналов, позволившей частично избавить программу от излишних простоев при обменах данными.

Показано, что применение неявных каналов коллективного решения позволило (при использовании минимальных средств программирования) перейти от последовательной явной схемы счета уравнения теп-

лопроводности к распараллеленной неявной схеме, отличающейся более высокой вычислительной устойчивостью (что позволяет значительно увеличить шаг интегрирования по времени, уменьшив время на моделирование), и получить заметное ускорение счета (в 1,1÷1,69 раза на восьмиядерной 16-потоковой вычислительной системе проекта Google's Compute Engine). Такой переход может иметь определенную ценность при разработке программ с нестандартными разностными схемами, для которых не существует стандартных, хорошо распараллеленных математических библиотек.

3.2. Частично транзакционная память

Еще одним новшеством второй версии Planning C является поддержка частично транзакционной памяти, которая сочетает нетранзакционные и транзакционные элементы. Как будет показано в следующей главе (там же будут приведены и некоторые результаты по эффективности применения такой памяти), данный формализм, в сочетании с авторегрессионными точечными каналами, способен дать новую стратегию параллельного программирования – сверхоптимистичные вычисления.

Итак, идея частично транзакционной памяти очень проста: за основу берется обычная нетранзакционная (классическая) память, для которой пишутся специальные классы, соответствующие основным видам транзакционных переменных (скаляр, массив, входной и выходной концы каналов), поддерживающие параллельный режим работы с фиксацией момента модификации и возможностью отмены такой модификации. В дополнение к классам вводится некая программная среда (набор данных и алгоритмов), ведущая учет объектов таких классов, запускающая требуемое количество транзакций и контролирующая непротиворечивость их совместной работы с применением отката транзакций, требующих пересогласования. При этом проблема частичной транзакционности решается автоматически: переменные по умолчанию нетранзакционны, транзакционными являются лишь объекты специальных классов.

В-целом, алгоритмы программной реализации таких элементов транзакционной памяти, как скаляры и массивы, достаточно хорошо известны и их реализация в Planning C не требует какого-либо детального

пояснения. Не вполне очевидными являются только механизмы согласования входных и выходных концов каналов, которые и будут рассмотрены далее.

3.2.1. Транзакционное согласование каналов

Проблема состоит в согласовании концов каналов, находящихся в разных транзакциях, если требуется откат для одной или обеих этих транзакций (сразу же отметим, что и *сам канал может являться источником решения об откате принимающей транзакции*, если средой исполнения было установлено существенное расхождение между спрогнозированным и реально принятым значениями). Задача осложняется тем, что если существует сеть каналов между транзакциями, то необходимо предусмотреть каскадный пересчет транзакций, следуя направлениям передач. Сформулируем следующие *правила согласований и откатов*:

1. Введем понятие графа транзакций (V, E), где V – множество транзакций, а E – множество дуг, указывающих направление передач по каналам.

2. Определим, с помощью обычных алгоритмов транзакционной памяти те транзакции, которые подлежат пересчету (при этом каналы в расчет не принимаются). Далее, в каждой принимающей транзакции сверяем значения, полученные из передающей транзакции, со значениями, предсказанными в текущей принимающей транзакции. Если расхождение превышает допустимое, то принимающая транзакция подлежит пересчету. Окрасим вершины подлежащих пересчету транзакций.

3. Если транзакция $v \in V$ окрашена (подлежит пересчету) и существует множество исходящих из нее дуг (v, x_k), то окрашиваем все вершины x_k – соответствующие транзакции также подлежат пересчету. Повторяем этот пункт, пока существуют дуги (m, n) такие, что m окрашена, а n – не окрашена.

4. Если транзакция $v \in V$ окрашена (подлежит пересчету) и существует множество входящих в нее дуг (y_k, v), таких, что y_k не окрашена, то восстанавливаем в среде канала $y_k \Rightarrow v$ значение, которое было передано в v на текущем этапе (это значение будет считаться еще не принятым).

Повторяем этот пункт, пока существуют дуги, к которым данное правило еще не было применено.

5. Делаем откат для всех транзакций, которым соответствуют окрашенные вершины, и запускаем их на пересчет.

3.2.2. Новые классы транзакционных переменных

Как уже было сказано выше, были разработаны четыре новых основных шаблонных класса:

1. Транзакционный скаляр **TScalar<тип>**. Имеет конструктор с одним параметром – уникальным строковым идентификатором транзакционного блока (пустая строка по умолчанию).

Объекты данного класса доступны на чтение и на запись (в операторе присваивания **=**), также возможно применение операторов модификации: инкремента (**++**), сложения (**+=**) и вычитания (**-=**).

2. Транзакционный массив **TArray<тип_элемента>**. Имеет конструктор с двумя параметрами:

TArray(число_элементов, идентификатор_транзакционного_блока = "").

Каждый элемент массива, фактически, является самостоятельной переменной типа **TScalar<тип_элемента>**, к которой могут быть применены все операции, указанные в предыдущем пункте. Доступ к элементу можно получить с помощью обычного оператора **[индекс]**. Также возможно присваивание массиву скалярного значения с помощью обычного оператора присваивания (**=**) – в таком случае указанное значение получают все элементы массива.

3. Транзакционный вход авторегрессионного точечного канала **TIn<тип_элемента>**. Имеет конструктор вида

TIn(const char * Name, int N = 1, int P = 2, double rel_eps = 0.001, int _owner = -1, const char * __id__ = ""),

где параметры **Name**, **N**, **P**, **rel_eps** имеют тот же смысл, что и для обычных авторегрессионных точечных каналов (см. выше), **_owner** – номер OpenMP-потока, которому принадлежит вход канала (если имеет отрицательное значение, то система определяет его самостоятельно, однако для этого конструктор необходимо вызывать уже после включения ре-

жима параллельной работы в частично транзакционной памяти), **__id__** – строковой идентификатор транзакционного блока.

Данный класс имеет лишь метод упрощенного приема с таймаутом, принимаются данные всех **N** элементов:

get(ссылка_на_буфер, таймаут_в_миллисекундах).

При наличии данных в канале, метод немедленно их принимает. Если же данных в настоящий момент нет, то метод работает аналогично **get_timed** с указанным таймаутом.

4. Транзакционный выход авторегрессионного точечного канала **TOut<тип_элемента>**. Имеет конструктор вида

TOut(const char * Name, int N = 1, int P = 2, double rel_eps = 0.001, int _owner = -1, const char * __id__ = ""),

где параметры **Name**, **N**, **P**, **rel_eps** имеют тот же смысл, что и для обычных авторегрессионных точечных каналов (см. выше), **_owner** – номер OpenMP-потока, которому принадлежит выход канала (если имеет отрицательное значение, то система определяет его самостоятельно, однако для этого конструктор необходимо вызывать уже после включения режима параллельной работы в частично транзакционной памяти), **__id__** – строковой идентификатор транзакционного блока.

Данный класс имеет лишь метод упрощенной передачи, отправляются данные всех **N** элементов:

put(ссылка_на_буфер).

3.2.3. Способы запуска параллельного режима с частично транзакционной памятью

Можно выделить два основных способа такого запуска, первый из которых является более универсальным (может быть применен не только в Planning C, но и в иных языках программирования), а второй – лучше вписывается в концепцию Planning C.

3.2.3.1. Отдельный программный блок в сочетании с директивами OpenMP

В данном способе запускается параллельный блок из K потоков (средствами OpenMP), в котором вызывается специальная конструкция **transaction_atomic(строковой_идентификатор_блока) { тело_блока },** генерирующая K согласуемых транзакций, каждая из которых может быть идентифицирована с помощью стандартной функции **omp_get_thread_num()**. Возможность указать для каждого блока свой идентификатор, помимо всего прочего, допускает *возможность вложения частично транзакционных блоков.*

Приведем пример программы, которая рассчитывает гистограмму F массива целых чисел A, суммирует все частоты и выводит сумму на экран.


```
#include <stdlib.h>
#include <iostream>

using namespace std;

const int N = 10000;
const int M = 600;
const int NF = 20;

int main() {
      int * A = new int[N];
      TArray<int> F(NF);

      srand(184415);

      for (int i = 0; i < N; i++)
          A[i] = rand() % M;

      double grain = 1.0 * M / NF;

      F = 0;
      #pragma omp parallel shared(A,F,grain)
         {
            int np = omp_get_num_threads();
            int id = omp_get_thread_num();

            for (int i = 0; i < N; i += np)
                  transaction_atomic("") {
```

```
                        if (i + id < N) {
                              int k = A[i + id] / grain;
                              if (k >= NF) k = NF - 1;
                              ++F[k];
                        }
                  }
      }

      int SF = 0;
      for (int i = 0; i < NF; i++)
            SF += F[i];
      cout << SF;

      delete[] A;

      return 0;
}
```

3.2.3.2. Новый режим исполнения группы этапов плана ППП/ФППВ

Как уже упоминалось ранее (п.2.4.2.2), ППП/ФППВ может обеспечить запуск исполнения группы этапов своего плана в параллель с применением классической транзакционной памяти. Согласно теореме ТА2, для этого достаточно ввести новую директиву запуска параллельного обсчета группы этапов плана в транзакционном режиме и постулировать неизменность плана в рамках каждой такой транзакции[1].

Следствие из ТА2. Теорема ТА2 будет справедлива и для случая частично транзакционной памяти, достаточно лишь ввести специальную новую директиву параллельного запуска в соответствующем режиме.

Пусть такой директивой будет **plan_group_soft_atomize**.

Общая схема ППП/ФППВ для параллельного запуска группы этапов в частично транзакционной памяти будет достаточно классической для Planning C: необходимо сначала построить группу этапов плана, а затем включить вышеуказанную директиву. Обычно для этого вводится специальный логический параметр ППП/ФППВ, который специ-

[1] Требование неизменности плана не является критическим в случае частично транзакционной памяти, поскольку план всегда может трактоваться как нетранзакционная перемення. Однако для унификации подходов к работе с обычной и частично транзакционной памятью, а также для упрощения построения транслятора, указанное требование было сохранено.

фицирует режим построения группы (true) или режим исполнения группы (false). При начальном запуске ПППВ/ФППВ данный параметр равен true. Тело ПППФ/ФППВ имеет общий вид:

```
if (параметр) {
     Планирование группы этапов;
     plan_group_soft_atomize;
} else {
     Исполнение этапа-транзакции в режиме частично
транзакционной памяти;
}
```

Следует заметить, что для удобства программирования такого режима, в Planning C введены несколько макросов-типов:

- **soft_transact_array(тип_элем)**, эквивалентный **TArray <тип_элем>**;
- **soft_transact_var(тип)**, эквивалентный **TScalar<тип>**;
- **soft_transact_in(тип_элем)**, эквивалентный **TIn<тип_элем>**;
- **soft_transact_out(тип_элем)**, эквивалентный **TOut<тип_элем>**.

Далее будет приведен пример программы, выполняющей те же функции, что и в предыдущем пункте, но с применением ПППВ/ФППВ. Ее важным достоинством является использование исключительно средств Planning C.


```
#include <stdlib.h>
#include <iostream>

using namespace std;

const int N = 10000;
const int M = 600;
const int NF = 20;

reenterable calc_histo(bool init, int np, int * A,
soft_transact_array(int) * F, double grain, int k) {
     if (init) {
          for (int i = 0; i < N; i += np) {
               for (int j = 0; j < np; j++)
                    plan_last(false, np, A, F, grain, i + j);
               plan_group_soft_atomize;
          }
     } else {
          if (k < N) {
```

```
                int _k = A[k] / grain;
                if (_k >= NF) _k = NF - 1;
                ++(*F)[_k];
            }
        }
}

int main() {
      int * A = new int[N];
      soft_transact_array(int) F(NF);

      srand(184415);

      for (int i = 0; i < N; i++)
          A[i] = rand() % M;

      double grain = 1.0 * M / NF;

      F = 0;
      calc_histo(true, 4, A, &F, grain, 0);

      int SF = 0;
      for (int i = 0; i < NF; i++)
           SF += F[i];
      cout << SF;

      delete[] A;

      return 0;
}
```


3.3. Обычная и интерполирующая мемоизация. Применение в программировании и вычислительной математике

Мемоизация – достаточно хорошо известный прием (см., например, [57]), позволяющий, при неоднократном обращении к одной и той же процедуре/функции с одним и тем же набором параметров, кэшировать результат, полученный при первом обращении, и возвращать его при последующих обращениях без выполнения данной процедуры/функции. Насколько известно автору, текущие реализации C++ не содержат стандартных конструкций, поддерживающих мемоизацию, соответственно, представляется логичным такие конструкции ввести, хотя бы в рамках Planning C.

Необходимо заметить, что если речь идет о процедуре/функции, выполняющей некий алгоритм, результаты которого описываются достаточно гладкой, но неизвестной заранее математической функцией, то такие результаты, теоретически, могут быть вычислимы (хотя бы в некоторых точках) с помощью интерполяции или экстраполяции, если известен набор ключевых точек. Отсюда можно заключить, что в таких случаях

а) набор ключевых точек (параметры, результат) может быть получен как побочный эффект мемоизации;

б) при определенных условиях вполне возможна *интерполирующая мемоизация*, когда для текущих параметров запомненного результата нет, но он может быть достаточно легко вычислен.

Следует подчеркнуть, что интерполирующая мемоизация такого рода будет давать *положительный эффект только при соблюдении следующих условий*:

а) время вычисления интерполированного результата C строго меньше среднего времени выполнения процедуры/функции T;

б) имеет место неравенство $P + K \cdot C < K \cdot T$, где P – время обучения предиктора, K – некоторое количество итераций, на которых обученный предиктор дает приемлемые по точности результаты;

в) выполняется периодический контроль качества предиктора, когда для заданного набора входных параметров вычисляется прогноз V, затем запускается мемоизированная процедура/функция, вычисляющая верный результат B, и, наконец, сравниваются V и B. Если погрешность прогноза не превышает некоторой заданной точности, то предиктор валиден и будет использоваться еще некоторую серию вызовов. В противном случае предиктор подлежит переобучению по имеющемуся набору мемоизированных точек, возможно, в сочетании с дополнительным набором новых точек, получаемых в результате прямого вызова процедуры/функции.

Интерполирующая мемоизация может быть применена, например, если процедура/функция решает прямую задачу химической кинетики, или, например, интегрирует ОДУ, результат расчета которого входит в какой-либо еще циклический математический алгоритм. Следует заметить, что, если рабочим элементом является, например, нейронная сеть, линейный экстраполятор или полином, построенный методом группово-

го учета аргументов (МГУА [8]), то возможен любопытный побочный эффект: мемоизированная процедура/функция может, фактически, стать интерфейсом обучения и вычисления отклика такого элемента – тем самым *нейронные сети, линейные экстраполяторы и МГУА-полиномы неявно вводятся в язык программирования* и могут быть применены в нем для самых различных программных целей.

Как показал анализ литературных источников, данный прием является новым, по крайней мере на уровне интерфейса прикладного программиста (схожие решения существуют для аппаратного уровня, например, для реализации кэшей микропроцессоров, см., например, [62]).

3.3.1. Синтаксис. Режимы мемоизации

Включение режима мемоизации для процедуры/функции производится с помощью специальной директивы, вставляемой в отдельной строке непосредственно перед заголовком:

директива_мемоизации = «**#pragma**» пробелы «**memoization**» [пробелы] «(» [пробелы] карта_параметров [пробелы] «)» [[пробелы] (МГУА | нейросеть|экстраполятор) [вид_контроля]]

карта_параметров = параметр {[пробелы] «**,**» [пробелы] параметр}

параметр = {«*»} вид_параметра

вид_параметра = входной | выходной | группирующий | порядковый

входной = «**i**»

выходной = «**o**»

группирующий = «**g**»

порядковый = «**t**»

МГУА = «**mgua**» [пробелы] «(» точность «)»

точность = вещественное_число_от_0_до_1

нейросеть = «**feed_forward**» пробелы «(» точность «**,**» коэффициент_обучения «**,**» макс_число_итераций «**,**» [описатель_нелинейных_слоев] «)»

экстраполятор = «**lin_extrapolator**» [пробелы] «(» точность «**,**» порядок_экстраполятора «)»

коэффициент обучения = вещественное_число

порядок_экстраполятора = целое_число_большее_нуля

описатель_нелинейных_слоев = число_нейронов_в_слое «**,**» актив_функция [«**,**» число_нейронов_в_слое «**,**» актив_функция]
актив_функция = экспон_сигмоида | линейная | гиперб_тангенс | ReLU
экспон_сигмоида = «**e**»
линейная = «**l**»
гиперб_тангенс = «**h**»
ReLU = «**r**»
вид_контроля = автоматический | по_условию
автоматический = «**controlled**» «(» число_разгонных_точек «**,**» число_предицируемых_точек «)»
по_условию = «**conditions**» «(» условие_отбора_точки_в_историю «,» условие_предикции «)»
условие_отбора_точки_в_историю = логическое_выражение
условие_предикции = логическое_выражение

Дадим некоторые пояснения. *Карта_параметров* указывается всегда, ее элементы специфицируют вид параметра – входной, выходной, порядковый или группирующий. Если параметр представляет собой указатель на входное/выходное/группирующее значение, то перед символом вида указывается оператор разыменования «*****». Если мемоизация производится для функции, то ее *результат всегда является одним из выходных параметров, заданным неявно*.

ВАЖНОЕ ЗАМЕЧАНИЕ. Для правильной работы мемоизации необходимо, чтобы параметры имели исчерпывающие спецификаторы типов, без использования **typedef**-определений для массивов. Иными словами, если параметр C является **float**-массивом 3×2, то он должен быть указан в формате: **float C[3][2]**.

Группирующий параметр введен для упрощения вычисления предикторов и повышения их качества. Если он определен (в таком случае он должен быть единственным!), то для каждого из обнаруженных в ходе работы значений этого параметра генерируется отдельный предиктор, обучаемый только по точкам, полученным при передаче в процедуру/функцию данного значения группового параметра. В частности, такой режим очень удобен при мемоизации процедур/функций, вычисляющих неким математическим алгоритмом значения некоторого поля. В этом случае попытки обучения единого предиктора для всех узловых значений поля могут быть малоуспешными или занимать чрезмерно большое

время на обучение, в то время как обучение отдельного предиктора для каждого узла способно дать заметно лучшие результаты. В этом случае следует ввести мемоизируемую процедуру, обрабатывающую за один раз один узел, принимающую ряд входных и один выходной (узловое значение поля) параметры, а также группирующий параметр – номер узла, все остальное среда исполнения проделает автоматически.

Порядковый параметр имеет значение только для экстраполирующей мемоизации. В этом случае его значение используется для установления порядка, в котором будут рассматриваться значения выходного параметра при построении линейного экстраполятора. Порядковый параметр может быть только один, кроме того, он должен быть первым параметром мемоизируемой процедуры/функции. При этом наличие прочих входных параметров возможно, но они не будут приниматься во внимание. Выходной параметр должен быть только одним (явно указывается в карте параметров или под ним подразумевается результат функции, если мемоизируется функция).

Режим обычной мемоизации включается, если в вышеуказанной директиве не указано ни утверждения «**mgua**», ни утверждения «**feed_forward**», ни утверждения «**lin_extrapolator**».

Режим интерполирующей/экстраполирующей мемоизации без контроля включается, если в директиве указано утверждение «**mgua**» или утверждение «**feed_forward**» или утверждение «**lin_extrapolator**», но не содержится указание «**controlled**» или «**conditions**». В данном режиме по умолчаниию выполняется обычная мемоизация, а для вызова предиктора используется специальный синтаксис с префиксом

«**predict_**» идентификатор_процедуры «(» параметры_вызова «)»

Для режима МГУА в директиве указывается только требуемая точность, для линейной экстраполяции – требуемая точность и число коэффициентов экстраполятора, для нейросети – требуемая точность, параметр для метода обратного распространения ошибки и максимальное количество итераций данного метода. Если *описатель нелинейных слоев отсутствует*, то нейронная сеть представляет собой однослойный линейный персептрон (число нейронов равняется количеству вычисляемых предиктором элементов). Если же *описатель присутствует*, то нейронная сеть имеет указанные нелинейные слои с указанными активационными функциями, а также линейный выходной слой (число нейро-

нов в котором равняется количеству вычисляемых предиктором элементов).

Режим интерполирующей мемоизации с автоматическим контролем включается при наличии указания «**controlled**». При этом процедура/функция начинает работу в режиме обычной мемоизации, собирая историю ключевых точек на протяжении величины **число_разгонных_точек**. Вводится внутренний параметр – коэффициент доверия результатам **w**, изначально равный единице. Далее включается режим *контролируемой предикции*, в котором: а) выполняются вызовы предиктора с возвратом спрогнозированных результатов (на протяжении числа вызовов, равного величине **w*число_предицируемых_точек**), б) осуществляется контроль (вычисляется предиктор, затем для тех же параметров вызывается исходная процедура и результаты сравниваются) – если контроль по точности проходит, то **w** увеличивается и переходим к (а), и так далее. Если же контроль по точности не проходит, то **w** уменьшается, предиктор пересчитывается и переходим к (б).

Режим интерполирующей мемоизации с контролем по условию включается при наличии указания «**conditions**». Данный режим является экспериментальным и, возможно, будет изменен в следующих версиях. Если при очередном обращении к процедуре/функции выполняется **условие_предикции**, то выполняется предикция с возвратом соответствующих результатов. Если же это условие не выполняется, то для получения результатов вызывается исходная процедура/функция. Если при этом выполняется **условие_отбора_точки_в_историю**, то текущая пара (параметры, результат) включается в историю. Как только размер собранной истории превысит максимальный (который определяется при вызове функции **set_memoization_max_history**), предиктор автоматически переобучается.

3.3.2. Библиотека memoization.h

Вышеперечисленные конструкции мемоизации были оформлены как расширения языка, подключаемые через стандартную библиотеку **memorization.h**. Данная библиотека содержит сканер, обнаруживающий прагму мемоизации и вызывающий дедуктивный макромодуль, генери-

рующий необходимые структуры данных для хранения мемоизированных пар (параметры, результат) и формирующий несколько вспомогательных процедур, поддерживающих как обычную, так и интерполирующую мемоизацию. Кроме того, библиотека содержит реализации основных классов интерполяторов и экстраполяторов (многослойной нейронной сети прямого распространения, линейных предикторов и полиномов МГУА).

С точки зрения программирования, данная библиотека содержит лишь одну явным образом вызываемую функцию

set_memoization_max_history(максимальное_число_точек),

которая устанавливает максимальное количество хранимых мемоизированных точек *для каждого предиктора*. Данная функция была введена с целью разумного ограничения времени на расчет предикторов, которое существенно зависит от размера такой истории. Следует пояснить, что при превышении данной величины на 20%, среда исполнения «прореживает» историю, доведя ее до указанной максимально возможной величины. При этом «прореживание» происходит вполне равномерно на протяжении всей имеющейся истории.

3.3.3. Пример классической мемоизации

Приведем пример программы с мемоизацией при поиске a-го числа Фибоначчи, которую оформим тремя различными способами:

```
#include <iostream>
#include <iomanip>

using namespace std;

#include "memoization.h"

#pragma memoization(i)
double fibb(int a) {
  if (a < 3) return 1;
  else return fibb(a-1)+fibb(a-2);
}

#pragma memoization(i,o)
void fibb(int a, double &result) {
```

```
  if (a < 3) result = 1;
  else {
    double b, c;
    fibb(a-1, b);
    fibb(a-2, c);
    result = b+c;
  }
}

#pragma memoization(i,*o)
void fibb(int a, double * result) {
  if (a < 3) *result = 1;
  else {
    double b, c;
    fibb(a-1, &b);
    fibb(a-2, &c);
    *result = b+c;
  }
}

int main() {
     cout << setprecision(16) << fibb(77) << endl;

     double r;
     fibb(77, r);
     cout << setprecision(16) << r << endl;

     fibb(77, &r);
     cout << setprecision(16) << r << endl;

     return 0;
}
```


3.3.4. Пример интерполирующей мемоизации без группирующего параметра

Пусть решается следующая математическая задача: дана труба с неизвестным коэффициентом теплоотдачи стенок. По трубе движется жидкость (вода), ее температура на входе и на выходе известна, также известна температура окружающей среды. Необходимо определить коэффициент теплоотдачи стенок.

Данная задача может решаться как проблема нелинейной оптимизации. Минимизируется функция ошибки расчета $F(\alpha)$, зависящая от ко-

эффициента теплоотдачи α. Для определения ошибки расчета задается известная температура на входе трубы и текущее значение коэффициента теплоотдачи α, затем, в численном эксперименте определяется температура на выходе $T_{эксп}$, которая сравнивается с известной температурой $T_{вых}$. То есть, проблема имеет вид:

$$\alpha_{рез} = \arg\min(F(\alpha));$$

$$F(\alpha) = (T_{эксп}(\alpha) - T_{вых})^2.$$

Итак, пусть $T_{эксп}$ находится путем численного интегрирования уравнения теплопроводности до установления при заданном α. Пусть расчетна сетка достаточно велика (имеет несколько тысяч узлов). Поместим этот расчет в процедуру и применим интерполирующую мемоизацию. Необходимые условия применимости интерполирующей мемоизации будут выполнены, если использовать нейросетевой интерполятор небольшого размера – тогда время его обучения и вычисления будет меньше, чем время численного решения уравнения теплопроводности, причем погрешность может быть невелика, поскольку $T_{эксп}(\alpha)$, скорее всего, является достаточно гладкой функцией. Воспользуемся трехслойной нейронной сетью с пятью нейронами в первом слое (гиперболический тангенс), тремя – во втором (гиперболический тангенс) и одним линейным нейроном в третьем слое. Применим автоматический контроль погрешности.

Приведем программу:

```
#include "memoization.h"
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <omp.h>

using namespace std;

const double tol = 1E-2; // Допустимая погрешность расчета
температуры

const double a2 = 0.143E-6; // Коэффициент
температуропроводности [м2/с]
const double U = 3.0; // Скорость движения воды по трубе [м/с]
const double c = 4183; // Теплоемкость воды [Дж/кг/К]
const double ro = 1000; // Плотность воды [кг/м3]
```

```
const double L = 1000; // Длина трубы [м]

const double Tenv = 273.15 + 5.0; // Температура среды [K]
const double Tin = 273.15 + 70.0; // Температура на входе [K]
const double Taim = 273.15 + 60.0; // Температура на выходе [K]

#pragma memoization(i,o) feed_forward(0.001, 0.05, 2000, 5, h,
3, h) controlled(7,2)
void get_t(double alpha, double & t1) {
      const int N = 1000; // Число узлов
      const double h = L/(N-1); // Шаг сетки
      const double tau = 0.01; // Шаг по времени

      double T[N] = { Tin }; // Граничное условие первого рода
слева
      double T1[N] = { Tin };

      // Интегрирование до установления -- решаем уравнение
переноса тепла явной схемой Эйлера
      bool stable = false;
      while (!stable) {
            stable = true;
            for (int i = 1; i < N-1; i++) {
                  double delta = tau*(-U*(T[i]-T[i-1])/h +
a2*(T[i-1]-2.0*T[i]+T[i+1])/h/h + alpha*(Tenv-T[i])/c/ro/h);
                  T1[i] = T[i] + delta;
                  if (fabs(delta) > tol) stable = false;
            }
            for (int i = 1; i < N-1; i++)
                  T[i] = T1[i];
            T[N-1] = T[N-2]; // Граничное условие второго рода
справа
      }
      t1 = T[N-1];
}

// Целевая минимизируемая функция (квадрат ошибки значения
температуры на выходе)
double F(double alpha) {
      double d, t1;
      get_t(alpha, t1);
      d = Taim - t1;
      return d*d;
}

double g(double alpha) {
      const double hg = 0.005; // Шаг для вычисления градиента
      return (F(alpha+hg)-F(alpha))/hg;
}
```

```
int main() {
      srand((unsigned int) time(NULL));

      double alpha;

      // Засекаем время
      double time0 = omp_get_wtime();

      // Поиск входной температуры градиентным методом
      double err = 1E100; // Квадрат ошибки
      alpha = 600; // Начальное приближение
      while (err > tol) {
            double grad = g(alpha); // Градиент

            double F0 = F(alpha);
            cout << "Error^2(" << alpha << ") = " << F0 << endl;

            double h = 10;
            do {
                  double aa = alpha - h*grad;
                  double Ft = F(aa);
                  if (Ft < F0) {
                        F0 = Ft;
                        alpha = aa;
                        cout << "Error^2(" << aa << ") = " << F0
<< endl;
                  } else
                        h /= 2;
            } while (h > 0.001);
            err = F0;
      }

      cout << "a2 = " << alpha << " " << "F = " << F(alpha) <<
endl;

      double time1 = omp_get_wtime(); // Вторая засечка времени
      cout << "Elapsed time (predictor) = " << (time1-time0) << "
sec." << endl;

      return 0;
}
```


Как показали результаты вычислений и замеров, данная программа нашла значение $\alpha = 1168{,}88$ за 1144,57 с. При этом аналогичная программа без мемоизации (ее можно получить из приведенной, просто исключив директиву мемоизации) нашла значение $\alpha = 1168{,}89$ за 10163 с.

Таким образом, при очень малой погрешности, для данной задачи интерполирующая мемоизация позволила получить результат в 8,88 раза быстрее. Обычная мемоизация не смогла в данном случае дать существенного выигрыша (что также было подтверждено экспериментально – время решения существенно превысило результат интерполирующей мемоизации [1144,57 с]), поскольку входные значения alpha являются вещественными числами и практически не повторяются в процессе решения.

3.3.5. Пример интерполирующей мемоизации с группирующим параметром

Пусть решается двумерная аэродинамическая задача, осложненная химической кинетикой [17]: моделируется обдувание здания, на крыше которого происходит активное выделение и горение метана, образуются продукты горения. Наиболее вычислительно затратной процедурой здесь является интегрирование подсистемы уравнений химической кинетики (15 уравнений) в каждом узле расчетной области. Выделим данную процедуру в отдельный блок и попробуем применить интерполирующую мемоизацию.

Прежде всего заметим, что вряд ли возможно применение единого интерполятора для всех узлов расчетной области, поскольку подсистема уравнений химической кинетики достаточно непроста и вряд ли имеет тривиальное аналитическое решение, которое может быть эффективно интерполировано нейронной сетью небольшой сложности. Поэтому, именно в данном случае представляется оправданным применение группирующего параметра – номера узла, который позволил бы обучить для каждого узла свой локальный интерполятор. Забегая вперед, скажем, что попытка применить в данном случае нейронную сеть даже в таком качестве завершилось очевидным неуспехом – мемоизация не только не дала выигрыша по времени, но даже замедлила решение. Поэтому было принято решение попытаться применить МГУА с автоматическим контролем погрешности мемоизации.

Полный текст программы здесь не приводится, ввиду его существенного объема, приведем только фрагмент с мемоизированной процедурой:

```
#pragma memoization(g,i,i,i,o) mgua(0.0000001)
controlled(800,10)
void get_kinetics(int group, double _time, double Tk, float
INC[15], float OUTC[15]) {
      for (int k = 0; k < KIN->NSubst; k++)
            KIN->Conc0[k] = INC[k];
      KIN->Tk = Tk;
      KIN->OneTaktKinetic(1, 0);
      for (int k = 0; !KIN->LossPrecision && k < KIN->NSubst;
k++)
            if (_isnan(KIN->Conc1[k]))
                  KIN->LossPrecision = 1;
      if (KIN->LossPrecision) {
            printf(" Oshibka vichislenii : %s.",
                  (KIN->KinErrorInfo.LossH == 0 ? " slishkom mnogo
iterasii" :
                  " poterya tochnosti - shag po vremeni stremitcya
k nulu "));
            printf(" Vichisleniya prervani v moment t=%lf.
Vipolneno %i iterasii\n",
                  KIN->KinErrorInfo.ReachTau, KIN->Iters);
            for (int k = 0; k < KIN->NSubst; k++)
                  KIN->Conc1[k] = KIN->Conc0[k];
      }
      for (int k = 0; k < KIN->NSubst; k++)
            OUTC[k] = max(0.0, KIN->Conc1[k]);
}
```


Здесь **group** – группирующий параметр (номер узла), **_time** – текущее время моделирования, **Tk** – температура в текущем узле, **INC** – массив входных концентраций веществ, **OUTC** – массив выходных концентраций веществ.

Как показали результаты вычислений и замеров, моделирование на интервале времени в 0,485 секунды при использовании мемоизации заняло 205,6 секунд, а при ее отсутствии – 271,91 секунды. При этом относительная погрешность не превысила 15÷16%. С учетом достаточно высокой сложности задачи такой результат может быть признан вполне удовлетворительным.

3.3.6. Пример экстраполирующей мемоизации

Пусть некоторая функция возвращает значения ряда чисел, в котором встречаются закономерности. Используя мемоизацию с линейным экстраполятором, построим предикцию таких значений.

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include "memoization.h"

using namespace std;

const int N = 14;
double seq[N] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 5.0, 4.0,
3.0, 2.0, 1.0, 0.0, -1.0 };

#pragma memoization(t) lin_extrapolator(0.25, 2) controlled(4,
0)
double series(int t) {
      return seq[t];
}

int main() {
      srand((unsigned int) time(NULL));

      for (int i = 0; i < N; i++)
            cout << seq[i] << " ";
      cout << endl;
      for (int i = 0; i < N; i++)
            cout << (i >= 4 ? predict_series(i) : 0.0) << "->"
<< series(i) << " ";
      cout << endl;

      return 0;
}
```


Данная программа выводит на экран следующие строки:

```
1 2 3 4 5 6 7 5 4 3 2 1 0 -1
0->1 0->2 0->3 0->4 5->5 6->6 7->7 8->5 6->4 11->3 3->2 1->1 0-
>0 -1->-1
```

Заметим, что возможны и более практически полезные применения мемоизации с линейным экстраполятором, например, для предсказания времени интегрирования уравнений химической кинетики в узлах расчетной сетки при численном моделировании сложных процессов механики сплошных сред. Для каждого узла сетки такое время является

рядом чисел и часто вполне предицируемо. Такая априорная информация может быть использована для тщательной балансировки загрузки процессоров при распараллеливании по пространству (геометрический параллелизм) [17].

3.4. Расширенная схема препроцессинга

Выше уже упоминались такие новые возможности Planning C как коллективное согласование дедуктивных макромодулей, оперативный ввод новых вычислительных топологий и, в более общем смысле, оперативное построение языковых расширений. Имеет смысл реализовать коллективное согласование макромодулей через единую базу фактов GNU Prolog, при этом, учитывая линейный характер анализа программы, необходимо обеспечить возможность модулям, идущим выше по тексту, согласовывать свое поведение с модулями, идущими ниже по тексту. Наиболее простой реализацией данного процесса может быть многократное выполнение препроцессинга с сохранением единой базы фактов между стадиями.

Что же касается поддержки возможностей оперативного ввода в язык новых стандартных вычислительных топологий и построения произвольным образом оформленных языковых расширений, то представляется оправданным введение в язык понятия сканирующего модуля или *сканера* (таких модулей может быть множество).

Сканер имеет в своем составе три регулярно-логических выражения [19]: левый контекст, основное и правый контекст. Текст программы должен просматриваться сканером, который находит фрагменты, подпадающие под конкатенацию вышеупомянутых трех выражений, и, либо заменяет фрагмент, соответствующий основному выражению, вызовом дедуктивного макромодуля (который позднее будет развернут в некий программный код), характерного для языка Planning C, либо, не производя замены, лишь помещает некий дополнительный факт (который прочие макромодули будут учитывать в своей работе) в общую базу фактов. Очевидно, что таким путем можно вводить в язык новые синтаксические конструкции (в том числе описывающие обычные и/или анонимные стандартные вычислительные топологии), в которых сканер бу-

дет обнаруживать и сворачивать фрагменты программы в параметризованные вызовы макромодулей, трансформирующих данные фрагменты и разворачивающих их в выходной код.

Исходя из вышеизложенного, необходимо реализовать в Planning C новую *расширенную схему препроцессинга*:

1. Пусть задано требуемое максимальное количество циклов препроцессинга K. Также задан исходный текст программы.

2. *Первая фаза*. Анализируется весь текст программы с исполнением директив препроцессора и выполнением макроподстановок.

3. *Вторая фаза*. Просматривается весь полученный к данному моменту текст программы, обнаруживаются все сканирующие макросы и директивы, устанавливающие используемые сканеры, режим и порядок их применения. Далее сканирующие макросы применяются к полученному на данный момент тексту программы в режиме парсинга или сканирования (будут рассмотрены далее). Найденные выражения заменяются на обращения к макромодулям или на пустые строки (с генерацией GNU Prolog-факта).

4. *Третья фаза*. Полученный текст программы просматривается еще раз — собирается таблица именованных констант и осуществляется поиск обращений к макромодулям, для которых выполняются подстановки значений констант в параметры (если параметром макромодуля является выражение, то оно вычисляется). Далее в точках обращений к макромодулям генерируется программный код. Для этого, если тело макромодуля содержит цели, формируется GNU Prolog-программа, в которую включаются библиотечные предикаты (характерные для Planning C), предикаты и цели (поочередно) макромодуля. Сгенерированный код логической программы обрабатывается для каждой цели путем вызова внешнего стандартного интерпретатора GNU Prolog (при этом логическая программа учитывает в своей работе имеющиеся в базе факты и, возможно, пополняет эту базу). Полученный при этом консольный вывод вставляется в тело модуля вместо соответствующей цели. После исключения из тела модуля предикатов получаем готовый к вставке в Planning C-программу код.

5. Если при выполнении какого-либо из дедуктивных макромодулей был вызван предикат **prune** или, если достигнуто максимальное число циклов препроцессинга, то препроцессинг заканчивается. В против-

ном случае сбрасываются все результаты препроцессинга (кроме единой базы фактов), загружается исходный текст программы и выполняется переход к пункту 2.

3.4.1. Управление препроцессингом

Введем директиву, определяющую максимальное количество стадий препроцессинга:

«**#**» «**preproc_passes**» «(» макс_число_стадий «)»

Дополнительно вводятся четыре новые директивы, управляющие способом применения сканирующих макросов на второй фазе препроцессинга:

а) *применение в режиме сканирования*. Добавление сканеров в список:

«**#**» «**add_scan**» «(» идентификатор_сканера {«**,**» идентификатор_сканера} «)»

Указание исчерпывающего набора сканеров:

«**#**» «**scan**» «(» идентификатор_сканера {«**,**» идентификатор_сканера} «)»

В данном режиме последовательно однократно перебираются все сканеры (в порядке, указанном в директивах), каждый из которых просматривает *всю программу* и формирует список всех необходимых подстановок. При этом все сканеры работают с одним и тем же текстом программы, полученным до второй фазы препроцессинга. После завершения перебора сканеров, сформированный список подстановок применяется к текущему тексту программы. Такой режим достаточно эффективен при небольшом количестве сканеров (применимых лишь к отдельным фрагментам программы), поскольку просмотр всей программы сканером может быть достаточно трудоемким процессом;

б) *применение в режиме парсинга*. Добавление сканеров в список:

«**#**» «**add_parse**» «(» идентификатор_сканера {«**,**» идентификатор_сканера} «)»

Указание исчерпывающего набора сканеров:

«**#**» «**parse**» «(» идентификатор_сканера {«**,**» идентификатор_сканера} «)»

Здесь программа просматривается последовательно и однократно. Для каждого очередного фрагмента программы делается попытка последовательно применить все сканеры (в порядке, указанном в директивах), причем, как только некий сканер идентифицирует фрагмент, в список подстановок добавляется требуемый элемент замены и на этом анализ фрагмента заканчивается. В отличие от первого режима, если встречается фрагмент, который не идентифицируется ни одним из сканеров, генерируется ошибка. Все сканеры работают с одним и тем же текстом программы, полученным до второй фазы препроцессинга. После завершения просмотра текста программы, сформированный список подстановок применяется к этому тексту. Данный режим обычно применяется, если необходим полный разбор программы, например, для реализации ее перевода на другой язык или ее автоматического распараллеливания. В этом случае парсинг, как правило, эффективнее сканирования.

3.4.2. Сканирующие макросы

Определим сканер, опустив при этом подробности синтаксиса и семантики регулярно-логических выражений (описаны, например, в [19]).

3.4.2.1. Синтаксис

сканер = «**#**» «**def_pattern**» пробелы идентификатор_сканера [пробелы] «**=>**»
[пробелы] ((«**[**» идентификатор_факта «**]**») | идентификатор_макромодуля) [пробелы] «(» [параметры_макромодуля_или_факта] «)» [пробелы] «**{**» тело_сканера «**}**» «**;**»
параметры_макромодуля_или_факта = параметр {[пробелы] «**,**» [пробелы] параметр}
параметр = строковая_константа_в_апострофах | XPath_выражение | список
список = «**[**» [параметр {[пробелы] «**,**» [пробелы] параметр}] «**]**»

тело_сканера = [левый_контекст] основное_выражение [правый контекст]

левый контекст = регулярно_логич_выражение ПС

ПС = перевод_строки

правый_контекст = регулярно_логич_выражение ПС

основное выражение = «@» «**begin**» ПС регулярно_логич_выражение ПС «@» «**end**» ПС

Необходимо дать некоторые дополнительные пояснения. Как только сканер находит фрагмент, соответствующий конкатенации левого контекста (если он определен), основного выражения и правого контекста (если он определен), фрагмент, соответствующий основному выражению, вырезается из программы и заменяется либо вызовом макромодуля, либо пустой строкой с генерацией факта (который помещается в единую базу фактов). При этом параметры факта или макромодуля определяются в соответствии со схемой, указанной в определении сканера.

XPath_выражение в параметрах указывается, обычно, либо для вызова каких-либо стандартных функций (например, **gid()**, выдающую уникальный идентификатор найденного фрагмента, соответствующего основному выражению; **random()** или **randomid()**, возвращающие, соответственно случайное число и случайный идентификатор), либо для извлечения значений переменных регулярно-логического выражения [19]. При этом, если переменная имеет единственное значение, результатом будет скалярное строковое значение, но если переменная имеет множество значений, результатом станет список таких значений, оформленный по правилам GNU Prolog. Если переменная не имеет значения, результатом будет пустая строка.

3.4.2.2. Теоретические аспекты

Теорема об адекватном трансформирующем предельном вычислителе ТАПВ3. Наиболее адекватным предельным вычислителем, способным реализовать трансформирующую связку «сканер-макромодуль», является композиционная расширенная машина Тьюринга КомРМТ [21].

Доказательство. Согласно ГПР2 [21], КомРМТ является сильной предельной распознающей объектно-событийной моделью (ОСМ), решающей задачу унификации шаблона, состоящего из комбинаторной группы регулярно-логических выражений, с последующей обработкой результатов некоторой логической программой, осуществляющей прямой или обратный логический вывод. Следовательно, КомРМТ способна унифицировать шаблон сканера, выделить из него данные и обработать логической программой, эквивалентной дедуктивному макромодулю, работающему в режиме обратного логического вывода. Доказано.

Теорема о реализуемости ТРБ3. Расширения языка на базе сканеров и макромодулей являются алгоритмически реализуемыми, в том числе на языках высокого уровня.

Доказательство. Поскольку на языке высокого уровня можно написать КомРМТ, которая, согласно ТАПВ3 может реализовать связку «сканер-макромодуль», то на языке высокого уровня можно написать произвольное расширение языка, построенное на таких связках.

3.4.3. Некоторые потенциальные применения. Примеры

Как уже упоминалось выше, наиболее значимыми применениями связок «сканер-макромодуль» являются ввод новых стандартных вычислительных топологий и построение иных произвольных расширений языка. Вопрос о вводе и применении новых стандартных анонимных топологий будет рассмотрен далее в отдельном пункте, здесь же приведем лишь два небольших примера прочих расширений языка, причем во втором будет применена коллективная работа макромодулей.

Первый пример. Предположим, возникла необходимость ввести в Planning C цикл **while** с обработкой прерывания по **break**, как это сделано в языке Python. Пусть такой цикл имеет общий вид:

```
while (условие) {
        тело_цикла
}
else
        обработчик_прерывания
```

Соответствующая программа с реализацией данной конструкции может иметь вид:

```
#include <iostream>

#scan(WhileElse)

#def_pattern WhileElse => while_else (gid(), /root/COND/@Value,
/root/BODY/@Value) {

(((^)|(\;)+|\}|\{|\)|\\n|(\\n|\\t|\b)else\b|(\\n|\\t|\b)do\b|\:)
((\s|\\t)*\\n)*)(\s|\\t)*
  @begin
    (\s|\\t)*
    (while(\\n|\s|\\t)*\(
           (\\n|\s|\\t)*((.{1,300}?)-
>{COND}\))?=>{Predicates.BAL($,')')}
        )
    (\s|\\t)*
    \{
    (\\n|\s|\\t)*((.{1,8192}?)-
>{BODY}\})?=>{Predicates.BAL($,'}')}
    (\\n|\s|\\t)*
    else
  @end
  (\\n|\s|\\t)+
};

#def_module() while_else(GID, COND, BODY) {
@goal:-brackets_off.
@goal:-
  write('bool __break'), write(GID), write(' = false;'), nl,
  write('while('), write(COND), write(') {'), nl,
  write('  __break'), write(GID), write(' = true;'), nl,
  write(BODY), nl,
  write('  __break'), write(GID), write(' = false;'), nl,
  write('}'), nl,
  write('if (__break'), write(GID), write(') '), nl.
};

int main() {
     int A[10];
     int k = 0;

     while (k < 10) {
          cin >> A[k];
          if (A[k] == 0) break;
          k++;
     }
```

```
      else
            cout << "Был введен ноль!" << endl;

      return 0;
}
```

Второй пример. Предположим, что возникла потребность в организации программных конструкций, реализующих параллельный блок со средствами обмена данными, эквивалентными средствам Parsytec Power XPlorer [37]. Для этого, вообще говоря, достаточно

а) превратить программный блок в анонимную функцию F, принимающую указатели на функции (GET_ROOT, ConnectLink), имитирующие получение конфигурации и установление связи в среде Power XPlorer (прочие функции [SendLink, RecvLink и другие] могут быть реализованы непосредственно в теле функции F);

б) создать ППППВ, в которую передается созданная анонимная функция F, в ней же она вызывается. Данная ППППВ определяет функции GET_ROOT и ConnectLink, указатели на эти функции передаются в F;

в) организовать и запустить вектор из созданных ППППВ.

Пусть конструкция, объявляющая соответствующий параллельный программный блок, выглядит следующим образом:

sun(число_вирт_процессоров) {
тело_блока
}

Напишем заголовочный файл **sun.h**, реализующий данное расширение. При этом заметим, что потребуется не менее двух циклов препроцессинга, поскольку на первом цикле необходимо обнаружить все преобразуемые программные блоки, а на втором – создать анонимные функции – тела блоков, запустить конвейер из ППППВ, вставить соответствующие ППППВ в точку, гарантированно предшествующую по тексту программным функциям, в которых были определены указанные блоки. Такая точка определена упоминанием вызова макромодуля **__sun_defs**(). При этом будет применена коллективная обработка; а) макромодуль **__make_sun** будет генерировать факты **sun_def(ID, NP, BODY)** , содержащие идентификатор **ID**, число требуемых виртуальных процессоров **NP** и, для справки, программный код **BODY** – тело соответствующей

анонимной функции F, б) макромодуль **__sun_defs** на втором цикле препроцессинга будет генерировать ПППВ для каждого факта **sun_def**.

```
#ifndef __SUN_H__
#define __SUN_H__

#preproc_passes(2)

#add_scan(topoSun)

#include <functional>
#include <iostream>
#include <list>
#include <stdio.h>

using namespace std;

#def_module() __sun_defs() {
@goal:-brackets_off.
@make_sun_def(ID, NP, BODY):-
  write('typedef struct { funnel(in, char) * recv; funnel(out,
char) * send; } sun_link_'), write(ID), write(';'), nl,
  write('typedef std::function<sun_link_'), write(ID), write(' *
(int Processor, int RequestId, int * Error)> sun_connector_'),
write(ID), write(';'), nl,
  write('typedef std::function<RootData_t * (void)>
sun_rooter_'), write(ID), write(';'), nl,
  write('typedef std::function<void(sun_rooter_'), write(ID),
write(' GET_ROOT, sun_connector_'), write(ID), write('
ConnectLink)> sun_function_'), write(ID), write(';'), nl,
  write('chain sun_'), write(ID), write('(sun_function_'),
write(ID), write(' worker) {'), nl,
  write('    list<sun_link_'), write(ID), write(' *> __Links;'),
nl,
  write('    RootData_t RootData;'), nl,
  write('    RootProc_t RootProc;'), nl,
  write('    RootData.ProcRoot = &RootProc;'), nl,
  write('    RootProc.nProcs = throw_num_stages();'), nl,
  write('    RootProc.MyProcId = throw_stage();'), nl,
  write('    RootProc.DimX = throw_num_stages();'), nl,
  write('    RootProc.DimY = 1;'), nl,
  write('    RootProc.DimZ = 1;'), nl,
  write('    RootProc.MyX = throw_stage();'), nl,
  write('    RootProc.MyY = 0;'), nl,
  write('    RootProc.MyZ = 0;'), nl,
  write('    sun_rooter_'), write(ID), write(' GET_ROOT = [&]
(void)->RootData_t * {'), nl,
  write('      return &RootData;'), nl,
```

```
  write('    };'), nl,
  write('    sun_connector_'), write(ID), write(' ConnectLink =
[&] (int Processor, int RequestId, int * Error)->sun_link_'),
write(ID), write(' * {'), nl,
  write('      char buf[120];'), nl,
  write('      sun_link_'), write(ID), write(' * L = new
sun_link_'), write(ID), write(';'), nl,
  write('      if (Processor > throw_stage()) {'), nl,
  write('         sprintf(buf, "1+%i+%i+%i", RequestId,
throw_stage(), Processor);'), nl,
  write('         L->recv = new funnel(in, char)(buf);'), nl,
  write('         sprintf(buf, "2+%i+%i+%i", RequestId,
throw_stage(), Processor);'), nl,
  write('         L->send = new funnel(out, char)(buf);'), nl,
  write('      } else {'), nl,
  write('         sprintf(buf, "1+%i+%i+%i", RequestId,
Processor, throw_stage());'), nl,
  write('         L->send = new funnel(out, char)(buf);'), nl,
  write('         sprintf(buf, "2+%i+%i+%i", RequestId,
Processor, throw_stage());'), nl,
  write('         L->recv = new funnel(in, char)(buf);'), nl,
  write('      }'), nl,
  write('      __Links.push_back(L);'), nl,
  write('      *Error = 0;'), nl,
  write('      return L;'), nl,
  write('    };'), nl,
  write('    worker(GET_ROOT, ConnectLink);'), nl,
  write('    for (auto L : __Links) {'), nl,
  write('        delete L->recv;'), nl,
  write('        delete L->send;'), nl,
  write('        delete L;'), nl,
  write('    }'), nl,
  write('}'), nl.
@insert_sun_defs:-
  sun_def(ID, NP, BODY),
  make_sun_def(ID, NP, BODY),
  fail.
@insert_sun_defs:-!.
@goal:-
  write('typedef unsigned char byte;'), nl,
  write('#define MyProcID MyProcId'), nl,
  write('typedef struct ProcInfo_s {'), nl,
  write('        int     nProcs;   /* Число виртуальных
процессоров */'), nl,
  write('        int     MyProcId; /* Идентификатор собственного
процессора */'), nl,
  write('        int     DimX;     /* Размер процессорной кладки
*/'), nl,
  write('        int     DimY;'), nl,
```

```
  write('        int     DimZ;'), nl,
  write('        int     MyX;      /* Позиция в процессорной
кладке */'), nl,
  write('        int     MyY;'), nl,
  write('        int     MyZ;'), nl,
  write('} ProcInfo_t;'), nl,
  write('#define RootProc_t ProcInfo_t'), nl,
  write('typedef struct RootData_s {'), nl,
  write('    RootProc_t * ProcRoot;'), nl,
  write('    int        * DummyRoot;'), nl,
  write('} RootData_t;'), nl,
  insert_sun_defs.
};

__sun_defs()

#def_pattern topoSun => __make_sun (gid(), /root/NP/@Value,
/root/BODY/@Value) {

(((^)|(\;)+|\}|\{|\)|\\n|(\\n|\\t|\b)else\b|(\\n|\\t|\b)do\b|\:)
((\s|\\t)*\\n)*)(\s|\\t)*
  @begin
    sun
    (\s|\\t)*
    \(
    (\s|\\t)*
    (\w+)->{NP}
    (\s|\\t)*
    \)
    (\s|\\t)*
    \{
     ((.{1,8192})->{BODY}\})?=>{Predicates.BAL($,'}')}
    (\s|\\t)*
  @end
};

#def_module() __make_sun(ID, NP, BODY) {
@goal:-brackets_off.
@goal:-assertz(sun_def(ID, NP, BODY)).
@make_sun_body:-
    write('sun_function_'), write(ID), write(' worker_'),
write(ID), write(' = [&] (sun_rooter_'), write(ID), write('
GET_ROOT, sun_connector_'), write(ID), write(' ConnectLink) {'),
nl,
    write('typedef sun_link_'), write(ID), write(' LinkCB_t;'),
nl,
    write('typedef sun_link_'), write(ID), write(' *
Option_t;'), nl,
```

```
    write('auto SendLink = [&] (sun_link_'), write(ID), write('
* Link, byte * data, long Size) {'), nl,
    write('  Link->send->put(data, Size);'), nl,
    write('  while (!Link->send->empty()) _Yield();'), nl,
    write('};'), nl,
    write('auto RecvLink = [&] (sun_link_'), write(ID), write('
* Link, byte * data, long Size) {'), nl,
    write('  Link->recv->get(data, Size);'), nl,
    write('};'), nl,
    write('auto ReceiveOption = [&] (LinkCB_t * L)->Option_t {
return L; };'), nl,
    write('auto SelectList = [&] (int n, Option_t * Options)-
>int {'), nl,
    write('  while(1) {'), nl,
    write('    for (int i = 0; i < n; i++)'), nl,
    write('        if (!Options[i]->recv->empty()) return i;'),
nl,
    write('  }'), nl,
    write('};'), nl,
    write(BODY), nl,
    write('};'), nl.
@make_sun_call:-
    write('plan_parallel_chain(1, '), write(NP), write(',
sun_'), write(ID), write('(worker_'), write(ID), write('));'),
nl.
@goal:-make_sun_body, make_sun_call, !.
};

#endif
```


Вызывающая программа (топология «звезда», мастер-процессор суммирует и выводит на экран идентификаторы подчиненных процессоров), использующая данное расширение, может выглядеть так:

```
#include "sun.h"

using namespace std;

#define NP 5

int main() {
    sun(NP) {
       int nProcs   = GET_ROOT()->ProcRoot->nProcs;
       int MyProcID = GET_ROOT()->ProcRoot->MyProcID;

       int error;
```

```
       if (MyProcID == 0) {
          LinkCB_t ** LinkList = (LinkCB_t **) malloc((nProcs-
1)*sizeof(LinkCB_t *));
          Option_t *  Options  = (Option_t *)  malloc((nProcs-
1)*sizeof(Option_t));

          int   Slave;
          int   f,g,v;
          int   sum = 0;

          for (f=0;f<nProcs-1;f++)
              {
               LinkList[f] = ConnectLink(f+1,1234,&error);
               Options[f]  = ReceiveOption(LinkList[f]);
              }

          for (g=0;g<nProcs-1;g++)
              {
                Slave = SelectList(nProcs-1,Options);
                RecvLink(LinkList[Slave],(byte *) &v,
sizeof(v));
                sum += v;
              }
          printf("Sum = %i\n", sum);

          free (LinkList);
          free (Options);
       } else {
          LinkCB_t * L = ConnectLink(0,1234,&error);
          SendLink(L,(byte *) &MyProcID, sizeof(MyProcID));
       }
    };
}
```


В заключение раздела по связке «сканеры-макромодули», упомянем, что с их помощью возможна реализация и более сложных идей, рассмотрение которых несколько выходит за рамки данной работы. В частности, речь может идти о порождающем программировании, когда создается ряд вышеуказанных связок, позволяющих создавать программу, начиная с некоторого декларативного высокоуровневого описания, которое последовательно распознается и неоднократно трансформируется, переводя исходную задачу на все более и более близкий к программному коду уровень (при этом могут быть использованы алгоритмы по типу описанных в работе [19]).

Также речь может идти о самомодификациях и самосогласованиях программы в результате применения таких связок, например, сканеры могут обнаружить реализацию алгоритма пузырьковой сортировки и вызвать макромодуль, замещающий такой фрагмент на реализацию быстрой сортировки. Также речь может идти об автоматическом выборе вычислительной топологии под решаемую задачу со вставкой соответствующих реализующих фрагментов кода.

Еще одним возможным применением может стать верификация алгоритмов работы программы: а) сканеры выделяют реализованные алгоритмы и сворачивают их в вызовы неких концептов, б) определяется связь между концептами, в) данные действия повторяются, пока не будет получен финальный граф концептов, который может быть проанализирован на соответствие постановке решаемой программой задачи.

3.5. Элементы функционального программирования. Библиотека reentera.h

Рассмотренные в предыдущих пунктах возможности расширения языка Planning C были применены для реализации ряда анонимных конструктов, а именно – ПППВ/ФППВ и ряда формализмов, включающих ПППВ/ФППВ в качестве элементов (цепи, топологии и т.п.). Проблема состояла в том, что первая версия Planning C допускала подобные определения не всегда и, если допускала, то достаточно формально, поскольку воспользоваться ими на практике было достаточно затруднительно. Связано это с рядом особенностей реализованного транслятора Planning C, который неявно добавлял в вызовы ПППВ/ФППВ ряд необходимых служебных параметров, а определения топологий, например, просто разворачивал в inline-фрагмент исполняемого кода.

Были написаны соответствующие сканеры, выделяющие различные анонимные определения (обычно в формате, подобном **auto NAME = A;**) и сворачивающие их в вызовы макросов, и сами макросы, генерирующие

а) тела анонимных конструктов в виде обычных анонимных функций **A**;

б) обычные ПППВ/ФППВ, являющиеся обертками **B**(**A**), вызывающими тела анонимных конструктов **A**;

в) при необходимости – прочие данные, в том числе auto-декларации переменных **NAME**, принимающих ссылки на анонимные функции **C**(**B**), также являющиеся обертками, вызывающими сгенерированные ПППВ/ФППВ (в которые передаются анонимные функции – тела анонимных конструктов **A**) **B**(**A**).

Соответствующие сканеры, макросы и необходимые директивы препроцессора были оформлены в виде подключаемого файла **reentera.h**.

ВАЖНОЕ ЗАМЕЧАНИЕ. Чтобы всегда иметь возможность различить специальные конструкции языка, относящиеся к анонимным и обычным формализмам, а также в связи с некоторыми особенностями примененного интерпретатора GNU Prolog, были введены специальные правила именования, сведенные в следующую таблицу 3.1.

Таблица 3.1

Сопоставление идентификаторов однотипных конструкций для анонимных и обычных формализмов

Для обычных формализмов	**Для анонимных формализмов**
plan_first	reent_first
plan_last	reent_last
plan_group_soft_atomize	reent_group_soft_atomize
plan_group_first	reent_group_first
plan_group_last	reent_group_last
plan_group_parallelize	reent_group_parallelize
clear_plan	clear_reent
plan_processor_id	reent_processor_id
plan_processors	reent_processors
throw_first	reent_next_first
throw_last	reent_next_last
throw_stage	reent_stage
throw_num_stages	reent_num_stages
plan_linear_num	reent_linear_num
plan_topology_num	reent_topology_num
plan_topology_quit	reent_topology_quit
plan_neighbours	reent_neighbours

3.5.1. Анонимные процедуры и функции с повторным входом

Рассмотрим общий формат определения анонимных ПППВ/ФППВ. Прежде всего, следует отметить, что по соображениям упрощения транслятора (и ввиду отсутствия какой-либо реальной потребности в более сложной реализации) сделано допущение о том, что переменная, принимающая ссылку на такой конструкт, всегда имеет тип **auto**[1], соответственно

декларация ПППВ/ФППВ = «**auto**» пробелы идентификатор_ссылочной_переменной [пробелы] «**=**» [пробелы] «**reenterable**» пробелы «(» [декларации_параметров] «)» [пробелы «**->**» тип_результата] пробелы «**{**» тело_процедуры_или_функции «**}**» «**;**»

Следует сделать замечание относительно реализации сканирующих макросов. Существенную роль имеет особый сканер **scan_fdefs**, выделяющий текущий блок программы (текущую процедуру или функцию), точнее, его заголовок **H**. Это необходимо для определения точки, перед которой макросом **make_fdef** будут вставляться определения новых служебных ПППВ/ФППВ, генерируемые макросами для реализации анонимных конструктов, определяемых в блоке с заголовком **H**.

Далее приведен пример программы, иллюстрирующий декларацию и применение анонимной ПППВ, выполняющей цикл (с выводом на экран значения счетчика) от значения параметра **cur** до значения параметра **last**.

```
#include <iostream>
#include "reentera.h"

using namespace std;

int main() {
   auto f = reenterable (int cur, int last)  {
      cout << cur << " ";
      if (cur < last)
         reent_first(cur+1, last);
      else
         cout << endl;
   };
```

[1] То же самое справедливо для многих анонимных конструктов, о которых речь пойдет далее.

```
   f(0, 5);

   return 0;
}
```

3.5.2. Анонимные вектора и цепи

Здесь необходимо различать два случая: а) вектора или конвейера (все элементы цепи имеют одинаковые тела и параметры) и б) цепи общего вида (элементы имеют разные тела и/или параметры).

3.5.2.1. Вектора и конвейеры

Синтаксис для такого случая следующий, напоминающий декларацию простой анонимной ППП/ФППВ:

вектор/конвейер = «**auto**» пробелы идентификатор_ссылочной_переменной [пробелы] «**=**» [пробелы] «**chain**» [пробелы] «**[**» [пробелы] число_элементов [пробелы] «**]**» [пробелы] «**<**» декларации_параметров «**>**» [пробелы] «**{**» тело_элемента_цепи «**}**» «**;**»

Все элементы обязательно имеют инициализационный этап. Запуск такого вектора/конвейера осуществляется в формате, эквивалентном вызову обычной процедуры (void-функции):

идентификатор_ссылочной_переменной «(» значения_параметров_через_запятую «)» «**;**»

Приведем пример программы с конвейером из четырех элементов:

```
#include <iostream>
#include "reentera.h"

using namespace std;

int main() {
  auto c = chain[4]<bool init, int val, int from> {
    if (reent_stage() == 0 && init)
       cout << reent_stage() << " is first. Send " <<
(reent_stage()+1) << endl;
    else if (!init)
       cout << reent_stage() << ": [" << val << "] received from
" << from << endl;
```

```
    if (init && reent_stage() < reent_num_stages()-1)
       reent_next_last(false, reent_stage()+1, reent_stage());
  };

  c(true, 0, -1);

  return 0;
}
```

Данная программа выводит на экран:

```
0 is first. Send 1
2: [2] received from 1
1: [1] received from 0
3: [3] received from 2
```

3.5.2.2. Цепи общего вида с неоднородными элементами

В данном случае ссылочная переменная не создается, фактически, записывается лишь вызов цепи с неявно создаваемыми анонимными ПППВ (по причине достаточной сложности указаний множеств параметров вызова в случае, если декларация и вызов цепи разделены). Здесь каждый элемент цепи (которых должно быть не менее одного) имеет собственный список параметров и инициализирующие значения. Элементы в цепи следуют друг за другом в том же порядке, в котором они перечисляются. Все элементы обязательно имеют инициализационный этап. Синтаксис следующий:

цепь_общего_вида = «**chain**» [пробелы] «**<**» декларации_параметров «**>**» [пробелы] «(» значения_параметров «)» [пробелы] «**{**» тело_элемента_цепи «**}**»

{ «**chain**» [пробелы] «**<**» декларации_параметров «**>**» [пробелы] «(» значения_параметров «)» [пробелы] «{» тело_элемента_цепи «**}**» } «**;**»

Рассмотрим пример программы с цепью общего вида из трех неоднородных элементов:

```
#include <iostream>
#include "reentera.h"

using namespace std;

int main() {
```

```
  chain<bool init> (true) {
    reent_next_first(false, 1);
  }
  chain<bool init, int N> (true, 0) {
    if (!init) {
       reent_next_first(false, N+1, "Calculated: ");
    }
  }
  chain<bool init, int N, char * msg> (true, 0, NULL) {
    if (!init) {
       cout<<msg<<N;
    }
  };

  return 0;
}
```

В результате работы программы на экран выводится сообщение:

```
Calculated: 2
```


3.5.3. Анонимные топологии общего вида

Остается определить синтаксические правила оформления анонимных топологий общего вида. Пусть это будут исключительно однородные топологии, тела элементов которых имеют идентичный код (как можно легко показать, с помощью такого подхода реализуемы и топологии с неоднородными элементами, достаточно конкатенировать их код в единую конструкцию, в которой вызов нужного фрагмента определяется с помощью некоторых управляющих параметров). Тогда синтаксис анонимных топологий может быть достаточно компактным, остается только решить возникающую проблемы *идентификации направлений передачи данных*, которая для неанонимных топологий решалась указанием имени ПППВ и ее индекса в топологии. Здесь же воспользуемся специальной конструкцией вида

this[индекс_элемента_топологии].

Резюмируя вышесказанное, синтаксис анонимной топологии общего вида следующий:

топология_общего_вида = «**auto**» пробелы идентификатор_ссылочной_переменной [пробелы] «**=**» [пробелы] «**chain**» [пробелы]

«**[**» «**]**» [пробелы] «**<**» декларации_параметров «**>**» [пробелы] «**{**» тело_элемента «**}**»

«**{**» описатель {«**,**» описатель} «**}**» «**;**»

описатель = [пробелы] (прямой_маршрут | обратный маршрут) [пробелы]

прямой_маршрут = индекс [пробелы] «**->**» [пробелы] индекс { [пробелы] «**->**» [пробелы] индекс }

обратный_маршрут = индекс [пробелы] «**<-**» [пробелы] индекс { [пробелы] «**<-**» [пробелы] индекс }

индекс = индекс_элемента_топологии

Для определений таких топологий действует то же *правило*, что и для обычных топологий: циклы допускаются, но в каждом цикле должен присутствовать хотя бы один обратный маршрут таким образом, чтобы по всем прямым маршрутам топология представляла бы собой лишь *сеть*. Индексация элементов в описателях ведется таким же образом, как и в обычных топологиях.

Запуск такой топологии осуществляется в формате, эквивалентном вызову обычной процедуры (void-функции):

идентификатор_ссылочной_переменной «(» значения_параметров_через_запятую «)» «**;**»

Приведем пример программы, реализующей анонимную топологию «кольцо»:


```
#include "reentera.h"
#include <iostream>

using namespace std;

const int NP = 5;

int main() {
    int SUM = 0;

    auto c = chain[] <input_proc Src, int from, int val> {
      int id = reent_linear_num();

      if ((id == 1) ^ (from >= 1)) {
         reent_next_first(this[1 + id % NP], id, val+1);
      }
      if (from >= 1) {
         SUM += val;
```

```
        if (id == 1) {
            cout << SUM << endl;
            reent_topology_quit();
        }
      }
    } {
      1 -> 2 -> 3,
      5<-4<-3, 5->1
    };

    c(empty_proc, 0, 39);

    return 0;
}
```

Данная программа в результате работы выводит на экран число «**210**».

3.6. Стандартные топологии. Библиотека stdtopo.h

Как уже упоминалось выше, идея ввода в язык понятия анонимных стандартных топологий изначально была одной из основополагающих при разработке механизма связок «сканеры-макромодули». Была разработана соответствующая библиотека-расширение **stdtopo.h**, содержащая определения ряда стандартных топологий и вспомогательных функций, позволяющих быстро определить индексы элементов, которым необходимо передать данные. Сведем соответствующую информацию в таблицу 3.2.

Таблица 3.2

Разработанные стандартные топологии и вспомогательные функции

Топология	Идентификатор	Вспомогательные функции
Гиперкуб	**hypercube**	- get_idxs(индекс) – возвращает вектор индексов в гиперкубе по линейному индексу; - get_idx(вектор_индексов) – возвращает линейный индекс по вектору индексов в гиперкубе.
Полный граф	**clique**	- num_others() – возвращает количество элементов топологии, с которыми связан текущий элемент; - other(номер) – возвращает линейный индекс связанного элемента с указанным номером.
Двусвязное кольцо	**ring**	- forw() – возвращает линейный индекс следующего элемента по часовой стрелке; - back() – возвращает линейный индекс следующего элемента против часовой стрелки.
Двусвязная труба	**tube**	- forw() – возвращает линейный индекс следующего элемента (справа); - back() – возвращает линейный индекс предыдущего элемента (слева).
Пирамида	**pyramide**	- forw() – возвращает линейный индекс следующего подчиненного по часовой стрелке; - back() – возвращает линейный индекс следующего подчиненного против часовой стрелки; - master() – возвращает линейный индекс центрального элемента; - num_slaves() – возвращает количество подчиненных элементов (в основании пирамиды – кольце);

		- slave(номер) – возвращает линейный индекс подчиненного с указанным номером.
Звезда	**star**	- master() – возвращает линейный индекс центрального элемента; - num_slaves() – возаращает количество подчиненных элементов; - slave(номер) – возвращает линейный индекс подчиненного с указанным номером.

При необходимости набор стандартных топологий может быть легко расширен путем простого редактирования файла stdtopo.h. Для этого достаточно выполнить следующие действия:

1. Отредактировать определение макроса make_std_topo, добавив утверждение @write_defs(идентификатор_топологии), которое будет помещать в генерируемый код анонимные вспомогательные функции, позволяющие быстро определить индексы элементов-партнеров по коммуникации. Если таких функций не предполагается, можно ограничиться утверждением вида

@write_defs(идентификатор_топологии):-!.

2. Добавить определение макромодуля, идентификатор которого совпадает с идентификатором топологии, а первые N-2 параметров (где N – общее количество параметров) – определяют набор метапараметров, которые указываются при объявлении топологии. Последние два параметра всегда одинаковы – это внутреннее имя генерируемой ПППВ – элемента топологии и набор переданных в него аргументов. Данный макромодуль должен генерировать определения цепей plan_parallel_chain и plan_parallel_reverse, которые описывают генерируемую топологию. Приведем пример такого макромодуля для топологии «двусвязное кольцо»:

```
#def_module(plan_topology) ring(N, Proc, Args) {
@write_chain(1):-
    !.
@write_chain(NN):-
    NN1 is NN-1,
    write_chain(NN1),
```

```
    write('  plan_parallel_chain('), write(Proc), write('['),
write(NN1), write(']'), write(Args), write('->'),
    write(Proc), write('['), write(NN), write(']'), write(Args),
write(');'),
    write('  plan_parallel_reverse('), write(Proc), write('['),
write(NN), write(']'), write(Args), write('->'),
    write(Proc), write('['), write(NN1), write(']'),
write(Args), write(');'),
    nl.
@goal:-
    write_chain(N),
    write('  plan_parallel_reverse('), write(Proc), write('['),
write(N), write(']'), write(Args), write('->'),
    write(Proc), write('[1]'), write(Args), write(');'),
    write('  plan_parallel_reverse('), write(Proc),
write('[1]'), write(Args), write('->'),
    write(Proc), write('['), write(N), write(']'), write(Args),
write(');'),
    nl.
};
```

Более подробную информацию можно получить непосредственно из примеров стандартных топологий, уже реализованных в stdtopo.h.

Синтаксис определения анонимной стандартной топологии в программе на Planning C следующий:

анонимная_станд_топология = «**auto**» пробелы идентификатор_ссылочной_переменной [пробелы] «**=**» [пробелы] «**topology**» [пробелы] идентификатор_топологии [пробелы] «**[**» значения_метапараметров_топологии «**]**» [пробелы] «**<**» декларации_параметров_тела «**>**» [пробелы] «**{**» тело_элемента_топологии «**}**» «**;**»

Необходимо заметить, что всякая стандартная топология является, на уровне реализации, топологией общего вида и требует явного завершения своей работы вызовом встроенной функции **reent_topology_quit()**.

Рассмотрим пример со стандартной топологией «двусвязное кольцо» из пяти элементов:

```
#include "stdtopo.h"
#include <iostream>

using namespace std;

const int NP = 5;
```

```
int main() {
    int SUM = 0;

    auto t = topology ring[NP] <input_proc Src, int from, int
val> {
      int id = reent_linear_num();

      if ((id == 1) ^ (from >= 1)) {
         reent_next_first(this[forw()], id, val+1);
      }
      if (from >= 1) {
         SUM += val;
         if (id == 1) {
            cout << SUM << endl;
            reent_topology_quit();
         }
      }
    };

    t(empty_proc, 0, 39);

    return 0;
}
```


Данная программа выводит на экран число «**210**».

Выводы к третьей главе

В данной главе впервые предложен новый вид технологических средств предикции для использовании в параллельном программировании – предикционно-решающие каналы. Предложены два вида каналов – авторегрессионные точечные и линейные (явные или неявные) коллективного решения. Изложены математические аспекты предикции в таких каналах, кратко описаны базовые средства программирования.

Совмещение каналов со средствами предсказания данных упрощает программирование ряда алгоритмов, связанных с численным моделированием (прогнозирования времени счета для балансировки загрузки, расчета прогнозных значений переменных, например, на стыках блоков расчетной области при параллельной обработке, и других алгоритмов), и позволяет, в ряде случаев, осуществлять скрытые переходы от явных

разностных схем к неявным, отличающимся большей устойчивостью счета, а также от последовательных алгоритмов счета к параллельным.

Введено понятие частично транзакционной памяти, сочетающей классические и транзакционные переменные, создан ряд классов транзакционных согласуемых переменных, определены особенности функционирования авторегрессионных точечных каналов в условиях согласования с откатами. Предложены два способа запуска исполнения в режиме работы с частично транзакционной памятью: более универсальный, который может быть применен в любом языке программирования, и специализированный, реализованный в рамках концепции ППП/ФППВ. Как будет показано далее, применение каналов в режиме частично транзакционной памяти позволяет реализовать достаточно интересный режим параллельной работы – сверхоптимистичные вычисления.

Предложены средства, поддерживающие мемоизацию процедур/функций. Введены соответствующие языковые конструкции. Предложена концепция интерполирующей мемоизации, позволяющей предсказать результат мемоизации даже в случае отсутствия текущего набора аргументов в кэше. При этом используется нейронная сеть прямого распространения, линейный экстраполятор или МГУА. Сформулированы алгоритмы, позволяющие контролировать точность предикции и гибко регулировать интервалы доверия ее результатам. Приведены примеры, демонстрирующие как классическую мемоизацию, так и мемоизацию с обучением единственного предиктора или группы предикторов, индексируемых по значению выбранного группирующего параметра. Показано, что для *некоторых задач из области вычислительной математики*, интерполирующая мемоизация с предиктором способна давать ускорение в 1,32÷8,8 раза.

Сформулированы новые средства, позволяющие оперативно вводить в язык новые стандартные вычислительные топологии, а также строить иные произвольные расширения языка (новые конструкции, элементы аспектно-ориентированного и метапрограммирования) с применением групп регулярно-логических выражений (объединенных в сканирующие макросы – сканеры, индентифицирующие требуемые языковые фрагменты) и, связанных с ними, многократно согласуемых дедуктивных макромодулей. Приведен синтаксис новых конструкций, до-

казана их алгоритмическая реализуемость на языках высокого уровня. Подробно описана стадия препроцессига программ на Planning C, учитывающая наличие сканеров в связке с макромодулями. Даны примеры реализации простых языковых расширений, потенциально способных дополнительно адаптировать язык программирования к решаемой задаче или к запросам программиста. Сформулирован ряд идей по возможности интеллектуализации обработки программ с применением связок «сканеры-макромодули», в частности, поддержки самомодификации программ и верификации реализованных алгоритмов.

С помощью сканеров и макромодулей в язык введены элементы функционального программирования, а именно – анонимные ППП/ФППВ, вектора/конвейеры, цепи и топологии общего вида, возможность обращения к указанным элементам через ссылочные переменные. Предложены соответствующие синтаксические конструкции.

Аналогичным образом в язык введен ряд специальных стандартных топологий, сформулирован соответствующий синтаксис, приведен пример применения. Рассмотрен вопрос о возможности ввода новых стандартных топологий.

Глава 4. Приемы программирования, специфичные для Planning C

Целью данной главы является дальнейшее повышение эффективности как процесса программирования, так и исполнения получаемого программного кода, используя новые, специфические приемы программирования на Planning C. С данной целью сформулируем следующие задачи:

а) предложить алгоритмическую схему дополнительного распараллеливания задачи обучения глубокой нейронной сети методом обратного распространений, а также некоторых задач численного моделирования, используя концепцию программирования с применением предицирующих каналов в условиях частично транзакционной памяти;

б) рассмотреть возможность непосредственного применения предикторов интерполирующей мемоизации в качестве встроенных в язык нейронных сетей, линейных предикторов и МГУА-полиномов;

в) предложить высокоуровневую алгебраическую нотацию сокращенной записи топологий и средства ее трансляции в программный код.

4.1. Распараллеливание циклов с зависимыми витками. Сверхоптимистичные вычисления

В данном разделе, на примере решения задачи об обучении нейронной сети методом обратного распространения ошибки, будет предложена общая схема дополнительного распараллеливания, которая также будет применена к решению еще двух задач из области численного моделирования.

Заметим, что идеи, реализованные в данной работе, в наибольшей степени схожи с идеями работы [48], где также идет речь о распараллеливании циклов с программным прогнозированием значений переменных. Однако подход [48] не предполагает контроля программистом и не опирается на идеи прогностических каналов, работающих в транзакционной памяти, – распараллеливание выполняется компилятором и его детали скрыты. Также есть отличия в механизме предикции – в работе

[48] используется простое распознавание шаблонных последовательностей значений в прогнозируемых переменных без попытки их приближения нейронными сетями. Кроме того, данная стратегия ориентирована на целочисленную предикцию (как и в работах [55, 63], предполагающих применение предикции для оптимизации вычислений) и мало применима для математических задач с вещественными переменными.

Схожей работой также является [42], в которой выполняется предикция результатов серии машинных инструкций с отменой. Однако в [42] механизмы отмены менее формализованы (в отличие от данной работы, в которой используется формализм частично транзакционной памяти) – отмена выполняется контекстно-зависимым кодом «ad hoc». Случай распараллеливания кода более чем на две параллельные стадии в [42] также не рассматривается. Распараллеливание также выполняется компилятором и не контролируется программистом. Стратегия [42] также ориентирована лишь на целочисленную предикцию.

С [42] схожа работа [47], в которой программная предикция выполняется с проверкой аппаратным предиктором процессора, однако следует заметить, что такой подход существенно ограничен аппаратными возможностями процессора. Кроме того, такой подход не обобщается на случай более чем 2 параллельных стадий, в отличие от изложенного в данной работе.

4.1.1. Задача об обучении нейронной сети

Известно, что метод обратного распространения ошибки для сетей прямого распространения [25] представляет собой цикл эпох обучения, на каждой итерации которого выполняется подстройка весов и смещений путем последовательного или параллельного прохода по всем обучающим парам. При параллельном проходе сначала для каждой пары вычисляются локальные поправки ко всем весам и смещениям, в конце эпохи данные поправки усредняются, и только в таком виде применяются к весам и смещениям. В таком случае можно использовать не более P ядер, где P – количество обучающих пар, с максимальным теоретическим ускорением в P раз. Однако можно добиться еще большего ускорения, если дополнительно распараллелить процесс вычисления локаль-

ных поправок к весам и смещениям для произвольной обучающей пары. Этот процесс является двухстадийным: на первой стадии вычисляется отклик сети, на второй вычисляются собственно поправки. Таким образом, по отношению к каждой отдельно взятой обучающей паре имеем цикл по эпохам, причем тело цикла содержит две последовательные части, а витки цикла являются зависимыми. Схематично такой цикл можно записать в следующей форме:

```
for (int i = 0; i < Число_эпох; i++) {
      Y = вычисление_отклика(X, веса, смещения);
      веса, смещения = вычисление_поправок(Y);
}
```

или в общем виде:

```
for (int i = 0; i < N; i++) {
      Y = f(X);
      X = g(Y);
}
```

Такой цикл характерен не только для задачи обучения нейронной сети, но и для других математических алгоритмов, например для метода частиц в ячейках [30], где на первой стадии тела цикла вычисляем перемещение частиц и определяемую ими сеточную функцию, а на второй стадии решаем уравнение в частных производных, правая часть которого содержит эту сеточную функцию. Заметим, что интерес также представляют циклы, тела которых могут быть разбиты на более чем две стадии – такой случай также будет рассмотрен в данной работе.

4.1.2. Идея распараллеливания. Оптимизация распараллеливания на базе аналитической модели

Напрямую распараллелить цикл указанного выше вида нельзя, но можно применить некоторые вспомогательные приемы, основанные на замене внутренних для цикла зависимостей по данным локальным предсказанием таких данных. Запустим первую и вторую стадию цикла параллельно и организуем между ними предсказывающий канал передачи данных, отличающийся от обычного канала наличием предиктора, который при отсутствии в канале данных генерирует прогнозные значения таких данных, основываясь на анализе ряда ранее поступивших в канал значений. В таком случае целесообразно воспользоваться достаточно

простыми методами, такими как линейная нецелочисленная предикция, чтобы не вносить больших дополнительных накладных расходов (следует заметить, что в схожих работах достаточно популярна малозатратная целочисленная предикция (по последнему значению [49], с шагом [39], контекстная [56], или гибридная [39]), однако в нашем случае она неприменима). Данный канал будет генерировать прогнозные значения Y^* в начале второй стадии и позволит вычислять вторую стадию параллельно с первой (обозначим такую схему расчета как «сверхоптимистичные вычисления» [18]).

Прогнозное значение $Y^*[k]$ будет вычисляться как

$$Y^*[k] = a_0 + \sum_{m=1}^{P} a_m Y[k-m],$$

где k – номер итерации цикла, a – вектор коэффициентов предикции, Y[p] – история реально полученных значений Y с p-й предыдущей итерации. Коэффициенты предикции определяются по истории данных с помощью метода наименьших квадратов. Данные коэффициенты периодически пересчитываются (в случае неоднократного стойкого расхождения прогнозных и реально полученных значений). Обычно такой пересчет происходит один раз в 50÷60 итераций. Чем меньше пересчетов, тем более эффективно работает предлагаемая схема.

В заключение тела цикла полученное в первой стадии значение Y будет сверяться с прогнозным значением Y^*: если относительная погрешность невелика (менее некоторой заданной величины ε, обычно ε = 0,01...0,1), то вычисление второй стадии можно принять ($Y = Y^*$, стадии выполнятся параллельно), если же погрешность велика, то результаты второй стадии отменяются и вторая стадия пересчитывается (при этом из канала будет читаться уже не прогнозное, а реальное принятое значение Y) на основе вычисленного в первой стадии значения Y, при этом стадии фактически выполнятся последовательно. Такие согласования и отмены удобно реализовать с применением формализма транзакционной памяти [33] в частичной форме. Ранее уже было введено понятие частично транзакционной памяти – это память, в которой присутствуют как согласуемые, так и обычные переменные и особым образом обрабатываются специальные переменные – предсказывающие каналы: в конце транзакции для каналов осуществляется контроль расхождения прогнозных значений (возможно, использовавшихся в расчете) и значе-

ний, реально полученных каналом. Такое расхождение (если оно больше ε), становится основанием для пересогласования транзакции, в которой присутствовал «приемный» конец канала. В таком случае тело цикла приобретает вид:

```
for (int i = 0; i < N; i++) {
      Включаем частично транзакционную память {
            Транзакция_1: {
                  Y = f(X); // Например, вычисляем отклик сети
                  Предсказывающий канал.put(Y);
            }
            Транзакция_2: {
                  Предсказывающий канал.get(Y);
                  X = g(Y); // Например, пересчитываем веса и
            смещения сети
            }
      }
}
```

При таком подходе теоретически возможно получить дополнительное ускорение в два раза. Однако на практике ускорение будет значительно меньше, поскольку часть времени расходуется на работу каналов (на предикцию и на вычисление коэффициентов предикторов) и частично транзакционной памяти, часть итераций не пройдет контроль по погрешности предикции и сведется к последовательному выполнению стадий, и, наконец, трудоемкости выполнения стадий должны быть приблизительно равны.

Первое обстоятельство является весьма существенным и может быть снято путем привлечения дополнительных вычислительных ресурсов, например, потоковых процессоров графических видеоускорителей. Второе обстоятельство является неизбежным и, наконец, последнее обстоятельство может быть устранено или адекватным выбором точки разбиения цикла на две части или, в какой-то степени, с помощью дополнительного внутреннего распараллеливания этих частей (если оно возможно) с адекватным распределением ядер вычислительной системы между частями с целью балансировки загрузки и, как следствие, минимизации общего времени счета (этот вопрос будет рассмотрен далее).

Минимизация общего времени счета производится путем автоматического выбора распределения ядер по K стадиям цикла на базе аналитических моделей времени счета (без предикции $T_k(N_k)$ и с предикцией $\tilde{T}_k(N_k)$) с эмпирическими коэффициентами, которые подбираются ме-

тодом наименьших квадратов по динамически собираемой статистике времени счета (при различных количествах ядер N_k на k-й стадии, автоматически варьируемых средой исполнения) из условия максимального правдоподобия. Данные модели, помимо обычных линейных членов, содержат член $c_k N_k$, выражающий наличие накладных расходов на организацию параллельной работы. Модели являются трехпараметрическими и имеют общий вид:

$$T_k(N_k) = a_k + \frac{b_k}{N_k} + c_k N_k;$$

$$\tilde{T}_k(N_k) = \mu_k + \frac{\gamma_k}{N_k} + \rho_k N_k,$$

где a_k, b_k, c_k, μ_k, γ_k, ρ_k – эмпирические коэффициенты, N_k – число задействованных ядер.

Полная задача целочисленной минимизации времени F (для общего случая, когда тело цикла может быть разделено на произвольное число стадий $K \geq 2$) имеет вид:

$$F(N_1,\ldots N_K) = \sum_{p=1}^{K} \beta_p \max\left[T_p(N_p), \tilde{T}_{p+1}(N_{p+1}),\ldots,\tilde{T}_K(N_K)\right];$$

$$F(N_1,\ldots N_K) \to \min;$$

$$N_k \in N;\ k = \overline{1,K};$$

$$\sum_k N_k \leq M,$$

где β_p – эмпирические коэффициенты, причем $\beta_1 = 1$, а коэффициенты с $p > 1$ отражают статистически определенную вероятность пересчета транзакции (при «промахе» предиктора) на p-й стадии, N – множество натуральных чисел, M – общее число ядер. Функция времени учитывает возможность работы транзакционного блока как с пересчетом транзакции, так и без него. Вероятности пересчета транзакции определяются динамически, путем сбора соответствующей статистики (выполняется средой исполнения автоматически) в процессе исполнения цикла.

Поиск минимума при $K = 2$ может осуществляться методом полного перебора, при $K > 2$ возможно использование варианта градиентного алгоритма:

1. Пусть N_k – число ядер на k-ю стадию. Изначально $\forall k$: $N_k = 1$.
2. Пока сумма N_k меньше общего числа ядер M:

2.1. Рассчитать $\forall k$: $Q(k) = F(N_1,\ldots,N_k+1,\ldots N_K)$.
2.2. Найти $p = \arg\min Q(k)$.
2.3. Увеличить N_p на единицу.

4.1.3. Апробация

Расмотрим задачи обучения нейронной сети прямого распространения, численного моделирования электростатической линзы и численного моделирования распространения тепла в тонком стержне с применением неявной разностной схемы. Эксперименты проведем на стандартной многоядерной (8 физических ядер процессора Xeon, 16 логических ядер) машине проекта Google's Compute Engine. Во всех экспериментах будем вести контроль дополнительной погрешности, вносимой режимом сверхоптимистичных вычислений (как будет показано далее, погрешность не превысила 3,5%).

4.1.3.1. Обучение нейронной сети

Решение данной задачи в общих чертах уже было описано выше: на первой стадии вычислялось значение отклика сети, которое через предсказывающий канал передавалось на вторую стадию – стадию коррекции весов и смещений сети. Обучалась пятислойная сеть прямого распространения из 79 (15, 20, 28, 15, 1 в различных слоях) нейронов (с нелинейными передаточными функциями «экспоненциальная сигмоида», за исключением линейного выходного слоя) на выборке из 217 обучающих пар, описывающих зависимость турбулентной вязкости от четырех параметров течения газа в различных точках расчетной области (аэродинамическая задача). Сеть имела 4 входа и один выход. Использовался стандартный метод обратного распространения, распараллеленный тремя различными способами (см. таблицу 4.1), включающими изложенный в данной работе. В первом режиме реализовывался описанный в данной работе подход без динамического решения вышеописанной задачи оптимального распределения ядер. Второй режим реализует тот же подход, но с решением задачи оптимального распределения ядер. Третий

режим соответствовал классическому варианту распараллеливания по обучающим парам без прогнозирования, без использования подхода, изложенного в данной работе. Для вычисления показателей ускорения дополнительно был произведен расчет той же задачи в непараллельном варианте (определялось время непараллельного решения t(1)). Ускорение S(M) на M ядрах рассчитывалось по стандартной формуле

$$S(M) = t(1)/t(M),$$

где t(M) – соответствующее время решения на M ядрах.

В таблице 4.1 приведены рассчитанные значения ускорения для различных режимов и числа ядер.

Таблица 4.1

Полученные данные по обучению нейронной сети

Режим	**Ускорение для различного количества ядер M**								
	16	14	12	10	8	6	4	2	1
Сверхоптимистичные вычисления, с автоматическим распределением ядер по стадиям	2,98	2,83	2,55	2,83	2,93	3,57	2,96	1,42	1
Сверхоптимистичные вычисления, с равномерным распределением ядер по стадиям	2,35	2,36	2,43	2,53	3,27	2,08	2,63	1,44	1
Обычный параллельный расчет без сверхоптимистичных вычислений	-	-	-	-	2,01	2,51	2,28	1,7	1

Из таблицы 4.1 очевидно, что наибольшее значение ускорения было получено именно в режиме сверхоптимистичных вычислений (с автоматическим распределением ядер) на 6 ядрах. Более того, почти во всех случаях такой режим работы при одинаковом M позволял получать ускорение, большее, чем при использовании обычного параллельного счета (до 46% преимущества) или режима сверхоптимистичных вычислений без подстройки числа ядер (до 72% преимущества).

4.1.3.2. Пучок заряженных частиц в электростатической линзе

Моделировалась динамика пучка из 15000 заряженных частиц, попавших в двумерную собирающую электростатическую линзу, на сетке 40×40 узлов. Решение задачи сводится к интегрированию (во време-

ни) смешанной системы дифференциальных уравнений: уравнения Пуассона для потенциала электрического поля и уравнений перемещения заряженных частиц. К этим уравнениям присоединяются замыкающие соотношения для вычисления вектора напряженности электрического поля и для расчета плотности заряда в ячейке сетки. Уравнение Пуассона решается методом верхней релаксации, для прочих уравнений применяется явный метод Эйлера первого порядка.

Решение задачи представляет цикл численного интегрирования по времени, каждая итерация которого выполнялась по двухстадийной схеме (представленной в данной главе): на первой стадии пересчитывались координаты частиц и определялась плотность заряда (в ячейках расчетной сетки), которая через предсказывающий канал передавалась на вторую стадию, решающую уравнение Пуассона (плотность заряда входит в его правую часть) для потенциала электрического поля. Ввиду относительно небольшой размерности задачи и малокоррегируемого дисбаланса загрузки стадий, на процессоре Xeon (использовалась та же стандартная многоядерная (8 физических ядер процессора Xeon, 16 логических ядер) машина проекта Google's Compute Engine) удалось получить ускорение лишь на 2 ядрах, в режиме сверхоптимистичных вычислений величина ускорения составила около 1,07. В то же время, в дополнительных экспериментах на машине с четырехъядерным процессором Intel Atom удалось получить ускорение около 1,33. Для расчета ускорений были замерены времена решения задачи t(1) на одном ядре.

4.1.3.3. Моделирование распространения тепла в стержне

Решение данной задачи несколько отличается по структуре от предыдущих. Было записано уравнение теплопроводности для одномерной области из 36000 узлов (с граничными условиями Дирихле – слева и справа была задана температура, превышающая начальную в остальных точках стержня), которая была равномерно поделена на M частей, где M – число используемых ядер. Используется неявная разностная схема (первого порядка точности по времени и второго порядка по координате), поэтому возникает задача распараллеливания по пространству процесса скалярной прогонки, решающего возникающую систему линейных

уравнений на каждой итерации. Достаточно интересным представляется решение с применением сверхоптимистичных вычислений из 2·M стадий для каждой итерации цикла интегрирования по времени, где на первых M стадиях (первый транзакционный блок) выполняется прямой ход прогонки, а на оставшихся M стадиях (второй транзакционный блок) – обратный ход.

При прямом ходе, стадии в начале расчета (кроме первой стадии) принимают значения прогоночных коэффициентов (слева), а в конце расчета (кроме последней стадии) передают вычисленные значения прогоночных коэффициентов (справа) через предсказывающие каналы, попарно связывающие стадии. При обратном ходе, стадии в начале расчета (кроме последней стадии) принимают значения температуры (справа), а в конце расчета (кроме первой стадии) передают вычисленные значения температуры (слева) через предсказывающие каналы, попарно связывающие стадии.

Соответственно, если при прямом ходе обнаруживается, что k-я стадия получила из канала неточно предсказанные значения коэффициентов для своего крайнего левого узла, то пересогласовываются все j-е стадии-транзакции, $k \le j \le M$. Аналогично, если при обратном ходе обнаруживается, что p-я стадия неточно предсказала температуру в своем крайнем правом узле, то пересогласовываются все s-е стадии-транзакции, $1 \le s \le p$.

Здесь стратегия автоматического распределения ядер по стадиям не использовалась, на каждой стадии задействовалось одно ядро. Использовалась та же многоядерная вычислительная система, что и выше. Полученные результаты по ускорению и эффективности распараллеливания сведены в таблицу 4.2. Для вычисления ускорения дополнительно было определено время счета данной задачи t(1) в непараллельном режиме. Ускорение S(M) рассчитывалось тем же образом, что и в предыдущих экспериментах, эффективность распараллеливания E(M) определялась как

$$E(M) = S(M)/M.$$

Таблица 4.2

Полученные данные по численному интегрированию уравнения теплопроводности

Показатель	**Число задействованных ядер M**								
	16	14	12	10	8	6	4	2	1
Ускорение	2,89	3,23	3,5	3,79	4,52	4,18	3,34	1,9	1
Эффективность распараллеливания	0,18	0,23	0,29	0,38	0,57	0,7	0,84	0,95	1

Из данных таблицы 4.2 очевидно, что с помощью сверхоптимистичных вычислений удалось получить ускорение до 4,5 (при эффективности распараллеливания, равной 57%), при этом погрешность результатов была близка к нулевой, чего обычно не отмечается при использовании иных подходов к параллельному численному интегрированию неявными методами, предполагающих предварительный расчет значений на стыках блоков расчетной области [17] по дополнительным полунеявным разностным схемам (расчетная область одномерная, поэтому воспользоваться подходами с транспонированием, без дополнительной погрешности, как в случае многомерной области, невозможно).

4.2. Интерполирующая мемоизация как средство неявного ввода в язык нейросетевых и МГУА-интерполяторов и линейных предикторов

Как уже упоминалось в предыдущей главе, в языке Planning C возможно «навешивание» механизма интерполирующей мемоизации на процедуру или функцию. При этом набор входных и выходных параметров такого программного блока вполне может рассматриваться как набор входов и выходов нейронной сети, линейного предиктора или МГУА-интерполятора, обучаемых и вычисляемых неявно, при задействовании вышеупомянутого механизма. Для этого достаточно

а) организовать процедуру/функцию с интерполирующей мемоизацией, указав тип требуемого интерполятора/экстраполятора;

б) произвести несколько вызовов такой процедуры/функции для обучения рабочего элемента [при этом необходимо перед вызовом помещать во входные параметры значения входов (входов сети/полинома

или порядкового параметра линейной предикции), а значения выходов помещать в выходные параметры либо в процессе исполнения процедуры/функции (например, зачитывая их из некоторой таблицы и/или файла), либо также перед вызовом (при этом тело процедуры/функции может быть пустым, но тогда режим **controlled** использоваться не должен)];

в) осуществить необходимые вызовы процедуры/функции с префиксом «**predict_**», передавая в нее входные значения и получая на выходе значения, рассчитанные требуемым интерполятором/экстраполятором.

Рассмотрим пример. Обучим нейронную сеть прямого распространения (два слоя, в первом – пять нейронов с активационной функцией «экспоненциальная сигмоида», во втором – один линейный нейрон) распознавать на рецепторном поле размером 5×4 символы двух классов: крестик (1) и нолик (0). После этого обратимся к программе с предложением распознать «зашумленный» нолик. В соответствии с вышеизложенной стратегией напишем программу:


```
#include "memoization.h"
#include <iostream>
#include <stdlib.h>
#include <time.h>

using namespace std;

float class1 = 1;
int cross1[5][4] = {
   { 1, 0, 0, 1 },
   { 0, 1, 1, 0 },
   { 0, 1, 1, 0 },
   { 1, 0, 0, 1 },
   { 0, 0, 0, 0 }
};

float class2 = 1;
int cross2[5][4] = {
   { 0, 0, 0, 0 },
   { 1, 0, 0, 1 },
   { 0, 1, 1, 0 },
   { 0, 1, 1, 0 },
   { 1, 0, 0, 1 }
};
```

```
float class3 = 0;
int zero[5][4] = {
   { 0, 1, 1, 0 },
   { 1, 0, 0, 1 },
   { 1, 0, 0, 1 },
   { 1, 0, 0, 1 },
   { 0, 1, 1, 0 }
};

#pragma memoization(i, o) feed_forward(0.001, 0.025, 5000, 5, e)
void recog(int c[5][4], float & code) {
}

int test[5][4] = {
   { 0, 1, 1, 0 },
   { 1, 1, 0, 1 },
   { 1, 0, 1, 1 },
   { 1, 0, 0, 1 },
   { 0, 1, 1, 0 }
};

int main() {
      srand((unsigned int) time(NULL)); // Случайные числа
необходимы подсистеме интерполирующей мемоизации для
инициализации предиктора

      recog(cross1, class1);
      recog(cross2, class2);
      recog(zero, class3);

      float c;
      predict_recog(test, c);
      cout << c << endl;

      return 0;
}
```


Данная программа стабильно выводит на экран значения, не превышающие **0,5**, то есть, правильно классифицирует тестовый нолик.

4.3. Сокращенная алгебраическая нотация записи топологий

Базовые средства описания вычислительных топологий в различных языках/расширениях не всегда удобны и не очень наглядны. Это либо массивы (MPI), либо списки (Planning C), либо вызовы специаль-

ных функций (ТВВ). В сложных случаях данная проблема решается путем составления специального алгоритма, который генерирует описание топологии (в Planning C, в частности, используются дедуктивные макромодули). Однако в более простых случаях решать задачу алгоритмически не вполне целесообразно, а составление описания вручную небезопасно по причине возможного внесения трудноуловимых ошибок.

Актуальна задача разработки новых, интуитивно понятных средств описания топологий малой и средней сложности, которые можно было бы легко транслировать в классическое списковое или алгоритмическое описание.

4.3.1. Основные теоретические положения

Как уже упоминалось выше, в Planning C топология описывается набором прямых цепей и обратных дуг. При этом порядок следования цепей/дуг не имеет значения, но имеет значений порядок следования элементов в каждой отдельной цепи/дуге. Здесь можно провести *аналогию с алгебраическими операциями*, которые могут быть, соответственно, коммутативными и некоммутативными. Если обозначить узлы некоторыми идентификаторами, коммутативную операцию соединения цепей/дуг — знаком «**+**», а некоммутативную операцию соединения узлов в цепи/дуге — знаком «*****», то получаем некоторую *алгебраическую нотацию записи топологий*. Например, запись вида «**A*B*C+B*A**» будет означать наличие трех дуг, «**A->B**», «**B->C**» и «**B->A**».

Для ее упрощения можно, по аналогии с обычной алгеброй, ввести возможность *указания скобок* «(» и «)», разрешить действие сочетательных законов сложения/умножения и распределительного закона умножения относительно сложения. Далее, можно ввести *операцию возведения в константную степень «***»* (эквивалентно умножению основания степени на само себя указанное константой число раз) и *умножения на константу* (эквивалентно сложению первого множителя с самим собой заданное вторым множителем-константой число раз).

Также имеет смысл разрешить *вычитание* (некоммутативную операцию, обозначаемую знаком «**-**»), для которого будут справедливы все классические законы алгебры. Очевидно, что такая операция может ис-

пользоваться *лишь для упрощения алгебраической записи топологии*, поскольку дуга/цепь с обратным знаком *не имеет какого-либо алгоритмического смысла*. Например, описание «**(A+B)**2-A*B-B*A**» преобразуется в «**A*A+B*B**».

Далее, обратим внимание на тот факт, что появление нескольких узлов с одним идентификатором в описании топологии должно, вообще говоря, сопровождаться указанием *уточняющего индекса*. Введем следующие *правила*:

а) если за идентификатором следует индекс в квадратных скобках «[» и «]», то узел с соответствующим идентификатором имеет *указанный индекс*;

б) если за идентификатором следует указание «[]», то узел с соответствующим идентификатором в конечной записи получит *уникальный индекс, больший или равный единице*;

в) если за идентификатором нет никаких указаний на индекс, то узел с соответствующим идентификатором в конечной записи получит индекс, *равный единице*.

Например, описание «**(A[]+B)**2-A[]*B-B*A[]**» сначала преобразуется в «**A[]*A[]+B*B**», а затем, после расстановки индексов, превращается в «**A[1]*A[2]+B[1]*B[1]**».

Суммируя вышесказанное, дадим формальное описание синтаксиса предложенной нотации и дополнительно предложим перечень основных алгебраических законов, по которым возможна трансформация полученных выражений. Итак, *синтаксис описывается следующими правилами в расширенной форме Бэкуса-Наура*:

выражение = сумма

сумма = сумма «**+**» унарное | сумма «**-**» унарное | унарное

унарное = «**+**» с_константой | «**-**» с_константой | с_константой

с_константой = число «*» произведение | произведение

произведение = степенное «*» произведение | степенное

степенное = с_переменными «**» число | с_переменными

с_переменными = «(» выражение «)» | элемент

элемент = идентификатор «[]» | идентификатор «[» число «]» | идентификатор

Сформулируем *основные алгебраические законы* для трансформации описаний топологий в предложенной нотации, в которых выраже-

ния, не содержащие констант, обозначены прописными буквами, а числовые константы — строчными:

1. A+B = B+A.
2. (A+B)+C = A+(B+C) = A+B+C.
3. –(–A) = A.
4. A+(–B) = A–B.
5. +A = A.
6. A–A = 0.
7. A+0 = A.
8. (A*B)*C = A*(B*C) = A*B*C.
9. k*A = (A+A+…+A) {k раз, k > 0}.
10. (A+B)*C = A*C+B*C.
11. A*(B+C) = A*B+A*C.
12. A+B*C = A+(B*C).
13. (–A)*(–B) = A*B.
14. (–A)*B = –(A*B).
15. A*(–B) = –(A*B).
16. A**k = (A*A*…*A) {k раз, k > 0}.
17. A*B**k = A*(B**k).
18. A**k*B = (A**k)*B.
19. A+B**k = A+(B**k).
20. A–B**k = A–(B**k).

Отметим, что законы 12-й и с 17-го по 20-й введены исключительно для установления приоритетов операций.

4.3.2. Применение специализированного макромодуля для реализации нотаций

Поскольку синтаксические правила разбора могут быть достаточно хорошо описаны декларативно, наиболее простой и уместной реализацией транслятора алгебраических нотаций в стандартные для Planning C описания топологий представляется дедуктивный макромодуль (правила синтаксического разбора и все преобразования записываются в форме GNU Prolog-предикатов). Такой модуль принимает в качестве параметров, *во-первых*, список соответствий между идентификаторами

ППП, входящими в запись, и их списками параметров и, *во-вторых*, собственно алгебраическую запись топологии.

Макромодуль выполняет следующие *действия*:

а) лексический разбор алгебраической записи;

б) упрощение записи до аддитивной формы, с выполнением операций возведения в константную степень, умножения на константу, раскрытия всех скобок, вычитания и изменения знака;

в) проверку аддитивной формы на наличие отрицательных элементов, которые не удалось вычесть (в этом случае исходная запись признается некорректной);

г) присваивание уникальных индексов тем узлам-идентификаторам полученной записи, которые были указаны с постфиксом «[]»;

д) анализ полученной формы с преобразованием ее в список дуг топологии;

е) выделение циклов, в каждом цикле одна дуга объявляется реверсивной, прочие дуги считаются прямыми;

ж) генерация описания топологии (синтаксического блока, включенного в конструкцию **plan_topology {}**).

Соответствующий макромодуль **altopo** включен в язык в формате подключаемого файла **altopo.h**. Приведем в качестве примера фрагмент макромодуля, содержащий решающие предикаты **neg**, **add** и **sub**, выполняющие, соответственно, *упрощающие процедуры изменения знака, сложения и вычитания*. Предикаты принимают списки цепей, соединенных по умолчанию операцией «**+**». Каждая цепь, в свою очередь, является списком, включающим элементы (идентификаторы с индексом) и знаки операции (в частности, **opSub** соответствует унарной или бинарной операции «**–**», а **opAdd** — «**+**»).

```
@neg([[opSub|TI]|TL],[TI|TL1]):-
  neg(TL, TL1),
  !.
@neg([[opAdd|TI]|TL],[[opSub|TI]|TL1]):-
  neg(TL, TL1),
  !.
@neg([[HI|TI]|TL],[[opSub,HI|TI]|TL1]):-
  neg(TL, TL1),
  !.
@neg([],[]):-!.
```

```
// [... +a ...] + [... -a ...]
@add([[opAdd|TI]|TL], L2, L):-
  append(L1, [[opSub|TI]|TL3], L2),
  !,
  append(L1, TL3, L4),
  add(TL, L4, L),
  !.
// [... +a ...] + [...  ...]
@add([[opAdd|TI]|TL], L2, L):-
  add(TL, L2, L3),
  append([[opAdd|TI]], L3, L),
  !.
// [... -a ...] + [... +a ...]
@add([[opSub|TI]|TL], L2, L):-
  append(L1, [[opAdd|TI]|TL3], L2),
  !,
  append(L1, TL3, L4),
  add(TL, L4, L),
  !.
// [... -a ...] + [... -a ...]
@add([[opSub|TI]|TL], L2, [[opSub|TI],[opSub|TI]|L]):-
  append(L1, [[opSub|TI]|TL3], L2),
  !,
  append(L1, TL3, L4),
  add(TL, L4, L),
  !.
// [... -a ...] + [... a ...]
@add([[opSub|TI]|TL], L2, L):-
  append(L1, [TI|TL3], L2),
  !,
  append(L1, TL3, L4),
  add(TL, L4, L),
  !.
// [... -a ...] + [...  ...]
@add([[opSub|TI]|TL], L2, L):-
  add(TL, L2, L3),
  append([[opSub|TI]], L3, L),
  !.
// [... a ...] + [... -a ...]
@add([TI|TL], L2, L):-
  append(L1, [[opSub|TI]|TL3], L2),
  !,
  append(L1, TL3, L4),
  add(TL, L4, L),
  !.
```

```
// [... a ...] + [...  ...]
@add([TI|TL], L2, L):-
  add(TL, L2, L3),
  append([TI], L3, L),
  !.
@add([], L, L):-!.
@add(L, [], L):-!.
@sub(L1, L2, L):-
  neg(L2, L3),
  add(L1, L3, L),
  !.
```

Далее приведен *пример применения макромодуля*: работа с топологией «бинарное трехуровневое дерево». В примере число **22** передается от корня всем узлам дерева, далее листовые узлы выводят полученные ими числа на экран, после чего топология завершает работу. Из примера достаточно очевидно, что примененная алгебраическая форма записи топологии существенно короче и нагляднее простого спискового описания топологии базовыми средствами.


```
#include "altopo.h"
#include <iostream>
using namespace std;
chain A(bool init, input_proc Ref, int Result) {
  static int leafs = 0;
  input_proc incoming[128];
  int n_in;
  input_proc outcoming[128];
  int n_out;
  plan_neighbours(true, A[plan_linear_num()], &n_in, incom-
ing);
  plan_neighbours(false, A[plan_linear_num()], &n_out, out-
coming);
  if (init) {
     if (n_in == 0)
        plan_first(false, empty_proc, Result);
     if (n_out == 0) {
        plan_critical(outCount) {
          leafs++;
        }
     }
  } else if (n_out == 0) {
     plan_critical(outResult) {
       cout<<Result<<" ";
       leafs--;
```

```
      }
      if (leafs == 0) plan_topology_quit();
   } else {
      for (int i = 0; i < n_out; i++) {
          throw_first(false, outcoming[i], Result);
      }
   }
}
int main() {
   const int N = 22;
   altopo([['A', '(true, empty_proc, N)']],
'A[1]*(A[2]*(A[]+A[]) +A[3]*(A[]+A[]))');
   cout<<endl;
}
```

В качестве еще одного, заключительного примера алгебраической нотации приведем *запись топологии «труба»*: «**A*B*C*D+D*C*B*A**»

Выводы к четвертой главе

Предложена новая общая схема вспомогательного распараллеливания циклов с зависимыми витками и внутренними зависимостями в теле, которая может быть применена при реализации упомянутых в работе и иных математических алгоритмов, допускающих расчет с определенными погрешностями (в отличие от иных известных работ). Исследовано применение данной схемы с разбиением линейного фрагмента программы (тела цикла) более чем на две стадии.

Предложенная схема (сверхоптимистичные вычисления) основана на совместном применении предсказывающих каналов и частично транзакционной памяти (такое решение является новым). *Данная алгоритмическая схема впервые успешно применена для ускорения решения задачи обучения нейронной сети методом обратного распространения ошибки*.

Дополнительно, с помощью сверхоптимистичных вычислений удалось ускорить решение задач моделирования распространения тепла в тонком стержне и моделирования динамики заряженных частиц в электростатической линзе. Во всех случаях дополнительная погрешность составила не более 3,5%. При использовании в качестве основной, в задаче о распространении тепла схема распараллеливания показала

максимальное ускорение около 4,5 на 8 ядрах. При использовании в качестве дополнительной, схема распараллеливания показала увеличение ускорения на 7÷46% для различных задач. Дополнительно реализована схема автоматического выбора разделения ядер системы по блокам, обрабатывающим различные стадии цикла, позволившая существенно увеличить ускорение (до 1,7 раз) по сравнению с механическим разделением множества ядер на равные части.

Рассмотрен вопрос о прямом применении в программировании предикторов, генерируемых при использовании интерполирующей мемоизации. Показано, что такой прием может быть использован, например, для обучения и вычисления отклика классифицирующей нейронной сети. Приведен соответствующий пример.

Предложена новая, более удобная и наглядная по сравнению с иными средствами алгебраическая нотация записи вычислительных топологий небольшой и средней сложности. Описан синтаксис. Сформулированы основные алгебраические законы для трансформации и упрощения топологий. Для получения описания топологий (в языковом формате Planning C) по алгебраической записи предложено применить специальный дедуктивный макромодуль. Поскольку синтаксические правила и законы алгебраических трансформаций могут быть достаточно легко записаны в предикативной форме, такой подход является вполне естественным и эффективным.

Заключение

Подведем итоги:

1. Предложен новый язык классического и параллельного программирования Planning C (расширение языка C/C++), отличающийся от аналогов наличием ряда конструкций, существенно упрощающих программирование ряда алгоритмов на базе массивов, линейных списков, стека, дека и очереди за счет применения идеи процедур с планированием повторного входа. Реализация данной идеи позволила создать набор минимально необходимых средств алгоритмизации, доказаны их полнота и эквивалентность классическим основным средствам структурного программирования в современных алгоритмических языках. Разработан соответствующий демонстрационный транслятор языка (доступен по ссылке: http://www.pekunov.byethost31.com/Progs.htm#Reenterable), то есть, реализуемость описанных в данной работе идей доказана построением.

2. Предложены базовые конструкции распараллеливания в Planning C, охватывающие основные виды параллелизма зависимых и независимых подзадач на базе процедур с планированием повторного входа. В отличие от аналогов, данные конструкции позволяют четко и непротиворечиво описывать наиболее общие случаи масштабируемых виртуальных топологий, а также кратко и эффективно описывать некоторые широко употребимые частные случаи (вектор, конвейер, «портфель задач»), контролируя корректность передач данных. Применение многих базовых средств языка дает полную уверенность в отсутствии программных тупиков.

Базовые распараллеливающие конструкции для отдельных ПППВ и векторов/конвейеров ПППВ особенно просты. Необходимо лишь переписать алгоритм для ПППВ и, если он выполняет допускающие параллельное исполнение этапы плана, распараллелить (после более простой отладки в последовательном режиме — в ряде случаев результаты исполнения программы не зависят от наличия директив распараллеливания, которые в последовательном режиме просто игнорируются). Особенно легко и органично распараллеливается код для исполнения на векторных расширителях (GPU/CPU с векторными командами).

Введены наборы как архитектурно-независимых (упрощающих программирование для разных архитектур), так и специализированных (позволяющих получить более эффективный код) классических средств распараллеливания для систем с общей или разделяемой памятью при опциональном наличии векторных OpenCL-расширителей.

3. Предложен декларативный подход к описанию масштабируемых виртуальных топологий на базе специальных синтаксических конструкций — макромодулей, использующих конструирующие Prolog-элементы. В отличие от аналогов, данный подход позволяет четко и непротиворечиво описывать формальные правила построения топологий, что потенциально способно снизить количество ошибок их описания. Предложенные макромодули обладают важным для практики побочным применением — позволяют реализовать порождающее программирование, с помощью которого, например, можно создавать настраивающиеся на особенности конкретной решаемой задачи программы. Макромодули испытаны на ряде тестовых задач построения классических топологий, показана адекватность как идеи, так и ее реализации.

4. Предложен новый вид технологических средств предикции для использовании в параллельном программировании – предикционно-решающие каналы. Предложены два вида каналов – авторегрессионные точечные и линейные (явные или неявные) коллективного решения. Совмещение каналов со средствами предсказания данных упрощает программирование ряда алгоритмов, связанных с численным моделированием (прогнозирования времени счета для балансировки загрузки, расчета прогнозных значений переменных, например, на стыках блоков расчетной области при параллельной обработке, и других алгоритмов), и позволяет, в ряде случаев, осуществлять скрытые переходы от явных разностных схем к неявным, отличающимся большей устойчивостью счета, а также от последовательных алгоритмов счета к параллельным.

5. Введено понятие частично транзакционной памяти, сочетающей классические и транзакционные переменные, создан ряд классов транзакционных согласуемых переменных, определены особенности функционирования авторегрессионных точечных каналов в условиях согласования с откатами. Применение каналов в режиме частично транзакционной памяти позволяет реализовать достаточно интересный режим параллельной работы – сверхоптимистичные вычисления.

6. Предложены средства, поддерживающие мемоизацию процедур/функций. Предложена концепция интерполирующей мемоизации, позволяющей предсказать результат мемоизации даже в случае отсутствия текущего набора аргументов в кэше. При этом используется нейронная сеть прямого распространения, линейный предиктор или МГУА. Сформулированы алгоритмы, позволяющие контролировать точность предикции и гибко регулировать интервалы доверия ее результатам. Приведены примеры, демонстрирующие как классическую мемоизацию, так и мемоизацию с обучением единственного предиктора или группы предикторов, индексируемых по значению выбранного группирующего параметра. Показано, что для *некоторых задач из области вычислительной математики*, интерполирующая мемоизация с предиктором способна давать ускорение в 1,32÷8,8 раза. Построением доказано, что возможно применение предикторов интерполирующей мемоизации в качестве встроенных в язык обучаемых нейросетей, линейных экстраполяторов и МГУА-полиномов.

7. Сформулированы новые средства, позволяющие не только оперативно вводить в язык новые стандартные вычислительные топологии, но и строить иные произвольные расширения языка (новые конструкции, элементы аспектно-ориентированного и метапрограммирования) с применением групп регулярно-логических выражений (объединенных в сканирующие макросы – сканеры, индентифицирующие требуемые языковые фрагменты) и, связанных с ними, многократно согласуемых дедуктивных макромодулей. Приведен синтаксис новых конструкций, доказана их алгоритмическая реализуемость на языках высокого уровня. Даны примеры реализации простых языковых расширений, потенциально способных дополнительно адаптировать язык программирования к решаемой задаче или к запросам программиста. Сформулирован ряд идей по возможности интеллектуализации обработки программ с применением связок «сканеры-макромодули», в частности, инкрементальной самомодификации программ и верификации реализованных алгоритмов.

8. С помощью сканеров и макромодулей в язык введены элементы функционального программирования, а именно – анонимные ППП/ФППВ, вектора/конвейеры, цепи и топологии общего вида, возможность обращения к указанным элементам через ссылочные переменные. Предложены соответствующие синтаксические конструкции. Ана-

логичным образом в язык введен ряд специальных стандартных топологий.

9. Предложена новая общая схема вспомогательного распараллеливания циклов с зависимыми витками и внутренними зависимостями в теле, которая может быть применена при реализации математических алгоритмов, допускающих расчет с определенными погрешностями (в отличие от иных известных работ). Исследовано применение данной схемы с разбиением линейного фрагмента программы (тела цикла) более чем на две стадии. Предложенная схема (сверхоптимистичные вычисления) основана на совместном применении предсказывающих каналов и частично транзакционной памяти (такое решение является новым), опробована на задачах обучения нейронной сети методом обратного распространения ошибки, моделирования распространения тепла в тонком стержне, моделирования динамики заряженных частиц в электростатической линзе. Во всех случаях дополнительная погрешность составила не более 3,5%. При использовании в качестве основной, в задаче о распространении тепла схема распараллеливания показала максимальное ускорение около 4,5 на 8 ядрах. При использовании в качестве дополнительной, схема распараллеливания показала увеличение ускорения на 7÷46% для различных задач. Дополнительно реализована схема автоматического выбора разделения ядер системы по блокам, обрабатывающим различные стадии цикла, позволившая существенно увеличить ускорение (до 1,7 раз) по сравнению с механическим разделением множества ядер на равные части.

10. Предложена новая простая высокоуровневая алгебраическая нотация сокращенной записи виртуальных топологий, потенциально более понятная, логичная и компактная (используются многие основные принципы классической алгебры) в сравнении с классическими алгоритмическими средствами описания топологий, ограничивающимися перечислением дуг [46, 51]. Применение предложенной нотации потенциально снизит количество ошибок при описании виртуальных топологий малой и средней сложности. Показана адекватность нотации для описания ряда нестандартных топологий.

Библиографический список

1. Адинец, А.В., Воеводин, Вл.В. Графический вызов суперкомпьютерам // Открытые системы.— 2008.— Т. 5.— № 4.— С. 32–41.

2. Балаев, Э.Ф. Численные методы и параллельные вычисления для задач механики жидкости, газа и плазмы: учеб. пособие / Э.Ф. Балаев, Н.В. Нуждин, В.В. Пекунов [и др.]; Иван. гос. энерг. ун-т. — Иваново, 2003. — 336 с.

3. Бобровский, С.И. Технологии Delphi 2006. Новые возможности / С.И. Бобровский.— СПб.: Питер, 2006.— 288 с.

4. Борисов, Е.С. Полуавтоматическая система декомпозиции последовательных программ для параллельных вычислителей с распределенной памятью / Е.С. Борисов // Кибернетика и системный анализ.— 2004.— №3. — С.139—150.

5. Воеводин, В.В., Воеводин, Вл.В. Параллельные вычисления. — СПб.: БХВ-Петербург, 2002. — 608 с.

6. Гергель, В.П., Стронгин, Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных машин: Уч. пос. — Нижний Новгород: Изд-во ННГУ, 2000. — 176 с.

7. Годунов, С.К. Разностные схемы / С.К. Годунов, В.С. Рябенький. — М.: Наука, 1973.— 400 с.

8. Дюк, В., Самойленко, А. Data mining: учебный курс.— СПб: Питер, 2001. — 368 с.

9. Захарова, Е.М., Минашина, И.К. Обзор методов многомерной оптимизации // Информационные процессы. — 2014. — Т.14.— №3.— С.256-274.

10. Кнут, Д. Искусство программирования, том 1. Основные алгоритмы. — 3-е изд. — М.: «Вильямс», 2006. — 720 с.

11. Коновалов, Н.А., Крюков, В.А. DVM-система разработки параллельных программ // Высокопроизводительные вычисления и их приложения: Тр. Всеросс. науч. конф. — М.: Изд-во МГУ, 2000. — С.33.

12. Корнилина, М.А. Динамическая балансировка загрузки процессоров при моделировании задач горения / М.А. Корнилина, М.В. Якобовский // Высокопроизводительные вычисления и их приложения: тр. Всерос. науч. конф. / МГУ.— М., 2000. — С.34—39.

13. Одинцов, И.О., Черноиванов, Д.И. Методы реализации стандарта OpenMP в компиляторах // Высокопроизводительные вычисления и их приложения: Тр. Всеросс. науч. конф. — М.: Изд-во МГУ, 2000. — С.140-144.

14. Остерн, М.Г. Обобщенное программирование и STL: Использование и наращивание стандартной библиотеки шаблонов C++ / М.Г. Остерн. — СПб.: Невский Диалект, 2004. — 544 с.

15. Пекунов, В.В. Искусственные нейронные сети прямого распространения. Описание с помощью расширенных машин Тьюринга, вербализация и применение в аэродинамике. — LAP LAMBERT Academic Publishing, 2016.— 177 с.

16. Пекунов В.В. Нейронные сети в задачах предикции, возникающих в программировании. Применение в целях оптимального распараллеливания программ численного моделирования. - LAP LAMBERT Academic Publishing, 2019. - 100 с.

17. Пекунов, В.В. Новые методы параллельного моделирования распространения загрязнений в окрестности промышленных и муниципальных объектов // Дис. докт. тех. наук. — Иваново, 2009. — 274 с.

18. Пекунов В.В. Сверхоптимистичные вычисления: концепция и апробация в задаче о моделировании электростатической линзы // Программные системы и вычислительные методы. 2020. № 2. С. 37 - 44. DOI: 10.7256/2454-0714.2020.2.32232 URL: https://nbpublish.com/library_read_article.php?id=32232 (дата обращения: 31.08.2020).

19. Пекунов В.В. Система порождения, реконструкции и преобразования программ PGEN++. - LAP LAMBERT Academic Publishing, 2020. - 173 с.

20. Пекунов, В.В. Теория объектно-событийных моделей. Индукция, моделирование и синтез последовательных и параллельных программ.— LAP LAMBERT Academic Publishing, 2012.— 132 с.

21. Пекунов В.В. Теория расширенных машин Тьюринга. Объектно-транзакционные эквиваленты. - LAP LAMBERT Academic Publishing, 2021. - 84 с.

22. Пекунов, В.В. Численное моделирование распространения загрязнений. Оптимизация и автоматизация распараллеливания / В.В.Пекунов / ГОУВПО "Ивановский государственный энергетический

университет В.И.Ленина", ГОУВПО "Ивановская государственная текстильная академия". — Иваново, 2009. — 304 с.

23. Пекунов В.В. Язык программирования Planning C. Инструментальные средства. Новые подходы к обучению нейронных сетей. - LAP LAMBERT Academic Publishing, 2017. - 171 с.

24. Петров, А.В. MC# — универсальный язык параллельного программирования / А.В. Петров, В.Б. Гузев // Информационные технологии. — 2008. — №4. — С.29-32.

25. Рассел, С. Искусственный интеллект: современный подход / С. Рассел, П. Норвиг.— М.: Вильямс, 2007. — 1408 с.

26. Роуч, П. Вычислительная гидродинамика / П. Роуч.— М.: Мир, 1980. — 612 с.

27. Рофэйл, Эш, Шохауд, Я. COM и COM+: Полное руководство.— К.: ВЕК+, К.: НТИ, М.: Энтроп, 2000.— 560 с.

28. Сидоров, С.Г. Разработка ускоренных алгоритмов обучения нейронных сетей и их применение в задачах автоматизации проектирования: дис. ... канд. тех. наук.— Иваново, 2003.— 161 с.

29. Сузи, Р. А. Язык программирования Python: Учебное пособие. — М.: ИНТУИТ, БИНОМ. Лаборатория знаний, 2006. — 328 с.

30. Хокни Р., Иствуд Дж. Численное моделирование методом частиц. М.: Мир, 1987. 640 с.

31. Чернышева, Л.П. Проблема обработки стыков при моделировании процессов в сплошных средах с помощью многопроцессорных систем / Л.П. Чернышева, Ф.Н. Ясинский // Тез. докл. Междунар. науч.-техн. конф. «Состояние и перспективы развития электротехнологий» (X Бенардосовские чтения). — Иваново, 2001. — Т.2. — С.41.

32. Чернышева, Л.П. Сравнение алгоритмов распараллеливания при решении уравнений в частных производных / Л.П. Чернышева // Тез. докл. Междунар. науч.-техн. конф. «Состояние и перспективы развития электротехнологий» (XI Бенардосовские чтения). — Иваново, 2003. — Т.1. — С.87.

33. Черняк, Л. Транзакционная память - первые шаги / Л. Черняк // Открытые системы. СУБД. — 2007.— №4.— С.12-15.

34. Шпаковский, Г.И., Серикова, Н.В. Программирование для многопроцессорных систем в стандарте MPI. — Минск: БГУ, 2002. — 324 с.

35. Эйсымонт, Л.К. Оценочное тестирование высокопроизводительных систем: цели, методы, результаты и выводы / Л.К. Эйсымонт // Суперкомпьютерные вычислительно-информационные технологии в физических и химических исследованиях: сб. лекций. — Черноголовка, 2000. — С.33—42.

36. Эндрюс, Г.Р. Основы многопоточного, параллельного и распределенного программирования. — М.: Издательский дом «Вильямс», 2003. — 512 с.

37. Ясинский, Ф.Н. Математическое моделирование с помощью компьютерных сетей: учеб. пособие / Ф.Н. Ясинский, Л.П. Чернышева, В.В. Пекунов; ИГЭУ. — Иваново, 2000. — 201 с.

38. Ayguadé, E.: An Experimental Evaluation of the New OpenMP Tasking Model / E. Ayguadé, A. Duran, J. Hoeflinger, F. Massaioli, X. Teruel // Proceedings of LCPC (Languages and Compilers for Parallel Computing)'2007.— P.63—77.

39. B. Calder, G. Reinman, and D.M. Tullsen. Selective Value Prediction, in 26th International Symposium of Computer Architecture, May 1999.

40. El-Ghazawi, T., Carlson, W., Sterling, T., Yelick, K. UPC: distributed shared memory programming, — Wiley, 2005.— 232 pp.

41. F# Language Reference.— https://docs.microsoft.com/en-us/dotnet/articles/ fsharp/language-reference/

42. C. Fu, M.D. Jennings, S.Y. Larin, and T.M. Conte. Software-Only Value Speculation Scheduling, Technical Report, Dept. of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC27695-7911, June 1998

43. Gehani, N.H., Roome, W.D. The Concurrent C Programming Language.— Silicon Press, 1989.

44. Getting Started with Erlang User's Guide Version 8.2.— http://erlang.org/doc/getting_started/conc_prog.html

45. Hayato Yamaki, Hiroaki Nishi, Shinobu Miwa, and Hiroki Honda. Data prediction for response flows in packet processing cache. // In Proceedings of the 55th Annual Design Automation Conference (DAC '18). ACM, New York, NY, USA, 2018. - Article 110.- 6 pp. - DOI: 10.1145/3195970.3196021

46. Intel® Threading Building Blocks (Intel®TBB) Developer Guide. — https://software.intel.com/en-us/tbb-user-guide#tutorial_title

47. E. Larson and T. Austin. 2000. Compiler controlled value prediction using branch predictor based confidence. In Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture (MICRO 33). Association for Computing Machinery, New York, NY, USA, 327–336. DOI: https://doi.org/10.1145/360128.360164

48. Li, Xiao-Feng & Du, Zhao-Hui & Zhao, Qing-Yu & Ngai, Tin-Fook. Software Value Prediction for Speculative Parallel Threaded Computations. First Value Prediction Workshop, 18-25, 2003.

49. M.H. Lipasti and J.P. Shen. Exploiting Value Locality to Exceed the Dataflow Limit, in 29th International Symposium on Microarchitecture, December 1996

50. Marlow, S. Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming.— O'Reilly Media, 2013.— 322 pp.

51. MPI: A Message-Passing Interface Standard Version 3.1 — http://mpi-forum.org/docs/mpi-3.1/mpi31-report/mpi31-report.htm.

52. Munshi, A., Gaster, B.R., Mattson, T.G., Fung, J., Ginsburg, D. OpenCL Programming Guide.— Addison-Wesley, 2011.— 648 pp.

53. OCaml. – URL: https://ocaml.org

54. Paprzycki, M., Zalewski, J., Parallel Computing in Ada: An Overview and Critique // Ada Letters.— 1997.— Vol. XVII.— No. 2.— P.55-62.

55. C. J. Pickett, C. Verbrugge, and A. Kielstra. Adaptive software return value prediction. Technical Report 1, McGill University, 2009.

56. Y. Sazeides and J.E. Smith. The Predictability of Data Values, in 30th International Symposium on Microarchitecture, December 1997.

57. Arjun Suresh, Bharath Narasimha Swamy, Erven Rohou, André Seznec. Intercepting Functions for Memoization: A Case Study Using Transcendental Functions. ACM Transactions on Architecture and Code Optimization, Association for Computing Machinery, 2015, 12 (2), pp.23. ff10.1145/2751559ff. ffhal-01178085

58. Taft, S.T., Moore, B., Pinho, L.M., Michell, S. Safe Parallel Programming in Ada with Language Extensions // Technical Report CISTER-TR-141010.— 2014.

59. Teti, D. Delphi CookBook.— Packt Publishing.— 2014.— 329 pp.

60. The Go Programming Language Specification.— https://golang.org/ ref/spec#Go_statements

61. The OpenACC ® Application Programming Interface Version 2.5.— http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf

62. Bradley Thwaites, Gennady Pekhimenko, Hadi Esmaeilzadeh, Amir Yazdanbakhsh, Onur Mutlu, Jongse Park, Girish Mururu, and Todd Mowry. 2014. Rollback-free value prediction with approximate loads. In Proceedings of the 23rd international conference on Parallel architectures and compilation (PACT '14). Association for Computing Machinery, New York, NY, USA, 493–494. DOI:https://doi.org/10.1145/2628071.2628110

63. Zhao, B. Wu, M. Zhou, Y. Ding, J. Sun, X. Shen, and Y. Wu. Call sequence prediction through probabilistic calling automata. Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA), 2014, pp. 745–762.

Приложения

Приложение П1. Краткое описание транслятора Planning C

Возможны различные подходы к построению транслятора: от прямого написания лексико-синтаксического анализатора до применения специальных систем поддержки разработки компиляторов, таких как lex/yacc (flex/bison). В данной области нами не предлагается каких-либо существенных инноваций: по соображениям простоты реализации был выбран вариант с разработкой собственного лексико-синтаксического анализатора на языке Free Pascal, для которого существует бесплатный и свободно распространяемый многоплатформенный компилятор с соответствующими библиотеками. Такой выбор позволил создать транслятор, работоспособный как в Linux-, так и в Windows-среде. Программа на Planning C транслируется в C++-код, который также работоспособен в обеих вышеназванных средах.

Здесь будут кратко описаны лишь наиболее существенные и интересные аспекты реализации транслятора (доступен по ссылке: http://www.pekunov.byethost31.com/Progs.htm#Reenterable).

Выбор нижележащих средств реализации различных видов параллелизма

При выборе низкоуровневых средств реализации параллелизма необходимо придерживаться следующих основных критериев: возможности интегрирования в C/C++-программы, функциональной полноты, распространенности, кроссплатформенности, удобства применения. Исходя из первого критерия, рассмотрение сразу можно ограничить интерфейсами распараллеливания, представляющими собой либо библиотеки функций, либо наборы директив, транслируемые в параллельный код непосредственно компилятором C/C++ (предпочтительный вариант) или специализированным вспомогательным транслятором.

Учитывая вышеизложенное, в качестве *средства распараллеливания при работе в общей памяти* был выбран стандарт OpenMP, а в качестве *средства распараллеливания при работе в разделяемой памяти* — интерфейс MPI-2. Эти средства отвечают всем сформулированным требованиям.

При выборе *средства распараллеливания для векторных расширителей* выбор свелся, фактически, к двум основным стандартам: OpenCL и OpenAcc. Однако OpenAcc поддерживается далеко не всеми компиляторами (например, не поддерживается в Microsoft Visual C++), поэтому, в конечном итоге был выбран OpenCL.

Шаблонный подход к трансляции

Во многих случаях (там, где это было синтаксически возможно) вызовы специфических Planning C-функций не претерпевали каких-либо изменений (реализации соответствующих функций вставлялись транслятором в конечный код). Там же, где такая возможность отсутствовала (например, в связи с необходимостью включения в вызов дополнительных контекстных параметров), функция либо заменялась непосредственно фрагментом кода, либо создавался (по шаблону) макрос с именем, совпадающим с именем функции, который и производил всю требуемую работу (в том числе вызывал внутренние функции, неявно включаемые в код транслятором). В случае, если код макроса варьировался в зависимости от контекста вызова Planning C-функции, для его множественных реализаций использовалась пара директив препроцессора #undef/#define.

В первой версии Planning C специфические конструкции (директивы, ключевые слова, операторы) транслировались преимущественно с применением шаблонированных фрагментов кода, вставлявшихся в точку упоминания соответствующей конструкции, содержащих код, макросы и вызовы различных внутренних функций. Аналогичная технология была применена и в Planning C 2.0, с тем отличием, что большинство новых средств было реализовано в виде подключаемых библиотек (h-файлов) с применением введенных в язык новых конструкций «сканер - макромодуль», позволяющих создавать подключаемые расширения языка. Таким образом, собственно в транслятор была введена лишь допол-

нительная стадия препроцессинга, на которой обрабатывались сканирующие макросы в связке с дедуктивными модулями, а также поддерживалось многократное согласование таких модулей через общую базу фактов.

Вышесказанное позволяет говорить, в целом, о *шаблонном подходе*, при котором транслятор использовал значительное количество параметризованных фрагментов, вставлявшихся в конечный код по мере необходимости. Это позволило существенно упростить процесс разработки и отладки транслятора. Более подробно об основных алгоритмах и подходах к трансляции можно прочитать в работе [23].

Работа с транслятором через командную строку

Вызов транслятора имеет следующий формат:

Reenterable.exe [-nosourcelines] [-extensionsonly] [-followdefines] [-Dmacro[=val]] [<inputfile.cpp> [outputfile]]

Если параметры не указаны, то транслятор выводит краткую справку о своем использовании. Если указано только имя входного файл **<inputfile.cpp>**, то транслятор осуществляет преобразование и присваивает входному файлу расширение «**.bak**», а результат преобразования помещает в файл, имя которого совпадает с исходным **<inputfile.cpp>**. Если же указаны и имя входного файла **<inputfile.cpp>**, и имя выходного файла **outputfile**, то результат трансляции входного файла помещается в выходной файл с указанным именем.

Если указан ключ **-nosourcelines**, то в выходной файл не будут включены директивы **#line**, ссылающиеся на строки исходного файла **<inputfile.cpp>**. В результате позднее, при компиляции выходного файла, в случае возникновения ошибки, компилятор будет ссылаться на строки выходного, а не исходного файла. Это полезно, если из сообщений компилятора не удается установить источник проблемы и приходится анализировать результат трансляции.

Если указан ключ **-extensionsonly**, то в выходной файл не будет помещен служебный код транслятора Planning C (декларации констант и переменных, подключения необходимых библиотек, реализации внутренних функций и классов, в частности, классов каналов и многих дру-

гих). Такой режим может быть полезен, если разрабатывается не Planning C – программа, а C++-программа с расширениями, вводимыми в стиле Planning C. В такую программу включать служебный код Planning C, соответственно, не нужно.

Если указан ключ **-followdefines**, то транслятор будет интерпретировать все директивы препроцессора, связанные с условной компиляцией (**#define**, **#undef**, **#if**, **#ifdef** и т.п.), и генерировать соответствующий «фильтрованный» код. Если же данный ключ не указан, то данные директивы не обрабатываются и не фильтруются и транслятор обрабатывает весь текст программы.

Возможно определение дополнительных символов **macro** с применением ключа **-D**. Это полезно, если код входной программы содержит директивы условной компиляции в зависимости от наличия или значений стандартных для классического компилятора C++ символов (_WIN32, _OPENMP и т.п.), априорно не определяемых транслятором Planning C.

Компиляция транслированной программы

Выполняется стандартным компилятором C++. При этом необходимо позаботиться о включении режима работы с OpenMP и (если соответствующие возможности используются) о подключении библиотек MPI-2 и OpenCL.

Кроме того, при компиляции в Linux, необходимо включить режимы поддержки транзакционной памяти и работы в стандарте **c++0x** и подключить библиотеки математики и Pthreads. Вызов компилятора C++ в Linux может выглядеть так:

```
g++ -o <выходной_файл> <входной_файл.cpp> -fopenmp -fgnu-tm -fpermissive
                    -std=c++0x -lm -lpthread –lOpenCL
```

Если исходная программа использует кластерные средства поддержки параллельной работы, то вместо **g++**, вероятно, будет использоваться **mpicxx**.

Для вызова стандартного компилятора MSVC++ в среде Windows может использоваться bat-файл вида:

```
@echo off
```

```
@set PATH="C:\Program Files\Microsoft Visual Studio
12.0\VC\bin";"C:\Program Files\Microsoft Visual Studio
12.0\Common7\IDE\";%PATH%
@set IPATH="C:\Program Files\Microsoft Visual Studio 12.0\VC\include"
@set IPATHSDK="C:\Program Files\Microsoft SDKs\Windows\v7.1A\Include"
@set IPATHMPI="D:\Program Files\MPICH2\include"
@set IPATHOCL="C:\Program Files\AMD APP SDK\2.9-1\include"
@set LPATH="C:\Program Files\Microsoft Visual Studio 12.0\VC\lib"
@set LPATHSDK="C:\Program Files\Microsoft SDKs\Windows\v7.1A\Lib"
@set LPATHMPI="D:\Program Files\MPICH2\lib"
@set LPATHOCL="C:\Program Files\AMD APP SDK\2.9-1\lib\x86"
@cl.exe /O2 -o %2 %1 mpi.lib OpenCL.lib wsock32.lib /Fa /MTd /EHsc /openmp
/I%IPATH% /I%IPATHSDK% /I%IPATHMPI% /I%IPATHOCL% /link
/LIBPATH:%LPATH% /LIBPATH:%LPATHMPI%
/LIBPATH:%LPATHSDK% /LIBPATH:%LPATHOCL% >_.err
```

Такой файл **start_c.bat** может быть вызван в формате:

start_c.bat <входной_файл_cpp> <выходной_файл_exe>

При этом сообщения компилятора попадут в файл **_.err**.

Запуск скомпилированной программы

Скомпилированная программа может потребовать наличия в системе установленного интерпретатора GNU Prolog (если в программе используются макромодули и, возможно, сканеры), а также среды исполнения MPI и подсистемы поддержки OpenCL. Разумеется, кроме этого необходимо наличие всех динамических библиотек C++, которые используются программой.

Также заметим, что если программа использует кластерные топологии и/или конструкты, то ее запуск производится через стандартную программу **mpirun**. В прочих случаях просто непосредственно запускается скомпилированный файл программы.

Приложение П2. Программа на на линейных каналах коллективного решения, осуществляющая переход от явной схемы к аппроксимированной неявной с параметризованным шагом по времени

```
#include <ostream>

using namespace std;

#include <omp.h>

 /* LU - разложение  с выбором максимального элемента по
диагонали */
 bool GetLU(int NN, int * iRow, long double * A, long double *
LU)
 {
  int i,j,k;
  try {
    memmove(LU,A,NN*NN*sizeof(long double));
    for (i=0;i<NN;i++)
        iRow[i]=i;
    for (i=0;i<NN-1;i++)
      {
       long double Big  = fabs(LU[iRow[i]*NN+i]);
       int iBig = i;
       long double Kf;
       for (j=i+1;j<NN;j++)
           {
            long double size = fabs(LU[iRow[j]*NN+i]);
            if (size>Big)
               {
                Big  = size;
                iBig = j;
               }
           }
       if (iBig!=i)
          {
           int V = iRow[i];
           iRow[i] = iRow[iBig];
           iRow[iBig] = V;
          }
       Kf = 1.0/LU[iRow[i]*NN+i];

       LU[iRow[i]*NN+i] = Kf;
       for (j=i+1;j<NN;j++)
           {
```

```
            long double Fact = Kf*LU[iRow[j]*NN+i];
            LU[iRow[j]*NN+i] = Fact;
            for (k=i+1;k<NN;k++)
                LU[iRow[j]*NN+k] -= Fact*LU[iRow[i]*NN+k];
           }
      }
    LU[(iRow[NN-1]+1)*NN-1] = 1.0/LU[(iRow[NN-1]+1)*NN-1];
  }
  catch (...) {
    return false;
  }
  return true;
 }
 /* Метод LU - разложения */
 bool SolveLU(int NN, int * iRow, long double * LU, long double
* Y, long double * X)
 {
  int i,j,k;
  try {
    X[0] = Y[iRow[0]];
    for (i=1;i<NN;i++)
        {
         long double V  = Y[iRow[i]];
         for (j=0;j<i;j++)
             V -= LU[iRow[i]*NN+j]*X[j];
         X[i] = V;
        }
    X[NN-1]*=LU[(iRow[NN-1]+1)*NN-1];
    for (i=1,j=NN-2;i<NN;i++,j--)
        {
         long double V = X[j];
         for (k=j+1;k<NN;k++)
             V -= LU[iRow[j]*NN+k]*X[k];
         X[j] = V*LU[iRow[j]*NN+j];
        }
  }
  catch (...) {
    return false;
  }
  return true;
 }
 /* Метод Гаусса-Зейделя  с выбором максимального элемента по
диагонали */
 bool SolveGaussZeidel(int NN, int * iRow, long double * A, long
double * LU, long double * Y, long double * X) {
  int i, j;
  double prev_eps, eps;
  int grow;
  int iters;
```

```
try {
  for (i=0;i<NN;i++)
      iRow[i]=i;
  for (i=0;i<NN-1;i++)
    {
     long double Big  = fabs(A[iRow[i]*NN+i]);
     int iBig = i;
     for (j=i+1;j<NN;j++)
         {
          long double size = fabs(A[iRow[j]*NN+i]);
          if (size>Big)
             {
              Big  = size;
              iBig = j;
             }
         }
     if (iBig!=i)
        {
         int V = iRow[i];
         iRow[i] = iRow[iBig];
         iRow[iBig] = V;
        }
    }
  for (i=0;i<NN;i++)
      X[i]=0.5;
  eps = 1E300;
  grow = 0;
  for (iters = 0; eps > 1E-8 && iters < NN; iters++) {
      prev_eps = eps;
      eps = 0.0;
      for (i=0;i<NN;i++) {
          long double V = Y[iRow[i]];
          double d = X[i];
          for (j=0;j<NN;j++)
              if (j != i)
                 V -= A[iRow[i]*NN+j]*X[j];
          X[i] = V/A[iRow[i]*NN+i];
          d -= X[i];
          eps += d*d;
      }
      eps = sqrt(eps);
      if (eps > prev_eps)
         grow++;
      else
         grow = 0;
  }
  return eps <= 1E-8;
}
catch (...) {
```

```
    return false;
  }
 }

const bool EXPLICIT_PERCEPTRON = false;

const int N = 200;

const double tau1 = 0.0025;
const double tau2 = 0.001;
const double tau3 = 0.005;

int main() {
      double seq[N] = { 1.0, 0.0 };
      double seq1[N] = { 0 };

         int * iRow = new int[N];
         long double * A = new long double[N*N];
      long double * LU = new long double[N*N];
         long double * Y = new long double[N];
         long double * X = new long double[N];
      double * seqn = new double[N];
      double * b = new double[N];
           double * seqp = new double[N];
           double * bb = new double[N];
         double * seqpp = new double[N];

      seq[N-1] = 2.0;

         for (int j = 0; j < N; j++)
             seqpp[j] = seq[j];

            vector<_funnel<double> *> r;

      funneled_perceptron_out<double> * outbb = new
funneled_perceptron_out<double>(true, false, r, "bbPFUNNEL", N,
1E-5, 3);
      funneled_perceptron_out<double> * outpi = new
funneled_perceptron_out<double>(EXPLICIT_PERCEPTRON, false, r,
"iPFUNNEL", N, 1E-5, 3);

            funneled_perceptron_in<double> * inbb = new
funneled_perceptron_in<double>(true, false, r, "bbPFUNNEL", N,
1E-5, 3);
            r.push_back(inbb->getRef());
            funneled_perceptron_in<double> * inpi = new
funneled_perceptron_in<double>(EXPLICIT_PERCEPTRON, false, r,
"iPFUNNEL", N, 1E-5, 3);
            inpi->getRef()->switchLU(false);
```

```
      while (!outpi->ready());
            while (!inpi->ready());

            if (EXPLICIT_PERCEPTRON)
               cout << "EXPLICIT";
            else
               cout << "IMPLICIT";
            cout<<" (FORECAST)"<<endl;

            double tlearn = 0.0;
      for (int i = 0; i < N + 50; i++) {
          for (int j = 0; j < N; j++)
              b[j] = sin(0.25*(i+j)/(N-1));
          outbb->put((void *) b, N*sizeof(double));
             seqn[0] = seq[0];
             seqn[N-1] = seq[N-1];
             for (int j = 1; j < N-1; j++)
                 seqn[j] = seq[j] + tau1*(seq[j-1] - 2.0*seq[j] +
seq[j+1] - b[j]);
             for (int j = 1; j < N-1; j++)
                 seq[j] = seqn[j];
          outpi->put((void *) seqn, N*sizeof(double));

                double t = omp_get_wtime();
                inbb->get((void *) bb, N*sizeof(double));
                if (inpi->get_timed((void *) seqp, -1.0)) {
                   double buf[N];
                   inpi->get_and_correct((void *) buf);
             }
             tlearn += omp_get_wtime() - t;
      }

      r.clear();
      funneled_perceptron_out<double> * goutbb = new
funneled_perceptron_out<double>(true, false, r, "gbbPFUNNEL", N,
1E-5, 3);
      funneled_perceptron_out<double> * goutpi = new
funneled_perceptron_out<double>(EXPLICIT_PERCEPTRON, false, r,
"giPFUNNEL", N, 1E-5, 3);

            funneled_perceptron_in<double> * ginbb = new
funneled_perceptron_in<double>(true, false, r, "gbbPFUNNEL", N,
1E-5, 3);
            r.push_back(ginbb->getRef());
            funneled_perceptron_in<double> * ginpi = new
funneled_perceptron_in<double>(EXPLICIT_PERCEPTRON, false, r,
"giPFUNNEL", N, 1E-5, 3);
            ginpi->getRef()->switchLU(false);
```

```
      while (!goutpi->ready());
           while (!ginpi->ready());

         for (int j = 0; j < N; j++) {
             double v = seqpp[j];
             seqpp[j] = seq[j];
             seq[j] = v;
         }
      for (int i = 0; i < N + 50; i++) {
          for (int j = 0; j < N; j++)
              b[j] = sin(0.25*(i+j)/(N-1));
          goutbb->put((void *) b, N*sizeof(double));
             seqn[0] = seq[0];
             seqn[N-1] = seq[N-1];
             for (int j = 1; j < N-1; j++)
                 seqn[j] = seq[j] + tau2*(seq[j-1] - 2.0*seq[j] +
seq[j+1] - b[j]);
             for (int j = 1; j < N-1; j++)
                 seq[j] = seqn[j];
          goutpi->put((void *) seqn, N*sizeof(double));

                double t = omp_get_wtime();
                ginbb->get((void *) bb, N*sizeof(double));
                if (ginpi->get_timed((void *) seqp, -1.0)) {
                   double buf[N];
                   ginpi->get_and_correct((void *) buf);
             }
             tlearn += omp_get_wtime() - t;
      }

      reinterpret_cast< _funnel_perceptron<double> *>(inpi-
>getRef())->parametrize(tau1, ginpi->getRef(), tau2);
      reinterpret_cast< _funnel_perceptron<double> *>(inpi-
>getRef())->use_parameter(tau3);

         for (int j = 0; j < N; j++) {
             seq[j] = seqpp[j];
         }
      double tpredict = omp_get_wtime();
      for (int i = N + 50; i < N+300; i++) {
          for (int j = 0; j < N; j++)
              b[j] = sin(0.25*(i+j)/(N-1));
          outbb->put((void *) b, N*sizeof(double));
          outpi->put((void *) seqn, N*sizeof(double));

                inbb->get((void *) bb, N*sizeof(double));
                if (inpi->get_timed((void *) seqpp, -1.0)) {
                   inpi->cancel_and_push((void *) seqpp);
```

```
            } else
                cout << "NO PREDICTION!" << endl;
        }
        tpredict = omp_get_wtime() - tpredict;

        cout << "PREDICTED: ";
        for (int j = 0; j < N; j++)
            cout << seqpp[j] << " ";
        cout << endl;

           memset(A, 0, N*N*sizeof(long double));
           A[0*N+0] = 1.0;
           for (int j = 1; j < N-1; j++) {
               A[j*N+j] = 1.0 + 2.0*tau3;
               A[j*N+j-1] = -tau3;
               A[j*N+j+1] = -tau3;
           }
           A[(N-1)*N+N-1] = 1.0;

        double tcalc = omp_get_wtime();
        for (int i = N + 50; i < N+300; i++) {
            double seqn[N];
            double b[N];
            for (int j = 0; j < N; j++)
                b[j] = sin(0.25*(i+j)/(N-1));
            if (EXPLICIT_PERCEPTRON) {
                  seqn[0] = seq[0];
                  seqn[N-1] = seq[N-1];
                  for (int j = 1; j < N-1; j++)
                      seqn[j] = seq[j] + tau3*(seq[j-1] -
2.0*seq[j] + seq[j+1] - b[j]);
                  for (int j = 1; j < N-1; j++)
                      seq[j] = seqn[j];
              } else {
                  Y[0] = seq[0];
                  for (int j = 1; j < N-1; j++) {
                      Y[j] = seq[j] - tau3*b[j];
                  }
                  Y[N-1] = seq[N-1];
                  SolveGaussZeidel(N, iRow, (long double *) A,
(long double *) LU, Y, X) ||
                    GetLU(N, iRow, (long double *)A, (long double
*)LU) &&
                    SolveLU(N, iRow, (long double *)LU, Y, X);
                  for (int j = 0; j < N; j++)
                      seq[j] = X[j];
              }
        }
        tcalc = omp_get_wtime() - tcalc;
```

```
      cout << "CALCULATED: ";
      for (int j = 0; j < N; j++)
                cout << seq[j] << " ";
           cout << endl;

           cout << "Learning = " << tlearn << " sec." << endl;
           cout << "Predicting = " << tpredict << " sec." <<
endl;
           cout << "Equivalent calculating = " << tcalc << "
sec." << endl;

      delete outpi;
      delete outbb;

           delete inpi;
           delete inbb;

        delete[] iRow;
        delete[] A;
        delete[] LU;
        delete[] Y;
        delete[] X;
        delete[] seqn;
        delete[] b;
        delete[] seqp;
        delete[] bb;
        delete[] seqpp;

      return 0;
}
```