МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное учреждение высшего профессионального образования «Южный федеральный университет»

Факультет математики, механики и компьютерных наук
Кафедра алгебры и дискретной математики

Малеванный Михаил Сергеевич

РЕАЛИЗАЦИЯ ДИРЕКТИВ OPENMP ДЛЯ ЯЗЫКА PASCALABC.NET

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ
по направлению 010400 – Информационные технологии

Научный руководитель –
доц. к.ф.-м.н. Михалкович Станислав Станиславович

Рецензент –
доц. к.ф.-м.н. Савельев Василий Александрович

Ростов-на-Дону – 2011

Оглавление

Введение

Система программирования PascalABC.NET — это реализация языка Object Pascal для платформы Microsoft.NET, сочетающая простоту языка Паскаль и огромные возможности платформы .NET. Система активно развивается и используется на факультете Математики Механики и Компьютерных Наук Южного федерального университета при обучении студентов основам программирования, а также для обучения школьников. В ряде учебных заведений PascalABC.NET является одной из сред для преподавания современного программирования [1].

OpenMP — открытый стандарт для распараллеливания программ. Описывает совокупность директив компилятора, библиотечных процедур и переменных окружения, предназначенных для программирования многопоточных приложений на многопроцессорных системах с общей памятью [2].

OpenMP хорошо подходит для распараллеливания программ на поздних этапах разработки, когда последовательный код уже готов. Для выделения в коде участков, пригодных для параллельного выполнения, выполняемых параллельно, используются директивы компилятора. Внесение других изменений в код, кроме вставки директив, как правило, не требуется. Если компилятор не поддерживает OpenMP, он игнорирует директивы и генерирует последовательный код. Этим обеспечивается совместимость распараллеленного с использованием OpenMP кода с компиляторами, не поддерживающими OpenMP, например, со старыми версиями PascalABC.NET.

Рассмотрим внутреннее представление компилятора PascalABC.NET. Основные структуры внутреннего представления компилятора — синтаксическое и семантическое деревья.

Синтаксическое дерево описывает синтаксически правильную программу без учета семантики. Дерево строится из набора классов, описы-

вающих различные элементы программы. Обработка дерева происходит при его обходе с помощью посетителя. Посетитель (visitor) — один из паттернов проектирования, позволяющий отделить данные от алгоритмов, их обрабатывающих (подробнее см. [3, стр. 314–327]).

Семантическое дерево содержит исчерпывающую информацию о правильной программе и не зависит от исходного языка. Оно также состоит из набора классов и для работы с деревом также используется паттерн «Посетитель».

Основные причины разделения на синтаксическое и семантическое деревья – принципиально разная структура этих деревьев, а также независимость конвертора синтаксического дерева в семантическое от используемого парсера [4].

Компиляция происходит в несколько этапов:

1. Этап синтаксического анализа. На основе исходного текста парсер строит синтаксическое дерево.
2. Этап семантического анализа. На основе синтаксического дерева конвертер строит семантическое дерево. Построенное семантическое дерево может быть сохранено на диск. Вместо первых двух этапов может выполняться чтение ранее сохраненного семантического дерева с диска.
3. Генерация IL-кода. Семантическое дерево переводится в IL-код для платформы .NET.

Преобразования внутреннего представления, связанные с распараллеливанием производятся на этапе семантического анализа. Синтаксическое дерево, описывающее код с директивами преобразуется в семантическое дерево, описывающее параллельный код. При этом, помимо синтаксического дерева используется также и семантическая информация, доступная на этот момент.

PascalABC.NET разработан на языке C#, работает на платформе .NET и генерирует исполнимые файлы под эту платформу. Это означает,

что для работы PascalABC.NET на машине должна быть установлена платформа .NET (на момент написания этой работы — версии 2.0 и выше). Платформа .NET представляет собой виртуальную машину, исполняющую предназначенный для неё промежуточный код.

Распараллеливание выполняется с использованием библиотеки TPL (Task Parallel Library) [5], входящей в состав платформы .NET 4. Для платформы .NET 3.5 существует расширение Reactive Extensions, содержащее эту библиотеку, многие возможности которого работают с использованием возможностей только платформы .NET 2.0. При выполнении работы были использованы только такие возможности, таким образом, требования к версии платформы .NET для работы как компилятор PascalABC.NET, так и создаваемых им программ не изменились.

Для работы распараллеленных программ требуется сборка System.Threading.dll из расширения Reactive Extensions. Она устанавливается в системе при установке PascalABC.NET и не требует установки расширения Reactive Extensions. Также эта сборка может быть помещена в директорию с программой, если программа была перенесена на компьютер, на котором не установлены ни PascalABC.NET, ни расширение Reactive Extensions.

Стандарт OpenMP предназначен для языков C, C++ и Fortran. В настоящее время OpenMP поддерживается такими компиляторами как:

- GCC, языки C, C++, Fortran.
- Intel Compilers, языки C, C++, Fortran.
- Microsoft Visual C++, язык C++.
- IBM XL, языки C, C++, Fortran.

И другими, полный список доступен на сайте [6].

Диссертация состоит из введения, постановки задачи, шести глав, заключения и библиографического списка.

В первой главе описаны основные элементы стандарта OpenMP, приведен синтаксис и описание директив, функций и переменной окруже-

ния, реализованных в ходе данной работы. Вторая глава посвящена краткому описанию внутреннего представления компилятора PascalABC.NET, на уровне которого производятся распараллеливающие преобразования. В третьей главе описана библиотека TPL, используемая в ходе работы. В четвертой главе описан разбор и обработка директив OpenMP, в соответствии с которыми выполняется распараллеливание программ. Пятая глава посвящена описанию распараллеливающих преобразований программ во внутреннем представлении, реализованных в ходе данной работы. Оценке эффективности распараллеливания посвящена шестая глава.

Постановка задачи

Реализовать в системе программирования PascalABC.NET поддержку следующих элементов стандарта OpenMP:

- Директива parallel for
- Директива parallel sections
- Директива critical
- Опция private для директив parallel for и parallel sections
- Опция reduction для директивы parallel for

1. Директивы OpenMP должны записываться как директивы компилятора PascalABC.NET.
2. Компилятор PascalABC.NET должен распараллеливать компилируемые программы в соответствии с вышеуказанными директивами и их опциями.

Глава 1. Элементы стандарта OpenMP

В этой главе приведены реализованные в ходе работы элементы стандарта с учетом синтаксиса языка и их описание.

Директивы

Распараллеливание в OpenMP выполняется явно при помощи вставки в исходный код программы специальных директив, а также вызова вспомогательных функций и работы с переменными окружения. При использовании OpenMP предполагается SPMD-модель (Single Program Multiple Data) параллельного программирования, в рамках которой для всех параллельных нитей используется один и тот же код.

Директивы OpenMP в программах на языке Pascal оформляются в виде директив компилятора.

Формат директивы:

```
{$omp directive-name [clause[[,] clause]...]}
```

Ключевое слово omp используется для того, чтобы исключить случайные совпадения имён директив OpenMP с другими именами.

Объектом действия большинства директив является один оператор (простой или составной), перед которым расположена директива в исходном тексте программы. В OpenMP такие операторы или блоки называются ассоциированными с директивой. Ассоциированный блок должен иметь одну точку входа в начале и одну точку выхода в конце. Порядок опций в описании директивы несущественен, в одной директиве большинство опций может встречаться несколько раз. После некоторых опций может следовать список переменных, разделяемых запятыми.

Программа начинается с последовательной области — сначала работает один поток (Thread), при входе в параллельную область порождается ещё некоторое число потоков, которые, в зависимости от директив, исполняют один и тот же код или разные участки кода. По завершении парал-

лельной области все нити, кроме одной — главной, завершаются, и начинается последовательная область. В программе может быть любое количество параллельных и последовательных областей.

Наиболее важные директивы OpenMP:

- директива объявления параллельной области:

```
{$omp parallel}
```

- директивы распределения нагрузки:

```
{$omp for}
{$omp sections}
```

Допускается также сокращенная запись директив:

```
{$omp parallel for}
{$omp parallel sections}
```

При такой записи будет объявлена параллельная область, ассоциированная со следующим за директивой оператором, и участки кода будут распределены между потоками этой области.

В ходе работы была реализована сокращенная запись директив параллельных циклов и параллельных секций, а также директива критических секций. Полностью стандарт описан в спецификации [2].

Директива parallel for

Синтаксис:

```
{$omp parallel for [опция[[,] опция]...]}
for ... to ... do
  ...
```

Эта директива относится к идущему следом за данной директивой циклу for. При этом цикл образует параллельную область, в которой порождается несколько потоков, а итерации цикла распределяются между этими потоками.

В качестве опций директивы могут выступать *private* и *reduction*.

Опция private

Эта опция служит для изменения вида переменной. Существует два вида переменных – частные и разделяемые. По-умолчанию, все переменные, объявленные вне параллельного, цикла являются разделяемыми, а переменные, объявленные внутри цикла — частными. С помощью этой опции переменную, объявленную вне цикла можно сделать частной, таким образом, каждый поток будет работать со своей копией этой переменной. Присваивание этой переменной в одном потоке не будет влиять на значение переменной в других потоках.

Синтаксис:

```
{$omp parallel for private(список переменных)}
```

Список переменных — переменная или несколько переменных через запятую.

Опция reduction

Эта опция позволяет определить переменную и оператор редукции.

Достаточно распространенной является ситуация, когда в цикле значение переменной накапливается в виде суммы, произведения или других операций, применяемых к значениям, получаемым на разных итерациях.

Напрямую распараллелить такой цикл нельзя из-за того, что на разных итерациях запись в переменную может выполняться одновременно, и возможна потеря изменений. Решить эту проблему позволяет редукция.

Синтаксис:

```
{$omp parallel for reduction(оператор : список переменных)}
```

Оператор – один из допустимых операторов редукции.

Список переменных — одна или несколько переменных через запятую.

При этом все переменные редукции будут объявлены частными, в начале цикла они будут инициализированы значением, зависящим от опе-

ратора редукции (см. таблицу 1), а результирующее значение будет получено путем применения оператора редукции к исходному значению и значениям, полученным каждым потоком. Таким образом, обеспечивается корректное вычисление значения переменной в параллельном цикле.

Оператор редукции	**Инициализированное значение**
+	0
*	1
–	0
and (побитовый)	~0 (каждый бит установлен)
or (побитовый)	0
xor (побитовый**)**	0
and (логический)	True
or (логический)	False

Таблица 1. Список операторов редукции и значений переменных.

Ограничения при распараллеливании циклов

При распараллеливании циклов существует ряд ограничений, налагаемых на цикл.

Итерации цикла должны быть независимы друг от друга. Не должно быть двух обращений к одному и тому же объекту (переменной, элементу массива, классу и т.д.) на разных итерациях распараллеливаемого цикла, как минимум одно из которых изменяет состояние этого объекта. Если такие обращения есть, то существует *циклически порожденная зависимость,* которая препятствует корректному распараллеливанию, так как при распараллеливании нарушается исходный порядок выполнения итераций. При наличии директивы цикл будет распараллелен, но результат выполнения такого цикла может быть непредсказуем. Стандарт не предусматривает никаких способов проверки наличия зависимостей, препятствующих распараллеливанию. Об этом должен позаботиться программист.

В некоторых случаях использование опций `private` и `reduction` позволяет избежать зависимостей.

Нельзя использовать побочный выход из параллельного цикла (оператор `break`). Если в исходном цикле есть побочный выход, такой цикл не распараллелится, и будет выдана ошибка компиляции.

Директива parallel sections

Директива **parallel sections** используется для задания конечного (неитеративного) параллелизма.

Синтаксис:

```
{$omp parallel sections [опция[[,] опция] ...] }new-line
begin
  structured-block
 [structured-block]
  ...
end;
```

Эта директива определяет набор независимых секций кода, каждая из которых выполняется своим потоком. Все секции начинают работу одновременно. Структурированный блок (`structured-block`) – это либо отдельный оператор, либо блок операторов, заключенный в операторные скобки `begin..end`.

Параллельные секции, так же как и итерации параллельного цикла, должны быть независимы друг от друга, чтобы распараллеливание было корректным.

Директива critical

С помощью директивы `omp critical` оформляется критическая секция программы.

Синтаксис:

```
{$ omp critical [(<имя_критической_секции>)]}
structured-block
```

В каждый момент времени в критической секции может находиться не более одного потока. Если критическая секция уже выполняется каким-либо потоком, то все другие потоки, пытающиеся выполнить оператор, ассоциированный с директивой критической секции с данным именем, будут заблокированы, пока вошедший поток не закончит выполнение данной критической секции. Как только работавший поток выйдет из критической секции, один из заблокированных на входе потоков войдет в неё. Если на входе в критическую секцию стояло несколько потоков, то случайным образом выбирается один из них, а остальные заблокированные потоки продолжат ожидание.

Все неименованные критические секции условно ассоциируются с одним и тем же именем. Все критические секции, имеющие одно и то же имя, рассматриваются единой секцией, даже если находятся в разных участках программы. Побочные входы и выходы из критической секции запрещены.

С одной стороны, критические секции предоставляют удобный механизм для работы с общими переменными. Но с другой стороны, пользоваться им нужно осмотрительно, поскольку критические секции добавляют последовательные участки кода в параллельную программу, что может снизить её эффективность.

Также, неосмотрительное использование критических секций может приводить к взаимоблокировкам. При взаимоблокировке два или более потока не могут продолжать работу, так как пытаются получить доступ к некоторому ресурсу, захваченному другим потоком. В данном случае таким ресурсом являются критические секции.

Допускается использование критических секций вне параллельных конструкций для синхронизации пользовательского кода, распараллеленного другими способами.

Вложение параллельных конструкций

Параллельные конструкции могут быть вложены как по тексту, так и в случае, если из одной параллельной конструкции вызывается процедура, содержащая другую параллельную конструкцию.

Эффективность такого распараллеливания может сильно варьироваться. Так, например, при распараллеливании рекурсивной процедуры, содержащей несколько параллельных секций, каждая из которых вызывает эту же процедуру, при большой глубине рекурсивных вызовов накладные расходы на создание новых потоков могут существенно превосходить выигрыш в скорости, связанный с распараллеливанием. В таком случае имеет смысл запретить распараллеливание вложенных конструкций. При этом только на самом верхнем уровне такая процедура будет распараллелена, а все вложенные параллельные области будут выполняться последовательно.

В то же время, если процедура содержит небольшое количество параллельных секций (меньше чем количество процессоров в системе), и глубина рекурсивных вызовов невелика, но каждый вызов требует существенного времени, запрет распараллеливания вложенных конструкций может ограничить эффективность распараллеливания, так как не все процессоры будут задействованы. В этом случае имеет смысл разрешить распараллеливание вложенных конструкций.

Для явного разрешения или запрета распараллеливания вложенных конструкций предусмотрены две функции и переменная окружения.

```
function omp_get_nested():integer;
procedure omp_set_nested(integer);
var OMP_NESTED:boolean;
```

Функция omp_get_nested возвращает 0 или 1 в зависимости от того, разрешено ли в момент ее вызова распараллеливание вложенных конструкций, 0 – запрещено, 1 – разрешено.

Процедура omp_set_nested позволяет запретить или разрешить распараллеливание вложенных конструкций. Если параметр равен нулю, процедура запрещает, иначе – разрешает распараллеливание.

Также можно работать напрямую с переменной OMP_NESTED. `False` означает запрет, `true` — разрешение распараллеливания.

Использование этих средств дает возможность управлять распараллеливанием вложенных конструкций, в том числе сочетая разное поведение в одной программе, разрешая в одних участках и запрещая в других.

Глава 2. Внутреннее представление компилятора PascalABC.NET

Стадии компиляции

Процесс компиляции здесь описывается в общих чертах. Более подробно структура компилятора PascalABC.NET, а также стадии компиляции описаны в [4].

В процессе компиляции программа из исходного текста на языке PascalABC.NET переводится в IL-код. Процесс компиляции состоит из следующих стадий:

1. Парсинг. На этом этапе парсер переводит исходный текст программы в синтаксическое дерево (разобранная программа без учета семантики).

2. Семантический анализ. Здесь синтаксическое дерево переводится в семантическое дерево. Семантическое дерево — это внутреннее представление программы, содержащее исчерпывающую информацию о правильной программе. Все ошибки компиляции выявляются либо в процессе парсинга, либо в процессе семантического анализа.

3. Генерация IL-кода. Правильная программа переводится из семантического дерева в байт-код.

Кроме того, существует механизм сохранения семантического дерева каждого unit-модуля исходной программы на диск в отдельный файл с расширением pcu и именем, как у исходного файла.

Если для некоторого модуля на диске есть созданный ранее pcu-файл, то вместо указанных стадий 1–2 применяется восстановление семантического дерева из pcu-файла.

Стадии компиляции можно схематично изобразить так (см. рис. 1):

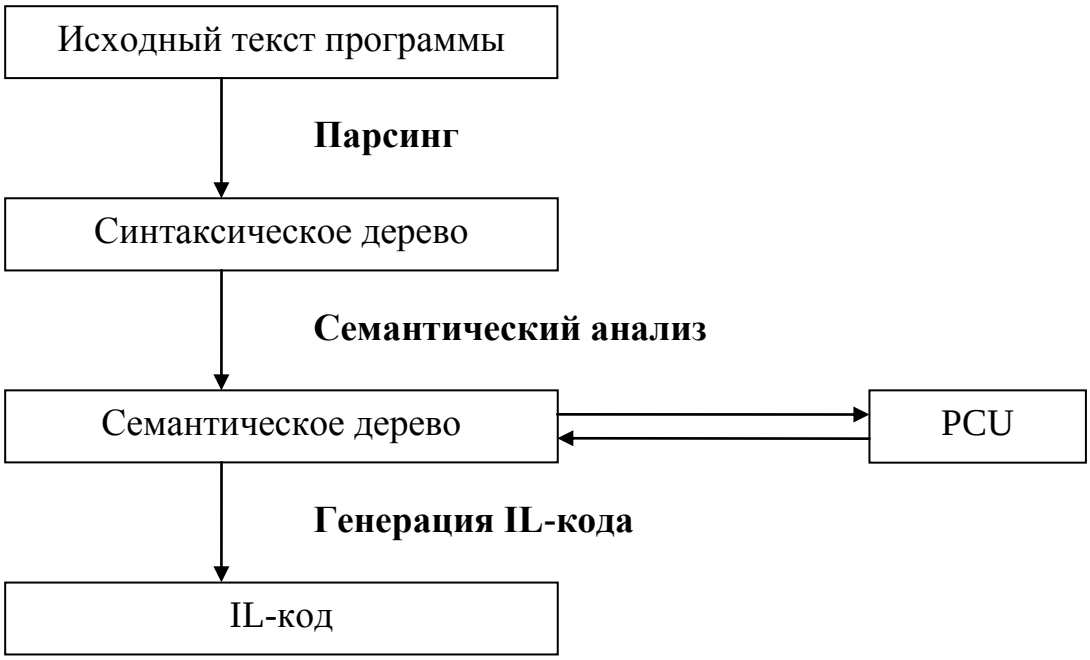

Рис. 1. Схема компиляции.

Внутренние представления, используемые на разных стадиях компиляции

Как видно из представленной выше схемы, в PascalABC.NET применяются два промежуточных внутренних представления: синтаксическое дерево и семантическое дерево. Каждое из этих представлений — это иерархия классов на языке C#.

Синтаксическое дерево — это набор классов, представляющих разобранную программу без учета семантики. Идентификаторы, встречающиеся в программе, представлены в виде строк, содержащих их имена. Синтаксическое дерево имеет довольно простую структуру, является деревом в терминологии теории графов. Фактически, синтаксическое дерево — это текст программы, переведенный парсером в удобное для последующей обработки представление.

Дальнейший перевод из синтаксического дерева в семантическое осуществляется посетителем по синтаксическому дереву, реализованному в модуле TreeConverter. Перевод из синтаксиса в семантику является самой

трудоемкой (как с точки зрения программирования, так и с точки зрения работы компилятора) частью всего компилятора PascalABC.NET. Алгоритмы поиска имен в таблицы символов, алгоритмы выбора перегруженных подпрограмм и многие другие алгоритмы, необходимые для анализа семантики, работают на этом этапе. Реализуемые автором в ходе данной работы распараллеливающие преобразования также выполняются на этом этапе. Получаемое на выходе семантическое дерево содержит всю информацию, необходимую для генерации IL-кода.

Семантическое дерево не является деревом в терминологии теории графов. Вхождения переменных, вызовы процедур содержат также ссылки на их определения. Поэтому узлы семантического дерева для двух функций, вызывающих друг друга, будут содержать ссылки друг на друга. Наглядно различия между синтаксическим и семантическим деревьями представлены на схеме (см. рис. 2) из [4, стр. 9]:

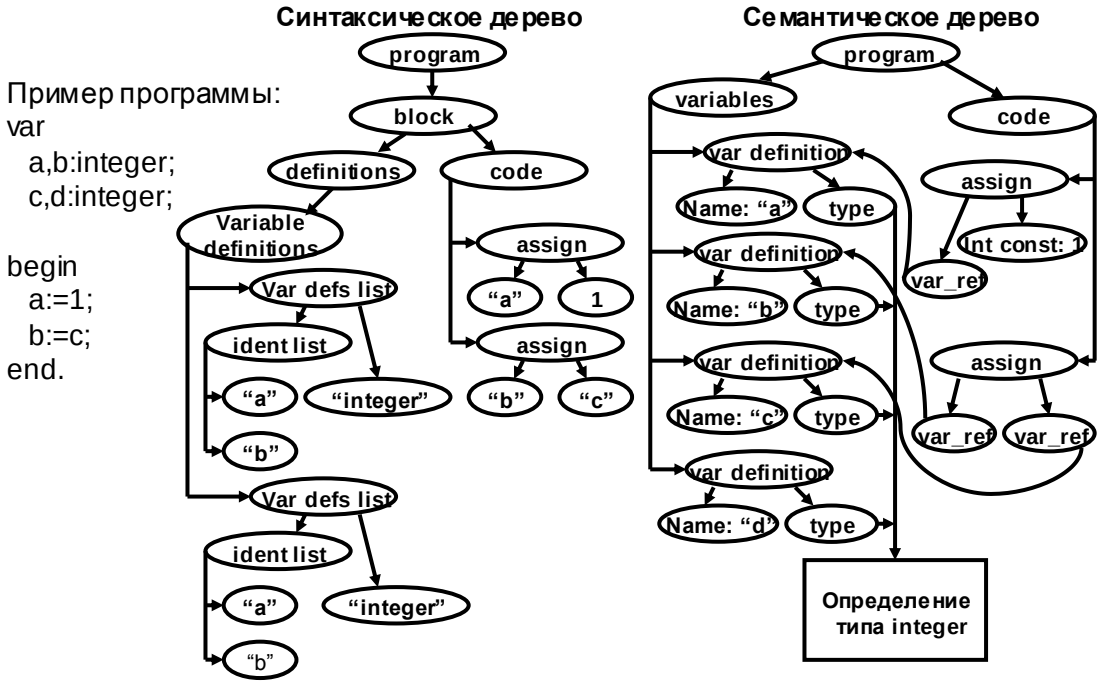

Рис. 2. Синтаксическое и семантическое деревья программы.

Разделение внутреннего представления на синтаксическое и семантическое деревья несколько нетрадиционно. Обычно в качестве внутреннего представления выступает одно дерево, которое сначала строится парсером, а затем аннотируется за несколько проходов. Основная мотивировка разделения на синтаксическое и семантическое деревья – принципиально разная структура этих деревьев, а также независимость конвертора синтаксического дерева в семантическое от используемого парсера. Следует отметить, что структура синтаксического дерева позволяет перевести программы на родственных языках (Pascal, Modula, Oberon) в синтаксические деревья близкой структуры, так что уже на этапе синтаксического дерева обеспечивается относительная независимость от языка [4, стр. 9].

Глава 3. Библиотека TPL

В работе используется библиотека TPL (Task Parallel Library). Библиотека содержит набор классов, предназначенных для облегчения написания параллельных программ.

Библиотека TPL входит в состав платформы .NET версии 4, а для версии 3.5 существует расширение Reactive Extensions [11], содержащее эту библиотеку в виде нескольких сборок, из которых используется сборка System.Threading.dll. Эта сборка включена в состав дистрибутива PascalABC.NET. В процессе установки системы программирования сборка добавляется в Global Assembly Cache операционной системы.

Распараллеливаемые программы подключают эту сборку во время работы, при первом выполнении параллельной конструкции. Сначала происходит поиск сборки в Global Assembly Cache, если сборка была найдена, она загружается и используется. Если в глобальном кэше сборка не была найдена (например, если на данном компьютере PascalABC.NET не был установлен), то сборка ищется в той директории, в которой находится исполняемый файл. Если здесь сборка была найдена, она загружается и используется. При переносе программы на другой компьютер достаточно поместить сборку в директорию с исполняемым файлом и программа будет работать параллельно. Если же ни в одном из этих мест сборка не обнаружена, программа будет выполняться последовательно. Таким образом, работоспособность программы сохраняется, хоть и в последовательном режиме. Для параллельного выполнения программы требуется сборка System.Threading.dll.

Параллельные конструкции, реализованные в ходе работы, используют два класса из библиотеки TPL. Распараллеливание циклов реализовано с использованием статического метода For класса Parallel, распараллеливание секций — с использованием класса Task из пространства имен

System.Threading. Оба класса расположены в пространстве имен System.Threading.Tasks.

В модуле PABCSystem.pas реализованы несколько процедур, обеспечивающих работу со сборкой System.Threading.dll и вызов ее методов. `__OMP_Available` при первом вызове пытается загрузить сборку и возвращает истину или ложь в зависимости от того, была ли загружена эта сборка. Если сборка загружена успешно, вызов метода `Parallel.For` осуществляется через процедуру `__OMP_ParallelFor`.

Метод `__OMP_ParallelFor` принимает три параметра – начальное и конечное значение счетчика цикла и делегат, с которым связана процедура с одним параметром – счетчиком цикла. Этот метод автоматически создает потоки и распределяет между ними нагрузку, вызывая из них процедуру, связанную с делегатом с разными значениями параметра в диапазоне цикла.

Библиотека реализует эффективные алгоритмы динамического распределения нагрузки, в частности, она порождает дополнительные потоки, в случае если некоторые из выполняющихся потоков оказались заблокированными, например, чтением файлов, ожиданием ввода от пользователя, чтобы избежать простоя процессоров. В статье [5] отмечается хорошая масштабируемость и высокая эффективность библиотеки, более высокая, чем при использовании пула потоков (`ThreadPool`).

Параллельное выполнение нескольких секций осуществляется через процедуру `__OMP_ParallelSections`, принимающую произвольное количество параметров-делегатов, для каждого из которых создается экземпляр класса Task, выполняющий параллельную секцию.

Класс `Task` реализует задачу, выполняемую асинхронно. Конструктор принимает параметр — делегат, с которым связана процедура. Выполнение процедуры начинается асинхронно в отдельном потоке после вызова метода `Start`. При вызове метода `Wait` происходит ожидание работы зада-

чи. Задача выполняется между вызовами методов `Start` и `Wait`, причем запуск задачи не задерживает выполнение потока, из которого задача запускается. Таким образом, можно запустить сразу несколько задач, которые при наличии свободных процессоров будут выполняться параллельно.

Глава 4. Обработка директив

Так как распараллеливающие преобразования определяются директивами и их опциями, перед выполнением преобразований нужно обработать директивы и извлечь из них необходимую информацию.

В ходе работы в грамматику парсера препроцессора [7] были внесены изменения, так как в исходном виде препроцессор не допускал наличие в тексте директивы некоторых символов, используемых в опции редукции.

Директива состоит из двух полей — имени и текста директивы. В дальнейшем рассматриваются только директивы с именем omp, т.е. имеющие вид {$omp ...}. Каждая директива, как и всякий узел синтаксического дерева, хранит свою позицию в тексте программы.

Два основных этапа работы с директивами – сопоставление директивы и узла синтаксического дерева, с которым она ассоциирована и разбор директивы, определение ее типа и опций.

В синтаксическом дереве все директивы хранятся в списке в узлах `program_module` и `unit_module`.

Сопоставление директив и узлов

Сопоставление директив и узлов синтаксического дерева осуществляется на основе их положения в тексте программы. Для этого был реализован посетитель синтаксического дерева (класс `SyntaxTreeNodeFinder`), который ищет наиболее внешний узел, целиком находящийся после заданной позиции в тексте.

Этот класс унаследован от класса `WalkingVisitor`, написанного автором ранее, который реализует только обход синтаксического дерева. В производном классе достаточно переопределить только функцию, вызываемую при посещении каждого узла. В этой функции происходит сравнение положения директивы и узла. Если узел целиком содержит директиву, происходит спуск по дереву на уровень ниже. Если узел расположен рань-

ше по тексту, чем директива — обход его подузлов не происходит, так как ни этот узел, ни его подузлы не является искомым.

Из узлов, расположенных позже по тексту, чем директива выбирается самый первый, подузлы таких узлов не посещаются. Таким образом, визитор находит самый внешний, расположенный выше всех по тексту узел, находящийся целиком после директивы.

Так как при обходе отсекаются ветви, заведомо не содержащие искомый узел, это существенно ускоряет поиск.

Пары «узел-директива» сохраняются в ассоциативном массиве (в классе `CompilerDirectivesToSyntaxTreeNodesLinker`), используемом в дальнейшем. Оба класса расположены в файле CompilerDirectivesToSyntaxTreeNodesLinker.cs

Разбор директив

Текст каждой директивы преобразуется в экземпляр класса DirectiveInfo, хранящей в удобном виде информацию о типе и опциях директивы.

Класс содержит:

- Тип директивы — перечисление.
- Список частных переменных, если это директива parallel for или parallel sections.
- Список переменных и операторов редукции, если это директива parallel for.
- Идентификатор, если это директива критической секции.

Разбор директивы происходит в конструкторе класса. Сначала определяется тип директивы, в зависимости от которого происходит анализ дальнейшей части текста директивы и заполняется поле Kind. Если обнаруживается неизвестный тип (не реализованная директива или ошибка в написании), разбор прекращается, выдается предупреждение и директива в дальнейшем игнорируется

Если это директива критической секции – весь дальнейший текст считается идентификатором.

Иначе в оставшейся части текста осуществляется поиск опций – списков частных переменных и редукции. Разбор опций реализован в виде двух конечных автоматов – один для списка частных переменных и один для оператора и списка переменных редукции.

При ошибках в синтаксисе выдается предупреждение и директива игнорируется.

Выдача сообщений об ошибках

Все сообщения об ошибках в директивах, параметрах, а также сообщения о том, что преобразование не выполнено, выдаются в виде предупреждений компиляции (Warning). При этом используется предусмотренная в системе программирования локализация сообщений об ошибках.

Для этого в PascalABC.NET\bin\Lng\ для каждого языка существует директория с именем, соответствующим коду языка. В данный момент это два языка – русский и английский (Rus и Eng соответственно). В каждой из этих директорий хранятся файлы, содержащие все текстовые строки, названия элементов экранных форм, сообщения и т.д. Все эти текстовые строки хранятся в виде «ключ=значение», где ключ – идентификатор строки, а значение – сама строка на соответствующем языке.

В коде системы PascalABC.NET записываются идентификаторы тех строк, которые необходимо выдать в случае ошибок, а на экран выдаются соответствующие им строки на том языке, который в данный момент используется.

Глава 5. Распараллеливающие преобразования программ во внутреннем представлении

Для выполнения распараллеливающих преобразований был выбран этап семантического анализа. На этом этапе конвертер строит семантическое дерево по имеющемуся синтаксическому дереву. Также в этот момент доступна таблица символов, предназначенная для быстрого поиска имен, встречающихся при компиляции [4, стр. 36]. Основная часть преобразований выполняется на уровне синтаксического дерева, ввиду простоты работы с ним. Для получения необходимой семантической информации используются узлы семантического дерева, а также осуществляется поиск в таблице символов.

Преобразования были реализованы в виде класса OpenMP, содержащего методы генерации параллельных конструкций для циклов, секций; методы генерации критических секций, а также вспомогательных методов и классов. Ввиду большого объема кода (более 2200 строк только в файле OpenMP.cs), в этой главе будут приведены только некоторые важные участки кода, все остальное описание преобразований будет сопровождаться указанием методов, в которых реализованы описываемые действия. Если не указано, в каком файле находится метод, считается, что он определен в файле OpenMP.cs.

Общий вид параллельных конструкций

Так как распараллеливание реализовано с использованием библиотеки TPL, программа зависит от сборки System.Threading.dll, которая может быть недоступна, в случае если программа была скомпилирована на одном компьютере и перенесена на другой. Поэтому для сохранения работоспособности программы принято решение генерировать и последовательный и параллельный код для параллельных конструкций. Во время работы программы производится попытка подключения сборки, и если сборка под-

ключена успешно, она используется и выполняется параллельный код. Иначе выполняется последовательный код.

Таким образом, в самом общем виде для параллельных конструкций генерируется семантическое дерево, соответствующее такому коду:

```
if __OMP_Available then
  parallel_block
else
  block;
```

`__OMP_Available` – функция из системного модуля, обеспечивающая подключение сборки. Возвращает истину или ложь в зависимости от того, была ли подключена сборка.

`block` и `parallel_block` – последовательный и параллельный вариант распараллеливаемого участка соответственно. При этом в последовательном варианте сборка не используется, благодаря чему сохраняется работоспособность программы при отсутствии сборки.

В условии также должна учитываться вложенность конструкций. Для этого в программе дополнительно объявляется переменная `$InParallelSection` (метод `CreateInParallelVariable`) которая используется в условии. Если распараллеливание вложенных конструкций разрешено, перед началом выполнения `parallel_block` ей присваивается значение true, а после – false (метод `CreateNestedRegionBorder`). По этой причине последовательный код для вложенных конструкций будут использоваться не только при отсутствии сборки System.Threading.dll, но и в случае если их распараллеливание запрещено.

С учетом этого параллельные конструкции выглядят так:

```
if __OMP_Available and not $InParallelSection then
begin
    if not OMP_NESTED then
        $InParallelSection := true;
    parallel_block;
    if not OMP_NESTED then
        $InParallelSection := false;
end
```

```
else
    block;
```

В зависимости от `OMP_NESTED` переменная `$InParallelSection` примет значение `true` либо останется `false`. Если в `parallel_block` содержится другая параллельная конструкция, то в зависимости от значения этой переменной будет выполнен либо последовательный, либо параллельный вариант кода.

Здесь и далее для объектов, создаваемых компилятором в процессе преобразования, используются названия, начинающиеся с символа $, чтобы избежать возможного совпадения с именами, используемыми в программе. Грамматика языка PascalABC.NET не позволяет использовать символ $ в идентификаторах.

Общее описание алгоритма

Здесь будет представлено общее описание алгоритма, отдельные этапы которого будут детализированы в дальнейшем.

Работа алгоритма происходит на этапе семантического анализа, когда построено синтаксическое дерево, в процессе обхода которого TreeConverter (посетитель синтаксического дерева) начинает строить семантическое дерево.

1. Перед началом обхода синтаксического дерева посетителем происходит сброс класса OpenMP в исходное состояние (Метод `InternalReset`).
2. Происходит обработка директив, процесс которой описан в 4 главе (Метод `InitOpenMP`). В результате получается ассоциативный массив, в котором по узлу можно получить директиву. Здесь же устанавливаются флаги `ForsFound` и `SectionsFound` в зависимости от того, есть ли в программе директивы `parallel for` и `parallel sections`.
3. После этого TreeConverter начинает обход синтаксического дерева. Этот посетитель содержит методы обработки всех узлов синтаксического де-

рева. Дальнейшая работа алгоритма происходит в трех методах этого посетителя:

```
statement_node convert_strong(SyntaxTree.statement st)
void visit(SyntaxTree.for_node _for_node)
void visit(SyntaxTree.statement_list _statement_list)
```

В первом осуществляется обработка критических секций, во втором — обработка параллельных циклов, в третьем — обработка параллельных секций. В начале и в конце этих методов вставлены вызовы методов класса OpenMP, причем, во втором и третьем методе эти вызовы выполняются только в случае, если соответствующие директивы были обнаружены, что определяется флагами, установленными на шаге 2.

Перейдем к описанию действий, выполняемых в этих трех случаях.

Преобразование параллельного цикла

В начале метода `visit` для узла `for_node` выполняется проверка, есть ли директива, соответствующая этому узлу. Если директивы нет, этот метод работает безо всяких изменений, генерирует узел семантического дерева, описывающий последовательный цикл и завершает работу.

Иначе, если узлу соответствует директива `parallel for`, проверяется текущее положение относительно параллельных конструкций. Это положение описывается переменной типа `ParallelPosition`:

```
enum ParallelPosition { Outside, InsideParallel,
                                 InsideSequential }
```

В любом из этих случаев нужно сгенерировать последовательную версию цикла. Параллельную версию нужно генерировать только в случае если в данный момент уже генерируется параллельная версия вышележащего цикла, либо если этот цикл не вложен в другие параллельные циклы. Признак необходимости генерации каждой версии цикла сохраняется во флагах `isGenerateParallel` и `isGenerate-Sequential`.

После этого выполняется оригинальный код метода `visit` для узла `for_node`.

Перед выходом из метода `visit`, когда последовательная версия цикла получена, проверяются флаги `isGenerateParallel` и `isGenerate-Sequential`. Если `isGenerateParallel` не установлен в true, то работа метода завершается, параллельную версию генерировать не нужно.

Иначе нужно сгенерировать и параллельную версию, и условный оператор, в зависимости от условия которого будет выполняться та или иная версия. Здесь происходит вызов метода `OpenMP.TryConvertFor`, который возвращает либо условный оператор, ветками которого являются две версии цикла, либо `null`. Если метод вернул не `null`, результатом работы метода visit является узел, сформированный методом `OpenMP.Try-ConvertFor`, иначе — последовательный вариант цикла.

`TryConvertFor` может вернуть `null`, в случае если при генерации параллельной версии обнаружена ошибка, например в цикле встретился оператор `break`.

Условный оператор, возвращаемый методом TryConvertFor, зависит от значения переменной `ParallelPosition`:

- Значение `ParallelPosition.Outside` требует вызова функции `__OMP_Available` и проверки вложенности.
- Значение `ParallelPosition.InsideParallel` требует только проверки вложенности.

Теперь рассмотрим метод `TryConvertFor`, генерирующий узлы семантического дерева, содержащие параллельный цикл.

Предварительный этап

Исходный последовательный цикл преобразуется в вызов метода `OMP_ParallelFor`, третьим параметром которого является делегат, исполняющий различные итерации цикла в диапазоне, задаваемом первыми двумя параметрами. В связи с чем, тело цикла выносится в отдельную процедуру с одним параметром-счетчиком цикла, а сам цикл заменяется на

вызов метода `__OMP_ParallelFor`, где в качестве третьего параметра передается созданная процедура.

Если тело исходного цикла состоит из единственного вызова процедуры с одним параметром-счетчиком цикла, эта процедура сразу передается третьим параметром в `__OMP_ParallelFor` и все преобразования, описанные далее вплоть до генерации условного оператора, не выполняются.

Иначе тело цикла необходимо преобразовать в отдельную процедуру. Некоторые используемые переменные, доступные в исходном цикле, могут оказаться недоступными после вынесения тела цикла в отдельную процедуру, если процедура не принадлежит области видимости этой переменной.

Например, в таком случае:

```
procedure p;
begin
  var a:integer:=0;
  {$omp parallel for}
  for var i:integer:=1 to 10 do
    ... := a;
end;
```

После преобразования вид семантического дерева будет соответствовать такому коду (здесь и далее преобразования семантического дерева для наглядности будут показаны в виде кода, соответствующего преобразованному дереву):

```
procedure LoopBody(i:integer);
begin
  ... := a;
end;

procedure p;
begin
  var a:integer:=0;
  __OMP_ParallelFor(1, 10, LoopBody);
end;
```

В процедуре `LoopBody` переменная a не определена. Генерация кода по такому семантическому дереву приведет к ошибкам.

Поэтому было решено тело цикла выносить не в процедуру, а в метод класса. Для всех переменных, которые становятся недоступными после вынесения тела цикла, в классе создаются поля с такими же именами, имеющие такой же тип, как и исходные переменные. Преобразованное тело цикла будет работать с этими полями, после чего остается только инициализировать значение этих полей значениями соответствующих переменных перед выполнением цикла и присвоить значение полей этим переменным после выполнения цикла.

Поиск таких переменных осуществляется посетителем синтаксического дерева, осуществляющим обход исходного цикла, класс `VarFinderSyntaxVisitor` (находится в файле VarFinderSyntaxVisitor.cs). Он также унаследован от класса `WalkingVisitor`, осуществляющего обход синтаксического дерева, с переопределением нескольких методов:

- `visit(ident value)` – здесь происходит поиск каждого идентификатора в таблице символов. В результате поиска становится известен тип идентификатора, в зависимости от которого этот идентификатор добавляется или не добавляется в список переменных, требующих создания полей класса. Так, например, глобальные переменные не требуют создания полей, так как они будут доступны и после преобразования, переменные, объявленные внутри цикла, также не требуют создания полей, так как их объявления выносятся вместе с телом цикла, а вот переменные, объявленные в том же блоке, что и цикл, требуют создания полей.
- `visit(dot_node _dot_node)` – здесь переопределено посещение подузлов. Такие узлы описывают обращение к части составного объекта, например, полю класса. В этом случае достаточно найти только имя составного объекта, а не всех его полей.
- `visit(procedure_call _proc_call)`. Оператор break не может быть использован в распараллеливаемом цикле. В синтаксическом

дереве этот оператор описывается узлом `procedure_call`. Поэтому здесь происходит проверка на использование этого оператора. Если оператор break был найден в исходном цикле, выдается предупреждение и генерация параллельного цикла прекращается. Оператор `continue` после вынесения тела цикла в отдельную процедуру должен быть заменен на `Exit`.

Результатом работы этого посетителя является список имен переменных и локальных констант, для которых требуется создание полей в генерируемом классе.

Следующим действием нужно учесть опции директивы OpenMP. В ходе разбора директивы были получены два списка имен переменных – частные переменные и переменные редукции.

Для этого был реализован вспомогательный класс `VarInfoContainer`, в котором в нескольких списках хранятся отдельно частные переменные, отдельно переменные и операторы редукции и отдельно все остальные переменные, а также локальные константы. В дальнейшем элементы разных списков будут обрабатываться по-разному. Функция `GetVarInfoContainer` создает экземпляр такого класса на основе списка переменных, полученных посетителем и экземпляра `DirectiveInfo`, описывающего текущую директиву.

В ходе работы этой функции также обнаруживаются следующие виды ошибок:

- Редукция по переменной-счетчику цикла. Это ошибка, так как счетчик цикла изменять нельзя.
- Редукция по несуществующей переменной. Если в опции директивы было указано несуществующее в программе имя.
- Редукция не по переменной, либо по переменной недопустимого типа. Если в опции директивы было указано имя константы, проце-

дуры, класса и т.д. Допускаются только целые, вещественные и логические типы переменных.

- Указание в опции `private` идентификатора, не являющегося переменной.

Для каждой из этих ошибок создается предупреждение компиляции, и этот идентификатор в опции директивы игнорируется.

После этого создается описание класса, содержащего все необходимые поля и метод-тело цикла. Сначала опишем, как будут обрабатываться частные переменные, переменные редукции и остальные переменные.

Обработка различных видов переменных

Для переменных редукции необходимо создавать как поле класса, так и локальную переменную. В начале итерации переменная будет инициализирована значением по-умолчанию, в зависимости от операции редукции, в конце итерации к полю класса и локальной переменной будет применена операция редукции, результат будет сохранен в поле класса. Изменения поля производится в критической секции, для этого в классе создается объект, который будет хранить блокировку.

Исходный цикл:

```
{$omp parallel for reduction(+:a)}
for var i:=1 to 10 do
  //тело цикла
```

Для этого цикла будет создан следующий класс:

```
$ForClass = class
  $a:integer;
  $Lock:object:= new Object;
  procedure Method(i:integer);
  var
    a:integer
  begin
    a:=0;
    //тело цикла
    lock $Lock do
      $a:=$a + a;
  end;
```

```
end;
```

А вызов будет иметь следующий вид:

```
var $ForObj: $ForClass:= new $ForClass;
$ForObj.a:= a;
__OMP_ParallelFor(1, 10, $ForObj.Method);
a:= $ForObj.a;
```

Каждый поток должен иметь свои экземпляры всех частных переменных. Опция `private` не обеспечивает ни инициализации таких переменных, ни сохранения результирующего значения в исходной переменной, поэтому достаточно объявить локальную переменную с таким именем и типом в методе-теле цикла. Например (здесь и далее считается, что в теле цикла есть обращение к переменной `a`):

```
{$omp parallel for private(a)}
for var i:=1 to 10 do
  //тело цикла
```

Для такого параллельного цикла будет создан следующий класс:

```
$ForClass = class
  procedure Method(i:integer);
  var
    a:integer
  begin
    //тело цикла
  end;
end;
```

А вызов будет иметь следующий вид:

```
var $ForObj: $ForClass:= new $ForClass;
__OMP_ParallelFor(1, 10, $ForObj.Method);
```

В результате различные итерации будут работать с различными экземплярами переменной.

Для всех остальных переменных создается только поле класса.

```
{$omp parallel for}
for var i:=1 to 10 do
  //тело цикла
```

Этому примеру будет соответствовать такой класс:

```
$ForClass = class
```

```
  a:integer;
  procedure Method(i:integer);
  begin
    //тело цикла
  end;
end;
```

Вызов будет иметь следующий вид:

```
var $ForObj: $ForClass:= new $ForClass;
$ForObj.a:= a;
__OMP_ParallelFor(1, 10, $ForObj.Method);
a:= $ForObj.a;
```

Создание класса

Класс создается из узлов синтаксического дерева. Такое решение было принято в связи с простотой работы с синтаксическим деревом. Дальнейший перевод класса, построенного в виде фрагмента синтаксического дерева, будет осуществляться уже работающим посетителем. Подробнее этот процесс будет рассмотрен позже, после описания процесса создания класса из узлов синтаксического дерева и вызова его метода.

Создание класса реализовано в функции `CreateClass`. В параметрах этой функции задается имя создаваемого класса и `VarInfoContainer`, содержащий списки переменных, которые будут обрабатываться, по описанным выше правилам. Для создания полей класса вызывается функция `CreateClassMember`. Она принимает узел семантического дерева, описывающий переменную или константу. `ConvertToSyntaxType` обеспечивает преобразование семантического типа в синтаксический. `ConvertConstant` обеспечивает преобразование значения константы.

Метод класса создается отдельно функцией `CreateMethod`. Позже, она также будет использоваться при генерации параллельных секций. В этой функции также реализована обработка переменных, передаваемых ей параметром `VarInfoContainer`, как описано выше.

Следующий шаг — генерация узлов синтаксического дерева, описывающих создание класса, инициализацию его полей и вызов `__OMP_ParallelFor`.

Создание вызова параллельного кода

Все эти узлы будут храниться в виде списка операторов `statement_list`. Этот список заполняется за несколько действий:

```
SyntaxTree.statement_list stl = CreateInitPart(ClassName,
           ObjName, Vars);
stl.subnodes.Add(CreateNestedRegionBorder(true));
stl.subnodes.Add(CreateOMPParallelForCall(dn,
           for_node.initial_value, for_node.finish_value));
stl.subnodes.Add(CreateNestedRegionBorder(false));
stl.subnodes.AddRange(CreateFinalPart(ObjName,
           Vars).subnodes);
```

- `CreateInitPart` принимает в качестве параметра `VarInfoContainer` и генерирует создание класса и инициализацию его полей.
- `CreateNestedRegionBorder` с параметром `true` создает условный оператор, проверяющий вложенность параллельной конструкции с присваиванием переменной `$InParallelSection` значения true.
- `CreateOMPParallelForCall` создает вызов библиотечной функции `__OMP_ParallelFor`, которая вызывает метод `__OMP_ParallelFor`.
- `CreateNestedRegionBorder` с параметром `false` создает условный оператор, проверяющий вложенность параллельной конструкции с присваиванием переменной `$InParallelSection` значения `false`.
- `CreateFinalPart` создает присваивание значений полей класса соответствующим переменным.

Перевод синтаксических узлов в семантические

Стоит напомнить, что весь описываемый процесс происходит во время обработки посетителем узла синтаксического дерева, описывающего

цикл for. При этом были созданы также вспомогательные узлы синтаксического дерева. Они содержат описание класса, а также создание класса, инициализацию его полей и вызов `__OMP_ParallelFor`. Эти узлы не хранятся в исходном синтаксическом дереве, они содержатся во временных переменных. Чтобы создаваемое семантическое дерево содержало и эти узлы, посетитель, выполняющий преобразование синтаксического дерева должен посетить и их.

В случае с узлами, описывающими создание класса и вызов `__OMP_ParallelFor` это достигается только вызовом функции visit посетителя с параметром – созданным узлом:

```
syntax_tree_visitor.ret.visit(stl);
```

Эта функция возвращает узлы семантического дерева, соответствующие построенному фрагменту синтаксического дерева.

В случае с узлами, хранящими описание класса необходимо учитывать то, что посетитель хранит свое состояние, которое описывает текущее пространство имен и стек пространств имен, текущую функцию, текущий класс и т.д. Вызвать visit, передав параметром созданные узлы нельзя, так как при этом происходят семантические проверки, зависящие, в том числе от состояния (контекста) посетителя.

Решением данной проблемы является сохранение контекста во временных переменных, и изменение контекста на такое состояние, в котором находится посетитель при обработке объявления классов. После этого можно вызвать функцию visit посетителя и передать ей параметром созданные узлы синтаксического дерева. Посетитель преобразует эти узлы в соответствующие им узлы семантического дерева и сохранит их в строящемся семантическом дереве. Таким образом, сгенерированное семантическое дерево будет содержать описание класса, хотя в исходном синтаксическом дереве его нет. После этого необходимо восстановить состояние

посетителя, чтобы он смог продолжить обход дерева. Таким образом, полностью генерация класса реализуется так:

```
ContextInfo contextInfo = new
    ContextInfo(syntax_tree_visitor);

string ClassName =
    syntax_tree_visitor.context.get_free_name("$for_class{0}");

try
{
    SyntaxTree.class_members member;
    SyntaxTree.type_declarations Decls =
    CreateClass(ClassName, out member, Vars);
    member.members.Add(CreateMethod("Method", syntax_body,
                for_node.loop_variable.name, member, Vars));
    syntax_tree_visitor.visit(Decls);
}
finally
{
    contextInfo.RestoreContext(syntax_tree_visitor);
}
```

Класс `ContextInfo` реализует сохранение и восстановление состояния посетителя.

Создание условного оператора

Как было описано ранее, параллельная конструкция имеет вид условного оператора, с двумя ветками – последовательным и параллельным вариантом цикла.

Последовательная версия была получена в ходе работы посетителя до начала преобразований и передана в метод `OpenMP.TryConvertFor`. Параллельная версия была получена в ходе работы вызываемого из него метода `GenerateOMPParallelForCall`. Если параллельная версия цикла была получена без ошибок, создается условный оператор, который и является результатом работы всех описанных преобразований. Его создание происходит в функции `CreateIfCondition`. Ей передаются оба варианта цикла. Также она учитывает вид условия: проверка доступности сборки System.Threading.dll требуется только для самой внешней параллельной

конструкции. Очевидно, что если выполняется параллельная версия цикла, то эта сборка доступна и нет необходимости проверять ее доступность для вложенных конструкций.

Преобразование параллельных секций

Преобразование параллельных секций во многом похоже на преобразование параллельного цикла.

Рассмотрим следующий участок кода (назовем его блоком секций):

```
{$omp parallel sections}
begin
  //секция 1
  //секция 2
end;
```

Для этого кода в семантическом дереве будет построен класс следующего вида:

```
$SectionsClass = class
  //поля, соответствующие переменным
  procedure Method0;
  //частные переменные первой секции
  begin
    //секция 1
  end;
  procedure Method1;
  //частные переменные второй секции
  begin
    //секция 2
  end;
end;
```

А сам вызов будет иметь следующий вид:

```
var $SectionsObj: $SectionsClass := new $SectionsClass;
//инициализация полей класса значениями переменных
__OMP_ParallelSections($SectionsObj.Method0,
                       $SectionsObj.Method1);
//присваивание переменным значения полей
```

Процесс начинается в методе `visit(SyntaxTree.statement_list _statement_list)`, где проверяется наличие директивы перед этим списком операторов. Если директивы нет, этот метод работает без изменений.

Если директива есть, то, как и в случае с параллельным циклом, устанавливаются флаги, отвечающие за необходимость генерации каждого варианта секций по тем же правилам. В конце этого метода при необходимости происходит вызов `OpenMP.TryConvertSections`, который возвращает сформированный узел семантического дерева — условный оператор с двумя ветками — последовательным и параллельным вариантом блока секций.

Затем для каждой секции происходит поиск используемых в ней переменных, которые станут недоступными после вынесения секций в отдельные методы генерируемого класса. Все переменные для всех секций объединяются в один общий список. Затем на основе этого списка и опций директивы строится объект `VarInfoContainer`, содержащий информацию о переменных различных видов. Единственное отличие от случая с параллельным циклом – параллельные секции не предусматривают редукции.

После этого для текущего блока секций создается класс, содержащий поля, определенные на предыдущем шаге. Для каждой секции в этом классе создается метод, реализующий эту секцию. Отличие от параллельного цикла состоит в параметре методов. Для параллельного цикла это был счетчик цикла. Здесь же создается неиспользуемый параметр типа Object, так как конструктор используемого для реализации параллельных секций класса Task принимает в качестве параметра процедуру с одним параметром типа `Object`.

Следующий шаг — сохранение состояния посетителя, запуск обхода посетителем построенного класса, в результате которого класс сохраняется в семантическом дереве.

После чего генерируется и преобразуется в узлы семантического дерева создание объекта, инициализация его полей, вызов `__OMP_ParallelSections` с параметрами-методами класса и присваивание переменным значений полей класса.

Преобразование критических секций

Критические секции реализованы с использованием средств языка PascalABC.NET – оператора `lock`:

```
lock объект_блокировки do
     оператор
```

При выполнении этого кода на объект накладывается блокировка, таким образом, одновременно только один поток может выполнять оператор.

Для правильной работы критических секций необходимо, чтобы эти объекты были созданы, а также, чтобы для разных критических секций с одним и тем же именем использовался один и тот же объект, чтобы исключить возможность одновременного выполнения критических секций с одним именем.

В начале метода `convert_strong(SyntaxTree.statement st)` посетителя TreeConverter проверяется наличие директивы `omp critical`, соответствующей этому оператору. Если директива есть, то для текущего узла вызывается метод `OpenMP.TryConvertCritical`.

При первом вызове этого метода происходит создание класса, хранящего все объекты блокировок для каждой критической секции, существующей в программе. Перебираются все директивы, из них извлекаются имена критических секций и создаются объекты с такими именами. Для «безымянной» критической секции создается объект с именем `$Default`. Объекты создаются как статические поля класса и сразу инициализируются. Таким образом, из любого места программы можно получить доступ к необходимому объекту. При создании класса используется подход, описанный ранее, при котором класс создается из узлов синтаксического дерева, а затем преобразуется в семантическое дерево уже работающим в данный момент посетителем, с изменением и восстановлением его состояния.

Затем метод `TryConvertCritical` создает оператор `lock`, накладывающий блокировку на объект, имя которого задано в директиве, а оператором является текущий оператор.

Оператор `lock`, созданный этим методом возвращается в `convert_strong`, и вместо исходного оператора преобразуется в узел семантического дерева. Так как своей частью оператор lock содержит исходный оператор, которому соответствует директива, то при последующем посещении этого подузла вышеописанный процесс будет повторяться. Чтобы избежать этого зацикливания, в начале метода `convert_strong` директива удаляется из ассоциативного массива, а в конце, когда оператор `lock` уже полностью обработан, снова помещается в ассоциативный массив. В результате только первый раз создается оператор lock, а при последующем посещении исходного оператора директива в ассоциативном массиве не обнаруживается и никаких дополнительных преобразований не происходит.

Так как при реализации критических секций были использованы только имеющиеся в языке PascalABC.NET возможности, критические секции не зависят от сборки System.Threading.dll, не требуют ее подключения, а поэтому сохраняют работоспособность при переносе программы на другой компьютер. Также директивы omp critical можно использовать для синхронизации кода, распараллеленного с использованием любых средств, а не только директив OpenMP.

Глава 6. Оценка производительности

Данная глава посвящена двум вопросам, связанным с производительностью. Первый — насколько повлияла реализация распараллеливающих преобразований на скорость компиляции. Второй — какой прирост производительности можно получить от использования распараллеливания реализованными средствами.

Скорость компиляции

Для выяснения влияния распараллеливающих преобразований на скорость компиляции была создана копия проекта PascalABC.NET, в которой были удалены все реализованные преобразования. Замер времени осуществлялся средствами самого компилятора. Компилятор замеряет время компиляции с точностью порядка 15 мс, поэтому для увеличения точности и уменьшения влияния внешних факторов каждый тестовый пример компилировался многократно, затем время усреднялось. Все результаты представлены в миллисекундах.

Тестовые примеры можно разделить на две группы:

1. Программы, не содержащие директив OpenMP. Для таких программ замедление, по возможности, должно быть минимальным.
2. Программы, содержащие директивы OpenMP.

Результаты для первой группы представлены в таблице 2:

Программа	Время компиляции без преобразований	Время компиляции с преобразованиями	Замедление в %
PABCSystem.pas	1021	1028	0,69%
Быстрая сортировка	142	143	0,70%
Игра "Жизнь"	143	142	-0,70%
Ханойские башни	85	85	0,00%

Таблица 2. Влияние преобразований на время компиляции программ без директив OpenMP.

Из таблицы видно, что влияние реализованных преобразований в этом случае незначительно. В этом случае выполняется только перебор

всех директив и сравнение их имен со строкой «omp», а также по 2 условных оператора, проверяющих состояние переменной логического типа при посещении каждого узла синтаксического дерева, условия которых в этом случае всегда ложны.

Иная ситуация наблюдается в случае, если в коде программы есть директивы OpenMP, и для них требуется выполнение преобразований.

Результаты представлены в таблице 3:

Программа	Время компиляции без преобразований	Время компиляции с преобразованиями	Замедление в %
Система частиц	96	99	3,13%
Быстрая сортировка	65	71	9,23%
Умножение матриц	59	68	15,25%
Сортировка массивов	71	78	9,86%

Таблица 3. Влияние преобразований на время компиляции программ с директивами OpenMP.

Здесь выполнение действий по разбору директив, поиску используемых переменных, генерации узлов синтаксического дерева с последующим преобразованием их в семантическое дерево приводят к некоторому замедлению компиляции.

Влияние распараллеливания

Теперь посмотрим, какой прирост скорости можно получить с использованием директив OpenMP.

Тесты проводились на двух компьютерах: с двухъядерным процессором Intel Pentium Dual-Core E2180 и трехъядерным AMD Phenom II x3 N830. Это позволяет в некоторой степени оценить не только прирост производительности как таковой, но и влияние увеличения количества процессоров на ускорение программы.

Измерение времени производится с использованием класса System.Diagnostics.StopWatch, который обеспечивает точность порядка микросекунд, однако для уменьшения влияния внешних факторов измерение

времени выполнения программ производится многократно с усреднением результатов и округлением их до миллисекунд.

Для начала, следует отметить следующий факт: так как распараллеливание выполняется с использованием сборки System.Threading.dll, которая подключается при первом выполнении параллельной конструкции (а точнее, при первом вызове функции `__OMP_Available`), на это тратится небольшое количество времени. Например:

```
uses System.Diagnostics;

begin
  var sw:StopWatch:= new Stopwatch;
  sw.Start;

  {$omp parallel for}
  for var i:=1 to 2 do;

  sw.Stop;
  writeln(sw.ElapsedMilliseconds);
end.
```

Здесь замеряется время выполнения пустого цикла из двух итераций. Время выполнения такой программы без директивы `omp parallel for` не превышает одной миллисекунды. Время выполнения программы с директивой – 66 миллисекунд. Все последующие параллельные конструкции уже не оказывают такого эффекта. Из этого следует, что использование OpenMP нецелесообразно, в случае если в программе имеется только одна параллельная конструкция, которая работает небольшое время по сравнению со временем, необходимым на подключение сборки System.Threading.dll.

Теперь перейдем к тестированию производительности на нескольких тестовых программах. Во всех программах сравнивается время выполнения (частота кадров для программы «система частиц») в последовательном варианте и в параллельном. В таблице 4 представлено ускорение параллельных программ на двух указанных процессорах, относительно последовательной программы на том же процессоре:

Программа	Phenom II x3	Pentium Dual-Core
Умножение матриц 1	2,86	2,00
Умножение матриц 2	2,71	1,77
Умножение матриц 3	0,21	0,13
Простые числа	2,76	1,75
Project Euler #74	1,93	1,67
Project Euler #32	1,44	1,49
Система частиц	1,70	1,57
Quick Sort	1,44	1,29
Quick Sort Nested	0,67	0,56

Таблица 4. Относительное ускорение от распараллеливания.

Умножение матриц (простой алгоритм). Различия между тремя вариантами заключаются в том, что для распараллеливания выбраны различные циклы. В первом случае распараллеливается самый внешний цикл, во втором – средний, в третьем – внутренний. Распараллеливание первых двух циклов не требует никаких опций, для распараллеливания внутреннего цикла необходима редукция, так как в нем происходит изменение накопление суммы в переменной. Три варианта были протестированы, чтобы показать, в каких случаях можно получить наибольший прирост производительности. Распараллеливание самого внешнего цикла наиболее эффективно, позволяет получить ускорение, почти равное количеству процессоров, чуть менее эффективно распараллеливание среднего цикла. Чем дольше работает каждая итерация, тем большее ускорение можно получить. Распараллеливание внутреннего цикла приводит к замедлению, так как на каждой итерации, где происходит только одно умножение и сложение, появляются блокировки. Также велики накладные расходы на вызовы процедуры, в которую было преобразовано тело цикла, а количество вызовов очень велико.

Простые числа — суммирование простых чисел в заданном диапазоне. Используется параллельный цикл с редукцией. Также наблюдается существенное ускорение, несмотря на наличие редукции, так как каждая

итерация выполняется значительное время (проверка простоты больших чисел).

Моделирование **системы частиц**. Гнездо циклов, из которых распараллеливается только внутренний, а внешний цикл, кроме внутреннего цикла содержит еще некоторые действия, выполняемые последовательно. Последовательные участки ограничивают максимальный прирост производительности в соответствии с законом Амдала [8, стр. 87].

Быстрая сортировка. Рекурсивный алгоритм с использованием параллельных секций. Рассматривается два случая: распараллеливание вложенных конструкций в одном случае разрешено (QuickSort Nested в таблице), в другом – запрещено (QuickSort). В первом случае наблюдается ускорение, хоть и небольшое, во втором – замедление, так как при каждом вызове рекурсивной функции происходит создание двух задач, сортирующих участки массива и затраты на их создание превышают эффект от распараллеливания.

Решения задач **32** и **74** с сайта **projecteuler.net** [9]. Параллельный цикл с редукцией. Прирост производительности в 1,5–2 раза. Прирост ограничивает редукция и работа с памятью.

Таким образом, наибольший прирост производительности наблюдается в случаях, когда каждая итерация распараллеливаемого цикла или параллельная секция работают значительное время. Редукция немного уменьшает прирост, а в сочетании с «короткими» итерациями может привести к замедлению. Чрезмерная вложенность параллельных конструкций также способна вызвать замедление программы. Однако два примера, показавшие замедление были искусственно созданы, чтобы показать границы применимости такого распараллеливания. Обе задачи можно решить и без замедления.

Заключение

Данная работа посвящена реализации поддержки компилятором PascalABC.NET директив OpenMP. Использование реализованных средств позволяет повышать производительность программ путем их распараллеливания. При этом требуется внесение минимума изменений в уже готовую программу, что хорошо подходит для оптимизации программ на поздних этапах разработки.

В ходе работы были реализованы следующие задачи:

- Произведен разбор и преобразование директив к удобному для дальнейшей работы виду.
- Реализована обработка ошибок в директивах и их опциях и локализация сообщений об ошибках.
- Параллельные конструкции во внутреннем представлении преобразованы в методы классов, содержащих используемые переменные в качестве полей.
- Вызов методов осуществляется с помощью библиотеки TPL, для работы с которой реализован набор системных функций.
- Библиотека TPL включена в дистрибутив PascalABC.NET и устанавливается в операционной системе при установке системы программирования.
- Реализована возможность программ сохранять работоспособность без библиотеки TPL.
- Проанализировано влияние реализованных преобразований на время компиляции.
- Проанализировано влияние реализованных возможностей на ускорение программ.
- Реализованы дополнительные элементы стандарта OpenMP — процедура omp_set_nested, функция omp_get_nested и переменная

OMP_NESTED, использование которых позволяет избежать падения скорости в некоторых случаях при распараллеливании рекурсивных процедур.

Реализованные в ходе работы возможности вошли в состав системы программирования PascalABC.NET версии 1.7, вышедшей 13 апреля 2011 года.

Список литературы

1. Сайт системы программирования PascalABC.NET. http://pascalabc.net/.
2. Спецификация OpenMP. http://www.openmp.org/mp-documents/spec30.pdf.
3. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб. : Питер, 2001.
4. Ткачук А.В. Язык, компилятор и система программирования PascalABC.NET. Дипломная работа, 2007 г.
5. Параллельная производительность: Оптимизация управляемого кода для многоядерных компьютеров. http://msdn.microsoft.com/ru-ru/magazine/cc163340.aspx.
6. Компиляторы, поддерживающие стандарт OpenMP. http://openmp.org/wp/openmp-compilers/.
7. Зарубин М.А. Препроцессор PascalABC.NET. Бакалаврская диссертация, 2009 г.
8. В.В. Воеводин, Вл.В. Воеводин. Параллельные вычисления. СПб. : БХВ-Петербург, 2002.
9. Project Euler. http://projecteuler.net/.
10. Водолазов Н.Н. Конвертор в семантическое дерево для компилятора PascalABC.NET. Магистерская диссертация, 2007 г.
11. Reactive Extensions for .NET. http://msdn.microsoft.com/en-us/data/gg577610.