ISSN 2071-2898

О р д е н а Л е н и н а
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В.Келдыша
Р о с с и й с к о й а к а д е м и и н а у к

А.Н. Андрианов, А.Б. Бугеря,
К.Н. Ефимкин, П.И. Колударов

Модульная архитектура компилятора языка Норма+

Препринт №

Москва — 2011

Андрианов А.Н., Бугеря А.Б., Ефимкин К.Н., Колударов П.И.

Модульная архитектура компилятора языка Норма+

В работе предлагается декларативный подход к созданию прикладного программного обеспечения для высокопроизводительных вычислительных систем на примере языка Норма+. Дано описание компилятора языка Норма+ и его внутренней структуры. Рассмотрены основные модули компилятора и вопросы создания новых модулей кодогенератора для различных архитектур вычислительных систем.

Ключевые слова: высокопроизводительные вычисления, декларативные языки программирования

Alexander Nikolaevich Andrianov, Alexander Borisovich Bugerya, Kirill Nikolaevich Efimkin, Pavel Ivanovich Koludarov

Modular architecture of NORMA+ compiler

In this article authors provide the declarative approach to software development for high-performance systems based on Norma+ language. Norma+ compiler with its internal modular architecture is introduced. Major compiler modules and methods allowed to create the new code generator’s modules for different computer systems are described.

Key words: high-performance computing, declarative programming languages

Работа выполнена при финансовой поддержке гранта РФФИ N 09-01-00329.

Оглавление

Введение.

Наличие проблем, связанных с разработкой эффективных параллельных программ для различных видов параллельных архитектур, является в настоящее время признанным фактом, который неоднократно обсуждался и продолжает обсуждаться, как с точки зрения природы этих проблем, так и с точки зрения ответа на вопрос – что делать для их решения.

Достаточно быстрое развитие новых аппаратных возможностей для поддержки параллельных вычислений, наблюдаемое в последнее время, еще больше усложняет проблему. Например, появление массово доступных многоядерных процессоров поставило вопрос об эффективном программировании для них. Практически одновременно появились массово доступные графические ускорители (графические процессоры), и опять возник вопрос об эффективном программировании для них. Агрессивное продвижение своих решений фирмами-производителями вычислительных систем, обладающих этими возможностями, часто дезориентирует прикладных специалистов, разрабатывающих параллельные вычислительные программы, толкает их на изменение средств разработки программ, хотя ясные и достаточно убедительные аргументы в пользу таких изменений отсутствуют. Так, применение технологии CUDA для эффективного программирования для графических ускорителей является, фактически, программированием на уровне ассемблера, с учетом тонких особенностей аппаратуры.

В настоящее время основной метод разработки эффективных параллельных программ для решения сложных вычислительных задач - это ручное программирование на языках Фортран или Си с использованием библиотеки передачи сообщений MPI (ставшей стандартом параллельного программирования “де-факто”), и/или с использованием стандарта OpenMP (встречалось реже, но в последнее время в связи с распространением многоядерных систем вновь приобрело актуальность).

Работы по созданию и продвижению новых средств и языков программирования (например, языков Chapel [1], X10 [2], Fortress [3], UPC [4], Coarray Fortran [5], Charm++[6], различных сред и технологий программирования графических процессоров, способов программирования реконфигурируемых вычислителей, построенных на программируемой логике (FPGA), например [7], и прочее) ведутся весьма активно, однако проблема простой разработки параллельных программ и утилизации новых возможностей вычислительной техники остается в настоящее время не решенной.

Надежды на автоматическое распараллеливание последовательной программы также пока не оправдываются: для параллельных вычислительных систем с общей памятью распараллеливающие компиляторы не обеспечивают должное эффективное распараллеливание, а для параллельных вычислительных систем с распределенной памятью проводятся лишь исследовательские работы в данном направлении. Для архитектур типа графических процессоров решение

задачи автоматического эффективного распараллеливания, по нашему мнению, вообще нереально в обозримой перспективе.

Декларативный подход. Язык НОРМА

Один из возможных подходов к решению задачи автоматизации параллельного программирования вычислительных задач является подход с использованием декларативных (непроцедурных) языков. При использовании этого подхода решение вычислительной задачи программируется на непроцедурном языке (понятия, связанные с архитектурой параллельного компьютера, моделями параллелизма и т.п. при этом не используются), а затем компилятор автоматически строит параллельную программу (учитывая архитектуру целевого параллельного компьютера, модели параллелизма и т.п.). С учетом отмеченных выше проблем привлекательность этого подхода в настоящее время усиливается.

Идеи декларативного программирования были сформулированы еще в прошлом веке, теоретические исследования этого подхода для класса вычислительных задач проведены в пионерских работах И.Б. Задыхайло еще в 1963 году [8]. Непроцедурный язык НОРМА и система программирования НОРМА [9-11] разработаны в ИПМ им. М.В. Келдыша РАН также достаточно давно и предназначены для автоматизации решения вычислительных сеточных задач на параллельных компьютерах. Расчетные формулы записываются на языке НОРМА в математическом, привычном для прикладного специалиста виде, затем компилятор языка НОРМА генерирует параллельную программу на языках программирования Фортран или Си с использованием библиотеки передачи сообщений MPI.

Многолетний опыт использования системы программирования НОРМА при решении различных вычислительных задач [10,12,13] показал, по нашему мнению, целесообразность ее развития, как в направлении определения новых возможностей языка НОРМА, так и поддержки компиляции с учетом новых архитектурных возможностей параллельных систем. Отметим также, что декларативный подход также очень перспективен и для решения такой актуальной на сегодняшний день проблемы, как автоматизация программирования графических процессоров и автоматизация программирования прочих гибридных архитектур.

Параллельное программирование для архитектур с несколькими уровнями параллелизма требует совмещения особенностей нескольких моделей параллелизма. Все современные высокопроизводительные вычислительные системы состоят из множества соединённых друг с другом вычислительных узлов. Каждый узел имеет несколько многоядерных процессоров с общей памятью и, возможно, один или несколько графических процессоров. При создании эффективной программы для такой системы модель передачи сообщений должна быть согласована с моделью с общей памятью и, возможно, с моделью графического процессора. А с технической точки зрения это

означает необходимость использования библиотеки передачи сообщений MPI и стандарта OpenMP и, возможно, CUDA или OpenCL в рамках одной параллельной программы. Важно подчеркнуть, что это качественно более сложная проблема, не сводящаяся к механическому объединению способов программирования с использованием библиотеки MPI и стандарта OpenMP. Для её решения необходима разработка новых методов организации иерархии параллельных вычислений, балансировки вычислений, методов функциональной отладки и отладки эффективности и т.д. Без решения этих задач новые возможности вычислительной техники останутся не использованными.

В настоящее время в ИПМ им. М.В. Келдыша ведутся работы по созданию новой версии компилятора программ на языке НОРМА, в котором реализована поддержка всех новых возможностей языка НОРМА и который обладает модульной структурой, позволяющей создавать различные кодогенераторы выходных программ для различных параллельных архитектур и различных языков и инструментов параллельного программирования.

Компилятор языка НОРМА+

Постановка задачи

Перед разработчиками новой версии языка НОРМА, получившей название НОРМА+, и компилятора для нее была поставлена следующая задача: чтобы при полной поддержке совместимости программ, написанных на языке НОРМА, обеспечить в новом компиляторе следующие основные новые возможности:

- Поддержка в качестве целевой платформы исполняемой параллельной программы многоядерных вычислительных систем, распределённых вычислительных систем, систем с графическими процессорами, а также различных видов гибридных вычислительных систем. Таким образом в качестве целевой платформы можно будет использовать любую из широко распространённых сейчас архитектур параллельных вычислительных машин.
- Поддержка различных конфигураций распараллеливания данных, и, возможно, поддержка динамического конфигурирования распараллеливания данных при запуске программы.
- Поддержка динамического распределения памяти в исполняемой программе.
- Введение понятия «безопасного переприсваивания» в языке НОРМА+.
- Ведение глобальных описаний и глобальных переменных в язык НОРМА+.
- Ведение оператора INCLUDE в язык НОРМА+.

Структура компилятора

Для реализации поставленных выше задач разработана новая архитектура компилятора для языка НОРМА+, ориентированная на возможность генерации выходных параллельных программ для различных параллельных архитектур и целевых платформ. Разработка компилятора ведется на языке Си++, компилятор предназначен для применения как в среде UNIX, так и в среде Windows.

Общую схему работы компилятора можно описать следующим образом.

Компилятор программ, написанных на языке НОРМА+ (далее - компилятор НОРМА+), представляет собой приложение с интерфейсом командной строки. При вызове компилятора НОРМА+ ему через параметры командной строки передаются имена файлов, содержащие исходные тексты программы на языке НОРМА+, и различные опции компиляции. Компилятор НОРМА+ читает текст программ из указанных файлов, производит синтаксический и семантический анализ программ на языке НОРМА+ и выдаёт диагностику по каждому файлу о наличии или отсутствии ошибок в анализируемой программе. По каждой ошибке указывается тип обнаруженной ошибки и местоположение (строка и позиция) в исходном файле ошибочной конструкции.

В процессе синтаксического и семантического анализа программы на языке НОРМА+ компилятор строит Абстрактное Синтаксическое Дерево (АСД), представляющее анализируемую программу в виде структурированного дерева в памяти компилятора. Затем, если ошибок не найдено, компилятор выполняет процесс кодогенерации. В зависимости от заданных опций компилятор НОРМА+ генерирует исполняемую программу на одном из языков программирования низкого уровня (Си или Фортран) в одной из моделей параллельного программирования (MPI, OpenMP, CUDA или гибридная).

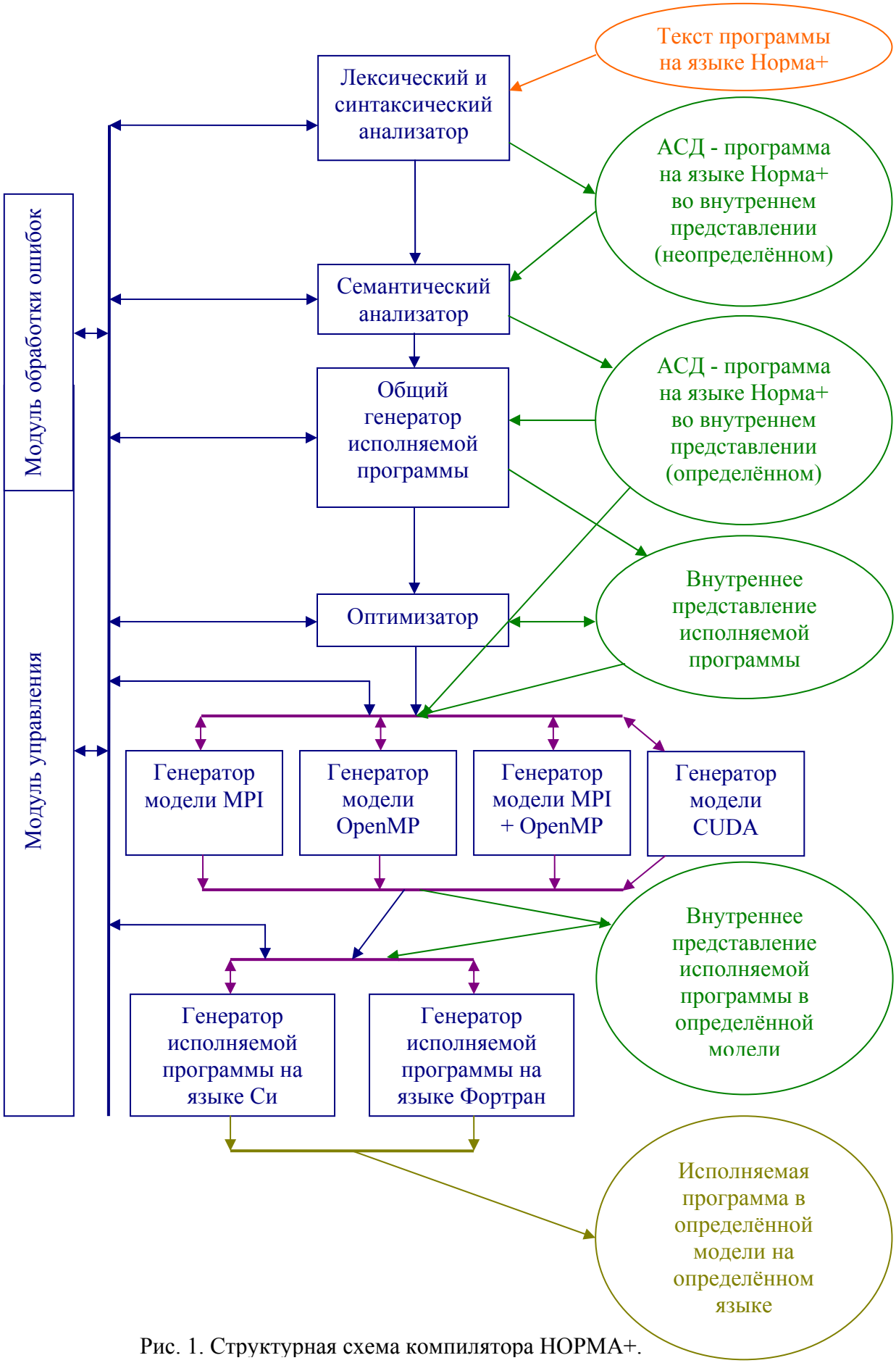

Рис. 1. Структурная схема компилятора НОРМА+.

Рассмотрим архитектуру компилятора НОРМА+, его составные части и их взаимодействие более подробно. На Рис. 1 приведена структурная схема компилятора НОРМА+. Модули компилятора показаны прямоугольниками. Взаимодействия между ними и передача потока управления показаны стрелками. В овалах на схеме показаны обрабатываемые структуры данных. Потоки данных между модулями компилятора и данными также показаны стрелками. В случае, когда потоки управления и потоки данных совпадают, для удобства изображения они показаны одной стрелкой.

В структуре компилятора НОРМА+ можно выделить два модуля, которые активны на протяжении всего рабочего цикла компилятора – от старта и разбора параметров командной строки до вывода сгенерированной программы в файл. Это модуль управления и модуль обработки ошибок (на схеме показаны вертикально слева). Эти два модуля компилятора взаимодействуют со всеми остальными модулями. Модуль управления анализирует параметры командной строки, а затем для каждого заданного файла с программой на языке НОРМА+ вызывает модули анализа и, в соответствии с заданными опциями, модули кодогенерации. В случае обнаружения каким-то модулем ошибок, как в синтаксисе или семантике программы на языке НОРМА+, так и в среде работы компилятора НОРМА+, эти ошибки предаются модулю обработки ошибок. Модуль управления после завершения каждого этапа рабочего цикла компилятора проверяет наличие ошибок, и, если они есть, модуль, отвечающий за следующий этап, уже не вызывается, а производится выдача диагностики об обнаруженных ошибках и процесс компиляции текущего файла прерывается.

Начинается процесс компиляции с того, что считанный файл с текстом программы на языке НОРМА+ передаётся модулю лексического и синтаксического анализатора. Модуль лексического и синтаксического анализатора осуществляет разбор текста, проверку правильности его синтаксиса и начинает строить Абстрактное Синтаксическое Дерево (АСД) программы на языке НОРМА+. АСД представляет анализируемую программу в виде структурированного дерева в памяти компилятора. Все обращения к АСД для создания, изменения и чтения узлов идут через специально созданный интерфейс набора классов на языке Си++. Во время синтаксического анализа некоторые типы узлов АСД носят неопределённый характер. Например, встретив при разборе выражения какой-то идентификатор, модуль лексического и синтаксического анализатора пока не может определить что это за идентификатор – переменная, константа или параметр области. Поэтому такой объект так и будет занесён в АСД как просто «идентификатор». Конкретизирован его тип будет на следующем этапе, модулем семантического анализа. В случае обнаружения какой-либо ошибки она рапортуется модулю обработки ошибок с указанием номера строки и колонки в исходном файле программы, а соответствующий узел АСД не строится. Затем модуль лексического и синтаксического анализатора пробует продолжить разбор

исходного текста с начала конструкции языка НОРМА+, следующей за конструкцией, содержащей ошибку. При отсутствии ошибок в программе в результате работы модуля лексического и синтаксического анализатора получается АСД с неопределёнными узлами.

АСД содержит полную информацию об исходной программе и передаётся дальше модулю семантического анализа, который конкретизирует типы всех неопределённых узлов, строит различные необходимые таблицы (переменных, областей, операторов и т.п.) и осуществляет семантические проверки исходной программы. В случае обнаружения какой-либо ошибки она также сообщается модулю обработки ошибок с указанием номера строки и колонки в исходном файле программы, а соответствующий узел АСД так и остаётся неопределённым. В случае же отсутствия ошибок, на выходе получается полностью определённый АСД, представляющий синтаксически и семантически корректную программу на языке НОРМА+, и набор различных служебных таблиц. На этом процесс анализа программы заканчивается.

Затем начинается процесс кодогенерации. Первым начинает работу модуль общего генератора исполняемой программы. Он порождает на основе исходной программы на языке НОРМА+ исполняемые операторы, определяет последовательность их выполнения, возможность параллельного исполнения и т.п. В результате модуль общего генератора исполняемой программы строит внутреннее представление исполняемой программы также в виде дерева в памяти компилятора, которое содержит конструкции общего типа, которые не зависят ни от выбранной модели параллелизма, ни от выходного языка. Все эти конструкции затем будут уточняться и дополняться другими модулями кодогенератора компилятора НОРМА+.

Затем, после того как общая структура исполняемой программы сгенерирована, вызывается модуль оптимизатора. Он оптимизирует исполняемую программу в целом и решает такие задачи, как например переопределение порядка вычисления операторов для оптимизации использования памяти.

Затем, в зависимости от заданных опций, вызывается кодогенератор для одной из поддерживаемых моделей параллелизма: MPI, OpenMP, CUDA или какой-либо гибридной. Заметим, что структурная схема компилятора НОРМА+ рассчитана на подключение любого количества различных модулей кодогенераторов моделей параллелизма, и, следовательно, возможности компилятора легко может быть расширены написанием нового модуля для новой модели. Модуль кодогенератора конкретной модели параллелизма производит действия по модификации исполняемой программы с учётом выбранной модели параллелизма. Например, в случае MPI производится распределение данных по процессорам, генерация операторов обмена данными и т.п. На выходе получается всё то же внутреннее представление исполняемой программы, но только конкретизированное для выбранной модели параллелизма.

Затем, в зависимости от заданных опций, вызывается кодогенератор одного из языков программирования (Си или Фортран), который переводит операторы внутреннего представления исполняемой программы в операторы выбранного языка программирования. Текст исполняемой программы в выбранной модели параллелилизма на выбранном языке программирования записывается в выходной файл и на этом процесс обработки файла заканчивается. Если в параметрах командной строки были заданы ещё другие файлы, то модуль управления запускает заново весь цикл компиляции для следующего файла. После компиляции последнего файла компилятор НОРМА+ завершает свою работу.

Классы, реализующие модули компилятора

Модуль управления

Модуль управления реализован функцией `main()`, которая начинает работу при запуске компилятора и которая является связующим звеном между всеми остальными модулями компилятора. Именно в ней установлено как, с какими параметрами, при каких условиях вызывать остальные модули компилятора. Класс `Ccontroller` также по большей части реализует модуль управления. Класс `Ccontroller` ответственен за разбор командной строки, хранение всех заданных опций, выдачу всех сообщений, в том числе и сообщений об ошибках. Объект класса `Ccontroller` существует в системе в единственном экземпляре и доступен всем остальным модулям как глобальный объект.

Модуль обработки ошибок

Модуль обработки ошибок реализован классом `CError` и методами класса `Ccontroller`, предназначенными для регистрации обнаруженной ошибки и выдачу отформатированного сообщения об обнаруженной ошибке пользователю. Все типы ошибок, диагностируемые компилятором, и их текстовые описания содержатся в классе `CError`.

Модуль лексического и синтаксического анализатора

К модулю лексического и синтаксического анализатора относятся классы `CSourceFile` и `CFileParser`. Класс `CSourceFile` предназначен для чтения файла с текстом программы на языке НОРМА+ и предоставления доступа к прочитанной информации. Класс `CFileParser`, используя класс `CSourceFile`, осуществляет разбор текста программы на языке НОРМА+, выполняет проверку правильности его синтаксиса и начинает строить АСД программы на языке НОРМА+. Если в тексте программы на языке НОРМА+ встречается директива INCLUDE, создаётся ещё по одному объекту классов `CSourceFile` и `CFileParser`, которые выполняют разбор включаемого файла. Следует отметить, что включаемый файл также может содержать в себе директиву INCLUDE, так что этот процесс имеет рекурсивный характер.

АСД

АСД представляет анализируемую программу в виде структурированного дерева в памяти компилятора. Все обращения к АСД для создания, изменения и чтения узлов идут через специально созданный интерфейс набора классов на языке Си++. В основе набора классов АСД лежит абстрактный класс `CNormaObject`, реализующий базовую функциональность для всех других классов АСДа. Все остальные классы прямо или через другие

классы наследуют `CNormaObject`. Иерархия классов приведена на Рис. 2а и 2б. Абстрактные классы, объекты которых существовать в АСД не могут, и которые были введены для описания общей сущности нескольких других классов, показаны в пунктирных прямоугольниках. Объекты остальных классов, показанных в сплошных прямоугольниках, могут присутствовать в АСД. Каждому такому объекту соответствует определённый элемент исходной программы на языке НОРМА+. На вершине АСД всегда находится объект типа `CProgramObject`, у которого могут быть несколько потомков типа `CPartObject, CFunctionObject, CDomainParameter` и т.д. `CPartObject` и `CFunctionObject` в свою очередь могут иметь потомков типа `COperatorAssume, CVariableObject, CIteration` и т.д.

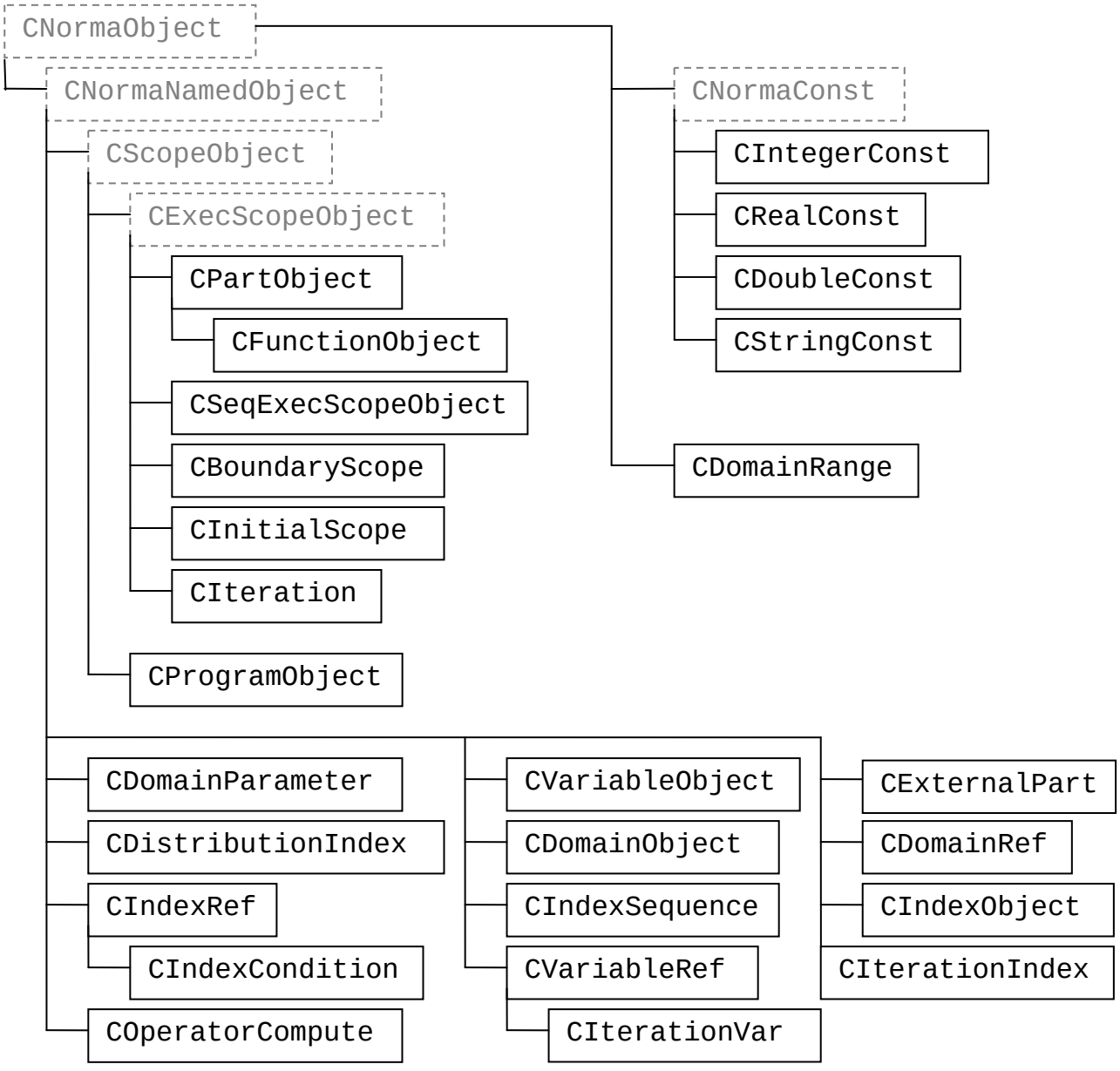

Рис. 2а. Иерархия классов АСД (начало).

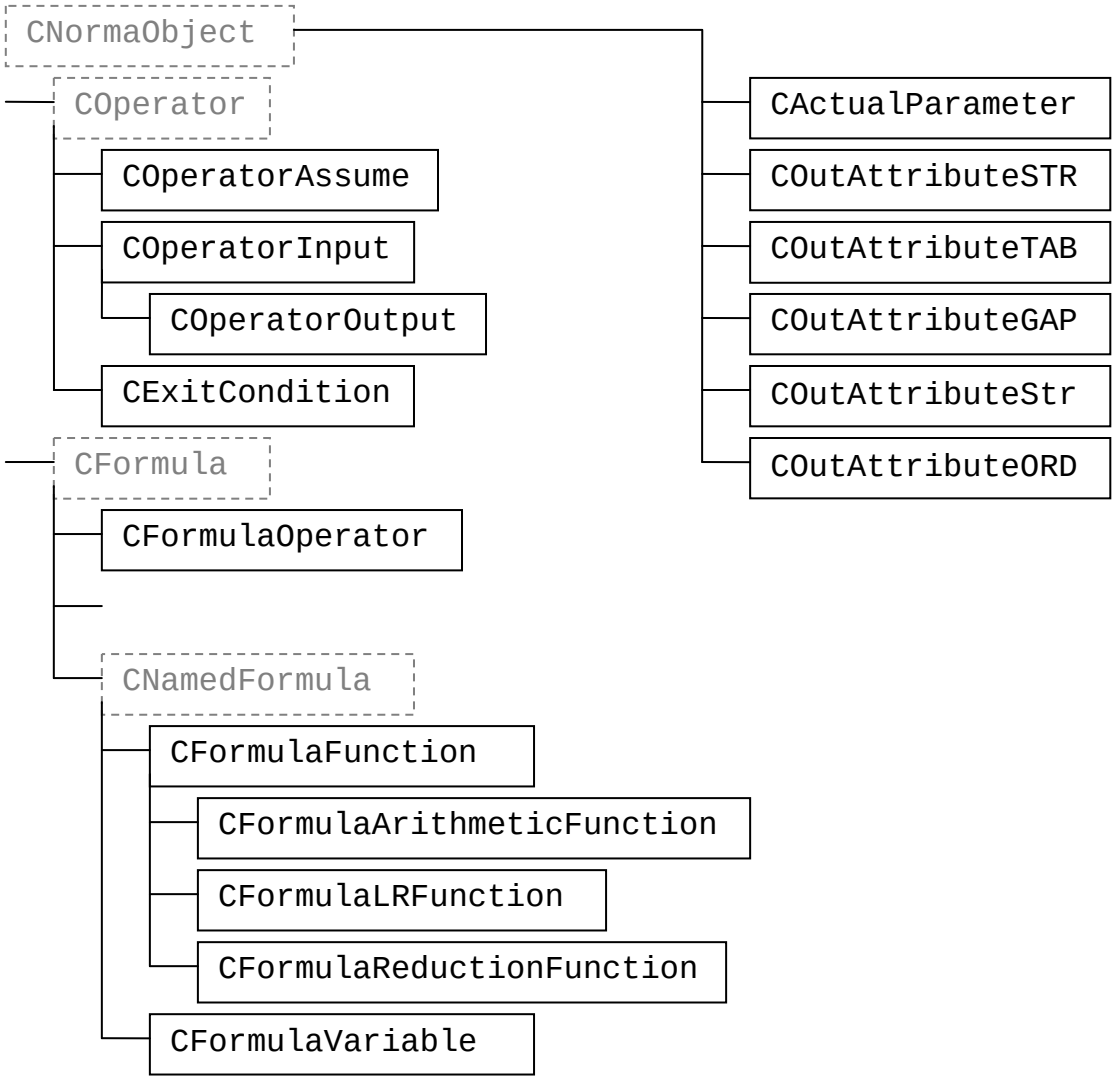

Рис. 2б. Иерархия классов АСД (окончание).

Модуль семантического анализа

Модуль семантического анализа реализован двумя виртуальными функциями класса `CNormaObject` и всех остальных классов АСД, наследованных от него. Это функция `ResolveNames()`, осуществляющая определение типов всех неопределённых узлов АСДа и строящая таблицы имён, и функция `Determine()`, выполняющая все необходимые проверки семантической корректности программы на языке НОРМА+ и строящая все прочие таблицы для кодогенератора. Эти функции определены во всех классах, где необходимо выполнить какие-либо действия по определению типов, семантическим проверкам или по накоплению различного вида информации для дальнейших действий. Будучи вызванными на каком-либо объекте, входящем в АСД, из этого объекта затем эти функции вызываются для всех потомков данного объекта. После построения АСД лексическим и

синтаксическим анализатором модуль управления вызывает последовательно функции `ResolveNames()` и `Determine()` для корневого элемента полученного АСД – объекта типа `CProgramObject`. Объект `CProgramObject` вызывает эти функции для всех своих потомков, те, в свою очередь – для всех своих. Таким образом осуществляется обход АСД, выполнение всех проверок и различного рода действий.

В результате работы модуля семантического анализа строятся таблицы переменных (класс `CVariablesTable`), индексов (`CIndexTable`), вызовов внешних разделов и функций (`CExternalPartsTable`). Для каждой переменной (класс `CVariableObject`) заполняется таблица: в каких операторах и на каких областях она требуется и в каких операторах и на каких областях она вычисляется, а для каждого оператора (класс `COperator`) заполняется обратная таблица: какие переменные и на каких областях в нём требуются и какие и на каких областях вычисляются.

Модуль общего генератора исполняемой программы

Модуль общего генератора исполняемой программы реализован классом `CGraphDependence` и методом `BuildGraphDependence()` класса `CProgramObject`. При вызове этого метода для корневого объекта АСД на основе АСД и созданных модулем семантического анализа таблиц определяется порядок выполнения операторов программы, и выделяются операторы, выполнение которых возможно параллельно.

Для этого строится так называемая параллельная ярусная схема выполняемой программы. Операторы, помещённые на один ярус этой схемы, могут выполняться в произвольном порядке и параллельно, так как они не зависят друг от друга. Операторы, помещённые на соседние ярусы, должны быть выполнены строго последовательно: сначала должны быть выполнены полностью все операторы предыдущего яруса, и только затем можно начинать выполнение операторов следующего яруса.

Параллельная ярусная схема программы является, фактически, представлением идеального (естественного) параллелизма, определяемого соотношениями и зависимостями между расчетными переменными программы.

Модуль кодогенератора для конкретной модели параллелизма

Модуль кодогенератора выходной программы для конкретной модели параллелизма реализован прежде всего набором классов, наследуемых от класса `OutPrgObject`. Этот класс реализует базовую функциональность для всех других классов кодогенератора. Все остальные классы прямо или через другие классы наследуют `OutPrgObject`. Иерархия классов приведена на Рис. 3

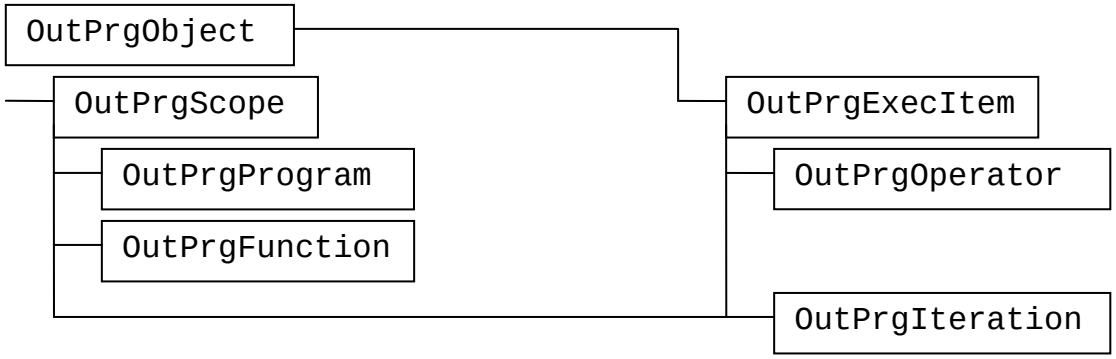

Рис. 3. Иерархия классов кодогенератора для конкретной модели параллелизма.

Данный набор классов реализует генерацию последовательной программы. Каждый из этих классов имеет виртуальную функцию `GenerateLIP()`, при вызове которой объект класса должен сгенерировать полностью код, относящийся к этому объекту. Таким образом, при вызове функции `GenerateLIP()` для объекта типа `OutPrgProgram` будет выполнена генерация всей выходной программы в конкретной модели параллелизма.

Каким образом выбирается конкретная модель параллелизма, будет объяснено чуть ниже, пока заметим, что последовательную программу, которая будет сгенерирована этим базовым набором классов, также можно считать программой «в конкретной модели параллелизма» - с отсутствующим параллелизмом.

Все классы данного набора имеют большое количество виртуальных функций, реализующих генерацию какого-либо примитива, например функция генерации начала цикла по области `GenerateLIPStartCyclesForDomain()`. И в функции `GenerateLIP()` прямо или через другие функции осуществляется обращение к этому множеству генераторов примитивов таким образом, что в результате получается генерация кода ко всему объекту.

Выбранный в компиляторе способ реализации кодогенераторов для различных моделей параллелизма достаточно прост: для создания конкретного кодогенератора (например, для модели параллелизма с общей памятью) достаточно создать набор классов, полностью соответствующий приведённому на Рис. 3 базовому набору классов и наследующий один в один этот набор. Например, класс `OpenMPProgram` наследует класс `OutPrgProgram`, класс `OpenMPOperator` – класс `OutPrgOperator` и т.д. И далее в этих классах конкретного кодогенератора необходимо переопределить функции генерации тех примитивов, которые в данной модели параллелизма должны быть реализованы иначе, чем в последовательной модели. А функции генерации тех примитивов, которые в данной модели параллелизма должны быть реализованы

так же, как и в последовательной модели, переопределять не нужно, и они должны вызываться из базового класса. Например, в кодогенераторе для модели параллелизма с общей памятью перед генерацией оператора необходимо проверить, есть ли в нём циклы, которые можно распараллелить. И, если есть, надо вставить соответствующие директивы, например OpenMP. А сам оператор генерируется точно так же, как и для последовательной программы – и это менять не нужно.

Модуль кодогенератора для конкретного языка программирования

Модуль кодогенератора для конкретного языка программирования реализован абстрактным классом `CodeGenerator`. Этот класс декларирует широкий набор виртуальных функций, предназначенный для генерации на каком-то языке программирования таких примитивов, как «объявить переменную», «создать условный оператор», «начать цикл по индексу» и т.п. Этот набор достаточен для того, чтобы кодогенератор для конкретной модели параллелизма создавал выходную программу, оперируя этими примитивами, и, таким образом, был бы языково независим. Затем, при создании кодогенератора для конкретного языка программирования (например, Фортрана) должен быть создан класс, наследующий класс `CodeGenerator` и реализующий все эти примитивы для этого конкретного языка программирования.

Заключение.

В настоящее время завершается реализация версии компилятора с языка НОРМА+ для модели параллелизма с общей памятью, выходные программы строятся на Фортране или Си с использованием технологии OpenMP. В дальнейшем планируется применение описанной выше схемы компиляции для разработки версии компилятора с языка НОРМА+ для графических процессоров с использованием технологии CUDA.

ЛИТЕРАТУРА:

1. http://chapel.cray.com/
2. http://x10-lang.org/home.html
3. http://labs.oracle.com/projects/plrg/PLDITutorialSlides9Jun2006.pdf
4. http://upc.lbl.gov/
5. R.W. Numrich, J.K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 1998, 17(2), pp 1-31.
6. http://charm.cs.uiuc.edu/
7. http://colamo.parallel.ru/
8. И.Б. Задыхайло. Организация циклического процесса счета по параметрической записи специального вида. Журн. выч. мат. и мат. физ., т.3, N2, 1963, с.337-357.
9. Андрианов А.Н., Бугеря А.Б., Ефимкин К.Н., Задыхайло И.Б. НОРМА. Описание языка. Рабочий стандарт. — М.: Препринт ИПМ им.М.В.Келдыша РАН, 1995, № 120, 52с.
10. Андрианов А.Н., Бугеря А.Б., Ефимкин К.Н., Колударов П.И. Система параллельного программирования НОРМА – подход, результаты разработки, применение. В сборнике Труды X Байкальской Всероссийской конференции “Информационные и математические технологии в науке, технике и образовании”. Часть 1. Иркутск, 2005, 5с.
11. Система НОРМА. http://www.keldysh.ru/pages/norma/
12. Andrianov A.N., Bazarov S.B., Bugerya A.B., Efimkin K.N. Solution of three-dimensional problems of gas dynamics on multiprocessor computers. Computational mathematics and modeling, 1999, v.10, N2.
13. Andrianov A.N., Efimkin K.N., V.Y. Levashov, I.N. Shishkova. The Norma language application to solution of strong nonequilibrium transfer process problem with condensation of mixtures on the multiprocessor system. Lecture Notes in Computer Science, 2001, v.2073.