УДК 519.6

СИСТЕМА АНАЛИЗА ИНФОРМАЦИОННОЙ СТРУКТУРЫ ПРОГРАММ

К. С. Стефанов[1]

В статье описывается автономная система анализа структуры программ. Данная система ориентирована на адаптацию последовательных программ для исполнения на параллельных вычислительных системах.

Ключевые слова: параллельные компьютеры, эффективность программ, анализ программ, информационная структура программ, эквивалентные преобразования программ, архитектура вычислительных систем.

1. Введение. В последнее время численные методы получают все более широкое распространение в различных областях. Появляются новые задачи и развиваются существующие методы. При этом требуются все большие вычислительные мощности. Однако традиционные последовательные системы не могут обеспечить нужной производительности в связи с рядом ограничений. Необходимые ресурсы могут быть созданы только путем построения параллельных вычислительных систем, которые могут давать практически неограниченную производительность. Одной из важнейших задач становится разработка программного обеспечения, максимально учитывающего особенности конкретной параллельной системы. Здесь мы имеем общую трудность при работе как с последовательными, так и с параллельными системами — успешность использования вычислительной техники для решения поставленных задач во многом определяется качеством программного обеспечения. Но создание эффективных программ для параллельных систем является гораздо более трудной задачей, чем для последовательных.

На заре развития ЭВМ также стояли проблемы создания эффективного программного обеспечения. Появлялись машины разнообразной архитектуры и с непохожими способами программирования. Но эти проблемы довольно быстро разрешились за счет, во-первых, использования внешней унификации модели архитектуры вычислительных систем — архитектуры фон Неймана — и, во-вторых, появления языков программирования высокого уровня, которые позволяли абстрагироваться от особенностей низкоуровневого программирования.

Принципы фон Неймана позволили пользователям не обращать внимания на особенности реализации общей модели в конкретной архитектуре используемой системы. Достаточно было знать общие принципы, а как они конкретно реализовывались в данной системе — не так важно. Для пользователя все системы выглядели, в основном, одинаково, если не опускаться на уровень системного программирования, где необходимо знать подробности устройства конкретных машин.

Именно языки программирования стали на долгое время ключом к эффективному использованию вычислительной техники. Во-первых, они давали пользователям возможность оперировать более привычными понятиями (переменная, повторение, вычисление по формуле), а не машинно-зависимыми языками, с которыми приходилось сталкиваться людям, писавшим программы для первых компьютеров. Во-вторых, языки программирования стали унифицированным средством для обмена программами и алгоритмами. Пользователь, написавший программу на языке ФОРТРАН, мог быть уверен, что сможет выполнить ее на любой системе, где есть компилятор этого языка. Также имелась гарантия, что с точностью до ошибок округления результаты, полученные на разных системах, будут совпадать.

При этом всю работу по максимально эффективному исполнению данного алгоритма на данной системе брали на себя компиляторы. Основной задачей компиляторов, кончено же, был перевод программы с входного языка в исполняемый код целевой архитектуры. В процессе компиляции производился также ряд оптимизаций, часть из которых диктовалась особенностями конкретной архитектуры (например, размещение наиболее часто используемых переменных в самую быструю память, к примеру, в регистры), а часть была общей, пригодной для любых систем (например, вынос инвариантов из цикла).

С течением времени компиляторы совершенствовались. И хотя генерируемый ими код иногда мог быть улучшен при ручной оптимизации, выигрыш в производительности не окупал затрат на такую работу, тем более что получаемый результат в этом случае был пригоден лишь для какой-то одной системы.

[1] Научно-исследовательский вычислительный центр, Московский государственный университет им. М. В. Ломоносова, 119992, Москва; e-mail: cstef@parallel.ru

Ситуация стала меняться с появлением параллельных систем. Одними из первых были векторно-конвейерные. При создании компиляторов для них стали возникать трудности, которые потом в полной мере проявились и на других параллельных системах. Основное, за счет чего мог проявиться выигрыш при использовании векторных компьютеров, — это наличие встроенных векторных команд, манипулирующих сразу большим массивом данных. Но далеко не во всех языках программирования в то время были конструкции, непосредственно переводящиеся в векторные команды. Возникла проблема, какие конструкции языка высокого уровня переводить в векторные команды, а какие компилировать с использованием обычных команд, которые векторные системы унаследовали от своих последовательных предшественников. Наиболее подходящая кандидатура для преобразования в эффективные векторные команды — самые внутренние по вложенности циклы. Однако далеко не всякий цикл может быть преобразован в векторную команду. Появилось множество способов определения того, можно ли данный цикл представить как векторную команду или нет. Надо заметить, что универсального способа определения векторизуемости, пригодного для любых циклов, нет до сих пор.

Еще сильнее ситуация усложнилась с появлением других типов параллельных архитектур — архитектур с общей и распределенной памятью и многомашинных комплексов. Для архитектур с общей памятью необходимо определять, возможно ли разбиение программы на несколько потоков вычислений, выполняющихся параллельно, и находить точки синхронизации между отдельными потоками. Для архитектур с распределенной памятью, кроме того, необходимо устанавливать, какая часть данных требуется каждому потоку. Также необходима минимизация обменов данными между разными процессорами.

Попытка создать компиляторы, автоматически извлекающие необходимые сведения из обычной программы и использующие их для перевода ее в код параллельной вычислительной системы, оказалась не очень успешной, так как стало понятно, что извлечение этих сведений крайне сложно и не существует достаточно универсального способа, пригодного для использования на широком множестве программ и широком классе систем. В то же время в компиляторы, особенно для векторных систем, продолжали (и продолжают) включаться некие достаточные условия, на основе которых компилятор в состоянии в некоторых случаях обеспечить эффективное исполнение фрагментов входной программы. Никаких гарантий эффективности применяемых признаков производители компиляторов не дают. И остается только гадать, то ли данная программа действительно не может быть выполнена на данной системе с данной производительностью, то ли компилятор не справился с ее оптимизацией.

Сложилась следующая ситуация. Теперь уже недостаточно взять хороший алгоритм и грамотно его перевести на язык программирования. Скорость работы стала во многом зависеть от компилятора для параллельной системы. Конечно, такая зависимость существовала и на последовательных системах. Но если раньше неуспех в применении какой-либо оптимизации влиял на производительность получаемого кода не очень сильно, то теперь от удачи оптимизации стало зависеть, имеет ли смысл вообще применять параллельную систему. Если компилятор справится с оптимизацией входной программы, то параллельные возможности системы используются, а если не справится, то нет смысла применять параллельную систему. Естественно, успех или неудача оптимизации рассматривается не для программы целиком, а для отдельных ее фрагментов, но это не снимает основного вопроса: “Насколько эффективно используются возможности параллельной системы?” Производители компиляторов обычно не дают никаких гарантий, что данный компилятор обеспечивает требуемую эффективность. А поскольку информации об используемых в конкретном компиляторе схемах оптимизации немного, то может случиться, что две программы, одна из которых является очень простой модификацией другой и реализует тот же алгоритм, выполняется в несколько раз быстрее исходной. В работе [1] проанализированы методы выявления параллелизма, заложенные в компиляторы для параллельных систем начала 90-х годов прошлого века. Оказалось, что сфера применения каждого из методов очень узка. В целом же при анализе зависимостей различных операторов компиляторам удается лишь в 15 % случаев без трудностей определить характеристики этих зависимостей. Несмотря на то, что за прошедшие 10 лет появилось много новых способов распараллеливания вычислений, в общем ситуация изменилась не сильно.

В результате забота о распараллеливании программ была перенесена с компилятора на пользователя. Стали активно развиваться системы и языки параллельного программирования (MPI, OpenMP, HPF, DVM, PVM и др.). Все они требуют от пользователя данных о параллельных свойствах программы. Часть из них представляют примитивы параллельного программирования (создание-уничтожение задач, коммуникационные примитивы, примитивы синхронизации), используя которые можно непосредственно задавать способ распараллеливания программы. Другие предоставляют возможности описания параллельных свойств программы, оставляя выбор способа их реализации исполняющей системе. При этом встает проблема, о которой уже упоминалось в связи с компиляторами для параллельных систем, —

достаточность предоставляемой информации, а также эффективность и полнота ее использования. Ведь для разных архитектур необходимы разные сведения, и информации, достаточной для качественного распараллеливания программы на векторной системе, может оказаться недостаточно для эффективного ее использования на массивно-параллельной системе. И наоборот, информация, необходимая для адаптации программы под кластерную систему, может оказаться бесполезной при адаптации под векторную.

Возникает вопрос о том, откуда брать эту информацию. Ведь, как уже отмечалось, даже самая простая задача — возможно ли параллельное исполнение данного цикла — на настоящий момент не решена в общем случае. И уж тем более нет универсальных способов для распараллеливания программ под распределенные системы. Хорошо, если создается новая программа и пользователь имеет возможность заранее заложить в нее параллельные свойства либо выбором применяемых алгоритмов, либо из каких-то других соображений, например, если свойства параллельности вытекают из характеристик моделируемых явлений. Но даже это является очень трудной задачей. Кроме того, существует огромное множество накопленных программ, библиотек, пакетов, которые также хотелось бы использовать на параллельных системах, реализуя при этом все их возможности. Ручной анализ программ — занятие крайне трудоемкое. Терять же весь накопленный багаж или переделывать его заново не выгодно.

В связи с этим стали создаваться автономные системы анализа, позволяющие по тексту программы определять некоторые ее свойства, которые впоследствии могут быть использованы при адаптации программы под параллельную систему. Раньше (в случае последовательных систем) процесс применения вычислительной техники для расчетов выглядел так: выбор алгоритма – запись его на языке программирования – компиляция – исполнение – использование полученных результатов. Теперь же добавляется адаптация к конкретной параллельной системе.

При этом программа начинает существовать в нескольких вариантах. Программа, полученная непосредственно после записи выбранного алгоритма на языке программирования, наиболее точно отражает этот алгоритм, но ее трудно эффективно использовать на параллельных системах — в ней не хватает необходимых для этого данных. Если же взять программу, уже адаптированную под параллельную систему, особенно если эта адаптация выполнена при помощи системы параллельного программирования, предоставляющей соответствующие примитивы, то сам применяемый алгоритм будет в большой степени скрыт использованием конструкций параллельного программирования. Такая программа сделана уже под конкретную архитектуру, и для использования ее на другой системе необходимо сначала очистить ее от лишних конструкций, получить исходный алгоритм и лишь потом снова выполнить адаптацию под новую архитектуру. Иными словами, для переноса программы уже мало собственно текста программы, которую непосредственно компилировали, желательно иметь текст программы до адаптации, так как в нем четче отражена структура реализованного алгоритма.

Если программа пишется заново, то процесс записи алгоритма на языке программирования может совмещаться с адаптацией его под параллельную систему. Но в этом случае мы получаем программу, написанную под одну конкретную систему, что сильно сужает круг ее применения. Если же при написании программы используются уже существующие библиотеки, еще не адаптированные под используемую параллельную систему, либо же хочется создать программу, пригодную для широкого круга систем, то приходится прибегать к системам анализа программ для выявления свойств, которые могут понадобиться при адаптации программ под параллельные вычислительные системы.

Эти системы анализа играют очень важную роль. Мы по-прежнему можем обработать последовательную программу компилятором и запустить ее на параллельной системе. Для некоторых архитектур компилятор произведет какую-то адаптацию программы под целевую систему и мы можем получить какой-то выигрыш в производительности. Будет ли полученная в итоге производительность достигать максимума того, что можно получить от данной системы при использовании данной программы, неизвестно. Но для большого класса систем (в основном это распределенные системы) компилятор при обработке последовательной программы не будет производить никаких действий по ее распараллеливанию. Таким образом, мы не получим никаких преимуществ от использования параллельной системы. Для того чтобы компилятор мог сгенерировать из программы код, использующий возможности параллельной системы, необходимо предоставить ему дополнительную информацию, извлечь которую и призваны системы анализа программ.

Зачастую эти системы построены по уже упоминавшемуся принципу. Есть некий набор признаков параллельности тех или иных конструкций. Если условия признака выполняются, отмечается наличие свойства. Если нет, пытаемся применить другой признак. Данный подход плох тем, что в случае, когда не применим ни один признак, непонятно, то ли программа не обладает необходимыми свойствами, то ли просто среди набора признаков нет того, который мог бы это свойство выявить.

С другой стороны, сложность построения систем анализа заключается в том, что неизвестен набор характеристик программы, зная которые можно указать метод адаптации данной программы под любую параллельную архитектуру. Существует несколько основных типов параллельных архитектур. Свойства, которые надо учитывать при адаптации программ под эти архитектуры, сильно различаются. Например, для векторных архитектур предпочтительным является возможность параллельного исполнения самых внутренних из вложенных циклов. При этом чем длиннее эти циклы, тем эффективнее используются векторные устройства машины. Для многопроцессорных систем, наоборот, предпочтительным является возможность параллельного исполнения как можно более "тяжелых" по вычислениям частей программы, с целью разнесения их на различные процессоры для независимого выполнения. В таком случае, чем больше эти части, тем реже необходима синхронизация и тем меньше будут простаивать процессоры. Если же у различных процессоров имеется еще и различная память, то необходимо принимать во внимание также и то, какие данные использует каждая часть кода и распределять их так, чтобы все необходимое для работы одного процессора было в памяти этого процессора, а пересылки между процессорами были бы минимальны. Нередко также встречаются смешанные типы архитектур, например, многопроцессорная система с общей памятью, где каждый процессор — векторный; или же кластерная система, каждый элемент которой — многопроцессорная система с общей памятью. При адаптации под такие системы приходится учитывать набор из свойств программ, применимых как при работе с одним типом архитектур, так и с другим. Иногда эти свойства могут противоречить друг другу, и приходится балансировать между использованием особенностей как той, так и другой архитектуры.

Кроме того, постоянно появляются новые типы архитектур. Чаще всего они представляют собой синтез каких-нибудь других типов архитектур. Но при этом часто бывает, что особенности новой системы таковы, что непосредственное использование уже существующих методов распараллеливания невозможно, и приходится либо придумывать новые методы, либо "подгонять" уже известные методы к особенностям новой архитектуры. В последнее время много говорят о квантовых компьютерах. Пока существуют в основном модели таких компьютеров и лишь единичные опытные образцы. Понятно, что использование этих компьютеров потребует очень сильной переделки алгоритмического багажа. Но абсолютно неясно ни то, каким будет интерфейс этих систем, ни то, какие требования будут предъявляться к программам для них. Тем не менее, адаптировать уже существующие программы и алгоритмы придется в любом случае, так как невозможность использования уже наработанного багажа может привести к тому, что внедрение новых систем сильно затянется. Поэтому крайне важно уметь определять как можно более разнообразные характеристики уже существующих программ в надежде, что эти характеристики, возможно, окажутся полезными и при переходе на принципиально новые вычислительные системы.

Итак, определение характеристик программ, которые полезны для адаптации под параллельные системы, является крайне сложной задачей. Она становится еще труднее оттого, что невозможно сказать, какие конкретно характеристики могут понадобиться для уже существующих архитектур и, тем более, могут оказаться полезными в будущем. Поэтому представляется крайне важным создание систем анализа программ, которые по тексту программы могут определять необходимые характеристики. Поскольку набор таких характеристик до настоящего времени не определен, необходимо выявление как можно более широкого множества характеристик, чтобы человек, занимающийся адаптацией программы под параллельную систему, мог решать, какие из них ему следует использовать. При этом желательно, чтобы класс анализируемых программ также был как можно шире, ибо неизвестно, с какими программами придется столкнуться в работе.

Построение таких систем только на основе набора неких достаточных условий не представляется приемлемым. В самом деле, после анализа с использованием таких условий мы имеем набор неких характеристик программы. Но выявили ли мы все, что можно использовать для распараллеливания данной программы? Достигнем ли мы после использования выявленных свойств максимальной для данной программы на данной архитектуре производительности, или это не предел? Если же использовать условия необходимые и достаточные, то можно точно сказать, обладает ли исследуемая программа данным свойством. Если да, то мы используем его для распараллеливания. Если же нет, то тем самым устанавливается невозможность никаким другим методом прийти к данному свойству, что дает возможность ответить на вопрос, достигнут ли предел увеличения производительности.

В настоящее время наметилась тенденция разделения функций программного обеспечения для параллельных систем. С одной стороны, в языки программирования встраиваются способы описания параллелизма программ. Правда, после этого языки параллельного программирования приобретают черты языков низкого уровня. Например, такие языки как PVM и MPI по существу являются коммуникационными автокодами, хотя и на макроуровне. Программы, написанные на таких языках, становятся за-

висящими от архитектуры конкретных вычислительных систем. На пользователя все в большей степени перекладывается забота об эффективности организации вычислительных процессов, включая обнаружение в программах и алгоритмах свойств параллельности и коммуникационных свойств, необходимых для адаптации под конкретную систему.

С другой стороны, постоянная нехватка должной поддержки со стороны языков программирования, компиляторов и операционных систем обеспечения эффективности процессов решения задач привела к созданию специализированных программных комплексов по анализу пользовательских программ и их преобразованию под требования конкретных параллельных языков программирования и собственно параллельных вычислительных систем. Эти комплексы реализуются на автономных компьютерах и не связаны с компиляторами целевых вычислительных систем. Поэтому на них могут быть реализованы самые передовые методы исследования и преобразования программ. Представляя собою автономные программные системы, эти комплексы оказались удобным инструментом для выполнения различных работ, когда программы одного вида нужно перевести в эквивалентные программы другого вида. Среди зарубежных автономных систем наиболее известны система Parafrase и созданная на ее основе серия пакетов KAP [2–4], а также система FORGE [5]. Отечественный представитель данного класса систем — система V-Ray [7, 8] — средство анализа и визуализации структуры последовательных программ.

Более подробно вопросы эффективного использования параллельных систем и анализа программ освещены в работе [8].

2. Основные принципы проектирования системы. Основная область применения системы — адаптация уже существующих последовательных программ, выполняющих большой объем вычислений, для исполнения на параллельных вычислительных системах. Кроме того, система может применяться для определения возможностей дальнейшего распараллеливания программ или использования их на другой платформе.

Поскольку бóльшая часть вычислительных программ написана с использованием языка ФОРТРАН, основной упор при создании системы делался на анализ программ именно на этом языке. Возможен анализ программ и на языке Си. Однако некоторые режимы работы для программ на языке Си не реализованы в связи с отличиями в языках и трудностью с использованием базовой теории анализа программ [6].

За основу для проектирования архитектуры системы взята архитектура системы V-Ray, которая является инструментальным средством анализа и визуализации структуры программ на языке ФОРТРАН. В то же время были сделаны изменения, отражающие ориентацию на исследование программ на двух языках программирования.

3. Основные принципы построения алгоритмов интерактивной визуализации. Мы не случайно выделяем визуальный анализ в качестве самостоятельной компоненты технологии исследования структуры программ. Практика показывает, что если что-то не срабатывает в автоматическом режиме исследования, то необходимы развитые средства визуализации, облегчающие восприятие структуры как программы в целом, так и отдельных ее составных частей. Это особенно важно в том случае, когда приходится иметь дело с заранее неизвестной программой, автор которой недоступен, но выполнить анализ и перенести программу на параллельную платформу все же необходимо. Часто визуализация сразу помогает найти правильное решение, так как некоторые понятия, трудно поддающиеся формализации, легко воспринимаются визуально.

Основной принцип, который мы закладываем в инструментальные средства визуального анализа структуры программ, состоит в том, что все основные компоненты программы должны быть показаны в наиболее наглядной форме. Последний тезис, конечно же, можно оспаривать в силу большой субъективности “степени наглядности”. Мы предоставляем пользователю самому выбрать из нескольких возможных вариантов компоновки изображения наиболее ему подходящий.

Были реализованы различные способы визуализации, включая способы отображения графов и способы разметки вершин графов.

Способы отображения графов не являются универсальными, и для каждого графа реализован свой набор способов отображения, который способен отразить особенности различных графов именно данного типа. Например, граф использования общей памяти является двудольным и его отображение должно учитывать этот факт. Графы, в качестве вершин которых выступают операторы или группы операторов одной функции, должны отображать относительное расположение операторов. Это позволит легче сопоставить вершины графа операторам исследуемой функции. Естественно, для каждой вершины предусмотрен вывод текста соответствующего ей оператора. Порядок расположения вершин графа, соотнесенный с порядком следования операторов по тексту программы, существенно облегчает понимание структуры программы.

Та же ситуация с разметкой вершин графов — каждый способ разметки связан со своим типом графов. Например, разметка циклов в соответствии с наличием у них свойства ParDo просто не имеет смысла для графа вызовов. А пометка процедур, текст которых недоступен или которые ниоткуда не вызываются, не имеет смысла для графа управления. В то же время разметка согласно глубине вложенных циклических конструкций имеет смысл как для графа вызовов, так и для графа управления процедуры, представленного в форме иерархического дерева. Для графа вызовов цвет вершины соответствует максимальной глубине вложенности всех циклов, содержащихся в процедуре. Для иерархического дерева глубина вложенности циклической конструкции, соответствующей вершине, написана на самой вершине.

Примеры реализованных способов отображения графов и разметки их вершин будут приведены ниже.

4. Ввод анализируемой программы. Система может анализировать как одиночные файлы, так и проекты, состоящие из многих файлов. Начальный диалог системы представлен на рис. 1. После выбора языка анализируемой программы можно загрузить одиночный файл кнопкой “Load”, либо создать проект из многих файлов кнопкой “Create”.

После этого осуществляется перевод во внутреннее представление. Все ошибки, выявленные на данном этапе, сообщаются пользователю (рис. 2).

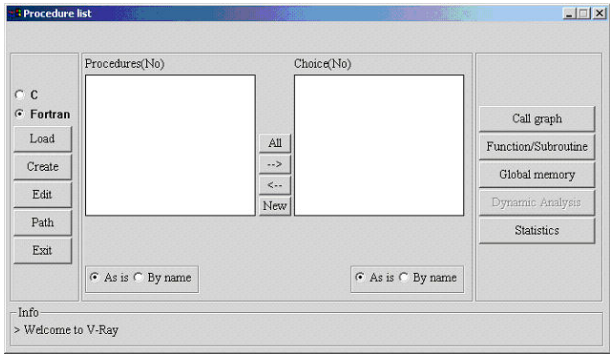

Рис. 1. Диалог выбора файлов с исходным текстом анализируемой программы

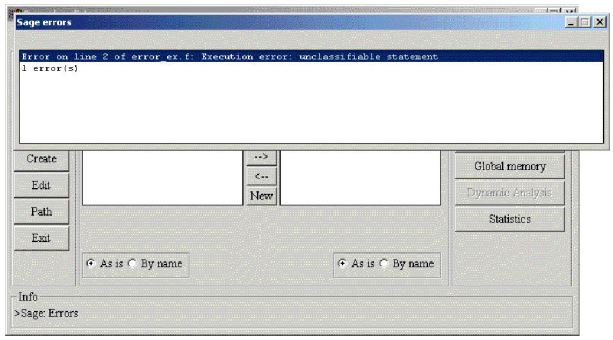

Рис. 2. Окно с сообщением об ошибке в исходном тексте

5. Анализ управляющей и модульной структуры программы

5.1. Граф вызовов процедур. Каждая вершина графа вызовов соответствует программной единице (процедуре или функции) исследуемой программы. В графе присутствует дуга, идущая из вершины A в вершину B, если подпрограмма B вызывается из подпрограммы A.

Разработаны различные способы изображения графа вызовов. Это необходимо для того, чтобы пользователь мог выбрать тот из них, который максимально учитывает особенности исследуемого проекта в части формы графа вызовов. Граф вызовов может быть близок к дереву или может иметь множество пересекающихся путей.

На рис. 3–7 представлены различные способы отображения графа вызовов для одной и той же программы. На рис. 3 представлен граф вызовов лишь части процедур, так как полный граф давал бы слишком мелкое изображение, в то же время на данном рисунке можно увидеть характерные особенности этого способа изображения.

Поддерживается построение графа вызовов с включением части процедур программы. Выбор включаемых процедур может проводиться как вручную (непосредственным указанием нужных), так и автоматически по определенным признакам. Можно включить встроенные процедуры языка (рис. 8), а также отобразить лишь те процедуры, которые вызываются из данной, или те процедуры, которые вызывают данную.

5.2. Способы разметки графа вызовов. Реализованы различные способы разметки графа вызовов. Вершины графа изображаются либо различной формы, либо различных цветов в зависимости от наличия у них некоторого признака.

На рис. 9 изображен граф вызовов, размеченный согласно глубине вложенности циклических конструкций. Различным цветом выделены процедуры, имеющие различную глубину вложенности входящих в них циклов. Позволяет определить наиболее вероятные цели для дальнейшего более глубокого исследования.

Пометка процедур, не вызываемых ни из каких других, предназначена для быстрого определения головной процедуры программы или тех процедур библиотеки, которые предназначены для вызова непо-

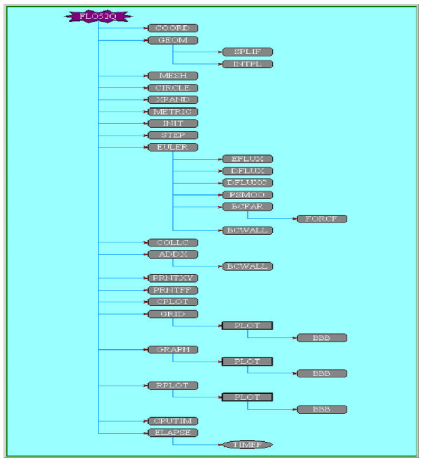

Рис. 3. Изображение графа вызовов в виде дерева

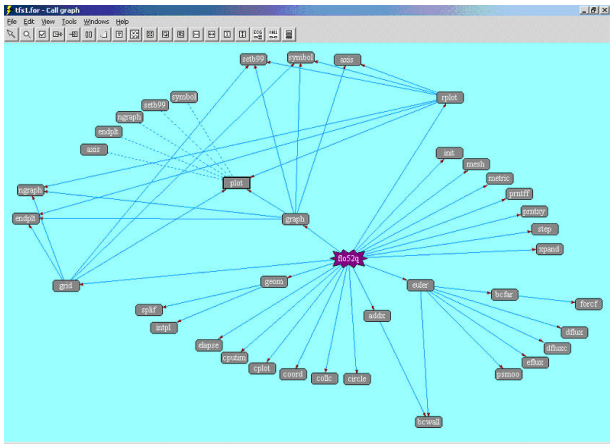

Рис. 4. Изображение графа вызовов с использованием орбитального алгоритма

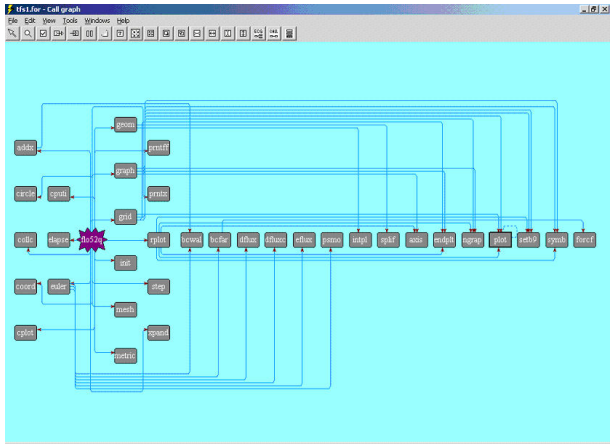

Рис. 5. Отображение графа вызовов методом “Regular”

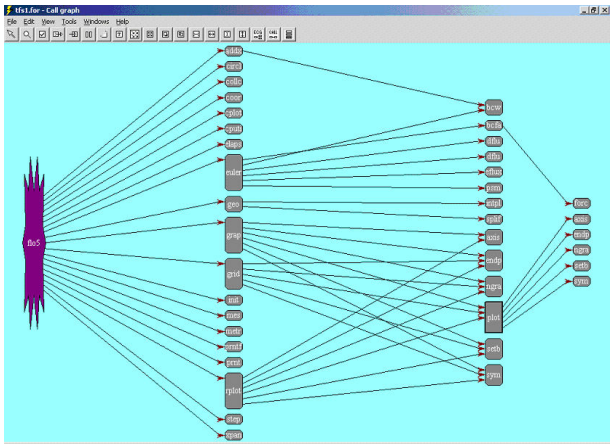

Рис. 6. Изображение графа вызовов в виде обобщенной ярусно-параллельной формы

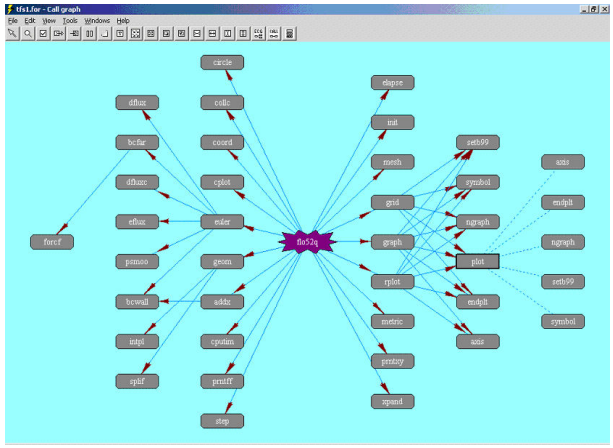

Рис. 7. Изображение графа вызовов методом “Graph”

средственно пользователем (в отличие от процедур, используемых другими процедурами библиотеки). Также помогает выявить неиспользуемые части кода — процедур, ранее вставленных в проект, которые перестали использоваться, но не были исключены из него.

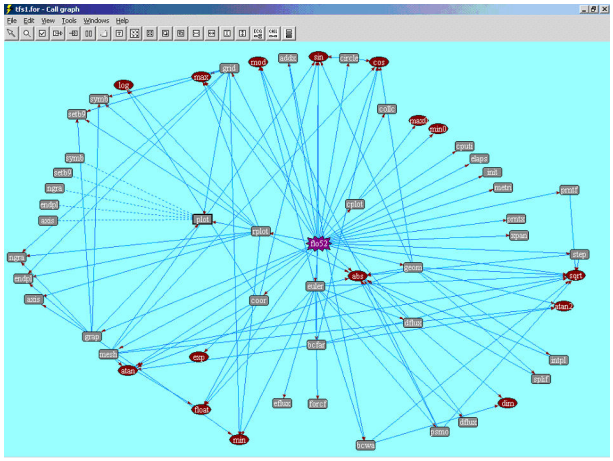

Рис. 8. Изображение графа вызовов с показанными встроенными процедурами и альтернативными точками входа в процедуры

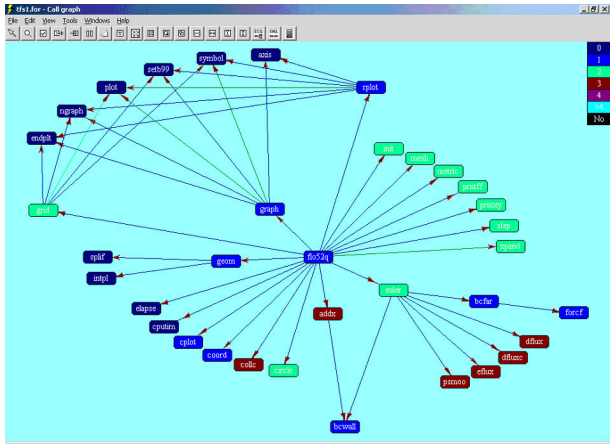

Рис. 9. Граф вызовов, размеченный по глубине вложенности циклических конструкций

Пометка процедур, текст которых недоступен, позволяет определить, насколько осуществим полный анализ проекта.

5.3. Циклический профиль путей графа вызовов. Циклический профиль путей графа вызовов — конструкция, в которой отражена последовательность вложенных циклов для каждого пути графа вызовов. Позволяет оценить суммарную вложенность циклов, из которых происходит вызов подпрограмм, начиная с тех, которые не вызываются ни из каких других. Таким образом, можно выявить подпрограммы, оптимизация которых может с большой вероятностью дать наибольший прирост производительности. Циклический профиль показан на рис. 10. На графе вызовов выделен путь, соответствующий выбранному пути в окне профиля.

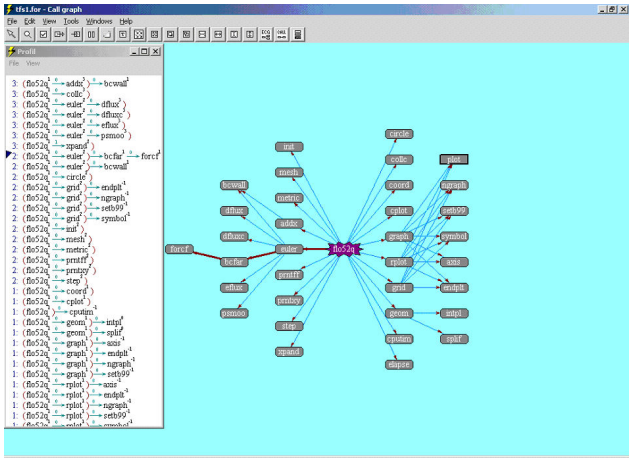

Рис. 10. Циклический профиль путей графа вызовов

5.4. Расширенный граф вызовов с циклическим профилем. Расширенный граф вызовов с циклическим профилем — граф вызовов, в котором в каждую вершину, соответствующую подпрограмме, вписан ее циклический профиль. Поддерживается разметка циклов в соответствии с наличием у них свойства ParDo. Пример расширенного графа вызовов с циклическим профилем приведен на рис. 11. Циклы, отображенные красным цветом, имеют тип ParDo (могут быть исполнены в параллельном режиме). Для каждой процедуры можно выбирать, какие циклы отображать — все или только те, из которых вызываются какие-нибудь другие процедуры. С целью уменьшения размера рисунка для части процедур убран циклический профиль.

6. Анализ управляющей структуры процедуры

6.1. Граф управления процедуры. Граф управления процедуры — граф, вершины которого пред-

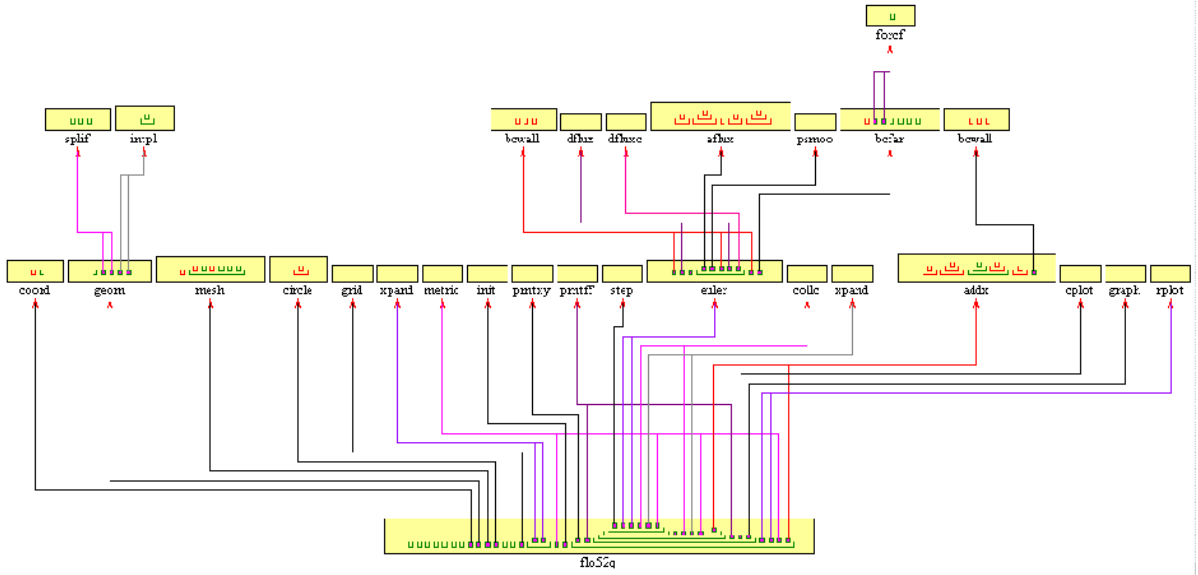

Рис. 11. Расширенный граф вызовов с циклическим профилем

ставляют группы операторов. Дуга между двумя вершинами графа управления проводится в том случае, если после выполнения программы, соответствующей начальной вершине дуги, управление сразу может быть передано на конечную. Вершинами графа могут выступать отдельные операторы, линейные участки и циклы. Для любой вершины графа поддерживается вывод текста фрагмента программы, соответствующего этой вершине.

6.2. Иерархическое дерево. Вершинами графа являются линейные участки программ, а также самые внешние циклы исследуемого участка. Пример иерархического дерева показан на рис. 12. Возможно перемещение внутрь и наружу циклов, представленных вершиной с числом. Это число отражает глубины вложенности циклических конструкций во фрагменте.

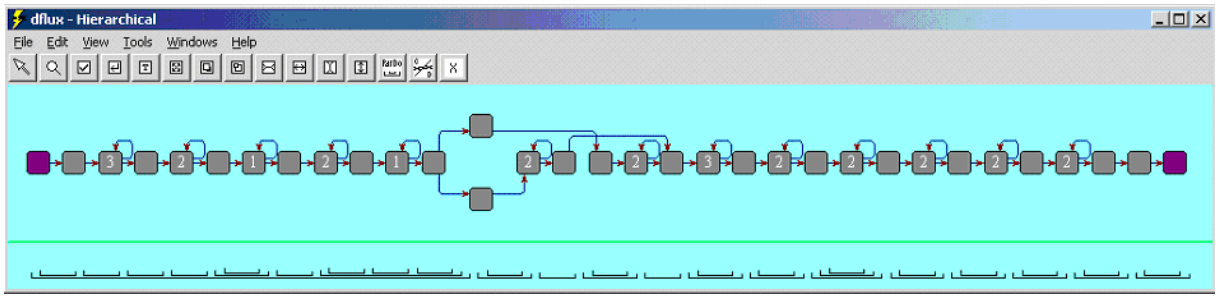

Рис. 12. Иерархическое дерево

6.3. Граф управления в терминах линейных участков. Вершинами графа являются линейные участки программ. Линейный участок — это такая последовательность операторов программы, управление в которой может быть передано только на первый оператор, и после этого обязательно выполнятся все операторы последовательности. Пример графа управления в терминах линейных участков показан на рис. 13.

6.4. Граф управления в терминах отдельных операторов. Вершинами графа являются отдельные операторы программы. Пример такого графа показан на рис. 14.

6.5. Циклический профиль фрагмента. Циклический профиль показывает структуру вложенности циклов исследуемого фрагмента. Каждый цикл исследуемого фрагмента изображается в виде горизонтальной скобки. Вложенность скобок соответствует вложенности циклов. Циклический профиль фрагмента может быть интерпретирован как боковой срез этого фрагмента. Циклический профиль фрагмента можно видеть на рис. 12–14 в нижней части окна.

6.6. Процедурный граф управления. Процедурный граф управления — граф управления, в ко-

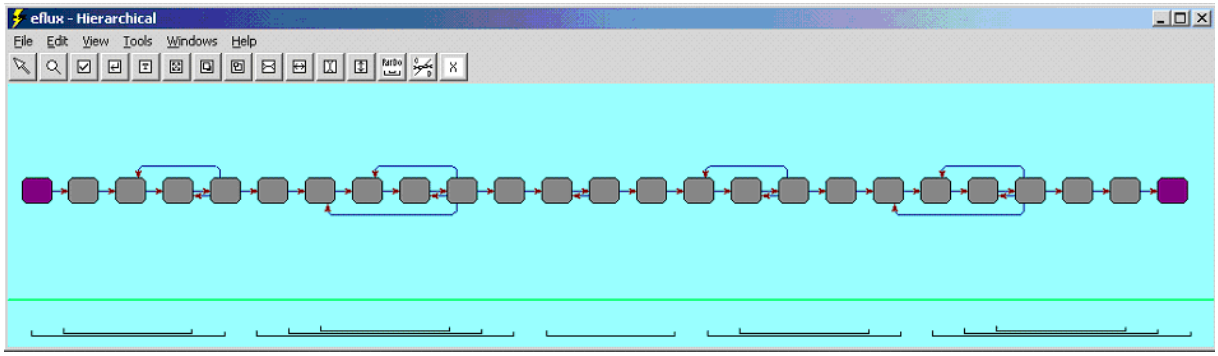

Рис. 13. Граф управления в терминах линейных участков

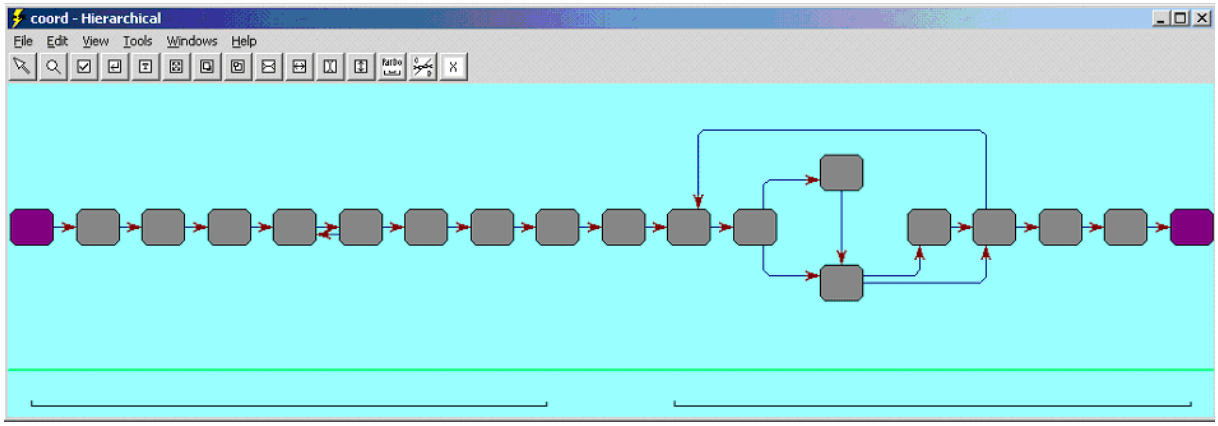

Рис. 14. Граф управления в терминах отдельных операторов

тором оставлены только вершины, соответствующие обращениям к другим подпрограммам и функциям. Позволяет определить, какие вызовы подпрограмм зависят от условных операторов. Процедурный граф управления показан на рис. 15.

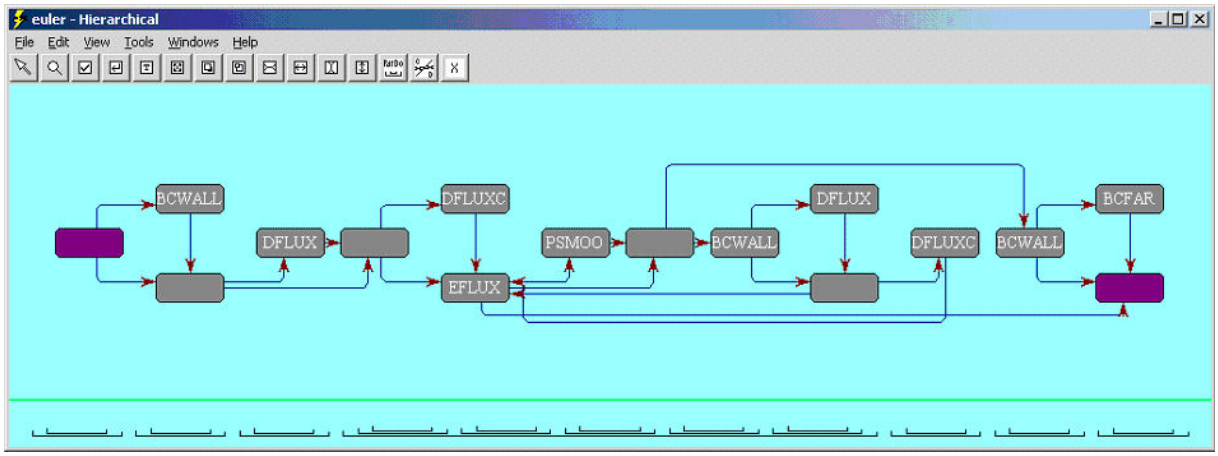

Рис. 15. Процедурный граф управления

7. Анализ глобального использования памяти. Основой анализа использования программными единицами глобальной памяти является граф использования общей памяти. Это двудольный граф, в котором одно множество вершин соответствует процедурам, другое — глобальным переменным. Граф показывает использование подпрограммами этих глобальных переменных. В случае языка ФОРТРАН глобальными переменными являются COMMON-блоки. Вершины разных множеств соединяются дугой

тогда и только тогда, когда соответствующая процедура имеет обращение к данной глобальной переменной (содержит в теле данный COMMON-блок).

Граф использования общей памяти отражает взаимодействие между процедурами, не выраженное в терминах передачи параметров и получения результата. Такое взаимодействие замаскировано. Между тем при адаптации программ под системы с распределенной памятью его необходимо учитывать.

Возможна детализация графа использования общей памяти. Во-первых, можно провести детализацию использования процедурами COMMON-блоков до отдельных переменных, составляющих этот блок. Во-вторых, можно детализировать способ использования каждой переменной — чтение, запись. Пример графа использования COMMON-блоков для программы на языке ФОРТРАН приведен на рис. 16.

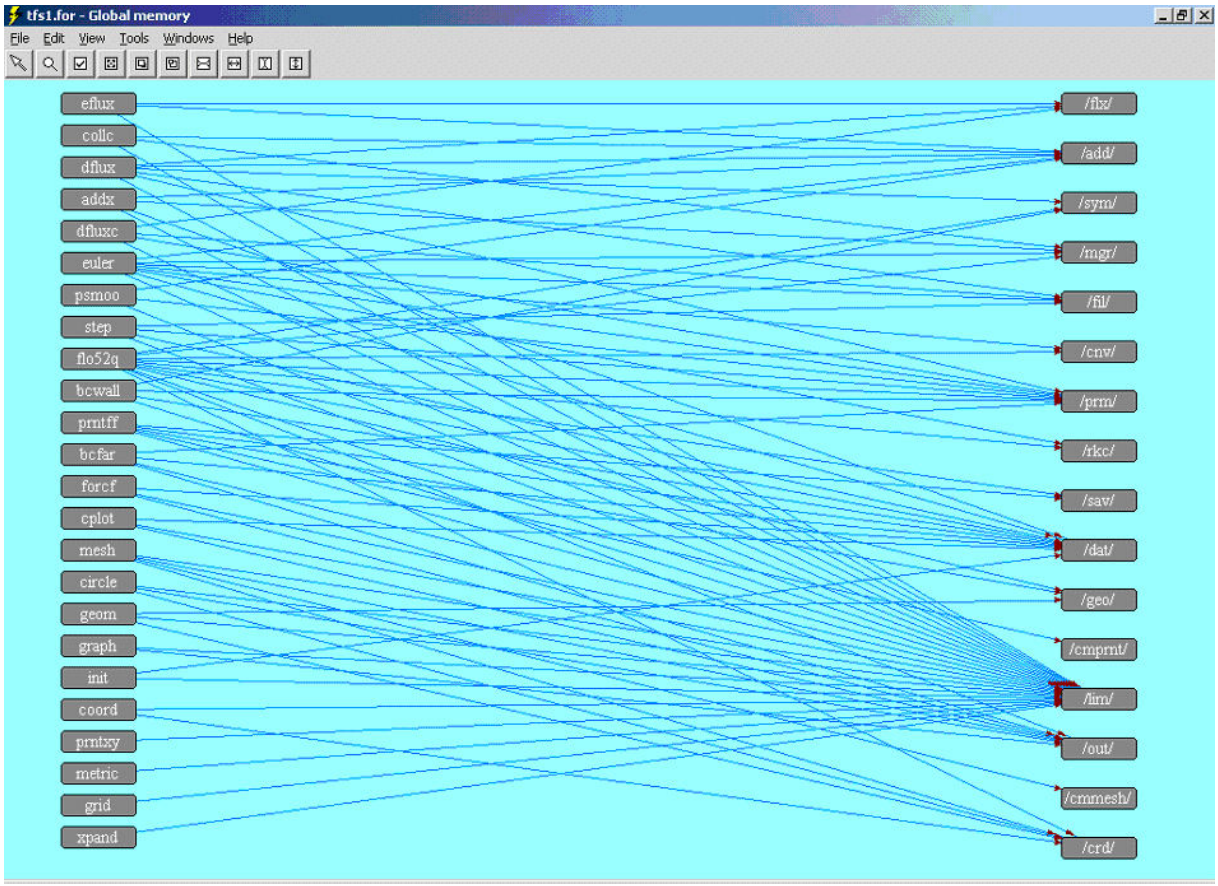

Рис. 16. Граф использования общей памяти

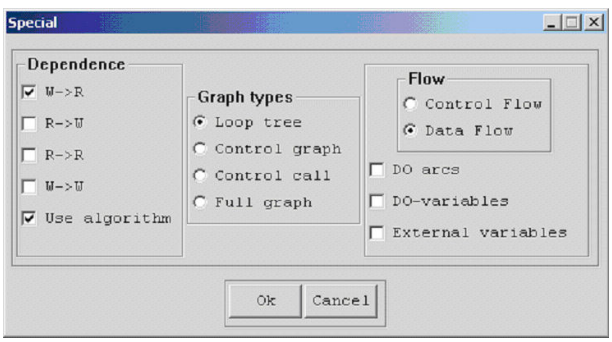

Рис. 17. Окно выбора типа графа

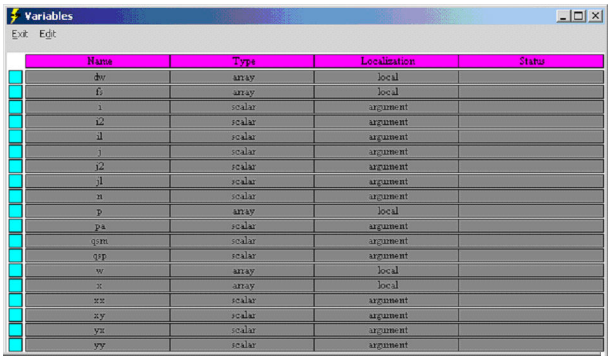

Рис. 18. Окно выбора переменных

8. Исследование информационной структуры программы

8.1. Граф зависимостей программы. Основной инструмент исследования информационной структуры программы — граф зависимостей [6]. Его вершины соответствуют операторам программы, группам

операторов или отдельным срабатываниям операторов. Дуга между вершинами проводится в том случае, если при соответствующих срабатываниях операторов происходит обращение к одной и той же переменной. Дуга всегда проводится от оператора, исполняющегося раньше, к оператору, исполняющемуся позже. Различают четыре типа графа зависимостей в соответствии с типом использования переменных в начальной и конечной вершинах дуги (чтение–чтение, чтение–запись, запись–чтение и запись–запись).

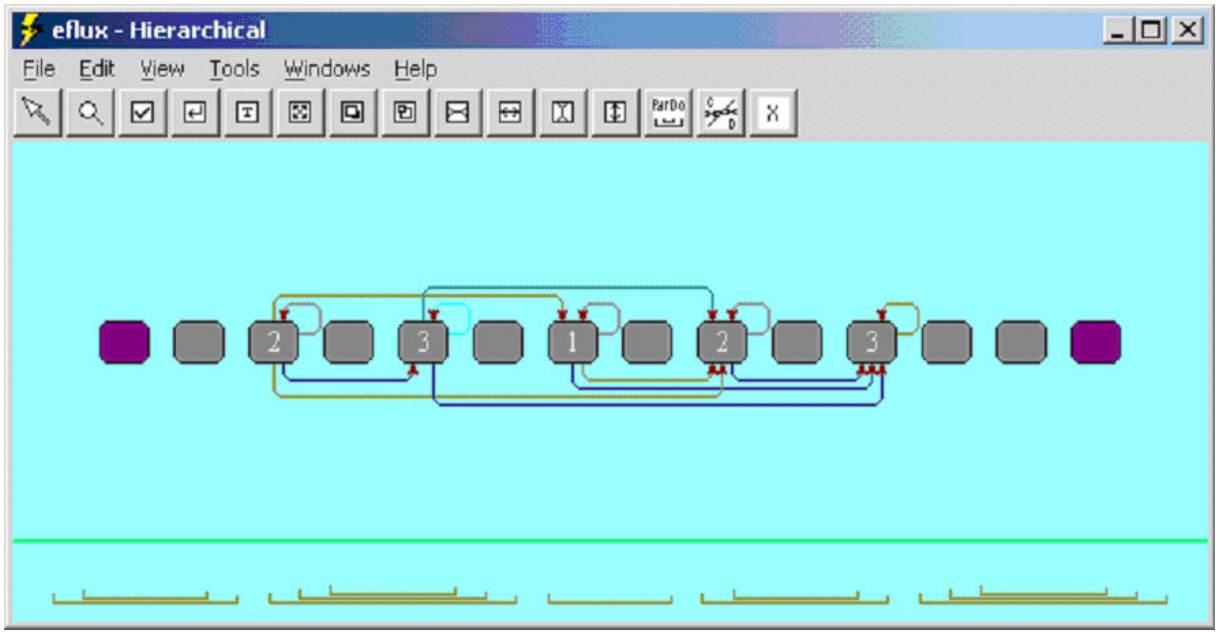

Рис. 19. Граф зависимостей по данным

8.2. Анализ информационной структуры программы. Для отдельной подпрограммы или ее фрагмента (отдельного цикла) система может строить графы зависимостей любого типа. Есть два режима построения графа зависимостей. Один — упрощенный, при его использовании строится надграф реального графа зависимостей (этот режим применяется также тогда, когда программа не может быть проанализирована с использованием точного метода). Второй метод основан на теории, описанной в [6, 8]. Он применим не ко всем программам, но при его использовании получаются точные графы зависимостей. Окно, в котором устанавливается, какой граф мы хотим построить, показано на рис. 17.

Граф зависимостей по данным можно строить по части переменных. Окно выбора переменных показано на рис. 18. Вид окна с графом зависимостей по данным приведен на рис. 19. Виды зависимостей выделены дугами разного цвета.

Работа выполнена при поддержке РФФИ (проекты № 05–07–90206 и 05–07–90292).

СПИСОК ЛИТЕРАТУРЫ

1. *Shen Zh., Li Zh., Yew P.-Ch.* An empirical study of Fortran programs for parallelizing compilers // IEEE Trans. on Parallel and Distributed Systems. 1990. **1**, N 3. 350–364.
2. *Davies J. et al.* The KAP/S-1: an advanced source-to-source vectorizer for the S-1 Mark II supercomputer // IEEE Proc. of ICPP. 1986. 833–835.
3. *Huson C. et al.* The KAP/205: an advanced source-to-source vectorizer for the Cyber 205 supercomputers // IEEE Proc. of ICPP. 1986. 827–832.
4. *Maske T. et al.* The KAP/ST-100: A Fortran translator for the ST-100 attached processor // IEEE Proc. of ICPP. 1986. 171–175.
5. The FORGE Product Set. Placerville: Applied Parallel Research Inc., 1995.
6. *Воеводин В.В.* Информационная структура алгоритмов. М.: Изд-во МГУ, 1997.
7. *Voevodin V.V., Voevodin Vl.V.* V-Ray technology: a new approach to the old problems. Optimization of the TRFD Perfect Club Benchmark to CRAY Y-MP and CRAY T3D supercomputers // Proc. High Performance Comput. Symposium'95. Phoenix, Arizona, 1995. 380–385.
8. *Воеводин В.В., Воеводин Вл.В.* Параллельные вычисления. СПб.: БХВ, 2002.
9. http://www.extreme.indiana.edu/sage

Поступила в редакцию
01.02.2005