ЧЕБЫШЕВСКИЙ СБОРНИК
Том 18 Выпуск 3

УДК 519.6 DOI 10.22405/2226-8383-2017-18-3-187-200

ПАРАЛЛЕЛИЗМ В СЛОЖНЫХ ПРОГРАММНЫХ КОМПЛЕКСАХ (ПОЧЕМУ СЛОЖНО СОЗДАВАТЬ ЭФФЕКТИВНЫЕ ПРИКЛАДНЫЕ ПАКЕТЫ)[1]

В. В. Воеводин[2] (г. Москва)

Аннотация

В работе рассмотрены виды параллелизма, применяемые в архитектурах современных компьютерных систем, и описаны способы их проявления в программах. Проанализированы шесть парадигм параллельного программирования, и показана связь парадигм с поколениями высокопроизводительных вычислительных систем. Рассмотрены методы описания и представления параллелизма с помощью разного рода моделей программ. Обсуждаются причины, определяющие сложности разработки эффективного программного обеспечения для параллельных вычислительных систем. Отмечается связь обсуждаемого материала с активно развиваемой Интернет-энциклопедией свойств и особенностей параллельных алгоритмов AlgoWiki.

Ключевые слова: Параллельные вычисления, архитектура компьютерных систем, технологии параллельного программирования, высокопроизводительные вычисления.

Библиография: 9 названий.

PARALLELISM IN LARGE SOFTWARE PACKAGES (WHY IS IT DIFFICULT TO CREATE EFFICIENT SOFTWARE)

V. V. Voevodin

Abstract

[1]Исследование выполнено в Московском государственном университете имени М. В. Ломоносова за счет гранта Российского научного фонда (проект № 14-11-00190).

[2]Воеводин Владимир Валентинович, член-корреспондент Российской академии наук, заместитель директора НИВЦ МГУ им. М. В. Ломоносова, voevodin@parallel.ru

In the article, types of parallelism used in architectures of modern computer systems are considered, and the ways of their manifestation in programs are described. Six paradigms of parallel programming are analyzed, and the relationship of paradigms to generations of high-performance computing systems is shown. Different methods of description and representation of parallelism based on various kinds of program models are considered. The reasons that determine challenges of developing efficient software packages for parallel computing systems are discussed. The connection between the material under discussion and the actively developed Internet encyclopedia of properties and features of AlgoWiki parallel algorithms is noted.

Keywords: Parallel computing, computer architecture, parallel computing technologies, high performance computing

Bibliography: 9 titles.

1. Введение

Разработка эффективного программного обеспечения всегда было дело не самым простым, но за последние годы ситуация значительно усложнилось. Создавая программу, разработчик всегда принимает компромиссное решение между эффективностью, переносимостью и продуктивностью, что определяет и технологию программирования, и стиль программирования, и всё остальное. Если нужно быстро получить работающий вариант программы, то акцент делается на продуктивности. Если важна работоспособность программы на многих архитектурах, то переносимость имеет явный приоритет. В реальности, в той или иной степени приходится учитывать все три параметра, что отражается на всем цикле разработки программного обеспечения.

Высокопроизводительные вычисления ориентированы, прежде всего, на минимизацию времени решения задач, а значит всё то, что определяет эффективность программ, должно рассматриваться особенно аккуратно. И в первую очередь – особенности архитектуры компьютеров: если они учтены, программа использует весь потенциал архитектуры, в противном случае эффективность снижается и, как следствие, падает производительность. Учесть нюансы конкретной архитектуры сложно, но вполне реалистично. Однако за последние 40 лет сменилось, по крайней мере, шесть поколений архитектур параллельных вычислительных систем, и каждая их них требовала как специальных свойств от алгоритмов, так и своего стиля написания программ. По существу, каждое новое поколение архитектуры компьютеров приводило к необходимости полного пересмотра программного обеспечения. Сделав эффективный пакет под одну архитектуру, уже через несколько лет в пакет нужно вносить серьезные изменения, иначе его конкурентоспособность будет потеряна. Компьютерный мир исключительно динамичен, и наибольшие сложности вызывают изменения, привносимые различными видами параллельной обработки данных. Из-за них приходится менять технологии про-

граммирования, детально исследовать используемые алгоритмы, проводить глубокий анализ динамических свойств программ.

Основная цель данной работы – проанализировать особенности параллелизма, свойственные вычислительным системам, и соотнести с возможными формами их отражения в программах. Это один их элементов суперкомпьютерного кодизайна, без учета которого невозможно проектировать эффективные параллельные приложения. В работе будут рассмотрены виды параллелизма, применяемые в архитектурах компьютерных систем (раздел 1), и способы их проявления в программах (раздел 2). Будут рассмотрены методы описания и представления параллелизма (раздел 3) с помощью разного рода моделей программ. В процессе изложения будет показана глубокая связь обсуждаемого материала с активно развиваемой энциклопедией свойств и особенностей параллельных алгоритмов AlgoWiki [1].

2. Параллелизм в компьютерных системах

Рассмотрим особенности организации шести поколений архитектур параллельных вычислительных систем. Это поможет не столько понять специфику поколения прошлых лет, сколько ощутить потенциал средств параллельной обработки, используемых при построении компьютерных систем, как современных, так и будущих.

Поколение векторно-конвейерных компьютеров начало активно развиваться с середины 70-х годов прошлого века, когда был выпущен суперкомпьютер Cray-1. Машины этого класса опирались на конвейерную обработку векторов данных, что поддерживалось векторными функциональными устройствами и векторными инструкциями в системе команд компьютеров. Самый эффективный способ выполнения программы – это векторизация, т.е. трансляция фрагментов исходного кода в векторные команды. Естественными кандидатами на векторизацию являются самые внутренние циклы. Необходимое условие векторизации состоит в отсутствии информационной зависимости между итерациями цикла, чтобы их можно было бы исполнять независимо друг от друга, в частности, параллельно. Полная векторизация программы подразумевает векторизацию всех самых внутренних циклов, однако на практике интерес представляет, конечно же, лишь вычислительное ядро: если оно векторизовано, то и выполнение программы будет эффективным. Еще один вид параллелизма в компьютере представлен независимыми функциональными устройствами, которые можно использовать в режиме зацепления наряду с операциями чтения/записи. В этом случае создаётся дополнительный уровень конвейерной обработки, что обеспечивает и дополнительное ускорение.

В 80-х годах стали появляться компьютеры, имеющие в своей архитектуре не один, а несколько независимых векторно-конвейерных процессоров: векторно-параллельные компьютеры. Типичные представители компьютеров этого класса того времени: Cray X-MP, Cray Y-MP, позднее Cray C90/T90. К

параллелизму рассмотренного выше векторно-конвейерного процессора добавился уровень нескольких процессоров (от 2 до 32), работающих независимо друг от друга. Требования на структуру программ стали иными. Для поддержки конвейерной обработки векторов, как и в предыдущем случае, нужен параллелизм внутренних циклов. Но теперь в программах нужно найти и описать дополнительный ресурс параллелизма, обеспечивающий независимую работу процессоров.

В начале 90-х годов широкое распространение получили массивно-параллельные компьютеры с распределенной памятью, объединяющие в своем составе тысячи относительно маломощных, но серийно производимых процессоров. Примером могут служить Intel Paragon XP/S (3680 процессоров Intel i860, 50MHz), IBM SP2 (512 процессоров IBM POWER2, 66MHz) или Cray T3D (1024 процессора DEC Alpha EV4, 149MHz). Для эффективного выполнения программ на подобной архитектуре требуются два действия. Во-первых, нужно выделить в алгоритмах и отразить в программах значительный ресурс параллелизма для обеспечения независимой работы большого числа процессоров. Во-вторых, нужно найти и описать такое распределение данных по процессорам, при котором обмен по коммуникационной сети в процессе выполнения программы был бы минимальным. Необходимость выполнения этих действий потребовала не только пересмотра алгоритмического багажа, новой технологии программирования (стандартом де-факто стала технология MPI [2]), но и полного переписывания программного обеспечения. Заметим, что к уровню параллелизма, отражающему множество независимых процессоров, добавляется и внутренний параллелизм самих процессоров. И если суперскалярность и VLIW отражают параллелизм на уровне команд, поддерживаются аппаратурой и/или компиляторами и, как правило, не требуют вмешательства программиста, то для полного использования потенциала конвейерности и параллельности необходимы явные изменения программы.

Чуть позже стали появлялись компьютеры с общей памятью. Общая память значительно упрощала взаимодействие между процессорами, значительно сокращая накладные расходы на обмен данными и синхронизацию. Появилась технология OpenMP [2], которая отражала идею работы параллельных программ в парадигме единого адресного пространства. А это значит – новая модель параллельной программы, новые средства и методы программирования, новые конструкции, и, следовательно, программы для компьютеров с общей памятью пришлось создавать заново.

В начале 2000-х годов появились компьютеры, соединившие особенности двух предыдущих классов: кластеры с распределенной памятью, построенные на основе узлов с общей памятью. Типичный пример: суперкомпьютер МГУ “Чебышев”, объединяющий 625 узлов, каждый их которых состоит из двух 4-ядерных процессоров. При программировании таких систем одну часть ресурса параллелизма нужно оставить для использования независимых узлов, а другую часть – под использование нескольких процессоров или ядер в рамках каждого узла. В параллельных приложениях первая часть

описывается, как правило, через MPI, и одновременно вторая с помощью OpenMP, и оба этих уровня параллелизма должны найти отражение в результирующей программе.

С 2007 года в архитектуре компьютеров начали активно использоваться ускорители: сначала графические процессоры от NVIDIA и AMD, затем ускорители типа Intel Xeon Phi. Но чтобы ускоритель был бы эффективно использован, следует учитывать, по крайне мере, три факта. Во-первых, ускоритель может работать параллельно с управляющим процессором (хостом). Во-вторых, для передачи данных на ускоритель требуется время. И, в-третьих, степень параллельности современных ускорителей очень велика: графические процессоры содержат сотни и тысячи ядер, да и общая степень параллельности процессоров семейства Phi составляет величину порядка 2000.

Сейчас ускорители стали обычным явлением, в том числе в составе больших кластеров. Например, в состав части узлов суперкомпьютера МГУ "Ломоносов" входят ускорители NVIDIA 2070/2090 [2], в состав каждого вычислительного узла суперкомпьютера МГУ "Ломоносов-2" входит один ускоритель NVIDIA Tesla K40, а все узлы китайского суперкомпьютера Tianhe-2 содержат по три ускорителя Intel Xeon Phi. Но что означает наличие ускорителей в архитектуре суперкомпьютеров с точки зрения создания приложений? Во-первых, в программе нужно выделить весомый ресурс параллелизма для использования множества вычислительных узлов. Во-вторых, дополнительный параллелизм должен быть в каждом параллельном MPI-процессе для использования нескольких ядер на узле. В-третьих, должен остаться значительный объём параллелизма, чтобы использовать возможности ускорителей на каждом узле. И все эти уровни должны быть представлены в программе, отражая сложную иерархичную модель параллелизма вычислительных систем.

Каждое из шести рассмотренных поколений привносило новые особенности в архитектуру компьютеров, что с необходимостью вызывало изменения в программах, часто кардинальные. В последние годы этот процесс идёт еще более активно: частота процессоров не растет, и эпоха бесплатного роста производительности завершилась. Это стараются компенсировать увеличением степени параллелизма в архитектуре, объединяя отработанные в предыдущих поколениях технологии с новыми идеями. Компьютеры, как и параллелизм в их архитектуре, становятся все более и более неоднородными, что опять требует и ревизии свойств алгоритмов, и пересмотра технологий создания программ. В самом деле, в современных компьютерах можно найти абсолютно все обсуждавшиеся выше элементы параллелизма, свойственные предыдущим поколениям параллельных вычислительных систем. Суперкомпьютер Tianhe-2 состоит из 16000 узлов, в каждом узле по два 12-ядерных процессора и по три ускорителя Intel Xeon Phi 31S1P. Появившаяся ещё на заре суперкомпьютерных технологий векторная обработка активно используется не только в больших машинах типа NEC SX-ACE и поддерживается процессорами Intel Xeon Phi, но также включена в архитектуру будущих

процессоров от ARM и Fujitsu. Практически все серверные процессоры являются суперскалярными, а многие графические процессоры опираются на идеи SIMD/VLIW обработки. Растет число ядер в процессорах и число узлов в кластерах. Компания Intel анонсировала интеграцию FPGA-сегментов с многоядерными процессорами. Компания NVIDIA объявила о выпуске процессора Volta: TESLA V100, который содержит 5120 CUDA-ядер и ориентирован на задачи глубокого обучения. Как должна быть написана программа, чтобы использовать всё это потенциальное богатство? Ясно, что в таких условиях поддерживать конкурентоспособность прикладных пакетов крайне не просто: приходится либо жертвовать эффективностью, либо вкладывать дополнительные средства и постоянно переписывать код, адаптируя его под бесконечные изменения в архитектуре вычислительных систем.

3. Параллелизм в программах

Теперь посмотрим на ту же проблему с другой стороны: как параллелизм может (или должен) быть представлен в программах, чтобы отражать все описанные выше особенности архитектуры? Проблема в том, что есть лишь несколько счастливых исключений, когда пользователям в этом плане ничего делать не нужно: суперскалярность и VLIW. Суперскалярность поддерживается на уровне аппаратуры, а планирование кода для использования VLIW-возможностей берёт на себя компилятор. В остальных случаях пользователи должны описать параллелизм явно. Безусловно, можно вызвать процедуру из уже оптимизированной библиотеки, которая скроет работу с параллелизмом. Однако для нас это сейчас принципиально ничего не меняет, поскольку проблема остаётся и просто перекладывается с пользователей на разработчиков библиотеки.

На самом верхнем уровне параллелизм программ можно разделить на два больших класса: конечный и массовый. Конечный параллелизм определяется независимостью какого-то числа фрагментов программы: операторов, линейных участков, циклов либо же просто каких-то ее больших или маленьких частей. Число независимых фрагментов определяется информационной структурой и не зависит от входных данных программы. Выразить конечный параллелизм в программах просто, например, с помощью директив OpenMP “#pragma omp sections”. Но в использовании данный вид параллелизма не удобен: ресурс параллелизма сильно ограничен и фиксирован, а вычислительная сложность, как и состав операций независимых частей, могут сильно различаться, привнося значительный дисбаланс в вычисления.

Массовый (итеративный) параллелизм определяется независимыми итерациями циклов. Поскольку число итераций циклов может быть значительным, отсюда и название. Это основной ресурс параллелизма, который позволят создавать масштабируемые программы. Вместе с этим, наличие массового параллелизма является всего лишь необходимым, но не достаточным условием для хорошей масштабируемости программ: многое зависит, в част-

ности, от особенностей взаимодействия программы с памятью.

Самый простой пример массового параллелизма в программе – это одиночный цикл с независимыми итерациями. Такой цикл может быть распараллелен, например, с помощью директив OpenMP “#pragma omp for”. С помощью последовательности эквивалентных преобразований текста программы (loop interchange, loop distribution, loop fusion) такой цикл во многих случаях можно сделать самым внутренним, и тогда возможна его векторизация.

Более сложный вариант представления массового параллелизма в программах задают многомерные вложенные циклические конструкции. В этом случае массовый параллелизм дополнительно делится на два класса: координатный и скошенный. Говорят, что гнездо циклов обладает координатным параллелизмом, если множества независимых итераций можно расположить на гиперплоскостях, перпендикулярных одной из координатных осей, использованных для описания пространства итераций данного гнезда. На рис.1 (а,б) показан фрагмент программы, являющийся двумерным гнездом циклов, и отвечающий фрагменту информационный граф. Данный фрагмент обладает координатным параллелизмом: необходимые гиперплоскости с независимыми итерациями перпендикулярны оси j.

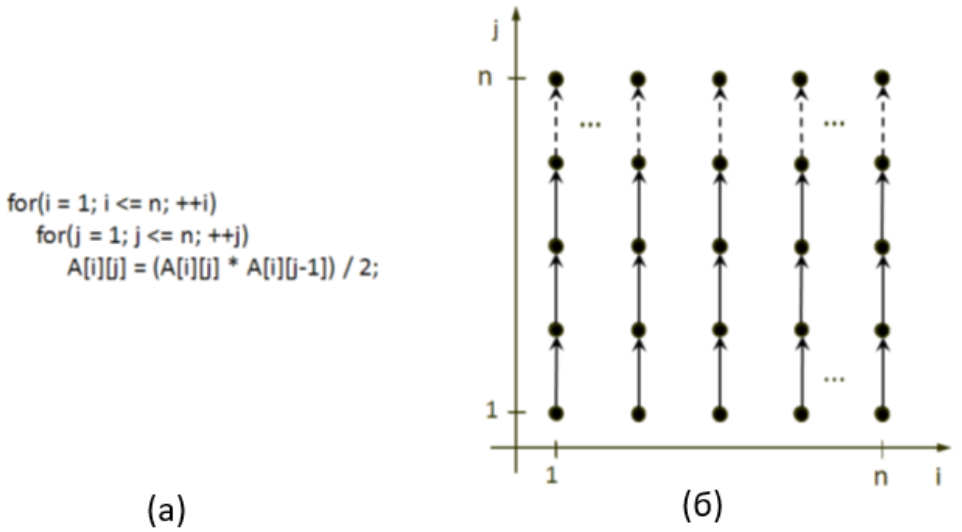

Рис. 1: Координатный массовый параллелизм в программах: исходный текст программы (а) и ее информационная структура (б)

Координатный параллелизм удобен на практике. Во-первых, для его выражения можно использовать директивы OpenMP типа “#pragma omp for”, которые достаточно расположить перед заголовком соответствующего цикла без необходимости дополнительного преобразования текста программы. Во-вторых, сбалансированность результирующей параллельной программы определяется сбалансированностью итераций цикла, что контролировать, как правило, проще.

В некоторых случаях программы обладают массовым параллелизмом, но гиперплоскостей с необходимыми свойствами нет. Типичный пример показан на рис. 2 (а,б,в): гиперплоскости с независимыми итерациями есть, но они расположены под углом ко всем координатным осям. Именно этот факт определил и специальное название для такого вида параллелизма – скошенный. В таком случае нельзя просто поставить аналогичную директиву OpenMP перед каким-либо циклом, да и для выражения такого параллелизма программу предварительно необходимо преобразовать, что не всегда тривиально. Это первый недостаток скошенного параллелизма. Вторая сложность связана с потенциально высокой несбалансированностью параллельной программы. Это более серьезная проблема, чем разовое преобразование текста, поскольку эта особенность определяет качество будущей параллельной программы. В самом деле, в приведенном примере (рис.2в) число независимых итераций в ярусах параллельно формы сначала постепенно растет от 1 до n, а затем снова уменьшается до 1.

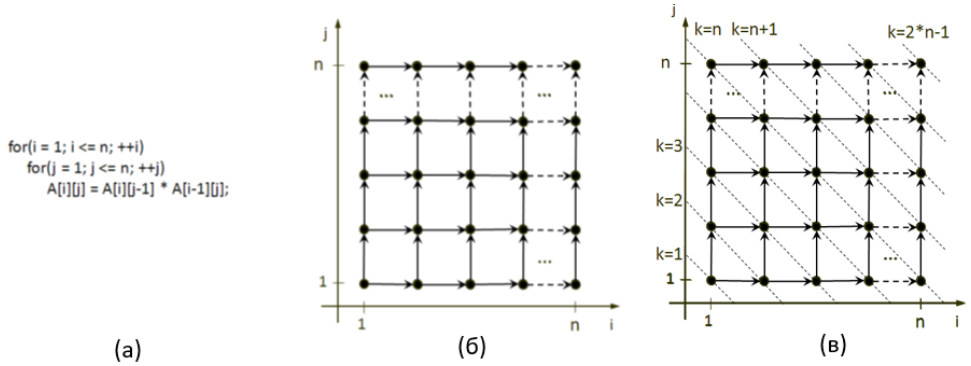

Рис. 2: Скошенный массовый параллелизм в программах: исходный текст программы (а), ее информационная структура (б) и гиперплоскости с независимыми операциями (в)

Скошенный параллелизм является частным случаем параллелизма, который может быть описан с помощью механизма разверток [4, 5]. В общем случае фронт волны, содержащий независимые операции, имеет форму более сложную, чем гиперплоскость, что не просто найти и описать. Вместе с этим, когда других вариантов для создания параллельной программы нет, то приходится использовать любую возможность, пусть даже сложную и неудобную.

Очень важное свойство параллелизма программ – его иерархичность. При этом иерархичность определяется как структурой программы в терминах процедур и функций, так и комбинацией конечного и массового параллелизма. Конечный параллелизм на внешнем уровне программы определяет несколько параллельных ветвей вычислений. Ветви могут содержать вызовы процедур, каждая из которых может отвечать фрагменту программы произвольной сложности. В ветвях может быть какое-то число одномерных или многомерных параллельных циклов, обладающих координатным или ско-

шенным параллелизмом. Каждая итерация таких циклов может быть единственным оператором присваивания, но может быть и сложным фрагментом программы, имеющим свой конечный параллелизм, где в параллельных ветвях снова могут содержаться циклы и так далее до отдельных операторов программы.

Подобная иерархическая структура параллелизма лежит в основе такого понятия, как потенциал (или ресурс) параллелизма программы. Определить и описать его не просто, поэтому на практике для получения эффективных параллельных программ, чаще всего, рассматривают лишь локальный параллелизм в пределах небольшого фрагмента. Иногда этого оказывается достаточным, но в некоторых случаях получить хорошую реализацию можно только после анализа всего ресурса параллелизма и последующего глубокого преобразования программы.

4. Представление потенциала параллелизма программ

Представление потенциального параллелизма в программах определяется упомянутой выше иерархичностью. Конечный и массовый параллелизм описывается разного рода информационными моделями программ. Для описания конечного параллелизма хорошо подходит информационный граф программы, в котором вершины отвечают фрагментам программы, а дуги – зависимости по данным, причем число вершин и дуг определяется лишь количеством операторов и никак не зависит от входных параметров программы. Для описания массового параллелизма необходимо обратиться к более общему понятию графа алгоритма (граф зависимостей по данным, data-flow граф). В отличие от информационного графа программы, где каждый оператор представлен лишь одной вершиной, в графе алгоритма для каждого срабатывания каждого оператора есть отдельная вершина. Исследование графа алгоритма позволяет выделить массовый параллелизм во всей его полноте, как координатный, так и скошенный.

Параллелизм программ (т.е., в конечном итоге, и параллелизм алгоритмов) нужно каким-то образом описать. Оснований для этого много, но основная причина – это обеспечение переносимости программ с одной архитектуры на другую с сохранением разумного уровня производительности. Если результаты анализа потенциала параллелизма отражены в программе, то становится понятным, на что можно рассчитывать при переносе программы на новую параллельную платформу.

Это сложный вопрос, который в полной мере не решен до сих пор. Но вопрос очень важный, который позволит накапливать и сохранять знание о параллельной структуре программ и алгоритмов. Частичное решение может быть получено с использованием традиционных технологий параллельного программирования, в которых предусмотрено выражение параллелизма. Возьмем, например, OpenMP: как конечный, так и координатный парал-

лелизм можно описать соответствующими директивами. Полного решения проблемы это не дает, но часть потенциала параллелизма уже становится понятной. Чтобы описать скошенный параллелизм, и тем самым потенциал параллелизма общего вида, требуется привлекать механизм разверток: эта техника в вычислительной практике пока не отработана, но она незаменима для обеспечения переносимости программ.

Обратим внимание на три дополнительных способа представления структуры программ, которые используются редко, но исключительно удачно показывают потенциал координатного параллелизма, наиболее востребованного на практике. Первый способ – это циклический профиль. Пример циклического профиля показан на рис.3. Это своего рода "разрез" программы (процедуры, фрагмента) или же взгляд "сбоку". Мы видим циклическую структуру данного фрагмента в целом, на которой одновременно выделены и все параллельные циклы (цифрами '1' и '2'). Подобная информация о структуре фрагмента исключительно важна на практике, поскольку, в частности, для распараллеливания можно сразу воспользоваться параллельностью внешнего цикла '1'. Вместе с этим, для векторизации нужен другой ресурс параллельности, выраженный в данном примере циклами с пометкой '2'.

Рис. 3: Циклический профиль программы, содержащий информацию о параллельной структуре циклов

Второй способ, учитывая структуру вызовов процедур, расширяет возможности циклического профиля по описанию координатного параллелизма на всю программу. Это циклический профиль путей программы. Данная конструкция исключительно удобна когда нужно быстро оценить как циклическую структуру всей программы в целом, так и ее циклическую сложность (см. рис.4). Для более точного анализа потенциала программы важно знать не только глубину вложенности циклических конструкций в процедурах, но и уметь оценивать общее число объемлющих циклов, в телах которых вызывалась каждая процедура.

$$10 : (GCMA^{3} \xrightarrow{2} DYNIMP^{1} \xrightarrow{1} HHSOLV^{3} \xrightarrow{1} GAUSPH^{2} \xrightarrow{1} SINGL^{5}),$$
$$8 : (GCMA^{3} \xrightarrow{2} RADFL\,2^{4} \xrightarrow{1} RADFSW^{3} \xrightarrow{1} VARP\,8^{4}) \xrightarrow{2} DELED\,3^{1},$$
$$7 : (GCMA^{3} \xrightarrow{2} DYNIMP^{1} \xrightarrow{1} DYNADD^{1} \xrightarrow{0} MMATR\,2^{4}).$$

Рис. 4: Циклический профиль путей программы

Конструкция вида $X^{\alpha} \xrightarrow{\gamma} Y^{\beta}$, используемая на рис.4 в качестве основного элемента каждого пути, означает следующее: процедура X вызывает процедуру Y в γ-мерном цикле, причем максимальная глубина вложенности циклов в самих процедурах X и Y равна α и β соответственно. Если

X содержит несколько вызовов Y, то на месте γ будет указано максимальное значение. Для локализации вычислительного ядра программы интерес представляет как значение α, так и $\gamma + \beta$. Самая левая колонка циклического профиля на рис.4 показывает максимальную суммарную вложенность циклов по каждому пути, а скобками отмечена та цепочка пути, на которой достигается указанное значение. Разметив подобным образом все пути графа вызовов, легко выделить наиболее "весомые которые должны стать кандидатами для детального анализа.

Третий способ: расширенный граф вызовов, является объединением циклического профиля, циклического профиля путей программы и графа вызовов. В каждую вершину графа вызовов добавляется циклический профиль соответствующей процедуры с указанием параллельности циклов профиля. Дуга, связывающая вершины A и B, выходит из того места циклического профиля вершины A, где расположен вызов процедуры B. Если вызовов несколько, то каждому отвечает своя дуга. Расширенный граф вызовов построить и изобразить не так просто, но он исключительно информативен, делая понятным ресурс координатного параллелизма всей программы.

Описанные в данном разделе методы очень хорошо подходят для представления иерархического параллелизма программ. В самом деле, комбинируя информационные модели (граф алгоритма, информационный граф программы) с описательными возможностями циклических профилей и графа вызовов, мы получаем полное описание ресурса координатного параллелизма программ. Важно и то, что составляя модельные программы, реализующие те или иные алгоритмы, мы одновременно получаем и способ описания потенциального параллелизма алгоритмов.

5. Заключение

Суммируя все описанные выше факты, становится понятным, почему столь сложно создавать эффективные прикладные пакеты [6]. Найденный вариант построения параллельной программы для одной платформы, скорее всего, не будет столь же хорошо отражать свойства других платформ, а, значит, не будет и должного уровня эффективности. Отсюда необходимость постоянной модификации прикладных пакетов и все новые и новые затраты на их разработку и развитие. В идеале, программа должна сама автоматически настраиваться на особенности каждой платформы, для чего она должна не только обладать всеми только что описанными сведениями, но и уметь модифицировать свою структуру. Пока это фантастика. В некоторых случаях программы могут “подстраиваться” под структуру иерархии памяти, меняя размеры рабочих массивов (так делается, например, в пакете ATLAS [7]), или выбирать наиболее подходящий код из нескольких заранее скомпилированных вариантов (многовариантная генерация кода), но не более. Принятие более сложных решений – это дело будущего, но к нему нужно двигаться, выделяя, описывая и накапливая ключевые особенности алгорит-

мов, программ и архитектур. В противном случае мы обрекаем себя на вечное переделывание программ под постоянно меняющиеся вычислительные системы. Отчасти на решение этой же задачи направлена и энциклопедия свойств и особенностей алгоритмов AlgoWiki, где основной акцент делается на глубоком исследовании и описании свойств алгоритмов с учетом специфики различных вычислительных платформ.

Поводом для написания данной статьи послужил юбилей Владимира Анатольевича Левина – основного идеолога известного пакета Fidesys [8]. Пакет позволяет осуществить полный цикл прочностного инженерного анализа, предоставляя исключительно развитую функциональность. Существуя в течение уже многих лет, пакет в полной мере отражает все те сложности, с которыми сталкиваются разработчики сложного вычислительно ёмкого программного обеспечения из-за постоянно меняющегося компьютерного мира [9]. И то, что Fidesys удается оставаться успешным проектом и много лет быть востребованным индустриальным и академическим вычислительным сообществом, лишний раз говорит об исключительно высоком уровне команды разработчиков.

Работа выполнена с использованием оборудования Центра коллективного пользования сверхвысокопроизводительными вычислительными ресурсами МГУ имени М.В. Ломоносова.

СПИСОК ЦИТИРОВАННОЙ ЛИТЕРАТУРЫ

1. A. Antonov, V. Voevodin, and J. Dongarra, “Algowiki: an Open encyclopedia of parallel algorithmic features"// Journal of Supercomputing Frontiers and Innovations, vol. 2, no. 1, 2015, pp. 4-18.

2. А.С.Антонов, “Технологии параллельного программирования MPI и OpenMP”. Изд-во Московского университета, М., 2012, 344с.

3. V. Sadovnichy, A. Tikhonravov, Vl. Voevodin, and V. Opanasenko "Lomonosov": Supercomputing at Moscow State University. In Contemporary High Performance Computing: From Petascale toward Exascale (Chapman & Hall/CRC Computational Science), pp.283-307, Boca Raton, USA, CRC Press, 2013.

4. V. Voevodin, Mathematical Foundations of Parallel Computing. World Scientific Publishing Co., Series in Computer Science, 1992, vol. 33, 364pp.

5. V. Voevodin, Vl. Voevodin, Parallel Computing. BHV-Petersburg, St. Petersburg, 2004, 608pp.

6. V. Voevodin, “Parallel Algorithms: Theory, Practice and Education”// Proceedings of the Joint Workshop on Sustained Simulation Performance, University of Stuttgart (HLRS) and Tohoku University, Springer International Publishing, 2016, pp. 3-10.

7. ATLAS installation guide. [Online] Available: http://math-atlas.sourceforge.net/atlas_install/

8. Fidesys – программный пакет для инженерного анализа. [Online] Available: https://cae-fidesys.com/ru

9. Левин В. А., Вершинин А. В., Сабитов Д. И. и др. Использование суперкомпьютерных технологий в задачах прочности. Пакет Fidesys // Суперкомпьютерные технологии в науке, образовании и промышленности. — 2-е изд. — М.: МГУ, 2010, с.161-166.

REFERENCES

1. Antonov, A.S., Voevodin, V.V. & Dongarra, J. 2015, “Algowiki: an Open encyclopedia of parallel algorithmic features *Journal of Supercomputing Frontiers and Innovations*, vol. 2, no. 1, pp. 4-18.

2. Antonov, A 2012, *Tehnologii parallel'nogo programmirovanija MPI i OpenMP [Parallel Programming Technologies MPI and OpenMP]*. Izdatelskij Dom MGU, Moscow, 344p.

3. Sadovnichy, V., Tikhonravov, A., Voevodin, V. & Opanasenko V. 2013, "Lomonosov": Supercomputing at Moscow State University. In Contemporary High Performance Computing: From Petascale toward Exascale” (Chapman & Hall/CRC Computational Science), CRC Press, Boca Raton, USA, pp. 283-307.

4. Voevodin, V. 1992 Mathematical Foundations of Parallel Computing (Series in Computer Science), World Scientific Publishing Co.Pte. Ltd., Singapure, vol. 33, 364p.

5. Voevodin, V.V., Voevodin, Vl.V. 2004, Parallel'nye vychislenija [Parallel computing], BHV-Peterburg, St. Petersburg, 608p.

6. Voevodin V. “Parallel Algorithms: Theory, Practice and Education”, (Proceedings of the Joint Workshop on Sustained Simulation Performance, University of Stuttgart (HLRS) and Tohoku University), Springer International Publishing, 2016, pp. 3-10.

7. ATLAS installation guide (2016), Available at: http://math-atlas.sourceforge.net/atlas_install/ (accessed 5 December 2017).

8. Fidesys – software package for engineering analysis (2017), Available at: https://cae-fidesys.com/ru/download (accessed 5 December 2017).

9. Levin, V.A., Vershinin A.V., Sabitov D.I., Nikiforov I.V. & Pendjur D.A. 2010 “Using a supercomputer technologies in strength problems. Fidesys Package”, Supercomputer technologies in science, education and industry, Izdatelskij Dom MGU, Moscow, vol.2., pp.161-166.

получено 22.05.2017
принято в печать 14.09.2017