ПРОБЛЕМЫ ПРОГРАММИРОВАНИЯ СУПЕРКОМПЬЮТЕРОВ НА БАЗЕ МНОГОЯДЕРНЫХ МУЛЬТИТРЕДОВЫХ КРИСТАЛЛОВ

В.В. Корнеев

На сегодня нет архитектурно независимой модели параллельной программы, которая бы годилась для систем из коммерчески доступных микропроцессоров и, например, систем с мультитредовой архитектурой Tera MTA. Подходы к программированию и эффективной загрузке вычислительных систем с миллионами и более процессоров, объединённых накристальными и межкристальными коммуникационными средами, даже не имеют общепризнанных альтернатив. Поэтому необходимо выделить существенные ограничения архитектуры суперкомпьютеров на базе многоядерных мультитредовых кристаллов, чтобы дать разработчикам программного обеспечения модель, выполнение требований которой гарантирует эффективное исполнение параллельных программ. Эта модель также необходима для выбора направления развития компиляторов, автоматически преобразующих программы на традиционных языках программирования в эффективные параллельные программы суперкомпьютеров на базе многоядерных кристаллов.

Необходимость использования многоядерных кристаллов

Основными препятствиями для дальнейшего роста производительности микропроцессоров традиционной архитектуры при увеличении степени интеграции и сопутствующей возможности роста тактовой частоты служат ограничение скорости распространения сигналов на кристалле, энергопотребление и тепловыделение [1]. С уменьшением технологических норм всё большая часть энергии потребляется не для вычислений, а для передачи и хранения данных. Уже при технологических нормах 90нм на тактовой частоте 1ГГц 64-разрядный блок операций с плавающей точкой занимает площадь менее, чем 1кв. мм и потребляет около 50 pJ энергии на одну операцию, в то время как при передаче данных на расстояние 14 мм, равное длине стороны кристалла, расходуется в 20 раз больше, около 1 nJ [2]. В микропроцессоре Itanium только 1% площади кристалла обрабатывает данные, а 99% занимают схемы управления и хранения данных. Рассеиваемая мощность на единицу площади современных кристаллов достигает значений, делающих невозможным дальнейший рост частоты для современных традиционных процессоров с единой сеткой тактовых сигналов, разведенной по всему кристаллу.

Нельзя сказать, что эта реальность развития кристаллов осознана только недавно - основные положения были изложены еще в работе [1], а в [3] была рассмотрена архитектура и организация функционирования вычислительной системы, коммуникации между процессорами которых как внутри кристаллов, так и между процессорами разных кристаллов основаны на принципе близкодействия, что позволяет преодолеть ограничения на рост тактовой частоты за счет использования только коротких проводников.

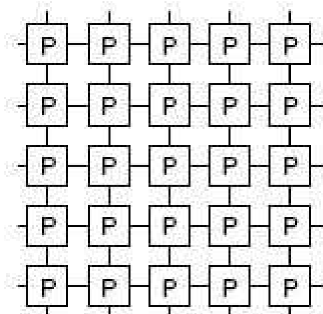

Рис. 1

При структуре кристалла, как на рис. 1., сигналы, включая тактовый сигнал, должны синхронно распространяться только внутри области кристалла, занимаемой одним процессорным ядром (P). Процессорное ядро представляет собой процессор с простой архитектурой, например, без внеочередного исполнения команд и других аппаратно затратных приёмов повышения загрузки функциональных устройств процессора, не обеспечивающих пропорционального этим затратам или большего роста производительности. Все процессорные ядра функционируют на одной тактовой частоте, но между процессорными ядрами возможен произвольный сдвиг фаз тактовых сигналов. Поэтому линии связи между процессорными ядрами должны обеспечивать асинхронную передачу данных.

Процессорное ядро (рис.2) состоит из обрабатывающего блока (процессора) и коммуникационного блока, в свою очередь, включающего входные и выходные очереди портов процессорного ядра и коммутатор. Коммуникационный блок служит, во-первых, для передачи в процессор программ и данных и выдачи из процессора результатов, а, во-вторых, для транзитных передач программ и данных в соседние процессорные ядра.

Возможны и другие варианты реализации накристальной коммуникационной сети, объединяющей процессорные ядра кристалла. Следует отметить, что архитектуры систем на кристалле могут отличаться от архитектур существующих многокристальных ВС. Дело в том, что ограничения для межкристальных и накристальных сетей различны. Для межкристальных сетей существенны число выводов кристалла, ограничивающее ширину линий, а также энергетические затраты, требуемые приемопередатчиками и линиями связи. Для накристальных сетей существенны технологические ограничения разводки широких линий по

кристаллу с исключением помех взаимовлияния линий и допустимое число уровней металлизации для разводки проводников в разных слоях.

Уже имеются реализации существенно многоядерных кристаллов, например Intel Teraflops Research Chip и Tile 64 фирмы Tilera. Эти кристаллы содержат порядка сотен процессорных ядер и имеют пиковую производительность порядка триллиона оп./с.

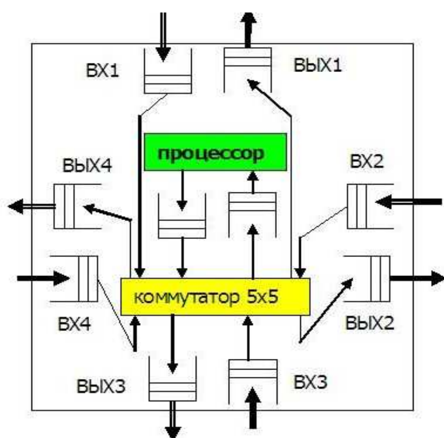

Рис. 2

Итак, используя многоядерные кристаллы можно преодолеть ограничения роста тактовой частоты и создать в ближайшие годы ВС, имеющие порядка 10 миллионов процессорных ядер, которые способны в совокупности достичь производительности 10 в 17 степени flops и более. Однако современные ВС, имеющие порядка 10 000 процессорных ядер, показывают низкую реальную производительность при вычислениях с интенсивным доступом к памяти по адресам, определяемым в ходе выполнения программы. Если в модели параллельной программы не будут предложены механизмы эффективного решения проблемы сокращения простоев процессорных ядер в ожидании данных из памяти, то достижение производительности, близкой к пиковой, будет возможно только для узкого класса задач, не требующих интенсивных обращений к памяти.

Способы сокращения простоя процессора в ожидании доступа к памяти и адекватное представление их в модели параллельной программы

Разрыв в быстродействии процессоров и блоков памяти существовал всегда и постоянно имеет тенденцию к увеличению: тактовая частота процессоров растёт с темпом 40%, а быстродействие схем динамической памяти только 9% в год. В этих условиях исторически первым способом преодоления указанного разрыва стало использование временной и пространственной локализации кодов программ и обрабатываемых данных. В процессоры были введены механизмы виртуальной адресации и кэш-памяти. Эффективное использование этих механизмов потребовало от пользователя формирования кода программ и размещения данных с учётом их локализации. Со временем с этой работой стали успешно справляться компиляторы. Поэтому, хотя для повышения эффективности пользователям в архитектурах современных процессоров предоставляются возможности управления механизмами виртуальной памяти и кэш-памяти, вносить эти механизмы в модель параллельной программы пока не представляется актуальным.

Другим способом преодоления разрыва в быстродействии процессоров и блоков памяти стало предложение работать с памятью в поточном режиме, организуя поток запросов к памяти и получая ответы в том же темпе. Тем самым создаётся режим, при котором время выполнения запроса к памяти определяется темпом выдачи запросов, а не временем выполнения одного запроса. Для реализации этого режима необходимо построить многоблочную расслоенную память и ввести в процессоры механизм отложенных обращений к памяти, допускающий исполнение команд, следующих за командой обращения к памяти, не дожидаясь её завершения. Запуск новых команд на выполнение прекращается при достижении предельных количеств незавершенных обращений к памяти по чтению или по записи, а также, если выполнение очередной команды невозможно без завершения предшествующего обращения к памяти.

Расслоение памяти предполагает задание распределения адресов по блокам памяти. Например, если предполагается предпочтительное обращение к памяти по последовательно увеличивающимся адресам, то блок памяти i будет содержать все такие адреса A, A mod N=i, где N – количество блоков памяти. Если порядок обращений к памяти произвольный, то может быть применено скремблирование, размещающее адреса памяти по блокам определённым псевдослучайным образом.

Вычислительная практика показывает, что интенсивность потока обращений к памяти, создаваемая аппаратурой и компиляторами мультискалярных и VLIW процессоров, недостаточна для скрытия задержки обращения к памяти. Для работы в поточном режиме, при котором время доступа в многоблочную память определяется темпом обращений к ней, требуется иная, более требовательная к пользователям, парадигма программирования: а именно, пользователи должны подготавливать программы, как большую совокупность тредов. И не традиционных POSIX тредов – pthreads, а «лёгких» тредов. Существующие средства программирования тредов, например стандартных POSIX тредов – pthreads, используют для взаимодействия тредов примитивы ОС. Их выполнение вызывает резкое снижение темпа выдачи обращений к памяти при исполнении программ с плохой локальностью данных. Кроме того, эти примитивы способны обеспечить взаимодействие не более сотни тредов, когда для достижения петафлопсного уровня производительности необходимы миллионы и более тредов.

Лёгкие треды состоят из небольшого числа команд и требуют небольшого объема стека для сохранения своего состояния – контекста треда, а также используют другой механизм межтредовой синхронизации и

коммуникации: добавление к каждому слову памяти full/empty (FE) бита и изменение семантики команд обращения к памяти. Значение FE бита full устанавливает, что слово памяти имеет содержимое, в противовес отсутствию содержимого при значении empty [4]. Команды writeef, readfe, readff и writeff, обращающиеся к ячейке памяти, могут выполняться только при определенном в них в первом компоненте суффикса значении бита FE и оставляют после выполнения значение этого бита, заданное программистом во втором компоненте суффикса команды. Выполнение команды задерживается, если FE бит не имеет требуемого значения. Например, команда writeff требует, чтобы перед её выполнением значение FE бита слова памяти, в которое будет запись, было full и оставляет после выполнения это же значение.

Синхронизация на базе FE бита не требует специальных разделяемых переменных, таких как «замки», семафоры и другие, весьма затратных по времени их определения и исполнения при миллионах тредов. Кроме того, применение неделимых (атомарных) последовательностей команд, вычисляющих, используя упомянутые выше разделяемые переменные, значение условия ветвления и выполняющих переход, в соответствии с полученным значением, существенно сложнее и более длительно, чем выполнение команд доступа к памяти при синхронизации динамически порождаемых тредов, в том числе лёгких тредов. Поэтому синхронизация на базе FE бита позволит выдавать максимально возможное в исполняемой программе количество обращений к памяти, генерируемых лёгкими тредами, и параллельно выполнять эти доступы в расслоенной памяти.

Для выполнения на традиционных процессорах программ, созданных на базе модели лёгких тредов, синхронизация которых реализуется FE битами, в [4] в рамках процессов Unix и POSIX pthreads разработана библиотека для порождения и управления лёгкими тредами, названными qthreads. В рамках этой библиотеки FE биты эмулируются хэш-таблицей. На уровне qthread API для поддержки локальности лёгких тредов в распределенной разделяемой памяти вводятся «смотрители», под управлением которых порождаются и протекают qthread-треды в одном и том же со смотрителем, реализованном как pthread, сегменте памяти, в частности, в памяти одного процесса. Показана эффективность использования qthread библиотеки на 48-процессорной ВС SGI Altix при выполнении быстрой сортировки по сравнению с реализацией этого алгоритма, используя libc qsort ().

Очевидно, что введение в модель параллельной программы лёгких тредов с контролируемой пользователем привязкой тредов к сегментам разделяемой памяти, требует от пользователя дополнительных усилий по сравнению просто с представлением алгоритма на языке программирования. Однако лёгкие треды необходимы для эффективной работы памяти в поточном режиме, и пока нельзя рассчитывать на их формирование компилятором.

Потоковое программирование как способ сокращения числа обращений к памяти

Кроме сокращения времени выполнения доступа в память ещё одним приёмом повышения производительности служит потоковое исполнение программ, обеспечивающее уменьшение количества обращений в память за счёт непосредственной передачи операндов между процессорными ядрами, минуя промежуточное хранение операндов в памяти. Следует уточнить, что речь идёт о потоковых программах специального класса, называемых также систолическими или волновыми программами, в которых для заданного графа межядерных связей программируются как вычисления, производимые в ядрах, так и обмены данными между ядрами с синхронизацией вычислений в ядрах с получаемыми данными. Программы, исполняемые ядрами, являются мультитредовыми с множеством лёгких тредов, протекающих в памяти ядра.

В [3] предложено создавать параллельные программы, межядерные потоки в которых программируются исходя из параметрического описания графов связей подсистем, на которых эти программы способны выполняться. Для выполнения таких параллельных программ операционная система должна сформировать связную подсистему процессорных ядер с требуемым программой типом графа межядерных связей и алгоритмом нумерации ядер, приписывающим каждому ядру уникальный номер из диапазона 0, …, N-1, где N – количество ядер в подсистеме. Алгоритм нумерации должен позволять по номеру ядра, на котором он исполняется, вычислять для любого номера i, i {0, …, N-1}, выходное направление, ведущее к ядру с номером i. При этом если номер ядра i и требуется определить выходное направление к ядру i, то результат равен 0, что означает нахождение в требуемом ядре. В случае использования сосредоточенного коммутатора, как это имеет место, например, в Merrimac [1] для всех номеров ядер, кроме собственного номера, должно выдаваться одно и то же направление, ведущее к коммутатору. Для распределенных коммутаторов, например для решётки процессорных ядер эти направления могут принимать значения 1, 2, 3, 4.

На рис. 3 показаны типы графов линейка (рис. 3а), кольцо (рис. 3б), дерево (рис. 3в) и решётка (рис. 3г), для которых могут быть созданы требуемые алгоритмы нумерации [3].

Следует отметить, что подсистемы с типами графов линейка и дерево могут быть сформированы на любом связном подмножестве ядер. Ядра подсистем с графами типов линейка и дерево нумеруются в прямом порядке нумерации вершин дерева, а типа решётка по строкам, как показано на рис. 3. Параллельные децентрализованные алгоритмы построения подсистем с требуемыми числом процессоров и графом межпроцессорных связей представлены в [3].

В каждом ядре подсистемы имеется таблица направлений, в которой для каждого направления хранится номер соседнего по этому направлению ядра и количество ядер поддерева, корнем которого служит соседнее

ядро. Тогда, имея в каждом ядре его окружение {номер j, таблицу направлений Tj, N – количество ядер в подсистеме}, можно определить направление передачи к каждому ядру подсистемы.

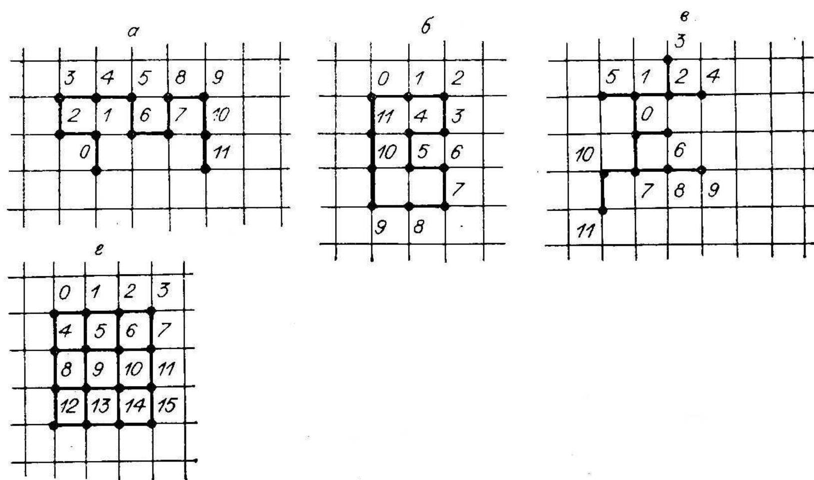

Рис. ٣

При создании параметрически настраиваемой программы пользователь должен выбрать тип графа подсистемы, на которой будет исполняться его программа. Далее, пользуясь атрибутами окружения, запрограммировать путевую процедуру, управляющую передачами данных в подсистеме. Подход к программированию аналогичен применяемому при использовании библиотеки MPI для работы с группой процессов с заданной топологией коммуникатора. Однако принципиальное отличие состоит в том, в библиотеке MPI топология группы процессов определяет виртуальные связи между процессами группы, в то время как в рассматриваемом подходе атрибуты окружения в каждом процессоре получают реальные значения в сформированной подсистеме.

Следует отметить, что программисты могут ограничить круг используемых графов подсистем: программы, созданные для типа графов линейка, могут быть автоматически трансформированы для исполнения на подсистемах с графом дерево.

Программа каждого процессорного ядра состоит из двух частей: путевой и вычислительной процедур. Взаимодействие между ними выполняется через FIFO-очереди ВХ0 и ВЫХ0. Вычислительная процедура берет данные из ВХ0, обрабатывает их и помещает результат в ВЫХ0. Соответственно путевая процедура берет данные из ВЫХ0 и помещает в ВХ0. Путевая процедура в каждом процессоре в соответствии с реализуемым алгоритмом задачи осуществляет выборки данных из входных очередей направлений и выходной очереди собственного процессора и передачу этих данных в выходные очереди направлений или входную очередь процессора. Тем самым, используя значения атрибутов окружения и элементы данных из очередей, в каждом процессорном ядре программно формируются потоки данных, специфичные для реализуемого алгоритма, и организуется параллельное функционирование процессоров ядер и передач данных в подсистеме. Следует отметить, что, как правило, может быть создана одна программа для всех процессорных ядер, настраиваемая в каждом ядре, используя его окружение

В [3] приведены примеры создания путевых и вычислительных процедур для ряда задач линейной алгебры и математической физики с ускорением параллельных вычислений прямо пропорциональным количеству используемых процессоров при соответствующем размере задачи.

Предлагаемая модель программирования с программной настраиваемостью структуры для формирования специфичных для реализуемого алгоритма межядерных потоков данных и мультитредовой на базе лёгких тредов обработки в ядрах дает возможность эффективно использовать ресурсы вычислительных систем на базе многоядерных кристаллов. Эта модель программирования требует от пользователя представления алгоритма вычислений как потоковой схемы или клеточного автомата [5]. Однако вопрос о том, следует ли создавать специальный язык программирования для таких систем или ограничиться библиотеками для работы с лёгкими тредами и окружениями остаётся открытым. Языки программирования предназначаются для компьютерного представления алгоритмов, поэтому следует ожидать развития компиляторов, автоматически или автоматизированно, с участием программиста, формирующих из программ на традиционных языках программы для схемной реализации вычислений на системах с программируемой структурой.

ЛИТЕРАТУРА:

1. Э.В. Евреинов, Ю.Г. Косарев. Однородные универсальные вычислительные системы высокой производительности. Новосибирск: Наука, 1966.
2. W. Dally et al. Merrimac: Supercomputing with Streams SC’03, November 15-21, 2003, Phoenix, Arizona, USA
3. В.В. Корнеев. Архитектура вычислительных систем с программируемой структурой. Новосибирск: Наука, 1985. andrei.klimov.net/reading/1985.Korneev.-.Arkhitektura.vychislitel'nykh.sistem.s.programmiruemoi.strukturoi.zip
4. K. Wheeler et al. Qthreads: An API for Programming with Millions of Lightweight Threads. Proceedings of 22nd IEEE International Parallel and Distributed Processing Symposium. IPDPS 2008
5. T. Tiffolli, N. Margolus. Cellular Automata Machines – USA: MIT Press.1987