УДК 681.31:323

ПОДХОД К ПРОГРАММИРОВАНИЮ СУПЕРКОМПЬЮТЕРОВ НА БАЗЕ МНОГОЯДЕРНЫХ МУЛЬТИТРЕДОВЫХ КРИСТАЛЛОВ

В. В. Корнеев[1]

Предложен подход к построению параллельных программ, адаптированных к архитектуре суперкомпьютеров на базе многоядерных мультитредовых кристаллов. Предлагаемый подход направлен на преодоление отрицательного влияния задержек доступа к памяти на эффективность исполнения программ и может рассматриваться как способ создания масштабируемых параллельных программ.

Ключевые слова: архитектура многоядерных мультитредовых кристаллов, масштабируемые параллельные программы, эффективность параллельных вычислений.

1. Введение. На сегодня нет архитектурно независимой модели параллельной программы, которая могла бы использоваться для создания эффективно исполняемых программ как для систем из коммерчески доступных микропроцессоров, так и, например, систем с мультитредовой архитектурой Tera MTA. Подходы к программированию и эффективной загрузке вычислительных систем с миллионами и более процессоров, объединенных 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.

Статья имеет следующую структуру. Во втором разделе представлены основные особенности архитектуры аппаратных средств перспективных многоядерных кристаллов и суперкомпьютеров на базе таких кристаллов, обусловленные необходимостью преодоления ограничений на рост тактовой частоты и степень интеграции кристаллов. Эти суперкомпьютеры будут содержать много миллионов процессорных ядер, взаимодействующих между собой по каналам "точка–точка" и имеющих доступ к блокам разделяемой памяти, распределенной по узлам суперкомпьютера. Констатируется, что рассматриваемый вариант архитектуры будет эффективен, если будут найдены архитектурные решения и способ программирования, позволяющие в совокупности существенно сократить простои процессорных ядер, обусловленные задержками доступа к памяти.

В третьем разделе рассматриваются архитектурные решения, позволяющие при адекватной им модели программирования существенно устранить потери производительности из-за простоев при доступе к памяти. Эти решения либо используют неявные и явные предсказания адресов обращения к памяти, либо формируют поток запросов, позволяющий получать доступ к памяти с задержкой, равной времени между запросами, а не длительности обращения к памяти.

В четвертом разделе приводится подход к созданию масштабируемых параллельных программ, способных выполняться на подсистемах процессорных ядер с выбранным при написании программы типом графа межъядерных связей подсистемы. Непосредственная передача данных между процессорными ядрами, минуя промежуточное хранение в памяти, позволяет сократить общее количество обращений к памяти, что уменьшает потери эффективности из-за простоев ядер в ожидании доступа к памяти.

Наконец, в пятом разделе резюмируется предлагаемый подход к созданию параллельных программ и модификации архитектуры процессорных ядер и узлов суперкомпьютеров, которые в совокупности должны привести к эффективным вычислениям на суперкомпьютерах на базе многоядерных кристаллов.

2. Необходимость использования многоядерных кристаллов. Основными препятствиями для дальнейшего роста производительности микропроцессоров традиционной архитектуры при увеличении

[1] Научно-исследовательский институт "Квант", 4-й Лихачевский пер., 15, 125438, Москва; зам. директора, e-mail: korv@rdi-kvant.ru

степени интеграции и сопутствующей возможности роста тактовой частоты служат ограничение скорости распространения сигналов на кристалле, энергопотребление и тепловыделение [1, 2]. С уменьшением технологических норм все бóльшая часть энергии потребляется не для вычислений, а для передачи и хранения данных. Уже при технологических нормах 90 нм на тактовой частоте 1 ГГц 64-разрядный блок операций с плавающей точкой занимает площадь менее, чем 1 мм^2, 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.

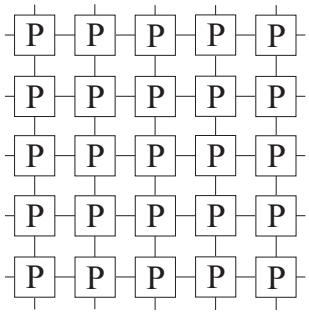

Рис. 1. Структура кристалла

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 приемов 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, включающего в себя 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.

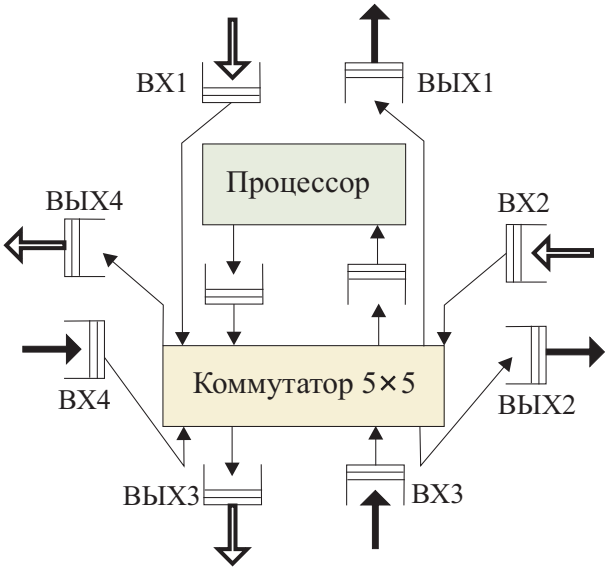

Рис. 2. Процессорное ядро

Возможны и другие варианты, а не только приведенный на рис. 2, 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 [4]. 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.

На периферии мультиядерного кристалла располагаются контроллеры блоков памяти и интерфейсные контроллеры, включая контроллеры межкристальных линий связи. Эти контроллеры мультиплексируют/демультиплексируют свободные порты линий накристальной сети по портам межкристальной сети, число которых ограничено количеством выводов кристалла. Кристалл в совокупности с блоками памяти, подключенными к его контроллерам блоков памяти, образует вычислительный узел суперкомпьютера.

Накристальная коммуникационная сеть служит для доступа процессорных ядер к контроллерам блоков памяти, внешних устройств и межкристальных линий связи. Наличие встроенных контроллеров памяти позволяет реализовать общую разделяемую всеми ядрами память суперкомпьютера. Каждое ядро имеет локальные кэш-памяти команд и данных, а через накристальную и межкристальные сети получает доступ ко всем блокам разделяемой памяти. При этом возможно программное разбиение глобального адресного пространства на области, объявляемые локальными для каждого процессорного ядра.

Таким образом, ведущими тенденциями развития архитектур кристаллов служат:

1) повышение тактовой частоты за счет уменьшения области кристалла, занимаемой процессорным ядром, в которой должен синхронно распространяться тактовый сигнал;

2) увеличение уровня параллелизма кристалла за счет реализации в нем совокупности (десятков, сотен и более) процессорных ядер вместо одного процессора;

3) введение в процессорные кристаллы контроллеров памяти, позволяющих строить суперкомпьютеры с разделяемой распределенной памятью большого объема путем объединения коммуникационной сетью локальных блоков памяти узлов суперкомпьютера, при этом с целью снижения трафика в кэш-памяти процессорных ядер кристаллов узлов допускается кэширование только данных из локальных блоков памяти узлов.

Уже имеются реализации существенно многоядерных кристаллов, например Intel Teraflops Research Chip [5] и Tile 64 фирмы Tilera [4]. Эти кристаллы содержат порядка сотен процессорных ядер и имеют пиковую производительность порядка триллиона операций в секунду.

Итак, использование многоядерных кристаллов позволяет преодолеть ограничения роста тактовой частоты и создать в ближайшие годы суперкомпьютеры, имеющие порядка 10 миллионов процессорных ядер, которые способны в совокупности достичь производительности 10^{17} flops и более. Однако современные параллельные суперкомпьютеры, имеющие порядка 10000 процессорных ядер, показывают низкую реальную производительность при вычислениях с интенсивным доступом к памяти по адресам, определяемым в ходе выполнения программы. Если 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 по адресам, определяемым в ходе вычислений.

3. Способы сокращения простоя процессора в ожидании доступа к памяти. Разрыв в быстродействии процессоров и блоков памяти существовал всегда и постоянно имеет тенденцию к увеличению: тактовая частота процессоров растет с темпом 40%, а быстродействие схем динамической памяти только 9% в год. В этих условиях исторически первым способом преодоления указанного разрыва стало использование временно́й и пространственной локализации кодов программ и обрабатываемых данных. В процессоры были введены механизмы виртуальной адресации и кэш-памяти, основанные на неявном или явном со стороны пользователя предсказании адресов доступа к памяти. В неявном предсказании используется предположение о последовательности адресов доступа к памяти или более сложные механизмы, например таблицы истории обращений к памяти, предсказания ветвлений и др. Эффективное использование этих механизмов потребовало от пользователя формирования кода программ и размещения данных с учетом их локализации. Со временем с этой работой стали успешно справляться компиляторы. При явном предсказании пользователь управляет механизмами виртуальной адресации и кэш-памяти, например программируя замещение данных в кэш-памяти так, чтобы необходимые данные оказывались в ней в момент обращения к ним. Однако как явное, так и неявное предсказание не дают эффекта при доступе к памяти по адресам, определяемым в ходе вычислений. 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: использование этих механизмов целиком остается на усмотрение пользователя.

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 создается 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 ее 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, $A \bmod N = i$, где N — количество блоков памяти. Если порядок обращений к памяти произвольный, то может быть применено скремблирование, размещающее данные по блокам памяти определенным псевдослучайным образом, чтобы увеличить вероятность последовательных обращений в разные блоки.

Вычислительная практика показывает, что интенсивность потока обращений к памяти, создаваемая аппаратурой и компиляторами мультискалярных процессоров и процессоров с архитектурой VLIW (Very Long Instruction Word), 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: пользователи должны подготавливать программы как большую совокупность тредов, но не традиционных POSIX-тредов — `pthreads` [6], а "легких" тредов [7–9]. Существующие средства программирования тредов, например стандартных POSIX-тредов, используют для взаимодействия тредов примитивы ОС. Эти примитивы весьма затратны по времени выполнения и способны обеспечить взаимодействие не более сотни тредов, тогда когда для достижения петафлопсного уровня производительности необходимы миллионы и более тредов.

Легкие 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` [7, 8]. 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 ее выполнением значение `FE` бита слова памяти, в которое будет внесена 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 др., а также механизмов синхронизации, весьма затратных по времени их определения и исполнения при миллионах тредов. Кроме того, применение неделимых (атомарных) последовательностей команд, вычисляющих с использованием упомянутых выше разделяемых переменных 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 легких тредов. Поэтому синхронизация на базе `FE` бита позволит выдавать максимально возможное в исполняемой программе количество обращений к памяти, генерируемых легкими тредами, и параллельно выполнять эти доступы в расслоенной памяти.

Для выполнения на традиционных процессорах программ, созданных на базе модели легких тредов, синхронизация которых реализуется `FE` битами, в [9] в рамках процессов Unix и POSIX `pthreads` разработана библиотека для порождения и управления легкими тредами, названными `qthreads`. В рамках этой библиотеки `FE` биты эмулируются хэш-таблицей. На уровне `qthread` API для поддержки локальности легких 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 с применением `libc qsort ()`.

Очевидно, что введение в модель параллельной программы легких тредов с контролируемой пользователем привязкой тредов к сегментам разделяемой памяти требует от пользователя дополнительных усилий по сравнению просто с представлением алгоритма на традиционном языке программирования. Однако легкие треды необходимы для эффективной работы памяти в поточном режиме, и пока нельзя рассчитывать на их формирование компилятором. Отметим, что эффективное исполнение легких тредов и средств их синхронизации требует расширения `FE` битом слов блоков памяти, а также модификации команд процессорных ядер, используемых для обращения к памяти.

4. Потоковое программирование как способ сокращения числа обращений к памяти. Кроме сокращения времени выполнения доступа в память еще одним приемом повышения производительности служит потоковое исполнение программ, обеспечивающее уменьшение количества обращений в память за счет непосредственной передачи операндов между процессорными ядрами, минуя проме-

жуточное хранение операндов в памяти. Следует уточнить, что речь идет о потоковых программах специального класса, называемых также систолическими или волновыми программами, в которых для заданного графа межъядерных 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 легких тредов, протекающих в памяти ядра.

В [3] предложено создавать параллельные программы, межъядерные потоки в которых программируются на основе 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 межъядерных связей и алгоритмом нумерации ядер, приписывающим каждому ядру уникальный номер из диапазона $0, \ldots, N-1$, где N — количество ядер в подсистеме. Алгоритм нумерации должен позволять по номеру ядра, на котором он исполняется, вычислять для любого номера i, $i \in \{0, \ldots, N-1\}$, выходное направление, ведущее к ядру с номером i. При этом если номер ядра равен 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 [2], для всех номеров ядер, кроме собственного номера, должно выдаваться одно и то же направление, ведущее к коммутатору. Для распределенных коммутаторов, например для решетки процессорных ядер, эти направления могут принимать значения 1, 2, 3, 4.

На рис. 3 показаны следующие типы графов: линейка, кольцо, дерево и решетка, 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 решетка — 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 T_j, N — количество ядер в подсистеме), можно определить направление передачи к каждому ядру подсистемы.

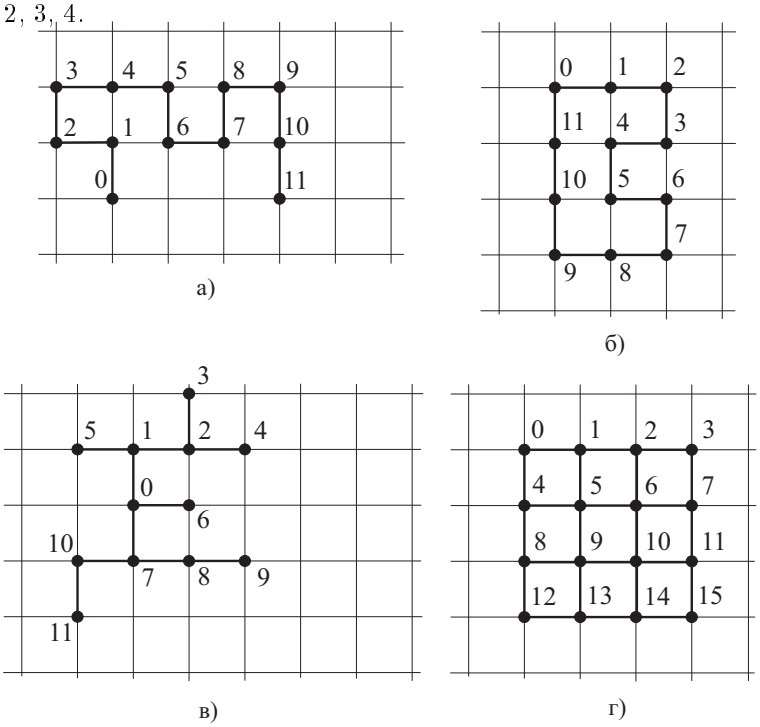

Рис. 3. Типы графов: а) линейка, б) кольцо, в) дерево, г) решетка

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, что 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 с учетом его окружения.

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.

5. Заключение. Предлагаемая модель программирования с программной настраиваемостью структуры для формирования специфичных для реализуемого алгоритма межъядерных потоков данных и мультитредовой на базе легких 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 [10] с локальными передачами данных и команд между процессорными ядрами по принципу близкодействия по каналам "точка–точка". Однако вопрос о том, следует ли создавать специальный язык программирования для таких систем или ограничиться библиотеками для работы с легкими тредами и окружениями, остается 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.

Хотя исполнение программ, созданных на основе предлагаемой модели, возможно и на процессорных ядрах традиционной архитектуры, для повышения эффективности их выполнения требуется введение в процессорные ядра многоядерных кристаллов механизмов управления глобальным адресным пространством, порождения и уничтожения легких тредов и их синхронизации на базе расширения слов памяти FE битами.

СПИСОК ЛИТЕРАТУРЫ

1. *Евреинов Э.В., Косарев Ю.Г.* Однородные универсальные вычислительные системы высокой производительности. Новосибирск: Наука, 1966.
2. *Dally W. et al.* Merrimac: supercomputing with streams SCT03 // Proc. of the 2003 ACM/IEEE Conf. on Supercomputing. Phoenix, 2003.
3. *Корнеев В.В.* Архитектура вычислительных систем с программируемой структурой. Новосибирск: Наука, 1985.
4. *Wentzlaff D. et al.* On-chip interconnection architecture of the tile processor // IEEE Micro. September–October. 2007. 15–31.
5. *Hoskote Y. et al.* A 5-GHz mesh interconnect for a teraflops processor // IEEE Micro. September–October 2007. 51–61.
6. Institute of Electrical and Electronics Engineers // IEEE Std 1003.1-1990: Portable Operating Systems Interface (POSIX.1). 1990.
7. *Li S. et al.* A heterogeneous lightweight multithreaded architecture // Parallel and Distributed Processing Symposium (IPDPS-2007). Long Beach, 2007.
8. *Kogge P. et al.* Computer systems with lightweight multi-threaded architectures. Patent US 2007/0198785 A1, August 23, 2007.
9. *Wheeler K. et al.* Qthreads: an API for programming with millions of lightweight threads // Proc. of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS-2008). Miami, 2008..
10. *Tiffolli T., Margolus N.* Cellular automata machines. Cambridge: MIT Press, 1987.

Поступила в редакцию
23.06.2009