МНОГОЯДЕРНЫЕ ПРОЦЕССОРЫ ИЗМЕНЯТ НАШЕ ПРЕДСТАВЛЕНИЕ О ПРОГРАММИРОВАНИИ

В.Е. Зюбин
Институт автоматики и электрометрии СО РАН
Россия, 630090, Новосибирск, просп. акад. Коптюга, 1
E-mail: zyubin@iae.nsk.su

У нас есть хорошая новость –
транзисторов будет много!
С форума IDF-2005, Новосибирск, апрель 2005.

Программирование разделено на две практически непересекающиеся области, два пространства, две Вселенные. Программирование высокоскоростных вычислений суперЭВМ и создание программ для ПК общего назначения. Их мирное сосуществование продолжалось в течение десятилетий. Специалисты по высокопроизводительным вычислениям свысока смотрели на происходящее в области ПК, а компьютерный мэйнстрим считал проблемы физического параллелизма вечным уделом малочисленных групп, финансируемых правительством и министерством обороны.

Гром грянул в апреле, когда одновременно и Intel, и AMD заявили о начале новой эры для ПК и приступили к поставке двуядерных процессоров. В планах – производство микросхем с сотнями ядер на одной подложке. Как это повлияет на программирование? – вопрос, на который невозможно ответить без четкого осознания отличий между физическим и логическим параллелизмом.

Закон Мура как он есть

19 апреля этого года фирма Intel праздновала 40-летие закона Мура, который вопреки расхожим слухам и домыслам крайне прозаичен: «Количество транзисторов на плате процессора увеличивается вдвое каждые два года». То, что на протяжении длительного времени рост числа транзисторов в чипах сопровождался увеличением производительности отдельно взятого процессора, вещь приятная, но, как выяснилось, вовсе не обязательная.

Десятилетия программирование для персональных компьютеров существовало в оранжерейных условиях. Работа над оптимизацией программы стояла если не на последнем, то далеко не на первом месте. Больше внимания уделялось процессу групповой разработки, объектным паттернам программирования, сокращению времени создания программы. Считалось, что скорость работы программы растет независимо от разработчика, просто в силу постоянного роста вычислительной мощности процессора, ускорения обмена данными с памятью, винчестером, видеокартой и другой периферией.

В действительности же происходило следующее: примерно с 1970 года до 1985 года производительность процессоров росла в основном за счет совершенствования элементной базы и увеличения тактовой частоты процессора. Затем, вплоть до 2000 года, основную роль стали играть архитектурные улучшения: конвейеризация, специализация, суперскалярность, спекулятивные вычисления, кэширование, увеличение разрядности.

И повышение тактовой частоты, и перечисленные архитектурные нововведения не влияли на процесс проектирования программ. Программы для ПК продолжали оставаться прозрачными, линейными и преимущественно объектно-ориентированными. В компиляторах предлагалось опционально указывать тип процессора целевой вычислительной платформы, но даже без перекомпиляции старые программы исполнялись на новых платформах значительно быстрее. И программисты жили в уверенности, что такое положение будет продолжаться бесконечно.

В 2001 прилетела первая «ласточка»: был исчерпан ресурс на повышение тактовой частоты процессора. Для рядового пользователя это прошло незамеченным, т.к. вывод процессора с тактовой частотой 3+ ГГц в массовое производство растянулся на несколько лет.

К 2005 году, с одной стороны, был освоен серийный выпуск 3-х ГГц процессоров, а, с другой стороны, были в основном исчерпаны и ресурсы на архитектурные совершенствования отдельно взятого процессора.

В апреле 2005 года фирмы Intel и AMD практически одновременно анонсировали начало продаж двуядерных процессоров для персональных компьютеров – по сути двух процессоров на одной подложке. Золотой век персонального компьютинга закончился. Теперь забота о повышении скорости исполнения программ целиком и полностью легла на плечи кодировщика.

Приятное обстоятельство в том, что программирование на нескольких процессорах – проблема не новая. В течение десятилетий этим занимались при организации высокопроизводительных вычислений. Однако такое программирование требует от разработчика специальной подготовки, особого образа мышления и высокой квалификации. И это очень неприятный момент.

В статье кратко рассматривается история, основные признаки и проблемы физического и логического параллелизма, обсуждаются серьезные последствия архитектурных нововведений, а также пути сглаживания возникающих противоречий.

В погоне за супервычислителем

В энциклопедиях сообщается, что программирование создавалось и развивалось как элитарная научная дисциплина, которая изучала способы составления, проверки и улучшения программ для ЭВМ. Под программой понималась упорядоченная последовательность действий для ЭВМ, реализующая алгоритм решения некоторой задачи. А алгоритмом же считалось всякое точное предписание, которое задаёт вычислительный процесс, направленный на получение полностью определяемого результата по исходным данным.

Следует отметить, что речь идет исключительно о вычислительных алгоритмах. И действительно, подавляющее число задач на заре программирования были именно такими: рассчитать траекторию полета ракеты, по метеорологическим данным предсказать погоду на завтра, найти экстремум функции (оптимальное решение) в экономической задаче, оценить требуемую мощность электромагнита для циклотрона или вычислить определенный интеграл с точностью до десятого знака.

Второй примечательный нюанс – энциклопедия умалчивает одно чрезвычайно важное правило, действующее в те времена. Правило настолько очевидное, что о нем даже не упоминалось, а именно – чем быстрее получено

решение, тем лучше. И дело даже не в том, что ожидание момента, когда машина «выплюнет» перфокарты с ответом, страшно нервировало программистов. Результат вычислений ждали конкретные потребители – ученые, инженеры, экономисты. Скорость вычислений определяла развитие народного хозяйства.

В случае с предсказанием погоды на завтра результат, полученный с опозданием, вообще абсолютно бесполезен. Задержка с дешифровкой важной информации, полученной при радиоперехвате, – это вопрос национальной безопасности. Возможность обсчитать ядерный взрыв – важный военно-политический козырь. Эти соображения послужили стартовым сигналом к началу гонки на межгосударственном уровне, целью которой было построить самую высокопроизводительную ЭВМ – суперкомпьютер.

Высокоскоростные вычисления напрямую связаны с вопросами национальной безопасности, и денег на это не жалели.

Увеличение производительности компьютеров ведется за счет совершенствования элементной базы и архитектурных решений. Оценить роль каждого из этих компонентов можно на простом примере. Один из первых компьютеров мира EDSAC (1949 г., Кембридж) имел время такта 2 микросекунды и производительность 100 арифметических операций в секунду. Спустя 50 лет в суперкомпьютере Hewlett-Packard V2600 один вычислительный узел имеет время такта 1,8 наносекунды при пиковой производительности около 77 миллиардов операций в секунду. Производительность компьютеров выросла минимум в 700 000 000 (семьсот миллионов) раз. Несложно подсчитать, что увеличение быстродействия за счет снижения времени такта составило всего лишь 1000 раз. Откуда же взялось остальное? Ответ очевиден – за счет новых архитектурных решений. Основное место среди них занимает принцип параллельной обработки данных, основанный на идее одновременного (параллельного) выполнения нескольких действий [1].

Физический параллелизм

Параллельная обработка имеет две разновидности: конвейерность и собственно параллельность. В том и другом случае предполагается архитектурное вычленение из системы отдельных физических компонентов.

В случае конвейерности идея заключается в разбиении операции на последовательные этапы длительностью в такт и реализации каждого из них отдельным физическим блоком. Если организовать работу таких блоков в виде конвейера – каждый блок, выполнив работу, передает результат вычислений следующему блоку и одновременно принимает новую порцию данных, – то получится очевидный выигрыш. Выигрыш выражается формулой:

$\frac{n \times d}{n + d}$, где

n – это количество выделенных этапов в операции, а d –количество обработанных операндов. В числителе – вариант выполнения операций без конвейеризации, а в знаменателе – вариант с конвейеризацией (после прохождения первого операнда через конвейер каждый следующий операнд обрабатывается за один такт).

Идея «честного» распараллеливания еще проще и демонстрируется на простой аналогии: если одному рабочему требуется 10 часов на изготовление 10 деталей, то 10 рабочим для этого требуется всего час.

К сожалению, несмотря на мощь и простоту идей физического параллелизма, установка 10 вычислительных модулей вместо одного не означает десятикратного увеличения вычислительной мощности системы на практике. Возможность распараллеливания определяется не только архитектурой вычислительной платформы, но и природой исходной задачи. Если в задаче нельзя выделить участки, допускающие параллельную обработку, или массив однотипных операндов, для конвейеризации, то сократить время ее выполнения за счет обсуждаемых приемов не удастся.

В общем случае это обстоятельство описывается законом Амдала, который ограничивает максимально достижимое увеличение производительности системы S следующей формулой:

$S \le \frac{1}{f + (1-f)/p}$, где

f – доля машинных кодов, которые не поддаются распараллеливанию;

p – число параллельно работающих процессоров в системе.

Важен также и знак «меньше или равно», который означает, что в системе присутствуют и дополнительные накладные расходы на организацию параллельной работы системы: загрузка системы, синхронизация вычислений, распределение параллельных участков – все это отдаляет практически наблюдаемые значения ускорения от идеала.

Но в целом идеи наращивания мощности за счет физической параллелизации вычислений широко используются на самых разных уровнях внутренней архитектуры суперкомпьютеров, объединяющих тысячи процессоров и выполняемых в рамках уникальных дорогостоящих проектов. Несмотря на единичность и закрытость, такие проекты затрагивают и рядового пользователя ПК. Архитектурные улучшения на уровне отдельного процессорного ядра легко поддается коммерциализации, и при массовом тиражировании цена решений, заимствованных у мега-систем, значительно снижается.

При внедрении в компьютерный мэйнстрим особенно привлекательны методы повышения скорости вычислений, не влияющие на глобальную структуру программ. Кроме уже рассмотренных конвейеризации и физического распараллеливания, использовались следующие методы:

- специализация (появление внутри процессоров блоков оптимизированных под определенный вид вычислений – например, математических сопроцессоров);
- кэширование («подтягивание» к вычислительному ядру данных из ОЗУ);
- спекулятивные вычисления (просмотр кода на несколько шагов вперед и проведение подготовительных вычислений).

Суперкомпьютеры и кластеры

На настоящий момент (апрель, 2005) наиболее мощный суперкомпьютер IBM Blue Gene/L показывает результат 135,5 триллионов операций с плавающей

точкой в секунду. Он состоит из 32 стоек, каждая из которых содержит по 1024 двуядерных процессора PowerPC, что позволяет рассматривать его как систему из 65 536 процессоров. Такие габариты определяют высокую стоимость суперкомпьютеров – от 15 млн. долларов, что практически исключает их приобретение мелкими и средними частными компаниями или научными учреждениями на бюджетные средства. А такая потребность имеется. Помимо военной области, высокоскоростные вычисления активно используются для обсчета сложных задач и моделирования в аэродинамике, нефте- и газодобыче, фармакологии, геологоразведке, синтезе новых материалов, киноиндустрии, финансовой сфере и т.д. и т.п.

Поиск дешевой альтернативы суперкомпьютерам привел к идее кластерных систем, когда параллельная система требуемой производительности собирается из компьютеров общего назначения (возможно, уже существующих в организации), объединяемых сетью. Кластерная система, по сути, убивает сразу даже не двух, а трех «зайцев»: а) утилизируются растрачиваемые впустую вычислительные мощности, б) по-прежнему обеспечивается штатная офисная работа сотрудников компании, в) достигается низкая стоимость системы за счет использования серийно изготавливаемых компонентов.

Остается только «правильно» разбить задачу на параллельные части и распределить их по процессорам.

Особенности программирование систем физического параллелизма

Простота идеи программирования физически параллельных систем на практике выливается в целый набор сложных методик и приемов, приводящих к разнообразным негативным последствиям.

В первую очередь, требуется обеспечить достаточный уровень параллельности алгоритма. Условие Бернштейна определяет признаки, при которых возможно распараллеливание. Процессы P и Q могут быть обработаны параллельно, только если:

а) результат выполнения P не служит входными данными для Q и наоборот,

б) выходные данные P и Q не являются одновременно входными данными для любого другого процесса.

Если результат P используется в Q, то должен исполняться перед Q. Если же результат P и Q записывается в одну ячейку, то их параллельное исполнение приведет к неопределенности.

При вычленении параллельных участков, как правило, приходится придавать алгоритму специальную форму.

Например, нахождение суммы массива из четырех элементов в последовательной форме предполагает три действия (рис.1).

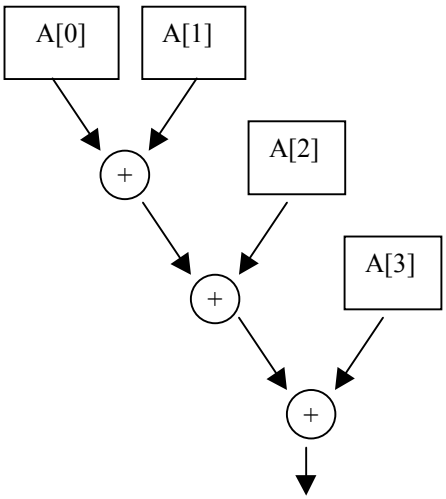

Рис.1. Последовательное суммирование четырех элементов массива.

Такое суммирование привычно записывается на языке Си:

```
<код>
for ( i=0, Sum=0;i<4;i++) {
    Sum += Array[i];
}
</код>
```

При распараллеливании на первом шаге одновременно суммируются соседние четные и нечетные элементы массива, а на втором – суммируются результаты, полученные на первом шаге (рис. 2).

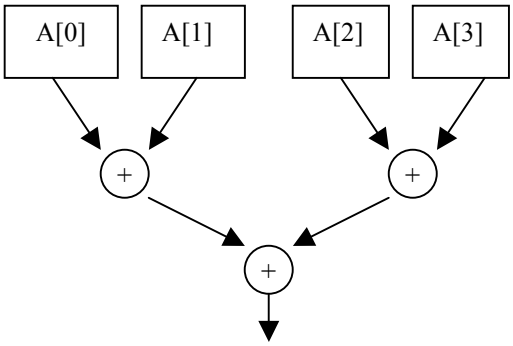

Рис.2. Параллельное суммирование 4-х элементного массива.

При этом важно отметить, что, несмотря на более быстрое получение результата (два действия вместо «классических» трех), преобразование привело к усложнению (в языке Си вообще отсутствуют выразительные средства, чтобы описать рассмотренный параллельный вариант).

В общем случае приведение алгоритма к форме, обеспечивающей сокращение времени вычислений, означает отход от формы, обеспечивающей наиболее наглядное представление. Программы, адаптированные для исполнения на параллельных архитектурах, теряют в наглядности.

Вычленение параллельных участков связано с другой важной проблемой, решаемой при программировании высокоскоростных вычислений, – обеспечить

равномерную загрузку процессоров системы, т.н. балансировку загрузки (load balancing).

Неравномерность загрузки приводит к тому, что один из процессоров, выполнив менее ресурсоемкое вычисление, будет ждать окончания вычислений на других процессорах, тем самым, растрачивая свои ресурсы впустую. Эта проблема может усугубляться неоднородностью системы, которая по понятным причинам характерна для кластеров.

Балансировка загрузки производиться разными способами:

- по запросу процессора, имеющего слишком много задач в очереди на исполнение;
- по запросу от простаивающего процессора;
- по командам некоторого выделенного процессора, реализующего механизм распределения загрузки;
- по алгоритмам, комбинирующим три вышеперечисленных способа.

В общем случае балансировка обеспечивается разбиением алгоритма на достаточно мелкие части.

Однако при разбиении на части необходимо учитывать другой нюанс многопроцессорных систем – затраты на обмен информацией и синхронизацию. Если выполнение задачи занимает 1 микросекунду, а передача исходных данных на другой узел и прием результата 2 микросекунды, то распараллеливание будет снижать вычислительную мощность системы.

Программист параллельной задачи, как Одиссей, всегда ходит между Сциллой и Харибдой: в многопроцессорной системе разбиение на слишком крупные части не позволяет равномерно загрузить процессоры и добиться минимального времени вычислений, а излишне мелкая «нарезка» означает рост непроизводительных расходов на связь и синхронизацию. Если параллельно исполняемые части «велики» по размеру, то в этом случае говорят о «крупнозернистом параллелизме» (coarse grain parallelism). Противоположный случай называют «мелкозернистым параллелизмом» (fine grain parallelism). «Зернистость» параллелизма, характеризующую размер и время исполнения независимых частей, вычленяемых в алгоритме, называют гранулярностью.

При проектировании параллельного алгоритма необходимо учитывать не только индивидуальные характеристики вычислительных элементов кластерной системы, но и их общее количество (выполнить параллельно десять задач невозможно, если в наличие только девять компьютеров), и пропускную способность каналов связи.

Таким образом физическому параллелизму присуща зависимость алгоритма от топологии вычислительной платформы.

Создание действительно эффективного алгоритма связано с большими трудовыми затратами на поиск специфической структуры алгоритма, оптимальной для данной топологии целевой системы. Найденная структура обеспечит минимальное время получения результата, но, как правило, имеет весьма неприглядную для сопровождения форму и, скорее всего, будет неэффективна для другой конфигурации. Более суровое следствие зависимости структуры алгоритма от вычислительной платформы – тотальная непереносимость не только исполняемых кодов, но и самого исходного текста.

Другая неприятность заключается в том, что нахождение глобальной структуры алгоритма, обеспечивающей минимальное время вычислений для заданной платформы, не автоматизируется. Программист должен определять ее вручную. Некоторая автоматизация распараллеливания алгоритма может быть

достигнута только на локальном уровне, в редких случаях явного указания числа итераций в цикле, при отсутствии внешних вызовов и т.п. Чаще же оценка эффективности распараллеливания носит вероятностный характер. В языках, предназначенных для физически параллельных систем, присутствует целый ряд дополнительных инструкций и прагм, необходимых для адекватной трансляции программ.

Весьма обнадеживающими выглядят работы по созданию языковых средств и методик обеспечения переносимых параллельных программ. Например, в рамках проекта «Пифагор» (школа профессора А.И. Легалова) [2]. Основная цель проекта – обеспечить разумный компромисс между быстродействием алгоритма и приемлемым уровнем его сопровождаемости (эволюционной расширяемости, переносимости, платформенной независимости). Однако этот подход имеет свои особенности – программирование предполагает использование функционального языка, – подход, как минимум, крайне непривычный для большинства практикующих программистов.

Продолжая говорить о специфических нюансах физического параллелизма, следует упомянуть вопросы надежности. Программист, пишущий физически параллельную программу, должен знать о таких вещах, как:

- взаимные блокировки параллельных участков (deadlock);
- несинхронный доступ, или гонки;
- возможность зависания параллельных участков;
- опасность использования сторонних процедур и библиотек;
- набор специализированных средств отладки физически параллельных программ;
- нелокальный характер ошибок;
- динамический характер ошибок и, как следствие, влияние средств отладки программ на корректность исполнения программы.

SISD параллелизм

По распространенной классификации Флинна системы физического параллелизма делятся на четыре категории:

- **Один поток инструкций/Несколько потоков данных (SIMD)**, при котором одна и та же операция производится несколькими процессорами над множеством данных, (например, поэлементное сложение двух массивов или векторов);
- **Несколько потоков инструкций/Один поток данных (MISD)**, при котором разные операции производятся над одним и тем же операндом, (очень редкий вид параллелизма, встречающийся, например, при попытке распараллелить процесс нахождения делителя некоторого числа);
- **Несколько потоков инструкций/Несколько потоков данных (MIMD)** – наиболее универсальный вид параллелизма, когда каждый из процессоров выполняет свое уникальное задание;
- и, наконец, **Один поток инструкций/Один поток данных (SISD)**, который в рамках физического параллелизма не рассматривается и даже не воспринимается как параллелизм. В первую очередь потому, что сложно придумать подходящий осмысленный пример такому экзотическому случаю [3].

Заметив, что параллельная обработка одного и того же операнда разными процессорами бессмысленна с точки зрения повышения производительности, можно, тем не менее, предложить вполне жизнеспособное обоснование для использования SISD многопроцессорных систем – повышение надежности вычислений на ненадежной элементной базе, когда истинный результат вычислений определяется по мажоритарному принципу.

Другой широко распространенный случай из практики промышленной автоматизации – т.н. «горячее» резервирование, когда один основной процессор формирует действительные управляющие воздействия на объект автоматизации, а несколько других абсолютно идентичных ему процессоров работают параллельно и подменяют основной в случае его отказа.

То, что такое использование параллельности игнорируется «классическим» параллелизмом, обусловлено исходной ориентацией физического параллелизма на достижение максимальной вычислительной мощности. Задачи, лежащие в области других интересов, попросту игнорируются.

Операционные системы

Точно также игнорируется и вопрос использования SISD параллелизма для одного процессора. И действительно. Попытка разбить выполнение некоторой задачи на части, допускающие параллельное исполнение, и затем выполнить их в рамках однопроцессорной архитектуры, приведет только к потере производительности. С точки зрения физического параллелизма, такой подход не имеет абсолютно никакого смысла. Смысл такого распараллеливания обнаруживается, если посмотреть на ситуацию с точки зрения удобства программирования.

«Разделяй и властвуй». Старая поговорка имеет для программистов форму закона. Сложность современных программных систем может быть преодолена только путем разбиения проблемы на множество обособленных частей. И наоборот. Программные комплексы произвольной сложности могут строиться только на основе независимых или, в крайнем случае, слабо зависимых компонентов. А это также подразумевает максимальную обособленность частей с целью минимизации перекрестных связей, что необходимо для эффективной организации групповой разработки.

В случае, когда выделение параллельных (а вернее, независимых) сущностей производится исходя из соображений удобства программирования, – в первую очередь, с целью снизить сложность описания системы, – говорят о логическом параллелизме.

Кстати, успех операционных систем во многом обусловлен именно этим – возможностью конфигурирования функциональной среды отдельно взятого компьютера из набора параллельно исполняемых компонентов-задач. При вызове менеджер задач в любой многозадачной операционной системе покажет около десятка процессов, выполняющих параллельно незаметную, но необходимую работу. В дополнение к этим процессам пользователь может запустить проигрыватель mp3-файлов, антивирусный мониторинг, редактор, почтовую программу и браузер… причем открытых для редактирования файлов и окон браузера может быть несколько. Попытка связать такую функциональность в целое и описать единым блоком однозначно приводит к комбинаторному взрыву сложности.

Логический параллелизм на уровне операционных систем называется многозадачным логическим параллелизмом. Планировщик – выделенная задача операционной системы – занимается переключением процессора с одной задачи на другую. Алгоритмы таких переключений могут быть как примитивными, так и весьма сложными. Самый простой – алгоритм разделения по времени. Планировщик работает с очередью задач и активизируется по прерываниям от таймера с некоторым заданным периодом (обычно это десятки миллисекунд). После активизации он сохраняет контекст текущей задачи, ставит ее в конец очереди, восстанавливает контекст следующей задачи из очереди и запускает ее на исполнение.

Поскольку задачи различны по исполняемым функциям и ресурсоемкости, возникает проблема, как распределить вычислительную мощность процессора соразмерно задачам. В этом случае базовый алгоритм может усложняться. Например, механизмом приоритетов, когда задачам присваиваются приоритеты и при смене задач управление передается наиболее приоритетной. В системе может быть предусмотрено динамическое изменение приоритетов. Планировщик может вызываться не только по прерываниям от таймера, но и от других источников прерываний. Вызов планировщика может быть предусмотрен и по требованию активной задачи, после выполнения необходимых высокоприоритетных операций.

Примечательно, что при увеличении частоты вызова планировщика с алгоритмом «разделение по времени» многозадачный логический параллелизм практически неотличим от физического MIMD-параллелизма. Единственный, но очень неприятный эффект: чем чаще активизируется планировщик, тем выше накладные расходы на организацию параллелизма. Вплоть до ситуации, когда при некоторой граничной частоте все вычислительные ресурсы системы занимает задача планирования.

При крайне простой постановке задача минимизации накладных расходов связана с большими сложностями. В примитивных алгоритмах планирования отсутствует гибкость. Сложный алгоритм планирования, обеспечивающий необходимую гибкость, связан с увеличением накладных расходов и непредсказуемостью динамического поведения системы [4]. Накладные расходы на организацию нетривиальных алгоритмов планирования, как правило, сильно зависят от числа обслуживаемых задач. Сходство систем многозадачного логического параллелизма с системами физического параллелизма приводят к схожим и весьма нежелательным эффектам, вплоть до коллапса системы.

Поиск алгоритма идеального планировщика сродни задаче поиска панацеи и обычно ведется в рамках задачи построения операционных систем реального времени (т.н. ОС РВ). Обсуждения тех или иных подходов к планированию, а также отличительных особенностей разных ОС, – это самые горячие темы сетевых форумов и конференций.

Необходимая ремарка, которую следует сделать по этому поводу, – эффективные алгоритмы планирования полезны не только в операционных системах, позиционируемых на рынке как ОС РВ, но и в ОС т.н. «общего назначения». Поэтому наработки, возникающие при решении задачи планирования, адаптируется на всех ОС без исключения. Кстати, происходит и обратное заимствование идей от ОС общего назначения к ОС РВ. Примером может служить появление диск-своппинга в ОС QNX версии 6. Наблюдается явная тенденция к унификации функциональности разных ОС.

Преимущества мелкозернистого логического параллелизма

Задача с уникальным набором данных и функций достаточно тяжеловесна для вызова и обслуживания. Многозадачный логический параллелизм предполагает небольшое количество задач (обычно в пределах одного десятка).

Между тем, существуют области, в которых необходим мелкозернистый логический параллелизм. В серверах баз данных, новостных интернет-порталах, мобильных сервисах, обслуживающих распределенные системы из нескольких сотен тысяч абонентов, требуется механизм независимого обслуживания тысяч одновременно поступающих запросов. В таких системах требуется не только не пропустить поступивший запрос, но и обслужить его, возможно, организовав интерактивный диалог, контроль достоверности, перезапрос информации в случае ошибки. Мелкозернистый логический параллелизм также чрезвычайно эффективен в задачах управления сложным промышленным оборудованием.

Необходимость отражать параллелизм процессов, протекающих на объекте управления, требует создавать по меньшей мере сотни параллельно исполняемых процессов. Например, эквивалентное монолитное решение в виде одного процесса для практической задачи управления выращиванием монокристаллического кремния [4] приводит к рассмотрению 10^{30} (sic!) различных ситуаций.

Сжатие информации достигается при логическом параллелизме за счет упрощения описания комбинаций независимых случаев. Если мы имеем L независимых участков кода, и i-й участок имеет N_i значимых ситуаций, то описание алгоритма как набора независимых частей – это описание суммы ситуаций, в то время как монолитное описание – это описание произведения ситуаций. Редукция сложности за счет распараллеливания выражается следующей формулой:

$$R = \frac{\prod_{i=1}^{L} N_i}{\sum_{i=1}^{L} N_i},$$

Невообразимое число ситуаций (10^{30}) может быть получено всего лишь из сотни параллельно исполняемых компонентов с двумя состояниями. В то время как независимое описание алгоритма – это рассмотрение всего 200 уникальных случаев. Редукция для рассматриваемого примера – $5 \cdot 10^{27}$ раз (!). Разумеется, приведенная формула редукции сложности описания не учитывает увеличения объема кода, возникающего при монолитном подходе.

Многопоточный логический параллелизм

Очевидные преимущества мелкозернистого логического параллелизма, недостижимого в рамках многозадачности, заставляет искать пути снижения накладных расходов. Известные решения для операционных систем – это легковесные процессы (light-weight process), – например, в Sun OS 5.x, – облегченный вариант задач, и потоки (thread), состояние которых полностью характеризуется значениями регистров-указателей на код и на стек.

Переключение процессора на поток минимизировано, таким образом, до операций сохранения/восстановления этих указателей. Планировщик по-прежнему присутствует и активизируется по прерываниям от таймера.

Необходимо отметить, что переключение потоков по времени не всегда удобно и приводит к непроизводительным простоям процессора. Если в некотором потоке для продолжения вычислений требуется внешнее событие (отклик оборудования, реакция абонента на запрос, подтверждение успешной передачи сообщения), то поток честно тратит отведенное ему время на ожидание:

```
<<код:>>
Step1();                        /* действие */
while (!EventOnStep1()) ;  /* ожидание реакции на действие */
Step2();                        /* следующее действие */
<</код>>
```

Если для появления реакции на действие требуется значительное время (например, событие – это реакция пользователя на запрос системы), то более приемлемой стратегий была бы передача управления после опроса.

Этот подход используется при организации многопоточности методом циклического опроса (round-robin), иногда обозначаемом термином «корпоративная многозадачность». При циклическом опросе потоки могут быть представлены просто функциями, которые опрашиваются в бесконечном цикле:

```
<<код:>>
main(){
   for (;;) {
     thread_1();
     thread _2();
     …
     thread _nn();
   }
}
<</код>>
```

Необходимость ожидания внешней реакции естественным образом задает разбиение потоков на части. Каждая из таких частей является простейшей функцией. Если присвоить каждой из таких частей число, то реализация потоков естественным образом выражается в терминах языка Си. По историческим причинам простейшие функции называют состояниями:

```
<<код:>>
thread_i(){
  switch (State_i) {
  …
  case STATE_1:
    Step1();                    /* действие */
    if (EventOnStep1()) State_i = STATE_2; /* Переход по событию */
    break;
  case STATE_2:
    Step2();                    /* следующее действие */
```

```
  …
  }
}
<</код>>
```

Таким образом, для многопоточности можно минимизировать или даже полностью исключить накладные расходы, присущие многозадачности. А простота организации делает многопоточность крайне привлекательной для организации мелкозернистого логического параллелизма с гранулярностью на уровне нескольких инструкций. Получаемый выигрыш от многопоточности компенсирует риски потенциально возможных ошибок (которые могут возникнуть, например, из-за общей области данных). Тем более, что эти риски могут быть полностью исключены за счет соответствующих механизмов.

Например, в языке Рефлекс – диалекте языка Си, предназначенном для описания алгоритмов управления, – безопасность и невозможность разрушения программы языковыми средствами обеспечены на концептуальном уровне.

В языке Рефлекс синтаксис и семантика Си были расширены понятием процесса – параллельно исполняемого участка кода, допускающего запуск, остановку, проверку своего состояния [4]. Автоматизированы рутинные операции по организации параллелизма, и пользователь может полностью сосредоточиться на собственно алгоритме. Возможность обращения по произвольному адресу исключена. Переменные языка типизированы. При таком подходе конфликты памяти попросту невозможны.

Отсутствие универсального алгоритма планирования в случае логического параллелизма может привести к необходимости использовать физический параллелизм. В задачах автоматизации «двухкомпьютерная» архитектура – это типичный случай. Алгоритм управления объектом, требующий мелкозернистый логический параллелизм, выполняется на т.н. программируемых логических контроллерах (ПЛК), например, на языке Рефлекс. Другой же компьютер, соединенный с ПЛК, предоставляет оператору полноценный мультимедийный интерфейс на базе многозадачной ОС.

Логический и физический параллелизм. Отличия и точки соприкосновения

Краткие резюмирующие характеристики физического и логического параллелизма приведены в таблице.

Сравнительные характеристики физического и логического параллелизмов

Физический параллелизм	Логический параллелизм
Цель использования	
Выполнение единичного вычислительного алгоритма за минимальное время	Удобство программирования и снижение сложности создания программ
Основной признак	
Одновременное исполнение одного алгоритма на нескольких процессорах	Независимое исполнение нескольких несвязанных или слабосвязанных алгоритмов на одном процессоре
Типовые платформы	

Многопроцессорные суперкомпьютеры, кластеры	Персональные компьютеры
Преимущества	
• Сокращение времени вычислений	• Сокращение стоимости создания программ; • Создание гибких настраиваемых программных комплексов
Недостатки	
• Высокие требования к квалификации разработчика, высокая стоимость разработки алгоритма; • Проблемное сопровождение алгоритмов (непереносимость, слабая читаемость); • Отсутствие универсального алгоритма распараллеливания; • Сложность отладки	• Ограничения на вычислительную мощность; • Непроизводительные затраты на планирование; • Отсутствие универсального алгоритма планирования

Основное отличие физического и логического параллелизма заключается в целях их использования.

Физический параллелизм – это подход, предназначенный для получения результата по входным данным за кратчайшее время.

Логический параллелизм – подход, при котором основной целью является сокращение времени разработки, а также упрощение создания, эксплуатации и, в общем случае, сопровождения программных систем.

В том и другом случае предполагается выделение участков алгоритма, допускающих независимое или слабо зависимое исполнение.

Поскольку время получения результата складывается из времени разработки алгоритма и времени расчета, то для задач физического параллелизма также крайне желательно обеспечение высоких скорости разработки и уровня сопровождаемости программ. Однако эти вопросы отходят на задний план и даже приносятся в жертву основной цели – сокращению времени получения конечного результата.

В свою очередь, в задачах логического параллелизма часто возникает желание снизить время выполнения алгоритма, например, чтобы расширить функциональность системы. Другое дело – до настоящего времени общепринятой практикой было ожидание новых решений от производителей вычислительных платформ.

Особенность физического параллелизма, обуславливающая сложность программирования, в том, что структура максимально эффективного (с точки зрения времени вычисления) алгоритма зависит от топологии вычислительной системы. Этот тезис легко демонстрируется на классическом примере сортировки. При O(N) сравнениях, необходимых для сортировки массива из N элементов, стоимость параллельных алгоритмов сортировок равна O(N^2). В то время как быстрые последовательные алгоритмы сортировок обеспечивают стоимость, равную O (N log N). Таким образом, для того, чтобы отсортировать массив из 1000 чисел на десяти вычислителях, надо разбить массив на десять частей, отсортировать их эффективным последовательным алгоритмом

сортировки, а затем параллельно слить эти части в общий список (стоимость параллельного слияния – O(*N*)) [3].

Как преодолеть кризис

К сожалению, бесплатные обеды, сопровождавшие закон Мура, закончились. Средства повышения вычислительной мощности отдельного процессора практически исчерпаны. Остались лишь небольшие резервы за счет увеличения кэшируемой памяти [5] – вещи, кстати, тоже весьма неоднозначной.

Разработки Intel в области повышения тактовой частоты пока не прекращены. Но они направлены, в первую очередь, на поиск новой элементной базы, либо альтернативной кремниевой, либо альтернативной транзисторной – и тут можно упомянуть, пожалуй, только кремниевые лазеры, работающие, впрочем, лишь в лабораторных условиях.

А поскольку путь от экспериментального образца до серийно изготавливаемого процессора занимает около десяти лет, то можно смело прогнозировать, что в ближайшее время не следует рассчитывать на технологические прорывы в области персональных компьютеров и возврат к удобной однопроцессорной архитектуре. Остается только предлагаемое на рынке «многоядерное» решение.

При этом необходимо констатировать, что архитектурные совершенствования поставили относительно однородную IT-индустрию на грань разделения на более мелкие области. Это вызвано тем, что физическое распараллеливание не работает на произвольном классе задач: увеличение разрядности процессора, использование гипер-потоковости (hyper-threading) и многоядерности могут иметь эффект, противоположный ожидаемому, т.е. после распараллеливания время исполнения программы может даже увеличиться.

Не всякая задача допускает физическое распараллеливание [5]. Целые классы задач (например, быстрое Фурье-преобразование, обработка запроса к базе данных, быстрые алгоритмы сортировок и т.п.) не получают заметного выигрыша от распараллеливания, либо не распараллеливаются в принципе.

У разработчиков появляется альтернатива – вместо изменения структуры программы уделить больше внимания локальной оптимизации кода… и до некоторого предела это будет вполне рабочим вариантом, что обусловливает рост интереса к средствам разработки, ориентированным на высокоскоростное исполнение в рамках однопроцессорной архитектуры

Возникает проблема совместимости старых, написанных для однопроцессорных архитектур, библиотек. Если библиотеки не поддерживают физически параллельное исполнение, то распараллеливание алгоритма может привести к ошибке. Не случайно Intel расширяет программистскую составляющую своей деятельности, в частности, направленную на создание безопасных библиотек. Эти библиотеки также предполагают и гибкую настройку на конкретную вычислительную платформу. По-видимому, в ближайшем будущем речь будет идти о динамически реконфигурируемых библиотеках, конечный вид которых будет определяться при загрузке задачи и выявлении топологии системы.

Не вполне ясен вопрос с объектно-ориентированным программированием, как таковым… исследования совместимости ООП с многоядерной архитектурой процессора на настоящий момент не известны. Текущий стандарт ISO на Си++

тактично обходит стороной вопросы физического параллелизма, а Intel пишет специализированные библиотеки на чистом Си.

Вполне возможно, что некоторое полезное использование многоядерной архитектуры без реинжиниринга старых программ могут обеспечить операционные системы на уровне задач. Однако в силу небольшого числа задач, такой подход имеет вполне естественные ограничения.

Будет по-прежнему сужаться область применимости специализированных ОС (т.н. ОС реального времени). Интересный факт, который необходимо отметить, – это видимые преимущества микроядерных (micro-kernel) архитектур, которые в отличие от монолитных ОС, оптимизированных для однопроцессорных систем, допускают физическое распараллеливание.

Обсуждается идея «семейного» компьютера, по сути, означающая возврат к многопользовательскому варианту вычислительных систем, отягощенному концепцией «умного» дома: мама закачивает по и-нету новый рецепт разогрева супа в микроволновой печи, дочь слушает музыку, сын играет в модный «3D-шутер», папа смотрит новостной канал веб-ТВ. Не вдаваясь в подробный анализ, можно заметить, что при обсуждении привлекательности нарисованной картины возникает много вопросов.

Создается впечатление, что в самой фирме Intel не совсем представляют себе область применимости своих нововведений. На форумах Intel, проводимых в этом году по всему свету, каждая секция начинается рассуждениями о необходимости многоядерной архитектуры, однако дальше распараллеливания обработки изображений и распознавания речи обоснование не продвигается, т.е. речь идет в основном об индустрии развлечений. Другой просматриваемый момент – Intel ищет новые техники и средства разработки. И это одна из основных целей экосистемы, создаваемой фирмой Intel, – сообщества конечных пользователей, разработчиков ПО и образовательных учреждений, объединенных вокруг Intel.

Но основной и очень щекотливый вопрос, стоящий на повестке дня: «Каким образом, с одной стороны, сохранить существующее удобство разработки и сопровождения ПО, а, с другой стороны, сохранить привлекательность новых платформ на основе физического параллелизма?» Вопрос архиважен потому, что именно быстрый рост производительности персональных компьютеров при сохраняющейся простоте функционального расширения программ определял до сего времени динамику IT индустрии. «Что дал Энди, то забрал Билл» - в этой расхожей фразе заключается главный секрет симбиоза, обозначаемого словом Wintel. Энди (Intel/производители аппаратуры) «давал» далеко не бесплатно, да и Билл (Microsoft/производители ПО), «забирая», получал существенную прибыль.

Основная схема развития IT-индустрии была следующей. С одной стороны, на рынок выбрасывалось достаточно привлекательное (well-enough), но и более требовательное к производительности персонального компьютера программное обеспечение, а с другой стороны, производители персональных компьютеров выбрасывали на рынок все более мощные обновленные решения. Межверсионная несовместимость, возникающая при этом по естественным (или почти естественным) причинам, приводила к тому, что пользователи выбрасывали свои старые ПК и ПО на помойку. Другими словами, происходил постоянный (примерно раз в 2-3 года) апгрейд персональных компьютеров, системного и прикладного ПО.

И эта идиллия закончилась. Наступила новая эра, в которую с очевидностью попадут далеко не все из существующих подходов.

Поиск новых областей применения для предлагаемых архитектур персональных компьютеров приобретает крайнюю степень актуальности.

А главной тенденцией, ожидаемой в программировании ближайшего времени, станет разработка новых языков и техник, совмещающих удобство разработки и физический параллелизм исполнения. Это может быть достигнуто только за счет исключения вопросов, связанных с физическим параллелизмом.

Увы, но существующие решения типа pthread и OpenMP, могут расцениваться лишь как паллиатив. Ручное управление физическим параллелизмом (explicit parallelism) отбрасывает программирование назад в эпоху машинных кодов и ассемблеров, реанимируя в программировании привязанность структуры программы к физической архитектуре вычислительной платформы.

Автоматическое распараллеливание (implicit parallelism) для многоядерных архитектур в большинстве случаев обеспечивает ускорение лишь с некоторой вероятностью. Промежуточным решением могут быть уже упоминавшиеся реконфигурируемые библиотеки (Integrated Performance Primitives, MathKernelLibrary).

Наиболее перспективны языки и техники, которым естественно присущ логический параллелизм, независимость или слабая зависимость компонентов.

В первую очередь, речь идет о многозадачном логическом параллелизме. Существующие наработки – это практически полуфабрикат для многоядерной платформы. Явный недостаток многозадачности – она слишком крупнозернистая. Практически невозможно придумать убедительный случай загрузки персонального компьютера достаточным количеством (десятком и более) действительно ресурсоемких задач.

Другая возможность – попытаться адаптировать к многоядерным архитектурам существующие техники и языки многопоточной организации программ. Опыт применения языка Рефлекс [4] в многопроцессорных системах показывает, что это вполне допустимо. Правда, появляющиеся при этом сущности означают достаточно радикальное изменение программы на уровне машинных команд. Например, происходит очевидный отказ от использования стека при организации структуры программы. При создании программ меняется и восприятие процесса программирования: классические функции дополнены процессами, которые при запуске порождают отдельный и независимый поток команд и данных.

Заключение

В основном достигнут технологический предел увеличения вычислительной мощности отдельно взятого процессора. Это подрывает статус-кво в отношениях между производителями аппаратуры, системного и прикладного ПО и конечными пользователями, и соответственно, угрожает динамике IT-индустрии в целом.

Ведущие фирмы-изготовители микроэлектронных компонентов, пытаясь сохранить тенденцию роста производительности, предлагают радикальные изменения в базовой архитектуре персональных компьютеров – многоядерные процессоры. Эти архитектурные изменения предполагают коррекцию общепринятых подходов при организации ПО, а именно – активное использование физического параллелизма.

Физический параллелизм, с одной стороны, не применим для большого класса разработанных последовательных алгоритмов, а, с другой стороны, в чистом виде означает возврат к низкоуровневому программированию… Со всеми вытекающими последствиями, начиная от повышения квалификационных требований к разработчикам и увеличения затрат на создание и реинжениринг ПО, что позволяет охарактеризовать возникшую ситуацию как кризисную.

Наступило время переосмыслить базовые концепции организации и распространенные техники разработки системного программного обеспечения с целью создания высокоуровневых методик и языков программирования, не предполагающих отдельного рассмотрения вопросов физического параллелизма.

При поиске решений, обеспечивающих и выигрыш в производительности, и бесшовный переход на многоядерные архитектуры, наиболее перспективно исследовать в первую очередь существующие методы и языки логического параллелизма.

Список литературы

1. Воеводин В.В. Параллельная обработка данных. Курс лекций // Лаборатория параллельных информационных технологий, НИВЦ МГУ, 2000 [http://parallel.ru/vvv/index.html]
2. Легалов А., Кузьмин Д., Казаков Ф., Привалихин Д. На пути к переносимым параллельным программам // Открытые системы, №5, 2003. [http://www.osp.ru/os/2003/05/036.htm]
3. Макконнелл Дж. Основы современных алгоритмов // М.: Техносфера, 2004. – 368с.
4. Зюбин В.Е., Петухов А.Д. Распределение вычислительных ресурсов в средах с многопоточной реализацией гипер-автомата // Труды III Международной конференции «Идентификация систем и задачи управления» SICPRO’04. Москва, 28-30 января 2004 г. Институт проблем управления им. В.А. Трапезникова РАН
5. Sutter H. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software // Dr. Dobb’s Journal, 30(3), March 2005. [http://www.gotw.ca/publications/concurrency-DDJ.htm]