фундаментальные проблемы многопроцессорных систем

крис касперски, no-email

многопроцессорные системы долгое время были уделом суперкомпьютеров, мощных серверов и высокопроизводительных рабочих станций. но вот они хлынули в массы, чего массы никак не ожидали. никто же ведь не предупреждал, что помимо достоинств многопроцессорным системам присущи еще и недостатки, многие из которых носят фундаментальный характер, вынуждающий инженеров искать компромиссные решения, за которые приходится расплачиваться конечным потребителям. увы, реальность такова, что многопроцессорная система это действительно очень сложная вещь (намного сложнее чем пылесос) и чтобы не попасться на удочку рекламной пропаганды необходимо обладать определенными знаниями, хотя бы на базовом уровне.

Рисунок 1 фото автора

введение
/ полоса 2, колонка 1, 2 */*

Обвальный рост объема обрабатываемых данных в какой-то момент превысил резервы разрядности процессоров и тактовой частоты. Очевидное решение — объединение нескольких процессоров в одну упряжку, позволило снизить остроту проблемы на некоторое время, но ожидаемого прорыва производительности не произошло?

Если заглянуть в TOP-500 самых мощных суперкомпьютеров мира (http://parallel.ru/computers/top500.list.html), можно обнаружить что _все_ они построены по многопроцессорной схеме, причем, наименьшее количество процессоров составляет 50 штук,

именно столько их содержится в компьютере Hitachi SR11000-K2, находящимся на 441 месте в списке, отсортированном в порядке убывания производительности. Самое большое количество процессоров (212.992) установлено в IBM eServer Blue Gene Solution, с большим отрывом лидирующим по производительности среди остальных, но эта вовсе не… хм… "реальная производительность", а, извините за выражение, грязный PR.

Подавляющее большинство суперкомпьютеров содержит порядка тысячи процессоров, причем, приведенные показатели производительности получены с помощью тестов, изначально "заточенных" под многопроцессорность, и представляющих собой хорошо распараллеливаемые математические задачи, решение которых обходит стороной проблемы когерентности, синхронизации, неоднородности иерархии оперативной памяти… На произвольно взятой задаче показатели наверняка изменятся. Если, допустим, мы пытаемся вскрыть криптоалгоритм, плохо поддающийся распараллеливаю, то на первое место вырвется Hitachi SR11000-K2, со своими процессорами POWER5+ 2300 MHz, из которых реально будет работать только один (задача-то ведь не распараллелена!), а IBM eServer Blue Gene Solution окажется далеко позади, потоку как собран на основе дешевых PowerPC 440 с тактовой частотой 700 MHz.

Отсюда следует вывод: производительность понятие абстрактное и чтобы получить достоверные данные (а не "попугаев") необходимо приложить его к конкретной задаче. На одних задачах лидируют одни архитектуры, на других — другие, вот только этих данных нет нигде. Производители (и независимые обозреватели!) обычно ограничиваются тем, что приводят теоретическую пиковую производительности и максимальную производительность, полученную по тестам LINPACK, показывающим на что способен тот или иной компьютер, если загрузить его идеализированной математической задачей.

Естественно, программисты (чьи программы предполагается использовать на многопроцессорных машинах) _должны_ учитывать особенности их архитектуры, максимально оптимизируя задачу на всех уровнях ее постановки - от алгоритма до реализации, но, увы, огромный спектр задач крайне плохо "ложиться" на концепцию многопроцессорных систем и влечет за собой те или иные издержки, приводящие к тому, что реальная производительность оказывается намного ниже пиковой.

Именно потому, существует множество схем построения суперкомпьютеров, каждая из которых рассчитана на свой класс вычислительных задач. А каждому классу сопутствуют свои проблемы, над решением которых и бьются инженеры. Очень давно бьются и на суперкомпьютерах эти проблемы… нет, не то, чтобы совсем решены, но, по крайней мере, достигли состояния стабильной стагнации. Произошел своеобразный идеологический "раздел" территорий, причем произошел уже давно — каждая отдельно взятая архитектура достигла совершенства в своей области, смирившись с существованием свойственный ей непреодолимых проблем. Революционный прорыв невозможен! Рваться-то некуда. Научные идеи не берутся из неоткуда и во всяком предметной области они конечны и исчерпаемы. Вот их и исчерпали. Теперь дальнейший рост производительности возможен лишь за счет наращивания количественных (тактовая частота и число процессоров), а отнюдь не качественных показателей (новый высокоэффективный протокол для поддержания когерентности, например).

Но… это все что касается суперкомпьютеров. Персоналкам, рабочим станциям и серверам еще предстоит осознать свое место мире и пройти стадию архитектурного дробления. А пока все они строятся по одному и тому же принципу и представляют собой сплошное нагромождение проблем, о некоторых (наиболее значимых) из которых, мы сейчас и поговорим.

Впрочем, первые шаги в этом направлении уже сделаны. В Windows Server 2003 появились первые механизмы, позволяющие программисту самостоятельно управлять планированием выполнения задач в многопроцессорных системах (до этого, планировка осуществлялась исключительно самой Windows).

>>> врезка расплата за бездумность
/ полоса 2, колонка 3 /**

"*Известный гуру в области ИТ Джин Амдал предложил схему для описания эффективности распараллеливания на многопроцессорном сервере: E = N/[NxC+(1-C)], где E — показатель, характеризующий прирост производительности сервера, N — количество процессоров, C — коэффициент, характеризующий долю нераспараллеливаемых команд в коде (0 <= C <= 1).*

Анализ этой зависимости позволяет сделать два основных вывода. Во-первых, при достаточно большой доле не распараллеливаемых команд в коде программ (допустим, C = 0,2)

рост производительности при определенном количестве процессоров прекращается (в нашем случае N=20). И во-вторых, наиболее эффективный способом повышения производительности вычислительной системы — не увеличение количества процессоров, а алгоритмическое совершенствование приложения.

С другой стороны, цена сервера зависит от максимального количества процессоров, которые можно установить в него. В 1999 г. автор рекомендовал одной уважаемой компании остановить выбор на двухпроцессорном CISC-сервере стоимостью около 12 тысяч долл., основываясь на том, что основное приложение обладает низкой распараллеливаемостью. Компания купила четырехпроцессорный CISC-сервер с двумя установленными процессорами, заплатив за него 27 тысяч долл. В 2001 г. возникла необходимость повысить производительность вычислений. Компания приобрела еще два процессора к ранее купленному серверу, заплатив за модернизацию около 7 тысяч долларов. Эффект был ошеломляющим для руководства компании: производительность не только не увеличилась вдвое, но упала примерно на 10%. В результате ошибочного выбора непроизводительные расходы на сервер составили сотни процентов от цены" (источник — http://www.in-line.ru/themes/INLINE/inlinenews-document.asp?folder=1532&matID=1739)

>>> врезка параллельными путями
/ полоса 2, колонка 3 */*

Распараллеливать вычисления можно по меньшей мере двумя путями — "руками" (программист самостоятельно разбивает программу на независимые блоки) или же поручить это дело компилятору. Очевидно, что компилятор, стиснутый рамками конкретной программы, всегда будет проигрывать естественному интеллекту, способному на качественно новые решения.

С другой стороны — каждая аппаратная архитектура имеет свои особенности и создание программы, эффективно работающей более чем на одной платформе — задача не из дешевых. К тому же вместе с параллельными компилятором обычно поставляются высокоэффективные библиотеки, уже оптимизированные вручную и зачастую содержащие существенную долю ассемблерного кода.

Фирма Intel выпускает для своих процессоров коммерческие версии компиляторов для языков Си/Си++ и Фортран, работающие как на Windows, так и на Linux.

параллелизм
/ полоса 3, колонка 1, 2, 3 */*

Фундаментальную проблему параллелизма легко объяснить на примере едкого, но меткого выражения: "если собрать девять беременных женщин, они не родят ребенка через месяц". Вот так же и с процессорами. Не все задачи могут быть распараллелены. И хотя девять женщин за девять месяцев родят девять детей (двойняшек мы в расчет не берем), в то время как одна женщина при прочих равных родит только одного, отсюда еще не следует, что многопроцессорность увеличивает производительность тех задач, что не поддаются параллелизму в принципе.

Хорошо, если нам нужно девять детей, а как быть, если нам нужен всего один? Или, переходя к более конкретным задачам, требуется найти ключ к зашифрованному посланию, разложить очень большое число на множители и т. д.

Параллельные вычисления зажаты в тиски двух крупнейших проблем: алгоритмической и технической. Математики должны разработать специальные методы решения задач, разбив одну задачу на несколько _независимых_ подзадач. Независимых — это значит, что одна подзадача не должна пользоваться результатами вычислений другой, что не всегда достижимо. Например, решение квадратного уравнения через дискриминант образует между подзадачами неустранимую зависимость по данным и потому эти подзадачи не могут выполняться параллельно.

С другой стороны, пусть у нас имеется два процессора и задача, разбитая на четыре подзадачи A -> B; C -> D; где знаком "->" отмечена зависимость по данным. Достаточно очевидно, если выполнять сначала подзадачи A и C, а затем B и D, то на _двухпроцессорной_ машине удастся добиться 100% распараллеливания, однако, это не значит, что на однопроцессорная машина решала бы эту задачу за вдове больше время. Разбиение задачи на подзадачи несет довольно высокие накладные расходы, которые математики пытаются снизить, но… увы! До полной победы в этой борьбе им далеко и в некоторых случаях накладные

расходы на распараллеливание съедают весь выигрыш от производительности, причем, чем больше у нас процессоров — тем выше издержки. В самом деле, в обозначенном примере мы неявно полагали, что подзадачи A, B, C и D выполняются за одинаковые промежутки времени, что практически никогда не достигается на практике. Допустим, что задача C решается на 50% быстрее, чем A. Тогда (при прочих равных) двухпроцессорная машина даст всего 1,75 прирост производительности по сравнению с однопроцессорной. И это только в теории! На практике же результат будет еще хуже. Значительно хуже. Почему?!

Тут на сцену выходит техническая проблема реализации многопроцессорной системы. Процессоры при выполнении программы (даже распараллеленной на 100%) вынуждены взаимодействовать как друг с другом, так и с разделяемыми ресурсами. В первую очередь с оперативной памятью. Ситуация, в которой один процессор читает ячейку памяти в то время как другой модифицирует ее приводит к неопределенному поведению системы и должна быть исключена либо на аппаратном, либо на программном уровне (на практике же — требуются совместные усилия как разработчиков железа, так и программного обеспечения). А если вспомнить, что пропускная способность оперативной памяти не безгранична, то и вовсе грустно становится. Чуть позже в мы покажем какие существуют типы организации памяти, а пока же посмотрим как ученые, программисты и инженеры борются с проблемами параллелизма.

Смотреть мы будем, естественно на примере суперкомпьютеров, где параллелизм появился чуть ли не с самого начала. Компания "Jet Infosystems" приводит диаграммы рассеивания пиковых значений производительности в зависимости от количества процессоров по годам для самых мощных компьютеров (см. рис. 2), из которых наглядно видно, что в 1997 году коэффициент зависимости производительности от количества процессов был далек от единицы и во многих случаях приближался к функции квадратного корня (и это при том, что тогда процессоров число процессоров не превышало четырехсот).

В 1998 году о квадратном корне все забыли и коэффициент вплотную приблизился к единице, а через год (когда количество процессоров удвоилось) практически сравнялся с ней. В 2002 году количество процессоров резко возросло, а технологии распараллеливания остались все теми же. Как следствие — коэффициент пропорциональности "обиделся" и упал, впрочем, незначительно.

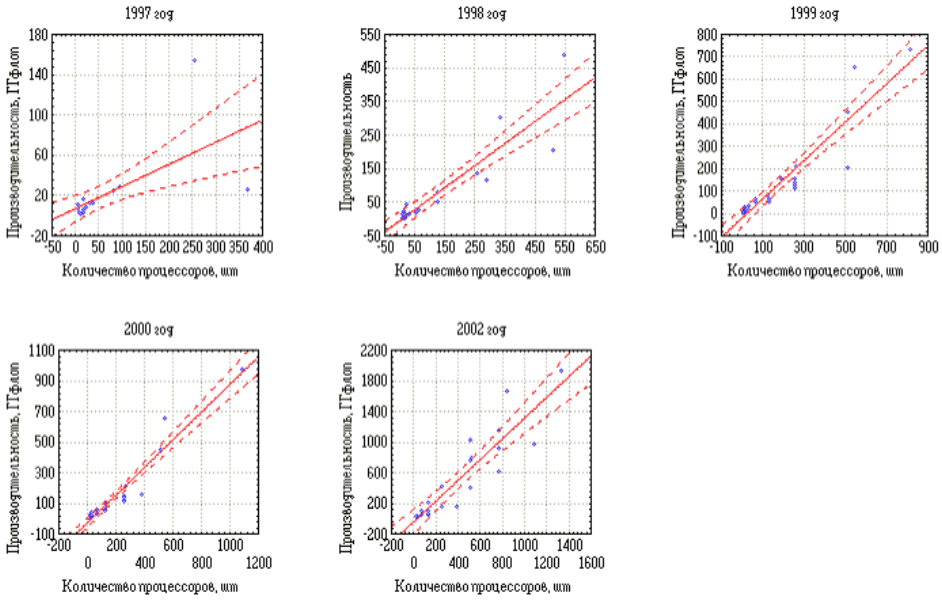


Рисунок 2 диаграммы рассеивания пиковых значений производительности в зависимости от количества процессоров по годам для самых мощных компьютеров по данным http://www.jetinfo.ru/2002/12/1/article1.12.2002395.html

Радуясь таким достижением, все же не будем забывать о том, что они относятся не к реальным приложениям, а специально подготовленным тестовым задачам. Кстати, о тестах. Возьмем уже упомянутый тест LINPACK-HPL 1.0a, соберем кластер на базе Intel Xeon с

тактовой частотой 2660 МГц и, заставив его обсчитывать матрицы разного размера (от 15000x15000 до 60000x60000) посмотрим как меняется производительность в зависимости от количества процессоров (см. рис. 3).

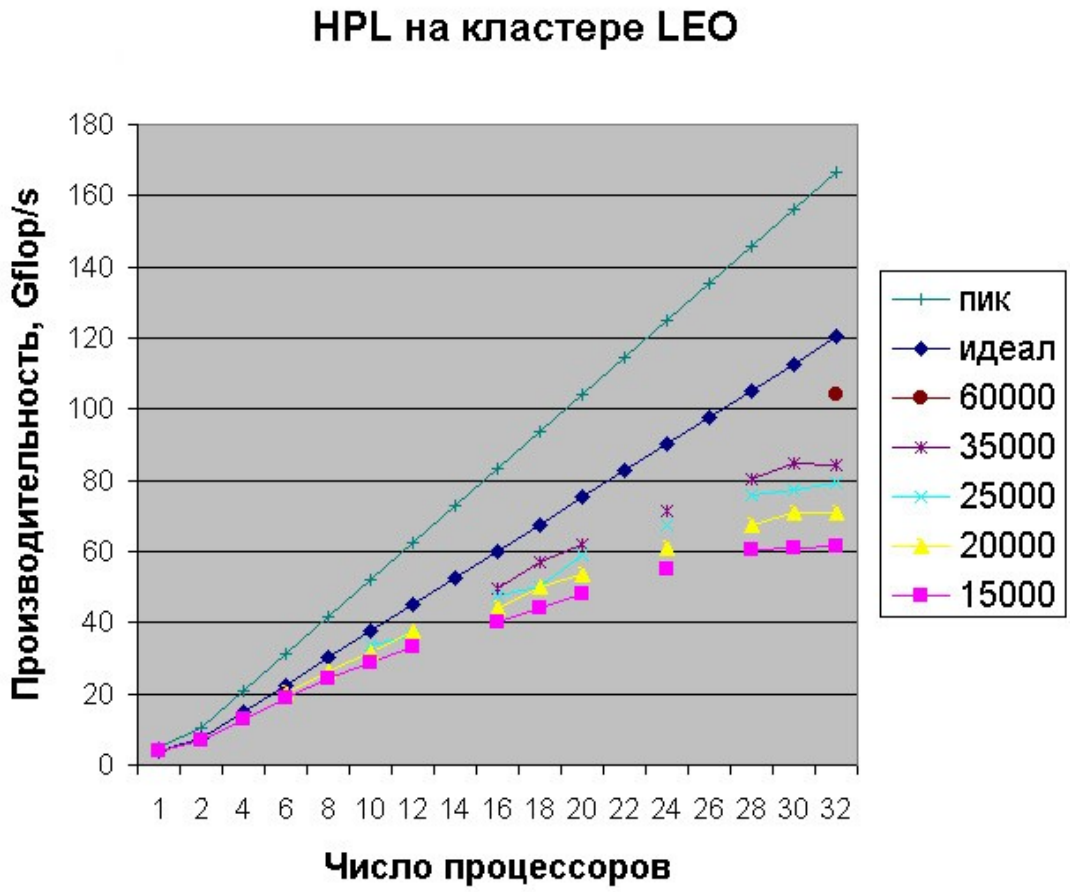


Рисунок 3 график, иллюстрирующий изменение производительность в зависимости от количества процессоров на кластере Xenon под тестом HPL (по данным http://parallel.ru/cluster/leo_linpack.html)

Мы видим, что реальная зависимость производительности от количества процессоров носит далеко не линейный характер и на матрицах 15000x15000 достигает своего насыщения уже на 24 процессорах. Матрицы большей размерности распараллеливаются значительно лучше, но… отстают от пиковой производительности как Россия от Америки, чтобы там ни говорили органы пропаганды.

На основании полученных данных мы можем построить график эффективности использования кластера (см. рис. 4), из которого видно, что на матрицах 15000x15000 эффективность _падает_ прямо пропорционально количеству процессоров в с коэффициентом пропорциональности ~1/2, а на матрицах максимального размера держится на уровне 0,9. Конечно, это не означает, что при добавлении новых процессоров производительность снижается. Естественно, она возрастает (куда же ей еще деваться при решении хорошо распараллеленной задачи) и программа выполняется за меньшее время, но! мы сейчас говорим не о производительности, а именно _эффективности_ использования. Процессоры не падают с неба, да еще и энергию (электрическую) потребляют. Добавление каждого нового процессора обходится все дороже и дороже, а вот… отдача от него все меньше и меньше.

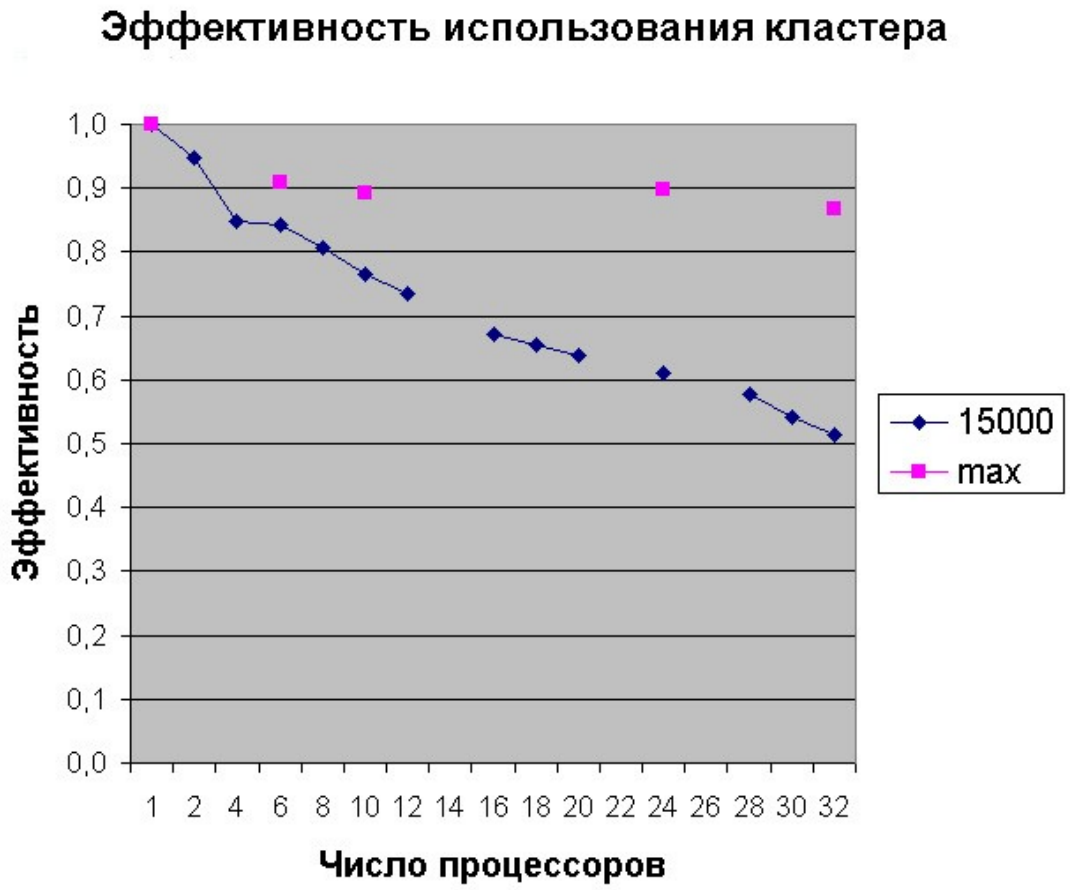


Рисунок 4 эффективности использования кластера (по данным http://parallel.ru/cluster/leo_linpack.html)

Таким образом, расчет стоимости владения многопроцессорной системой выливается в весьма непростую экономическую задачу. Все мы хотим, чтобы компьютер работал быстро, и даже быстрее, но пробовал ли кто-нибудь подсчитать стоимость процессорного времени? Допустим, сервер обработает вдовое больше запросов за заданный промежуток времени, но потребует при этом вчетверо больших вложений. Окупятся ли они? В некоторых случаях имеется жесткий временной лимит, отведенный для решения задачи при превышении которого полученные данные становятся ненужными или снижают свою актуальность (взять хотя бы прогноз погоды — кому он нужен на вчерашний день? проще на термометр посмотреть). Тогда мы вынуждены ставить столько процессоров, сколько это требуется. В остальных же случаях, большое количество процессоров работает против нас и рентабельность вычислительной системы существенно снижается.

когерентность
/ полоса 4, колонка 1, 2, 3 */*

Когерентность происходит от латинского слова "cohaerens", что буквально переводится как "находящийся в связи", а в более широком смысле означает скоррелированность (согласованность). Применительно к многопроцессорным системам когерентность означает, что процессоры согласуют свою работу при обращении к совместно используемым ресурсам и в первую очередь — оперативной памяти (порты ввода/вывода и дисковую подсистему мы рассматривать не будем, поскольку об этом успешно заботиться операционная система).

Даже в полностью распараллеленном приложении, процессоры вынуждены взаимодействовать друг с другом, обмениваясь данными (мы уже приводили простейший пример с задачей A -> B; C -> D). Вот вполне типичная ситуация. Имеется ячейка памяти X, содержащая значение A, которое считывает процессор CPU1, после чего процессор CPU2 записывает в X значение B. Допустим, что процессор CPU1 (незнающий, что содержимое X уже

изменено) увеличивает его на единицу (а почему бы и нет) и записывает обратно, уничтожая тем самым результат работы процессора CPU2.

Теперь вспомним, что все современные процессоры имеют кэш память, причем этой памяти очень много (зачастую намного больше мегабайта в пересчете на каждый процессор). Задумаемся, что произойдет, если при решении некоторой подзадачи процессор CPU1 запишет все (или часть вычисленных данных) в свой собственный кэш и тут же переключится на решение другой подзадачи, предоставляя процессору CPU2 возможность продолжить обработку данных, которую он считает из… основной оперативной памяти, содержимое которой осталось неизмененным и результатов вычислений там нет.

Конечно, кто-то может сказать: а нечего перекладывать дальнейшую обработку данных на процессор CPU2, пусть ей занимается CPU1! Все это верно, конечно. Никто же не спорит! Но… как быть если операционная система не позволяет закреплять потоки за процессорами? И что делать если, нам, например, необходимо просуммировать N чисел на K процессорах? Да нет ничего проще! Разбиваем N чисел на K блоков, считаем сумму каждого из них на всех процессорах независимо от других (от перестановки слагаемых сумма, как известно, не меняется), после чего нам останется только сложить K чисел. Если N много больше K (а обычно так и есть), то время выполнения задачи обратно пропорционально количеству процессоров с коэффициентом пропорциональности близким к единице. Близким, а не равным потому что пропускная способность памяти — это еще одна фундаментальная проблема, но о ней мы еще поговорим, а сейчас обратим наше внимание на то, что на последней стадии операции мы _вынуждены_ обращаться к результатам вычислений других процессоров, а они у каждого из них с вероятностью близкой к единице находятся в кэш-памяти.

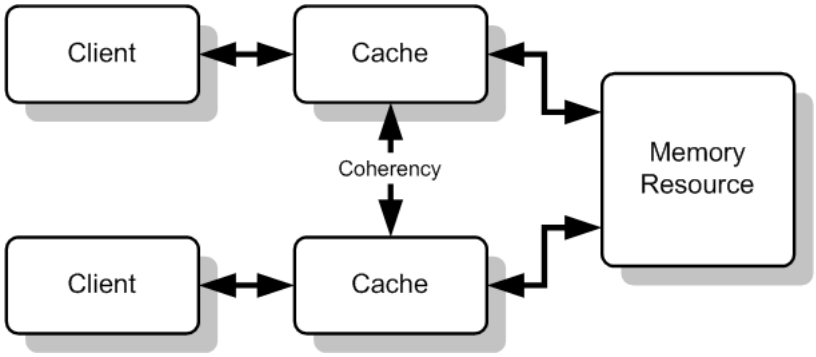
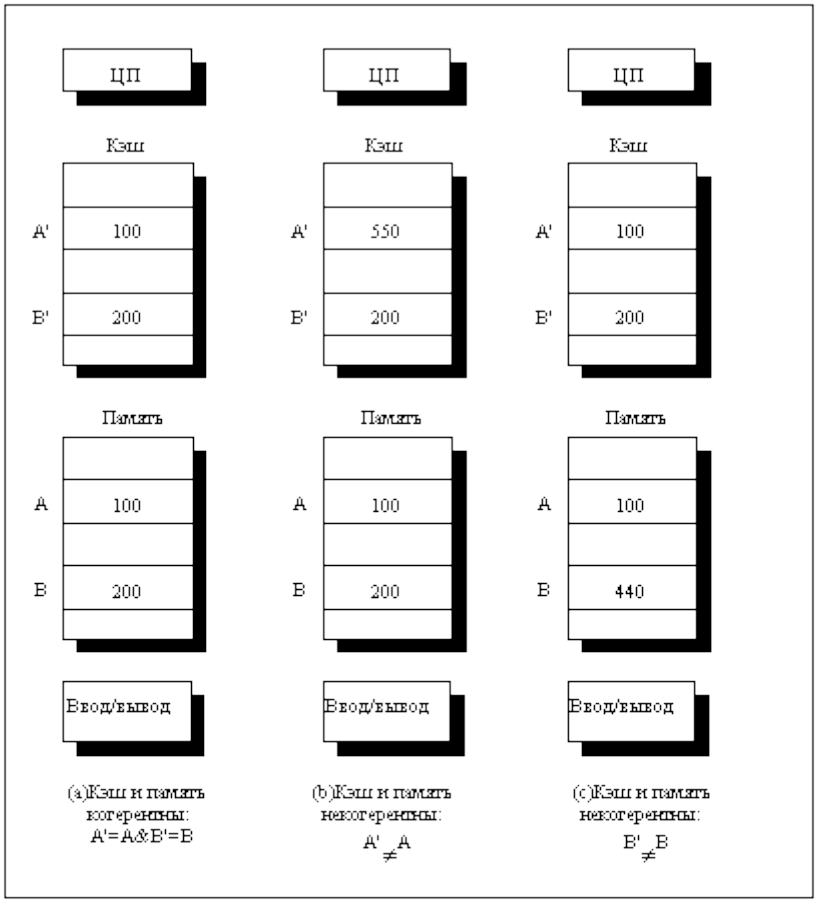


Рисунок 5 кэш-память всех процессоров должна быть когерентна

Существует два выхода из положения: либо вообще отказаться от кэширования, каждый раз обращаясь напрямую к оперативной памяти (да! мы даже ячейку в регистр теперь не можем поместить), либо разработать протокол, позволяющий процессорам информацией о том, какие именно ячейки находятся в кэш памяти, какие из них изменены и если процессор CPU1 обращается к ячейке памяти, которую процессор CPU2 загрузил в своей кэш и модифицировал, то CPU2 должен либо переслать модифицированную ячейку в кэш процессора CPU1, либо выгрузить кэш-строку (минимальную порцию обмена кэша с памятью) в основное ОЗУ, откуда его сможет подхватить CPU2.

Первый способ откинем сразу. Отказ от кэширования на данном этапе развития вычислительной техники невозможен, уж слишком велика разница быстродействия ядра процессора и оперативной памятью. Так что приходится согласовывать кэш-память процессоров посредством тех или иных протоколов, задействующих системную шину, тактовая частота которой зачастую в десятки раз ниже частоты ядра и которая используется для общения процессора с "внешним миром". Короче говоря, без накладных расходов тут никак не обойтись и хотя даже на дешевых процессорах типа Pentium-III/4 помимо системной шины предусмотрены специальные выводы, разгружающие шину и берущие задачи по согласованию кэшей на себя, частота их импульсов все равно намного меньше, чем у ядра. Потому что проводники, соединяющие соседние процессоры намного длиннее, чем внутрикистальные перемычки и максимально возможная тактовая частота упирается в чисто физические ограничения, которые очень сложно преодолеть.



A' и B' - кэшированные копии элементов A и B в основной памяти

a) Когерентное состояние кэша и основной памяти.

b) Предполагается использование кэш-памяти с отложенным обратным копированием, когда ЦП записывает значение 550 в ячейку A. В результате A' содержит новое значение, а в основной памяти осталось старое значение 100. При попытке вывода A из памяти будет получено старое значение.

c) Подсистема ввода/вывода вводит в ячейку памяти B новое значение 440, а в кэш-памяти осталось старое значение B.

Рисунок 6 иллюстрация проблемы когерентности кэш-памяти

Протоколов для поддержки когерентности разработано много. Это и MSI (образованный по первым буквам флагов Modified/Shared/Invalid, отражающих состояние кэш-блоков), и MESI (Modified/Exclusive/Shared/Invalid), и MOSI (Modified/Owned/Shared/Invalid), и многие другие. Подробно обсуждать их достоинства и недостатки никакого смысла нет, поскольку указанные протоколы реализованы внутри процессора и являются неотъемлемой частью его архитектуры, на которую конечный потребитель воздействовать не в состоянии.

Возвращаясь к нашим баранам, продолжим гнуть перспективную партийную линию: вместо того, чтобы гонять данные между кэш-памятью разных процессоров, намного лучше (и правильнее) стремиться обрабатывать данные на том процессоре, в кэш-памяти которого они уже находятся. Многоядерные процессоры привлекательны в том смысле, что кэш второго уровня у них общий (нет никаких расходов на поддержку когерентности), а индивидуальные для каждого ядра кэши первого уровня находятся внутри кристалла и потому могут согласовывать свое содержимое с минимальными накладными расходами.

Появление гетерогенных многопроцессорных систем (содержащих два и более многоядерных процессоров) породило проблему привязки потоков к "своим" ядрам. До недавнего времени все процессоры (как физические, так и виртуальные) были полностью равноправными с точки зрения операционной системы и потому поток, начав свое выполнение на ядре A процессора CPU1 мог продолжить его где угодно: и на CPU1-A, и на CPU1-B, и на CPU2-A, и на CPU2-B — это уж как фишка ляжет. Очевидно, что первый вариант является

наиболее предпочтительным. Второй вариант — немного похуже, но тоже в общем-то ничего. А вот два последних варианта — это смерть производительности.

Разработчики операционных систем отреагировали достаточно оперативно. Во-первых, они изменили алгоритмы планировки так, чтобы преимущество получал тот процессор, на котором поток был прерван, а, во-вторых, предоставили в распоряжение программиста API-функции, позволяющие закреплять потоки за процессорами в принудительном порядке, обусловленным спецификой решаемой задачи. Естественно, алгоритмы планировки возымели действия стразу (достаточно перейти с Windows 2000 Server на Windows 2003 Server, чтобы почувствовать разницу), а вот новые API-функции требуют полного редизайна всего существующего программного обеспечения, что влечет за собой намного больше издержки, чем обновление операционной системы (что касается UNIX'a, то там вообще достаточно обновить ядро, не трогая всего остального).

организация оперативной памяти / полоса 5, колонка 1, 2, 3 */*

Многопроцессорные системы обычно приобретаются для обработки огромных объемов данных. В кэш они, естественно, не вмешаются и основная нагрузка ложится на оперативную память, которую можно рассматривать как своеобразный кэш дисковой подсистемы (не путать с кэш-буфером дискового драйвера!) и от объема которой производительность зависит ничуть не меньше, чем от количества процессоров.

График, изображенный на рис. 7 отображает динамику изменения коэффициентов линейной парной корреляции пиковой производительности с объемом оперативной памяти и количеством процессоров. Как нетрудно видеть, объемы оперативной памяти неуклонно растут, следовательно, увеличивается и интенсивность взаимодействия процессоров с ОЗУ и чтобы подсистема памяти не оказалась самым узким местом системы, необходимо выбрать правильную архитектуру.

Рисунок 7 динамику изменения коэффициентов линейной парной корреляции пиковой производительности с объемом оперативной памяти и количеством процессоров (по данным http://www.jetinfo.ru/2002/12/1/article1.12.2002395.html)

Наиболее популярны системы с разделяемой памятью, общей для всех процессоров и называемой UMA (Uniform Memory Access — Память с Однородным Доступом). Система, построенная по такому принципу изображена на рис. 8. Она довольно проста в реализации и стоит недорого, но, увы, не обходится без недостатков. Про необходимость поддержки когерентность кэш-памяти мы уже упоминали, а теперь попробуем рассчитать пропускную

способность системной шины. Системная шина современных процессоров обладает гораздо большей пропускной способностью нежели микросхемы динамической памяти (статическая память все еще остается слишком дорогой игрушкой, чтобы найти себе применение в качестве основного хранилища данных). Даже на персональных компьютерах начального уровня модули памяти приходится объединять парами, иначе потенциал процессорной шины так и окажется потенциалом.

Выделить каждому процессору по паре модулей памяти — не проблема, то есть это с экономической точки зрения не проблема, а с технической — ничего (хорошего) у нас не получится. Архитектура не дает. Ведь на системах с однородной памятью, все модули памяти равноправны и ни один из них не закреплен за каким-то конкретным процессором, а потому два и более процессоров могут обращаться к одному и тому же модулю.

Для предотвращения конфликтов совместного доступа контроллер памяти содержит планировщик запросов, который ставит все запросы в очередь, организованную наиболее оптимальным (с точки зрения производительности) образом. Страничная организация динамической памяти не допускает индивидуальных обращений к ячейкам памяти и оперирует целыми страницами длинную в несколько килобайт. Для ускорения доступа каждая открываемая страница копируется в буфер статической (т. е. сверхбыстродействующей) памяти и потому при последовательном чтении/записи мы получаем вполне приемлемую производительность. Но вот если мы начинаем читать/записывать память из разных мест адресного пространства, производительность падает драматически!!! Чтобы хоть как-то выкрутиться из ситуации, память делится на банки и каждый банк может удерживать отрытой ровно одну страницу в течении некоторого времени, а потом закрывать ее на перезарядку. Но количество банков невелико, а схема отображения адресного пространства на банки динамической памяти в общем случае неизвестна и потому реальная пропускная способность подсистемы памяти в UMA-системах намного меньше пиковой, причем, с ростом количества процессоров она стремительно падает.

Поэтому, компьютеры, построенные на базе UMA-архитектуры обычно содержат небольшое количество процессоров (не больше 32х, а чаще всего намного меньше).

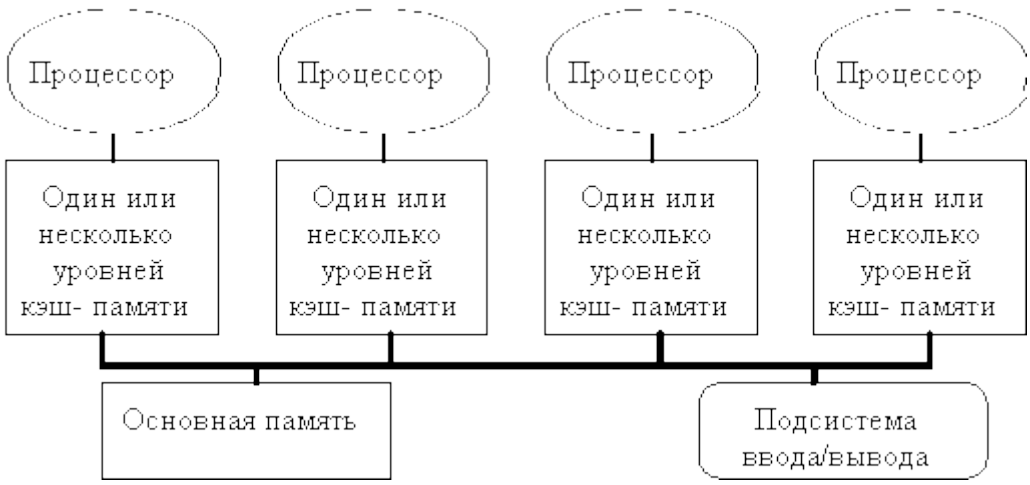


Рисунок 8 блок-схема компьютера, построенного по принципу UMA-архитектуры

Конечно, для ПК слова "32 процессора" выглядят просто фантастично, но… в "серьезных" компьютерах количество процессоров измеряется сотнями и такие машины активно используются и в бизнесе, и в науке, и во многих других отраслях. UMA-системы для таких целей не подходят в силу плохой масшбарируемости и там господствует NUMA-архитектура (Non-Uniform Memory Access — Память с Неоднородным Доступом), схематично изображенная на рис. 9.

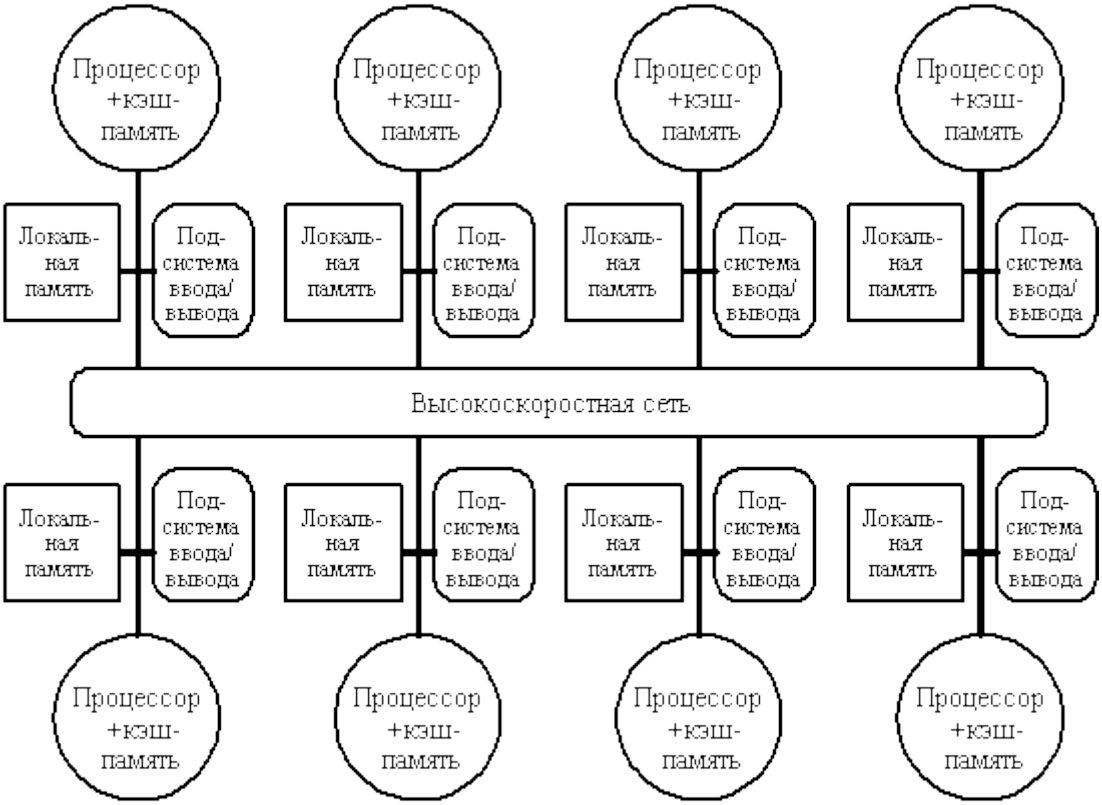


Рисунок 9 блок-схема компьютера, построенного по принципу NUMA-архитектуры

В самых общих чертах идея заключается в "закреплении" модулей памяти за "своим" процессором (набором процессоров). Как говорится, просто как и все гениальное. Но это оно на бумаге просто, а в реальной жизни сразу же возникает проблема поддержки когерентности — должны же как-то процессоры обмениваться данными. Возможных решений множество, но в большинстве популярных NUMA-архитектур (к которым, в частности, принадлежит и знаменитый Cray T3D), разделяемые данные не кэшируется. Процессор со своей собственной памятью превращается в узел, соединенный с другими узлами либо шиной, либо посредством локальной сети. В узлах сети размещены контроллеры, перехватывающие запросы к памяти и определяющие: является ли данный запрос локальным или удаленным. Локальные запросы обслуживаются как обычно, а при удаленных соответствующему узлу передается запрос на "подкачку" данных. В правильно спроектированной программе большинство запросов выполняется локально и межпроцессорный обмен данными невелик и фактически сводится к управляющим/координирующим сообщениям.

Очевидно, если программу, написанную в расчете на UMA систему запустить на компьютере, построенным по NUMA архитектуре, то производительность последнего окажется где-то на уровне дешевой рабочей станции. Увы, даже гигабитный Ethernet, соединяющий соседние узлы друг с другом, заметно отстает по пропускной способности от системной шины со всеми вытекающими отсюда последствиями.

В Windows поддержка NUMA-памяти появилась лишь начиная с Vista и Server 2008, однако, на рынке высокопроизводительных компьютеров Microsoft чувствует себя достаточно слабо, не говоря уже о том, что такие компьютеры практически всегда поставляются с уже установленной операционной системой, разработанной их непосредственным производителем и обычно представляющем собой тот или иной клон UNIX'а.

заключение

В ближайшие годы следует ожидать скачкообразного увеличения количества процессоров на рабочих станциях и серверах, а вместе с этим и неизбежного снижения их эффективности. Проблемы параллелизма действительно фундаментальны и, если их не смогли в полной мере решить разработчики суперкомпьютеров, то какие у нас есть основания надеяться, что рабочие станции и сервера найдут волшебный эликсир?!

Производительность, конечно, продолжит расти, но не так быстро как она росла все эти годы, а вот цены на "быстродействие" ощутимо возрастут. Увы, золотому веку дешевых мегагерц наступает конец. До сих пор производители программного и аппаратного обеспечения строили свой бизнес на _росте_ производительности. Теперь же ситуация изменилась и им необходимо научиться зарабатывать деньги в новых условиях. Условиях, ростом аппаратных мощностей можно пренебречь.

С другой стороны, многопроцессорные системы намного лучше масштабируются, что в разы удешевляет их апгрейд. Так что ситуация крайне неоднозначна и кто окажется лидером, а кто аутсайдером – остается только гадать.

/* полоса 6, колонка 1, 2, 3 — все картинки (условно) */