ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ № 3(3), 2010, с. 73–84

УДК 004.382.2

А. О. Лацис

Модели программирования для современных суперкомпьютеров

Аннотация. В современных суперкомпьютерах широко применяется оборудование, принципиально отличающееся от оборудования вычислительных кластеров 5–6-летней давности. Это относится как к вычислительному, так и к коммуникационному оборудованию. Новое, более эффективное коммуникационное оборудование обладает рядом преимуществ, которые плохо выражаются в терминах старых технологий параллельного программирования. В статье исследуется вопрос о том, какие именно модели и технологии параллельного программирования идут на смену традиционным и почему.

Ключевые слова и фразы: суперкомпьютер, модель программирования, технология параллельного программирования, сеть двусторонних обменов, сеть односторонних обменов, кэш-когерентность, латентность, агрегация.

Введение

Модель программирования, основанная на применении MPI [2], прочно закрепилась в качестве основной и не имеющей практической альтернативы за прошедшее «кластерное десятилетие» [1]. Однако, последние примерно 3 года ознаменовались быстрым (и все ускоряющимся) изменением облика типичного суперкомпьютера. Поскольку базовая модель программирования — это, в первую очередь, логическая абстракция наиболее существенных свойств вычислительного оборудования в глазах программиста, эти изменения в оборудовании не могут не привести к пересмотру доминирующей роли модели MPI, к появлению новых моделей и технологий.

Перечислим кратко те принципиальные изменения в облике сегодняшних суперкомпьютеров, которые наиболее очевидно требуют пересмотра устоявшейся модели программирования.

Работа поддержана проектами РФФИ № 08–07–00068–а, № 09–07–13598–офи_ц и № 10–01–05009–б.

(1) Совершенствование системы межузловых коммуникаций. Современные сети, ориентированные на применение в вычислительных кластерах, предоставляют возможность прямого доступа вычислительного узла к памяти других узлов со скоростями и задержками, сравнимыми со скоростью и временем задержки локального доступа в оперативную память [6]. Можно уверенно прогнозировать появление сетей с еще более низкими уровнями задержек в самом ближайшем будущем. В этих условиях использование модели MPI, ориентированной на сравнительно редкие межузловые обмены крупными порциями данных, с принудительной синхронизацией на каждом обмене, представляется явным анахронизмом.

(2) Рост числа процессорных ядер в составе вычислительного узла. Современные узлы вычислительных кластеров содержат обычно 8–16 процессорных ядер, и это число имеет тенденцию к быстрому росту [3, 5]. Процессорные ядра внутри вычислительного узла объединены мощнейшей «коммуникационной подсистемой» — симметрично (или почти симметрично) доступной, кэш-когерентной общей памятью. Модель MPI наличие этой «коммуникационной подсистемы» фактически игнорирует.

(3) Появление гибридных вычислительных архитектур. Все чаще в составе вычислительного узла современного суперкомпьютера можно увидеть, наряду с многоядерными процессорами общего назначения, те или иные сопроцессоры — ускорители с принципиально иной, нежели у «обычного» процессора, архитектурой [1]. Эти вычислительные устройства программируются совершенно иначе, нежели универсальный процессор, и вводят в программу дополнительный уровень коммуникаций: помимо привычных коммуникаций между узлами, программист вынужден налаживать также обмены данными между процессорами и сопроцессором внутри узла. Логика обменов данными в программе объективно усложняется, а требования к эффективности ее реализации — растут. Появляется острая необходимость записывать обмены данными гораздо проще, но не за счет эффективности, а с одновременным ее повышением.

Словом, процессоры (процессорные ядра) в современном суперкомпьютере связаны друг с другом гораздо теснее, нежели процессоры типичного кластера, скажем, 5-летней давности. Коммуникационное оборудование суперкомпьютеров в последние годы сделало заметный шаг в сторону систем с общей памятью, но пока не прошло эту дистанцию полностью. При этом, ставшая за многие годы привычной модель программирования MPI по-прежнему преобладает. Это приводит к порочному кругу замедления прогресса в разработке и применении как новых моделей программирования, так и коммуникационного оборудования, способного их поддержать. Новые модели не появляются, поскольку для уже имеющегося оборудования имеются хорошие реализации MPI, а новое оборудование, способное дать программисту нечто большее, чем MPI, не появляется, поскольку нет новых моделей, в рамках которых это новое оборудование можно было бы использовать.

Настоящая работа посвящена попытке ответа на два вопроса:

- какие базовые модели программирования могли бы прийти на смену модели MPI (или дополнить ее возможности) уже сегодня;
- как могло бы выглядеть вновь создаваемое коммуникационное оборудование с точки зрения наилучшей поддержки таких новых моделей.

1. Иерархия и спектр классов коммуникационного оборудования

Хорошо известно [1], что многопроцессорные вычислительные системы делятся на мультикомпьютеры (системы без общей памяти) и мультипроцессоры (системы с общей памятью).

Мультикомпьютеры, в свою очередь, распадаются на два подкласса: системы, коммуникационные сети которых ориентированы на двусторонние обмены данными, и системы на базе сетей односторонних обменов данными.

Мультипроцессоры, в свою очередь, также распадаются на два подкласса: системы с симметричным доступом узлов к общей памяти (SMP) и системы с асимметричным доступом (NUMA).

Получается двухуровневая иерархия (рис. 1).

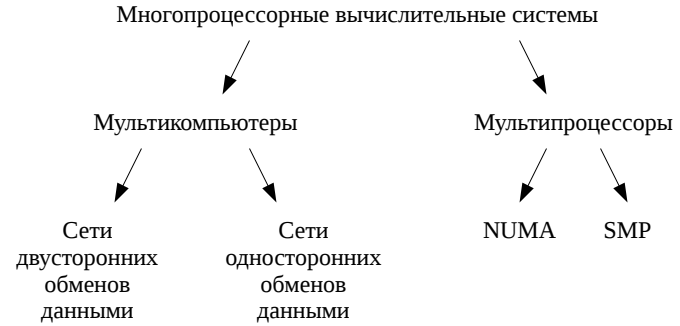

Рис. 1. Иерархия классов коммуникационного оборудования

С другой стороны, расположив подклассы коммуникационного оборудования в естественном порядке, мы получим уже не иерархию, а линейный спектр, в котором классы коммуникационного оборудования расположены по нарастанию степени тесноты связи между узлами.

(1) Сети двусторонних обменов. Доступ узлов в память друг друга отсутствует, для передачи данных из узла в узел требуются согласованные действия передающего и принимающего узла.
(2) Сети односторонних обменов. Доступ узла в память другого узла возможен, но только путем обращения к специальным функциям (подпрограммам): get() и put(). Грубо говоря, узел может работать с памятью другого узла примерно так же, как с файлом на диске.
(3) Общая память NUMA. Доступ узла в память другого узла возможен непосредственно, теми же машинными командами (языковыми конструкциями), которые применяются для доступа к собственной памяти. Однако, команды доступа в чужую память выполняются гораздо (иногда — во много десятков раз) медленнее, чем команды доступа в свою память. Программист обязан учитывать эту разницу в скорости доступа.
(4) Общая память SMP. Вся память всех узлов общая и равно доступная, само понятие доступа в «чужую» память лишено смысла.

Коммуникационному оборудованию на базе сети двусторонних обменов естественным образом соответствует модель программирования MPI.

Коммуникационному оборудованию на базе общей памяти SMP столь же естественным образом соответствует модель программирования OpenMP [9], за многие годы своего существования доказавшая свою практическую применимость и полезность, распространенная весьма широко.

Ничего подобного MPI и OpenMP по степени распространенности, применимости и общепризнанности для двух промежуточных классов коммуникационного оборудования пока не существует.

В то же время, в области аппаратных решений именно в области двух промежуточных классов в последние годы происходит бурный рост. Сегодня уже нет никаких технических препятствий к тому, чтобы системы, например, NUMA были столь же дешевыми и столь же легко масштабировались, что и системы на базе сети двусторонних обменов данными. В то же время, цель сделать дешевыми и масштабируемыми системы SMP сегодня столь же далека, что и 10–15 лет назад.

Таким образом, вопрос о новых базовых моделях параллельного программирования на сегодняшний день можно переформулировать следующим образом:

Возможно ли сформулировать промежуточную (между MPI и OpenMP) базовую модель параллельного программирования, к поддержке которой могли бы стремиться разработчики коммуникационного оборудования, а к использованию — прикладные программисты, или же единственным разумным шагом по «улучшению MPI» является только переход на OpenMP?

Искомая модель, если ее удастся построить, должна быть «промежуточной» в двух смыслах. Она должна давать программисту больше удобств, больше возможностей, нежели MPI, и делать это за счет новых свойств оборудования (поддержки односторонних обменов и/или NUMA).

2. Наивная попытка построения промежуточной модели

Рассмотрев в первом приближении два промежуточных класса коммуникационного оборудования, легко подобрать базовые модели

программирования, на первый взгляд, соответствующие им напрямую.

Для NUMA — систем естественной базовой моделью является модель PGAS (Partitioned Global Address Space) [1]. Эта очень простая и интуитивно понятная модель на практике представлена сегодня двумя языками — Co-Array Fortran[7] и UPC [8].

В обоих языках параллельная программа состоит из порождаемых при запуске занумерованных процессов (ветвей), как в MPI-1. Во всех ветвях исполняется строго одна и та же программа. В качестве основного средства синхронизации используются барьеры. По умолчанию переменные и массивы, объявленные в тексте программы, свои в каждой ветви, как в программах, написанных с использованием MPI. Данные, расположенные в единственном экземпляре в общей памяти и доступные всем ветвям, надо объявлять специальным образом.

Оба языка являются расширениями своих базовых языков (Фортрана и C, соответственно) путем добавления специального типа данных — распределенного массива. Находясь в асимметрично доступной общей памяти, каждый распределенный массив доступен целиком каждому из процессов параллельной программы. При этом он поделен регулярным образом на части, каждая из которых доступна «быстро» одному из процессов, и «медленно» — всем остальным. Часть распределенного массива, доступную быстро данному процессу, будем называть локальной, или «своей» для данного процесса. Остальную часть распределенного массива назовем удаленной, или «чужой», для данного процесса.

Способы организация распределенных массивов в Co-Array Fortran и UPC несколько различаются.

В UPC применяется сплошная адресация распределенного массива. Можно, например, объявить одномерный массив длиной 1000 элементов таким образом, что элементы с 0-го по 99-й окажутся своими для нулевого процесса, с 100-го по 199-й — для первого, и так далее, по 100 элементов на процесс. В общем случае, распределение массива — блочно-циклическое, размер блока задается при объявлении массива.

В Co-Array Fortran, напротив, применяется не сплошная адресация распределенного массива. Нет возможности, например, объявить одномерный массив таким образом, чтобы, перебирая значения индекса, можно было попадать в данные разных процессов. Вместо

этого выделяется специальная «процессная размерность» — всегда последняя, индекс указывается в квадратных скобках. Например:

A(I,J)[K] — это элемент (I,J) двумерного массива A, расположенного в K-м процессе. Распределенный массив Co-Array Fortran, таким образом, напоминает книгу, страницы которой — его порции, расположенные в разных процессах. На каждой странице есть строка N и столбец M, но, перебирая строки, перебраться на следующую страницу без перелистывания нельзя. Плавного перехода со страницы на страницу нет.

На первый взгляд кажется, что оба эти языка задают примерно одинаковый и очень комфортный стиль программирования. В самом деле, достаточно распределить данные в массивах и циклы их обработки так, чтобы каждый процесс обращался преимущественно к своим данным, и лишь изредка — к чужим, расставить надлежащим образом барьеры, и задача решена. Особенно удобна принятая в UPC сплошная адресация: например, цикл прохода по распределенной размерности массива в каждом процессе может включать в основном обращения к своим элементам, но на краях затрагивать немного чужих (соседних). Записываются эти обращения к элементам массива одинаково (общая память!), просто обращения к чужим элементам занимают большее время. К сожалению, в действительности все не так просто и удобно.

Все дело в конкретной степени асимметрии нашей NUMA-памяти. Если асимметрии — 2–3-кратная, как в многопроцессорных серверах с процессорами AMD, то все хорошо. А если 200–300-кратная, как в больших, масштабируемых NUMA-системах? В этом случае многое зависит от того, читаем мы чужую память или пишем в нее. Если пишем — все в порядке: много мелких записей в чужую память контроллеру памяти легко агрегировать и конвейеризовать, время обращения к чужим данным в итоге окажется терпимым. С чтением хуже: прозрачная для программы аппаратная конвейеризация невозможна. Если процессором выдана команда «прочитать значение по адресу X», то с выдачей следующей аналогичной команды придется подождать до завершения предыдущей. В нашем примере прохода циклом по своей части массива с эпизодическим захватом соседних краев обращения к чужим элементам неизбежно будут обращениями по чтению, а не записи. По указанным только что причинам, чтение отдельных элементов массива с 200-кратной задержкой на каждом — удовольствие очень дорогое, даже если оно осуществляется редко.

Если бы аппаратура умела хотя бы агрегировать чтение отдельных элементов массива, и эта проблема была бы решена. Способ прозрачной для программы аппаратной агрегации чтений хорошо известен, называется он кэшированием. Но его использование означает, что NUMA-память должна быть кэш-когерентной!

Таким образом, «наивный» стиль программирования в модели PGAS имеет шанс быть эффективным, адекватным аппаратуре не просто на NUMA, а только на ccNUMA-системах. ccNUMA-системы заметно дороже и хуже масштабируются, нежели nccNUMA-системы. Следует ли отсюда, что для nccNUMA-систем программировать можно только с использованием MPI?

Не следует ни в коей мере. Модель PGAS годится и для них, но стиль программирования должен быть другим, к сожалению, менее комфортным для программиста. Один шаг «назад, к MPI» сделать все же придется.

3. Попытка №2, реалистичная

Попробуем модифицировать «наивный» стиль программирования в модели PGAS таким образом, чтобы он годился для nccNUMA-систем. Чем придется пожертвовать? Очевидно, возможностью обрабатывать чужие данные там, где они находятся, не копируя их в свою память. Организуем данные «по MPI-ному», выделив в каждом процессе место, куда необходимые этому процессу чужие данные копируются для обработки. Это позволит нам пользоваться для доступа в чужую память записью, а не чтением. Выше мы отмечали, что записи в чужую память естественным образом конвейеризуются и агрегируются аппаратурой, причем прозрачным для программы образом. Кэш-когерентность не требуется. Теперь мы уже не можем идти по распределенному массиву, изредка захватывая «чужие» элементы — надо организовать распределенный массив, своя порция которого в каждом процессе содержит место для копирования в нее этих чужих элементов. Сплошная адресация UPC не требуется, зато идеально подходит не сплошная адресация Co-Array Fortran. Программист потерял, казалось бы, самое ценное — возможность написать оператор доступа к элементу массива, не думая о том, к своему элементу это доступ или к чужому. Доступ к чужим элементам (как правило, запись «своих оригиналов» в «чужие копии») теперь выделен в программе структурно. Про каждый оператор программы можно сказать, к каким именно данным — к своим или к чужим —

он обращается. Однородность доступа принесена в жертву, что осталось? И лучше ли это чем-нибудь, нежели MPI?

Осталось довольно много: односторонняя дисциплина доступа, прозрачная конвейеризация и агрегация множественных мелких обменов.

Программисту в этой дисциплине не надо думать о выстраивании логики рандеву при сложных схемах обмена данными между процессами — каждый процесс просто пишет часть своих данных туда, где они потребуются, причем в любом порядке и любыми порциями. Упаковывать данные в длинные сообщения не требуется — это аппаратура сделает сама. По сравнению с MPI, появилась возможность инкрементального распараллеливания, упростилась отладка. Можно уже не тосковать о том, как здорово было бы сначала отладить все посылки, и лишь затем перейти к отладке соответствующих приемов. Значительно упростилась, например, параллельная реализация численных методов на неструктурированных сетках, для которых характерно дробное, не регулярное размещение в памяти передаваемых из процесса в процесс данных.

Словом, промежуточная между MPI и OpenMP модель программирования действительно построена. Она заметно проще как при первоначальном изучении, так и при регулярном использовании, чем MPI, при этом программы получаются более эффективными в тяжелых режимах.

4. Взгляд с другой стороны и еще одна модель

Выше мы отметили, что некоторая нетривиальная языковая возможность — сплошная адресация распределенных массивов в UPC — в нашей промежуточной модели оказалась избыточной. Если однородность доступа не используется, эта возможность не нужна. Программы на UPC писать можно и нужно, но, если они должны быть эффективными, писать их придется «в стиле Co-Array Fortran». Какие еще выводы, полезные разработчикам систем программирования и, быть может, даже коммуникационного оборудования суперкомпьютеров, можно сделать из нашего рассмотрения?

Главный вывод лежит на поверхности. Схема разбиения коммуникационного оборудования на классы, приведенная во Введении,

нуждается в уточнении в части двух промежуточных классов. В отдельный класс, порождающий свойственную ему модель программирования, следует выделить ccNUMA-системы. Системы же nccNUMA, с точки зрения модели и стиля программирования, должны быть отнесены в тот же класс, что и сети одностороннего обмена сообщениями. Ведь если доступ в чужую память все равно структурно выделен в программе, уже не так важно, как он записывается: в виде прямого присваивания элементу чужого массива, или же в виде обращения к библиотечной функции копирования значения в чужой массив. Это верно как на программном, так и на аппаратном уровне.

На программном уровне это означает, что для использования нашей промежуточной модели вовсе не обязательно иметь специальные языки. Такие же в структурном отношении программы можно писать с использованием библиотек односторонних обменов (модель put/get). Становится ясно, почему, например, свободно распространяемая реализация UPC выполнена на базе именно библиотек односторонних обменов, таких, как Gasnet, ARMCI и shmem [8, 9]. (Замечание. Речь идет именно о свободно распространяемой реализации, в которой режим сплошной адресации по «распределенному» указателю реализован не эффективно, но не требует NUMA). Становится понятна популярность shmem среди пользователей машин производства Cray, многие из которых исторически являлись nccNUMA-системами [4].

Не остались без полезного вывода и разработчики коммуникационного оборудования. С их точки зрения вывод звучит так: если ccNUMA по тем или иным причинам не получается, вместо nccNUMA можно реализовать быструю сеть односторонних обменов с аппаратной конвейеризацией и агрегацией записи. Специально бороться за nccNUMA не стоит — это оборудование не порождает модели программирования, отличной от модели для хорошей сети односторонних обменов.

Общий вывод звучит примерно так.

Базовой моделью параллельного программирования для современных MPP-систем, дающей программисту возможность почувствовать их преимущество перед слабо оснащенными в коммуникационном отношении кластерами прошлого, является один из вариантов модели PGAS, который с полным на то основанием можно назвать

«моделью shmem». На эту модель имеет смысл ориентировать и программное, и аппаратное обеспечение. На базе shmem хорошо реализуются несколько более высокоуровневые технологии программирования — например, UPC и Co-Array Fortran, при условии, что использоваться они будут для написания программ «в стиле shmem». Сам термин «модель программирования shmem» часто встречается в статьях и руководствах, написанных разработчиками фирмы Cray, которые, очевидно, давно поняли все рассказанное в этом тексте, но боялись сказать вслух.

5. Несколько заключительных замечаний

Библиотеки одностороннего обмена данными традиционно вызывают некоторую оторопь при первоначальном знакомстве у программистов, воспитанных на MPI. Традиционный набор вопросов включает в себя: «а как процесс узнает, что в память соседу уже можно писать»? В отношении OpenMP, например, таких вопросов почему-то не возникает. Все становится на свои места, если понять, что технология односторонних обменов — это не «причудливый и неестественный MPI», а просто несколько тяжеловесный по форме записи способ доступа в единую, распределенную общую память в стиле Co-Array Fortran. Модель эта гораздо ближе по стилю использования к OpenMP, чем MPI, и может быть рекомендована для приложений с двухуровневым параллелизмом (вместо OpenMP+MPI — OpenMP+shmem).

В качестве промежуточной, модель программирования shmem обладает еще одним, несколько неожиданным, преимуществом.

Хорошо известно, что базовые модели для более «сильных» (в смысле рис. 1) классов коммуникационного оборудования часто пытаются использовать на более «слабых» классах в качестве высокоуровневых надстроек, облегчающих жизнь программиста. Например, широко известная библиотека односторонних обменов Global Arrays первоначально была реализована на сети двусторонних обменов — просто потому, что для некоторых задач мыслить односторонними терминами уже тогда казалось удобнее. При таком «прыжке вверх» по спектру классов коммуникационного оборудования велик риск получить неэффективно реализованную систему. Риск этот тем выше, чем длиннее «прыжок». Типичный пример неудачной попытки прыжка через весь спектр — продукт Cluster OpenMP фирмы Intel

[5]. Эта реализация OpenMP на базе сети двусторонних обменов обладает формальной работоспособностью, но вопиюще неэффективна. Характерный допустимый размер зерна параллелизма даже на очень простых обменных шаблонах не меньше, а многократно больше, чем при использовании MPI. Модель shmem, именно в силу своей промежуточности, в качестве такой высокоуровневой надстройки над сравнительно слабой сетью, годится («прыжок» получается не такой длинный). При грамотной программной реализации агрегации и конвейеризации записи даже на базе MPI можно получить вполне работоспособную реализацию shmem. Такого рода программная надстройка, конечно, не может заменить полноценной аппаратной реализации, но в некоторых отношениях может быть полезна на практике, например, для тех же расчетов на неструктурных сетках.

Список литературы

[1] Лацис А. О. Параллельная обработка данных. Москва : «Академия», 2010. ISBN 978–5–7695–5951–8. — 336 с. ↑[], 3, 1, 2
[2] Интерфейс MPI, http://www.mpi-forum.org. ↑[]
[3] Фирма AMD, http://www.amd.com. ↑2
[4] Фирма Cray, http://www.cray.com. ↑4
[5] Фирма Intel, http://www.intel.com. ↑2, 5
[6] Фирма QLogic, http://www.qlogic.com. ↑1
[7] Язык Co-Array Fortran, http://www.co-array.org. ↑2
[8] Язык UPC, http://upc.lbl.gov. ↑2, 4
[9] Сайт Parallel.ru, http://www.parallel.ru. ↑1, 4

A. O. Lacis. *The modern supercomputer programming models.*

Abstract. The modern supercomputer hardware strongly differs in principle from the 5–6-years old hardware. This is true both for the computational hardware and the communication one. The new, better communication hardware has some inherent advantages, that hardly can be properly expressed in terms of the traditional parallel programming technologies. The question of new parallel programming technologies, coming to replace the old ones, is discussed in the article.

Key Words and Phrases: supercomputer, programming model, parallel programming technology, dual-sided transfer network, single-sided transfer network, cache-coherency, latency, aggregation.

Образец ссылки на статью:

А. О. Лацис. *Модели программирования для современных суперкомпьютеров* // Программные системы: теория и приложения : электрон. научн. журн. 2010. № 3(3), с. 73–84. URL: http://psta.psiras.ru/read/psta2010_3_73-84.pdf