Перспективные подходы к созданию масштабируемых приложений для суперкомпьютеров гибридной архитектуры

Климов Ю.А. (ИПМ им. М.В. Келдыша РАН),
Орлов А.Ю. (ИПС им. А.К. Айламазяна РАН),
Шворин А.Б. (ИПС им. А.К. Айламазяна РАН)

Программирование — это по существу планирование и детализация громадного трафика через фоннеймановское узкое горло.
— Д. Бэкус, “Can Programming Be Liberated from the von Neumann Style?”, 1977

Введение

На протяжении всей истории использования ЭВМ для высокопроизводительных вычислений главной задачей программиста является организация оптимального расположения и перемещения данных в памяти машины. Такое положение вещей отражает эпиграф к данной статье, который является цитатой из знаменитой лекции Джона Бэкуса, произнесенной им в 1977 году по случаю вручения ему премии Тьюринга [1]. Именно в этой лекции им было впервые введено понятие «фоннеймановского узкого горла» — канала между вычислительным

устройством и памятью, проблему недостаточной пропускной способности которого впоследствии стали называть «стеной памяти».

В течение нескольких десятков лет для решения этой проблемы было изобретено множество механизмов, таких как многоуровневые ассоциативные кэши, предвыборка, мультитредовость, внеочередное исполнение инструкций и векторные инструкции, все из которых в том или ином виде присутствуют во всех современных высокопроизводительных процессорах. Совершенствование аппаратной поддержки позволило на протяжении многих лет увеличивать производительность вычислений, не меняя существенным образом парадигму программирования. Тем не менее, такое развитие событий привело к появлению иерархии памяти, которую программисту необходимо учитывать для максимально эффективного использования машины. Например, для компьютера на базе современной многосокетной материнской платы, мы имеем, оценивая грубо, три уровня памяти, существенно различающиеся по скорости доступа: кэш, память своего процессора, память чужого процессора.

Переход от последовательных программ к многонитевым, исполняемым в пределах SMP-системы, требует аккуратного разделения данных между нитями. Однако высокие скорости коммуникаций в пределах платы, а также поддержка со стороны компиляторов (OpenMP [9]), делают этот переход относительно несложным.

Гораздо хуже обстоит дело с программированием систем, имеющих распределенную память. Относительно низкие характеристики межузловых сетей привели к появлению новой парадигмы программирования, основанной на передаче сообщений (классическим представителем является библиотека MPI [6]). Таким образом, при создании эффективной программы для типичного современного суперкомпьютера кластерного типа, необходимо иметь в виду не только «фоннеймановское узкое горло», но сразу несколько уровней иерархии памяти и особенности передачи данных внутри них и между ними. Именно это является основной сложностью при написании распределенных параллельных программ, и этим же объясняется тот факт, что для некоторых типичных задач, решаемых на SMP-системах, распределенные версии до сих пор не были созданы (например, некоторые из тестов NAS [5]).

Сложности написания эффективной распределенной программы велики, но обозримы. Для адекватного использования уровней иерархии памяти зачастую прибегают к написанию так называемых «гибридных»[1] программ, использующих MPI на межузловом уровне и OpenMP на внутриузловом. Критическим же с точки зрения сложности программирования стало массовое использование в составе суперкомпьютеров ускорителей вычислений (в первую очередь GPGPU [2]), то есть появление уже по-настоящему, на аппаратном уровне, гибридных машин. Гибридность, то есть неоднородность состава вычислительного узла, вносит в иерархию памяти дополнительные уровни — обмен данными между вычислительными устройствами разного типа, плюс зачастую нетривиальная иерархия памяти внутри ускорителя. Количество переходит в качество — корректно и эффективно запрограммировать перемещение данных в такой расслоенной системе становится титанической задачей. Кроме того, чрезвычайно остро встает проблема балансировки вычислительной нагрузки на различные устройства с учетом их особенностей. В результате на сегодняшний день практически нет примеров решения на гибридных суперкомпьютерах сколь-нибудь нерегулярных задач, таких

[1]Здесь имеется в виду *гибридное программирование*, что не имеет прямого отношения к понятию *гибридная аппаратура*.

как, например, вычисления на адаптивных стеках.

Использование в составе суперкомпьютера вычислительных ускорителей позволяет значительно сократить необходимый размер и, следовательно, цену и стоимость владения машиной при том же пиковом уровне производительности. Не случайно три из пяти самых мощных машин мира на данный момент — на втором, четвертом и пятом местах (по данным списка TOP500 за июнь 2011 года [7]) — построены с использованием ускорителей на базе GPGPU. По мнению многих авторитетных специалистов в области, использование ускорителей — набирающая силу долговременная тенденция. Сама аппаратура ускорителей и способ их интеграции в вычислительные блоки могут претерпеть значительные изменения, но с большой долей уверенности можно утверждать, что суперкомпьютеры эксафлопсного класса будут гибридными. Возможно, на смену GPGPU придут MIC как более легкие в программировании или FPGA как более энергоэффективные или что-нибудь более новое. Однако практически нет сомнений, что программистам придется задействовать для решения задач целый зоопарк разнородных вычислительных устройств со сложной иерархией памяти. Отсюда можно сделать вывод, что проблемы, обсуждаемые в данной работе, не потеряют своей актуальности, и основная задача, которую придется решить на пути достижения реальной эксафлопсной производительности — это, как и многократно в истории вычислительной техники, задача оптимизации перемещения данных в памяти машины.

В следующем разделе мы несколько подробнее остановимся на сложностях программирования гибридных суперкомпьютеров. Затем будет сделана попытка очертить спектр возможных подходов к их программированию и указать наиболее перспективную по мнению авторов область этого спектра, в развитие которой имеет смысл инвестировать усилия.

Почему же так сложно написать эффективную программу для гибридной машины?

Множество уровней параллелизма

Прежде всего отметим, что и в современных, и в перспективных суперкомпьютерах — как в гибридных, так и с однородной структурой узла — имеется несколько уровней параллелизма. Можно выделить по крайней мере узловой, процессорный (уровень сокета), ядерный, векторный, конвейерный виды параллелизма, а также параллелизм уровня инструкций.

Высокой эффективности вычислений (КПД относительно пиковой производительности машины) можно достичь лишь при условии эксплуатации параллелизма на всех уровнях, заложенных в аппаратуре. Для этого при разработке параллельной программы необходимо поделить вычислительную работу на всех уровнях параллелизма (отчасти в этом поможет аппаратная поддержка и компилятор, отчасти программисту придется действовать вручную). Для многих алгоритмов такое разделение сделать сложно. Однако, даже если удалось разделить работу, производительность получившейся программы может быть весьма невысока. Главными причинами низкой производительности являются коммуникационные издержки и неоднородность загрузки (дисбаланс), которые приводят к простою

вычислительных устройств.

- **Коммуникационные издержки.** Задача, которая была поделена для исполнения на отдельных вычислительных устройствах, требует передачи данных между ними. Передача данных по коммуникационной сети происходит не мгновенно, поэтому возникают накладные расходы. Например, процессор не может продолжить работу, пока данные не будут переданы или получены. С ростом количества узлов, процессоров, ядер и других поставщиков параллелизма влияние коммуникационных издержек возрастает. Возникает необходимость разрабатывать параллельные программы таким образом, чтобы общая эффективность не зависела от этих издержек.
- **Неоднородность загрузки.** При разделении общей задачи на подзадачи возможна ситуация, когда работы у одного вычислительного устройства окажется меньше, чем у другого. Тогда первое будет простаивать в ожидании, пока второе выполнит свою часть работы и передаст нужные данные. При небольшом размере установки программист может вручную сбалансировать нагрузку для отдельных устройств. Однако, когда таких устройств много, и они неоднородны (имеют разную производительность, программируются на разных языках, имеют разную дисциплину доступа к памяти), оказывается, что вручную это сделать остается возможным лишь для весьма ограниченного круга задач.

Остановимся подробнее на этих причинах низкой эффективности параллельных приложений.

Множество уровней иерархии памяти

С точки зрения программирования гибридного суперкомпьютера, основная трудность сокрытия коммуникационных издержек заключается в чрезвычайно сложном устройстве памяти.

Наличие иерархии памяти является, по-видимому, фундаментальным свойством вычислительных систем, что привносит определенные трудности на пути к продуктивным и эффективным вычислениям. Можно выделить две принципиально разные идеи, имеющих своей целью преодоление этих трудностей: первая состоит в сокрытии от программиста «неудобных» особенностей архитектуры; другая же, напротив, заключается в том, чтобы явно вынести эти особенности в парадигму программирования и предоставить соответствующие средства для работы с ними. Далее будет рассмотрена конкретизация этих идей по отношению к принципам распределения, перемещения и обработки данных в многоуровневой и, в общем случае, неоднородной системе.

У современных машин можно выделить некоторые из уровней иерархии памяти следующим образом:

- регистры процессора;
- кэши различных уровней (например, L1 — индивидуальный кэш ядра, L3 — общий для всех ядер процессора);
- память процессора;
- память других процессоров узла;
- память других узлов.

Как видим, уровней немало, и, как правило, без потери эффективности не удается скрыть всю иерархию от программиста.

В системе с общей памятью аппаратура автоматически перемещает данные между различными «частями» памяти; например, данные из физической памяти могут автоматически, то есть незаметно с точки зрения программиста,

перемещаться в различные уровни кэша и обратно. В системе же с распределенной памятью перемещение данных между уровнями должно быть явно реализовано программистом.

Современные суперкомпьютеры являются, за редким исключением, системами с распределенной памятью, то есть обладают явно выраженной иерархией памяти. А гибридность вносит дополнительный уровень — каждый ускоритель вычислений имеет свою собственную память. Причем, если речь идет о GPU-ускорителях, то их внутренняя память сама по себе неоднородна, и современные технологии программирования GPU подразумевают, что программист сам должен заботиться не только о копировании данных на GPU и обратно, но также и о рациональном перемещении данных внутри устройства.

Качественно разная организация вычислений на классических процессорах и ускорителях вычислений вынуждает программиста применять разные коммуникационные шаблоны на разных уровнях системы, что значительно усложняет его работу.

Проблемы балансировки

Гибридные суперкомпьютеры обладают вычислительными устройствами разной производительности: процессорами общего назначения и ускорителями вычислений, такими как GPU и FPGA. Более того, для одного и того же алгоритма требуется написать разные программы под разные типы вычислительных устройств. Использование столь разнородных вычислительных устройств по-новому — гораздо острее — ставит проблему балансировки нагрузки на различные устройства.

Следует отметить, что для ряда задач (например, LINPACK [4]) программист может вручную сбалансировать нагрузку, хотя это потребует значительных усилий. А для ряда других задач (например, использующих неявные методы и/или адаптивные сетки) такая ручная балансировка становится невероятно трудной деятельностью.

На классических суперкомпьютерах, в которых процессоры обладали одинаковой производительностью, балансировка зачастую осуществлялась с помощью статического разделения данных между процессорами на равные порции, что обычно давало приемлемую эффективность. На гибридных же машинах для применения аналогичного подхода требуется учитывать реальную производительность каждого вычислительного устройства на каждом из используемых алгоритмов. Например, для получения максимальных значений производительности теста LINPACK требуется многократный запуск данного теста при различных значениях параметров, которые задают распределение данных и вычислений. Для эффективного решения задачи «с первого запуска» такой способ неприменим.

Одним из возможных выходов является использование систем динамической балансировкой нагрузки. Система автоматически (возможно, с некоторыми подсказками от программиста) должна достаточно эффективно загрузить все имеющиеся вычислительные ресурсы.

Возможно, в результате будет получена меньшая производительность, чем при идеальной ручной настройке приложения. Однако необходимо учитывать, что написание программы с возможностью ручной настройки, как и сама ручная настройка и тестовые запуски, занимают много времени. Поэтому, при использовании таких

систем возможен выигрыш как в скорости разработки программ, так и в скорости получения результата.

Традиционные подходы к программированию суперкомпьютеров

Существующие подходы к программированию суперкомпьютеров обладают различной выразительностью — в них по-разному проведена граница между тем, что описывает программист, и тем, что система делает автоматически.

Условно можно выделить две крайности: автоматизировать как можно больше или, наоборот, всё отдать в руки программиста.

Неявное распределение данных и вычислений между процессами, неявное перемещение данных

Наиболее популярной реализацией данного подхода является OpenMP [6] для систем с общей памятью. При использовании данной системы программист не задает ни распределение данных между процессорами, ни перемещение данных, ни распределение работы. Он только указывает, что работа может быть распределена — витки цикла или циклов могут выполняться параллельно. Остальное сделают рантайм-система и аппаратура общей памяти.

Была предпринята попытка перенести данный подход практически без изменений на распределенную память — Intel Cluster OpenMP [10]. Однако данное решение в большинстве случаев показывает крайне низкий результат. Во многих случаях производительность оказывается даже ниже, чем при использовании одного вычислительного узла. Это, в основном, объясняется тем, что характеристики современных связей между узлами суперкомпьютера значительно уступают характеристикам внутриузловых соединений между процессорами и/или ядрами.

С другой стороны, данный подход крайне удобен для программистов, особенно для неспециалистов в области высокопроизводительных вычислений — с помощью совсем небольших изменений OpenMP позволяет превратить последовательную программу в параллельную для многоядерных установок с общей памятью. Поэтому попытки эффективно адаптировать данный подход для систем с распределенной памятью продолжаются. Отдельно отметим преимущество данного подхода в отношении балансировки вычислительной нагрузки: программист не привязывает работу к конкретному вычислительному устройству, и система, таким образом, имеет возможность перераспределять нагрузку оптимальным образом. К сожалению, учет в полностью автоматическом режиме всей иерархии памяти гибридных суперкомпьютеров не представляется возможным, что сводит на нет преимущества данного подхода на системах с распределенной памятью.

Явное распределение данных и вычислений между процессами, явное перемещение данных

С появлением многоузловых, в том числе кластерных, архитектур стало ясно, что первый подход перестает быть эффективным. Какую-то часть работы по распределению и перемещению данных всё же приходится возлагать на программиста. В результате можно сформулировать следующий подход, в некотором смысле противоположный первому — явное распределение, обработка и перемещение данных.

Данный подход получил наиболее широкое распространение на больших машинах — его реализацией является библиотека передачи сообщений MPI. Среди других современных реализаций следует отметить системы SHMEM [7], CAF [8], отчасти Charm++ [11] и другие.

В этих системах программист явно распределяет данные между процессами и явно программирует перемещение данных, например, с помощью функций MPI_Send() и MPI_Recv() в библиотеке MPI.

Данный подход доказал свою работоспособность для однородных вычислительных систем с малым количеством уровней распределенной памяти.

Очевидным недостатком данного подхода является слишком низкий уровень абстракции свойств аппаратуры, приводящий в случае с гибридными суперкомпьютерами к критическому уровню нагрузки на интеллект программиста, что уже обсуждалось выше.

Одним из усовершенствований такого подхода является виртуализация понятия вычислительного устройства. В отличие от примитивного распределения работы по физически процессорам, здесь программист заранее не знает, на каком конкретно вычислителе будет исполняться та или иная часть программы. Такая виртуализация позволяет легко переносить отдельные части программы между вычислительными устройствами и тем самым добиваться 100% загрузки всех имеющихся устройств. В то же время система должна распределять эти части так, чтобы влияние коммуникационных издержек на общую производительность было минимально. Charm++ является хорошим примером системы программирования, где поддержана виртуализация, а также имеется автоматическое и ручное распределение нагрузки.

Альтернативный подход: явное распределение данных и работы между процессами, неявное перемещение данных

Наблюдая историю развития вычислительной техники, можно сделать вывод, что первый подход (неявное распределение, перемещение и обработка данных) уже исчерпал себя или близок к тому. Многие замечательные механизмы такие как кэш, предвыборка, мультитредовость, и т.д. перестают справляться с проблемой стены памяти, которая неуклонно обостряется при масштабировании вычислительных систем и добавлении неоднородности. Так, например, накладные расходы по поддержанию когерентности кэша в рамках многоузловой системы возрастают настолько, что ставят под вопрос целесообразность ее использования.

Второй подход (явное распределение, перемещение и обработка данных) тоже сталкивается с серьезными трудностями, вызванными усложнением структуры распределенной памяти (за счет ускорителей), что влечет за собой радикальное усложнение программирования и, следовательно, снижение продуктивности.

Можно заметить, что оба предложенные выше подхода являются, в некотором смысле, экстремальными — делать всё явно либо всё неявно. Очевидно, существуют промежуточные подходы, и имеет смысл исследовать их, чтобы попытаться найти

более адекватно соответствующие архитектуре гибридных суперкомпьютеров.

Предлагаемый альтернативный подход подразумевает автоматизацию перемещения данных и балансировку нагрузки, в то время, как основной заботой программиста становится обработка локальной порции данных на том или ином типе вычислительных устройств.

Заметим, что многие вычислительные задачи обладают общим коммуникационным шаблоном, а это позволяет один раз реализовать механизмы перемещения данных в рамках проблемно-ориентированной библиотеки, которая будет многократно использоваться для реализации целого спектра вычислительных задач. При использовании такой библиотеки программист должен будет реализовать вычислительную часть своей задачи, выбрать и, возможно, незначительно адаптировать библиотечный коммуникационный шаблон. Остальное возьмет на себя библиотека и рантайм.

Библиотеки такого рода должны содержать наиболее часто используемые коммуникационные шаблоны с возможностью их настройки под конкретную задачу и конкретную аппаратуру. Это позволит освободить программиста от коммуникационной составляющей на начальном этапе разработки приложения и и даст возможность сосредоточиться на вычислительной составляющей. Впоследствии, если производительность коммуникационной библиотеки окажется недостаточной, программист сможет изменить или даже полностью переписать реализацию коммуникационного шаблона.

В результате использования такого рода библиотек значительно повышается продуктивность разработки эффективных программ, предназначенных для исполнения на гибридных суперкомпьютерах, уменьшается число ошибок (ведь значительное количество ошибок обычно связано именно с перемещением данных), а также повышается сопровождаемость и переносимость программ.

Данный подход на сегодняшний день является не столь популярным, как MPI, но всё же существует некоторое количество работ в этом направлении. В качестве примера можно привести DVM-систему [12, 13], где программист с помощью специальных директив (примерно в том же духе, что в OpenMP) задает распределение данных и вычислений (витков цикла) по процессорам и узлам кластера, в то время как характер перемещения данных не программируется универсальным образом, а выбирается как один из небольшого количества стандартных параметризованных шаблонов. Основным недостатком DVM-системы является статическое задание распределения данных. Это значительно затрудняет ее применение для нерегулярных задач, а также поднимает вопрос балансировки нагрузки в пределах гибридного узла (однако авторы DVM-системы ведут работы в этом направлении в рамках проекта DVMH [14]).

Другим примером может быть система StarPU [15], предназначенная для программирования гибридной вычислительной машины в пределах одного SMP-узла. В данной системе программист определяет отдельные вычислительные процедуры, называемые «codelet», способные выполняться как на классическом процессоре, так и на ускорителе вычислений. Затем задается разделение данных на отдельные фрагменты и описывается их обработка с помощью codelet-ов. Рантайм-система анализирует зависимости между вызовами codelet-ов и организует вычисление

codelet-ов на всех доступных вычислительных устройствах с учетом как времени выполнения каждого codelet-а на различных устройствах, так и времени передачи необходимых данных на выбранное устройство. Таким образом программист не думает о балансировке и загрузке различных по своей природе вычислительных устройств, он описывает только вычислительные процедуры (codelet-ы) и связи между ними. Эффективной загрузкой вычислительных устройств в рамках одного узла занимается рантайм-система StarPU. Для использования нескольких гибридных узлов кластера StarPU предлагает использовать немного видоизмененную библиотеку MPI, то есть передача данных между узлами и балансировка загрузки узлов полностью программируется разработчиком, что не всегда является удовлетворительным решением.

В общем виде перспективный с точки зрения авторов подход может быть описан следующим образом.

- Программист разбивает данные на порции и описывает вычисления, производимые с одной такой порцией. Данное разбиение должно быть не фиксированным, а параметрическим — для реализации балансировки рантайму необходимо иметь возможность разбить задачу на порции удобного размера, чтобы эффективно загрузить все вычислительные устройства. В некоторых случаях полезно предоставить рантайм-системе возможность изменять разбиение данных в процессе вычислений. Обработка данных может быть описана двояко — в виде версии для CPU и версии для ускорителя, чтобы позволить рантайму распределять каждую такую порцию на соответствующее вычислительное устройство в динамике.
- Программист выбирает один из коммуникационных шаблонов, предлагаемых библиотекой, и указывает его параметры. Полезно также предоставить возможность изменять библиотечный шаблон или создавать новый на тот случай, если подходящего не нашлось в библиотеке.

В результате программист фокусирует свое внимание на эффективном выделении локальной порции данных и ее обработки. Задачей же рантайм-системы является организация передачи необходимых данных в распределенной системе памяти и балансировка нагрузки на вычислительные устройства.

Заключение

Подытоживая, перечислим описанные в статье основные подходы к созданию параллельных приложений, два из которых являются уже традиционными, а третий пока сравнительно редко применяется:

1. Распределение, обработка и перемещение данных делаются неявно.
2. Распределение, обработка и перемещение данных делаются явно.
3. Распределение и обработка данных делается явно, а перемещение — неявно.

Процесс развития вычислительных систем иногда требовал смены парадигмы программирования. Это выражается в том, что прежние, уже привычные, подходы становятся неэффективными или недостаточно продуктивными. Можно сказать, что эволюционное развитие проходит через периоды кризисов. В настоящее время, в том числе в связи с появлением гибридных машин, назревает очередной такой кризис, который необходимо преодолеть на пути к вычислениям эксафлопсного масштаба.

Можно сделать вывод о необходимости искать альтернативные пути развития методов и подходов к программированию новых и существующих типов аппаратуры. Одним из таких путей видится третий подход. Пока не вполне ясно, насколько он будет хорош на практике, однако, даже если время покажет его состоятельность, это не отменяет необходимости исследования новых направлений.

Благодарности

Авторы выражают искреннюю благодарность Алексею Лацису за интереснейшие и плодотворные обсуждения того видения современного состояния и тенденций развития высокопроизводительной вычислительной техники, на которое опирается эта небольшая работа.

Работа выполнена при поддержке Минобрнауки России (госконтракт № 07.514.11.4014).

Литература

1. John Backus. *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs* // Communications of the ACM. Volume 21. Number 8. Pages 613-641. August, 1978. ACM New York, NY, USA. DOI: http://doi.acm.org/10.1145/359576.359579. URL: http://www.stanford.edu/class/cs242/readings/backus.pdf .
2. *Графические процессоры общего назначения* // URL: http://gpgpu.org/
3. *Список Top500 за июнь 2011 года* // URL: http://top500.org/list/2011/06/100
4. *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers* // URL: http://netlib.org/benchmark/hpl/
5. *Набор тестов NAS* // URL: http://www.nas.nasa.gov/Resources/Software/npb.html
6. *Библиотека MPI* // URL: http://www.mpi-forum.org/ .
7. *Библиотека SHMEM* // URL: http://www.shmem.org/ .
8. *Coarray Fortran 2.0 at Rice University* // URL: http://caf.rice.edu/
9. *Стандарт OpenMP* // URL: http://openmp.org/wp/
10. *Intel Cluster OpenMP User's Guide* // URL: http://software.intel.com/file/6330
11. *Parallel Languages/Paradigms: Charm ++ - Parallel Objects* // URL: http://charm.cs.uiuc.edu/research/charm
12. *DVM-система* // URL: http://www.keldysh.ru/dvm/ .
13. В. А. Крюков. *Разработка параллельных программ для вычислительных кластеров и сетей* // Информационные технологии и вычислительные системы. № 1-2, 2003 г., стр. 42-61. URL: http://www.keldysh.ru/dvm/dvmhtm1107/publishr/cldvm2002web.htm
14. *DVMH — DVM для гетерогенных систем* // URL: http://www.keldysh.ru/dvm/dvmhtm1107/rus/dvmh.html
15. *StarPU — унифицированная рантайм-система для гетерогенных многоядерных архитектур* // URL: http://runtime.bordeaux.inria.fr/StarPU/