Возможности оценки сложности параллельного программирования

В.Л. Авербух,[1] М.О. Бахтерев,[1] А.Ю. Казанцев,[1] В.В. Косенко[2]

ИММ УрО РАН[1], Уральский Государственный Университетет[2]

Работа предлагает постановку задачи оценки параллельного программирования. Рассматривается ряд подходов к оценке сложности параллельных программ, а также оценки необходимых трудозатрат программиста для создания надежного и эффективного параллельного кода.

1. Введение. Метрики программирования

Утверждение о том, что параллельное программирование сложно, стало общим местом в соответствующей специальной литературе еще с 80-ых годов XX века. Вместе с тем, необходимо разобраться, чем же оно сложно и как в этом плане соотносятся различные парадигмы параллельного программирования. Анализ сложности программирования полезен, прежде всего, при выборе инструментария для разработки новых программных комплексов. Одновременно интерес к такому анализу проявляют создатели новых средств параллельного программирования.

Известны основные аспекты языка программирования применимые к любому языку. Прежде всего, это уровень и охват (широта области применения) [1]. Уровень языка можно определить как универсальную меру количества деталей, которые программист должен указать для достижения желаемого результата. Язык является процедурным, если пользователю нужно последовательно, по шагам определять задание. Количество и размер требуемых шагов различается в разных процедурных языках. Для достижения одного и того же результата в высокопроцедурных (низкоуровневых) языках, например, ассемблерах требуется много малых детализированных шагов, а в низкопроцедурных языках (языках высокого уровня), требуется заметно меньше намного более крупных шагов с значительно меньшей детализацией.

Охват (широта области применения) языка подразделяет языки от языков общего или широкого назначения до узко специализированных языков.

Отметим, что ассемблер, в общем-то, может быть использован практически, в любых случаях. Таким образом, у него наиболее широкий охват, он имеет большую область применения чем, все остальные языки. Однако это не делает его наиболее удобным средством программирования.

В литературе хорошо представлены методы измерения качества и сложности программ.

Наиболее простыми являются количественные метрики программы, такие как среднее число строк для функций (классов, файлов) или среднее число строк, содержащих исходный код для функций (классов, файлов). Существуют подходы к оценке к количественной оценке сложности понимания программы, трудоемкость кодирования программы, уровню языка. Для анализа сложности программирования в рамках метрик Холстеда [2] используется понятие информационного содержания программы и оцениваются необходимые интеллектуальные усилия при ее разработке. Также существует метрика, характеризующая число требуемых элементарных решений при написании программы. Выведены формулы для данных метрик, в которые подставляются параметры уже готовых программ. Существуют также подходы к разработке метрик параллельных программ и метрик производительности параллельных программ. В последнем случае метрикой называется зависящая от времени функция, характеризующая некоторые аспекты производительности параллельной программы. Примеры простых метрик производительности - нагрузка центрального процессора, использование памяти, число операций с плавающей запятой. Надо учитывать также измеримые (объективные) показатели, например, скорость программирования, количество ошибок и время, необходимое для их обнаружения, время, необходимое для получения результата (работающей программы). Существуют и субъективные параметры типа “эстетического” удовлетворения процессом программирования, уровня утомление и пр. Среди возможных составляющих метрик сложности следует учитывать субъективные оценки того или иного языка. Можно отметить, что эстетическое удовлетворение от программирования получается, когда программа просто написана — код красиво смотрится, ничего лишнего (минимизированы абстракции) — и понятно на взгляд что происходит без

дополнительных комментариев. Эмоциональное удовлетворение возможно связано с простотой написания программы, когда минимизировано внимание к техническим сторонам программирования. Более объективным является учет таких параметров как длина программы, скорость ее написания, скорость отладки и пр.

В связи с нашими целями мы хотим получить не метрики качества или производительности уже готовых программ, а сравнение того насколько одна парадигма параллельного программирования сложнее или проще другой.

Для характеристики прагматических аспектов языков используются понятие адекватности. В случае средств параллельного программирования можно использовать понятие адекватности в описании параллелизма.

2. Явный и неявный параллелизм

Различается явный и неявный параллелизм. Неявный параллелизм программы учитывается распараллеливающими компиляторами. Распараллеливание компилятором представляется очень интересным. Мы пишем простой последовательный код, а компилятор за нас его преобразует в “правильный” параллельный. Однако на практике, чтобы компилятор мог что-либо распараллелить, нужно учитывать модель будущего параллелизма. Для получения эффективно работающей программы необходимо знание о методиках распараллеливания, которые использует данный компилятор. В тоже время больших результатов в плане эффективности может достичь специалист, не только владеющий навыками параллельного программирования, но и обладающий знаниями в данной предметной области, разбирающийся в сути использованных вычислительных методов и алгоритмов.

Существует возможность реализовать неявное распараллеливание за счет функциональных языков программирования. Распараллеливание основано на семантике понятия функции, не требующей вычисления ее аргументов в каком-то определенном порядке. Однако, отсутствие “привычного” императивного порядка исполнения порождает сложности в реализации ввода/вывода. В рамках функционального программирования нет адресов, указателей, перезаписываемых областей памяти, а только управление, данные и их метки.

Явный параллелизм задается создателем алгоритма и программистом. На всех этапах человек должен держать в голове ответ на вопрос “где и как это будет работать параллельно”. В рамках различных парадигм параллельного программирования существуют как весьма “актуальные” и широко известные и повсеместно используемые в настоящее время библиотеки - MPI и OpenMP, так и менее “актуальные” (по разным причинам), например, High Performance Fortran (HPF) (и близкие к нему среды программирования) или Cilk [3].

Можно заметить, что явный высокоуровневый параллелизм по данным (например, HPF) прост в использовании, но применим не для всех приложений. Более современные многонитевые среды (например, OpenMP) – несколько сложнее в использовании. Можно также указать на накладные расходы при обеспечении синхронизированного доступа к общей памяти и на недостаточную масштабируемость программ. При реализации Cilk’а была проведена большая работа по уменьшению накладных расходов. Авторы добивались удобства и простоты использования среды параллельного программирования. Вместе с тем, существуют проблемы переносимости на системы типа NUMA, связанные с моделью памяти, и на кластерные комплексы.

Поддерживающий парадигму обмена сообщениями MPI можно отнести к низкоуровневым средам (хотя и более высоким, чем прямое распараллеливание при помощи операторов fork/join).

Ниже мы проанализируем возможность оценки различных аспектов языков программирования для некоторых “параллельных” парадигм.

3. Анализ параллелизма

Каковы же возможные критерии для качества параллельного программирования? На что необходимо обращать внимание при проектировании средств параллельного программирования, и при их анализе? Сперва отметим, что не бывает операций без данных (или ещё точнее, без памяти, в которой эти данные записаны), а данные без операций существуют. Представляется, что это часть ментальной модели любого программиста. Если мы посмотрим на уровень интерфейса процессора, то увидим, что любая операция им совершаемая совершается на уровне работы с некоторыми областями хранения данных. Это могут быть регистры, биты состояния машины,

ячейки памяти. То есть, имеет место создание кода в том стиле, когда в исходном тексте указываются и операции, и указания на то, как получить данные для этих операций. Переменная не обязательно должна ассоциироваться с конкретной ячейкой памяти, она может быть плавающей, но имя переменной для компилятора всегда означает указание на то, как извлечь данные. Так обстоят дела в широко используемых языках программирования, будь то императивные языки или функциональные. Логические языки отрывают нас от простой ментальной конструкции (***у меня есть данные, я хочу их преобразовать в другие***), заменяя её более сложной конструкцией (***у меня есть логические высказывания, я хочу сделать из них всевозможные выводы и посмотреть, смогу ли я вывести из них некое другое высказывание***). Для некоторых задач такой уровень абстракций вполне адекватен, но многие алгоритмы не выражаются напрямую таким образом. (Или выражаются, хоть и понятно, но очень громоздко.)

Попытки отделить поток управления от данных предпринимались неоднократно. Такие языки действительно дают очень короткий код, действительно позволяют этот код повторно использовать. Однако так можно задавать лишь достаточно примитивные схемы композиции операций.

Распараллеливание подразумевает следующие действия: (1) выделение независимых участков кода; (2) описание (неким образом) процедуры получения обрабатываемых данных в этих участках кода; (3) независимое получение некоторых наборов данных, результатов параллельных шагов вычисления; (4) описание процедур сбора этих данных.

Когда мы пишем последовательные программы, мы не задумываемся о независимости и передачи данных, хотя в реальности и в этом случае происходит множество внутренних передач данных (современный процессор - это сложная сеть). Работу по организации передач, руководствуясь именами переменных, берёт на себя компилятор, а потом и процессор (руководствуясь кодом). Код явно не разбивается на независимые (напомним, что независимость и параллельность в контексте программирования - синонимы) части. Однако достаточно независимыми участками являются функции, на чём и построена основная концепция таких систем, как Cilk. Не особо заостряя внимание на этом, программист в традиционных языках весьма активно разбивает код на независимые участки, разбрасывает по ним данные, а потом собирает эти данные вместе.

Код не надо разбивать на независимые участки, потому что в программе определяются функции, то есть, в контексте последовательной программы, всё это делается неявно. Относительная свобода и легкость кодирования в этом случае связана с тем, в описание операций можно свободно вставлять ссылки на данные (имена переменных), которые, собственно, являются тоже кодами определённых операций для компилятора.

Отправная точка нашего анализа заключается в идее ***оценки уровня связи операторной части программы с данными***. Благодаря свободной манипуляции именами переменных у нас есть возможность достаточно легко производить декомпозицию и композицию привычных, последовательных программ. Именно она постоянно проводится при разбиении кода на строки, процедуры, функции, при перестановки всего этого местами, при распределении по модулям, и т.п. По нашему мнению такая свобода получается благодаря возможности свободно ссылаться на данные в выражениях. Для параллельного программирования ключевой момент заключен в композиции и декомпозиции кода.

Рассмотрим с позиций предложенного критерия известные среды MPI и OpenMP.

4. Анализ MPI

MPI своей системой демонов, библиотек, интерфейсов и пр. позволяет:

(1) абстрагировать программу от конкретной операционной системы и её особенностей и “тонких” мест, возникающих при работе со средой передачи данных (сетью). В рамках одной операционной системы абстрагироваться можно от конкретной сетевой среды.

(2) автоматически организовывать запуск процессов пользователя на машинах MPI-кластера, автоматически их нумеровать и объединять в MPI-сеть. Это, резко упрощает программирование по сравнению с низкоуровневым использованием tcp/ip-сокетов.

Важно также то, что пользователю позволяется писать всю программу в виде одного исполняемого файла, и запускать её на всей системе.

(3) MPI практически всё взаимодействие сводит к двум основным операциям обмена данными Send (*отправить*) и Recv (*получить*). Отправка и получение происходит на уровне сообщений.

Сообщения различаются системой по тройке (номер процесса-отправителя (rank), номер коммуникатора (comm), тэг (tag)). Чтобы получить сообщение, отправленного при помощи вызова Send(rank, comm, tag, ...), нужно использовать вызов Recv(rank, comm, tag,...) (*Запись вызовов схематична.*)

Благодаря MPI стало возможно сравнительно легко программировать для распределённых систем. Именно его использование открыло дорогу супервычислениям. Однако программирование на MPI низкоуровневое и требует существенных усилий для того, чтобы превращать ***данные*** для одних процессов в ***данные*** для других процессов. При этом, коды производящие это преобразование (управление памятью + сам MPI + схемы согласования сообщения) негибки. Перекраивать архитектуру вычисления очень сложно. Имеют место также “потери” в работе с данными по сравнению с традиционным последовательным программированием. С MPI мы теряем ссылки на данные. Более того, в какой-то мере мы теряем саму концепцию данных. Дело в том, что, сообщения не ведут себя как данные. Они ведут себя как нечто, что переписывается из адресного пространства одного процесса в адресное пространство другого. И после удачного переписывания исчезает. Сообщение - это не данные. Содержимое сообщений не может прямо использоваться в формулах, описывающих наше вычисление. Нельзя свободно модифицировать и составлять код, используя ссылки на эти данные в нужных местах программы, для того чтобы связывать процесс вычисления воедино. Сообщения существуют только в момент коммуникации, а превращать их в данные, которыми можно оперировать в выражениях - это уже задача программиста. Процедура этого превращения не такая простая: нужно оперировать памятью, оперировать MPI, и как-то устанавливать в двух процессорах соответствие между тройками (rank, comm, tag) и ссылками на доступные для обработки данные. В результате, при программировании на MPI мы очень сильно теряем в возможностях по композиции и декомпозиции программ. Теоретически, там полная свобода, но практически эта свобода даётся слишком большими усилиями, и поэтому, при практическом использовании MPI очень часто применяется ограниченно, с использованием только гораздо менее эффективных механизмов широковещательной рассылки данных типа Allgather. Анализируя популярность этого метода с позиций нашей основной идеи, можно сказать, что это самый простой способ превращать разбросанные по процессам блоки данных, в данные, которые достаточно свободно можно использовать в вычисляющих выражениях. (Просто в каждом процессе всё собирается в один буфер, свой для каждого процесса, но являющийся копией всех таких буферов в системе.)

Таким образом, достоинства MPI связаны с тем, что он скрывает сетевую сложность. Недостатки связаны с слишком большими трудозатратами, требующимися для того, чтобы превращать поток сообщений в данные.

5. Анализ OpenMP

Что даёт нам использование OpenMP? Прежде всего, (кажущуюся?) свободу от параллельного программирования. Как мы уже указывали, основная идея заключается в том, чтобы возложить работу по генерации кода для параллельных вычислений с общей памятью на компилятор, оставив программисту обязанность давать компилятору подсказки. Некоторые компиляторы могут включать режим трансляции с распараллеливанием даже без участия программиста, сами разбираясь в том, какие участки можно выполнять параллельно, какие нет, какие надо немного трансформировать, используя, например, политопные модели. Важным достоинством OpenMP является то, что мы можем взять уже существующую последовательную программу и попытаться проинструктировать компилятор о том, какие участки кода можно параллельно выполнять, а какие нет.

При этом модель выполнения у OpenMP достаточно проста. Есть основной поток исполнения, который, доходя до параллельного кода, запускает несколько нитей. Эти нити выполняют параллельный код. Основной поток исполнения тоже выполняет определённую часть вычислений. Потом все нити опять сходятся в одной точке, из которой продолжается выполнение последовательное. Имеет место использование простого принципа fork/join.

Укажем, однако, на ограничения данного подхода. OpenMP работает только с кодом. Он не занимается особым анализом данных. Нет возможности данные как-то перемешивать. Предполагается, что основная задача заключается в создании кода. А на данные особого внимания не обращается, тем более, они всё равно находятся в общей памяти и легко доступны.

OpenMP ограничивает контроль программиста и его свободу в манипулировании потоками данных и в композиции параллельных участков. Понижается понимание происходящего, потому что принципы распараллеливания достаточно сложны, а его детали скрыты. Особое внимание необходимо уделять управлению памятью. Сложно отлаживать ошибки, возникающие при директивах компилятору преобразовать некий принципиально “непараллельный” код в параллельный. (Хотя какие-то возможности предохраняющие от этого существуют.) Более того, в рамках OpenMP простые концепции (например, запустить процедуру умножения блоков матрицы на N процессов, с определенными блоками для процесса i) приходится выражать непрямым способом. Необходимо правильно расположить данные в массивах, дать правильные указания на распараллеливание цикла, а потом надеяться, что это даст эффект.

Кроме того, существуют известные архитектурные ограничения. Как известно, OpenMP не приспособлен для работы на кластере. Приходится его использовать вместе с MPI, что усложняет всё дело. MPI требует виртуозной работы с буферами в памяти, OpenMP скрывает детали кода, который с этими буферами работает, и надо как-то подгонять первое к неявным правилам работы второго.

6. Заключение

Как уже отмечалось, нас интересует возможность оценки сложности параллельного программирования. Именно программирования, а не самих готовых параллельных программ, несмотря на то, что и эти метрики также представляют значительный интерес. Наша работа представляет собой постановку задачи на исследование.

Кроме рассмотренных выше основных парадигм параллельного программирования существуют и другие подходы к распараллеливанию процессов.

Среди них следует обратить внимание на язык bluespec [4], в принципе созданный для проектирования электронной аппаратуры, но используемый и для параллельного программирования.

В ряде источников он характеризуется как функциональный язык, но это не совсем верно. Bluespec – пример языка асинхронного программирования (или иначе - программирования по событиям). В нем изначально заложена параллельная модель вычислений.

Программа на bluespec language содержит модули, имеющие интерфейсы. Модули содержат набор правил, с помощью которых описываются события, вызывающие реакцию программы. В языке поддерживается двухуровневая параллельность - на уровне модулей (код разных модулей может выполняться одновременно) на уровне операций внутри правил (если используем параллельную композицию 'a | a'). Модуль рассматривается как некий ресурс. То есть для некоторой задачи модуль X пишется так, чтобы на практике можно было использовать сразу несколько таких модулей одновременно. Разработчики языка bluespec, в частности, сообщают, что текст на этом языке на порядок короче и понятнее, чем на C++, хотя сама программа уступает коду на C++ по эффективности.

Еще одна парадигма параллельного программирования предлагается в рамках проекта RiDE [5]. Рассматривается методика и технические средства для программирования в параллельных распределённых средах. Цель разработки - упростить процесс создания параллельных программ, и сделать это не в ущерб эффективности исполнения вычислительных кодов. RiDE - это низкоуровневая система, обеспечивающая, несмотря на эту низкоуровневость, перенос языка композиции/декомпозиции с именованием данных и вычислений на уровень распределённой системы. Естественная декомпозиция операций, выраженная достаточно явным образом на языке RiDE, затем выполняется средой программирования по возможности параллельно. Сама декомпозиция может быть и не параллельной. Она может выполняться последовательно из-за специфичного потока данных в ней, но это не выходит за рамки RiDE. Параллельное же исполнение возникает само собой естественным образом, если это позволяют зависимости по данным. Конечно, параллелизм и здесь зависит от действий программиста. Программист может контролировать процесс исполнения, он знает архитектуру распараллеливания. Интерфейс к ней хоть и, не такой прямой, как в MPI, но всё же позволяет управление этим процессом. Прототип среды программирования показал работоспособность заложенных в ее основу идей. Интерес представляет большая компактность кода по сравнению с кодом, написанном на MPI.

В связи с анализом процесса распараллеливания возникает ряд соображений, связанных с визуальными языками параллельного программирования. Казалось бы, визуальное представление

параллельных программ жизненно необходимо из-за сложности самого параллельного программирования. Работы в данном направлении начались, практически, одновременно с разработкой систем визуального программирования [6] и достаточно активно продолжаются по сей день. Однако серьезных результатов не наблюдается. Очевидно, что попытки полностью перевести все операции параллельных языков в визуальную форму (как это было сделано в [7]) не дают желаемого эффекта. Возможно, что причины неудач связаны с тем, что визуализация должна быть адекватной задаче и описываемому процессу, а универсальный, чисто визуальный язык может породить излишнюю сложность и абстракцию в конкретной задаче. Не ясно как визуально отображать действия и операции. Возникают также проблемы с визуальным представлением данных, так необходимых для гибкой композиции вычислительных модулей.

Мы проанализировали возможности оценки распараллеливания с позиций оценки уровня связи операторной части программы с данными. Оценка сложности программирования в рамках той или иной парадигмы является важной, (хотя и вспомогательной) задачей, связанной с выбором инструментария программирования, с оценкой направлений развития области разработки сред параллельного программирования.

Кроме рассмотренных подходов к анализу параллелизма в рамках различных парадигм параллельного программирования, возможны и другие методики исследования, например, изучение ментальных моделей распараллеливания и методов их отображения в программу. Эти вопросы связаны с проблематикой психологии программирования и требуют дополнительного изучения.

Нами предполагается дальнейший анализ проблем параллелизма и поиск возможных способов оценки средств параллельного программирования.

Литература

1. Холстед М.Х. Начало науки о программах. М.: Финансы и Статистика, 1981.
2. Chang S.-K. Visual Languages: A Tutorial and Survey // Visualization in Programming. (Lecture Notes in Computer Science 282). Berlin. Springer-Verlag. 1987. Pp. 1-23.
3. Frigo M., Leiserson C. E., Randall K.H. The implementation of the Cilk-5 multithreaded language // PLDI '98: Proceedings of the ACM SIGPLAN 1998. 1998. New York, pp. 212-223.
4. Bluespec. Courses and Research Papers and Articles. http://www.bluespec.com/why-bluespec/technology-references.htm.
5. Бахтерев М.О., Васёв П.А. Методы распределённых вычислений на основе модели потока данных. Прототип системы // Тезисы XII Международного семинара «Супервычисления и математическое моделирование». Саров, РОСАТОМ, 2010, стр. 14-16.
6. Pong M.C. A Graphical Language for Concurrent Programming // Proceeding of the 1986 IEEE Workshop on Visual Languages (June 1986), pp. 26-33.
7. Al-Mulhem M., Ali Sh. Visual Occam: Syntax and semantics // Comput. Lang. 1997, V. 23, N 1, pp. 1-24.