Лекция 1. Введение

О названии курса

Текст, предлагаемый вашему вниманию, — это не методическое пособие и не методические указания, не конспект полного курса лекций и не учебное пособие. Это некоторые учебные материалы, которые, как я надеюсь, помогут вам полноценно использовать современные высокопроизводительные вычислительные системы. Сами материалы появились как обобщение моего личного опыта работы с такими системами и результатов изучения большого количества учебной и научной литературы. Естественно, что многие удачные идеи, цитаты и примеры из других учебников, вписавшиеся в концепцию моего изложения, перекочевали в данный текст. В конце каждой лекции приводится список источников, использованных для ее подготовки.

Курс, трижды менявший название, читался на протяжении ряда лет в разном формате и в разных местах. Сначала это был курс для двух полугрупп факультета управления и прикладной математики МФТИ под названием «Методы параллельной обработки данных», затем курс приобрел статус межбазового и стал называться «Параллельное программирование», а тот же самый курс, прочитанный в РГГУ им. Канта, шел под вывеской «Методы параллельной обработки информации на суперкомпьютерах». Ни одно из названий, на самом деле, полностью не отражало сути прочитанных курсов.

Конечно, речь всегда шла о суперкомпьютерах, о параллельном программировании и о методах использования и того, и другого. Но слова «суперкомпьютер», «методы» и «программирование» я бы хотел исключить из названия того, о чем я рассказываю.

Для этого есть несколько причин.

В последнее время в средствах массовой информации часто повторяются слова «суперкомпьютеры» и «супервычисления». Обычно читателям понятно только то, что супервычисления, как правило, проводятся на суперкомпьютерах. Попробуем разобраться, что это такое.

Начнем с термина «суперкомпьютер» [1]. Что подразумевается под этим термином? В 1986 году в Оксфордском толковом словаре по вычислительной технике было дано определение: «Суперкомпьютер — это очень мощная ЭВМ с производительностью более 10 миллионов операций с плавающей точкой в секунду (10 MFlops)». В начале 90-х годов прошлого века в это определение была внесена поправка — не 10 MFlops, а 300 MFlops. В 1996 году поправку обновили — уже не 300 MFlops, а 5 GFlops (5 миллиардов операций в секунду). Если каждые пять лет в определение понятия «суперкомпьютер» необходимо

вносить исправления, то, может быть, что-то неправильно с самим определением? Производительность вычислительных комплексов постоянно увеличивается (спасибо разработчикам аппаратного обеспечения - hardware!), и то, что еще вчера считалось совсем недоступным, сегодня — уровень домашних компьютеров. Производительность самых мощных компьютеров ныне измеряется терафлопсами – один терафлопс (Tflops) составляет 10^{12} операций с плавающей точкой в секунду.

По мере своего развития вычислительные системы постоянно дешевеют. Те компьютеры, которые вчера еще были доступны единицам пользователей, сегодня становятся доступными тысячам. К этому ведет вся логика эволюции вычислительных систем. Наиболее развитые компьютеры обладают максимальной стоимостью. Поэтому для суперкомпьютеров появилось следующее определение: «Суперкомпьютер — это вычислительная система, стоимость которой превышает 1–2 миллиона долларов». Однако при внимательном рассмотрении не каждая вычислительная система с высокой стоимостью является суперкомпьютером. В Интернете регулярно встречаются сообщения о дорогостоящих компьютерах с клавиатурами и мышками из чистого золота, инкрустированных бриллиантами и полированными костями тиранозавров. Стоимость их очень большая, но являются ли они при этом суперкомпьютерами?

По определению Гордона Белла и Дона Нельсона, суперкомпьютер – это любой компьютер, масса которого больше одной тонны. Значительная доля истины в этом определении есть, так как современные вычислительные системы, сильно отличающиеся по соим мощностям от «персоналок», стремительно прибавляют в весе. Причем, значительную часть составляет инженерное обеспечение, например. Тяжелая и дорогостоящая система охлаждения. Для такого мощного комплекса уже не хватает нескольких вентиляторов, как в персоналке.

Есть еще два определения суперкомпьютера, которые не позволяют отнести то или иное устройство к классу суперкомпьютеров, но обладают общефилософским смыслом.

Одно из них гласит: «Суперкомпьютер — это компьютер, мощность которого всего на порядок меньше мощности, необходимой для решения современных задач». Оно иронически определяет суперкомпьютер с точки зрения обычных пользователей. Второе, сформулированное в 2001 году известным разработчиком вычислительных систем Кеном Батчером (Ken Batcher), звучит так: «Суперкомпьютер — это устройство, сводящее проблему вычислений к проблеме ввода/вывода».

Увы, мы так и не пришли к нормальному определению суперкомпьютера. Так что в дальнейшем будем полагать, что суперкомпьютер – вычислительная система с большим

числом процессоров (счетных ядер), обладающая производительностью, на несколько порядков большей, чем у персональных компьютеров.

А нужно ли все это?

В последние годы во всем мире наблюдается бум построения мощных вычислительных систем. Страны, научные организации и университеты соревнуются за попадание в верхние строки рейтингов производительности. Особенно ярко эта тенденция проявляется в России. Вот несколько наиболее мощных компьютеров из TOP-50 нашей страны на сентябрь 2011 года [2]:

1. Научно-исследовательский вычислительный центр МГУ имени М. В. Ломоносова — компьютер «Ломоносов» — 1373,06 TFlop.
2. РНЦ Курчатовский институт — 123.65 Tflop.
3. Южно-Уральский Государственный университет — 117.64 Tflop
4. Межведомственный суперкомьютерный центр РАН — 123.88 TFlop.
5. Москва (государственный сектор) — 88,77 TFlop
6. Научно-исследовательский вычислительный центр МГУ имени М. В. Ломоносова — компьютер «Чебышёв» — 60 TFlop.

…

11.Москва, Институт прикладной математики им. М.В.Келдыша РАН (К-100) — 107.9 Tflop

Здесь указана только пиковая производительность компьютера. Суперкомпьютер из Екатеринбурга занял более высокое место, чем суперкомпьютер из МСЦ потому, что при чуть более низкой пиковой производительности он показал лучшие результаты по тестам на пакете Линпак (LINPAK - стандартный пакет для решения очень больших систем линейных алгебраических уравнений, обычный тест для любого суперкомпьютера). Компьютер из Института прикладной математики имени Келдыша К-100 из-за особенностей архитектуры показал невысокие результаты на пакете Линпак и занимает в рейтинге только 11 место.

Наиболее мощными в мире [3] сейчас считаются японский вычислительный комплекс RIKEN — 8.776 PFlop, суперкомпьютер в Тайваньском суперкомпьютерном центре - 4.701 PFlop и вычислительный комплекс Oak Ridge National Laboratory (США) — 2.331 PFlop. (В одном петафлопсе 1000 терафлопс). В США объявлено о начале разработки эксафлопного компьютера [4], один эксафлопс равен миллиону терафлопс..

Однако во многих случаях эти потрясающие вычислительные мощности оказываются загруженными не полностью и используются неэффективно. Почему? Неужели не хватает задач, которые требовали бы для своего решения их применения?

Задачи, конечно, есть. Приведем несколько примеров [5].

Рассмотрим задачу прогноза погоды в масштабах всей планеты. Для расчета атмосферных явлений будем использовать ячейки размером 1 кубическая миля до высоты в 10 миль. Тогда при шаге по времени в 1 минуту при решении уравнений математической модели с целью получения прогноза погоды на 10 дней нам потребуется 10^{15} операций с плавающей точкой. Это как раз и составит 10 дней работы персонального компьютера производительностью 1.5 Gflop. Для проверки правильности результатов останется лишь выглянуть в окно. Понятно, что решение такой задачи требует использования более мощной техники.

Еще более впечатляющими по требуемому времени выполнения являются задачи астрофизики и биофизики. Для моделирования развития галактики из 10^{11} звезд на один шаг интегрирования уравнений математической модели требуется примерно 1 год времени работы современного персонального компьютера. Таких шагов для получения сколь-нибудь адекватных результатов требуется несколько миллиардов. А для моделирования образования белка методами молекулярной динамики у вас уйдет 10^{25} машинных команд, что займет на одноядерном персональном компьютере с тактовой частотой 3.2 Ghz 10^6 веков. Понятно, что при таких временах расчетов ни один исследователь результатов дождаться не сможет.

Есть задачи, и есть техника для их решения. Что же мешает эффективному использованию высокопроизводительных вычислительных комплексов? Для ответа на этот вопрос проще всего привести цитату из лекции Дейкстры, прочитанной им при получении Тьюринговской премии [5]: „*To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*“ (Примерный перевод: Вот ведь незадача: пока не было вычислительных машин, программирование не являлось проблемой, когда мы сделали немного слабых компьютеров, программирование стало простой проблемой, сейчас мы имеем гигантские компьютеры, и программирование становится поистине гигантской проблемой)

Эта гигантская проблема заключается в том, что для решения имеющихся задач на существующих мощных компьютерах нужно написать соответствующие программы, а это не всегда просто. Мы столкнулись с очередным кризисом программного обеспечения.

Три кризиса программного обеспечения

В процессе развития вычислительных систем программное обеспечение (software) и аппаратное обеспечение (hardware) эволюционировали совместно, оказывая влияние друг на друга. Появление новых технических возможностей приводило к прорыву в области создания удобных, безопасных и эффективных программ, а свежие программные идеи стимулировали поиски новых технических решений [6]. Несоответствие выросших технических способностей ЭВМ решать сложные задачи и существующего программного обеспечения трижды приводило к кризисам software, два из которых были успешно преодолены.

Напомним читателям, что все языки программирования подразделяются на языки низкого и языки высокого уровня. Соответственно, программирование можно разделить на низкоуровненвое и высокоуровневое. Языки низкого уровня называются так, потому что они приближены к системе команд процессора (или других устройств). Примерами таких языков служат различные ассемблеры, автокоды и т.п. Программирование сложных задач на таких языках возможно, но представляет значительные трудности.

Первый кризис программного обеспечения можно датировать концом 50-х – началом 70-х годов прошлого века, когда программирование на языках низкого уровня вошло в противоречие с возможностями компьютеров. Тогда назрела необходимость перехода на более высокий уровень абстракции и переносимости программ. Решением появившейся проблемы стало развитие языков высокого уровня (ALGOL, FORTRAN, C и т. д.) для машин неймановской архитектуры.

Тогда же стали возникать служебные программы. Во первых. Это трансляторы и компиляторы, позволявшие переводить программы с языка высокого уровня (приближенного к языку общения между людьми, содержащему сложные синтаксические конструкции) на язык низкого уровня, а затем – в машинные коды. Тогда же стали появляться системы управления заданиями, первые операционные системы.

Второй кризис software разразился в 80-е – 90-е годы прошлого века. Технический уровень вычислительных систем стал позволять разработку сложных и надежных программных комплексов, содержащих миллионы строк кода и написанных сотнями программистов. Но существовавшее программное обеспечение сдерживало этот процесс. Выходом из сложившейся ситуации стало появление объектно-ориентированных языков программирования и инструментария для поддержки больших программных проектов.

До начала нынешнего века повышение производительности массовых вычислительных систем осуществлялось двумя путями. Первый путь – экстенсивный – это повышение плотности полупроводниковых элементов на одном кристалле и

увеличение тактовой частоты их работы без существенного изменения архитектуры компьютеров. Второй путь - интенсивный - внедрение элементов параллелизма в компьютерных комплексах — от параллельной выборки разрядов из памяти до многих устройств, одновременно выполняющих различные операции [7].

В 1965 году один из основателей фирмы Intel — Гордон Мур — сформулировал эмпирический закон, который обычно принято записывать так: количество полупроводниковых элементов на кристалле и производительность процессоров будут удваиваться в среднем каждые полтора–два года. Однако в реальности закон Мура носил экономический характер и первоначально утверждал, что стоимость производства одного транзистора будет вдвое снижаться каждые полтора года. Закон Мура в экономической форме, по мнению сотрудников Intel, будет действовать вплоть до 2015 года. Начиная с 2005 года, интенсивный путь повышения производительности компьютеров стал превалирующим — появились многоядерные процессоры. Теперь закон Мура в среде людей, близких к информационным наукам, принято формулировать так: удвоение количества ядер на процессоре будет происходить каждые полтора–два года. Начался третий кризис software. Существующая массовая парадигма программирования — создание последовательных программ — пришла в противоречие с массовым появлением нескольких исполнителей в вычислительной системе. Сейчас в продаже почти невозможно найти ноутбук с одним вычислительным ядром на процессоре. Скоро в продаже появятся мобильные телефоны с многоядерными процессорами внутри. Каким окажется путь разрешения этого кризиса, покажет время.

Однако в настоящий момент без изменения парадигмы программирования в математическом моделировании невозможно остаться на острие научных исследований.

Последовательная и параллельная парадигмы программирования

В научной среде до сих пор не существует единого мнения о том, что следует понимать под термином «парадигма программирования» [8]. Под парадигмой программирования мы будем понимать совокупность этапов работы, которую должен проделать специалист в области математического моделирования от постановки задачи до получения результатов ее решения на компьютере.

При решении задачи на последовательной вычислительной системе (один процессор и одно ядро) принято считать [9], что эта совокупность состоит из пяти этапов.

1. Постановка задачи.

На этом этапе проводится предварительная работа, коллективы заказчиков (физиков, химиков, биологов и т.д.) определяются, что они хотят от математиков, программистов.

На этом обязательном этапе никакого творчества не происходит, специалисты в разных предметных областях вырабатывают свой единый язык, на котором им предстоит общаться.

2. Создание математической модели.

Это первая часть так называемой «Триады Самарского». Именно академик Александр Андреевич Самарский [10] впервые сформулировал тезис, что с появлением ЭВМ математика становится экспериментальной наукой. Записывая уравнения (математическая модель, описывающая физический, биологический, социальный феномен), мы можем решать их приближенно с помощью компьютера, менять данные и параметры задачи, смотреть, что будет с системой при изменении тех или иных условий.

3. Разработка алгоритма решения в рамках созданной математической модели.

Это – основная часть работы математика – прикладника. После того, как сформулированы уравнения математической модели, но до того, как начать писать программу, математик – прикладник должен выбрать метод приближенного решения задачи (одной и той же системе уравнений модели могут соответствовать разные методы решения) и разработать алгоритм реализации метода (одному и тому же методу могут соответствовать различные алгоритмы). Здесь необходим специалист, обладающий глубокими знаниями математики и некоторым чутьем – многие вопросы выбора метода и алгоритма для сложных задач неформализованы.

4. Написание программы, реализующей алгоритм, в одной из выбранных моделей программирования и на выбранном алгоритмическом языке. Модель программирования определяет основные идеи и стиль программной реализации, абстрагируясь от алгоритмического языка и, частично, от hardware. Например, модель функционального программирования, модель объектно-ориентированного программирования, модель продукционного программирования и т. д.

5. Работа программы как набора процессов и/или нитей исполнения на вычислительной системе и получение результатов.

После того, как прошли все 5 этапов, наступает этап интерпретации результатов, здесь снова необходимо взаимодействие разных специалистов. Иногда после этого снова наступает этап 1.

При решении задачи на параллельной вычислительной системе (несколько процессоров и/или несколько ядер) в этой совокупности появляются дополнительные этапы. Теперь этапов становится восемь:

1. Постановка задачи.

2. Создание математической модели.

3. Разработка алгоритма.

Эти этапы ничем не отличаются от случая реализации проекта на компьютере с неймановской архитектурой.

4. Декомпозиция алгоритма (*decomposition*). При параллельной реализации алгоритма мы предполагаем, что он будет выполнен несколькими исполнителями. Для этого нужно выделить в алгоритме наборы действий, которые могут быть осуществлены одновременно, независимо друг от друга — декомпозировать алгоритм. Различают два вида декомпозиции — по данным и по вычислениям.

Если в алгоритме сходным образом обрабатываются большие объемы данных, то можно попробовать разделить эти данные на части — зоны ответственности, каждая из которых допускает независимую обработку отдельным исполнителем, и выявить вычисления, связанные с зонами ответственности. Это — декомпозиция по данным.

Другой подход предполагает разделение вычислений на зоны ответственности для их выполнения на разных исполнителях и определение данных, связанных с этими вычислениями. Это — декомпозиция по вычислениям (функциональная декомпозиция).

Декомпозиция возможна не всегда. Существуют алгоритмы, которые принципиально не допускают при своей реализации участия нескольких исполнителей.

5. Назначение работ (*assignment*). После успешного завершения этапа декомпозиции весь алгоритм представляет собой совокупность множеств наборов действий, направленных на решение подзадач отдельными исполнителями. Наборы действий одного множества допускают одновременное и независимое выполнение. Множества могут содержать различное количество наборов и, соответственно, реализовываться на разном количестве исполнителей. Не исключено, что часть множеств будет содержать всего один набор и требовать всего один процессор (ядро).

В реальности количество имеющихся ядер всегда ограничено. На данном этапе необходимо определить, сколько исполнителей вы собираетесь задействовать и как распределить подзадачи по исполнителям. Основными целями назначения подзадач являются балансировка загрузки процессоров (ядер), уменьшение обменов данными между ними и сокращение накладных расходов на выполнение самого назначения. По времени способы назначения разделяются на две категории:

• статические — распределение выполняется на этапе написания, компиляции или старта программы (до реального начала вычислений);

• динамические — распределение осуществляется в процессе исполнения.

6. Аранжировка (*orchestration – слово можно перевести на русский язык как аранжировка музыкального произведения или взаимодействие*). После завершения этапа назначения мы находимся в состоянии композитора, подготовившего все партитуры для исполнения своей симфонии. Уже написана аранжировка – проведена раскладка музыкального произведения по всем инструментам. Аранжировщик определяет, в какой момент слушатели должны слышать, например, скрипки, а в какой момент тромбоны. Но нормальное звучание оркестра возможно лишь при наличии дирижера, который синхронизирует деятельность отдельных музыкантов и вносит в исполнение свой стиль. Роль дирижера тоже определяется на этапе аранжировки.

Его целью является выбор программной модели и определение требуемой синхронизации работы исполнителей, которая во многом будет зависеть от программной модели.

Остановимся подробнее на программных моделях для работы на параллельных вычислительных системах. Хотя классификация и названия программных моделей до конца не устоялись, мы выделим четыре основных:

• Последовательная модель. Предполагает, что нагло наплевав на два предыдущих этапа, вы пишете обычную последовательную программу в одной из последовательных моделей программирования для последующего автоматического ее распараллеливания компилятором или специальными программными средствами. Преимущество модели — ничего лишнего не надо делать по сравнению с последовательным вариантом, недостаток — автоматическое распараллеливание имеет крайне ограниченные возможности.

• Модель передачи сообщений. Предполагает, что работающее приложение состоит из набора процессов с различными адресными пространствами, каждый из которых функционирует на своем исполнителе. Процессы обмениваются данными с помощью передачи сообщений через явные операции send/receive (отослать сообщение/принять сообщение). Преимущество этой модели заключается в том, что программист осуществляет полный контроль над решением задачи, а ее недостаток — в сложности программирования.

• Модель разделяемой памяти. Предполагает, что приложение состоит из набора нитей исполнения (thread’ов), использующих разделяемые переменные и примитивы синхронизации. Выделяются две подмодели: явные нити исполнения — использование системных или библиотечных вызовов для организации работы thread’ов и программирование на языке высокого уровня с использованием соответствующих прагм.

Первая подмодель обладает хорошей переносимостью, дает полный контроль над выполнением, но очень трудоемка. Вторая подмодель легка для программирования, но не дает возможности полностью контролировать решение задачи.

• Модель разделенных данных. Предполагает, что приложение состоит из наборов процессов или thread'ов, каждый из которых работает со своим набором данных, обмена информацией при работе нет. Такая модель применима лишь для ограниченного класса задач.

7. Написание программы, реализующей алгоритм, в выбранной модели программирования и на выбранном алгоритмическом языке.

8. Отображение (*mapping*). При запуске программы на параллельной компьютерной системе необходимо сопоставить виртуальным исполнителям, появившимся на предыдущих этапах парадигмы программирования, реальные физические устройства. В зависимости от выбранной модели программирования это может осуществляться как человеком, проводящим вычислительный эксперимент, так и операционной системой.

В дальнейших лекциях мы остановимся на изучении этапа декомпозиции. Но прежде, чем декомпозировать алгоритм, нужно понять, а стоит ли это делать, достаточно ли выбранный алгоритм эффективен?