Алексей Николаевич Сальников
email alexey.salnikov@gmail.com, salnikov@cs.msu.ru
VK asalnikov
telegram ???
Skype salnikov_alexey

Организационные вопросы

1. Параллельные вычисления.
2. Распределённые вычисления.

Письменный допуск: 10 вопросов/задач, в т. ч. написать простенький код на бумажке.

Параллелизм

Человек задумывает программу как последовательную. Но всёё параллельное выполняется ужасно долго, и уже много лет компы так не делают. Они пытаются самостоятельно параллелить и асинхронизировать всё, что можно.

Зачем параллелить?

1. Большой объём работы. Что такое "большой"? Из сложности вычислений знаем об оценках сложности алгоритмов:
   (a) Временная сложность.
   (b) Пространственная сложность. f(size(input))

Короче, параллелить надо.

Как оценить эффективность?

График "расстояние между вычислениями"—"быстродействие".

Уровни параллелизма:

1. Вентили. FPGA/ПЛИС (программируемые логические интегральные схемы, процессор собирается как конструктор под поставленную задачу, обеспечивая высокое быстродействие, параллелизм и др.) Процессор описывается как абстракция "схема из функциональных элементов с задержками". Впрочем, эти вентили медленнее, чем традиционные процессоры, без возможности программирования. Программируются схемы на VHDL и Verilog. В Таганроге придумали COLAMO — интерпретатор чего-то, похожего на Паскаль, в программу для ПЛИС, стараясь выполнить всё в один такт.
2. Регистры, АЛУ, УУ, шины, прочее. У процессора есть система команд (то, что процессор может делать), они реализованы через микрокод (то есть захардкоженная в процессор последовательность элементарных действий, которые он выполняет при поступлении команды).
3. Ядра в процессоре (многоядерность). Ядра могут лезть в чужие кэши, есть общий кэш. Ядра общаются по шине типа Hypertransport.
4. Процессоры в системе (многопроцессорность). Имеем в виду, что на одной плате.
5. MPP (Multiprocessor) Плат может быть много, но все соединены, условно говоря, в один комп, управляемый одной ОС. Типичный пример — мейнфрейм.
6. Вычислительный кластер. Время передачи данных между машинами уже велико. Нужно с этим считаться. Характеристики: надёжность и время передачи данных.
7. Географически разделённый кластер. Сеть ненадёжна ("обычно данные доходят"). Скорость передачи данных не гарантируется (нужно, чтобы объём вычислений значительно превышал количество данных, которые нужны для них). Компьютеры кластера ненадёжны (можно заставлять несколько компов вычислять одно и то же, и Вы всё равно будете после этого в выигрыше (уменьшив риск того, что вычисление прервётся из-за выхода одного компа из строя)).
8. GRID-система (сеть из кластеров). Сеть может падать, но сами элементы надёжны (кластер же отказоустойчив). Передача данных по сети слабо гарантируется, но вычисления очень объёмны.

Классификация Флина по потокам команд и данных

УУ — поток команд.

АЛУ — поток данных.

S – Single
M – Multiple
I – Instruction
D – Data

SISD – одно АЛУ, одно УУ, архитектура как по фон Нейману.
SIMD – берём одну инструкцию, но применяем её к нескольким регистрам (т. е. к множеству данных). [Векторные процессоры и векторные инструкции]
MISD – в жизни особо не встречается, к нему можно отнести предсказание переходов.
MIMD – почти всегда в жизни встречается именно оно.

Характеристики процессора:

1. Конвейер. TODO: скопировать из википедии, зачем Разгон конвейера: время, через которое быдет выдан результат, после того, как в него попадёт первая инструкция. Далее ответы будут выдаваться каждый шаг. Конвейеру мешают:

   (a) промахи кэша (при переключении потока это будет почти наверное)

   (b) зависимости в данных

   С конвейером умеет обращаться компилятор, стараясь сделать получше. Флаги gcc: -march, -mtune.

2. Суперскалярность.

3. Предсказание переходов. Нужны теневые регистры (не видимые программному коду). Если есть ветвление, то все ветки выполняются одновременно (если могут), результат хранится в теневых регистрах, а когда выясняется результат условия, просто выдаётся нужный уже посчитанный результат. Хорошо работает на RISC-системах, ибо там фиксированный размер команд.

4. Набор инструкций (векторные инструкции: стандарты MMX, SSE, AVX). Обычные регистры начинают считаться "сцепленными векторными регистрами".

5. VLIW. Большая длинная команда, в которой кодируются отдельные маленькие команды, которые позволяют параллелиться на уровне процессора. Плюсы: компилятор может всё очень распараллелить. Минусы: это слишком сложно для компилятора. Часты полупустые VLIW, когда команд недостаточно. Код программы увеличивается в размере. Кэш-промахи чаще, ибо всё большое. Примеры: Itanium, Эльбрус.

6. SMT (Simultaneous Multiple Threading) / Hyperthreading Поддерживаем два потока команд (для разных процессов). Если команды не конфликтуют по обращениям к АЛУ, например, то запускаем обе. Иначе одну. За счёт этого ускорение. Ныне дублируем регистры, прочее. Выигрыш идёт за счёт всяких маленьких ожиданий, типа обращений к кэшу. Ускорение может быть почти как при многоядерности. Безопасность: программа злоумышленника может видеть чужие регистры.

7. Многоядерность.

Литература

1. Архитектура. Документация от производителей. Стандарт Infiniband.

2. Администрирование и анализ программ. Книги от специалистов: Лацис А. О. "Как построить и администрировать суперкомпьютер". С. А. Жульматий, Дацюк. "Администрирование суперкомпьютеров и кластерных систем". Учебники: В. В. Воеводин, Вл. В. Воеводин. "Параллельные вычисления" (см. про старые системы и про векторные машины). Воеводин В. В. "Вычислительная математика и структуры алгоритмов" К. Ю. Богачёв. "Основы параллельного программирования" Гергель В. П. (Декан нижегородского фивта) "Теория и практика параллельных вычислений". Jan Foster "Design and building . . . "

3. Технологии программирования OpenMP Barbara Chapman "Using openMP" MPI William Gropp "Using MPI": части 1, 2, 3.

Конвейер: m стадий, t_i работает каждая стадия, массив данцемула вычисления времени срабатывания конвейера. Конвейер с зацеплением: параллелит какие-то части, например, часть работы отдаёт сумматору, часть - умножителю, они работают вместе, а потом в общих стадиях комбинируются в результат.

Лёгкие ядра. Рядом с обычным ядром и его обвязкой располагается множество мелких ядер(SPU), которые умеют поменьше, сами попроще, но их много. Пример: IBM Cell. Так устроены графические процессоры: общие большие регистры, общая большая память, множество мелких ядер, образующих трёхмерную решётку (3D Mesh). К решётке можно добавить ядер, соединив видеокарточки по SLI. Для фрагментов решётки определённого размера (трёхмерного кубика) своё УУ. Такты считаются для каждого кубика отдельно, и все элементы кубика за такт занимаются одной и той же работой (например, перемножают параллельно одну и ту же матрицу). OpenCL переносим, будет эмулироваться на процессоре, если подходящего графического устройства не нашлось, но умеет мало. NVIDIA CUDA. Говорят, программировать приятнее.

Классификация многопроцессорных компьютеров по доступу к памяти

1. Адресное пространство
2. Скорость обращения к ячейке.

1. SMP (Symmetrical Multiprocessing)/UMA (Uniform Memory Access) Адресное пространство общее. Обращение из любого процессора к любой ячейке памяти за одно и то же время. Это — шинная архитектура. Банки памяти и процессоры висят на одной шине. Дорого делать, если процессоров больше 2. И если процессоров много, будут конфликты за шину, и не будет значительного прироста производительности.
2. NUMA(Non Uniform Memory Access) Обращение к разным ячейкам памяти за разное время. У каждого процессора появляется “своя” память, к которой обращения быстрее всего. Это — процессоры со своей памятью, соединённые своей шиной, провязанные все общей шиной.
3. ccNUMA (Cache Coherent NUMA). Записи в память производятся на самом деле в кэш. Кэш - отдельная многопроцессорная система, которая сама синхронизируется, когда потребуется, и обеспечивает желаемое поведение. Синхронизация кэшей - аппаратный протокол, и его реализовывать - дорого, если процессоров много.
4. MPP (Massive Parallel Processing) Отказыеваемся от общего адресного пространства. Отдельный шкафчик с платами. Называем кластером, если не один шкафчик. Каждая машина — узел кластера. На всём этом должна стоять одна ОС, и работать это должно как единое целое.

Пример. Вычислительный центр.

Контора, в которой стоят компы (т. е. вычислительные системы). Обращение к нему пользователи ведут через Интернет. Есть сайтик, почтовый сервер (рассылает уведомления о проблемах на вычислительных системах). Сайтик — высокоуровневый интерфейс, в котором выдают логины и просят у пользователей годовые отчёты об их деятельности.

Устройство:

1. Интерфейсная машина. На неё заходят, компилируют, редактируют код, делают всякие вещи. Запускать на ней вычисления не следует, на это админы ругаются.
2. Контрольный сервер (один или много). Если всё не так, то зажигается красная лампочка. Или совсем красная лампочка, если всё ещё хуже. Он может сказать, какая именно машина вышла из строя, чтобы можно было её заменить. Возможно, всё это происходит со свистелками.
3. Сервер очередей. Иногда распределение задач занимает отдельный сервер.
4. Узлы кластера, на которых всё считается.
5. Хранилище (storage). Тоже кластер, процессоры слабые, но большие жёсткие диски.

Сети, которые всё это соединяют:

1. Управляющая сеть. Обычная компьютерная сеть, например, Ethernet. Предназначена для управления и подгрузки кода, который будет выполняться.
2. Сеть синхронизации. Для синхронизации часов на различных процессоров (точнее, чем NTP, который с точностью до секунд). Он не просто скачком переводит время, а ускоряет часы машин, которые отстают, и замедляет, когда машина спешит. Реализуется в виде коаксиального кабеля, по которому идут импульсы точного времени.

3. Сеть ввода-вывода. Может быть объединена с вычислительной сетью. Некоторые её узлы связаны с хранилищем, другим узлам данные передаются через другие узлы как через прокси.
4. Сеть взаимодействий типа "точка-точка". Если какие-то два узла кластера хотят общаться между собой быстро и много, их соединяют так.
5. Сеть коллективных операций. Если надо что-то распространять по узлам кластера почти так же, как в точка-точка.

Софт для обеспечения работы кластера

Задачи:

1. Мониторинг живучести узлов и оборудования. ganglia собирает информацию от отдельных узлов в одном месте, nagious анализирует её и информирует, если что-то не так. Работает для небольших кластеров. Если кластер большой, то встаёт задача анализа данных, поиска аномалий по куче логов.
2. Запуск. Если кластер большой, то одновременное включение всех узлов вызовет просадку напряжения в сети. Поэтому все узлы включаются постепенно (сигналы по протоколу SMPI или IPMI). Перед этим включается система охлаждения (это важно). Узел получает IP-адрес обычно по протоколу dhcp. Если это будет происходить очень быстро, будет большая нагрузка на сеть, пакеты с IP-адресами могут потеряться, и узлы не получат ссебе адрес. Поэтому нужно модифицировать DHCP-протокол. Можно сделать сеть управляющих компов звёздчатой структуры (но тогда будет нагрузка на промежуточные компы, это нужно учесть).
3. Где взять образ ОС? Дисковая система: базовый образ рассылается по протоколу tftp почти так же, как в предыдущем пункте. Они поднимают NFS(Network File System). NFS имеет только один сервер, поэтому пользовательские данные там не хранятся. Пользовательские данные хранятся в параллельной файловой системе. Они пытаются распределять данные так, чтобы они были близко к обрабатывающему их узлу, следят за надёжностью и обеспечивают POSIX-интерфейс обращений к ней. Примеры: lustre и gpfs
4. Авторизация пользователей. Как выяснять информацию о пользователях с узла кластера? Можно на каждом узле хранить passwd, shadow, group, gshadow и синхронизировать их rsync. Почти нет нагрузки на сеть (синхронизация же редко). LDAP – иерархическая база данных, в которой можно хранить данные о пользователях. Active Direcctory в Windows — реализация LDAP-протокола. Большая нагрузка на сеть, если частые обращения. Могут даже замедлять работу кластера.
5. Средства разработки. В основном на кластерах висит какая-то реализация MPI. Кросс-компиляция: у кластера есть разные виды машин (интерфейсная, вычислительный узел, ...). Поэтому используется компилятор, который компилирует программу не обязательно для архитектуры машины, на которой он стоит. Он использует чужой набор команд, а линковщик использует чужие библиотеки. Исполняемый файл отправляется на другую машину для исполнения.
6. И т. д.

Совет: дебаг можно проводить на модели кластера, например, из двух таких же машин, как там. Она позволит отловить совсем наивные ошибки. При отправке же на кластер можно очень долго стоять в очереди, чтобы программа потом упала за 4 минуты работы.

Управление очередями (Batch sheduling system) PBS — Portable Batch System В 70-е использовали файл паспорта задачи, простой текстовый файл, который описывал множество ресурсов, которые должна потребить программа, которая будет выполнена. PBS использует shell-скрипт, в котором специального вида (# pbs ...) комментарии описывают ресурсы, которые будут использованы:

1. число процессоров/узлов
2. время (но тут нужно быть аккуратным: убивать задачи, которые превысили время, надо аккуратно, ибо их хозяин может рассердиться)
3. память (оперативная, дисковая обычно не смотрится), лицензии на коммерческое ПО (например, fluid; кластеры для коммерческих нужд (не научных/образовательных) обычно используют много проприетарного ПО; система управления лицензиями flexlm)
4. Оборудование (например, узлы с графическими ускорителями)
5. Имя узла (можно попросить исполняться на конкретном узле; исполняться задача обычно начинает реже)

Популярные планировщики: openpbs, Moab, slurm (slurm-lnl в Debian), LoadLeveler, Sun Grid Engine (с продвинутой системой распознавания пользователя с сертификатами).

slurm

Интерфейсная машина, демон-контролёр (на своей машине), демон-пускач. Общение пользователя происходит через клиентское приложение с контролёром. Он знает текущее состояние кластера и очередь задач. Он решает, когда что выполнять. Если саму очередь поддерживать сложно, выделяется отдельный демон ведения очереди (планировщик заданий). Когда контролёр решает поставить задачу в очередь, он говорит демонам-пускачам, которые, работая как init, создают нужное дерево процессов. Когда выполнение закончится, он же и завершает процессы. Он же следит за здоровьем кластера с помощью вышеупомянутых ganglia и nagious.

В очередь можно поставить только shell-скрипт. Делается это с помощью sbatch. Параметры к нему могут быть такими же, как паспорт задачи. Можно указать файлы ввода-вывода. Интерактивности не предполагается.

1. srun делает то же самое, только блокирует консоль до завершения.

set -a печатает все переменные окружения из скрипта запуска. В том числе job id. Можно будет смотреть информацию (например, о доступных ресурсах, о занятости кластера, ...) командой sinfo, Смотреть очередь можно командой squeue. scancel убивает задачу. scontrol позволяет интерактивно просматривать и настраивать задачу через общение с сервером-контролёром. Например, можно запретить принимать новые задания в течении 3-х дней. Используется сисадминами. smap визуально рисует загруженность узлов.

slurmctld — главный демон. ??? — демон-пускач. slurmbdb управляет отправкой событий в базу данных при операциях с задачами. Восстанавливает после, например, перебоев с электроснабжением.

Алгоритмы планирования

1. FIFO. Единственная честная. Низкая загрузка кластера зачастую. Несправедливость к маленьким задачам (они будут ждать так же, как большие).
2. FIFO с приоритетами.
3. Gang-алгоритм. Делим задачи в завичимости от потребляемых ресурсов в несколько FIFO с приоритетами.
4. Backfill. Вычислительная система — стакан, дно — многомерная поверхность, на которой указано множество ресурсов. Вертикально растёт время. Задача — сыграть в тетрис. Окно - промежуток времени и множества ресурсов, которое может быть захвачено процессом. Приходящий процесс пытается занять ближайшее по началу выполнения окно, в которое он помещается. В случае конфликтов в ранние окна ставятся задачи, у которых приоритет выше.