Параллельные вычисления

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

Параллелизм

Человек задумывает программу как последовательную. Но всё последовательное 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))

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) Отказываемся 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.

Система приоритетов задач пользователя

Как сделать так, чтобы никого не обделить? Топорков В. В. — человек, занимающиеся алгоритмами планирования задач, в первую очередь планированием задач для GRID. Проблемы с честностью:

1. Могут быть срочные задачи относительно фона.
2. "Переключение" на более срочную задачу. Сложность в хранении состояния оперативной памяти, промежуточных вычислений. Это очень много. Поэтому задачу просто убивают, но в очереди задача остаётся на первом месте, чтобы начать выполнение сначала. Поддержка контрольных точек: в какие-то моменты, определяемые событием в коде или просто астрономическим временем, программа по своей инициативе сбрасывает своё промежуточное состояние в файловую систему. Есть библиотеки, автоматически расставляющие контрольные точки, но они не очень хороши: программист лучше знает, когда вставить точку.
3. Замечание: если задача хотела себе много ресурсов, а пока она стояла в очереди, ресурсы пропали (кластер отвалился), то она не начинает выполняться, когда подходит её очередь, а остаётся ждать в начале очереди, пока ресурсы вернут. Считается, что задача когда-то начнёт выполняться. Так что это не проблема с честностью.
4. Приоритеты у пользователей (id), групп (gid), проект (account в терминах slurm)
5. Partition – кусок кластера. Можно отдать его, например, группе пользователей, и если они будут ставить задачи в этот раздел, то они получают наибольший приоритет (остальных пользователей оттуда выгоняют). Популярно, если нужно что-то быстро посчитать, но не отдавая весь кластер. Минусы: мощности меньше, чем у всего кластера.
6. Некорректное использование вычислительной системы. Пример: на кластере 21к процессоров, одному пользователю надо 20к процессоров на 10 минут, другому – 1.5к на 2 недели. Первый может оказаться в пролёте. Надо стимулировать пользователей не ставить такие задачи, как второй. На каждого пользователя заводится

счётчик суммарных процессор-часов. И есть разные отметки количества (т. е. пользователей "раскидывают по корзинам"). Пользователииз корзин с меньшим временем имеют больший приоритет, чем все из больших корзин. Счётчик сбрасывается когда админ захочет, например, раз в месяц. Такой подход используется в РАН. Второй подход: на каждого человека отводится какое-то количество процессор-часов. Применяется в коммерческих вычислительных центрах.

"Рыночные" алгоритмы планирования. Каждому пользователю выдаются фишки приоритета, которые он раскладывает на свои задачи. Устраивается аукцион среди всех задач с такими фишками.

Интерконнект в кластере

Отличия от сети: Предсказуемость (считается, что узлы чаще всего доступны). Топология не меняется во время жизни кластера (не факт). Ещё что-то.

Характеристики:

(a) Латентность (накладные расходы на передачу данных, сек). Неотделмио от самих данных, поэтому меряется в процессе передачи данных. Это задержка между началом передачи и окончанием приёма. Латентность растёт вместе с объёмом данных (накладные расходы тоже растут, и поэтому отделить объём данных от латентности не получится).

(b) Скорость передачи данных (пропускная способность, байт/сек). Пример: грузовик с винчестерами обеспечивает охрененную пропускную способность, но очень высокую латентность.

(c) Темп выдачи сообщений. Важна для посылки коротких сообщений. Время, которое затрачивается перед повторной отправкой сообщения ("насколько часто Вы можете флудить в сеть").

(d) Устойчивость. "Как данные доставляются в среднем". Можно сказать, что это дисперсия времени передачи одинаковых сообщений между двумя фиксированными узлами в фиксированном направлении. Если интерконнект, дисперсия мала по сравнению со временем передачи. Если через Интернет, до дисперсия может многократно превышать время передачи. При отправке больших порций данных влияние этого отклонения нивелируется, но для малых (и частых) сообщений она очень значима.

Способы соединения узлов в кластере

Способ соединения узлов в кластере называется топологией (т. е. устройство графа соединений). Топология выбирается при проектировании СК исходя из его задач. Вершины графа – узлы, рёбра – линки. "Вычисляющая сущность" является узлом (кружок). Узлом может являться "свич" – устройство, занимающееся только передачей через себя данных (квадратик). Эти две роли можно и совмещать (выпуклый многоугольник).

Классические топологии:

1. Звезда. Один свитч и несколько компов. Так обычно устраивают компьютерные классы. Все компы висят на одном проводе. В реальности провод закольцован, и он оптоволокно.
2. Линия. У каждого узла 2 сетевых вывода. Соединяется по цепочке. Применяется редко, для небольших вычислительных кластеров и в основном для конвеерной обработки данных: когда можно подсчитать кусочек, передать на следующую машину, а самому заняться задачей, аналогичной исходной. Применяется военными, ибо у них часто есть много стадий предобработки данных.
3. Кольцо. Зацикливаем линию. Могут быть проблемы со скоростью передачи данных (по витой паре, по оптоволокну не очень много) между первым и последним элементом кольца, если они далеко друг от друга физически. Часто применяется для соединения территориально удалённых вещей оптоволокном (довольно дёшево, бовольно быстро).
4. Решётка. Популярна, ибо соответствует структуре задач, появляющихся естественным образом во всяких урматах и вычислительной физике (если часто идут обращения лишь к соседям). Например, рассчитать прочность моста, промоделировать климат, рассчитать обтекание чего-нибудь газом или жидкостью. Например, максимальный транзит $2n - 2$.
5. Тор. Решётка, свёрнутая бубликом.
6. Гиперкуб. Ранга 1 – отрезок, ранга 2 – квадрат, ранга 3 – куб, ранга 4 – тессеракт. Напоминает решётку, но разница в том, что тут действительно лишь один кубик. Максимальный транзит – ранг гиперкуба, рост

количества вершин и рёбер экспоненциален. Используется, когда очень важно минимизировать транзит, за счёт стоимости.

7. Деревья. Для сбора информации в один приёмник, например.
8. Смешанные. Например, если у решётки выясняется, что время передачи до удалённых вершин велико, можно пробросить провода каждые 2 узла. Например, 6D-тор. Это тор с дополнительными линиями (который оказывается шестимерным тором). Это не означает, что у задачи шестимерная топология, это просто пробросили доп. рёбра, чтобы уменьшить транзит. Можно, например, сделать решётку из гиперкубов (быстрая связь внутри групп, медленная между группами). На этом построена топология Dragonfly (там трёхуровневая вложенность, но нечто попроще, чем гиперкубы). Топология Butterfly. Если линков много, то это называется жирным деревом.

Характеристики, влияющие на латентность:

1. Максимальный транзит (диаметр графа) – расстояние между двумя наиболее удалёнными вершинами. Сумма переходов (транзитов) как функция от числа узлов.
2. Средний транзит.
3. Надёжность (сколько рёбер нужно удалить, чтобы граф стал несвязным).
4. Стоимость (отношение числа рёбер к числу вершин, прочее).
5. Степень вершины.
6. Величина бисекции (минимальное количество рёбер, которое нужно удалить, чтобы граф разбился на две части с одинаковым числом вершин). Больше бисекция – выше стоимость топологии, но лучше для передачи данных. Это не то же самое, что надёжность. Надёжность – о потере лишь одного узла. Бисекция – о разбиении графа пополам, с надёжностью это связано мало, скорее это о передаче данных.
7. Масштабируемость. Сколько рёбер нужно добавить, чтобы добавить ещё хотя бы один узел с сохранением топологии сети. Например, чтобы добавить узел решётки, нужно добавить целый столбец, и рёбра ко всем ним. Хуже всего с гиперкубом, ибо нужно построить ещё один гиперкуб и провести много рёбер.

Пример. Телефонные сети.

Как были устроены телефонные сети в прошлом (когда уже было много телефонных станций, а не только одна на город). Вы звоните на телефонную станцию барышне. Просите соединить с кем-то. Если он на другой телефонной станции, то ждём освобождения канала между станциями. После барышня вручную вставляет Ваш кабель в разъём канала, с другой стороны делают то же самое, но для принимающей стороны. Это называется "свитч с установлением соединения". Проблема: на время установления соединения (барышень ведь конечное число) и передачи данных (каналов ведь не бесконечное число) никто больше пользоваться не может. Решение: делим всё на пакеты, и передаём по кускам. Проблема: частые переключения заставляют свитч ощутимо греться. Свитч теперь обладает оперативной памятью, в которой хранит пакеты перед отправкой. Постоянно подключены несколько кабелей, и со всеми можно общаться одновременно. Современные свитчи есть комбинация: там есть и аппаратная часть, отвечающая за переключение линков, и программная, отвечающая за пакеты.

Организация маршрутизации

Статическая. Динамическая. В каждый конкретный момент времени существует какой-то маршрут. Как именно пойдёт сообщение определяется самой сетью. Например, она может учитывать загрузку отдельных узлов, географическое расположение узлов, прочее. Гарантируется время доставки в среднем. Устойчивость будет плохой. Обычно маршруты составляются заранее, а при необходимости передачи данных берётся первый маршрут, если где-то по длине маршрута свитч занят, берётся следующий маршрут. Если все заняты, то ждём. Конкретная организация интерконнекта обычно запатентована создавшей её фирмой.

Infiniband

Для канала связи вычисляется его пропускная способность и сохраняется в виде какого-то числа фишек. Фишки свои для каждого линка из каждого узла. У каждого устройства есть GID (Gloabl Identificator) и

LID(Local Identificator). Сетью управляет отдельно выделенное устройство Subnet Manager. Когда появляется новое устройство, Subnet Manager выдаёт ему LID, пересчитывает все таблицы пропускной способности (меряет пропускные способности проводов, то есть), которые нужно пересчитать, пересчитывает таблицы маршрутизации и рассылает их тем, кому это нужно знать. Сеть Infiniband – сеть Петри (сущность из дискретной математики). На каждом переходе пороговое значение. Сообщение длины n стоит какое-то количество фишек. Переход срабатывает, если фишек скопилось достаточное количество. При этом от него отнимается стоимость. Если все фишки израсходованы, передачи не происходит.

Если выясняется, что узел долго не отвечал на запросы, он помечается как умерший и удаляется из таблиц маршрутизации тех, кто с ним общался (Subnet Manager перестраивает их). Читать спецификацию Infiniband, презентации от Mellanox.

RDMA — Remote Direct Memory Access.

Представим обычный отдельный узел, с процессором, памятью, периферийными устройствами, и HCA-адаптером (Host Channel Adapter). Этот адаптер умеет писать напрямую в память (минуя процессор). В современных компах он это делает по шине PCI Express. Вспомним технологию DMA (Direct Memory Access): адаптер пишет куда-то и инициирует прерывание, сообщая процессору, что ему что-то пришло по сети. Infiniband же не сообщает процессору.

То есть, узел может писать в память другого узла, не информируя их об этом, и не влияя на скорость выполнения программ на этом узле (мы же минуем процессор).

Зачем так делать?

1. Можно складывать пакеты в чужую память с целью потом как-нибудь их использовать.
2. Можно класть данные в узел и не ждать пподтверждения. Такое явление называется односторонними коммуникациями (One-Sided Communications), которые поддерживаются, например, MPI версии $\geqslant 2.2$. Функции put, get (понятно, что делают), fence ("барьер" для синхронизации, позволяет удостовериться, что узел закончил делать всё то, что у него было) и ...

Как это выглядит со стороны ОС?

Вспомним функцию IPC chmem.

1. Завести IPC-объект в ядре.
2. Присоединиться к IPC-объекту.
3. Отображение IPC-объекта на виртуальные адреса процессора.

У нас же драйвер Infiniband резервирует кусок оперативной памяти, с которой потом оперирует, а общение с ним происходит через какое-то API (не обязательно API сокетов). Далее он заводит в виртуальной памяти окно и отображает эти виртуальные адреса на тот кусок оперативной памяти через стандартный механизм ОС (таблицы страниц и прочее). Это можно делать ручками, общаясь просто с драйвером Infiniband, но нормальные люди используют, например, функции MPI. Есть, например, функция `MPI_Win_create`. Если в узел пишут сразу несколько других узлов, возможны Race Conditions, и поэтому специальный узел — синхронизатор — борется с этим и сообщает получателю, что ему всё дошло, и данные можно обрабатывать.

OpenSHMEM — другая штука, которая делает похожие вещи.

Оценка параллелизма программы

Пусть есть задание с размером входа n.

$T(1, \text{вход})$ — время работы наилучшей последовательной реализации на данном входе.

$T(p, \text{вход})$ — время работы параллельной реализации на p процессорах на данном входе с выбранным способом распараллеливания.

1. Ускорение $S(p, \text{вход}) = \frac{T(1,\text{вход})}{T(p,\text{вход})}$. Во сколько раз быстрее мы решим задачу.
2. Эффективность $E(p, \text{вход}) = \frac{S(1,\text{вход})}{p}$. Насколько полно загружено оборудование (сколько времени процессоры именно решают задачу, а не синхронизируются и прочими вещами занимаются). В теории всегда $\geqslant 1$. Надо стремиться сделать её как можно ближе к 1.
3. Масштабируемость. Измеряется для ускорения. $K = \max\limits_{p}[S(p, \text{вход}) \geqslant K]$. Каково максимальное количество процессоров, на которых ускорение будет $\geqslant K$? Суть в том, что при увеличении ресурсов до какого-то предела

будет ускорение, а потом может даже быть замедление (возрастают накладные расходы: процессоры могут простаивать из-за отсутствия данных или заниматься в основном пересылкой данных, а не вычислениями) (см. график). Или наоборот, программа, рассчитанная на работу на тысячах процессоров, может дико тормозить на трёх из-за того, что в ней слишком много механизмов параллеливания. Может быть участок со сверхлинейным ускорением (оно происходит не всегда; например, оно может случиться потому, что из-за того, что мы обрабатываем не все данные, а кусок, что-то стало помещаться в кэши).

4. Слабая масштабируемость. Имеет место, если для каких-то $(p, \text{вход})$ выполняется $\frac{\max\limits_p[S(p,\text{вход})\geqslant K]}{\max\limits_p[S(p\cdot\alpha,\text{вход}\cdot\alpha)\geqslant K]} = const$.

   Если она уже перестала выполняться, дальше увеличивать количество процессоров смысла нет.

Зачем это нужно? Полезно оценить, сколько ресурсов надо программе, чтобы она выполнилась за желаемое время.

Классическая задача: уравнение теплопроводности на стержне Пусть $u(l, x)$ — температура, $u(0, x) = t(x)$ — начальное распределение тепла, $\frac{du}{dt} = \frac{d^2u}{dx^2} + f(x, t)$ — уравнение, по которому температура меняется. Хотим температуру в любой момент времени в любом месте.

Трёхточечный шаблон: температура в точке зависит от своей температуры в прошлый момент времени и от температуры соседей.

Стержень разбивается на кусочки, и надо на каждом шаге обмениваться данными о соседях.

Один кусок — 10 времени на операции над числами с плавающей точкой, передача — 5 времени.

$T(1, L) = 10L$

$T(p, L) = \frac{L+p-1}{p} \cdot 10 + 5 \cdot 2$

`MPI_Isend` позволяет делать пересылку асинхронно (здесь посчитали, что он работает для двух посылок).

$S(p, L) = \frac{10L}{fracL+p-1p\cdot 10+10} = \frac{Lp}{L+2p-1} \geqslant 2$

$p \geqslant 2 + \frac{6}{L-4}$

Гипотетическое ускорение $\lim\limits_{p\to+\infty} S(p, \text{вход})$

Закон Амдала

Пусть α — доля последовательных вычислений (то есть те действия, которые должны быть выполнены всеми процессорами без исключения; в противоположность параллельным, при которых процессоры делят между собой операции), β — доля параллельных вычислений, $1 = \alpha + \beta$. Тогда $S(p) = \frac{1}{\alpha+\frac{\beta}{p}+delay(\beta,p)}$, где $delay$ — задержка сети (в реальности она зависит от конкретных двух узлов, которые общаются, и от текущего состояния кластера (не перегружены ли свитчи), но приближённо можно оценить и так).

Программа состоит из множества ячеек памяти и операторов, которые с этими ячейками работают. Переменные бывают входные и выходные. Для того, чтобы программа параллелилась, должны быть операторы, которые можно переставлять местами. Пусть есть оператор S_i и перед ним S_j. S_i зависит от S_j, если их множества переменных пересекаются.

Виды зависимостей:

1. in $\to$ in. Можно параллелить как угодно.
2. in $\to$ out. Антизависимость. Можно переименовать регистры так, чтобы конфликта не было.
3. out $\to$ in. Истинная зависимость.
4. out $\to$ out. Возможно, можно просто выкинуть первое присваивание. Но если есть потенциальная возможность изменения или считывания той же ячейки (где-нибудь посередине, например, из функции, тело которой мы не знаем), то так сделать нельзя.

   Как всё параллелится? Рисуется граф зависимостей. Оставляется самый длинный путь между любыми двумя вершинами. Граф рассматривается как сеть, программа выполняется от истока к стоку, время работы есть критический (самый длинный) путь. Когда есть смысл параллелить программу? Если количество процессоров больше, чем максимальная ширина яруса графа, улучшения не будет. Так делают компиляторы для маленьких фрагментов кода.