Параллельные и распределенные вычисления

Akhtyamov Pavel

4 января 2016 г.

Аннотация

Данный документ представляет электронную версию конспектов лекций “Параллельные и распределенные вычисления”, которые были прочитаны на третьем курсе ФИВТ МФТИ в 2015 Сальниковым Алексеем Николаевичем и Прокопцевым Дмитрием. Естественно, конспект не претендует на полноту. О дополнениях, неточностях можно сообщать меня по почте akhtyamovpavel@gmail.com или в ВК `vk.com/akhtyamovpavel`. Сканированные конспекты первой половины курса “Параллельные вычисления” доступны на yadi.sk/d/SDN10wK1mLAYT. Автор надеется, что конспект лекций будет дополняться.

Часть I
Параллельные вычисления

Список литературы

[1] Воеводин В.В., Воеводин В.В. Параллельные вычисления. СПб., БХВ-Петербург, 2002, 608 с.

[2] Воеводин В.В. Вычислительная математика и структура алгоритмов. М.: Изд-во МГУ, 2006, 112 с.

[3] Гергель В.П., Фурсов В.А. Лекции по параллельным вычислениям. Самара: Издательство СГАУ, 2009, 164 с.

[4] Гергель В.П. Теория и практика параллельных вычислений. М.: ИНТУИТ

[5] Антонов А.С. Параллельное программирование с использованием технологии OpenMP. М.: Изд-во МГУ, 2009, 77 с.

[6] Антонов А.С. Параллалльное программирование с использованием технологии MPI. М.: Изд-во МГУ, 2004, 72 с.

[7] Ian Forster. Designing and Building Parallel Programs. The University of Chicago, 1995, 430 p.

[8] William Gropp. Ewing L. Lusk, Anthony Skjellum. Using MPI – Portable Parallel Programming with the Message–Passing Interface. MIT press, 2014, 336 pp. (последняя версия, найденная мной в Интернете)

[9] William Gropp, Ewing L. Lusk, Rajeev Thakur. Using MPI-2: Advanced Features of the Message Passing Interface. MIT Press, 1999, 406 pp.

1 Характеристики параллельных программ

В данном разделе рассмотрим, каким образом может быть оценена скорость/производительность той или иной программы.

Обозначим через $T(n)$ время выполнения параллельной программы на n процессорах. Через $T(1)^*$ обозначим время выполнения последовательного алгоритма (без инструментов распараллеливания).

Выделим четыре основных параметра для оценки программы:

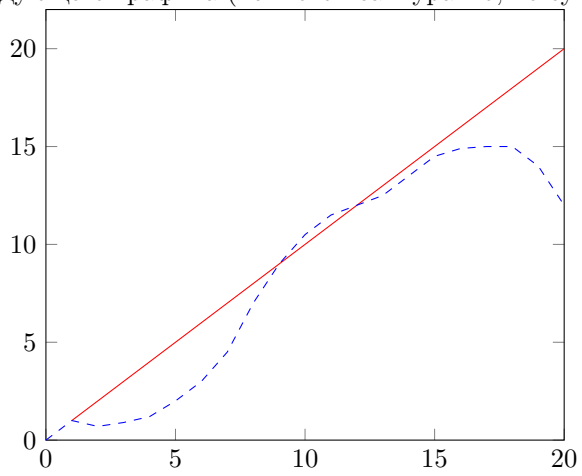
1. Ускорение — величина, равная $S(n) = \frac{T(1)}{T(n)}$, причем желательно измерять следующую характеристику: $S(n)^* = \frac{T(1)^*}{T(n)}$ для правильной оценки работы алгоритма. Несложно заметить, что $S(n)^* = S(n) \cdot \frac{T(1)^*}{T(1)}$, где второй множитель характеризует замедление программы, связанные с накладными расходами. Далее будем подразумевать под $S(n)$ величину $S(n)^*$.

2. Масштабируемость программы — величина $\sup\limits_{n\in\mathbb{N}}\{S(n) \geq 2\}$, которая показывает способность получать ускорение с ростом числа процессоров (но не размера задачи).

3. Слабая масштабируемость программы — величина, аналогичная масштабируемости программы, в случае роста задачи пропорциональну числу задействованных процессов (иными словами, способность сохранять ускорение при неизменном объёме задачи на **каждый узел процессора**).

4. Эффективность — величина, равная $E(p) = \frac{S(p)}{p}$. Данная характеристика показывает полноту использования ресурсов на каждой машине (замедляющим фактором, в данном случае, могут быть, к примеру, накладные расходы на передачу данных). Несложно заметить, что по теоретическим рассчетам величина $S(p)$ не превосходит 1. Однако возможен случай сверхлинейного ускорения, при котором $S(p) > 1$. К примеру, такая ситуация может зависить от устройства железа (устройства кеша — больше данных теперь может быть разложено по кешам на параллельно работающих системах и т.п.). Но при этом может сложиться парадоксальная ситуация: КПД паровоза при больших нагрузках может стать больше, чем КПД суперкомпьютера/кластера (ибо возможны многочисленные промахи по кешам (cash miss), проблемы в передаче сообщений в системе *interconnect* и т.п.). Как видно из перечисленного выше, главной задачей для дизайнеров архитектуры является настройка железа для максимального использования ресурсов.

Далее опишем, как может меняться производительность в зависимости от числа процессоров при помощи следующего графика (немного неаккуратно, но суть отображает):



Из него видно, что сначала идут накладные расходы на запуск программы и пересылку начальных данных, далее идёт сдвиг в сторону идеальной прямой и даже сверхлинейное ускорение. Впоследствии система получает небольшое ускорение, достигает предела вычислительной системы (или входных данных, зависит от задачи), а далее получаем значительное падение засчёт накладных расходов.

Далее рассмотрим на примере закона Амдала более подробную оценку эффективности системы.

Закон Амдала строится для ускорения $S(p)$. Есть части кода последовательные и параллельные. Положим $1 = \alpha+\beta$, где α — доля последовательной части кода, а β — параллельной. Как тогда можно посчитать ускорение: ранее была обозначена следующая формула для ускорения: $S(n) = \frac{T(1)}{T(n)} = \frac{1}{\alpha+\frac{\beta}{n}} = \frac{1}{\alpha+\frac{1-\alpha}{n}}$ (1). Тогда какое максимальное ускорение можно получить в принципе? Действительно, можно определить $\lim\limits_{n\to\infty}\frac{1}{\alpha+\frac{1-\alpha}{n}} = \frac{1}{\alpha}$. К примеру, если $\alpha = \frac{1}{2}$, то программу нельзя ускорить более, чем в два раза.

Вывод. *Для того, чтобы получить ускорение, необходимо уменьшить временную долю выполнения последовательной части.*

На самом деле, описанная выше модель является идеальной. На самом деле, в реальных системах существуют накладные расходы. Поэтому с учетом накладных расходов, формула (1) переписывается в следующем виде: $S(n) = \frac{1}{\alpha+\rho(n)+\frac{\beta}{n}}$, причем параметр зависит от числа процессоров, входных данных.

Накладные расходы уходят на:

1. Порождение “объектов” (не зависит от объёма входных данных).

2. Синхронизация (не зависит от объёма входных данных, явно не очевидна, ибо зачастую выполняется засчёт пересылки данных) — плохо при MPI_Barrier на 10000 процессах.

3. Барьерные эффекты (проблема вместимости в кеш, сверхлинейное ускорение, к примерам) — зависят от аппаратуры (есть в последовательном эффекте, но они явно не выражаются, ибо выполняется код только на одной машине).

4. Накладные расходы на передачу данных (наиболее сильно проявляется при передачи данных по сети — Interconnect, менее — при передачи одного процессора к другому). К примеру, есть $t_{start} + dataSize \cdot (maxflow)$, где $maxflow$ — максимальная пропускная способность.

5. Накладные расходы на ввод/вывод.

Пример. Предположим, что у нас есть некий массив длины n, имеем процессоров в количестве P штук. Мы хотим посчитать сумму чисел на массиве. Положим за единицу времени — время вычисления суммы, а за двойку — время передачи по сети четырех байт информации. Посчитаем время, затраченное на подсчет суммы.

Далее автор пошел считать :-)

Вывод. *При передаче данных нет выигрыша вообще, если нет — то ускорения на 3 процессорах будет достаточно, и, вообще, оптимальное ускорение в идеальной модели при $n \to \infty$ получается при количестве процессоров, равных $p \sim \sqrt{n}$.*

2 Зависимости по данным программного кода

Далее рассмотрим вопрос о зависимостях и связях между ними. Допустим, что мы имеем некоторый код. Программный код представляет собой совокупность переменных и операторов, способных изменять значение переменных. Обозначим через s_i — i-ый оператор языка.

```
int a,b;
a =
1;
b = 10;
```

Рассмотрим ход выполнения программы (а не последовательности написанных операторов: то, что написано позже, может выполняться раньше, к примеру, генерация конструкторов). А можем ли мы поменять процесс выполнения операций для получения ускорения работы программы? Обозначим In — множество переменных, которые необходимы для данного оператора, а Out — множество переменных, которое меняется после выполнения операции (обычно, некоторого набора операции, но надо уточнять). Говорят, что операторы находятся в зависимости, если они имеют непересекающийся набор необходимых операторов.

Заметим, что зависимоть $In \to In$ не накладывает никаких ограничений — значение переменной прочитали несколько раз, при этом инструкции такого рода можно спокойно изменять. Зависимость $Out \to In$ является истинной: данные операторы нельзя поменять местами (важно значение переменной, которое мы записали ранее). Следующая зависимость $In \to Out$ является очень странной: 1) обычно можно сразу записать значение переменной, 2) читать неинициализированную переменную – не очень доброе дело; 3) перезапись значения после использования переменной – сомнительный шаг). Случай 3) иллюстрирует следующий пример:

```
b = 1;
a = b + 1;
c = b + 2;
b = ∅;
```

Говорят, что переменная b находится в антизависимости. И при хорошем надлежании, от нее можно избавиться, но надо следить глобально за кодом (замечание с лекции), ибо в данном примере после кода нам неважно значение переменной b.

Зависимость $Out \to Out$ плоха в том случае, если мы не знаем, что происходило между этими операциями.

Всеми зависимостями занимается компилятором. Можно построить граф зависимостей по данным для операторов, на вершинах записать время срабатывания оператора (τ_i). Тогда время выполнения графа — длина наибольшей цепочки (критический путь). Хорошо использовать количество процессоров, равное ширине максимального уровня. Если есть транзитный путь и прямой путь в таком графе, то лучше удалить прямой путь. В суперскалярной архитектуре можно решить, куда направить вычисление для вентилей.

Естественно, можно задаться следующим вопросом: можно ли устроить процесс автоматического распараллеливания программы с учетом всех зависимостей. На самом деле, ответ на поставленный вопрос отрицателен. Действительно, при построении графа зависимостей могут возникнуть следующие проблемы:

1. Наличие указателей — мы не можем сказать, что происходит с памятью под этими указателями. Но компилятор может найти набор заклинаний и распознать их. Иногда можно распознать параллелизм по наличию некоторых шаблонов для получения информации.

2. Наличие вызовов разных функций — о внутреннем строении функции сложно что-либо сказать (особенно если в нем есть рекурсивный вызов).

Таким образом, в общих случаях пользуются автоматическими оптимизациями компилятора (если роль ускорения не является критической). В ситуациях, для которых производительность играет важнейшую роль, построение графа зависимостей является задачей программиста или проектировщика приложения.

3 Суперкомпуктеры

Поймем, как измеряется производительность суперкомпьютеров. FLOPS (Floating Operations Per Second) - количество операций с плавающей точкой, которое можно делать в секунду.

Пример 1. Персональный компьютер - 6 ядер, 16 Gb оперативки, 6Tb HDD, 60 GFLOPS, Windows, Linux, Mac OS - что можно запихать в один узел.

Текущая производительность - между пента и экзофлопсом. Компьютер экзофлопсной производительности будет построен на других основах, считают специалисты (надо будет, к примеру, переделать OS).

Пример 2. БЭСМ-6 (1968) - Большая Электронно-Счётная Машина. Производительность - 1 MIPS (10^6 операций - инструкций в секунду, плавающей точки не было — эмуляция на целочисленной арифметике), 1 процессоров, OS - КРАБ, RAM 128 Kb, объем - 512 Kb на барабане, 3 Mb на ленте.

Пример 3. "Ломоносов", 2010 год. Производительность — 397 TFLOPS, число процессоров — 10260 (ядер 41040), OS — Clustrx (Linux), RAM 73920Gb, места — 1382400Gb (80 место в мире) — занимает приблизительно лекционную аудиторию (есть Ломоносов версии 2 — занимает первое место в стране, 31 место в списке).

Пример 4. K, Япония, 2011 год (5 место). Производительность — 10510 TFLOPS, число процессоров — 88128 (ядер 705024), OS - mod Linux, RAM 1410048Gb — занимает место ангара. Построен достаточно большой центр — большое здание охлаждает компьютер (жидкостное охлаждение, крутые процессоры — обогревание районов, энергии жрет как один микрорайон — 12 МВатт) — на первом месте в Green 500.

Крутость определяется тестом по перемножению матриц, и количество операций в секунду — характеристика машины. Есть два списка - Top-500 и Green-500 – производительность делится на потребленную мощность. Сегодня, считают, что более важные операции — действия на графах. Поэтому появился список Top-Graph-500 (на первом месте - все тот же японский компьютер). Но в них есть проблема — много хаотических передач данных, т.е. он лучше работает по передаче. Тест построен так: обход графа в ширину, при этом мера производительности — количество тысяч ребер, обрабатываемых за одну секунду. Итак, данный тест проверяет случаи, в которых не так много вычислений, как передачи данных.

Тенденции в мире по поводу OS: в целом, все работают под Linux. По организациям — по количеству используется в Research области (прикладной, проверка правильности технологических решений), Science (Academic) занимается устройством мира в целом, по количеству — индустрия (проектирование всякой всячины). В нашей стране — по науке и образованию в 2011 году (сейчас уже доля промышленности выросла по этой мере). По графику видно, что в один момент ударились в образовании.

Есть специальные тесты для проверки коммуникаций, по которым можно проверять скорость (чем темнее, тем медленнее). В "Ломоносове" шум неравномерно распределен по системе (причем общая память намного шустрее работает, чем передача между узлами).

Как можно измерять производительность суперкомпьютера:

1. Просто посчитать (IPS) — более-менее для целочисленной арифметики.

   (a) Пиковая производительность — максимальная теоретическая производительность (не достигается).

(b) Тесты от производителя — оптимизация железа (если будет использоваться определенный тест, то разработчики подгоняют железо под параметры теста). Top-500 (HPL написаны на C (Джек Донгара) — используется матрица и методом Гаусса решается система линейных уравнений, есть проблемы с выбором строки или столбца; разработчик может подогнать размер матрицы – подходило для кеша, к примеру) подвергается большой критике.

(c) Реальная производительность на задачах

2. Система тестов NASA Parallel Banchmark, Fortran (люди решили написать более интеллектуальные тесты), использующая разные модели распараллеливания, используются практические задачи (к примеру, запуск и полет космических кораблей). Есть 13 областей для распараллеливания.

   (a) MG (MultiGrid) — решается система дифференциальных уравнений (теплопроводности, к примеру), и есть много различных сеток — проблема совмещения;

   (b) CG (Conjugate Gradients);

   (c) FT - метод быстрого преобразования Фурье (FFTW) — возникает хаотический паттерн (Butterfly) для обращения в память;

   (d) IS (Integer Sort) — берется набор чисел, разбитый по определенным блокам, в меру интенсивный обмен;

   (e) EP — генерация независимых случайных величин, конкретная оценка работы узлов в кластере по отдельности;

   (f) SP;

   (g) LU — аналог HPL;

   (h) UA — уравнение теплопроводности в кубе при мигрирующем источнике тепла (там, где тепла много, надо сгущать сетку — проблемы при обращении в память);

3. HPCG (Джек Донгара, написан на C++) — система линейных уравнений методом сопряженных градиентов (минимум на квадратичной функции), алгоритм зашит + типы матриц, но можно использовать свои данные (под каждый тип — определенный способ передачи информации через сеть), поэтому всякая неэффективность вылезет сразу. Матрицы берутся из разных прикладных областей. Есть серьезные отличия от Graph-500: здесь есть серьезный обмен и вычисления.

4 Infiniband

Infiniband — самая популярная система интерконнекта, которая на сегодняшний день применяется в кластерах. Одним из важнейших отличий Infiniband — ограничения на провода: (1) Infiniband через оптоволокно; (2) Infiniband через медь.

Бывает Qdr (количество проводов внутри кабеля — влияет на стоимость технологии: Infiniband гарантирует, что коллизии между проводами не будет). Технология Infiniband поддерживает RMA (Remote Memory Access) — инициированный пакет находится не только в сетевухе, но и может оказаться сразу на месте (позволяет экономить на время доставки). Далее, в Infiniband есть широковещание — широковещательный пакет внутрь Infiniband (а не внутрь Ethernet, как обычно понимается).

В Infiniband есть три понятия:

1. HCA — host - подключение, причем подключенная к PCI-Express; причем сейчас выгоднее послать данные, чем записать на свой жесткий диск (своеобразный кеш).

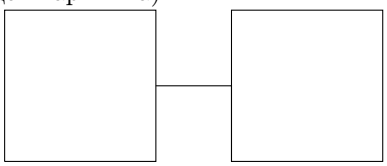
2. DCA — host для назначения данных (приемника, отправляется через switched fabric) — switch-и соединяются между собой. Более того, используется для организации хранилищ (storage). Но разделение узлов через switch есть проблема: чтение из файла мешает для вычисления данных (напряжение switch). Поэтому делают отдельную сеть для организации ввода/вывода (обе сети: основная и ввода – реализованы через Infiniband).

3. Каждое устройство, которое подключено к Infiniband, получает свой GID (Group Identificator), почти как MAC-адрес, но не совсем:

4. LID (local identifier), и при этом есть subnet manager (знает состояние Infiniband сети, в случае проблем — lifetime перестройка данных) — id в терминах subnet manager: (если происходит подключение карты к Internet, то ей присваивается LID). Протокол при начале работы выбирает несколько фишек — и если мы не можем их передать, и протокол маршрутизации выберет новый путь с большим количеством фишек (реализовано на основе модели сети Петри, см. далее). Задача subnet manager —

выставить очередной линк и установить величину порога (завуалированная пропускная способность линка). В Internet (Ethernet) очень сложно работать на такой модели, ибо внешняя среда непредсказуема. Такой способ организации сети гарантирует отсутствие коллизий. Пакет не отпускается, пока он не будет передан (в отличие от Ethernet). При добавлении устройства прописываются данные в таблицу маршрутизаций (subnet manager делает). Хорошо для внутренней работы, но не для организации сетей (ибо сеть будет иметь огромную таблицу маршрутизации, что плохо).

Вопрос: как передать данные? У каждого switch есть набор адресов, куда надо пересылать данные. (Здесь будет картинка)



Нотация. Сеть Петри: есть некоторые узлы (граф) и некоторые пороги. В вершинах графа существуют фишки — могут перемещаться из узла в узел. При этом фишки перескакивают через порог, если в ней накопилось достаточное число фишек.

5 Обзор технологий параллельного программирования

Технологии можно построить по уровням. Можно построить пирамиду по принципу удаленности от железа:

1. Предоставление ОС (pthread, socket, fork, clone) — набор системных вызовов (уровень ОС). Задач: обеспечить переносимое относительно конкретного железа параллельное использование ресурсов.

2. Низкоуровневые библиотеки параллельного программирования (может использовать библиотеки из первого уровня, MPI, openMP, CUDA, openshmem). Задача: абстрагирование от конкретной технологии, через которую происходит передача данных (к примеру, может использоваться Infiniband или разделяемая память IPC). Библиотека на Runtime может определить, какую технологию лучше использовать при передаче данных в рамках узлах кластера. При вводе/выводе MPI может понять, какую коммуникационную сеть использовать (а при этом сами не знать, что в действительности произошло). MPI обеспечивает механизм переносимости ОС. OpenMP предоставляет независимый интерфейс (но компилятор должен уметь организовывать, но реализация идет лучшим способом, как в Windows, так и в Linux, и в Android: если есть thread, то можно использовать определенную опциональность). CUDA должен обеспечить передачу данных на графическую карту и обратно. Openshmem будет хорошо работать при наличии RMA (см. выше), иначе использует MPI.

3. Обертки над вторым уровнем.

   (a) К примеру, binding Python MPI.

   (b) Другой способ — написание высокоуровневых библиотек (на низких уровнях использует (2), а предоставляет программный интерфейс) —

      i. Cublas: (blas – операции с линейной алгеброй). Для IBM есть реализация интерфейса BLAS: ESSL, PESSL (использует конкретные инструкции для IBM), для Intel — mkl, для графической карты — cublas.

      ii. Curandom — библиотека для случайных чисел на графической карте (цель: использование случайных чисел на графической карте).

      iii. FFTW — библиотека для быстрого преобразования Фурье в MPI.

      iv. Hypre работает средствами MPI и openMP (при этом в начале происходит настройка на архитектуру и специфичные тесты: размеры кеша, количество кешей ит.д.), и при этом перераспределяет данные для вычисления данных по линейной алгебре.

      v. Plasma работает на приципе data-flow: если есть маленькие элементы, то удобно сразу собирать данные поточно.

      vi. Petsi — библиотека линейной алгебры.

      vii. Cilk++

   (c) Написание высокоуровневых языков параллельного программирования: компиляция происходит в две стадии – (1) Компиляция в код, к примеру, MPI, C++; (2) натравка компиляторов для параллельных программ. Примеры: DVM — “расширенный MPI + Fortran”, подстановка

pragma. Charm++ – набор charm-ов, которые обмениваются активными сообщениями (вместе с сообщением приходит код для обработки сообщения). При такой модели можно избегать глобальных синхронизаций. При этом сообщения обрабатываются в отложенном режиме (аналогично функциональным языкам): при приходе сообщения вызывается метод char-a (char – объект C++, заранее нельзя сказать, на каком числе будет выполняться код [на практике – не меняется число узлов]).

(d) PGAS: X10, UPS. Модель: n процессоров, у каждого своя память, есть окно, которое будет синхронизироваться для всех процессов (иллюзия общей памяти — при выполнении команды put данные будут положены). Хорошо работает на архитектуре NUMA, (на кластерной архитектуре приемлемо будет работать только с RMA).

4. Библиотеки и языки предметной области: обычно внутри себя используют второй уровень, но могут пользоваться третим. Программист даже не подозревает, что его программа будет работать параллельно (предоставляется язык, понятный для его предметной области): речь идет о DSL (Domain Specific Language): все, в итоге, отображается в параллельную программу, но таких языков все-таки мало. **Все детали скрыты от программиста!**

   (a) Норма: описываются операции над сетками — необходимо для решения сеточных уравнений (теплопроводности, колебаний, Навье-Стокса → отображается в параллельный код). У человека практически нет возможности влезать в код (сравнимо с влезанием в ассемблерный код на C++).

   (b) Games, Gromacs — движение атомов.

   (c) Namd — использует Charm++.

   (d) Flulend, Open Focus, Flow Vision и огромное количество.

6 MPI Advanced. Односторонние коммуникации, MPI_IO

Некоторые устройства подддерживают RMA. Односторонние коммуникации сделаны в MPI для их поддержки. Для этого существует механизм окон. В памяти каждого процесса выделяется область (окно), которая может располагаться произвольным образом в каждом из процессов (даже может быть разного размера). При этом, в каждом окне есть локальная составляющая окна (мы можем знать, где мы находимся в другом процессе — смещение относительно начала окна). Параметры узнаются от функций, которые регистрируют окно.

При регистрации окна в памяти соответствующего процесса происходит отображение в сетевом оборудовании на данный процесс (сетевая карта через PCI-Express кладет данные в память, и процессор в этом мероприятии не задействован, только через коммуникационное оборудование). Хотим получить синхронизированные данные в каждом окне. При этом не организовывается запись в ячейку памяти. Поэтому создается все при помощи функций:

```
MPI_WIN win;
MPI_WIN_CREATE(...); // создаёт окно
MPI_Put(void* buffer, size, MPI_Type, where,); // можно создать shared окно или private окно
--- при этом тот, кто получает инфу, не знает об этом, на PCI-Express объявляется
событие регистрации записи
MPI_Get(...); // можно подсматривать чужую память
MPI_PutR(...); // начиная с версии 3, тогда можно просматривать все через request-ы
MPI_GetR(...); // начиная с версии 3;
```

Нотация. Есть схожая библиотека Open Shmem.

При этом за чтением и записью одновременной никто не следит. Чтобы следить за данными, можно использовать следующие функции:

```
MPI_BARRIER(...); // нежелательно
MPI_WIN_Fence(...); // вводится состояние эпохи: операции с локальной памятью закончены;
гарантирует, что закончилась эпоха
```

```
(после действия никто не производит, к примеру, Put-ы закончились)
MPI_Accumulate(...) // аналог Reduce, выполняет при заполнении необходимые операции (используя RMA)
```

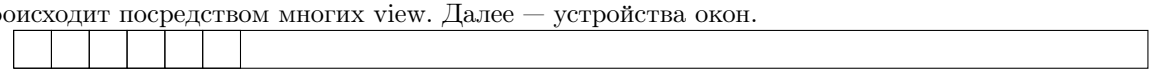
В MPI 3 появились коллективные неблокирующие операции:

```
MPI_IBarrier()
MPI_IBCast()
```

По идее, MPIch может поддерживать такие операциию. Эти штуки крайне сложно реализовать, поэтому ее не было в стандартах (особенно в аппаратуре). При помощи новых штук MPI быстрее работает.

Следующая интересная вещь — MPI_IO. К примеру, у нас есть файл. Позволим в этот файл писать разным процессам через коммуникационную сеть: реализация MPI пытается это оптимизировать (надо ведь доставить информацию до узлов ввода-вывода). Стандартные интерфейсы могут работать, но MPI может быстрее.

В файле создается View. Операция записи производит в View атомарно. Каждому процессу отдается пространство во View. Каждый из процессов пишет в свою область файла (тем самым гарантируется, что разные процессы пишут в разное место, и не надо будет скакать, ибо участки находятся рядом). Операция происходит посредством многих view. Далее — устройства окон.



Функции:

```
MPI_File_open();// открывается файл
MPI_Set_View(); // создается окно - привязывается к типу данных
MPI_Read(); // оптимизирована, отправляются в сумме (но возможно даже, через другую сеть);
MPI_Read_at(); // lseek + read
MPI_Write(); //
MPI_Write_at(); // lseek + write
```

Можно профилировать приложения при помощи pmpi, к примеру, исходник — P_MPI_Send (внутри MPI_Send вызывается P_MPI_Send — можно отправлять всю инфу в журнал). Управляется дополнительными ключами в mpicc.

Не стоит пользоваться client-socket соединениями (один открыл порт 80, организуется сеть через внешний коммуникатор, есть в стандарте: кому-то надо было или хороший contibutor якобы, пример, слежение данных за птицами через wi-fi).

Порождение MPI процессов по ходу дела — MPI_Comm_spawn().

В ранних версиях MPI если один процесс погибал, то убивается все приложения. Сейчас есть MPI_Err_Handler — callback на неблагоприятное событие. Если кто-то сдох, то устанавливается статус и срабатывает MPI_Err_Handler().

Часть II
Распределенные вычисления (Дмитрий Прокопцев, dprokoptsev@gmail.com)

Основным источником литературы является конспект лекций, аудиозаписей и некоторых видеозаписей. Дополнительно информацию можно искать при помощи Google.

7 Основы распределенных систем

Лучше расширять сеть географически. Проблема состоит в том, что с ростом узлов вычислительной системы надежность системы будет падать. Считается, что надежность узла равна $99,5\%$: если у нас 1000 машин, то в любой момент времени в среднем будет лежать 5 (может рубануть электричество, сгореть, затопить и т.д.). Никакие подобные события не будут являться основанием неработоспособности системы (потребности пользователей необходимо удовлетворять). В целом, мы будем заниматься построением систем с учетом выходом из строя.

Определений распределенных систем много, но будем считать, что это та система, которая сохраняет работоспособность при отказе части узлов.

Исторически хотели завуалировать принцип работы системы: надо создать иллюзии работы на одном узле. К примеру, есть программа, мы хотели, чтобы отдельные части (функции) работали в разных частях.

К примеру, локальная программа имеет вид:

```
int divisor(int x);
```

У нас еще маленькая машина, которая вызывает divisor, и огромная машина. Поэтому если есть клиентский код, затем происходит его упаковка (client stub), которая декодируется в байт-код, к примеру, JSON, чтобы можно было разобрать код далее, пересылает RPC-server, пересылает по сети, распаковывает через server stub, и доходит до server code (и сервер не знает, где вообще находится код).

Но есть некоторые проблемы: 1) каким образом надо представлять данные (проблема сериализации); 2) надо написать stub-ы для распаковки и упаковки (будем описывать интерфейсы на специальном языке: какие функции, аргументы, как можно сериализовать). Но не всякий язык годен для этого. К примеру, рассмотрим код на C++

```
void min_max(int* begin, int* end, int* min, int* max).
```

Но указатели — это области памяти, поэтому надо будет более высокая оберка:

```
void min_max(int[] range, out int, out int);
```

На RPC-сервере остается задание посылки/пересылки/отправки на распаковку. Поэтому возможна настройка stub-ов на нескольких языках. В некоторых языках есть свой средства сериализации (Java, Python) — посылать напрямую. Но при этом дихотомии на сервере и клиенте не возникает (один и другой узел может быть использован в качестве сервере).

Но что, если мы хотим не только вызывать функции? Поэтому уже есть желание получит распределенно-объектные системы (посылать сообщения от объектов).

Пример банковской системы:

```
void deposit(acc_id, amount);
void withdraw(acc_id, amount);
```

Хочется дергать системы, но есть проблема: выполнилась одна из них, но появилась проблема выполнения другой. Возникает идея транзакций: но где их вызывать. Непараллельный код транзакций:

```
class TX {

    void deposit(...);
    void withdraw(...);
    void commit(...);
    void rollback(...);

}
TX* begin_tx();
```

Можно ли перенести это на RPC? Проблема в указателе. Можно переслать машинный адрес, но нам нужен физический доступ из необходимого места. К примеру, писать распределенную файловую систему:

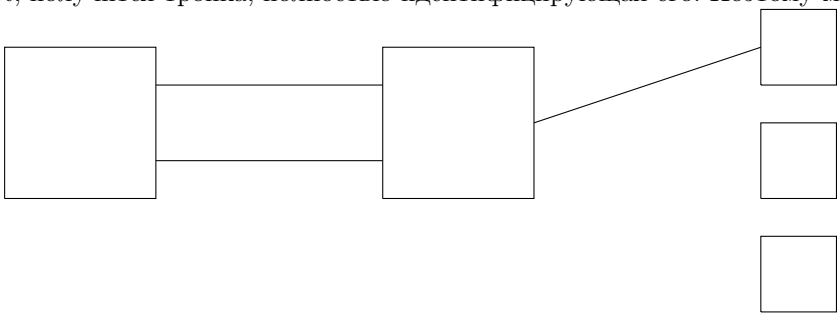
```
class File {

    void read(void *, size_t);
    void write(const void *, size_t);
    void close();

}
File* open(p* dh);
```

Но хотим пересылать ссылки на объект. Обычно указывается еще один объект ссылки, и будем хранить кроме локальных ссылок, еще будут храниться глобальные ссылки. Поэтому рядом с RPC-серверами в модели появляются хранилища объектов. И теперь вызов функции будет происходить следующим образом: как мы будем отправлять указатели: идем в хранилище, достаем идентификатор, и отправить обратно ($map < ObjID, void* >$). Далее оборачиваем идентификатор в клиенте, и отдаем обратно. Поэтому у нас будет целый класс-заглушка (хотим вернуть похожее на прокси-класс). Здесь опять же нет никакой дихотомии (можно пересылать на разные узлы).

На удаленном вызове процедур была еще проблема: как узнать, где код должен выполняться. Обычно знания конфигурируются извне (к примеру, на старте). Но, к примеру, можно присобачить к objId host и port, получится тройка, полностью идентифицирующая его. Поэтому мы получаем хороший прокси.



К примеру, у нас есть узлы 0-99, 100-199, ... Тогда если клиент запрашивает begin: то обратно пересылается через прокси с $objectID$, и затем идет общение напрямую, минуя прокси. Но вопрос: некоторые ссылки нужны уже не будут. Первый подход — вызов delete: происходит удаление объекта при вызове, но надо всем договориться, чтобы все корректно освободили объект и закончили работу с ним. Второй подход — хранить число ссылок на объект (и когда никто не используют — удалять их). Надо следить за копированием ссылок, следить за циклическими ссылками. Оба подхода имеют минусы. На локальной машине можно сделать сборку мусора, но такой объект не подходит. Поэтому проблема освобождения остается в силе.

Но возникают замечательные проблемы с RPC: если что-то отвалилось/вырубилось (умерший компьютер). В течение которого времени стоит ожидать ответ? Мы не знаем политики определения настроек, и как восстанавливать данные. Но если говорить про TSP, то можно запросить снова. Но есть критичные для повторения побочные эффекты (все, кроме вычисления результатов функций).

Далее, предположим, что машина умерла (память сдохла), и все ссылки умерли. Но хранить ссылки везде плохо — получается лишний траффик (пропускной способности не останется). Но еще сложно отличить умершую машину от умершей сети (выдернули link, а затем вернули, и начали использовать ссылки их невалидно). Важно помнить, что запросы доходят только через некоторое время!

Все вышеперечисленные проблемы наделены проблемы излишней абстракции. И когда мы не знаем семантики, мы не в состоянии обеспечить абстрагировать распределенность системы, т.е. создавать иллюзию локальности. Решить проблемы в рамках данной парадигмы практически невозможно.

Все это было в районе девяностых систем. А что, если не пытаться решить все проблемы сразу? Распределенные вычислительные системы, к примеру, решают свои проблемы. Распределенные базы данных решают другие проблемы (хранения данных). Таким образом, если решать конкретные задачи, то мы получим лучшие результаты, решить больше проблем. В итоге можно предоставить более хорошую иллюзию.

Сейчас этот метод (без специализации) не очень часто используется, но в зачаточном состоянии есть наличие RPC. К примеру, в распределенной базе данных мы хотим читать 10 строк, а затем "позвонить" и читать из курсора еще 10 строк (по сути ссылку на объект БД). Похожие на RPC системы можно встретить в public interface. (К примеру, Google Maps API: но + под капотом идут перезапросы, распределителей, обход проблем внешними средствами).

8 Парадигмы проблем, виды отказов

У нас есть распределенная система и часы, которые следят за временем. Но иногда ломаться может все. Рассмотрим некоторые парадигмы:

1. Fail-stop — узел сдох окончательно, требуется реконфигурация.

2. Fail-recovery — узел помер на время, он может быть восстановлен самостоятельно или посредством человека. При этом реконфигурация не требуется. Но есть проблема со стиранием некоторых состояний в узле (и надо решать актуальность системы)

Заметим, что в этих случаях узлы играют так, как мы ожидаем (посылают сообщения по протоколу, может быть, ломаются). Но могут быть злонамеренные узлы (не играющие по правилам):

3. Byzantine (Византийские узлы) — он может сказать одному узлу, что живой, а другому — что умер (или передавать специально неактуальную среду). Такие системы могут быть созданы, к примеру, для посылки мусора, или получать секретные данные, поддельные DNS-серверы. Чтобы решать проблемы, можно собрать общее мнение у узлов (хотя бы $\frac{2}{3}$ от всего количества).

Еще может падать линк (соединение между узлами).

1. Perfect — совершенные каналы связи (принимает все сообщения ровно один раз в том же порядке). Можно воссоздать, к примеру, в switch-е или доверительной системе.

2. Если сообщения теряются +− честно (Fair-Loss links) — для любого сообщения существует положительная вероятность доставки сообщения. Из Fair-loss можно сделать Perfect (посылать много раз одно сообщение).

3. Byzantine — перестановка сообщений местами, замена сообщений. Встречается в более частой среде, нежели чем между узлами (к примеру, Московское метро). Но при этом, если поставить определенные требования к сообщениям (к примеру, криптографическое шифрование). Поэтому можно изменить Byzantine в Fair-Loss.

Таким образом, по транзитивности можно получить из Byzantine соединения в Perfect. При этом еще можно добиваться подтверждения сообщения. При этом, если не хотим хранить сообщения, то можно хранить их номер. Поэтому можно добиться результата с определенной вероятностью успеха.

Но, конечно, не всегда можно различить отрыв линка от отрыва узлов.

Осталось рассмотреть часы. Но мы хотим узнавать факты относительно производительности системы:

1. Sync — есть верхнее ограничение на время пересылки и время обработки (тогда можно делать предположения относительно нашей системы). К примеру, если есть протокол, который через время τ должен выполнять операцию, то если через 3τ должны получить ответ.

2. Eventually sync — система почти всегда асинхронна, но есть гарантия, что наступит определенный момент времени, когда наступит синхронизация.

3. Async — асинхронные системы.

При написании распределенной системы надо выбрать мир, в котором мы собираемся создавать распределенную систему (от этого будет зависеть сложность алгоритмов: sync — достаточно простые алгоритмы). Но достаточно сложные алгоритмы существуют (к примеру, BitCoin).

9 Алгоритмы на распределенных системах

Алгоритмы на распределенных системах решают злободневные задачи.

Пример. Разделяемый регистр (Shared Register). В ней есть две операции:

```
-> write(v)
-> read() -> v
```

Но система настолько сложна (проблема репликации, проблема разрыва и тому подобное). При переходе на ненадежные системы простейшие операции могут быть достаточно сложны

Пример. Broadcasting — надо гарантировать, что до каждого узла дошло определенное значение. Если система надежная, то можно послать все подряд. Но могут быть проблемы: 1) узел ушел, а затем вернулся; 2) узел, посылающий сообщения, сдох. Небольшое решение проблемы: получил одно, передай другим, и пометь сам, что получил сообщение.

Пример. Consensus — есть операции $propose \to x$, $decide \leftarrow x$. Несколько узлов посылают значения, а затем решается, какое значение выбрать.

Пример. Leader Election — иногда в равноправной системе надо выбрать лидера самостоятельно. При этом мы хотим, чтобы лидер всегда был один. Могут происходить события $leader \leftarrow n$.

Проблемы с алгоритмами начинаются в том случае, когда от системы требуется меньше проблем. Далее будем рассматривать модель **Consensus** в случае Fair-Recovery.

Алгоритм. *(PAXOS) В данной системе будут существовать определенные роли:*

1. Client.
2. Voter (Acceptor).
3. Coordinator (Proposer).
4. Learner.

Изначально клиент выбирает proposer и посылает ему propose на достижение консенсунса. Далее proposer выбирает номер попытки и посылает сообщение $prepare(i)$. Часть из них будет отвечать на этот запрос: посылается $ack(i, x)$ или опровержение $nak(j)$. У каждого $voter$ есть текущее значение i и x. Если $propose$ получает больше половины подтверждений. Если все посылки были пустыми, то посылается далее $accept$ на $voter$. Если в этом случае будет $accept$, то $voter$ посылает сообщение на $learner$. И если $learner$ получает более половины подтверждений $decide$, то алгоритм считается выполненным. Иначе если получаем $nak(j)$, то можно послать $nak(j + 1)$. Больше половины — чтобы кворумы пересекались, и система могла восстановить актуальную информацию.

Если оторвался $voter$, то ничего страшного. Но есть проблема, если отлетел $proposer$, то тогда клиент может начать посылать узлы p_2. Но при этом такая система может реагировать нормально (в итоге, до $learner$ дойдет два сообщения $decide\ x$, хотя было на p_2 $decide\ y$). Пример:

1. $P1 \to decide\ x$
2. $V \to ack(1, empty)$
3. $P2 \to prepare\ 2$
4. $P2 \to ack(2, empty)$
5. $Wake\ 1$
6. $decide\ y\ from\ p_2$
7. $P1 \to accept(1, x)$
8. $V \to nak(2)$
9. $P1 \to prepare(3)$
10. $V \to ack(3, y)$
11. $P1 \to accept(3, y)$
12. ack
13. $decide\ y$

В итоге, можно заметить, что системе неважно, к какому значению пришел результат.

Самая плохая ситуация:

1. $P1 \to decide\ x$
2. $V \to ack(1, empty)$
3. $P2 \to prepare\ 2$
4. $P2 \to ack(2, empty)$
5. $Wake\ 1$
6. $P1 \to accept(1, x)$

7. $nak(2) \to P1$
8. $P1 \to prepare(3)$
9. $Ack(3, empty) \to P1$
10. $P2 \to accept(2, y)$
11. $Nak(3) \to P2$
12. $P2 \to prepare(4)$
13. $Ack(4, empty) \to P2$.

И такой процесс может продолжаться игра. Но нельзя гарантировать одновременно $leaveness$ и $safety$. Решение проблем: 1) крут тот, кто имеет высокий приоритет; 2) выполнить *sleep* на определенное время. Тогда вероятность завершения процесса очень сильно увеличивается. Таким образом, алгоритм сойдется почти наверное.

В реальной жизни, внутри одного узла может выполняться сразу несколько ролей. Этот алгоритм адресован к одному консенсусу (после этого произойдет крах, и желательно закопатить узлы). Поэтому новая попытка достичь консенсус не должна зависеть от предыдущей. Обычно эта проблема разрешается установкой определенного id к номеру достижения консенсуса. Данный алгоритм является основным блоком для достижения согласия.

10 Распределенные базы данных

База данных — набор структурированных данных, удобных для модификации и использованиии и интерпретируемых через некоторый понятный человеку язык. Стандартная модель — магазин: *customer*, *order*, *goods*. В базе данных есть система управления БД (СУБД), которая позволяет выполнять операции $SELECT$, $UPDATE$, $INSERT$, $DELETE$. Плюс еще есть концепция транзакций: атомарной в предметной области операции (все выполнены или ни одно не выполнено).

Свойства транзакций:

1. Атомарность
2. Консистентность (*consinstency*) — после транзакций сохраняются инвариантов.
3. Изолированность (*isolation*) — параллельные транзакции выполняются последовательно.
4. Долгожительность (*durability*) — после коммита откат самостоятельно выполнен быть не может.

Появляется такое сочетание, как $ACID$. Но в реальных БД это не так: обычно, бывают проблемы с изолированностью.

Пример 5. Есть два узла, каждое из которых хранит одно число (как общее хранилище), и мы хотим делать операции *read* и *write*. При *write* мы бы хотели записывать в оба узла. То есть, если есть значения x_1 и x_2 с инвариантом $x_1 = x_2$. И создаются многочисленные проблемы, аналогичные рассмотренным в прошлом разделе.

В обычной БД либо она работает, либо она падает. В распределенной БД некоторые операции могут дойти и выполниться, а некоторые — нет. Поэтому к распределенным БД были выдвинуты следующие требования:

1. **Availability** (доступность) — в работоспособной системе любая операция должна успешно завершаться за конечное время.
2. **Consistency** (согласованность/линеаризуемость) — в идеальном мире запрос получается запрос, и сразу на него получается ответ, но появляется прослойка клиента. Поэтому появляется некоторый промежуток времени для отправки сообщения и получения данных. Если операции не пересекаются, то все ОК. Линеаризуемость: если операции пересекаются по времени, то они выполняются, но при этом далее будет гарантированно получено, а операция, которая закончилась ранее, не видит результат операции (к примеру, *write*).
3. **Partition tolerance** — сеть между узлами может меняться на сколь угодно большое время, и она должна как-то жить с этим (не предъявляет надежность). При этом понятно, что мы не должны ждать операции с availability (гарантии выполнены).

В 2000 году была сформулирована теорема.

Теорема. *(CAP-теорема) Все три свойства не могут быть одновременно выполнены в распределенной системе*

Доказательство. Идея доказательства: если требовать надежную сеть, то получим $!P$, если отказ на запись — то $!A$, если необходим *stale read*, то не $!C$ □

Таким образом, можно соорудить такой треугольник:

1. **CP-systems** — пытаемся раскинуть все данные по узлам, и запускаем алгоритм достижения консенсуса, поэтому есть кто-то отлетел, то ребята собирают кворум и совместно договариваются. Другой подход — проксирование через лидера (выбор лидера, а затем незаметно пересылают данные лидеру, который их упорядочивают). Если лидер умирает, то выбирается новый лидер. Так как можно

2. **AP-systems** — храним всю информацию в копиях, а затем на запись говорим, что запрос выполнен, а откладываем ее на потом (появляется понятие *eventual consistency* — существует момент, когда появится необходимая запись).

3. **AC-systems** — если нет линка, то говорим, что ненадежное состояние.

CAP-теорема дала новый толчок. Но бывают БД, которые сложно причислить к одному из типов треугольника (они почти есть, но строгой математической основы нет). Поэтому внутри треугольника может быть множество подтипов. В реальности, мы не можем отказаться от partition tolerance, но на самом деле, между A и C нет дихотомии. А если лучше задуматься, то мы можем создать барицентрическую систему координат (тетраэдр), в которой можно получить лучшее ускорение.

Насчет A, можно договориться, какие операции выполнять, какие нет. Насчет C, необходимо придумать правила согласованности. И таких моделей больше одной:

1. **Linearization** — есть дополнительный аспект: если между write есть read, и получены новые данные, то мы должны во второй раз вернуть новое значение

2. **Sequential consistency** — концепция последовательного доступа с одного клиента. Пример доставки: если есть операция от клиента, то он говорит OK и доставляет выполняет после. Далее на другом узле происходит синхронизация, поэтому новый откат может привести к чтению назад. Аналогичная ситуация происходит и в Твиттере. Есть проблемы: 1) write buffering, 2) read caching.

3. **Casual consistency** — некоторые операции должны соблюдаться причинно-следственные связи (такие операции должны быть выполнены последовательно). Остальные операции могут быть выполнены параллельно (к примеру, комментарии в LiveJournal могут появляться параллельно).

4. **Serializability** — никаких требований по упорядочиванию операции. С одной стороны эта модель слабая, а с другой — очень сильная: хотим глобальной картины мира, при этом операция либо сразу подтверждается. В итоге, сериализуемость не всегда может быть сериализована. Таким образом, сериализуемость дает порядок на глобальной картине мира.

Конечно, есть еще более слабые связи.

Пример. Если три узла: n_1, n_2, n_3. $client \to n_2 \to n_2 \to return$. Далее, $client \to n1 \to n3 \to fail$. А в конце, $client_2 \to n_2 \to 10$. Произошло получение записи, которой нет. (read ancommited). Небольшой плюс — производительность.

Чем больше мы двигаемся в линеаризуемость, тем больше мы должны платить. Иногда важно только то, что мы получили некоторую информацию.

Если у нас есть логика: после $fail$ отдать новый $write$, то получаем *eventual consistency*.

В итоге, когда проектируем распределенную систему, надо выбрать уровень согласованности, который нам необходимо.

Пример. Если банкомат не подключен к сети, то необходимо сделать выбор: банк идет на поступок — идет упор на доступность, а если баланс ушел в нуль, то мы потом будем начислять проценты (из-за своей несогласованности).

Реальные базы данных редко находятся в одной точки треугольника (обычно идет тонкая настройка относительно необходимой операции).

Пример. Есть Postgres (конец 1990-ых годов), master и набор slave-ов. В Master node можно отсылать операции (SELECT, INSERT, UPDATE, DELETE), а тот реплицирует на slave-ы (master подтверждает их). А запросы можно отправлять на slave-ы или на Master. Но надо необходимо назначать master и slave-ы. Поэтому если происходят проблемы, то происходит read-only чтение до тех пор, пока не придет человек и починит. Но сейчас уже есть сериализуемость и линейность. Но и этот подход не очень: кластер отправляется в read-only, или сделали запрос, в этот момент ломается сеть (частично происходит запись, происходит fail, но при этом могут появляться ошибочные записи об ошибках). Поэтому надо стараться делать записи индепотентными (повторять их). Еще одно решение - создать primary key на каждом узле отдельно. update set x = 10 иденпотентная, а update set x = x + 1 — нет. Postgres дает реляционный подход к данным, и по идее, он и применяется (в области распределенных реляционных базах данных вроде бы ничего нет).

Пример. База данных **Redis**. Напомним, что в треугольнике между C и A можно свободно двигаться в терминах CAP-теоремы. Redis в этом месте сдвинут в сторону согласованности и заточен под высокую производительность операций. У нас есть набор узлов (node), зачастую работает чисто с памятью. Redis — простой key-value хранилище, есть операции $read(k) \to v$, $write(k, v)$, $cas(k, old, new)$ — запись новых данных, если сравнение со старым значением проходит. Есть выделенный мастер, который обрабатывает все запросы на изменение. Redis настроен на производительность, делает операции локально, подтверждает их, а затем асинхронно реплицирует их (не смотря на результат). Достоинством такого подхода является следующее: интервал между *write* очень мал, около 10000 операций в секунду. Если master умирает, то система приходит в хаос. Перевыбор мастера происходит так: если мастер находится не в большинстве, то происходит перевыбор. Пример: 5 узлов, разбиение на 2-3 (мастер там, где два узла). Поэтому есть два мастера, и кластер уходит в полную рассинхронизацию. В итоге, после восстановления, большая часть реплик потеряется. Процент потерь порядка 56%. С доступностью у нас проблемы, с согласованностью тоже. Но есть производительность. Поэтому их можно держать в кешах, комментарии пользователей на предобработке (данные не предоставляют ценности, ибо лежат еще где-то или нужны на короткое время).

Пример. База данных **MongoDB**. Эта база данных — не SQL, но дает больше простора. Она хранит документы в формате JSON.

```
{''name'': ''john'', ''orders'': [1,2,3], ''foo'': {''x'': ''y'', ...}}
```

Похоже по возможностям SQL, но можно делать изменения в рамках одной таблице. Репликация происходит аналогично Redis: есть кластер и выделенный мастер. Мастер выбирается репликами сам (как и в Redis). Только мастер обрабатывает записи, и только он имеет read-only. Но если нам необходим stale-read (еще не произошла реплика на новые данные). Все операции read происходят из одной реплики (какая к нам ближе, к той можем и обращаться). Но на write есть методология write concern: список того, что надо сделать, чтобы запись стала успешной.

1. 0 — все разрешено.
2. 1 — мастер работает аналогично Redis.
3. 2 — запись должна быть обработана на мастере и на одной из реплик (первая успешная репликация не на мастере возвращает OK). Происходит защита данных при отрубании одного из узлов (но проблемы связности есть).$Majority$ $(\frac{n}{2} + 1)$ — перевыборы мастера проходят спокойно: выберется машина с более актуальными данными. Все операции в Mongo журналируемы, поэтому они договариваются. Но нет транзакционности. К примеру:
   (a) $read \to 5$
   (b) $write\ 10 \to fail$ (операция происходила локально, но на репликации ничего не произошло, и новые данные будут возвращены)
   (c) $read \to 10$

   Таким образом, нет атомарных коммитов на данные.

В момент разбиения могут быть stale reads, помимо dirty reads: перевыборы происходят не одновременно (два узла считают себя мастерами). К примеру, идет запись $w\ 10 \to M2$, проходит OK, а на $M1$ может пройти read через 5. Но если мы готовы мириться с небольшими простоями, более того, можно использовать $compare - swap$, есть в виде $findAndModify$, который еще и возращает документ. Поэтому если отправить на majority, то все договариваются, появляется линеаризуемость, но производительность в этом случае сильно падает.

Далее будем говорить о базах с *high availability* (предыдущие базы старались удовлетворять согласованность насколько можно).

Пример. База данных Riak (последователь DynamoDB). Это key-value storage (strings). Есть достаточно большое число узлов, мы считаем несколько хешей от ключа (обычно считают один хеш, а дальше инкремент). Все хеши отображаются на общий круг, который делится на секторы приблизительно равного размера (консистентое хеширование). Несколько виртуальных узлов объединены одним физическим узлов. Когда мы хотим выполнить операцию, то считаем хеш, идем в ту область, где он может храниться. При этом стараются сделать так, чтобы соседние данные лежат на разных нодах. При этом каждый узел (в нашем случае, полагаем три) можно выполнять read-write операции. При этом если через R обозначить число узлов на чтение, а W — на запись, то неравенство $R + W \geq N$ дает согласованность системы. При этом мы читаем данные последней актуальности. Если в нашем случае $R = W = 1$, то имеем полную свободу на каждом узле, и при разделении надо будет сливать данные. Поэтому, варьируя значения величин R и W, можно настраивать систему. К примеру, если $R = 1$, $W = N$, то имеем полное чтение, но запись очень медленная и требует согласованности всех данных, поэтому они согласованны. Если $R = N$, $W = 1$, то получаем обратную ситуацию (только сливать данные непонятно как). Что произойдет в случае одновременной записи на разных репликах? Первый подход — кто последний записал, тот и молодец. Но такой подход некорректен, ибо нельзя достичь консенсунса. К примеру, если $R = 2$, $W = 2$. Если мы питаемся работать на M_1,M_2 и $\{M_3\}$, то тогда первый записал, реплицировал, но при этом с M_3 можно пробить данные на первый узел (затерев старые данные). Второй подход ($CRDT$) — при одновременной операции разрешение конфликтов отправляется пользователю. Но если операция ассоциативная, коммутативная, идемпотентая, то можно применить композицию $((x_0 \oplus a)\ (x_0 \oplus b) \to (x_0 \oplus a \oplus b))$, и Riak может предоставлять возможность для слияния данных. При $R = W = 1$, можно достигнуть *eventual consistency*, но даже *read your rights* (нельзя даже гарантировать, что записанные свои данные не будут получены).

Пример. База данных Cassandra. Заточена под запись чисто новых данных в достаточно больших количествах (обновлять данные **нельзя**). Поэтому нет проблемы разрешения конфликтов (access log отлично подходит для этого, и поэтому про нее можно забыть). Раз писать надо много, то у нас $W = 1$, при этом $R = N$ (обычно аггрегирующий запрос для некоторой статистики). В остальном Riak и Cassandra похожи. Внутреннее устройство: есть журналы на каждом узле (SST — sorted table, данные скидываются в сортированную таблицу), а далее создаем новый журнал. Чтобы найти сортированные данные, можно использовать бинпоиск, а слияние — $MergeSort$. Чтобы SST не плодились, происходит чистка, и создается новая, более новая таблица (происходит подмена метаданных). Между узлами выглядит аналогично Riak.

Вывод. *Есть два подхода в строительстве распределенных систем: 1) есть мастер — получаем большую согласованность; 2) (R/W/N) есть запросы на все узлы, но при этом нет консистентности, можно достичь eventual consistency. Но линеаризуемость достигается только некоторыми ухищрениями (но это практически не нужно). Если нам нужны все изменения, то нужен только write-only, иначе — мириться с этим.*

Пример. База данных ZooKeeper. В каждой записи около сотен данных, хранит всякого рода конфигурации систем. Есть несколько узлов, и на каждую операцию запускается поиск консенсуса. Это одна из немногих баз, которая обеспечивает строгую линеаризацию в условии ненадежных сетей.

11 Распределенные файловые системы

Перед тем, как говорить о распределенных базах данных, поговорим о файловых системах в целом. Набор баз данных дают достаточно большой функционал (не только реляционнные, к примеру, $key-value\ storage$). Такой интерфейс давал консистентность и тому подобное, он уже сильно похож на файловую систему, где ключ — имя файла. Но почему есть отличие от обычной ФС? В key-value storage запись занимают килобайты, но файлы могут иметь большой объем данных (нескольких гигабайт). Но все целиком читать сложно, поэтому появляются фрагментарные операции над файлами (чтение фрагмента $read(file,\ offset,\ size)$, запись $write(file,\ offset,\ size)$, и замена ключа записи (переименование)). Более того, данная система должна работать быстро (быстрое отображение видео).

Первым способом создания распредленной системы являются системы **NFS** и **CIFS.** Каждому каталогу приписано имя хоста (к примеру, host1:/var/lib/file). Тогда, чтобы обратиться к локальному файлу, надо обратиться к ядру (VFS), а затем идет обращение к диску. При этом рядом между NFS и user space сажаем NFS Daemon с тем же набором операций. Преимущество: нет проблем с когерентностью, нам не надо практически переделывать интерфейс, локальный интерфейс совпадает с сетевым (и не догадываться, что работают с файлом из другой машины), и после монтирования даже можно не подозревать о связи; плюс хорошая производительность, сетевая прозрачность. Но может быть такое, что один файл открыт на двух разных машинах, и проблемы экономии (кеши, буферизация) играют немаловажную роль.

Немного о буфферизации: происходит чтение одного куска, быстрое чтение, при записи: запись в буфер, а затем синхронизация при записи в диск.

В итоге, возникает проблема когерентности кешей: разные клиенты не могут организовать корректные операции кешей. Поэтому появился механизм **opporunistic locks**. Почти всегда файл открывается на монопольном режиме. Поэтому ФС может отдавать блокировку, которую может отзывать. После открытия файла у другого клиента, посылается отзыв, сбрасываются кеши, и отключается кеширование.

Проблемы: данная система не является распределенной в обозреваемом смысле (нет реплицируемости, сеть работает идеально, винчестеры не сносятся). Поэтому такие системы применяться не могут (только в идеальных условиях).

Идея: поставить proxy рядом с демоном, которая умеет обращаться с демонами, каждый из которых находится на другой машине. При этом конфигурации могут быть разными (разные части файлов — на разных узлах (+ производительность), или реплицирование (+ избыточность, надежность)). Такая концепция с ретрансляторами со схожим IPC называется ClusterFS. Заметим, что конструкция может не древовидной (может быть еще один proxy, который ходит в те же back-end).

Но данная концепция слабо отражает реальные проблемы (при слете может произойти рассинхронизация). ClusterFC работает в режиме FailStop (она не вернется без человеческого вмешательства). Данная конструкция не переживает сетевого разбиения (крупная рассинхронизация), на разных репликах знает разная информация.

Нотация. Cluster split brain — сколько головной боли может быть.

Но за счет разбиений можно создавать огромные файловые системы, которые работают с большой скоростью, если нам не важно полностью надежное хранение информации. Таким образом, данная концепция является **AC-системой**.

В 2000 году Google опубликовал статью про собственную файловую систему GoogleFS (которая была уже заменена пару раз). Рассмотрим ее внимательно. Данная система работалась на ломающемся железе. Google делал систему для себя (не нужна вся мыслимая функциональность).

1. Создание файла
2. Последовательное чтение файла (рандомное плохо работает)
3. Дописывание в конец файла
4. Удаление
5. Переименование

Файлы разбиты на блоки одинакового размера (около 64Mb). Есть несколько узлов, на котором файл живет. При этом есть информация о файлах (MetaInfo Server): список файлов, список блоков, где находится блок, и тому подобное. Есть несколько Storage (хранилище) в которых хранятся файлы, клиент хочет получить файл (получаем chunk, обращаемся к MetaInfo, получаем список машин, где находится primary Node, идем по определенному chunk-у и). Дописывание к chunk-у: storage распространяет информацию, но commit не происходит, получаем подтверждение от других узлов. Как то,лько информация должно до пользователя, происходит commit. Далее происходит линейное реплицирование (данные от всех пользователей будут записаны, ибо имеем append). После того, как все будет упорядочено, происходит реплицирование на файлы, запись, а затем возвращается информация о записи к пользователю. Create, erase, rename не произвиодит работу с блоками. Snapshot-ы делаются быстро: копирование версии происходит за $O(1)$. Если отвалился один из Storage, то происходит реплицирование на новый Storage. Для понимания синхронизации, в блоках хранится номер версии (если запись не доехала, то Storage будет считаться недоступным, пока он не сотрет данные и синхронизируется). Если отвалился primary, то идет обращение к secondary, и он назначается primary (нет проблемы с несколькими primary). Поэтому получаем лианеризуемость и разбиение, но при этом не будет консистентности.

Но, возможно, при обвале сети, может произойти dirty read.

Что произойдет, если обломится Meta сервер, то можно их наплодить. Но внезапно выясняется, что meta-сервер является распределенной базой данных, просто он хранит намного меньше данных (информация о блоках много меньше, чем размеры файлов). Большая часть проблем с согласованностью прячется в более малой инстанции. Раз распределенная система хранит внутри себя другую, то свойства в терминах CAP-теоремы **наследуются**.

В некоторых случаях отказ согласованности переносится на случайных характер, но если мы будем запускать консенсус и контролировать их (то далеко не уедешь), то приходится идти на уступки. Вероятность недоставки больше 0, но настолько мала, что может происходить редко, или предупреждать пользователей о **dirty reads**.

Со скоростью операции не все очень хорошо (мы идем в распределенную БД, потом идем в DataStorage, + запросы между дата-центрами). Но все чтения последовательны, и мы их читаем по долгим позициям. Поэтому клиентский код может заранее запросить данные (1001 блок запрашиваем заранее, чтобы снова

не получать информации), и на низком старте будет лететь информация. Значит, можно значительно ускорить получение данных. При этом мы получаем high latency, но получаем high throwput (состояние абсурда, произвольное чтение — очень медленно, последовательно — быстро).

Что нам дало запись в конец?

1. Упрощение коммита (если в середину, то надо будет отложить ее, но если что, нужно будет откатить данные; появляются большие проблемы — перезапись такой же информации, что может привести к тем же проблемам). Если мы будем добавлять, и при получении дописываем сразу данные в конец, но запоминаем, что не обновляем информацию, а просто переставляем указатель (откат — обратная перестановка указателя). Поэтому можно составить цикл двухфазных коммитов (не нужен консенсус, трехфазный коммит и тому подобные сложные системы).

2. Примонтировать такую файловую систему сложно. Клиентский код должен знать, в какой файловой системе мы работаем. Надо делать привязку к модели (как обычно и происходит).

Таким образом, рассмотрены два типа систем. При этом если нам необходимы лишь некоторый функционал, но мы не должны абстрагироваться от среды, в которой мы живем.

12 Распределенные системы вычислений

У нас были распределенные системы, которые умели хранить данные. Теперь будем преобразовывать их. Но при этом произвольным образом так сделать, конечно, нельзя (ибо мы получаем множество проблем, аналогичных ранее) Поэтому народ, которые это делали в Google в 2004 году, решили ограничить пользователя в совершаемых действиях (меньше возможностей работы с системой).

Они сделали следующую вещь: они были поклонники функциональности (есть структура данных список, и некоторые функции:

```
map: (func, list) → list2 // применять функцию к каждому элементу списка
reduce: (func, list) → return f(x_1, f(x_2, ..., f(x_n, 0)))
```

)

Поэтому Google реализовал фреймворк MapReduce. У нас есть список пар [(key, value), ...]. Есть операции:

```
map: (key, value) → [(k_1, v_1), ...]
reduce: (key, [v_1, ..., v_n]) → [(k_1, v_1), ..., (k_n, v_n)]
```

С помощью таких операций можно выполнять достаточно широкий класс задач: от grep до достаточно сложной вещи $TF \times DF$ (если есть набор $(url, text)$, то можно посчитать такую вещь). Операции слияния, соединений тоже можно делать при помощи MapReduce.

Заметим, что каждая операция является изолированной (map требует только одну запись). У данных функций нет побочных эффектов — только результат. Таким образом, операции можно запускать столько раз, сколько нам понадобится.

Рассмотрим, каким образом может быть устроена такая тривиальная система. У нас есть много машин, на каждой из них лежит некоторый объем данных. Операция *map* выполняется параллельно на каждой машине. Более того, можно распараллелить данные даже внутри локальной машине (в несколько параллельных потоков, равных числу процессоров, к примеру). После обработки данных, были получены промежуточные результаты. Далее необходимо запустить операцию reduce. Нам необходимо перегруппировать данные. Первое, что приходит в голову — для каждой записи посчитать хеш от ключа, а далее разбить все на блоки по группам хешей (и начать пересылать данных). Полученный набор несильно упорядочен, но их можно отсортировать на локальной машине. В итоге, записи группируются по куску данных, и все записи, относящиеся к одному ключу, идут подряд. И после этого уже можно получать reduce, и шаг вычислений после reduce считается законченным.

Дополнительно можно бить еще на блоки по суммарному числу процессоров (3 машины, 16 процессоров — 48 ключей), и тогда на 48 процессорах можно параллельно запустить операцию *reduce*. Далее можно повторять итерации алгоритма $map - reduce$.

Ясно, что данные надо дублировать (важно хранить изначальные данные). Но если умер процессор, который обрабатывал данные, то можно запустить данные на соседней реплике (ибо нам не нужна интерактивность — задача послана, ждем результат). Таким образом, можно брать дополнительную копию на случай падения. Единственная гарантия — надо отслеживать, какие фрагменты данных откуда были получены (чтобы не было повторных данных, идея — хранить под-chunk-и для хранения данных).

Почему после map должен идти reduce? Поэтому сделаем pipeline, который может преобразовывать данные (к примеру, map→reduce→reduce). Поэтому появляется свойство сортированности таблицы. Заметим, что map делает несортированную таблицу, а reduce на вход требует сортированный вход. Значит, необходим сортировщик данных. В современных реализациях есть операции map, reduce и sort. Граф операций может быть произвольным ациклическим, но с одним условием: перед цепочкой reduce должен идти sort.

Следующая мысль: таблицы могут быть составными, в ней могут храниться подтаблицы. Поэтому, может происходить дописывание данных в таблицу. Каждый раз, когда мы делаем append, таблица не может быть сортированной, но sort схлопывает все блоки в одну таблицу. Таким образом, можно убить старые данные, ибо сущности стали более общими.

Если есть несколько таблиц, в которых данные посортированы. Поэтому можно выполнять merge-sort (аккуратно дополнять мета-информацию). В итоге, конкатенация может происходить за время $O(1)$. В итоге, кроме worker-ов, нужен мета-сервер для дирижирования кластеров. Как и в случае с базой данных, единственной точкой отказа будет этот сервер. Только в этом случае делается упор на согласованность (ибо не было real-time, но хороший high-availability).

Когда мы сабмитим задачу, необходимо выполнить функции. Но до кластера необходимо дотащить информацию до машин. Поэтому необходима сериализация мета-данных. Можно использовать технологию streaming (исполняемый файл — пересылай данные). Проблемы есть с C++ или с Java. На Java можно оторвать класс. На C++ нельзя выдрать кусок адресного пространства. Поэтому приходится переслать программу, но понять, что программа запускается как worker, а не как master, и надо запустить команду. В итоге, накладываются ограничения:

```
int main(int argc, char** argv) {

    initMapReduce(argc, argv); // init code

}
```

Такой код будет написан практически везде, ибо необходимо будет заменять структуру данных.

13 Безопасность в распределенных системах

Ранее мы не говорили про злонамеренность данных в распределенных системах. Если система работает не в 100-процентно надежной системе, то мы не можем верить тому, что может быть запущено на нашем компьютере (непонятно что происходит на пересылке данных (Москва - провайдер - оптоволокно - провайдер - Новосибирск)). Кто угодно может покопаться в наших данных. Проблемы: подслушивание, притворство данных и много дополнительных проблем.

Первое решение проблемы — криптография. Криптография основана на операциях, которые в обратную сторону сложно вычислимо (произведение простых чисел). Хеш — функция, которая однонаправленно меняет данные (невозможно подобрать такое x, что $hash(x) = y$, кроме как прямого перебора). Если хеш-функция хорошая, ибо тогда можно гарантировать, что если хеши совпадают, то и сообщения совпадают. Примеры: SHA1, SHA256.

Далее рассмотрим механизм симметричного шифрования данных. Пусть есть ключ K, есть операции шифрования $E(m, k) \to c$, и дешифрования $D(c, k) \to m$, которое по m и c невозможно узнать k прямой подменой текста. Из таких механизмов есть 3DES (устаревший) и AES.

Ассиметричное шифрование — ключи шифрования e и d: $E(m, e) \to c$, $D(c, d) \to m$. Одним из известных алгоритмов является RSA.

Алгоритм DH согласования ключей работает таким образом: он может согласовать ключ участниками сессионного обмена, чтобы его нельзя было узнать третьим лицом (смотри комбинаторику на первом курсе).

Таким образом, мы можем удостоверять подлинность того, что мы послали: RSA может работать как шифрование, так и проверять (закрытый ключ у себя, но открытый ключ публиковать, и возникает понятие цифровой подписи).

Кажется, что это дает достаточный механизм защиты. Есть одна проблема: все механизмы не решают проблему аутентификации участников сети. К примеру, по DH решили делать случайный ключ. Представим двух участников сети.

$a \to x(third) \to (bad\ message)y$. Далее он посылает x^a первому, в итоге создается впечатление, что ключ согласован (но на самом деле посередине есть расшифровщик, который все это портит). А участники не догадываются, что что-то происходит. С таким классом атак ($MITM$ - man in the middle) очень сложно делать. Мы можем сделать надежный канал до первого человека, выдавшего себя за участника сообщения. В итоге, нам необходимо проделать механизм защиты.

Самый простой способ: создать два pre-shared-key, и потребовать совпадение в двух участниокв (send x, return $hash(x+psk)$). После установления соединения можно установить подлинность соединиения (при этом надо сделать это после, ибо иначе man in the middle может пропустить необходимую данные). Поэтому данные операции (проверка подлинности и согласования ключа) должны идти одновременно. Поэтому посылаем числа x, y, считаем $hash(x+psk)$, $hash(y+psk)$, отправляем $hash(x+psk)$, но согласованность получаем из-за хранения $hash(y+psk)$. Есть недостаток: на каждую пару соединений создавать новый ключ. Если хранить общий ключ, то данные могут утечь. Если хранить по отдельности, то очень долго. Поэтому в Интернете используется PEAP-key.

Данный метод использует концепецию ассиметричного шифрования (есть пара (e,d) - ключ шифрования и дешифрования). Процесс согласования: 2) скажи свой ключ дешифрования, кто-то производит дешифрование. Но человек посередине генерирует сессию и подстраивается. Поэтому можно подписать данные еще одним закрытым ключом (e', d') (sertification authority), e' хранится в жесточайшем секрете, а d' можно разложить по всем участникам сети. Для нового участника тихо генерируем данные (e, d), при этом данный d подписывается e' в удостоверяющем центре (подпись — $E(d, e')$). В итоге добавление и утечка пользователей происходит нормально. Но в крупных системах возникает проблема раздачи таких центров. Решение 1 — хранить данные об удостоверяющих центрах - плохо, большая система, много расходов. Решение 2 — древовидная структура удостоверяющих центров (каждый из которых реализует свой центр)

Пример. HTTPS-протокол, можно просмотреть список всех путей сертификатов.

Наличие подписанного сертификата мало о чем говорит (можно доверять только отец-сын). Поэтому неудивительно, с учетом того, что одни службы гоняются за другими, и АНБ может выписать специальный хитрый сертификат, если необходимо будет поймать вас (ситуация с Gmail и Ближним Востоком, отправлялась отдельно цепочка в Google).

Проблема доверия имеет место налицо. Сайт банка, с зеленым замком в вашем браузере, может быть не тем, кем представляется (просто за сайт кто-то доверился, но при этом, возможно, является мошенническим лицом).

Поэтому ничему нельзя верить, нельзя подписывать все публичным ключом — это очень опасно. Надо понимать, от чего происходит защита. Альтернативный способ — установление сети доверия (нет дерева с самым доверенным узлом, а есть к примеру 4 узла, и собирается достаточное число подписей, но такой цепочки может и не быть). В этом месте нет способа, не делая более продвинутой работы.