

# 12. Параллельная обработка

Сухорослов Олег Викторович

27.11.2023

# План лекции

- Параллельные вычисления
- Параллельная обработка запросов
- Параллельная обработка данных

# Параллельные вычисления

# Параллельные вычисления: Зачем?

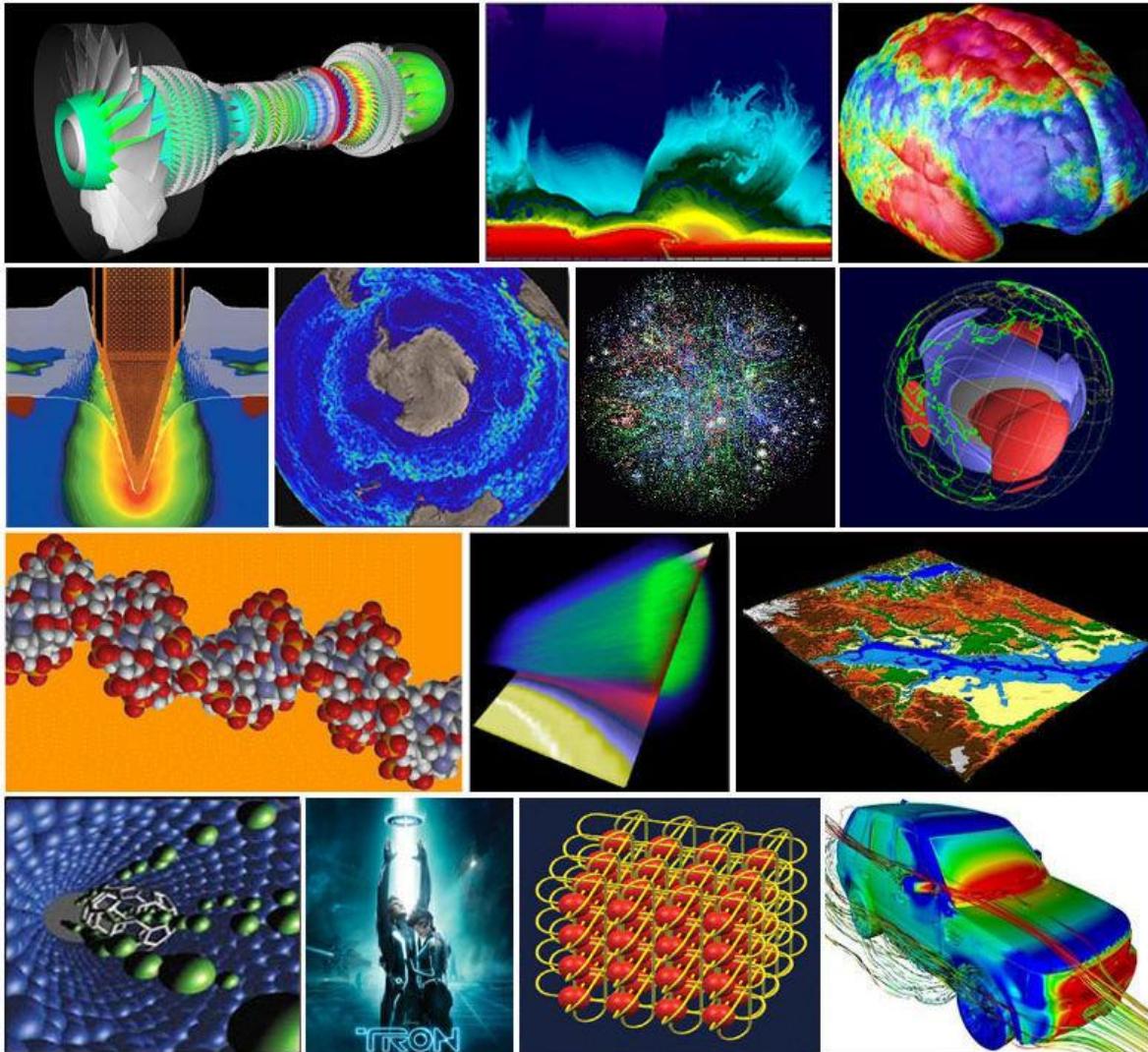
Использование нескольких процессоров для

- решения задачи за меньшее время
- решения больших задач, чем на одном процессоре

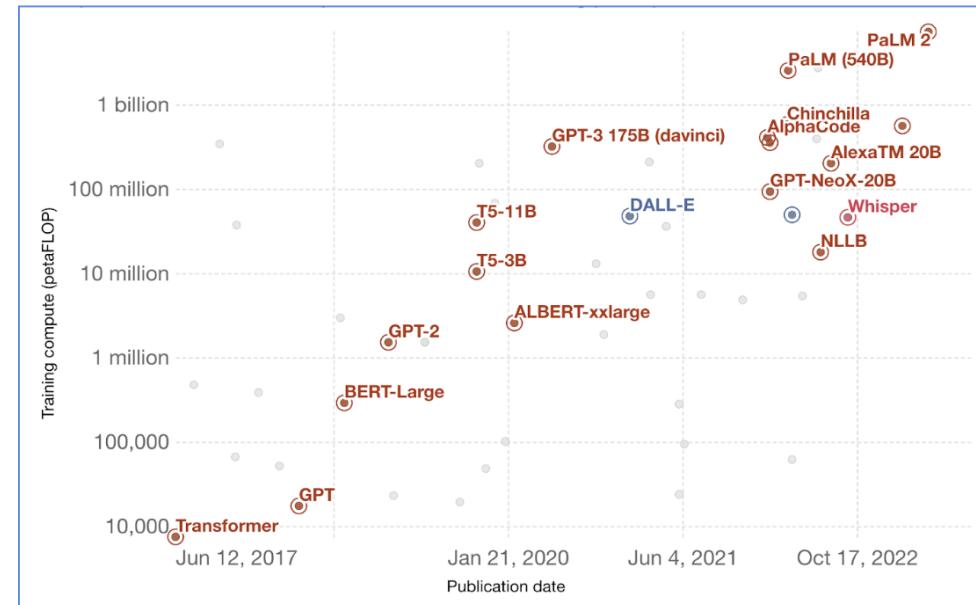
# Специфика параллельных вычислений



# Примеры задач



LLM model size growth



# Параллельные вычисления: Как?

- Создание параллельного алгоритма
  - Поиск параллелизма в последовательном алгоритме, модификация или создание нового алгоритма
  - Декомпозиция исходной задачи на подзадачи, которые могут выполняться одновременно
  - Анализ зависимостей между подзадачами
- Реализация параллельного алгоритма
  - Распределение подзадач между исполнителями (процессорами, узлами...)
  - Организация взаимодействия между подзадачами
  - Учет архитектуры целевой параллельной системы
  - Запуск, измерение и анализ показателей эффективности

# Классы параллельных систем

- Single Instruction, Multiple Data (SIMD)
  - SIMD-инструкции CPU
  - Графические процессоры (GPU)
- Multiple Instruction, Multiple Data (MIMD)
  - Системы с общей памятью
  - Системы с распределенной памятью
  - Гибридные системы
- Современные системы сочетают в себе оба класса

# Массивно-параллельная система (МПР)



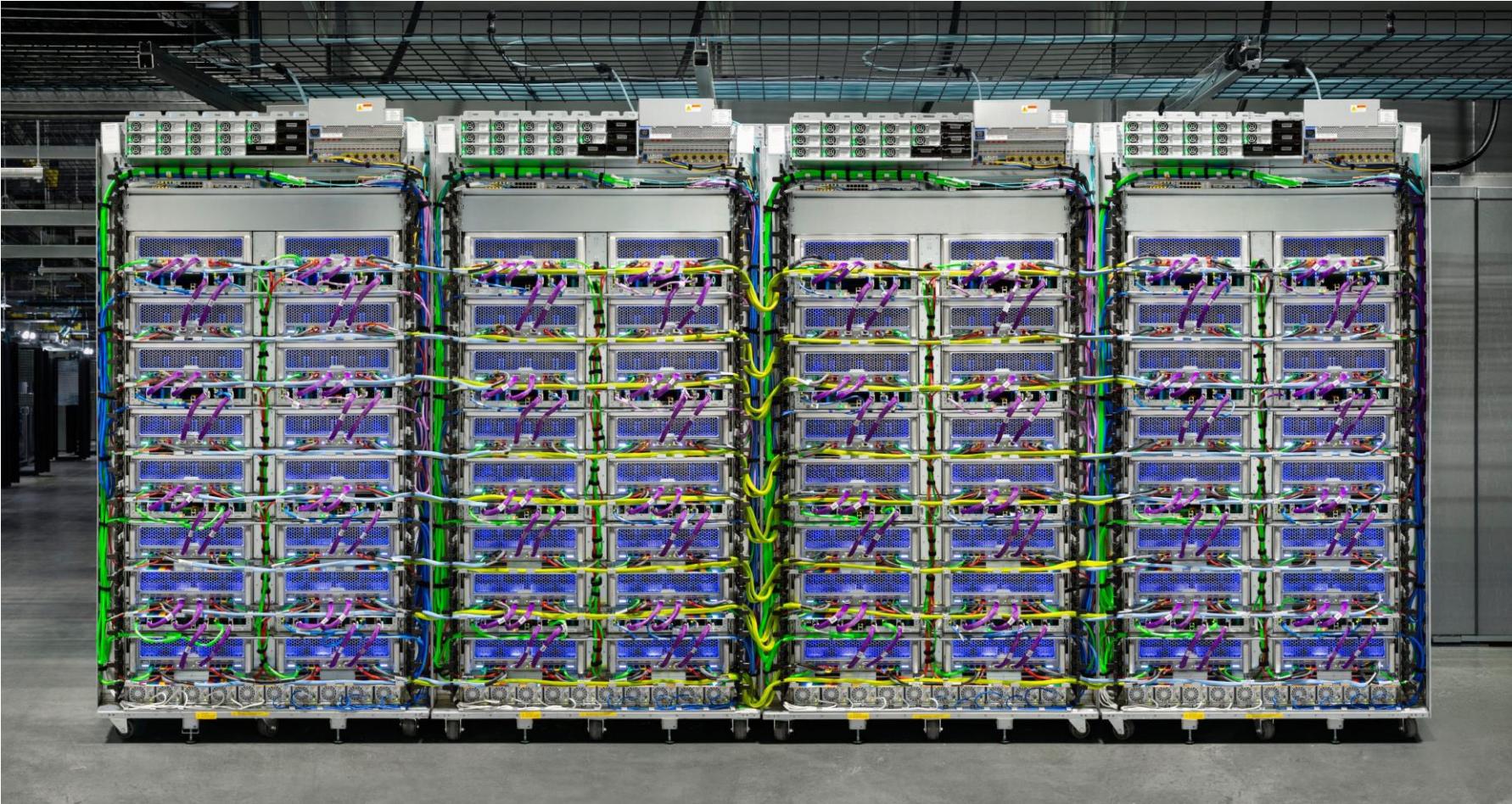
# Beowulf Cluster (1990-e)



# Современный НРС-кластер



# Специализированные системы



<https://www.semianalysis.com/p/tpuv5e-the-new-benchmark-in-cost>

# Показатели эффективности

- Ускорение (speedup)

$$S_p(n) = \frac{T_1(n)}{T_p(n)}$$

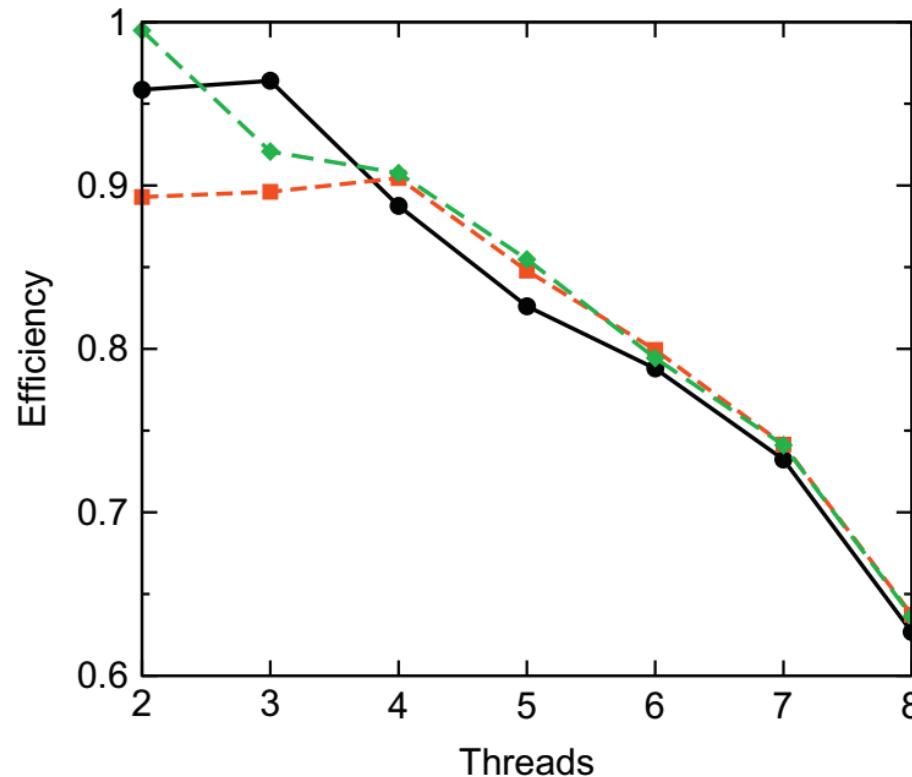
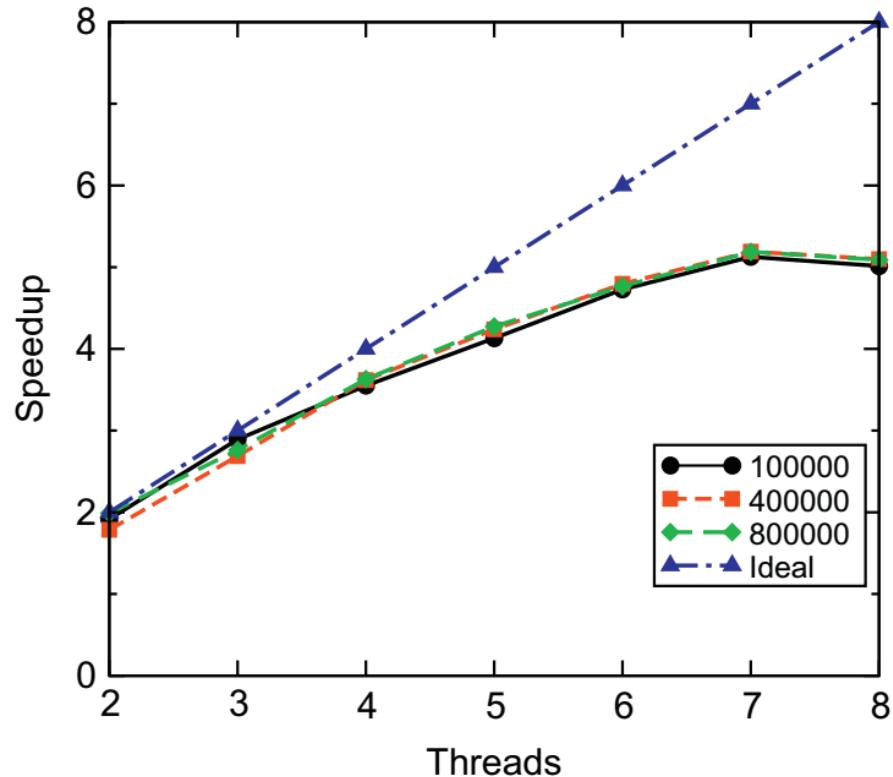
- Эффективность (efficiency)

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T_1(n)}{p T_p(n)}$$

# Ускорение

- $S = p$ 
  - Идеальный случай
- $S < p$ 
  - Последовательные (нераспараллеленные) части алгоритма
  - Накладные расходы:  $T_p = \frac{T_1}{p} + T_{overhead}$
- $S > p$ 
  - Сверхлинейное ускорение
  - Рабочие данные помещаются в кэше, параллельный поиск

# Типичные кривые



# Закон Амдала

- Доля последовательных вычислений:

$$f = \frac{T_{seq}}{T_1}$$

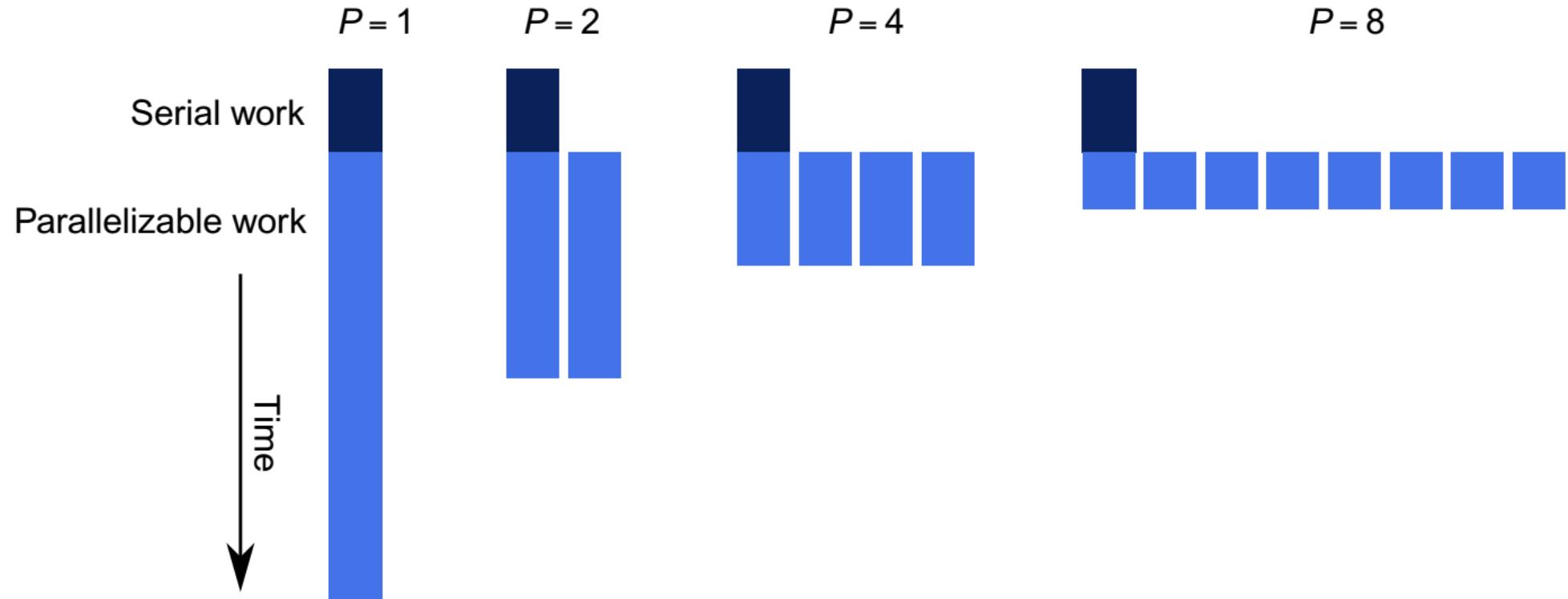
- Время выполнения параллельной реализации:

$$T_p = fT_1 + \frac{(1 - f)T_1}{p}$$

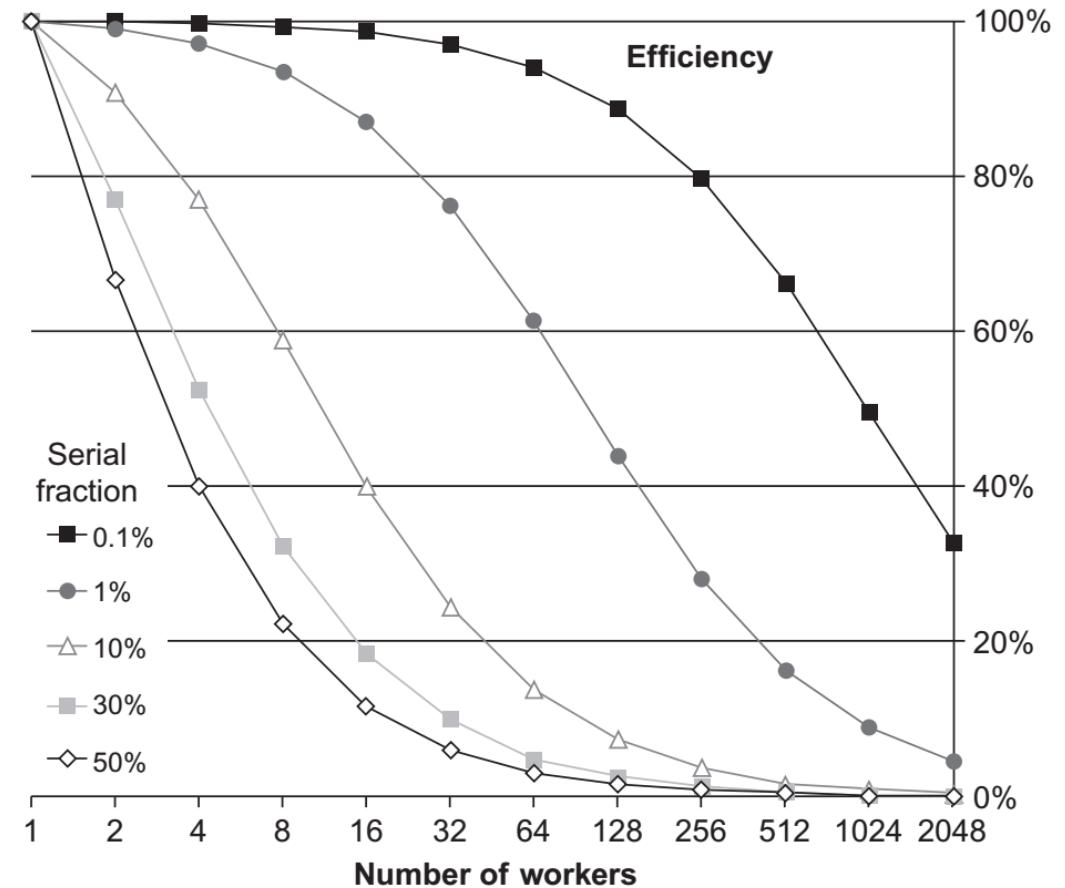
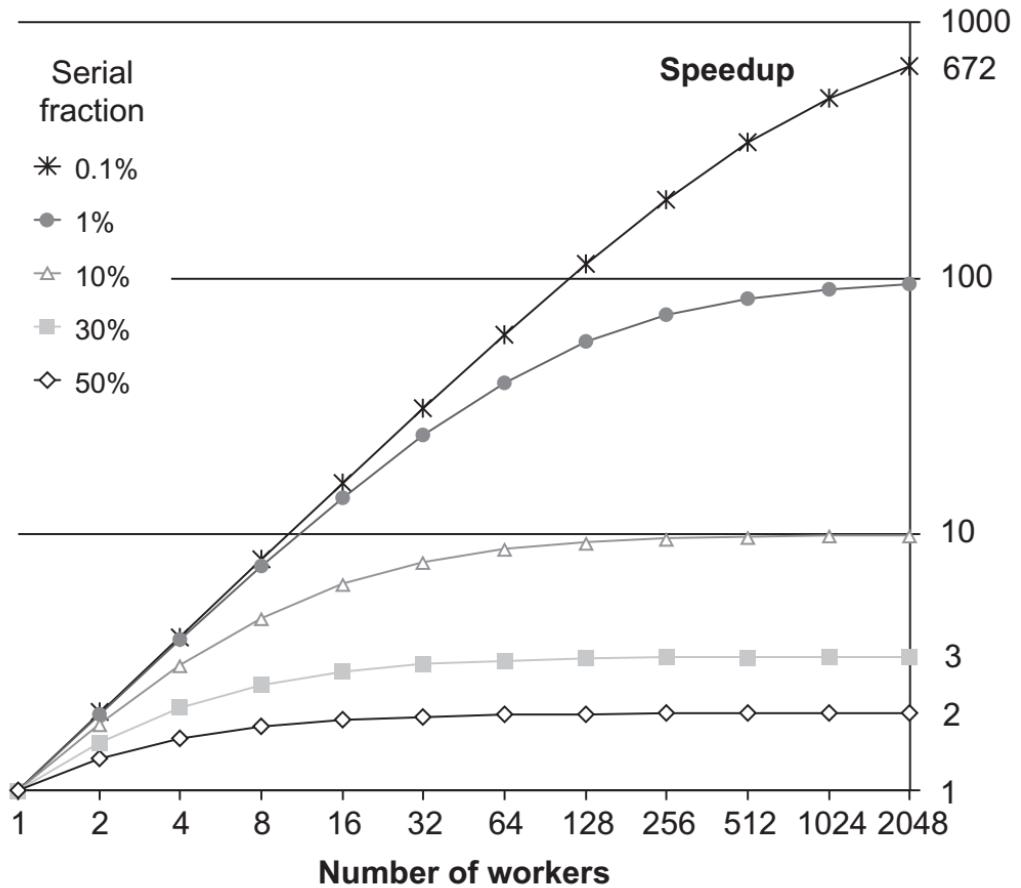
- Максимальное достижимое ускорение:

$$S_p = \frac{T_1}{T_p} = \frac{1}{f + \frac{1-f}{p}}, \quad \lim_{p \rightarrow \infty} S_p = \frac{1}{f}$$

# Закон Амдала



# Закон Амдала



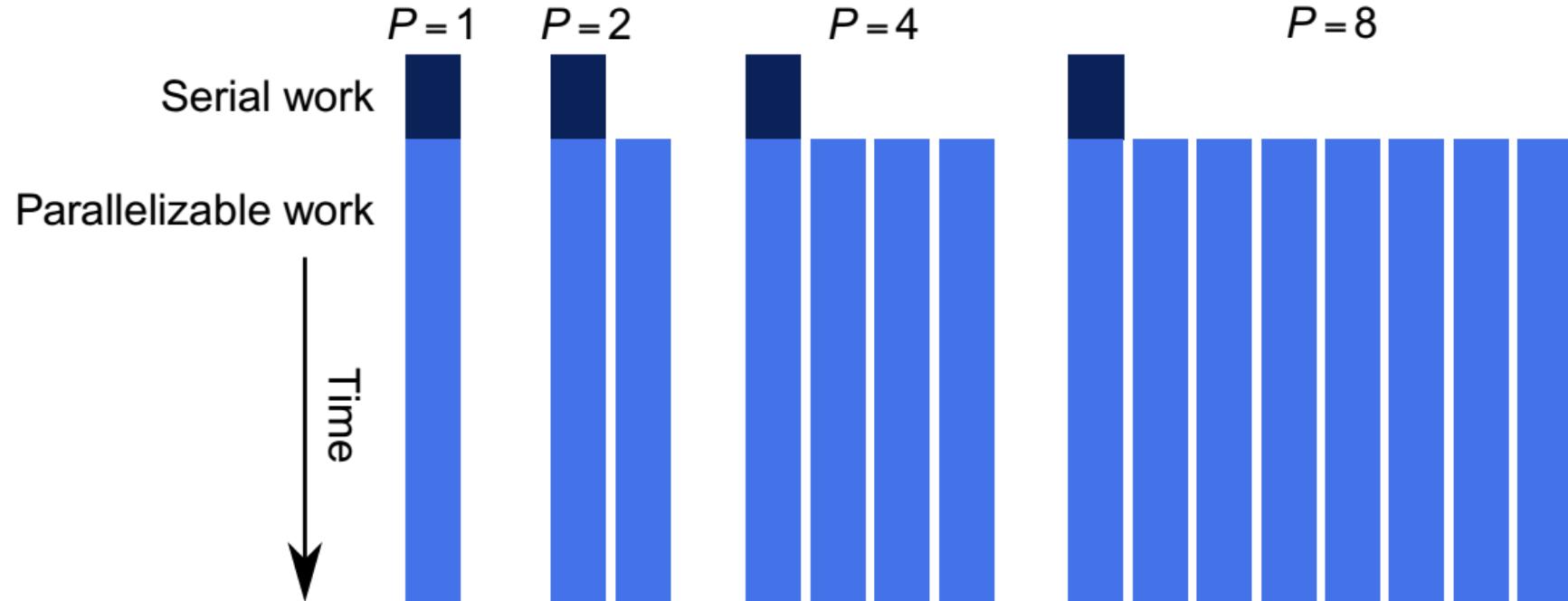
# Последовательные части

- Присутствуют изначально
  - Разные активности, которые нельзя распараллелить
  - Инициализация и завершение работы
  - Чтение входных данных и запись результатов
- Появляются в результате распараллеливания
  - Запуск исполнителей (потоков, процессов...)
  - Создание и распределение подзадач
  - Координация и синхронизация между исполнителями
  - Сбор и объединение результатов

# Как быть?

- Закон Амдала: размер задачи зафиксирован, хотим как можно больше уменьшить время решения
- На практике при увеличении числа процессоров имеет смысл **увеличивать размер задачи**, оставляя время решения таким же
- Эффект Амдала: доля последовательных вычислений уменьшается при увеличении размера задачи
  - Умножение матриц: ввод/вывод  $\sim n^2$ , вычисления  $\sim n^3$

# Закон Густафсона-Барсиса



Gustafson J. [Reevaluating Amdahl's Law](#) (1988)

# Масштабируемость

- Параллельная программа является **масштабируемой**, если при увеличении числа исполнителей можно сохранять значение эффективности
- **Сильная масштабируемость (strong scaling)** – эффективность сохраняется без необходимости увеличения размера задачи
- **Слабая масштабируемость (weak scaling)** – для поддержания заданной эффективности требуется увеличивать размер задачи пропорционально числу исполнителей

# Параллельная обработка запросов

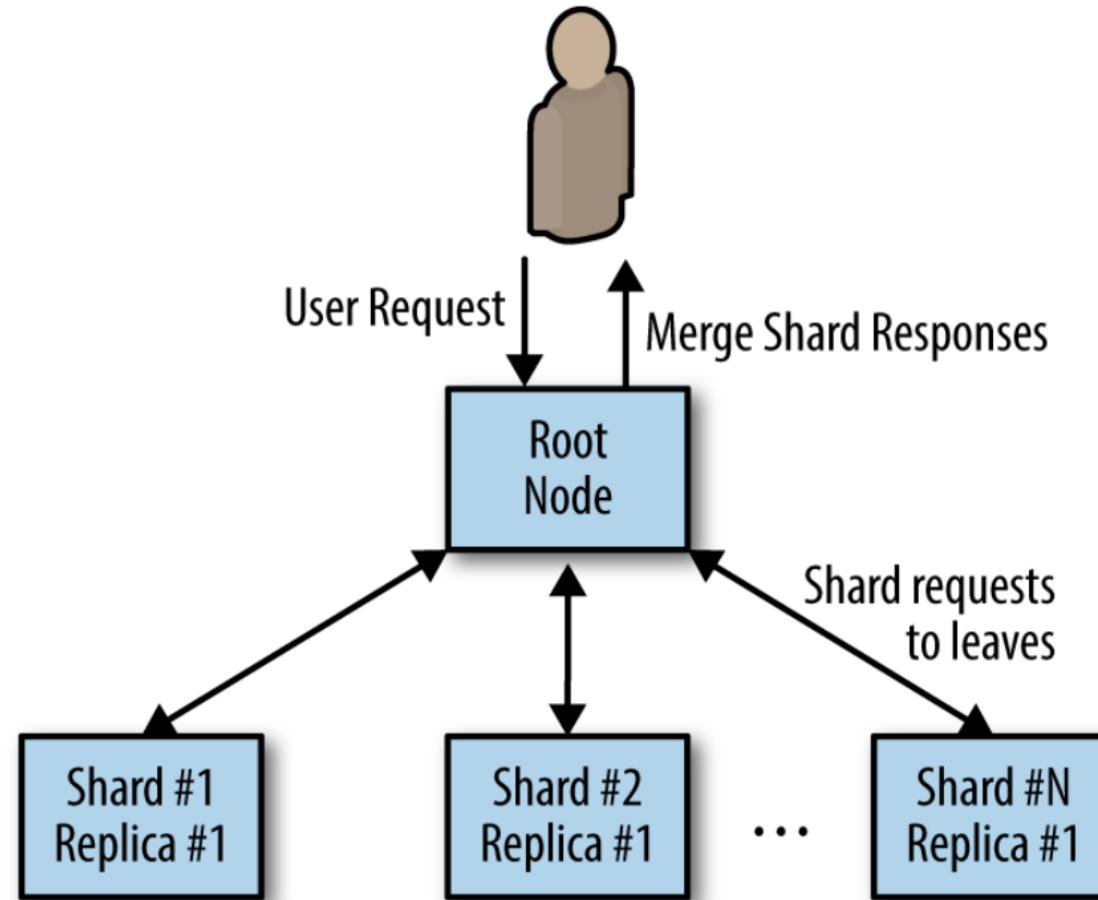
# Техники горизонтального масштабирования

- Репликация stateless-сервиса + балансировка нагрузки
- Кэширование
- Разбиение stateful-сервиса по данным (шардинг)
- Параллельная обработка

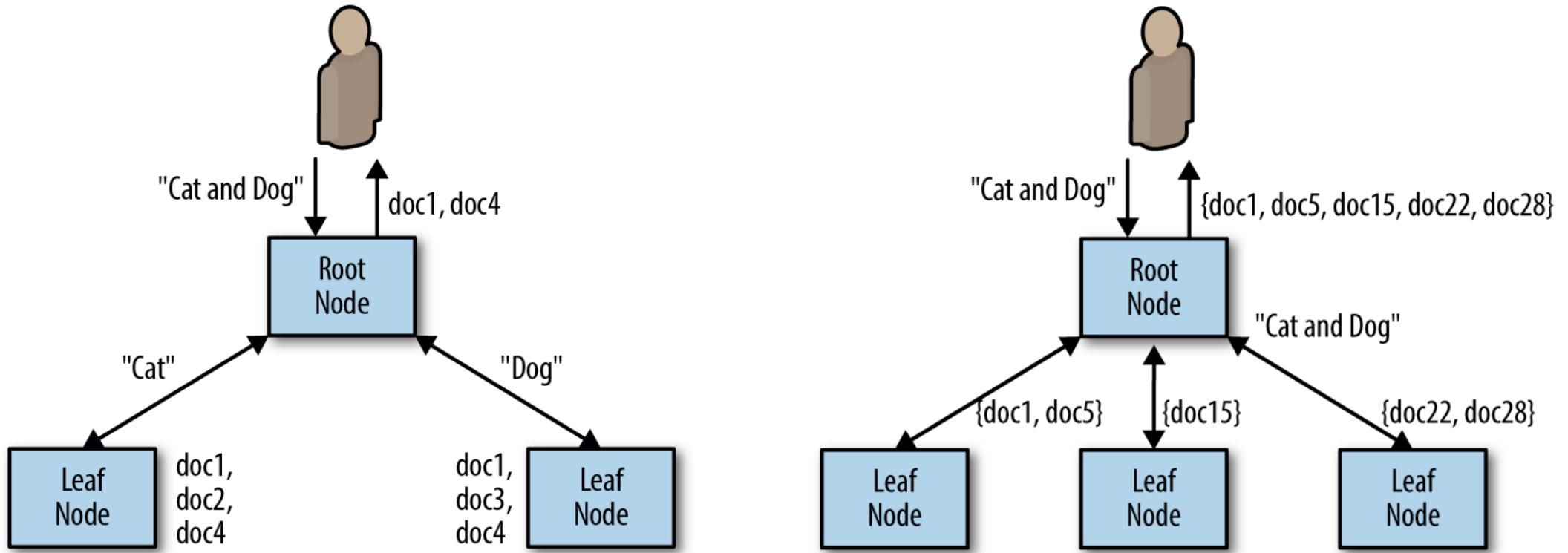
# Масштабирование тяжелых запросов

- Запрос требует проведения вычислений или чтения данных
- Объемы требуемых вычислений и данных растут
- Как обеспечить масштабируемость?

# Параллельная обработка (Scatter/Gather)



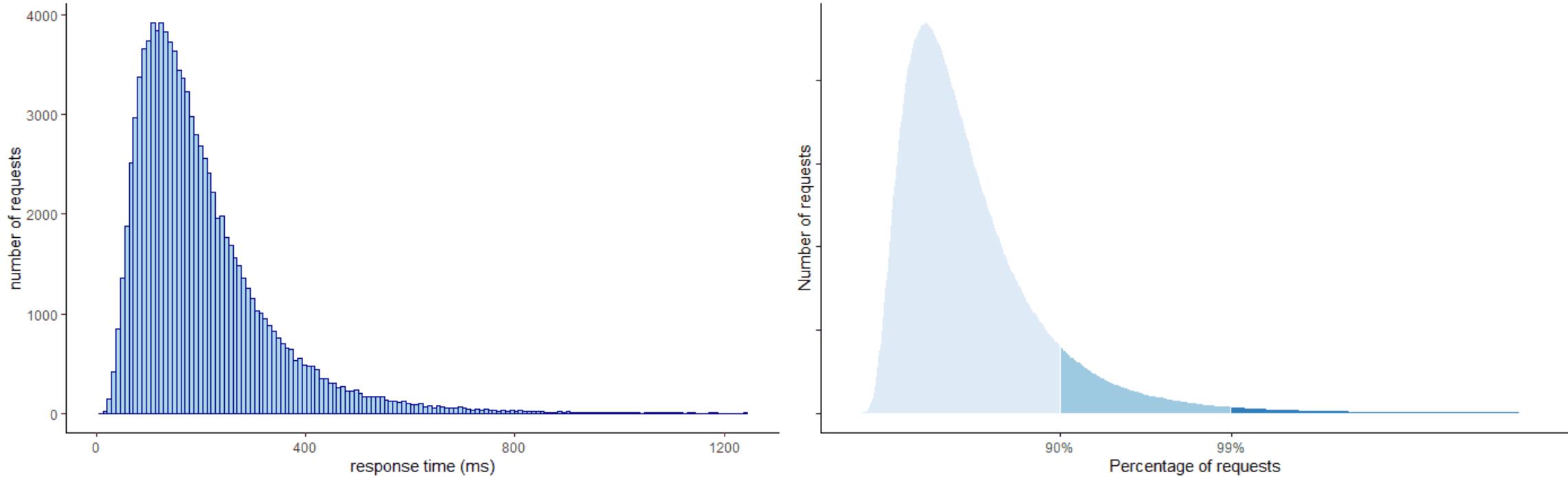
# Распределенный поиск



# Выбор числа серверов?

- См. закон Амдала
- Фиксированные накладные расходы на обработку подзапросов
- Дополнительные накладные расходы на распаралеливание

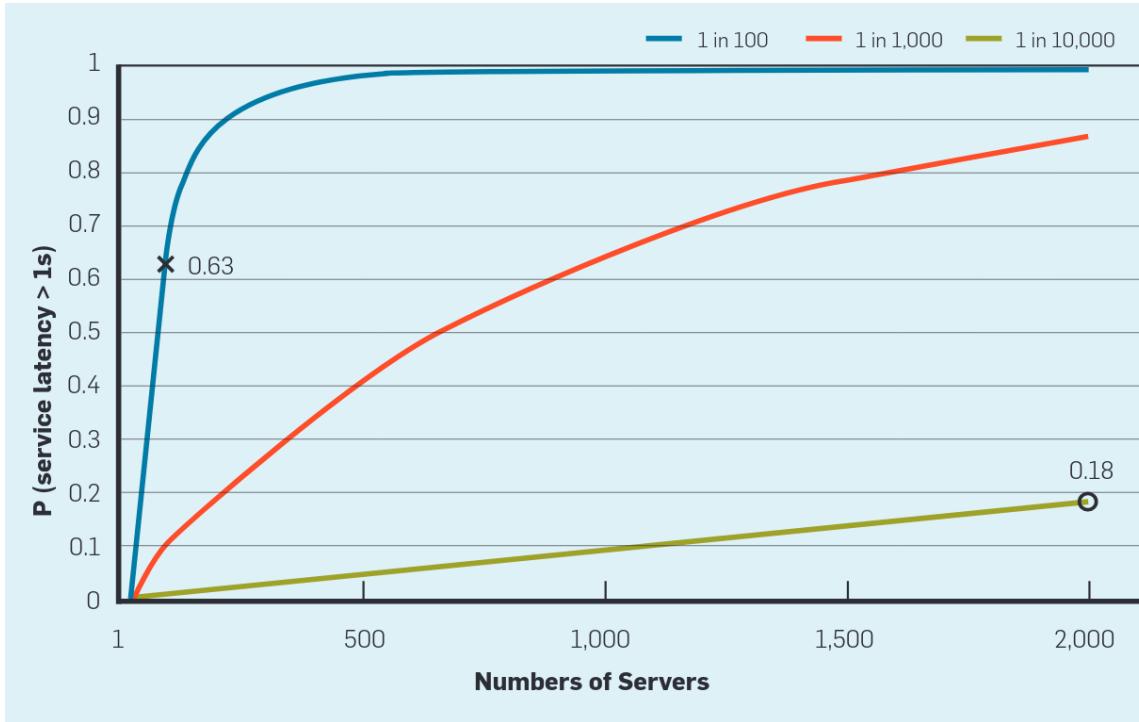
# Tail Latency



<https://robertovitillo.com/why-you-should-measure-tail-latencies/>

# Tail Latency Amplification

- Время обработки запроса определяется самым медленным сервером
- С ростом числа серверов риск задержки увеличивается



	50%ile latency	95%ile latency	99%ile latency
One random leaf finishes	1ms	5ms	10ms
95% of all leaf requests finish	12ms	32ms	70ms
100% of all leaf requests finish	40ms	87ms	140ms

Dean J., Barroso L.A. [The Tail at Scale](#) (2013)

# Устойчивость к росту задержки

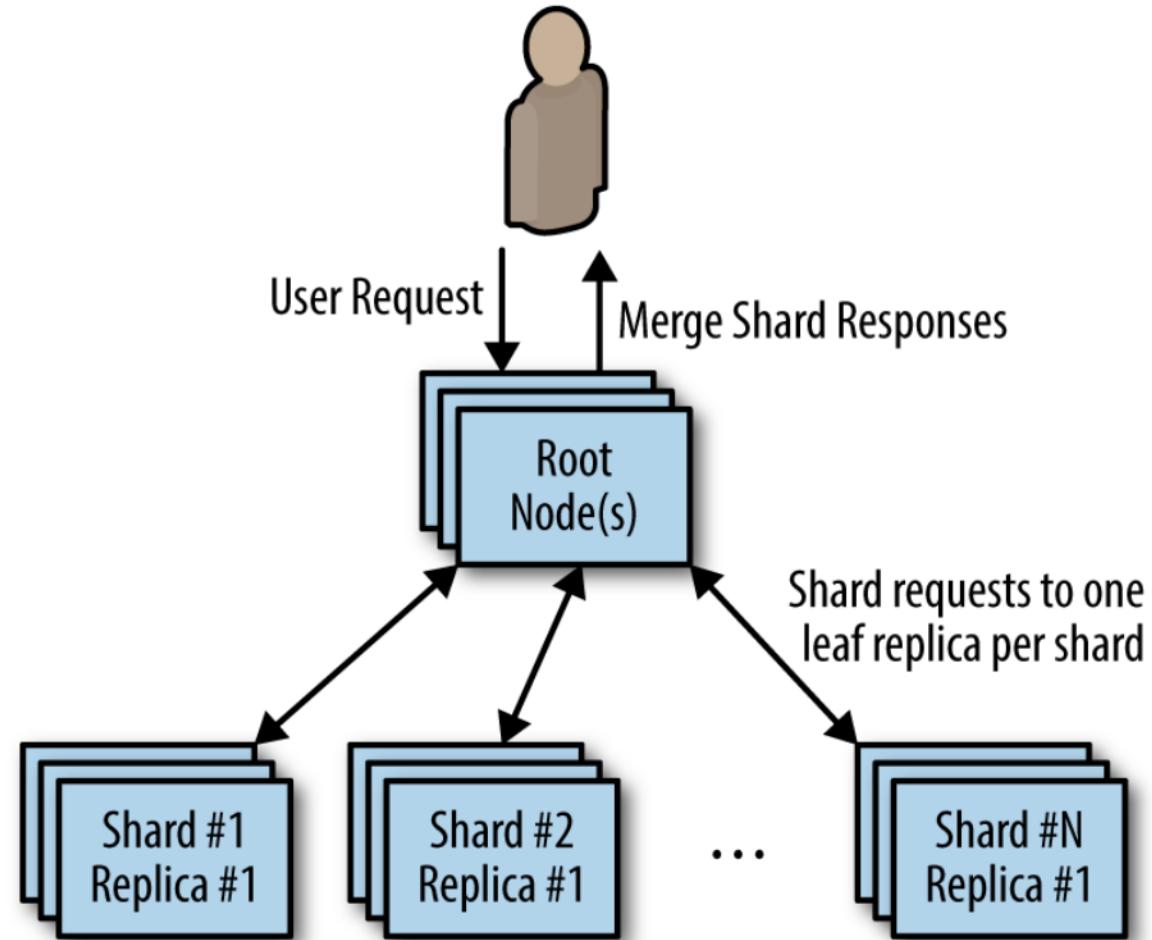
Fault-tolerant techniques were developed because guaranteeing fault-free operation became infeasible beyond certain levels of system complexity. Similarly, **tail-tolerant techniques** are being developed for large-scale services because eliminating all sources of variability is also infeasible.

Dean J., Barroso L.A. [The Tail at Scale](#) (2013)

# Стратегии борьбы с ростом задержки

- На уровне отдельных запросов
  - Ждать ответы на подзапросы до таймаута и выдавать возможно неполный ответ
  - Отправка дублирующего подзапроса после таймаута (hedged requests)
  - Отправка дублирующих подзапросов с быстрой отменой (tied requests)
- На уровне системы в целом
  - Разбиение данных на мелкие порции и их динамическое назначение на машины
  - Дополнительная репликация горячих порций данных
  - Временное исключение медленных машин из обработки запросов

# Распределенный поиск (+репликация)



# Параллельная обработка данных на кластере

# Проблемы

- Требуется хранить и обрабатывать все больше данных
- Современные задачи намного превышают возможности одной машины
- Данные нельзя разместить полностью в памяти
- Данные хранятся и обрабатываются в отдельных системах
- Отказы в больших распределенных системах становятся нормой
- Реализовывать обработку данных в таких системах очень сложно
  - Требуются удобные высокоуровневые модели программирования
  - Требуются универсальные и масштабируемые среды выполнения

# Typical New Engineer



- Never seen a petabyte of data
- Never used a thousand machines
- Never **really** experienced machine failure

*Our software has to make them successful.*

Google™

# Web Search

- Сбор содержимого Web (crawling)
  - offline, загрузка большого объема данных, выборочное обновление, обнаружение дубликатов
- Построение инвертированного индекса (indexing)
  - offline, пакетные (batch) задания, периодическое обновление
  - обработка большого объема данных, предсказуемая нагрузка
- Ранжирование документов для ответа на запрос (retrieval)
  - online, интерактивные запросы, задержка ~10-100 миллисекунд
  - большое число клиентов, пики нагрузки

# Построение индекса

- Исходные данные и требуемый результат

$[(\text{id}, \text{content})\dots] \rightarrow [(\text{term}, [\langle \text{id}, \text{tf} \rangle \dots])\dots]$

- Извлечение слов из каждого документа

$(\text{id}, \text{content}) \rightarrow [(\text{term1}, \langle \text{id}, \text{tf1} \rangle), (\text{term2}, \langle \text{id}, \text{tf2} \rangle)\dots]$

- Группировка промежуточных данных для каждого слова

$[(\text{term1}, \langle \text{id1}, \text{tf1} \rangle), (\text{term1}, \langle \text{id2}, \text{tf2} \rangle)\dots] \rightarrow (\text{term1}, [\langle \text{id1}, \text{tf1} \rangle, \langle \text{id2}, \text{tf2} \rangle\dots])$

- Агрегация результатов для каждого слова

$(\text{term}, [\langle \text{id1}, \text{tf1} \rangle, \langle \text{id2}, \text{tf2} \rangle\dots]) \rightarrow (\text{term}, \text{top\_documents\_for\_term})$

# Модель программирования MapReduce

- Базовой структурой данных являются пары (*ключ, значение*)
- Программа описывается путем определения функций

$$map: (k1, v1) \rightarrow [(k2, v2)]$$
$$reduce: (k2, [v2]) \rightarrow [(k3, v3)]$$

- На практике чаще всего

$$reduce: (k2, [v2]) \rightarrow (k2, v3)$$

# Вычисление частот встречаемости слов

```
1: class MAPPER
2:   method MAP(docid a, doc d)
3:     for all term t  $\in$  doc d do
4:       EMIT(term t, count 1)

1: class REDUCER
2:   method REDUCE(term t, counts [c1, c2, ...])
3:     sum  $\leftarrow$  0
4:     for all count c  $\in$  counts [c1, c2, ...] do
5:       sum  $\leftarrow$  sum + c
6:     EMIT(term t, count sum)
```

# Другие примеры

- Поиск в тексте (grep)

```
map: (docid, content) → [(docid, line)]
reduce: нет
```

- Группировка и сортировка по ключу

```
map: (key, record) → (key, record)
reduce: (key, [record]) → (key, [record])
```

- Анализ посещаемости сайта

```
map: (logid, log) → [(url, visit_count)]
reduce: (url, [visit_count]) → (url, total_count)
```

# Другие примеры (2)

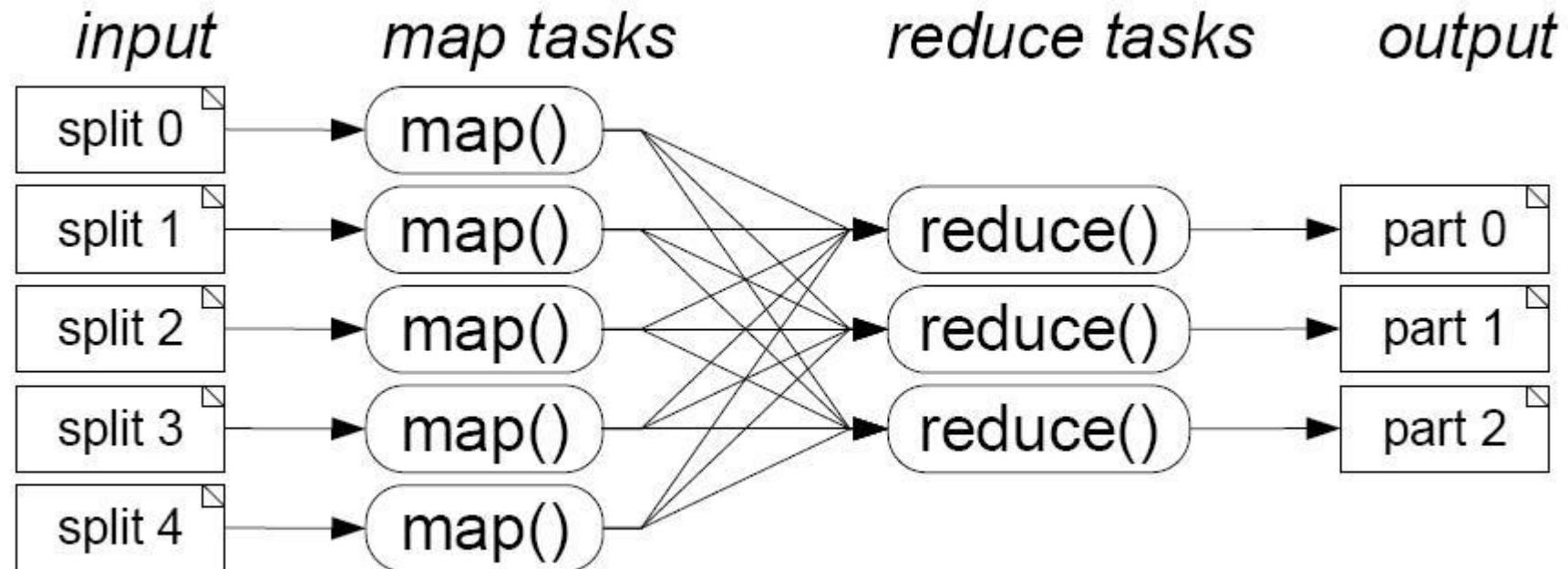
- Вычисление ключевых слов по сайтам

```
map: (docid, <url, content>) → (hostname, doc_term_vec)
reduce: (hostname, [doc_term_vec]) → (hostname, host_term_vec)
```

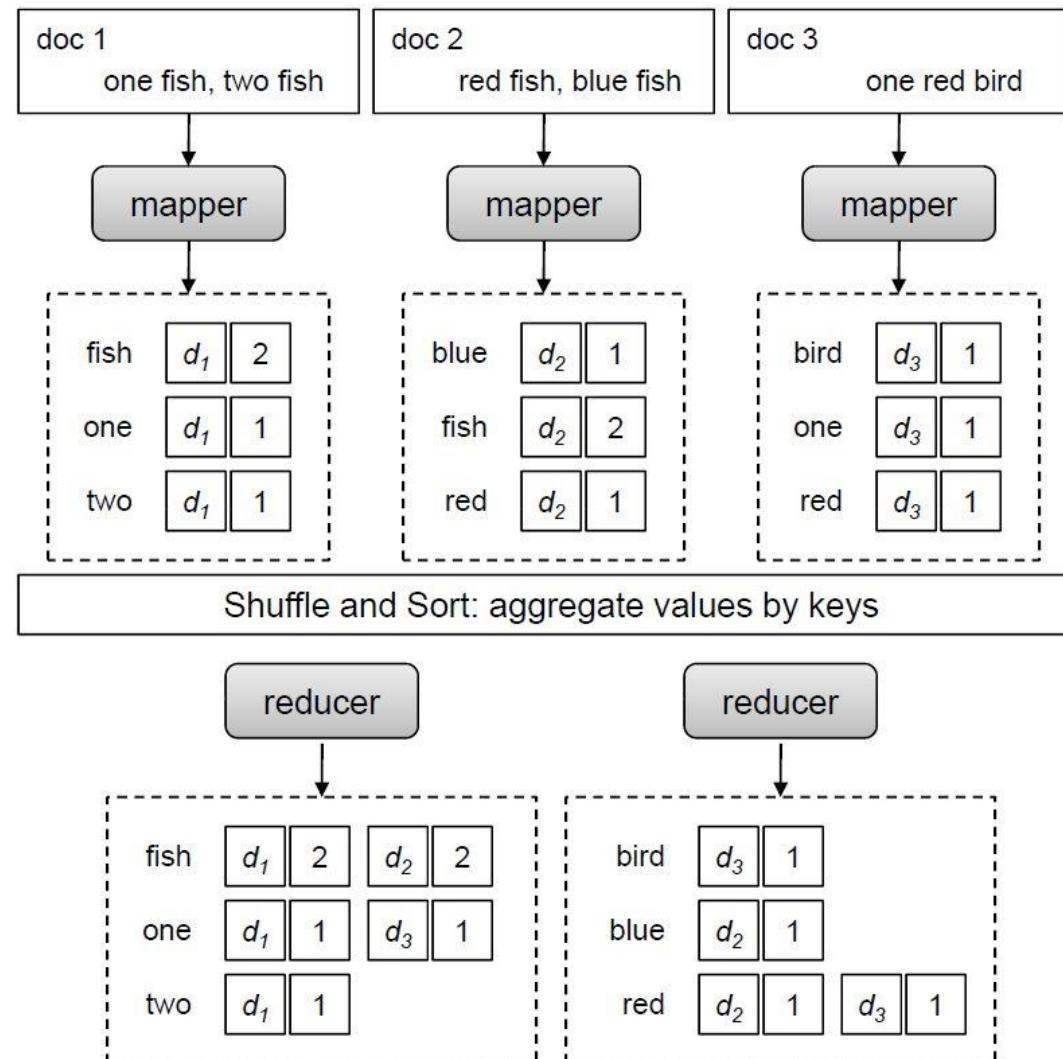
- Обращение Web-графа (кто ссылается на страницу?)

```
map: (docid, content) → [(url, docid)]
reduce: (url, [docid]) → (url, [docid])
```

# Параллелизм по данным



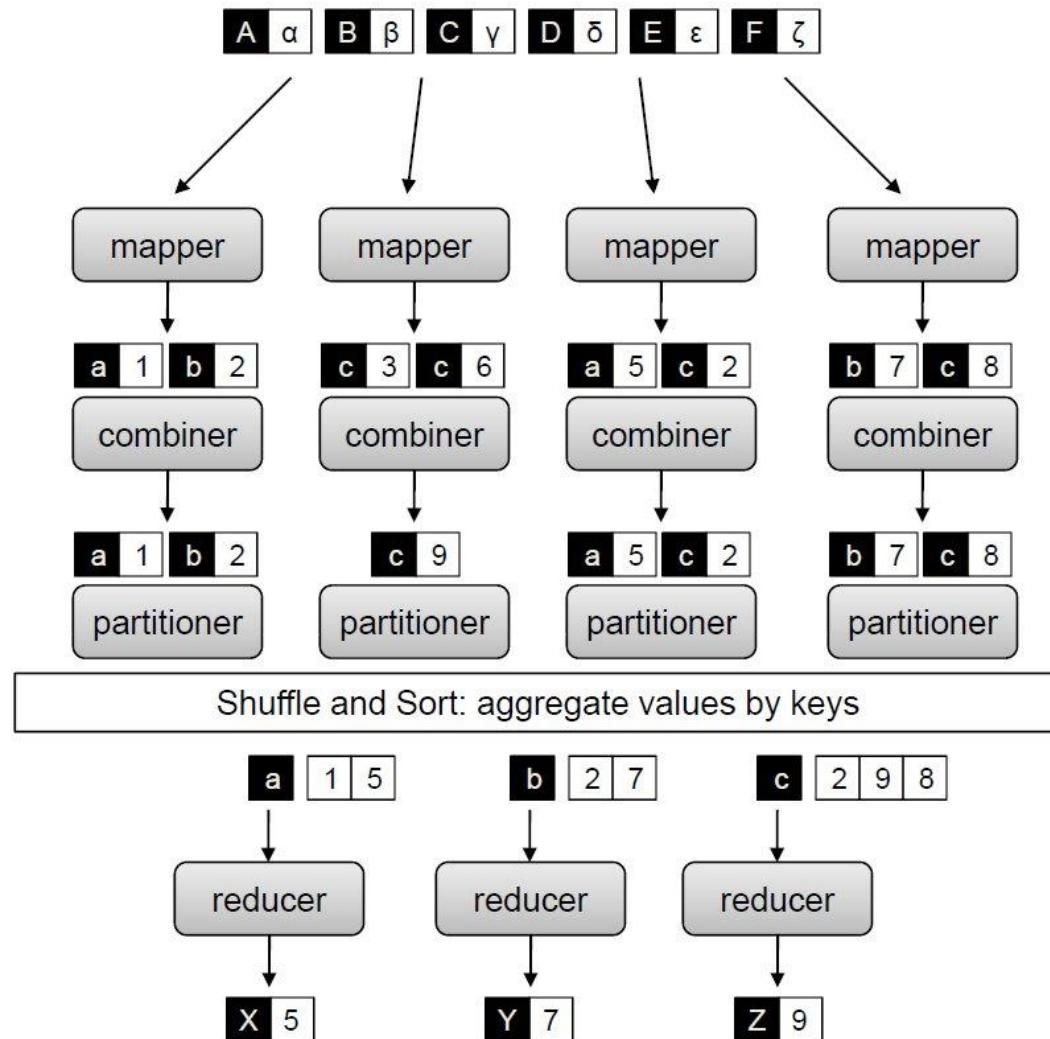
# Распараллеливание построения индекса



# Дополнительные функции

- $\text{partition}: (k2, \text{num\_reducers}) \rightarrow \text{reducer\_id}$ 
  - Определяет распределение промежуточных данных между reduce-процессами
  - Пример:  $\text{hash}(k2) \bmod \text{num\_reducers}$
- $\text{combine}: (k2, [\nu2]) \rightarrow [(k2', \nu2')]$ 
  - Осуществляет локальную агрегацию промежуточных данных после *map* в рамках одного тар-процесса
  - Для ассоциативных и коммутативных операций может использоваться *reduce*
- $\text{compare}: (k2, k2') \rightarrow \{-1, 0, 1\}$ 
  - Определяет отношение порядка между промежуточными ключами
  - Управляет сортировкой и порядком ключей в результирующих данных

# Полная схема вычислений



# MapReduce

- Модель программирования для описания процедур обработки данных
- Среда выполнения (runtime) для параллельной обработки больших объемов данных на кластере
- Программная реализация (Google, Hadoop, Spark, YTsaurus...)

# Google MapReduce

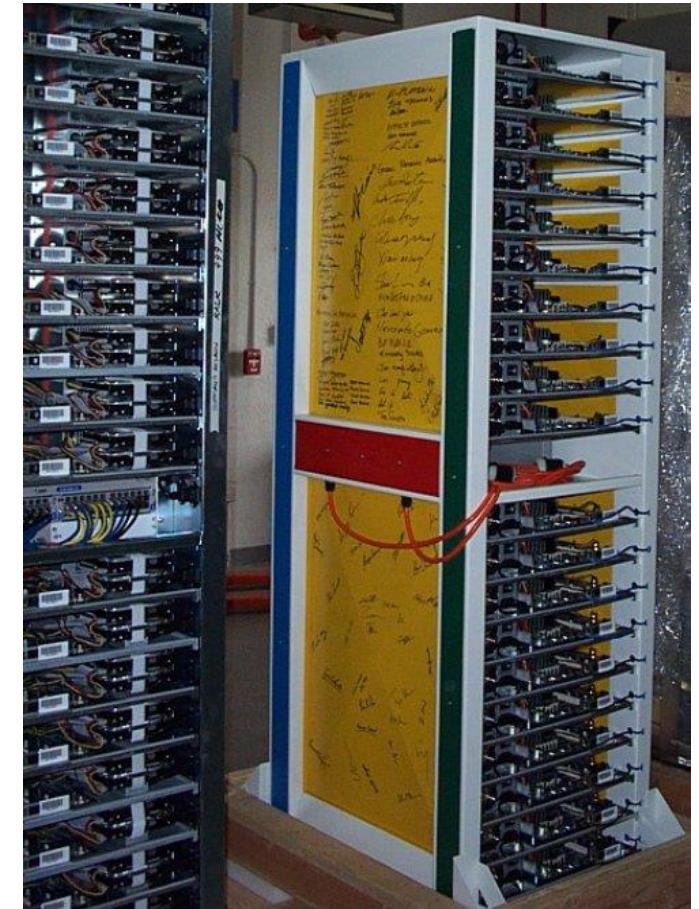
- MapReduce: Simplified Data Processing on Large Clusters  
(Jeffrey Dean, Sanjay Ghemawat)
  - 2004: [оригинальная статья на OSDI'04](#)
  - 2008: [статья в Communications of the ACM](#)



<https://habr.com/ru/articles/432324/>

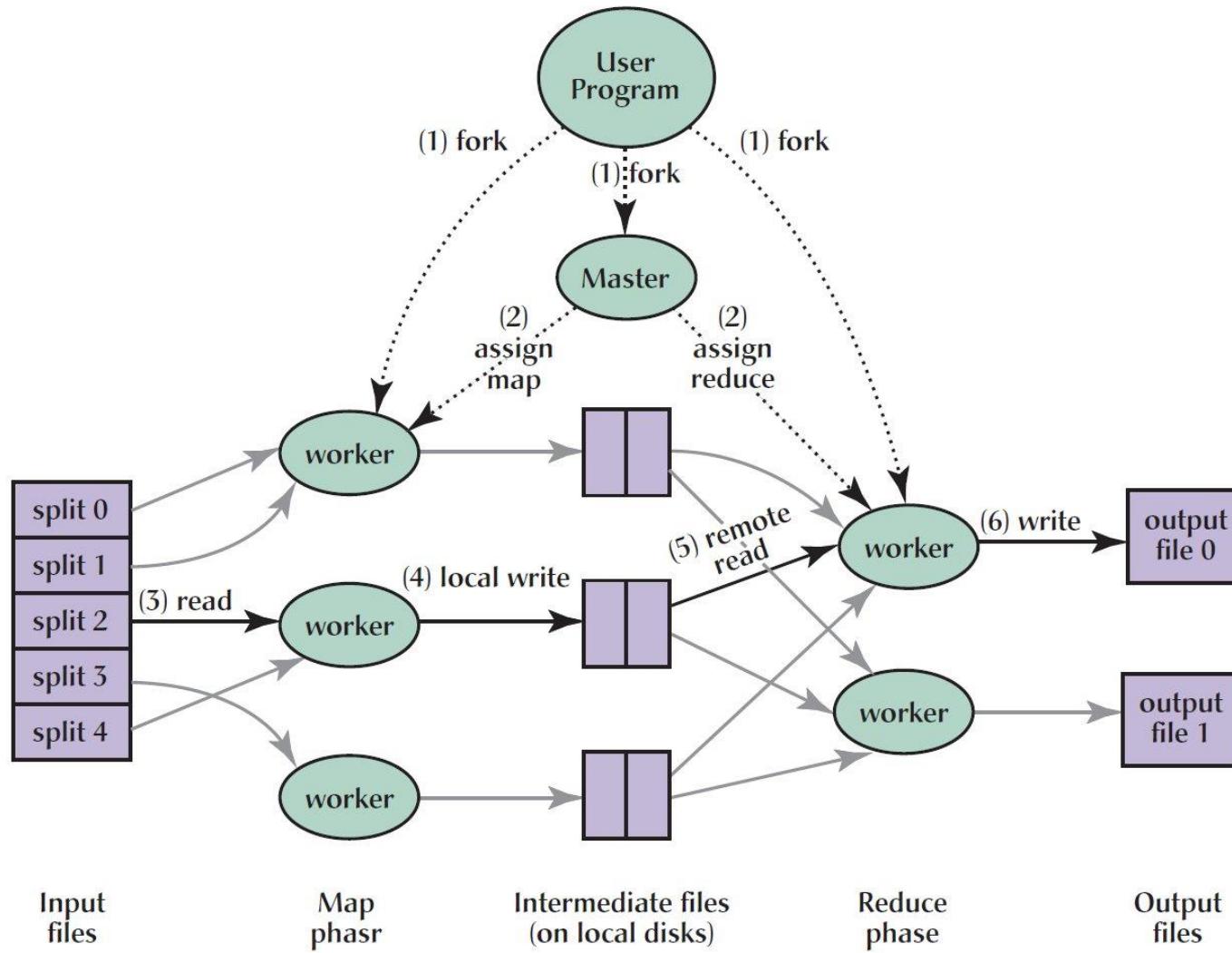
# Инфраструктура Google (2000-е)

- Кластеры из бюджетных серверов
  - PC-class motherboards, low-end storage/networking
  - Linux + своё ПО
  - Сотни тысяч машин, отказы являются нормой
- Распределенная файловая система GFS
  - Поблочное хранение больших файлов, write-once-read-many
  - Последовательные чтение и запись в потоковом режиме
  - Репликация блоков для отказоустойчивости
- Для хранения и обработки используются одни серверы
  - Перемещение вычислений дешевле перемещения данных
- Планировщик
  - Распределяет ресурсы кластера между приложениями



[Handling Large Datasets at Google: Current Systems and Future Directions](#) (Jeff Dean, 2008)

# Google MapReduce



# Мастер

- Управляет выполнением одного MapReduce-задания
- Запускает рабочие процессы на узлах кластера
- Распределяет map/reduce-задачи между рабочими процессами
- Хранит состояния всех задач (status, worker)
- Осуществляет координацию между map- и reduce-задачами
  - Получает информацию о файлах с промежуточными данными от map-задач
  - Передает эту информацию reduce-задачам
- Предоставляет информацию о статусе вычислений через HTTP-сервер

# Оптимизации

- Локальность данных
  - Направлять тар-задачи на узлы, хранящие требуемые данные или находящиеся рядом
- Локальная редукция данных
  - Выполнять после *tar* функцию *combine*
- Совмещение операций
  - Загрузка и сортировка промежуточных данных
- Спекулятивное выполнение
  - В конце тар- или reduce-фазы запустить оставшиеся задачи на нескольких узлах
  - Способ решения проблемы отстающих (т.н. stragglers)

# Обработка отказов

- Сбой при выполнении задачи
  - Повторные попытки, пропуск плохих записей
- Отказ рабочего узла
  - Сбой аппаратуры, ПО или отзыв узла планировщиком
  - Определяется через периодические сообщения от рабочего (heartbeat)
  - Перезапуск тар-задач: всех (выполненных и незавершенных),  
уведомление reduce-процессов
  - Перезапуск reduce-задач: только незавершенных
- Отказ мастера
  - Аварийное завершение MapReduce-программы

# Семантика выполнения программы

- ...в присутствии отказов и спекулятивного выполнения
- Для детерминированных функций *map* и *reduce* гарантируется совпадение результата вычислений с результатом последовательного выполнения программы
- Для недетерминированных функций *map* и *reduce* гарантируется совпадение результата каждой *reduce*-задачи с результатом последовательного выполнения программы

# Преимущества MapReduce

- Модель программирования
  - Высокий уровень абстракции за счет скрытия деталей организации вычислений
  - Позволяет разработчику сконцентрироваться на решаемой задаче
  - Легкость добавления новых стадий обработки данных
- Реализация
  - Автоматическое распараллеливание и распределение задач
  - Устойчивость к отказам
  - Масштабируемость

# Недостатки MapReduce

- Глобальная синхронизация только между map и reduce
  - Задачи, требующие наличия общего состояния во время вычислений?
- Синхронизация между заданиями через файловую систему
  - Итеративные алгоритмы?
- Пакетная обработка больших порций данных
  - Интерактивный анализ данных?
  - Инкрементальное добавление небольших порций?
  - Online-обработка данных в потоковом режиме?

# Материалы

- [Multicore and GPU Programming: An Integrated Approach](#) (глава 1)
- [Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services](#) (глава 7)
- [The Tail at Scale](#)
- [MapReduce: Simplified Data Processing on Large Clusters](#)
- [Data-Intensive Text Processing with MapReduce](#) (глава 2)
- [Web search for a planet: The Google cluster architecture](#)
- [The Google File System](#)
- [The Datacenter as a Computer](#)