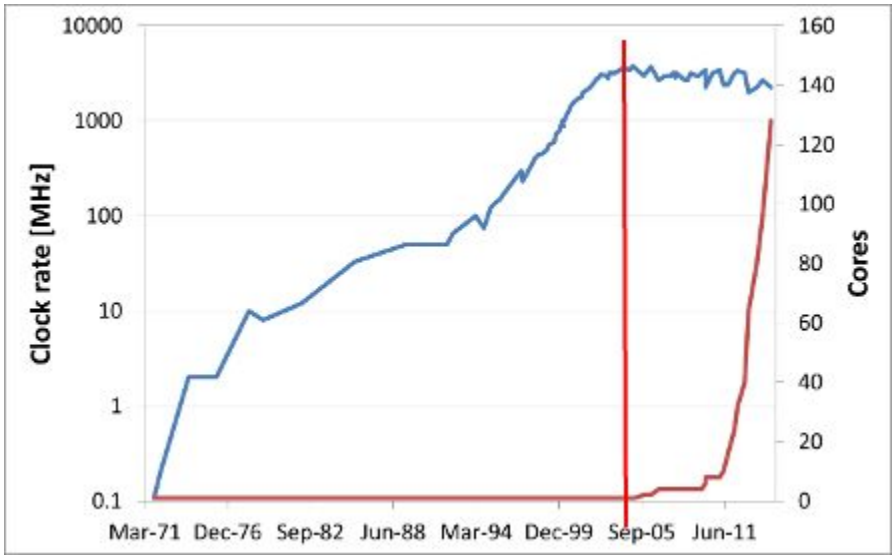
Многопоточная приоритетная очередь

Частично основано на докладе D. Alistarh “Relaxed Data Structures” с SPTDC 2017

Зачем многопоточные структуры?

ИТМО

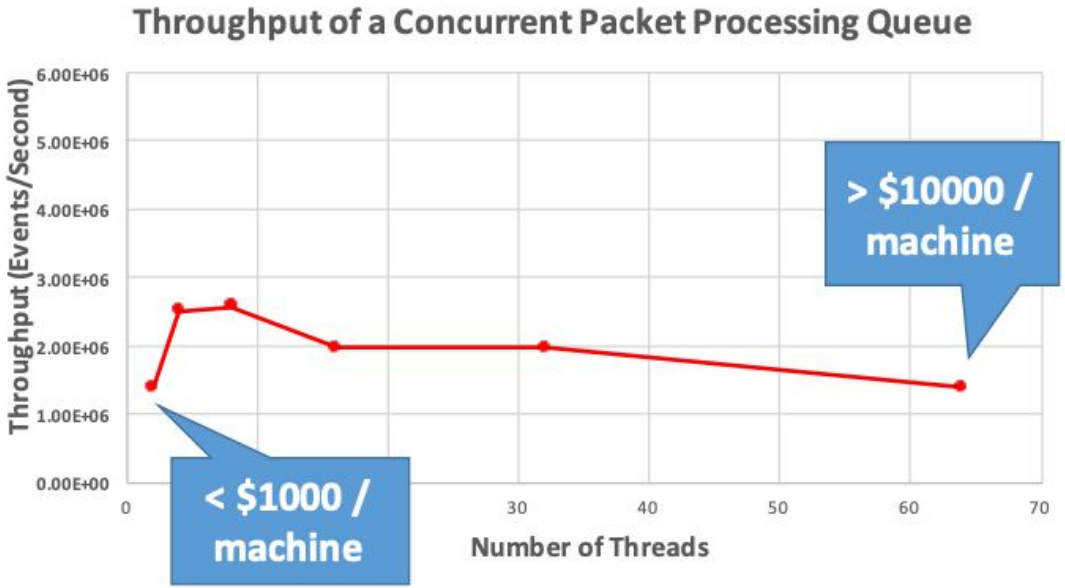
- Частота и производительность ядер перестали сильно увеличиваться
- Поэтому надо уметь использовать несколько ядер
- Важно как устроена синхронизация



Но есть проблемка...

ИТМО

Только ли с очередью проблемы?..



Strong Ordering Semantics

ИТМО

Структуры данных с Strong Ordering Semantics:

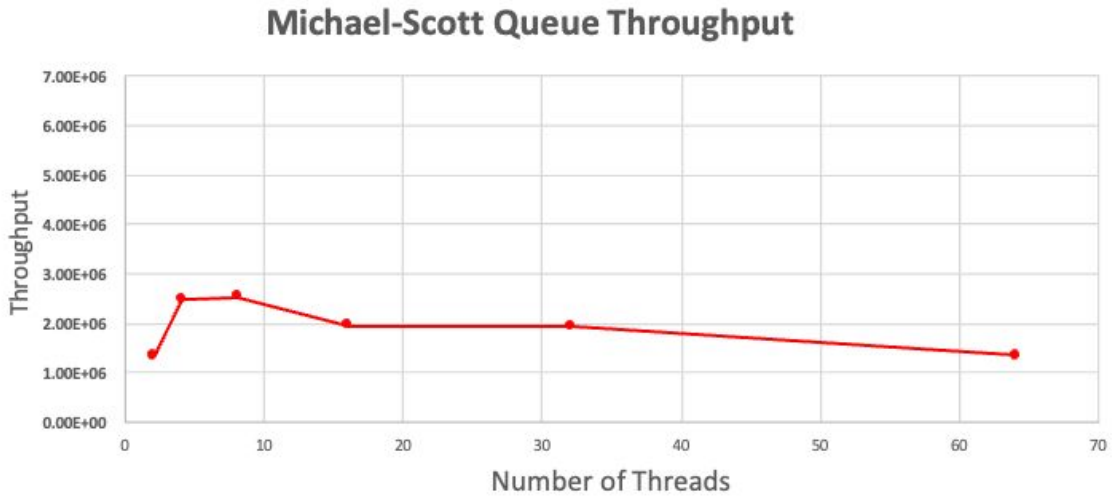
- Стек
- Очередь
- Счётчик
- Приоритетная очередь

Теорема: Пусть у нас есть n процессов в системе. Любая операция в детерминированной структуре данных с Strong Ordering Semantics может занять не менее n шагов.

Ellen, Hendler, Shavit, On the Inherent Sequentiality of Concurrent Objects, SICOMP 2013

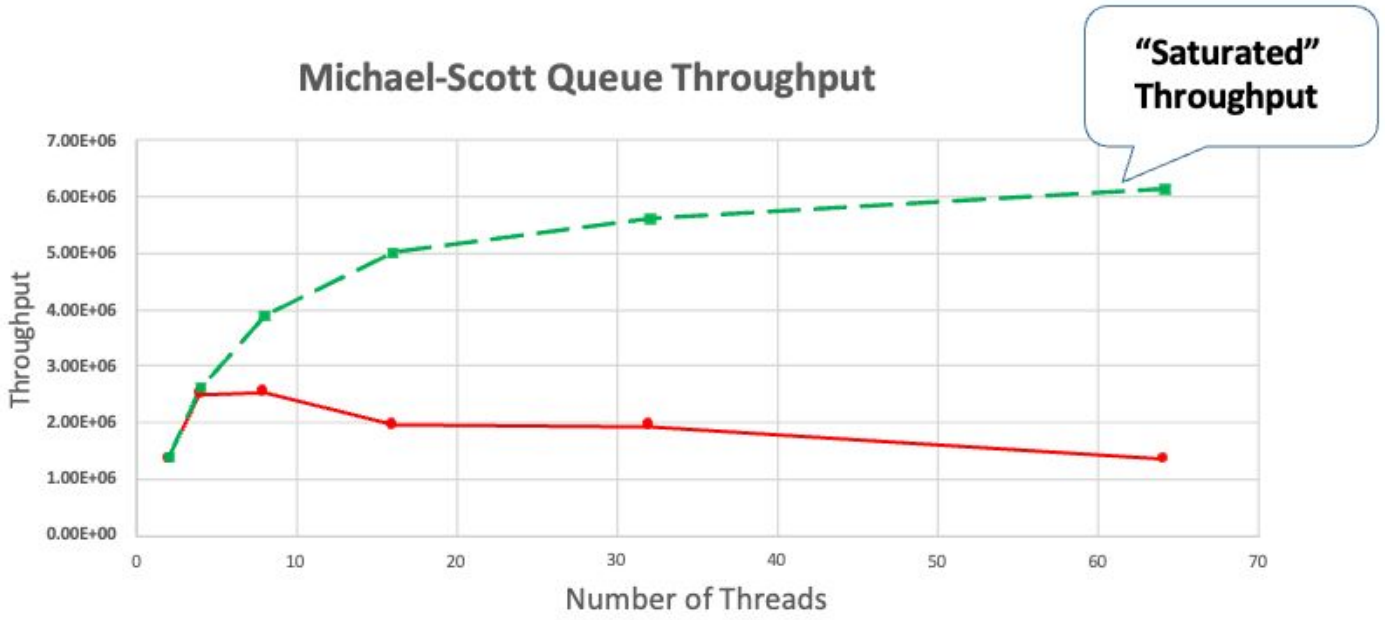
Ещё раз на графичек взглянем

ИТМО



Почему график такой плохой?

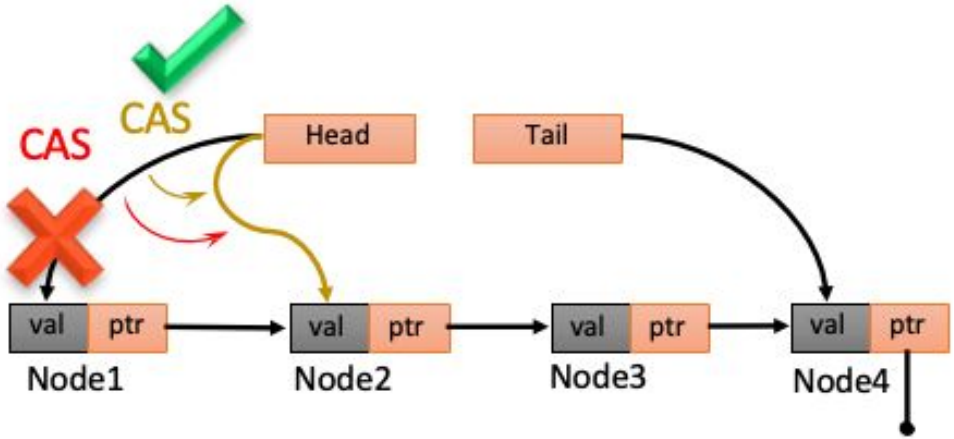
ИТМО



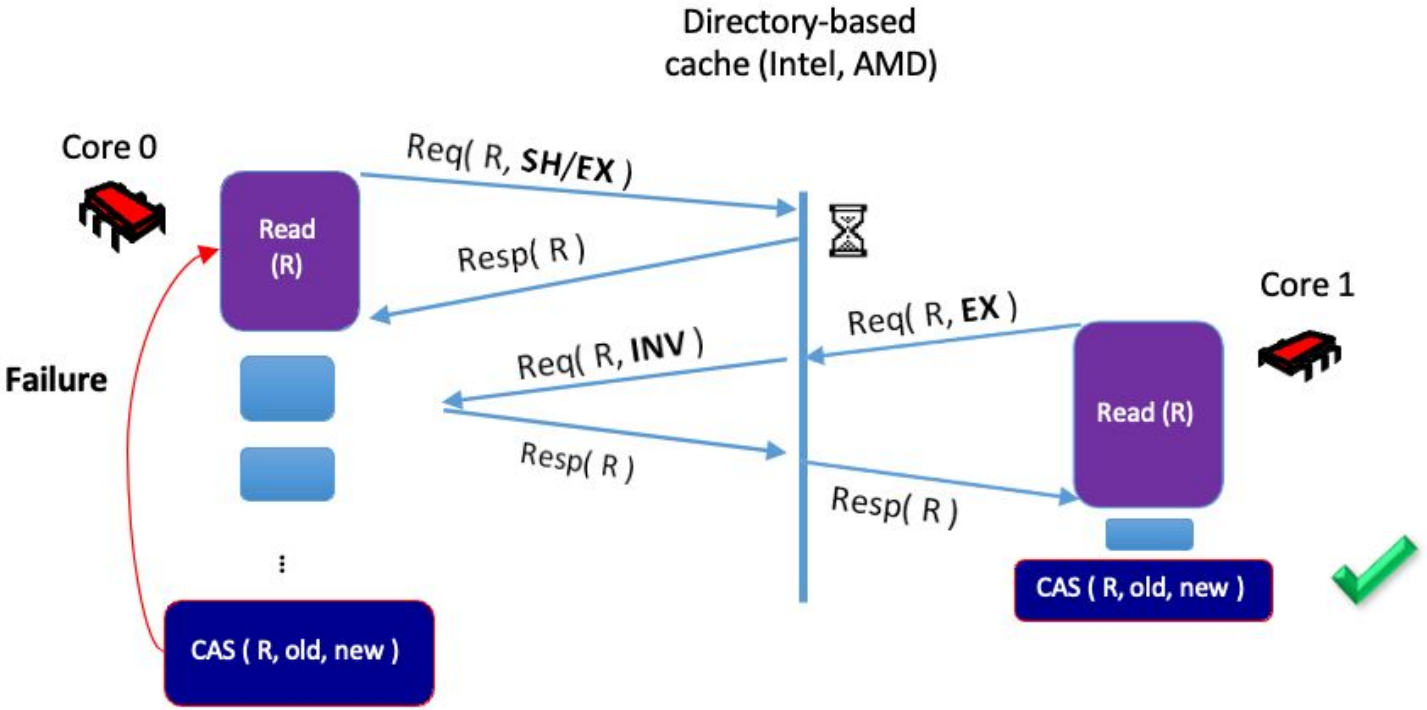
Lock-free очередь

ИТМО

```
V dequeue() {
    while (true) {
        node = head
        next_node = node.ptr
        if (CAS(head, node, next_node)) {
            return node.val
        }
    }
}
```



А вот беда с кешом...



Приоритетная очередь

ИТМО

Операции:

- `ExtractMin()`
- `Insert(<K, P>)`

Обычное использование

```
v = deleteMin()
for x in N(v) {
    c = calculate(v, x)
    insert(x, c)
}
```

Используется:

- Графовые алгоритмы
- Планировщик ОС (например, Windows 10)
- Биологические симуляции
- Belief-propagation

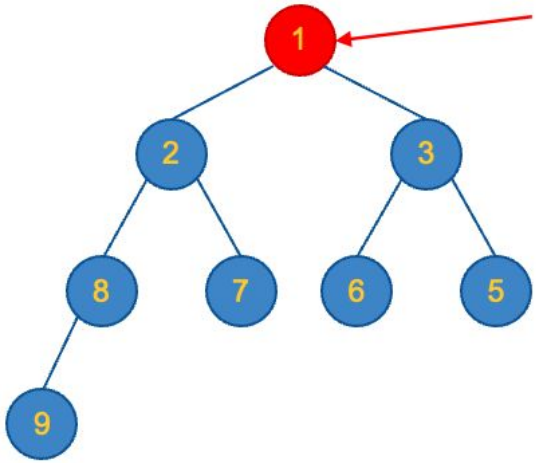
Конкурентная приоритетная очередь

Самая простая и быстрая приоритетная очередь - куча

Проблема: все операции идут через корень

- Проблемы с кешом
- Синхронизации много

Не особо подходит для многопоточности



Альтернативные структуры

ИТМО

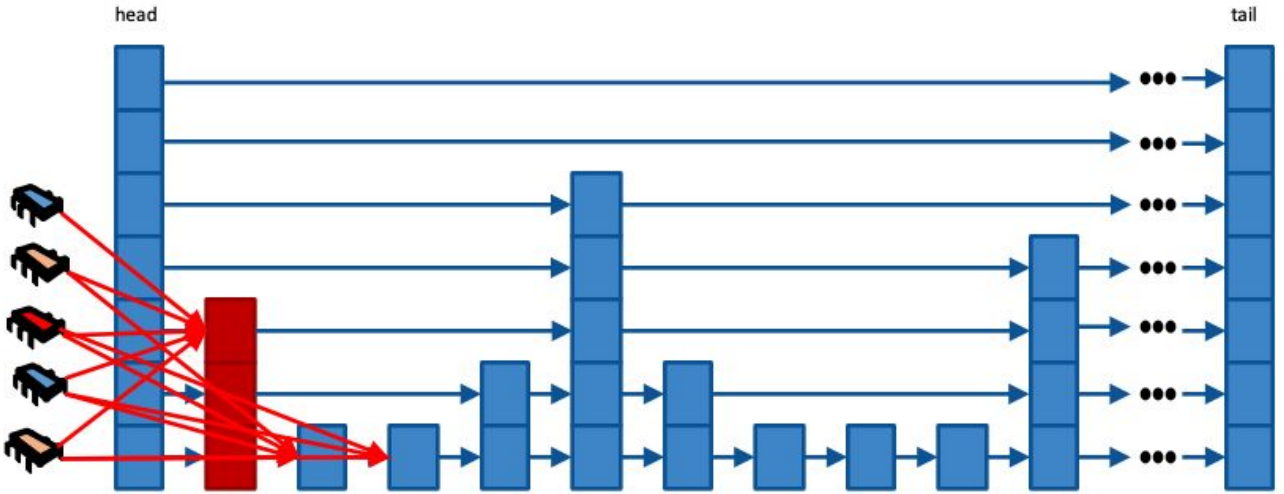
- По времени вставки $\log$, хочется использовать дерево поиска
- Но у него минимальный элемент ищется долго
- Оказывается, есть альтернатива: SkipList
- Она ещё и позволяет взять минимум за $0(1)$

SkipList

ИТМО

W. Pugh, Concurrent maintenance of skip lists, 1990
I. Lotan and N. Shavit, Skiplist-Based Concurrent Priority Queues, 2000

Still have large contention during extractMin



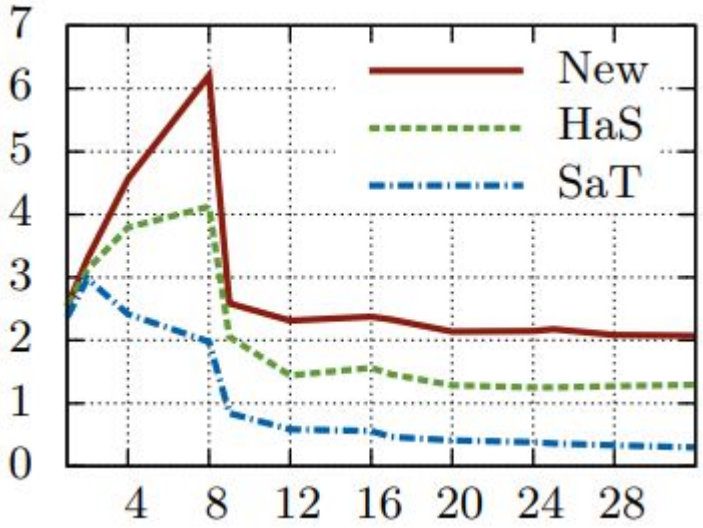
Неасимтотическое улучшение

Linden and Jonsson, A Skiplist-Based Concurrent Priority Queue ..., 2013

Идея:

- Удаляем только логически
- Храним первый неудалённый
- Когда набралось много удалённых, удаляем пачкой

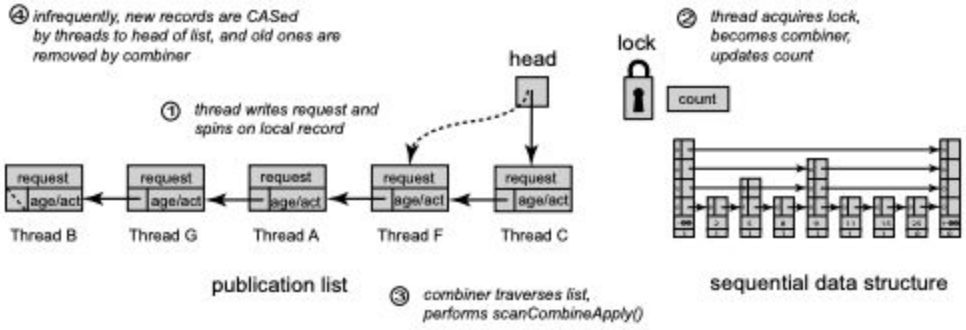
Всё ещё проблема с контешеном



Неасимтотическое улучшение 2

ИТМО

- У нас на extractMin все операции толпятся
- Надо ограничить нагрузку на память
- Сериализуем все операции и будем применять пачкой
- D. Hendler et al., Flat combining and the synchronization-parallelism tradeoff, 2010



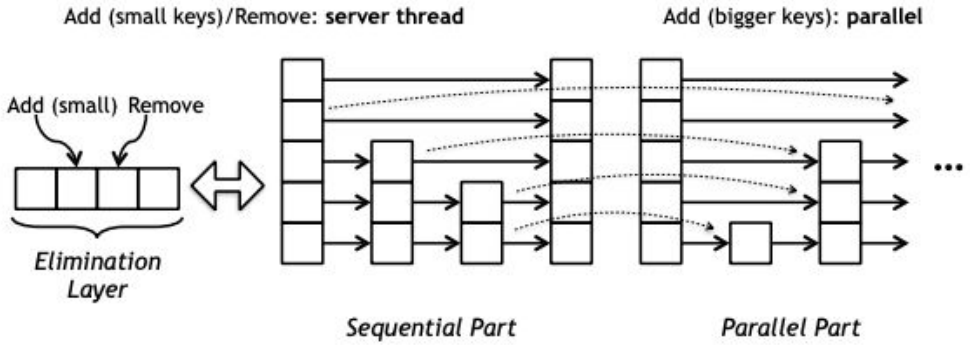
Flat-Combining

ИТМО

Проблема flat-combining, что он сериализует все операции

Но insert-то можно делать параллельно!

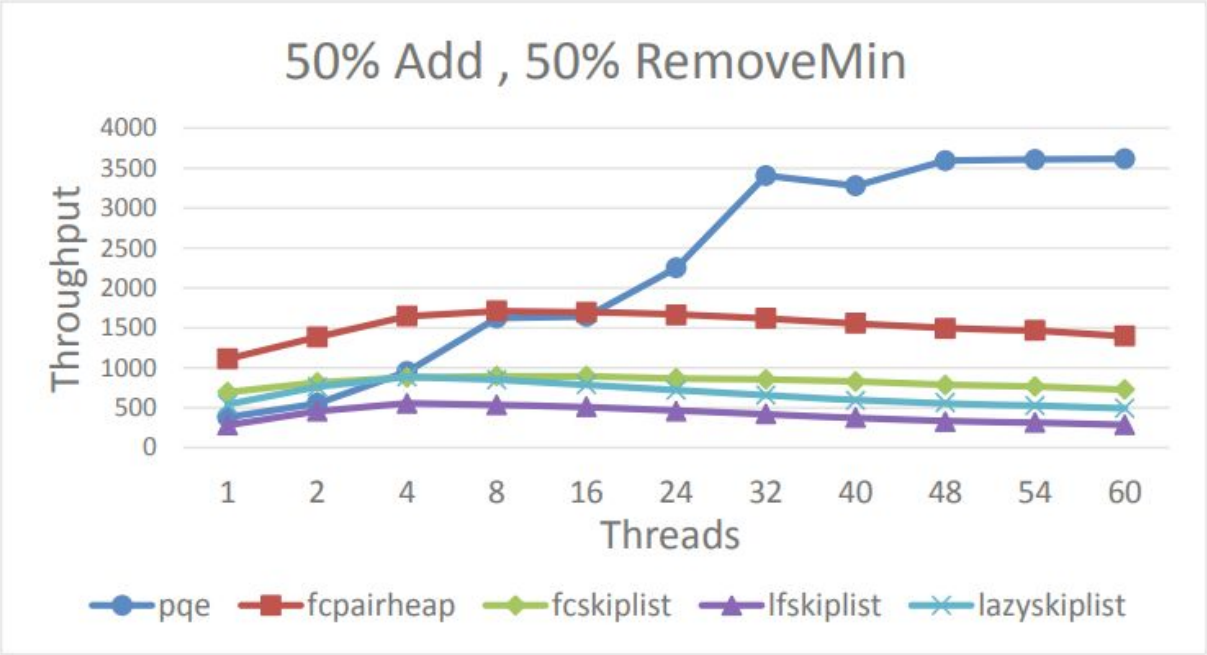
Разобьём структуру на части: младшая часть с flat-combining и старшая часть с обычным SkipList



Calciu et al., The Adaptive Priority Queue with Elimination and Combining, 2014

Улучшение такое себе

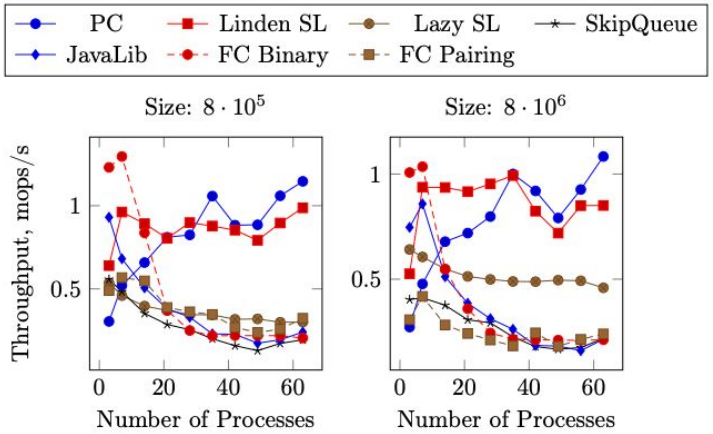
ИТМО



Parallel combining

ИТМО

- SkipList достаточно медленный, а куча - быстра
- Flat-Combining последовательный
- Но ждущие процессы могут применять операции все вместе!



Aksenov et al., Parallel Combining: Benefits of Explicit Synchronization, 2018

Остановимся подумать

ИТМО

- Теперь остановимся подумать на секунду
- Возьмём Дейкстру - как нам вообще её конкурентно сделать
- Берём элемент из очереди и релаксируем
- НО! Взяли элемент и заснули... Всё равно придётся перевставлять
- Тогда зачем нам в этом случае приоритетная очередь с гарантиями?
- Будем релаксировать! Начнём для интереса со счётчика

SprayList

ИТМО

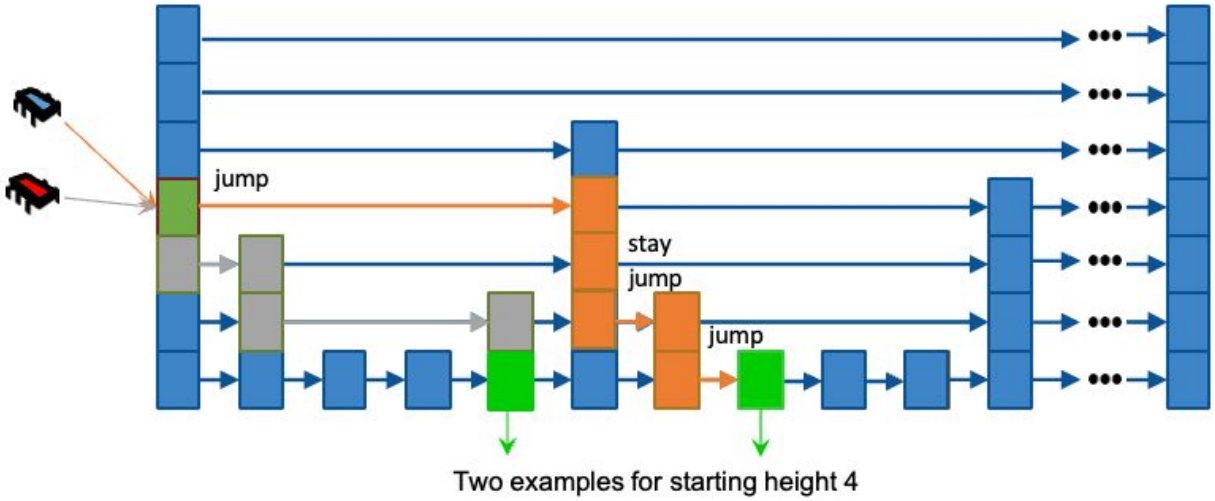
- В первую очередь, хочется что-то сделать со SkipList
- Собственно, “распылить” запросы и вытащить какой-то элемент из начала
- Допустим, хотим только первые X элементов

```
spray()
    start at height log X
    at each level, flip a coin to stay or jump forward
    repeat for each level
```

SprayList

ИТМО

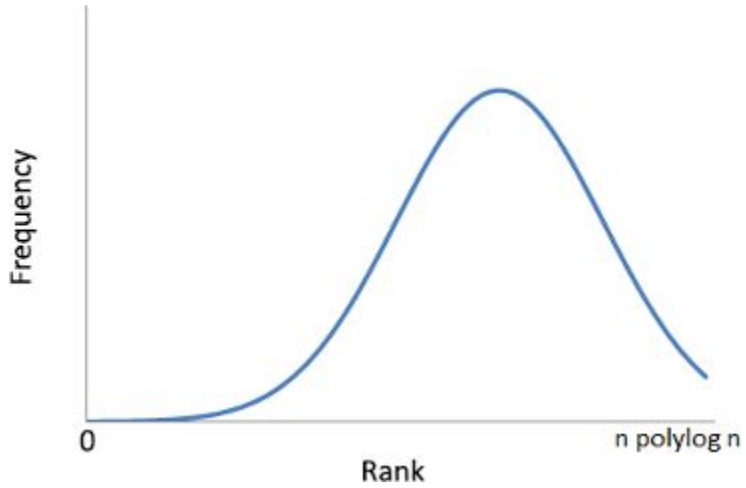
- Вероятно, хотим, чтобы каждый процесс получал свой элемент, поэтому X = P
- Матожидание начинать с уровня log P получаем как раз P
- Другое дело, что w.h.p. получается P log P



Небольшая проблема

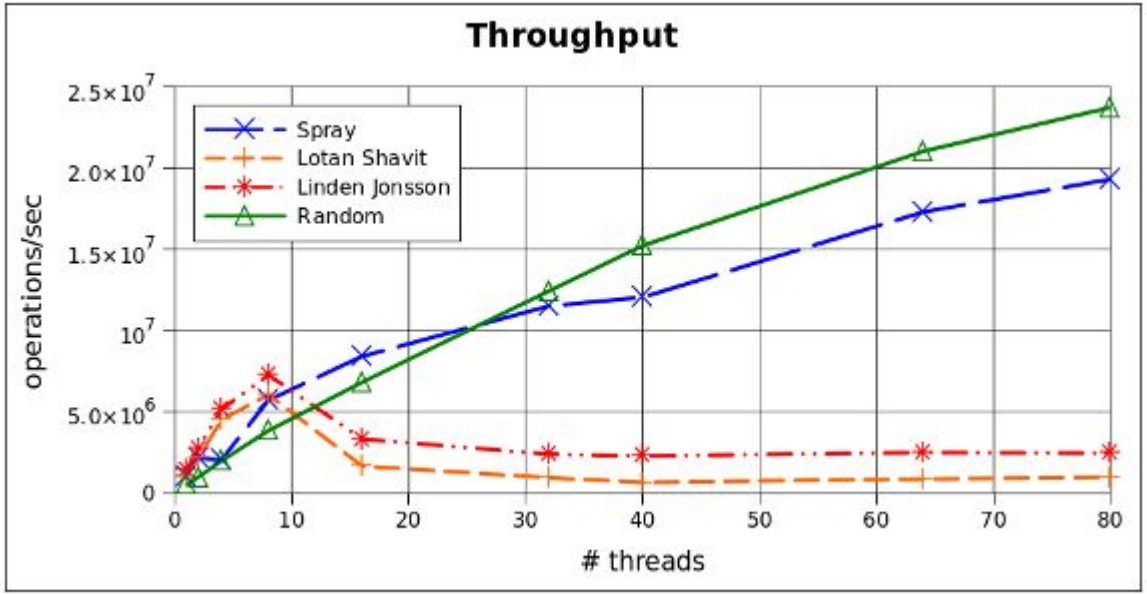
ИТМО

- Маленькие элементы имеют меньше вероятность вытащиться
- Вставим фиктивные элементы в начало



Простецкий бенчмарк

ИТМО



Можно ли лучше?

ИТМО

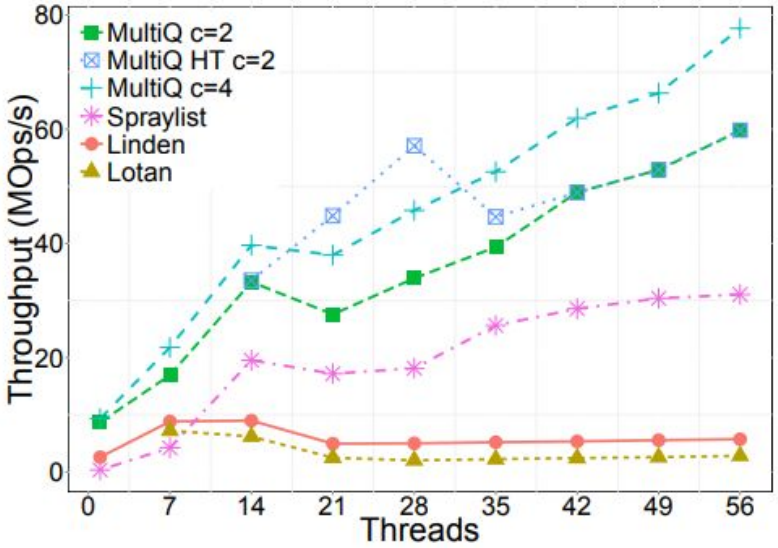
- Как мы поняли, SkipList медленнее кучи
- Можно ли что-то придумать с кучей?
- MultiQueue!

- Есть n куч, каждая с блокировкой
- `Insert:` берём случайную кучу и вставляем
- `ExtractMin:` выбираем две случайные ключи и вытаскиваем лучший

Какие-то такие результаты

Теорема: Если у нас n очередей, то ошибка будет O(n) в матожидании и O(n log n) w.h.p.

Alistarh et al., The Power of Choice in Priority Scheduling, 2017



Комментарий

ИТМО

- Если брать только одну очередь, то ошибка со временем будет расти
- Можно снизить нагрузку на блокировки, если выбирать две очереди только с вероятностью β.
- Тогда получим ошибку $O(n / (\beta^2 \log \beta))$ в матожидании и $O(n \log n / \beta)$ w.h.p.

Проверка факт

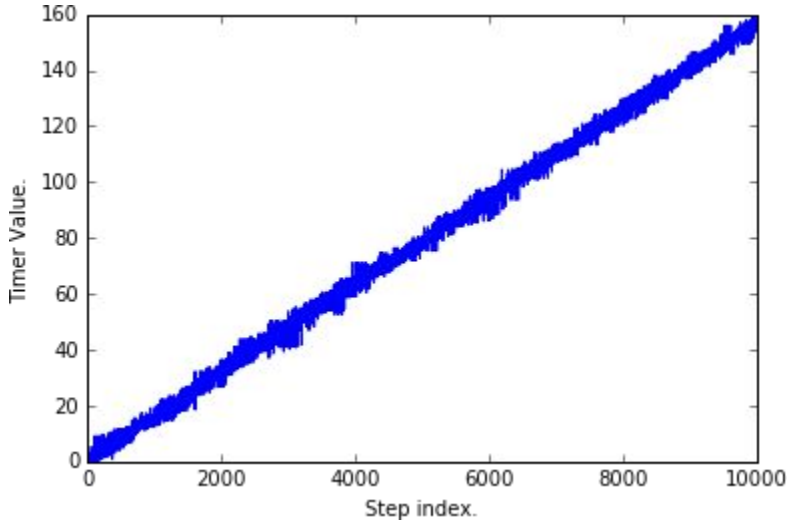
ИТМО

Напишем простой счётчик с помощью этой идеи.

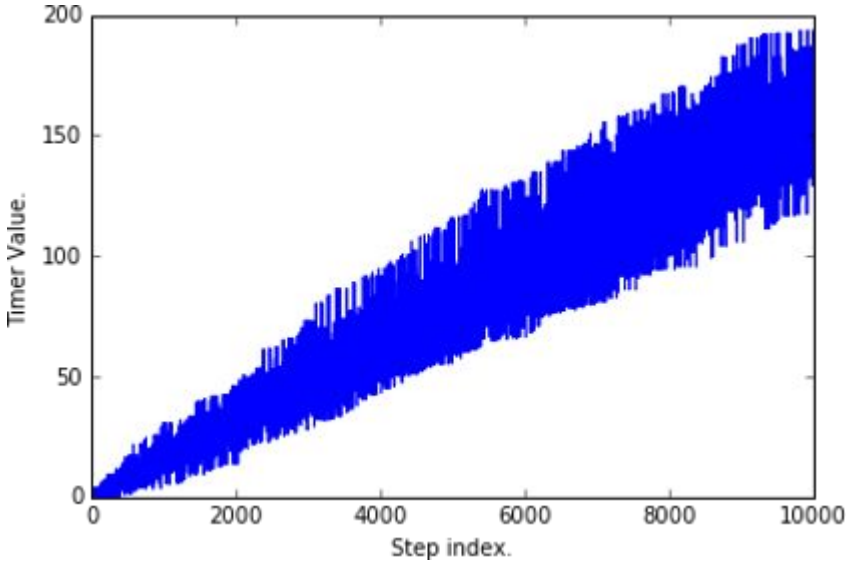
```
atomic<int> C[n]
int read() {
     i = random(0, n - 1)
     return C[i] * n
}

void increment() {
     i = random(0, n - 1)
     j = random(0, n - 1)
     if (C[j] < C[i]) i = j
     C[i].fai()
}
```

Эмпирический тест

- 64 счётчика
- Значение счётчика по OY



Эмпирический тест. Одиночный выбор



Применение

ИТМО

Дано:

- n задач
- DAG зависимостей из m рёбер
- Порядок на задачах π

Мы должны запустить задачи в этом порядке

Но при этом если нет зависимостей - можно запустить в любом

Пример:

- Жадные задачи на графах (MIS, MM, coloring)
- Алгоритм Дейкстры (но тут мы не знаем π)

Алгоритмы на слабой очереди

ИТМО

```
Q = PriorityQueue({v, π(v)})
while not Q.empty() {
    vt = Q.ExtractMin()
    Process(vt)
}
```

```
Q = RelaxedPQ({v, π(v)})
while not Q.empty() {
  vt = Q.ApproxMin()
  if vt has
      unprocessed predecessor {
    Q.insert({vt, π(v)})
      // Failed, reinsert
    continue
  }
  Process(vt)
}
```

Вопрос: появляется инверсия приоритетов - насколько много?

Теорема: Если используется k очередей, то число итераций n + O(m/n poly(k))

Alistarh et al., Relaxed Schedulers Can Efficiently Parallelize Iterative Algorithms, 2017

Уточнение под задачи

ИТМО

Minimal Independent Set

```
Q = RelaxedPQ({v, π(v)})
while not Q.empty() {
  vt = Q.ApproxMin()
  if vt is dead {
    continue
  } elif vt has live predecessor {
    Q.insert({vt, π(v)})
      // Failed, reinsert
    continue
  } else {
    Add vt to MIS
    Mark all of vt's neighbours dead
  }
}
```

Теорема: Алгоритм делает n + poly(k) итераций

Dijkstra

Теорема: Алгоритм делает $n + O(k^2 d_{max} / w_{min})$ итераций.

Машинное обучение где-то есть?

ИТМО

Belief-propagation

Набор случайных переменных $X = (X_1, ..., X_n)$

$$\begin{aligned} &\psi_i \colon D_i \to \mathbb{R}^+ && \text{for } i \in V, \\ &\psi_{ij} \colon D_i \times D_j \to \mathbb{R}^+ && \text{for } \{i,j\} \in E, \end{aligned}$$

$$\Pr[X = x] \propto \prod_i \psi_i(x_i) \prod_{ij} \psi_{ij}(x_i, x_j),$$

Задача: посчитать маргинализацию $P[X_i = x]$

Belief-Propagation

ИТМО

- Используется эвристический алгоритм итеративного обновления
  - Почти как PageRank
- Запускается, пока ошибка не станет маленькой
- Гарантии завершения нет, но если завершается - всё корректно
- Неасимптотическая оптимизация: обновлять в некотором приоритете
- Ну, и тут, здравствуйте, приоритетная очередь

Aksenov et al., Relaxed Scheduling for Belief Propagation, 2020

А есть ли применение choice of 2?

ИТМО

- Допустим, у нас есть n машин, которые обучают сеточку, но у нас лимитирована коммуникация и нельзя сделать AllReduce
- Если машинки сидят по себе - они расходятся, а если их друг с другом переговариваться? **Сходится!**

```
% i and j are chosen uniformly at random, with replacement
upon each interaction between agents i and j
    % each agent performs a local SGD step
    X^i ← X^i − η^i g̃^i(X^i)
    X^j ← X^j − η^j g̃^j(X^j)
    % agents average their estimates coordinate-wise
    avg ← (X^i + X^j)/2
    X^i ← avg
    X^j ← avg
```

Aksenov et al., PopSGD: Decentralized Stochastic Gradient Descent in the Population Model, 2020

Спасибо за внимание