Relaxed Algorithms

Никита Коваль, Роман Елизаров

ИТМО 2020

Кто не умеет писать BFS?

```
val Q = Queue<Node>()
start.distance = 0 // INF for others
Q.add(start)
while Q.isNotEmpty() {
  val u = Q.remove()
  d = u.distance
  for (v : u.edges) {
    if v.distance != INF: continue
    v.distance = d + 1
    Q.add(v)
  }
}
```

Параллельный BFS: 1st attempt

N потоков

```
val Q = ConcurrentQueue<Node>()
start.distance = 0 // INF for others
Q.add(start)
while Q.isNotEmpty() {
  val u = Q.remove()
  d = u.distance
  for (v : u.edges) {
    if v.distance != INF: continue
    v.distance = d + 1
    Q.add(v)
  }
}
```

Параллельный BFS: 1st attempt

N потоков

```
val Q = ConcurrentQueue<Node>()
start.distance = 0 // INF for others
Q.add(start)
while Q.isNotEmpty() {
  val u = Q.remove()
  d = u.distance
  for (v : u.edges) {
    if v.distance != INF: continue
    v.distance = d + 1
    Q.add(v)
  }
}
```

Этот алгоритм корректен?

Параллельный BFS: 1st attempt

N потоков

```
val Q = ConcurrentQueue<Node>()
start.distance = 0 // INF for others
Q.add(start)
while Q.isNotEmpty() {
  val u = Q.remove()
  d = u.distance
  for (v : u.edges) {
    if v.distance != INF: continue
    v.distance = d + 1
    Q.add(v)
  }
}
```

Этот алгоритм корректен?

Вероятно, останется только один поток...

Параллельный BFS: 2nd attempt

N потоков

```
val Q = ConcurrentQueue<Node>()
start.distance = 0 // INF for others
Q.add(start)
activeNodes++
while activeNodes > 0 {
  val u = Q.remove()
  d = u.distance
  for (v : u.edges) {
    if v.distance != INF: continue
    v.distance = d + 1
    Q.add(v); activeNodes++
  }
  activeNodes--
}
```

Параллельный BFS: 2nd attempt

N потоков

```
val Q = ConcurrentQueue<Node>()
start.distance = 0 // INF for others
Q.add(start)
activeNodes++
while activeNodes > 0 {
  val u = Q.remove()
  d = u.distance
  for (v : u.edges) {
    if v.distance != INF: continue
    v.distance = d + 1
    Q.add(v); activeNodes++
  }
  activeNodes--
}
```

А теперь корректен?

Параллельный BFS: 2nd attempt

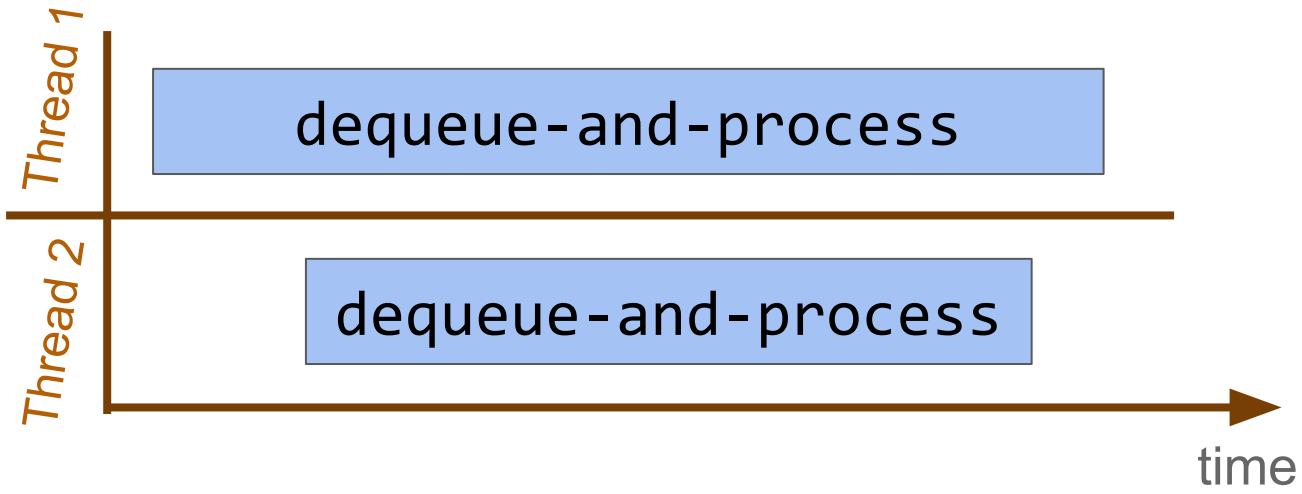
```
val Q = ConcurrentQueue<Node>()
start.distance = 0 // INF for others
Q.add(start)
activeNodes++
while activeNodes > 0 {
  val u = Q.remove()
  d = u.distance
  for (v : u.edges) {
    if v.distance != INF: continue
    v.distance = d + 1
    Q.add(v); activeNodes++
  }
  activeNodes--
}
```

А теперь корректен?

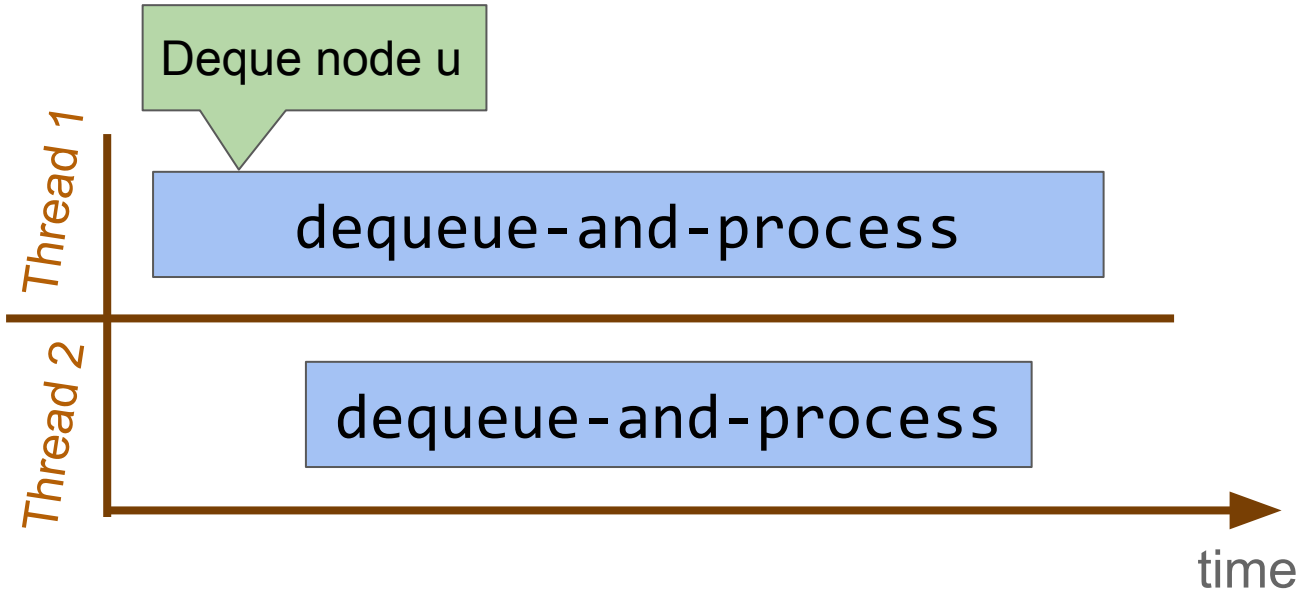
N потоков

Можем считать количество потоков без работы

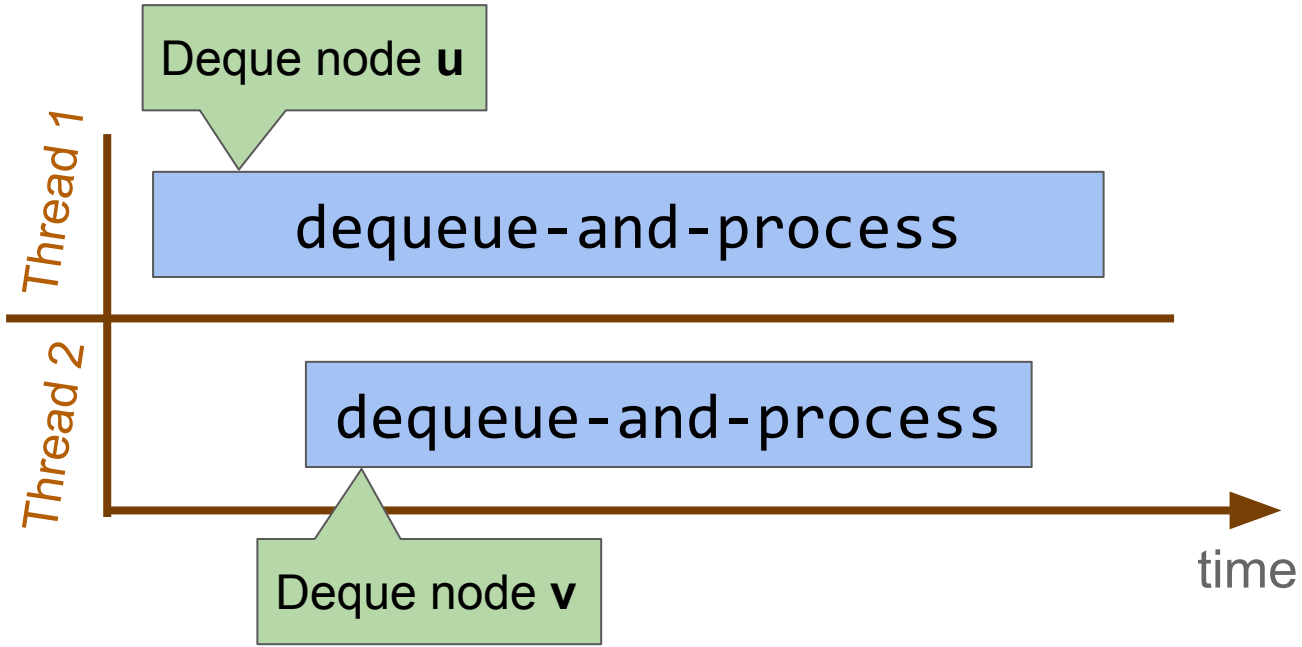
Гарантируется ли порядок обработки вершин?



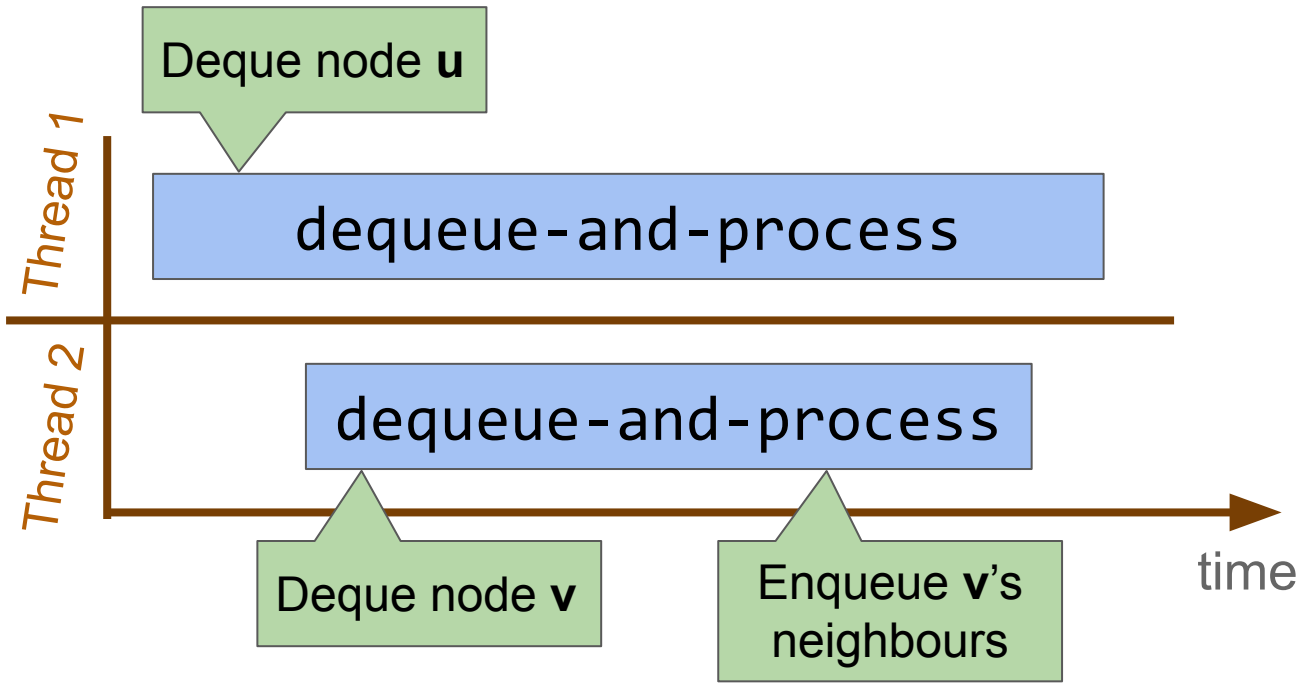
Гарантируется ли порядок обработки вершин?



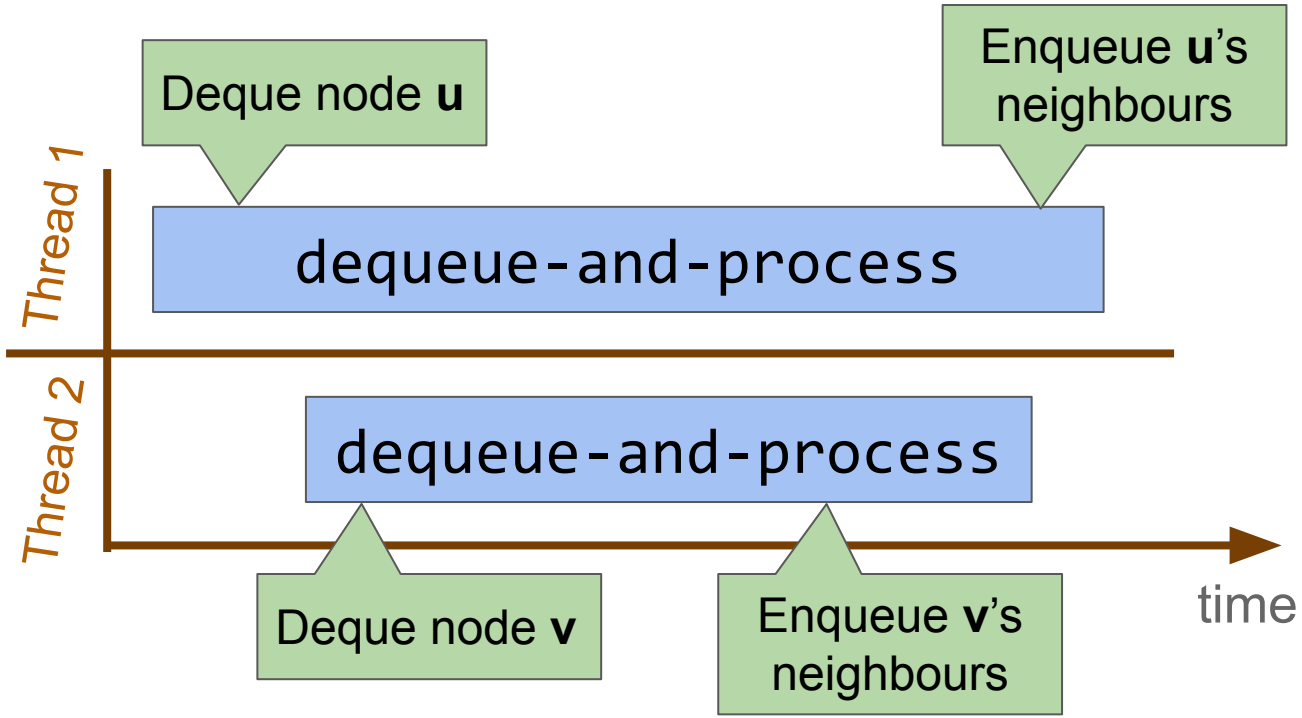
Гарантируется ли порядок обработки вершин?



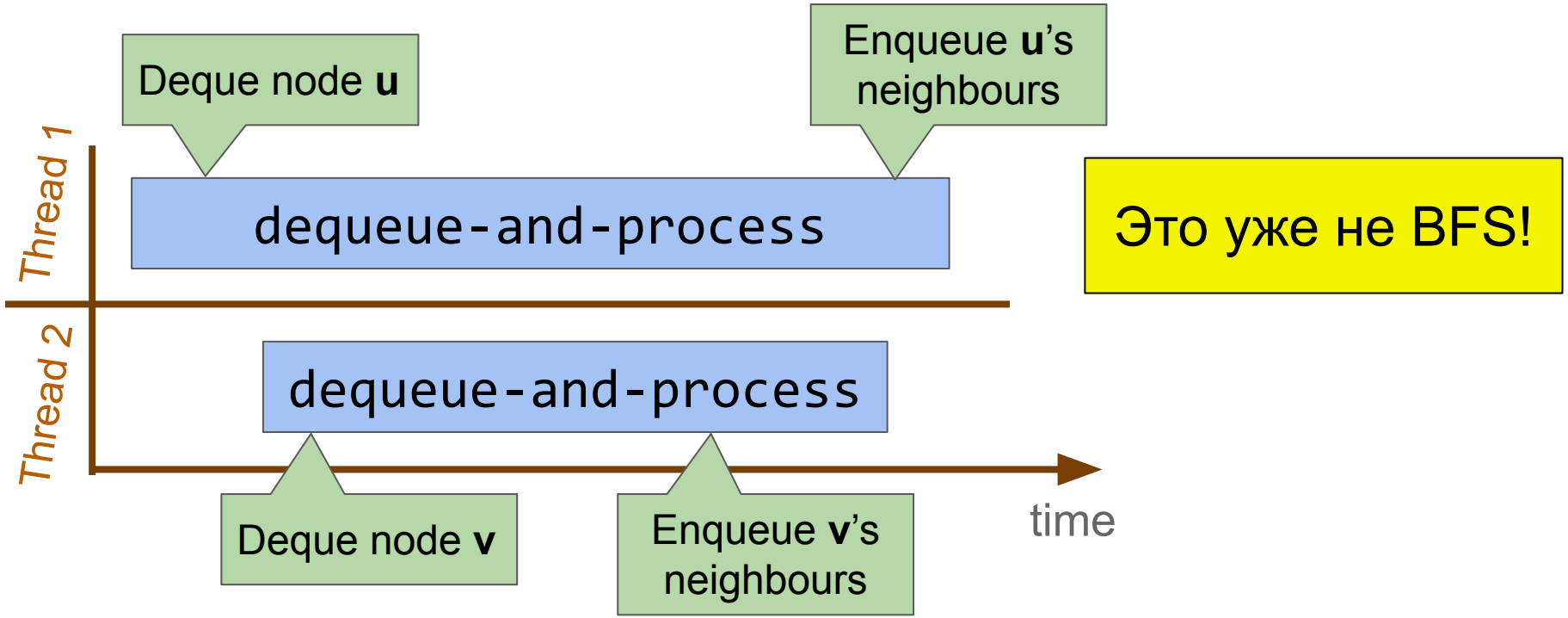
Гарантируется ли порядок обработки вершин?



Гарантируется ли порядок обработки вершин?

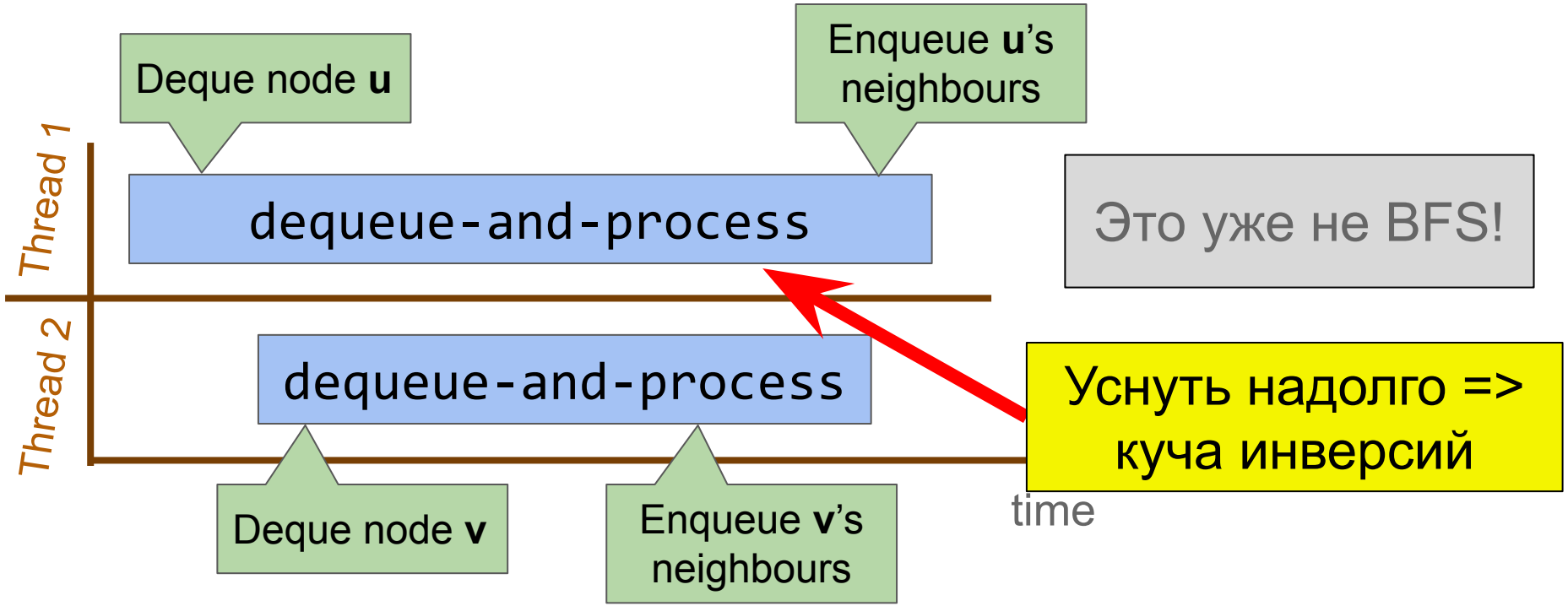


Гарантируется ли порядок обработки вершин?



Это уже не BFS!

Гарантируется ли порядок обработки вершин?



Параллельный BFS: 3rd attempt

N потоков

```
val Q = ConcurrentQueue<Node>()
start.distance = 0 // INF for others
Q.add(start)
activeNodes++
while activeNodes > 0 {
  val u = Q.remove()
  d = u.distance
  for (v : u.edges) {
    if v.updateDistIfLower(d + 1) {
      v.distance = d + 1
      Q.add(v); activeNodes++
    }
  }
  activeNodes--
}
```

Можем посетить одну вершину несколько раз

Параллельный BFS: Trade-Offs

- Последовательный BFS посещает каждую вершину ровно один раз
- Параллельная версия может посетить вершину много раз
- Где мы выигрываем и проигрываем?
  - **Win:** параллельная обработка вершин
  - **Loss:** лишняя обработка вершин
- На реальных графах **Win >> Loss**
  - Например, на деревьях нет повторной обработки вершин

Алгоритм Дейкстры

- Инкрементально улучшает кратчайшие расстояния
  - Прямо как в нашем параллельном BFS

Алгоритм Дейкстры

- Инкрементально улучшает кратчайшие расстояния
  - Прямо как в нашем параллельном BFS
- Использует `extractMin` вместо `dequeue`

Алгоритм Дейкстры

- Инкрементально улучшает кратчайшие расстояния
  - Прямо как в нашем параллельном BFS
- Использует `extractMin` вместо `dequeue`

- В последовательном алгоритме каждая вершина посещается один раз
- Многопоточная версия => вершина может быть обработана несколько раз
  - Тот же trade-off

Параллельный BFS: нужна ли нам очередь?

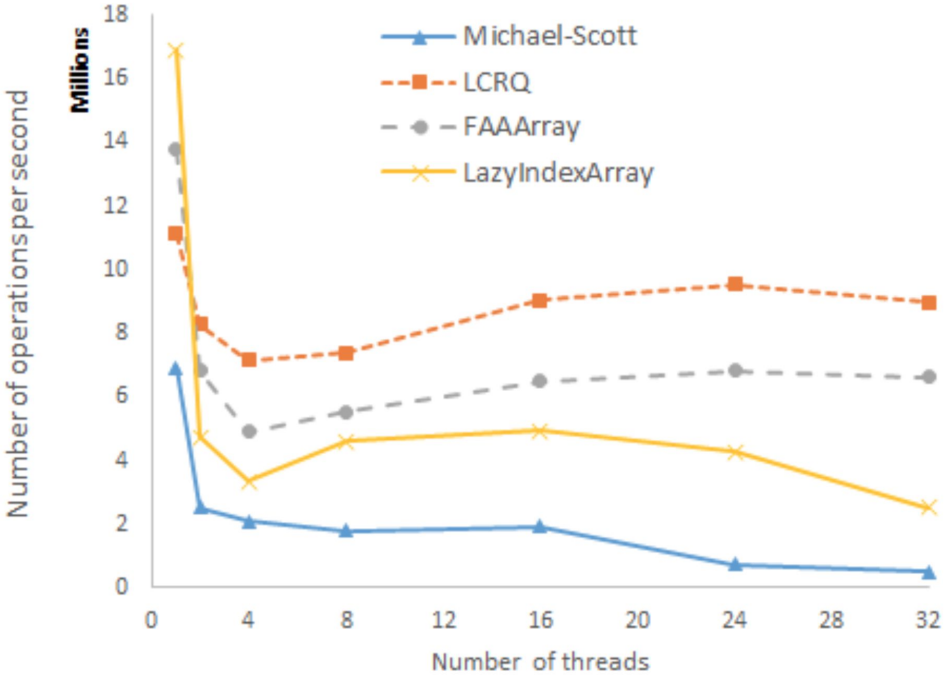
N потоков

```
val Q = ConcurrentStack???<Node>()
start.distance = 0 // INF for others
Q.add(start)
activeNodes++
while activeNodes > 0 {
  val u = Q.remove()
  d = u.distance
  for (v : u.edges) {
    if v.updateDistIfLower(d + 1) {
      v.distance = d + 1
      Q.add(v); activeNodes++
    }
  }
  activeNodes--
}
```

Нужна ли эта честность?

- Честные FIFO/приоритетные очереди не делают реализацию параллельных версий BFS/Dijkstra проще
  - Оба алгоритма не полагаются на честность из-за возможных инверсий
- Насколько честные очереди нам на самом деле нужны?
  - Точность убивает масштабируемость
  - Произвольный порядок *может* сильно увеличить лишнюю работу

Масштабируемость очередей



Don’t scale
Never have
Never will

http://concurrencyfreaks.blogspot.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html

Нужна ли эта честность?

- Честные FIFO/приоритетные очереди не делают реализацию параллельных версий BFS/Dijkstra проще
  - Оба алгоритма не полагаются на честность из-за возможных инверсий
- Насколько честные очереди нам на самом деле нужны?
  - Точность убивает масштабируемость
  - Произвольный порядок *может* сильно увеличить лишнюю работу

Relaxed data structures!

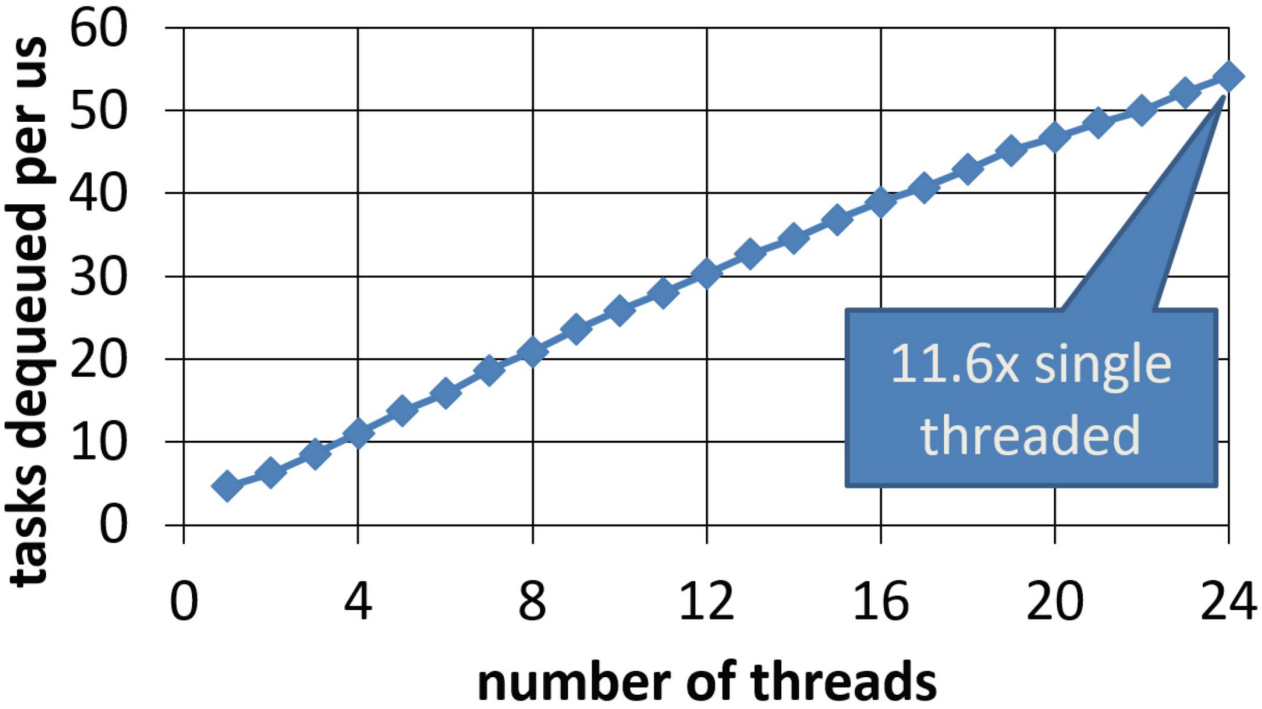
Multi-Queue: Concurrent Relaxed Queue

- Берём любимую последовательную/многопоточную версию
  - Чем более cache-friendly, тем лучше
  - Последовательные версии будем защищать блокировкой
- Заведём N независимых очередей
  - где N - количество потоков
- Операции `dequeue`/`enqueue` будут работать над случайно выбранными очередями
- `Dequeue` будет возвращать элемент, близкий к минимальному

Priority Multi-Queue Example

- 2N *последовательных* binary heaps с блокировками
- Добавление:
  - Выбираем случайную очередь **q**
  - Пытаемся взять блокировку на **q**
  - Вставляем пару `<element, priority>`
- Удаление:
  - Выбираем *две* случайные очереди
  - Пытаемся взять блокировки на них
  - Удаляем минимальный элемент
- При неуспешном взятии блокировки начинаем заново

FIFO Multi-Queue Performance



Задачка на подумать

Аркадий решил соптимизировать счётчик. Для этого вместо одной переменной он завел целый массив чисел размера *K*. Недолго размышляя, в качестве увеличения счетчика он решил увеличивать случайный элемент массива с помощью операции Fetch-And-Add, атомарно увеличивающей число, а для получения значения – просто проходить по всем элементам и их суммировать. Хотя такой подход и выглядит небезопасным (ведь во время суммирования могут происходить увеличения), написанные Василием тесты ошибку не нашли. Помогите ему доказать или опровергнуть корректность.

```
fun inc() {
    i := rand(K)
    FAA(&A[i], +1)
}

fun get() {
    sum := 0
    for (i := 0; i < K; i++)
    sum += A[i]
    return sum
}
```