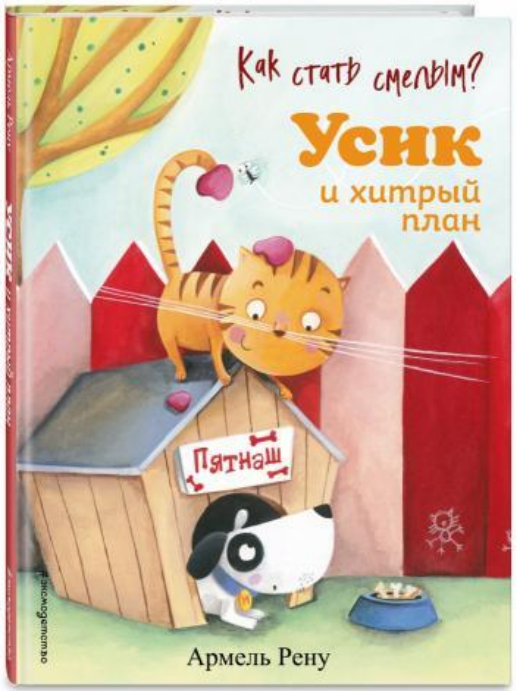
УНИВЕРСИТЕТ ИТМО

Введение в параллельное программирование

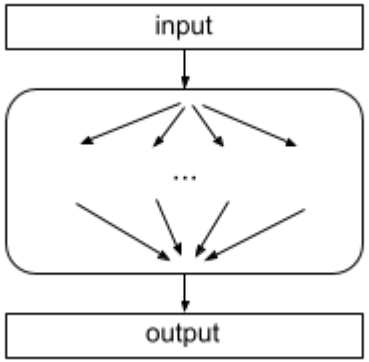
Виталий Аксенов
aksenov.vitaly@gmail.com

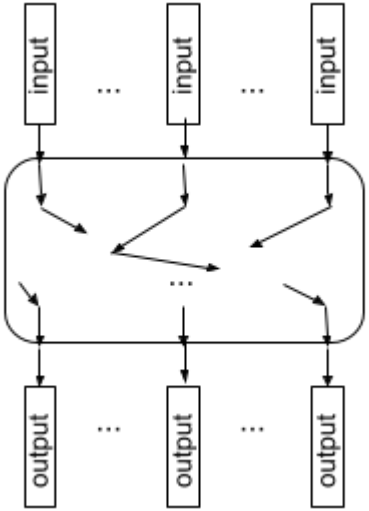
Хитрый план

- Введение
- Модель
- Операции над массивом
- Различные алгоритмы



Parallel vs Concurrent



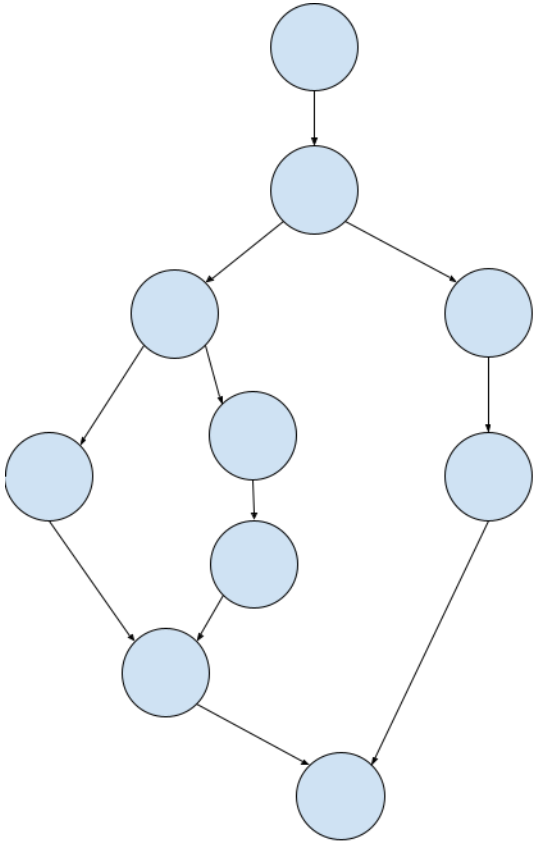


PRAM

- У каждого процессора своя программа
- Все инструкции выполняются по одной синхронно
- Вариации:
  - EREW, Exclusive-Read Exclusive-Write
  - CREW, Concurrent-Read Exclusive-Write
  - CRCW, Concurrent-Read Concurrent-Write
    - Uniform
    - Random
    - Priority
- Ослабление модели: Bulk-Synchronous Model

Fork-join

- Два примитива: fork и join – или fork2join
- Получается DAG исполнения
- Work – последовательное время работы
- Span – время работы на бесконечном числе процессоров
- Нужен правильный шедулинг



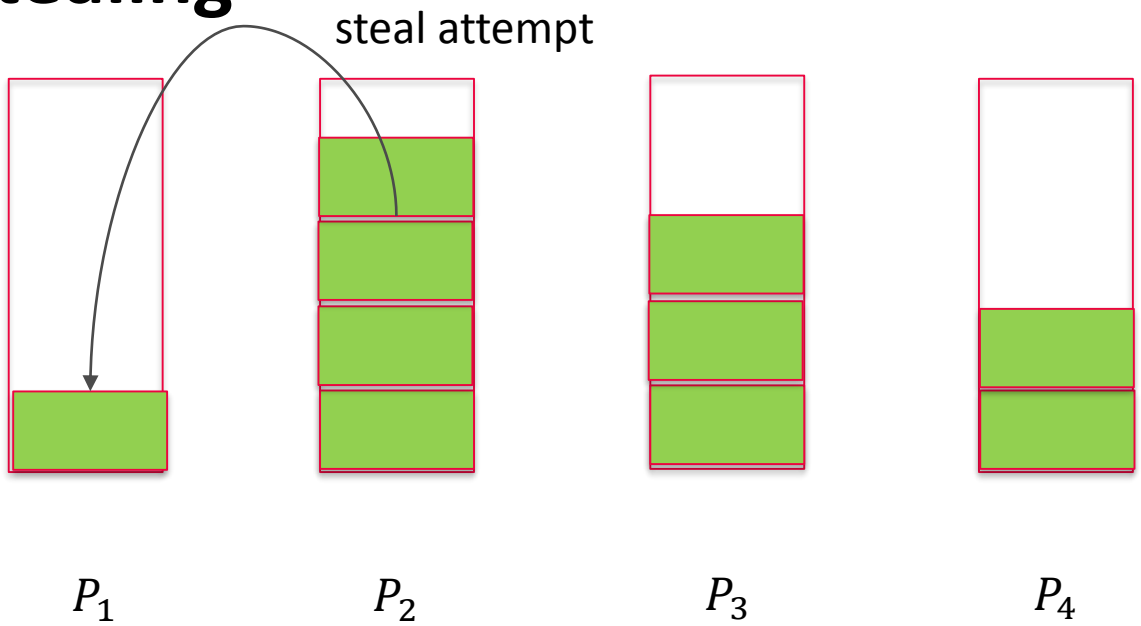
Work: 10

Span: 7

Fork-join Scheduling

- Если построен граф (DAG), то можно выполнять задания по уровням.
- Теорема Брента: $T_p = \frac{W}{p} + S$.
- Work-stealing: $E[T_p] = O(\frac{W}{p} + S)$.
- Потенциальное число процессов $\leq \frac{W}{S}$

Work-Stealing

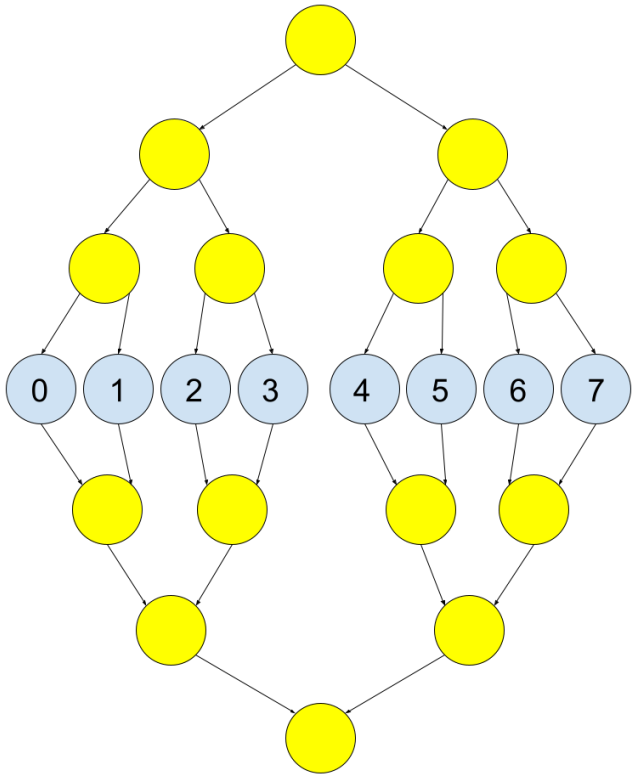


Work-Stealing

- Пусть есть вычисление с W work и S span, которое запускают на неблокирущем Work-Stealing шедулере с P процессами. Время работы:
  - $O(\frac{W}{P} + S)$ в мат. ожидании
  - $O(\frac{W}{P} + S + \log\frac{1}{\epsilon})$ с вероятностью не менее $1 - \epsilon$

Parallel For

```
parallel_for(l, r, body):
    if l == r - 1:
        body(l)
        return
    m = (l + r) / 2
    fork2join(
        [&] { parallel_for(l, m, body) },
        [&] { parallel_for(m, r, body) }
    )
```



parallel_for(0, 8, body)

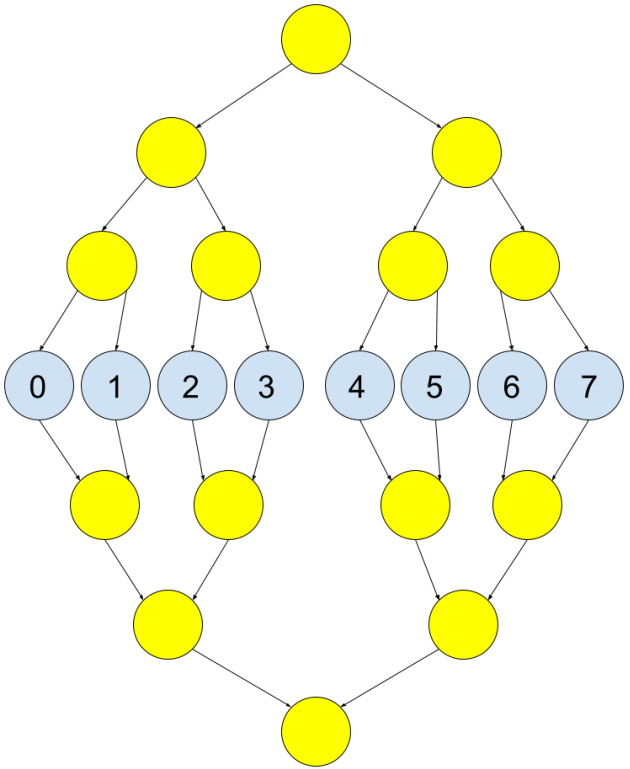
Parallel For ⟶ Map

```
parallel_for(0, n, [&] (int i) { b[i] = f(a[i]) })
```

Complexity(f) = $O(1)$

Work: $O(n)$

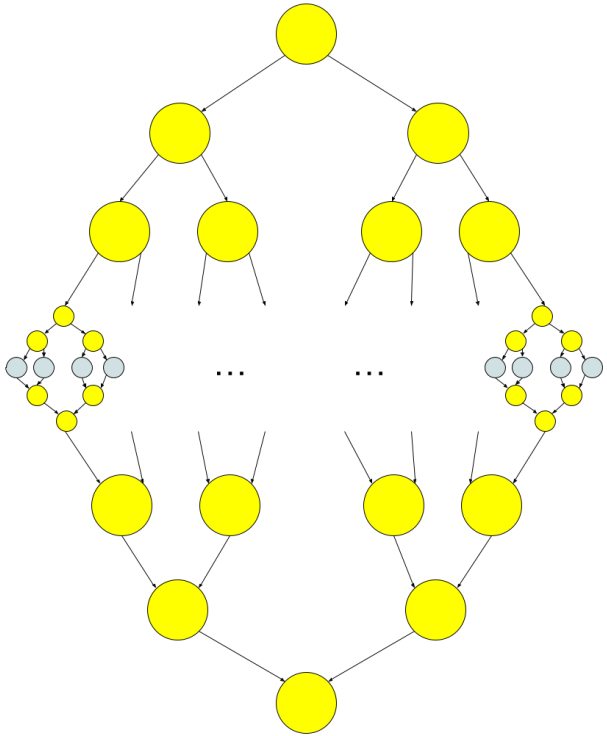
Span: $O(\log n)$




```
parallel_for(0, 8, body)
```

Два вложенных Parallel For

```
parallel_for(0, n, [&] (int i) {
    parallel_for(0, m, [&] (int j) {
        b[i][j] = f(a[i][j])
    })
})
```



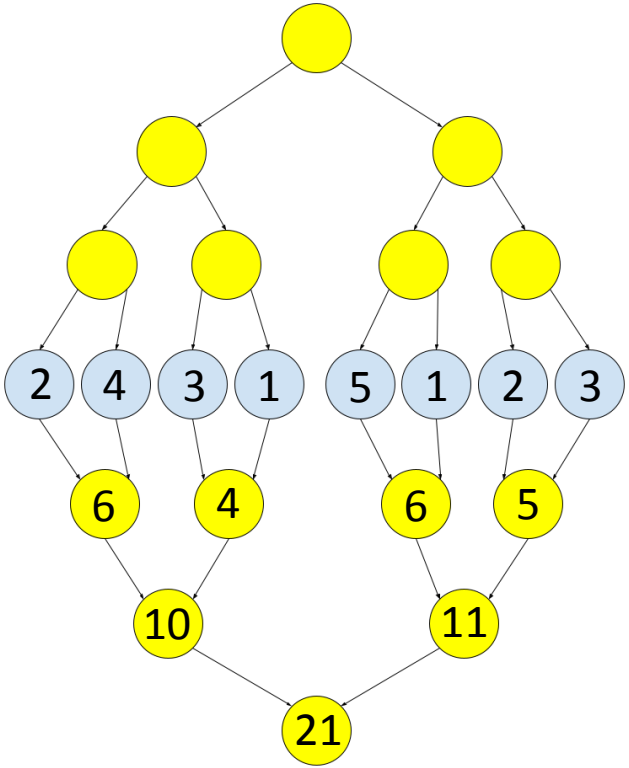
Work: $O(n \cdot m)$

Span: ? $O(\log n + \log m)$ or $O(\log n \cdot \log m)$

Parallel For ⟶ Reduce

```
reduce(l, r, op):
    if l == r - 1:
        return a[l]
    m = (l + r) / 2
    fork2join(
        [&] { left = reduce(l, m, op) },
        [&] { right = reduce(m, r, op) }
    )
    return op(left, right)
```

Work: $O(n \cdot comp(op))$
Span: $O(comp(op) \cdot \log n)$

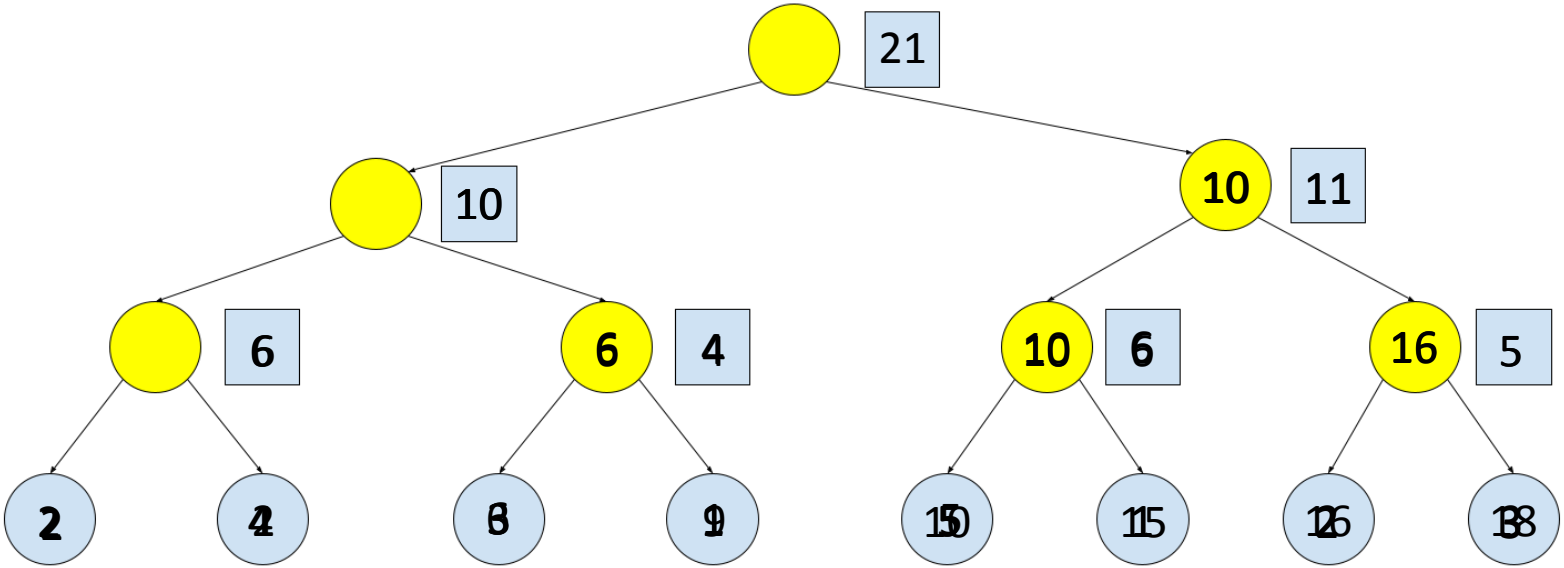


reduce(0, 8, +)

Reduce → Scan

- Префиксная сумма (или любая другая ассоциативная функция)
- Inclusive: $y_i = \sum_{j=0}^{i} x_j$
- Exclusive: $y_i = \sum_{j=0}^{i-1} x_j$

Reduce → Scan



Scan → Filter

```
filter(a, f):
    flags = map(a, f)
    sums = scan(flags)
    answer = new Object[sums[-1]]
    parallel_for(0, len(answer), [&] (int i) {
        if flags[i] == 1:
            answer[sums[i] - 1] = a[i]
    }
    return answer
```

Granularity Problem

- Оказывается, реализации написанные таким образом работают не очень хорошо. Почему?
- На то есть причина – проблема гранулярности
- С одной стороны мы получаем наибольший параллелизм, а с другой получаем большой оверхед на создание такого большого числа потоков.
- Надо балансировать. Поэтому придётся писать код иначе.

Reduce. Revisited.

```
reduce(l, r, op):
    if |r - l| < BLOCK_SIZE:
        for (int i = l; i < r; i++)
            ans = op(ans, a[i])
        return ans
    m = (l + r) / 2
    fork2join(
        [&] { left = reduce(l, m, op) },
        [&] { right = reduce(m, r, op) }
    )
    return op(left, right)
```

Решение проблемы

- Перед запуском функции мы проверяем размер задачи, например, n log n для сортировки, сравниваем с `BLOCK_SIZE` и решаем – надо ли запускать последовательную версию
- Можно это делать более-менее автоматически.

Acar, Aksenov, Charguéraud, Rainey, Provably and Practically Efficient Granularity Control. PPoPP 2019

- **`cilk_for`**, например, бьёт на 8p равных частей. Решает много проблем.

Сортировочки. Quick sort

```
quicksort(a):
    if len(a) < BLOCK_SIZE:
        return quicksort_seq(a)
    x = a[random()]
    l = filter(a, "< x")
    m = filter(a, "= x")
    r = filter(a, "> x")

    fork2join([&] { l = quicksort(l) },
              [&] { r = quicksort(r) } )

    return l ++ m ++ r
```

Work: $O(n \log n)$
Span: $O(\log^2 n)$

Когда мало процессов, лучше делать последовательный partition
Work: $O(n \log n)$
Span: $O(n)$

Сортировочки. Merge Sort.

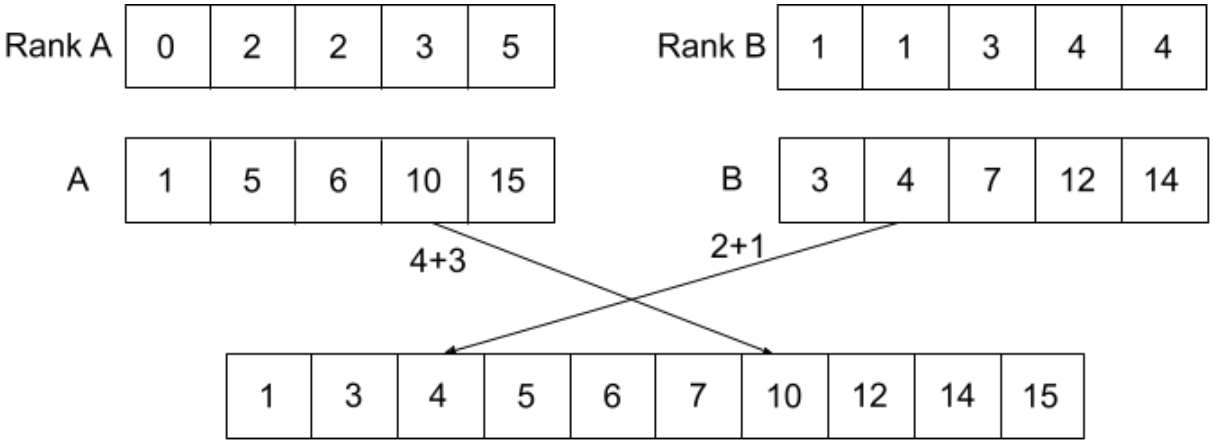
```
mergesort(a):
    if len(a) < BLOCK_SIZE:
        return mergesort_seq(a)
    l = a[:len(a)/2]
    r = a[len(a)/2:]

    fork2join([&] { l = mergesort(l) },
              [&] { r = mergesort(r) } )
    return merge(l, r)
```

Теперь нам нужно научиться, как объединять два отсортированных массива в параллель

Сортировочки. Merge из MergeSort

- Пусть у нас 2 массива: A и B.
- Считаем `rank(A[i], B)`, тогда в объединённом массиве `A[i]` будет на позиции `i+rank(A[i], B)`

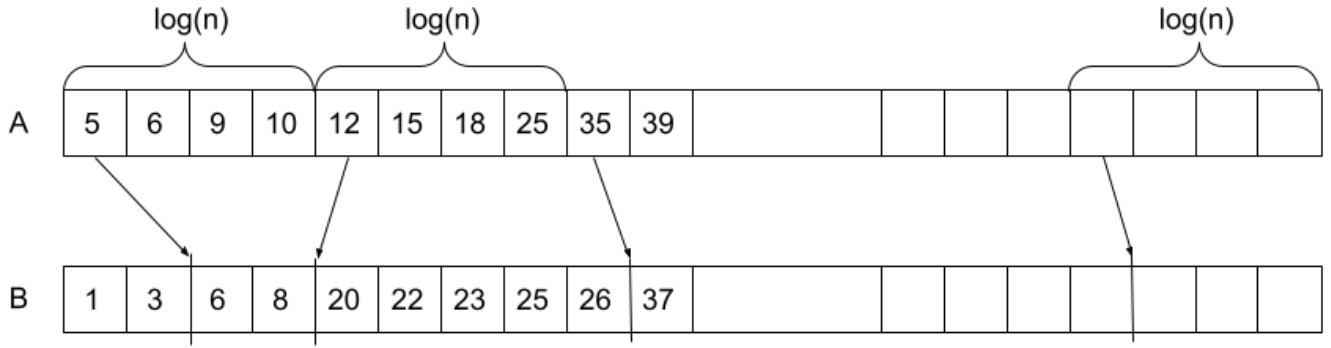


Сортировочки. Merge из MergeSort

- Самый простой способ посчитать rank – это сделать бинарный поиск для каждого элемента.
- Каждый rank – $O(\log n)$.
- Тогда Merge работает за $O(n \log n)$, а не $O(n)$.
- Глубина рекурсии $O(\log n)$.
- Это ведёт к $O(n \log^2 n)$ в MergeSort.

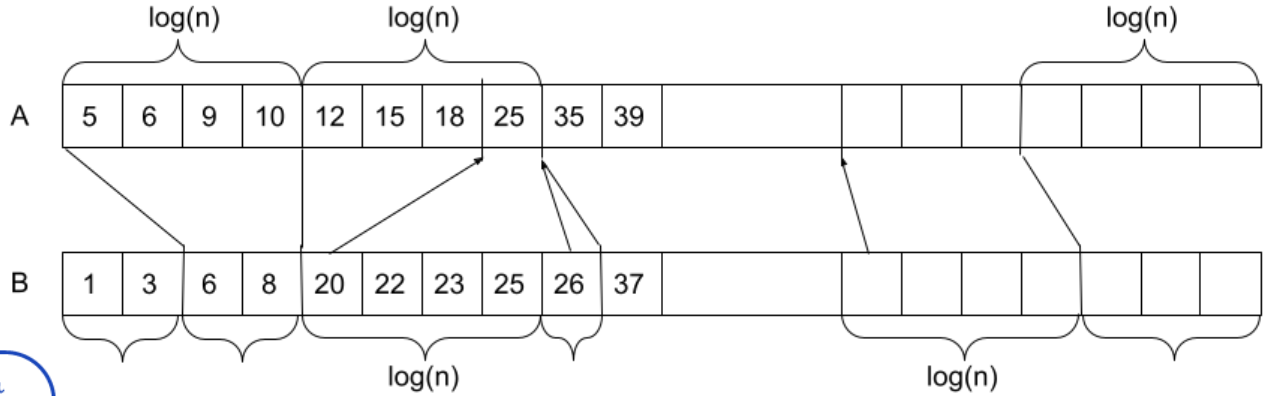
Оптимизация идеи

- Хочется сделать бин-поиск, но он стоит $O(\log n)$ на элемент
- Но мы можем себе позволить это сделать $O\left(\frac{n}{\log n}\right)$ раз.



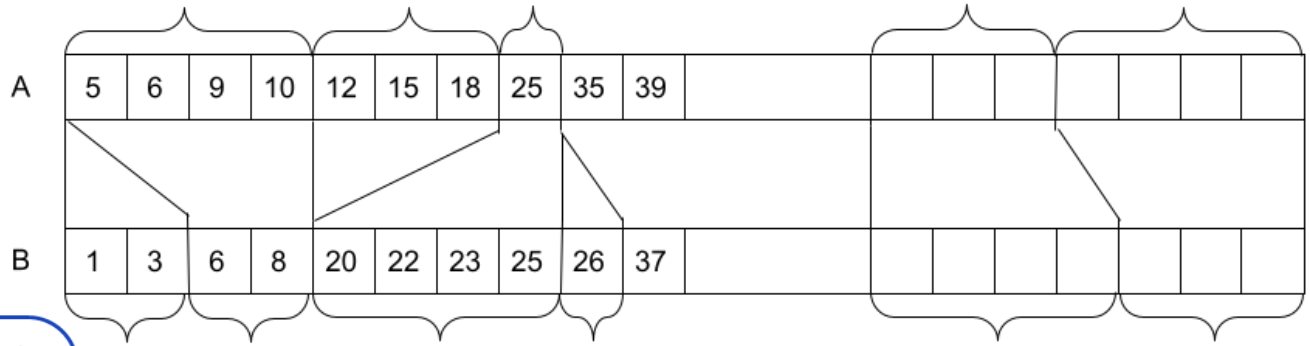
Оптимизация идеи

- Теперь мы знаем куда вставлять кусочки длины $O(\log n)$
- Если кусочки в массиве B меньше $O(\log n)$ – хорошо, если больше, то нужно проделать тоже самое с кусочками из B.



Оптимизация идеи

- Теперь у нас есть пары кусочков длины не более $O(\log n)$ и их не более $O(\frac{n}{\log n})$ штук.
- Мы можем теперь или насчитать rank, или параллельно слить.



MergeSort. Асимптотика.

- Merge.
  - Найти положение каждого из $O\left(\frac{n}{\log n}\right)$ кусочков A – $O(n)$ work и $O(\log n)$ span.
  - Найти кусочки из B – $O(n)$ work и $O(\log n)$ span.
  - Объединить пары кусочков – $O(n)$ work и $O(\log n)$ span.
  - Суммарно: $O(n)$ work и $O(\log n)$ span.
- MergeSort.
  - Глубина рекурсии $O(\log n)$.
  - $O(n \log n)$ work и $O(\log^2 n)$ span.

MergeSort $\longrightarrow$ Divide-and-Conquer

- MergeSort – стандартный пример на Divide-and-Conquer.
- Алгоритмы с Divide-and-Conquer достаточно просты к параллелизации.
- Ещё один такой пример – Batched Binary Search Tree.
- Нужно вставить m новых элементов в бинарное дерево поиска размера n

Batch-Parallel BST

```
insert(T, a):
    Tl, Tr = split(T, a[len(a) / 2])
    fork2join([&] {
        Tl = insert(Tl, a[:len(a) / 2])
    }, [&] {
        Tr = insert(Tr, a[len(a) / 2:])
    })
    return join(Tl, a[len(a) / 2], Tr)
```

Trees: AVL, Treap, RBTree, WBTrees.

Join – $O(\log n)$
Split – $O(\log n)$

Span: $O(\log n \cdot \log m)$

Work: $O(m \cdot \log(1 + \frac{n}{m}))$

Blelloch et al., Just Join Parallel Sets, 2016

Сортировочки. Bucket Sort + Digit Sort.

- Пусть все значения не превышают k.
- Бьём массив на куски размера k.
- Считаем в каждом блоке подсчётом за $O(k)$.
- Получим $O(\frac{n}{k})$ массивов длины $O(k)$, на которых надо посчитать частичные суммы, чтобы узнать новые позиции у элементов.
- $O(n)$ work и $O(k + \log n)$ span.
- Если значения не превосходят $n = \left(n^{1/\alpha}\right)^{\alpha}$:
  - α раз повторяем процедуру для $k = n^{1/\alpha}$;
  - $O(\alpha \cdot n)$ work и $O(\alpha \cdot \left(n^{1/\alpha} + \log n\right))$ span.

Работа с памятью

- Описанные алгоритмы на самом деле очень сильно используют создание и освобождение памяти. Потоки могут делать это одновременно.
- Нам нужны масштабируемые аллокаторы.
- Их не очень много. Советую tcmalloc или jemalloc.

Спасибо за внимание!

www.ifmo.ru

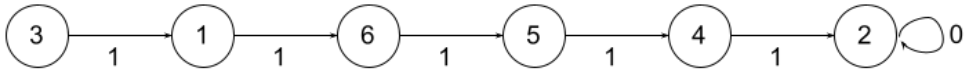
ITMO more than a UNIVERSITY

List Ranking

- Следующая задача является одной из важных для параллельных алгоритмов.
- Дан связный список как массив со ссылками на следующего.
- Надо посчитать расстояния от элемента до конца списка.
- Сначала мы рассмотрим алгоритм с Binary Jumping – тоже полезной техникой, но с $O(n \log n)$ work.
- Затем мы рассмотрим стандартный рандомизированный алгоритм с $O(n)$ work. Его можно дерандомизировать.

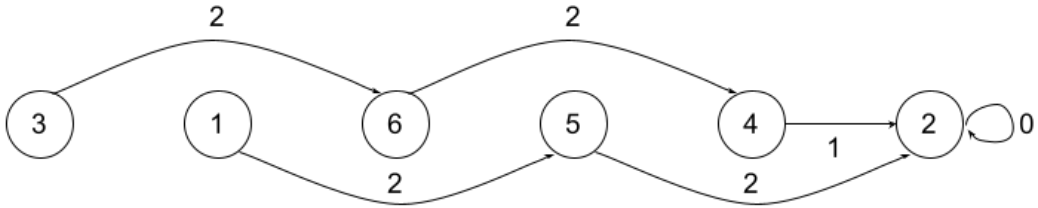
List Ranking. Binary Jumping.

```
rank(L):
    x = findEnd(L) // L[x] = -1
    D = {1, …, 1}
    D[x] = 0
    for it = 0 … log(n):
        D_2 = new int[len(L)], L_2 = new int[len(L)]
        parallel_for(0, n, [&] (int i) {
            D_2[i] = D[i] + D[L[i]]
        })
        parallel_for(0, n, [&] (int i) {
            L_2[i] = L[L[i]]
        })
        D = D_2, L = L_2
```



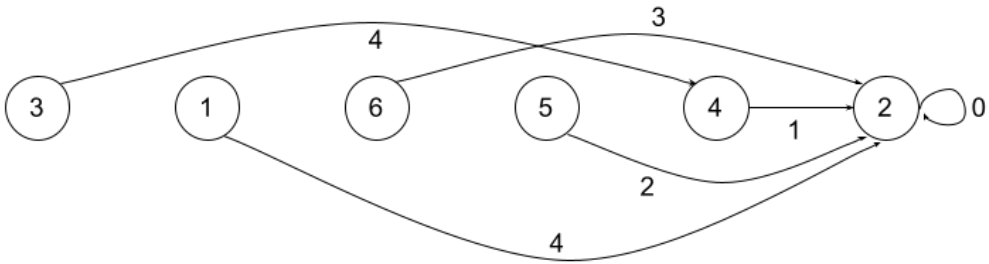
List Ranking. Binary Jumping.

```
rank(L):
    x = findEnd(L) // L[x] = -1
    D = {1, …, 1}
    D[x] = 0
    for it = 0 … log(n):
        D_2 = new int[len(L)], L_2 = new int[len(L)]
        parallel_for(0, n, [&] (int i) {
            D_2[i] = D[i] + D[L[i]]
        })
        parallel_for(0, n, [&] (int i) {
            L_2[i] = L[L[i]]
        })
        D = D_2, L = L_2
```



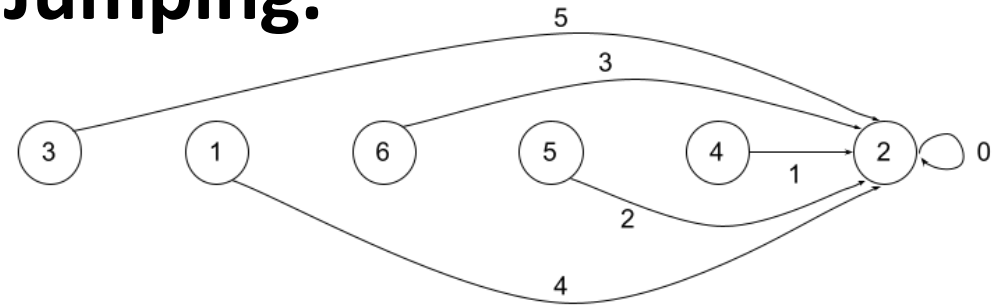
List Ranking. Binary Jumping.

```
rank(L):
    x = findEnd(L) // L[x] = -1
    D = {1, …, 1}
    D[x] = 0
    for it = 0 … log(n):
        D_2 = new int[len(L)], L_2 = new int[len(L)]
        parallel_for(0, n, [&] (int i) {
            D_2[i] = D[i] + D[L[i]]
        })
        parallel_for(0, n, [&] (int i) {
            L_2[i] = L[L[i]]
        })
        D = D_2, L = L_2
```



List Ranking. Binary Jumping.

```
rank(L):
    x = findEnd(L) // L[x] = -1
    D = {1, …, 1}
    D[x] = 0
    for it = 0 … log(n):
        D_2 = new int[len(L)], L_2 = new int[len(L)]
        parallel_for(0, n, [&] (int i) {
            D_2[i] = D[i] + D[L[i]]
        })
        parallel_for(0, n, [&] (int i) {
            L_2[i] = L[L[i]]
        })
        D = D_2, L = L_2
```



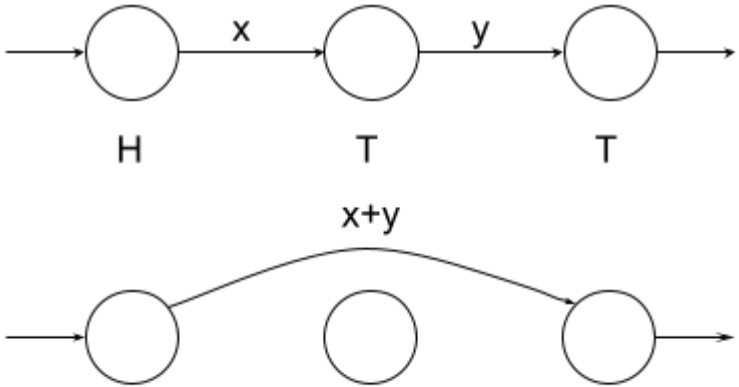
Work: $O(n \log n)$
Span: $O(\log^2 n)$

List Ranking

- У задачи есть достаточно очевидное решение за $O(n)$. (Попробуйте придумать сами :P)
- Предыдущее решение не work-optimal, а мы пытаемся этого добиться.
- Перейдём к рандомизированному решению за $O(n)$ work.
- Пусть у нас есть магический параллельный рандом – обычно используется специальная хеш-функция.

List Ranking. Randomized.

- Будет несколько итераций, после каждой из которых список уменьшается.
- На каждой итерации:
  - Вершинка кидает монетку.
  - Если у неё Head, а у соседа Tail, то соседа можно сжать.
  - Запоминаем все изменения и фильтруем живые вершины.
- Потом бежим по итерациям с конца, и восстанавливаем список с расстояниями.



List Ranking. Randomized.

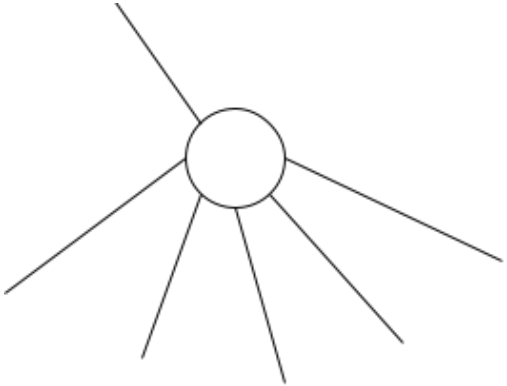
- Матожидание сжатых вершин на итерации - $\frac{n}{4}$.
- Каждую итерацию число вершин уменьшается где-то в константу.
- Значит, время работы – $O(n)$, а число итераций – $O(\log n)$.
- Каждая итерация потребляет $O(\log n)$ span, значит, у всего алгоритма $O(\log^2 n)$ span.
- Все оценки можно получить with high probability.

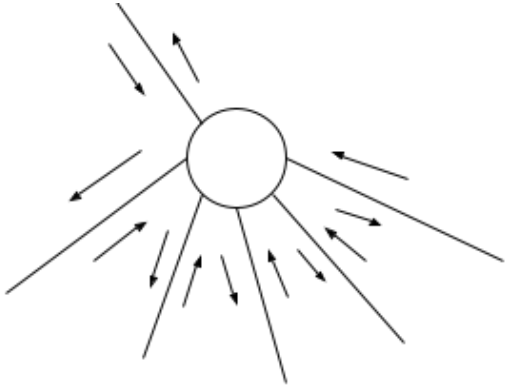
List Ranking ⟶ Eulerian Circuit

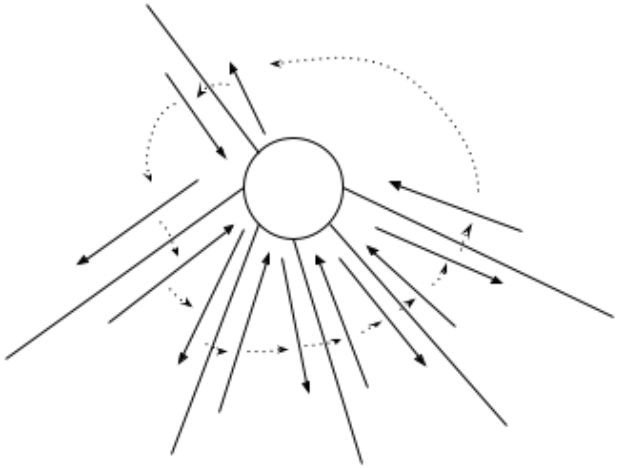
- Давайте поговорим о такой структуре данных как дерево.
- Дано подвешенное дерево – найти глубины всех вершин.
- Кажется, такое невозможно сделать быстро, особенно, когда дерево является линией.
- А магическим образом мы сведём эту задачу к List Ranking с помощью Эйлерова обхода.

Eulerian Circuit

- Раздваиваем рёбра и соединяем их а список “по кругу”.







Eulerian Circuit

- Чтобы подвесить дерево – надо разрезать одно ребро в списке.
- Дальше для каждого ребра считаем его номер в списке с помощью List Ranking – понимаем какое из пары рёбер-дубликатов идёт от корня, а какое к корню.
- Ребру от корня ставим “+1”, ребру к корню ставим “-1”.
- С помощью List Ranking на сумму знаем глубину каждой вершины.