


**Нижегородский государственный университет им. Н.И. Лобачевского**

**Институт информационных технологий, математики и механики**

Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

*ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ГРАФОВ*

# Лекция 5. Вычисление минимального остовного дерева

Пирова А.Ю.
Кафедра ВВиСП

# Содержание


- Предметные области
- Постановка задачи
- Последовательный Алгоритм Борувки
- Параллельный Алгоритм Борувки для общей памяти (GBBS)
- Параллельный Алгоритм Борувки для распределенной памяти (KAMSTA)
- Результаты экспериментов
- Заключение

# Постановка задачи

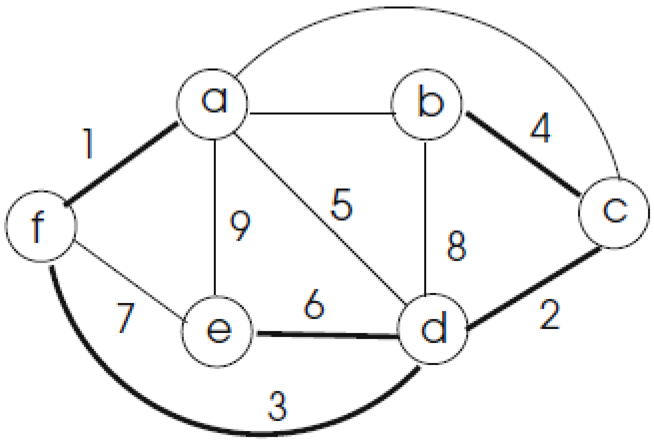
- Пусть дан связный неориентированный взвешенный граф $G = (V, E, w(e))$, $|V| = n, |E| = m,$ веса ребер $w$: $E \rightarrow \boldsymbol{R}^{+}$.
- Найти: ациклическое подмножество ребер исходного графа, соединяющее все вершины графа, такое, что сумма весов ребер будет минимальна:

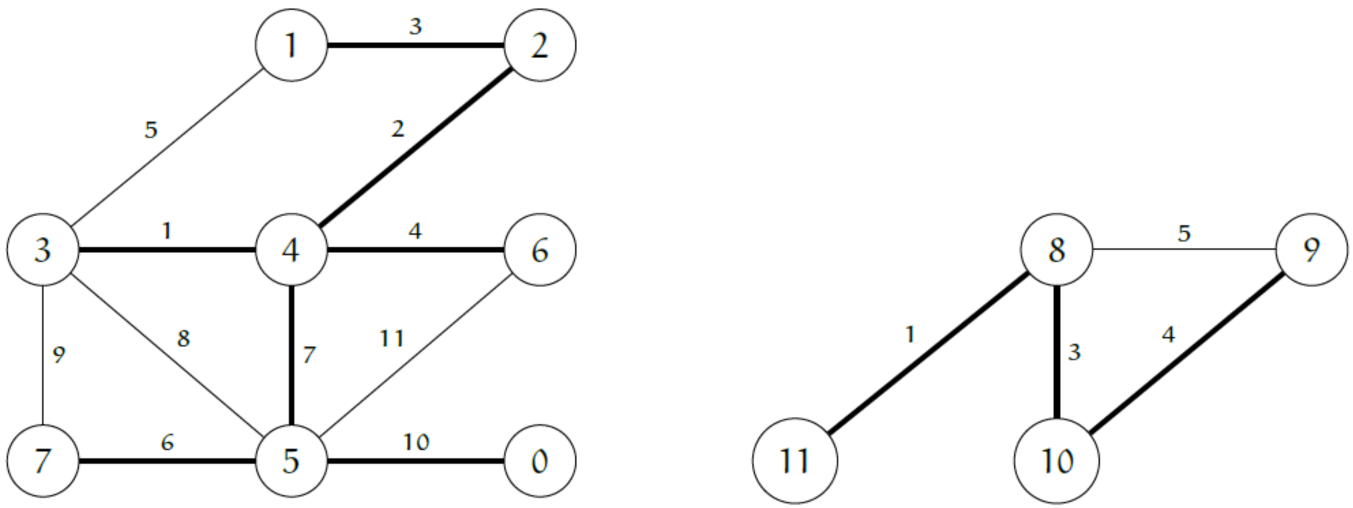
$$T^{*} = arg\min_{T \subseteq E} \sum_{e \, \epsilon \, T} w(e)$$

здесь $T$ – все возможные деревья-подграфы $G$, которые соединяют все вершины исходного графа (*остовные деревья*).

- $T^{*}$ называется *минимальным остовным деревом (minimum spanning tree, MST)*
- *Для не связного* графа можно найти *минимальный остовный лес (minimum spanning forest, MSF)*.

# Постановка задачи



# Применение

- Построение сети, соединяющей все объекты, с наименьшими затратами (электрические, телекоммуникационные, транспортные, компьютерные сети)
- Как этап приближенного решения задач: задача коммивояжера (NP-трудная), поиск максимального потока в сети, идеальное паросочетание минимального веса (содержит все вершины), кластеризация и др.
- Регистрация и сегментация изображений, feature extraction в компьютерном зрении
- …

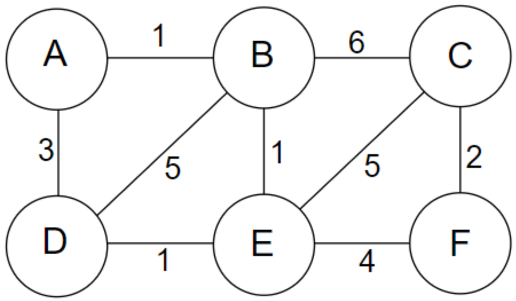
# Свойства минимального остовного дерева

- Связанные понятия:
  - *Разрез* (*cut*) графа $(S, V\backslash S)$ – разбиение множества вершин на два подмножества.
  - Ребро $(u, v)$ *пересекает разрез* $(S, V\backslash S)$ (crossing edge), если его концы лежат в разных частях разреза: $u \in S$ и $v \notin S$.

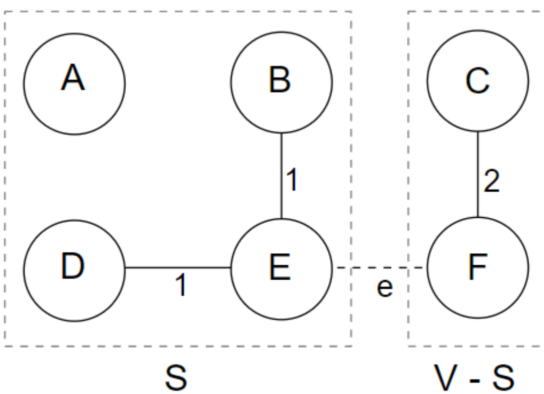
# Свойства минимального остовного дерева

- (**уникальность**) если веса всех ребер различны, то MST единственно
  → если есть ребра с одинаковым весом, то MST не единственно.
  (например, все ребра графа одинакового веса)
- Если веса графа неотрицательные, то MST – подграф минимального веса, содержащий все вершины графа.
- (**свойство цикла**) Для любого цикла $C$, если вес ребра $e$, $e \in C$ больше, чем вес любого другого ребра из $C$, то ребро $e$ не входит в MST.
- (**свойство разреза**) Пусть $A$ – подмножество ребер, которое входит в некоторое MST графа $G$. Для любого разреза $(S, V \backslash S)$, для которого ни одно из ребер $A$ не пересекает разрез, если $(u, v)$ – ребро минимального веса среди ребер, пересекающих разрез, $u \in S$ и $v \notin S$. Тогда ребро $(u, v)$ входит в MST графа $G$. (→ его можно добавить в $A$)
  → если все веса графа различны, то существует одно MST $T^*$ графа $G$, которое содержит ребро $(u, v)$.

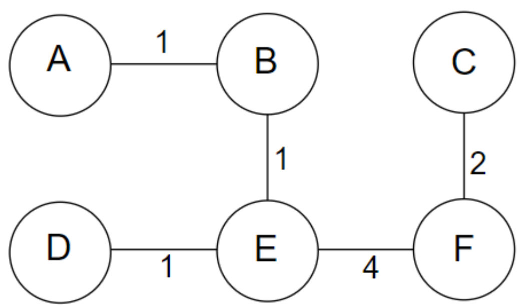
# Свойства минимального остовного дерева




The cut:




MST T:




Свойство разреза:
BC, EC, EF пересекают разрез
EF имеет минимальный вес →
EF входит в MST

# Алгоритмы нахождения MST

- Классические алгоритмы
  - Борувки (1926)
  - Краскала (1956)
  - Прима (1957). Он же – алгоритм Ярника (1930)
- Вычислительная сложность - $O(m \log n)$
- Жадный принцип

- Новые алгоритмы:
  - Яо (1975) – $O(m \log \log n)$
  - Черитон, Тарьян (1976)
  - Quick Kruskal, или алгоритм Краскала с фильтрацией (Осипов, Сандерс, Синглер, 2009) - $O(m + n \log n \log \log n)$

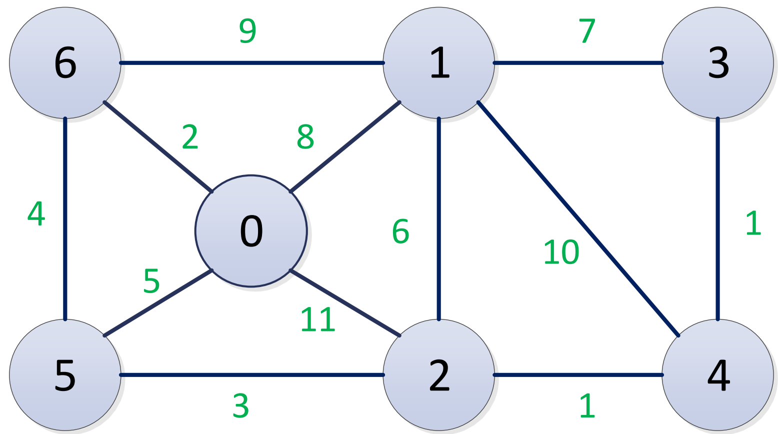
# Алгоритм Борувки

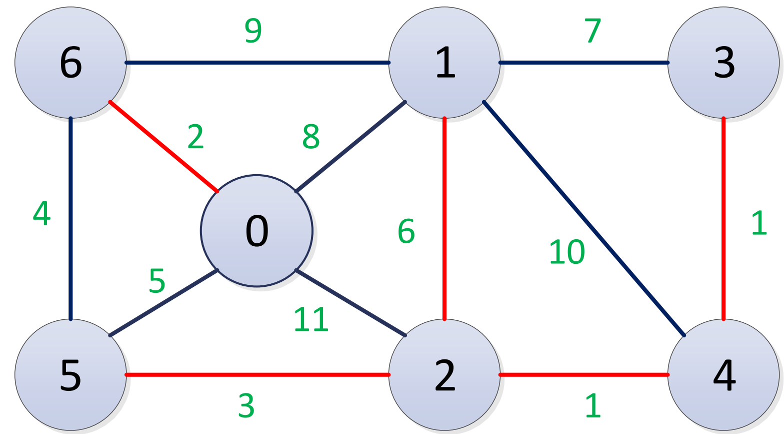
- Алгоритм использует свойство разреза графа.
- Идея: построим MST как объединение деревьев, «выращенных» из отдельных вершин графа путем добавления ребер минимального веса.

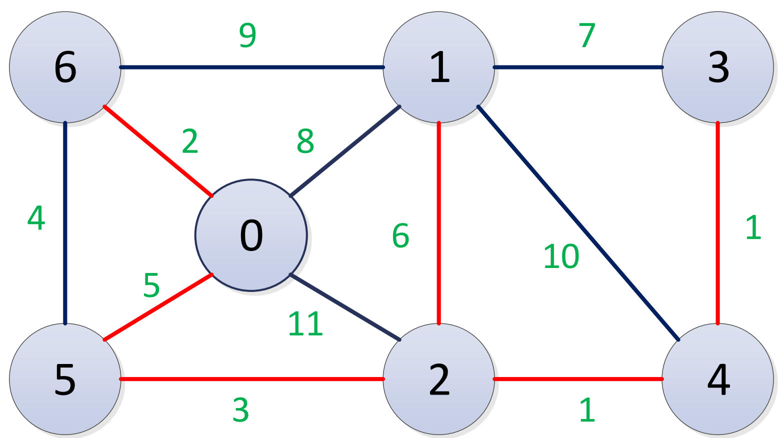
Вход: граф $G(V, E, W)$
Выход: MST $T(V_T, E_T)$

1. Пусть каждая вершина $v \in V$ – отдельная компонента связности. $T = \emptyset$.
2. Пока $T$ содержит меньше, чем $n - 1$ ребро (или пока больше одной компоненты связности):
   1. Для каждой компоненты связности $C_x$:
      1. Найти ребро минимального веса $(u, v)$, соединяющее $C_x$ с соседней компонентой связности $C_y$.
      2. Добавить $(u, v)$ в $T$.
      3. Объединить компоненты связности $C_x$ и $C_y$.

# Алгоритм Борувки. Пример



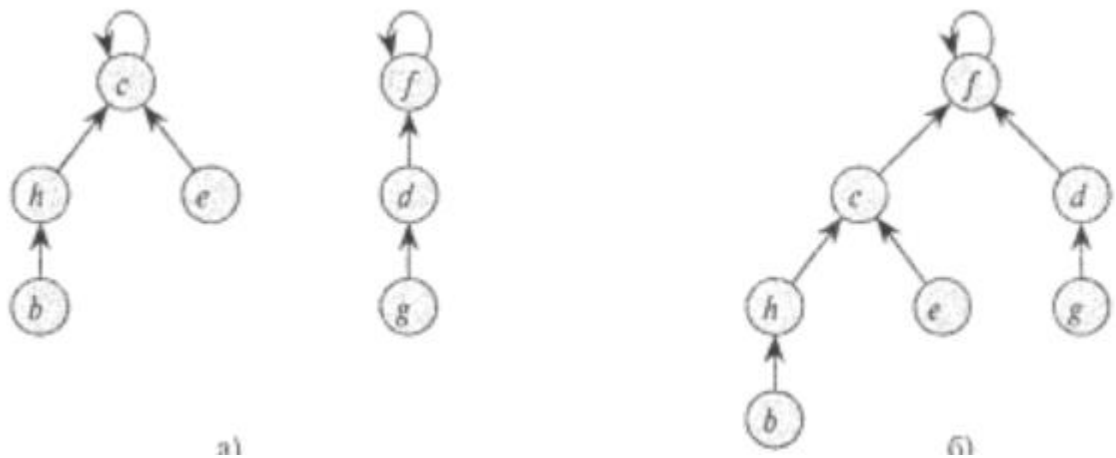
# Алгоритм Борувки

Вход: граф $G(V, E, W)$
Выход: MST $T(V_T, E_T)$

1. Пусть каждая вершина $v \in V$ – отдельная компонента связности. $T = \emptyset$.
2. Пока $T$ содержит меньше, чем $n - 1$ ребро (или пока больше одной компоненты связности):
   1. Для каждой компоненты связности $C_x$:
      1. Найти ребро минимального веса $(u, v)$, соединяющее $C_x$ с соседней компонентой связности $C_y$.
      2. Добавить $(u, v)$ в $T$.
      3. **Объединить компоненты связности $\boldsymbol{C_x}$ и $\boldsymbol{C_y}$.**
         1. Определить новые индексы вершин (relabeling)
         2. Объединить списки смежности вершин ИЛИ перенумеровать список ребер
         3. Удалить повторяющиеся ребра

# Структура данных Union-Find

- Структуры данных:
  - Граф хранится как список ребер (чаще всего) или в виде списков смежности
  - Для хранения информации о компонентах связности используются *разделенные множества* (disjoint sets, union-find).
  - Каждая компонента связности хранится как дерево. Корень дерева – «представитель» компоненты
  - **Базовые операции**:
    - Создание множества
    - Найти, какому множеству принадлежит вершина
    - Объединение множеств



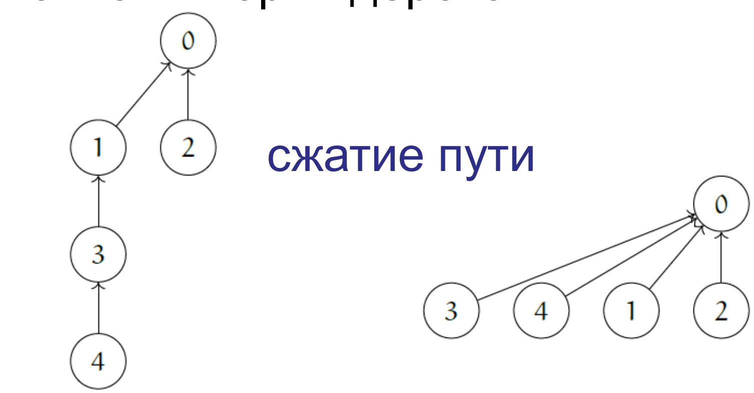
# Структура данных Union-Find

```
MAKE_SET(x)
    p[x] ← x
    rank[x] ← 0

UNION(x, y)
    LINK(FIND_SET(x), FIND_SET(y))

LINK(x, y)
    if rank[x] > rank[y]
        then p[y] ← x
        else p[x] ← y
             if rank[x] = rank[y]
                then rank[y] ← rank[y] + 1

FIND_SET(x)
    if x ≠ p[x]
        then p[x] ← FIND_SET(p[x])
    return p[x]
```

❑ Хранение:
p[x] – родитель в дереве-компоненте
rank[x] – верхняя граница высоты дерева-компоненты

❑ Объединение деревьев:

- Дерево с бОльшим rank становится родителем
- «Сжатие пути»: все вершины вдоль пути поиска родителя X становятся потомками корня дерева




(a) A union-find tree. (b) After find(4).

# Параллельный алгоритм базовый

Вход: граф $G(V, E, W)$
Выход: MST $T(V_T, E_T)$

1. Пусть каждая вершина $v \in V$ – отдельная компонента связности. $T = \emptyset$.
2. Пока $T$ содержит меньше, чем $n - 1$ ребро (или пока больше одной компоненты связности):
   1. **Параллельно** для каждой компоненты связности $C_x$:
      1. найдем ребро минимального веса $(u, v)$, соединяющее $C_x$ с соседней компонентой связности $C_y$. → Получим множество ребер $E_T$
   2. Удалить дубликаты в $E_T$. Добавить $E_T$ в $T$.
   3. Объединяем компоненты связности:
      1. **Параллельно** Определить новые индексы вершин (relabeling)
      2. **Параллельно** Объединить списки смежности вершин ИЛИ перенумеровать список ребер
      3. **Параллельно** Удалить повторяющиеся ребра

# Параллельный алгоритм, GBBS
# Шаг алгоритма Борувки

```
Parents[0, . . . , n) := 0
procedure BORŮVKA(n, E)                       ▷ E is aprefix of minimum weight inter-component edges
    Forest := {}
    while |E| > 0 do
        P[0, . . . , n) := (∞, ∞)             ▷ array of (weight , index) pairs for each vertex
        for i ∈ [0, |E|) in parallel do
            (u, v, w) := E[i]                  ▷ the i-th edge in E
            priorityWrite(&P[u], (w, i), <)    ▷ < lexicographically compares the (weight, index) pairs
            priorityWrite(&P[v], (w, i), <)
        for u ∈ [0, n) where P[u] ≠ (∞, ∞) in parallel do
            (w, i) := P[u]                     ▷ the index and weight of the MSF edge incident to u
            v := the neighbor of u along the E[i] edge
            if v > u and P[v] = (w, i) then    ▷ v also chose E[i] as its MSF edge; symmetry break
                Parents[u] := u                ▷ make u the root of a component
            else
                Parents[u] := v                ▷ otherwise v < u; join v's component
        Forest := Forest ∪ {edges that won on either endpoint in P}     ▷ add new MSF edges
        PointerJump(Parents)   сжатие пути     ▷ compress the parents array (see Section 3)
        E := map(E, fn (u, v, w) → return (Parents[u], Parents[v], w))  ▷ relabel edges
        E := filter(E, fn (u, v, w) → return u ≠ v)                     ▷ remove self-loops
    return Forest
```

# Параллельный алгоритм, GBBS
# Основной цикл

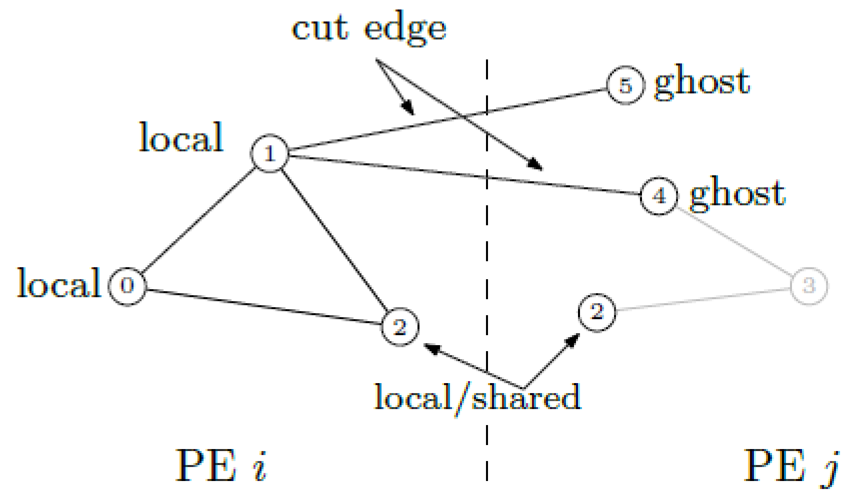
- Оптимизация - **Фильтрация**: основной алгоритм выполняется за несколько итераций, каждая – на подмножестве ребер минимального веса.

```
procedure MinimumSpanningForest(G(V, E, w))
    Forest := {}
    Rounds := 0
    vertexMap(V, fn u → Parents[u] = u)        ▷ initially each vertex is in its own component
    while G.numEdges() > 0 do
        T := select min(3n/2, m)-th smallest edge weight in G
        if Rounds = 5 then T := largest edge weight in G
        E_F := extractEdges(G, fn (u, v, w_uv) → return w_uv ≤ T)
        Forest := Forest ∪ Borůvka(|V|, E_F)
        packGraph(G, fn (u, v, w_uv) → return Parents[u] ≠ Parents[v])        ▷ remove self-loops
        Rounds := Rounds + 1
    return Forest
```

Изменить граф: удалить все ребра, не удовлетворяющие условию

# Алгоритм Борувки для распределенной памяти

- Авторы: Сандерс, Шимек, 2023
- https://github.com/mschimek/kamsta
- Хранение графа: распределенный массив ребер. Ребра упорядочены лексикографически
- Дополнительно на каждом процессе хранится массив первых в списке ребер с каждого процесса (для бинарного поиска)
  - Обозначения:
    - Ребро $e = (src(e), dist(e))$.
    - $V_i = \{src(e): e \in E_i\}$. – локальная вершина



# Алгоритм Борувки для распределенной памяти

- Базовая версия

локально

коммуникация

**Algorithm 1** High-level overview of our distributed Borůvka-MST algorithm. By $i$ we denote the rank of a PE. The set $T_i$ stores the MST edges.

**function** MST($G_i = (V_i, E_i)$)
  $G_i, T_i \leftarrow$ LOCALPREPROCESSING($G_i$)
  **while** $\sum |V_i| >$ threshold **do**
    $E_i^{\min} \leftarrow$ MINEDGES($G_i$)
    $L_i^{\text{local}}, T_i \leftarrow$ CONTRACTCOMPONENTS($E_i^{\min}, T_i$)
    $L_i^{\text{ghost}} \leftarrow$ EXCHANGELABELS($L_i^{\text{local}}, G_i$)
    $G_i' \leftarrow$ RELABEL($L_i^{\text{local}}, L_i^{\text{ghost}}, G_i$)
    $G_i \leftarrow$ REDISTRIBUTE($G_i'$)
  $T_i \leftarrow$ BASECASE($G_i, T_i$)
  **return** REDISTRIBUTEMST($T_i$)

построить MST из локальных ребер → останутся только ребра между разными процессами

Обменяться названиями вершин, локальных для другого процесса

Отсортировать ребра, убрать петли, разослать первое ребро (allgather)

$E_i^{min}$ - локальное ребро, $L_i^{local}$ - локальная компонента связности

# Алгоритм Борувки для распределенной памяти

- Базовая версия

**Algorithm 1** High-level overview of our distributed Borůvka-MST algorithm. By $i$ we denote the rank of a PE. The set $T_i$ stores the MST edges.

**function** MST($G_i = (V_i, E_i)$)
  $G_i, T_i \leftarrow$ LOCALPREPROCESSING($G_i$)
  **while** $\sum |V_i| >$ threshold **do**
    $E_i^{\min} \leftarrow$ MINEDGES($G_i$)
    $L_i^{\text{local}}, T_i \leftarrow$ CONTRACTCOMPONENTS($E_i^{\min}, T_i$)
    $L_i^{\text{ghost}} \leftarrow$ EXCHANGELABELS($L_i^{\text{local}}, G_i$)
    $G_i' \leftarrow$ RELABEL($L_i^{\text{local}}, L_i^{\text{ghost}}, G_i$)
    $G_i \leftarrow$ REDISTRIBUTE($G_i'$)
  $T_i \leftarrow$ BASECASE($G_i, T_i$)
  **return** REDISTRIBUTEMST($T_i$)

локально

коммуникация

На каждом процессе дублировать граф, продолжить вычисления

Собрать дерево

# Алгоритм Борувки для распределенной памяти

- Алгоритм с фильтрацией

начало

**function** FILTER-MST($G_i = (V_i, E_i)$)
  $G_i, T_i \leftarrow$ LOCALPREPROCESSING($G_i$)
  REC-FILTER-MST($G_i, T_i, P$)
  **return** (REDISTRIBUTEMST($T_i$))

**function** FILTER($G_i = (V_i, E_i), P$)
  $L_i^{\text{local}} \leftarrow$ REQUESTLABELS($V_i, P$)
  $L_i^{\text{ghost}} \leftarrow$ EXCHANGELABELS($L_i^{\text{local}}, G_i$)
  $E_i' \leftarrow$ RELABEL($L_i^{\text{local}}, L_i^{\text{ghost}}, G_i$)
  $E_i'' \leftarrow \{(u,v) \in E_i' \mid u \neq v\}$
  **return** REDISTRIBUTE($(V_i, E_i'')$)

**function** REC-FILTER-MST($G_i = (V_i, E_i), T_i, P$)
  **if** ISSPARSE($G_i, |P|$) **then**
    **return** MST($G_i, P$)
  $w_{\text{pivot}} \leftarrow$ PIVOTSELECTION($G_i$)
  $E_i^{\leq} \leftarrow \{ (u,v,w) \in E_i \mid w \leq w_{\text{pivot}} \}$
  $E_i^{>} \leftarrow \{ (u,v,w) \in E_i \mid w > w_{\text{pivot}} \}$
  $T_i \leftarrow$ REC-FILTER-MST($(V_i, E_i^{\leq}), T_i, P$)
  $(V_i', E_i^{>'}) \leftarrow$ FILTER($E_i^{>}, P$)
  **return** REC-FILTER-MST($(V_i', E_i^{>'}), T_i, P$)

Запрашиваем представителя компоненты связности для каждой вершины $V_i$ из распределенного массива $P$

Случайно выбранные ребра сортируются с помощью алгоритма распределенной сортировки. Рассылается медиана $w_{pivot}$

# Результаты экспериментов

- Влияние локальной предобработки

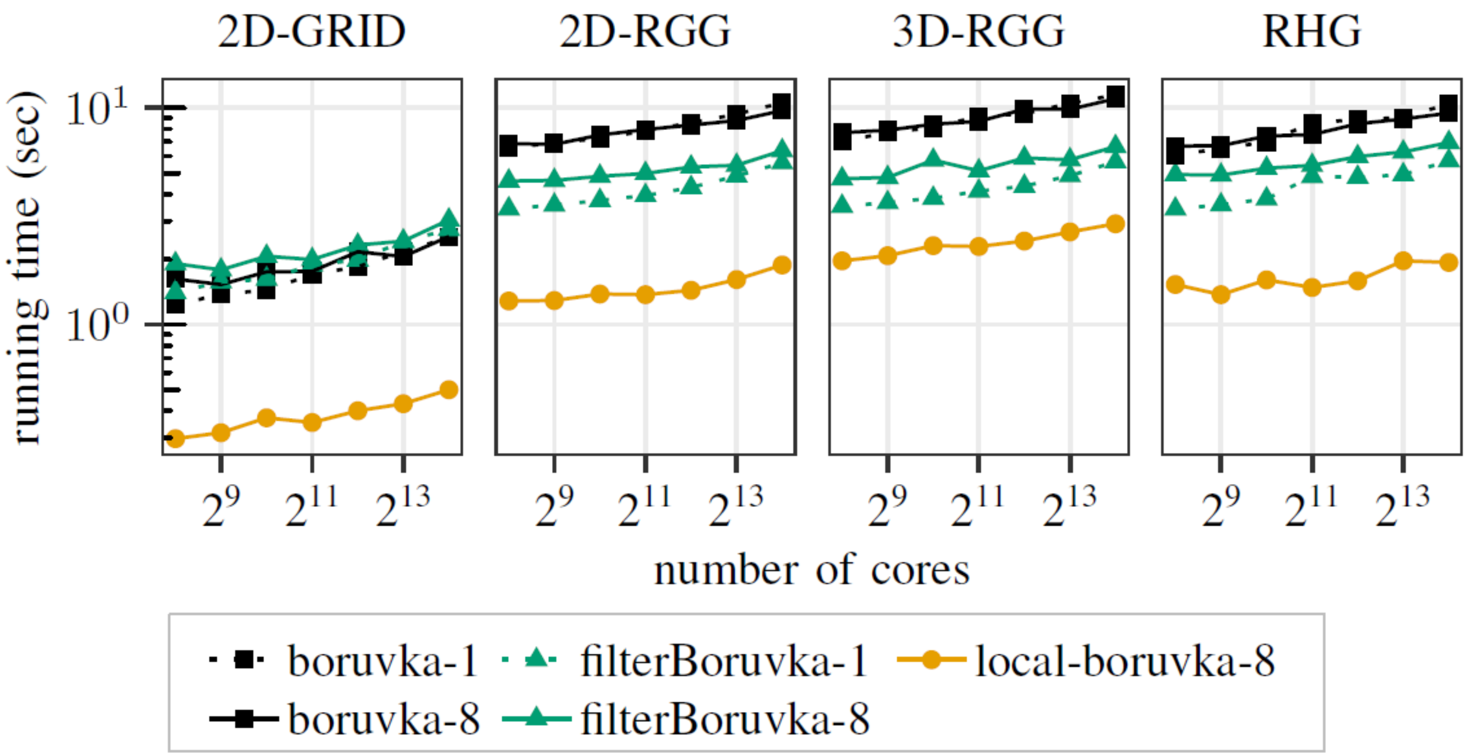



Fig. 4. Running time of our algorithms without local preprocessing on highly-local graphs with $2^{17}$ vertices and $2^{23}$ edges per core. Our fastest variant with local preprocessing enabled – `local-boruvka-8` – is given as a baseline.

# Результаты экспериментов

- Время работы алгоритма по фазам

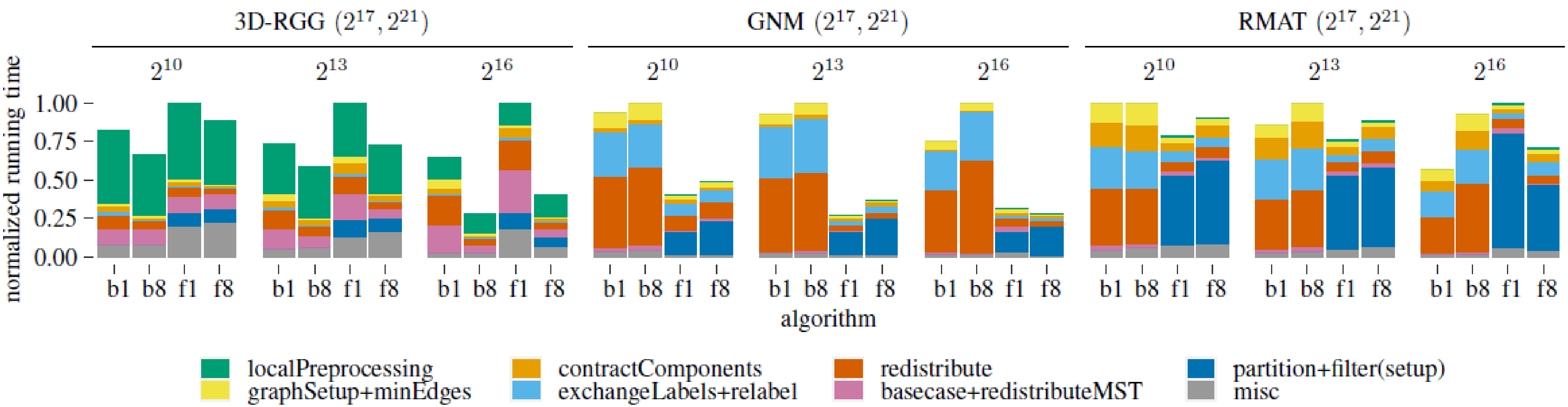



Fig. 6. Normalized running times to the range $[0, 1]$ of different steps of our algorithms with respect to the slowest variant in each graph×number-of-cores configuration.

Для RMAT и GNM локальная предобработка была пропущена,
Большую часть времени заняли коммуникации между процессами

# GBBS

## VertexSubset Interface

| | | Work | Depth |
|---|---|---|---|
| size | : unit $\rightarrow$ int | $O(1)$ | $O(1)$ |
| vertexMap | : (vtxid$\rightarrow$ unit) $\rightarrow$ unit | | |
| vertexMapVal | : (vtxid$\rightarrow$ E) $\rightarrow$ E vset | $O(\lvert U \rvert)$ | $O(\log n)$ |
| vertexFilter | : (vtxid$\rightarrow$ bool) $\rightarrow$ vset | | |
| addToSubset | : (vset $*$ vtxid sequence) $\rightarrow$ unit | $O(1)$ amortized | $O(\log n)$ |

## Bucketing Interface

| | | Work | Depth |
|---|---|---|---|
| makeBuckets | : int $*$ (identifier $\rightarrow$ bktid) $*$ bktorder $\rightarrow$ buckets | $O(n)^{\dagger}$ | $O(\log n)^{\ddagger}$ |
| getBucket | : (bktid $*$ bktid) $\rightarrow$ bktdest | $O(1)$ | $O(1)$ |
| nextBucket | : buckets $\rightarrow$ (bktid, identifier sequence) | presented in Theorem 4.1 | $O(\log n)^{\ddagger}$ |
| updateBuckets | : buckets $*$ (identifier, bktdest) sequence $\rightarrow$ unit | | |

# GBBS

| Vertex Interface | | | Work | Depth |
|---|---|---|---|---|
| *Neighborhood operators:* | map | : (edge → unit) → unit | $O(\lvert N(v)\rvert)$ | $O(\log n)$ |
| | reduce | : (edge → R) ∗ R monoid → R | | |
| | scan | : (edge → R) ∗ R monoid → R | | |
| | count | : (edge → bool) → int | | |
| | filter | : (edge → bool) → edge sequence | | |
| | pack | : (edge → bool) → unit | | |
| | iterate | : (edge → bool) → unit | $O(d_{it})$ | $O(d_{it})$ |
| | i-th | : int → edge | $O(1)$ | $O(1)$ |
| | degree | : unit → int | | |
| | getNeighbors | : unit → nghlist | | |
| *Vertex-Vertex operators:* | intersection | : (nghlist ∗ nghlist) → int | $O(l \log(h/l + 1))$ | $O(\log n)$ |
| | union | : (nghlist ∗ nghlist) → int | | |
| | difference | : (nghlist ∗ nghlist) → int | | |

## Graph Interface

| | Operator | Type | Work | Depth |
|---|---|---|---|---|
| *Graph operators:* | numVertices | : unit → int | $O(1)$ | $O(1)$ |
| | numEdges | : unit → int | | |
| | getVertex | : int → vertex | | |
| | filterGraph | : (edge → bool) → graph | $O(n+m)$ | $O(\log n)$ |
| | packGraph | : (edge → bool) → unit | | |
| | extractEdges | : (edge → bool) → edge sequence | | |
| | contractGraph | : int sequence → graph | $O(n+m)^{\dagger}$ | $O(\log n)^{\ddagger}$ |
| *VertexSubset operators:* | edgeMap | : vset ∗ (edge → bool) ∗ (vtxid → bool) → vset | $O\left(\sum_{u \in U} d(u)\right)$ | $O(\log n)$ |
| | edgeMapVal | : vset ∗ (edge → O option) ∗ (vtxid → bool) → O vset | | |
| | srcReduce | : vset ∗ (edge → O) ∗ O monoid ∗ (vtxid → bool) → O vset | $O\left(\lvert U \rvert + \sum_{u \in U'} d(u)\right)$ | $O(\log n)$ |
| | srcCount | : vset ∗ (edge → bool) ∗ (vtxid → bool) → int vset | | |
| | srcPack | : vset ∗ (edge → bool) ∗ (vtxid → bool) → int vset | | |
| | nghReduce | : vset ∗ (edge → R) ∗ R monoid ∗ (vtxid → bool) ∗ (R → O option) → O vset | $O\left(\sum_{u \in U'} d(u)\right)^{\dagger}$ | $O(\log n)^{\ddagger}$ |
| | nghCount | : vset ∗ (edge → bool) ∗ (vtxid → bool) ∗ (int → O option) → O vset | | |

# Литература


1. Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Алгоритмы: построение и анализ, 3-е издание. –М.: «Вильямс», 2013. –1328 с.
2. Dhulipala L., Blelloch G. E., Shun J. Theoretically efficient parallel graph algorithms can be fast and scalable //ACM Transactions on Parallel Computing (TOPC). – 2021. – Т. 8. – №. 1. – С. 1-70.
3. Sanders P., Schimek M. Engineering Massively Parallel MST Algorithms //arXiv preprint arXiv:2302.12199. – 2023.
4. Erciyes K. Guide to graph algorithms: sequential, parallel and distributed. – Springer, 2018.

# Контакты

Нижегородский государственный университет

http://www.unn.ru

Институт информационных технологий, математики и механики

http://www.itmm.unn.ru

Пирова А.Ю.

anna.pirova@itmm.unn.ru