High-Performance Graph & Data Analytics

Spring 2022

An efficient representation for sparse graphs:

Compressed Sparse Row

Bosisio Andrea – Karra Salvatore Gabriele

Goal

Implementation of a graph data structure for *sparse* graphs

- **Efficiently population:**

  `populate(edge_list)`

- **Fast traversing:**

  `get_neighbors(vertex)`

- **Limited memory usage**

Why CSR?

Brief comparison with other data structures

- **Adjacency matrix**
  - Too much memory: $O(|V|^2)$
- **Adjacency List**
  - It was the example implementation
  - Adding a neighbor is not efficient (dynamic memory allocation)
- **COO**
  - A lot repeated information
  - Neighbors iteration: $O(|E|)$: we don't know where each vertex starts

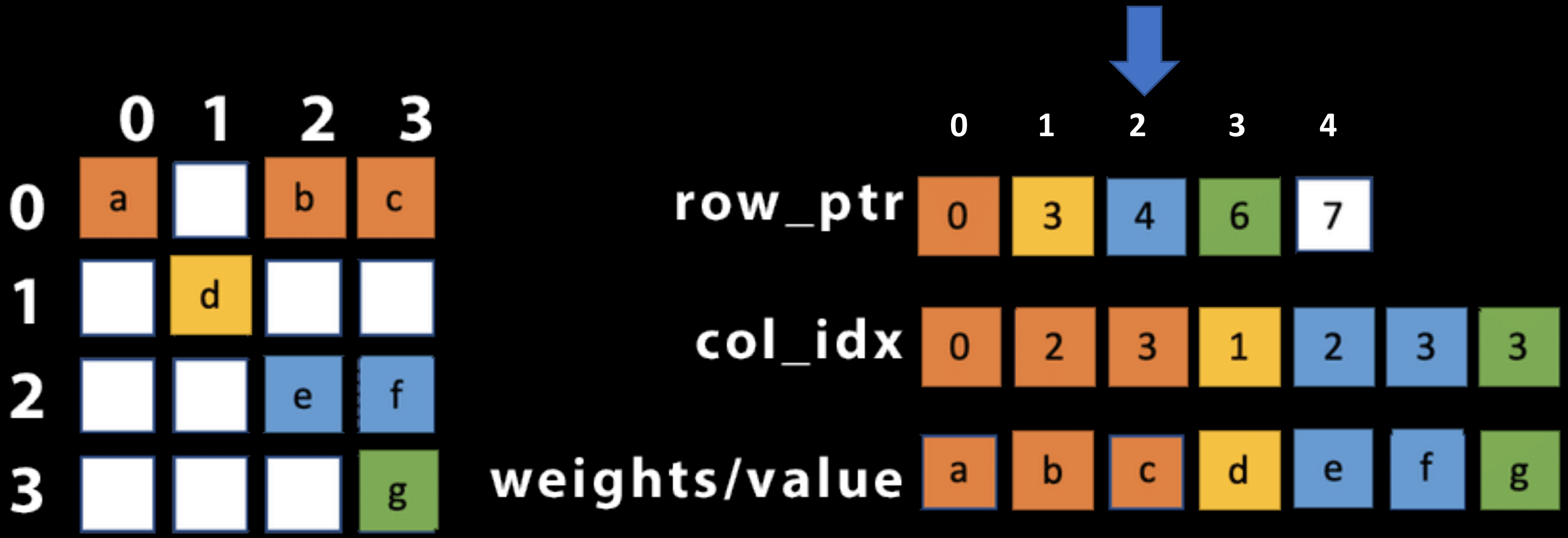
Why CSR?

Advantages of using this data structure

- **Smart information storage**
  - No duplicated data
  - It exploit the fact that data are sorted

  → good space complexity (see later…)
- **Fast out-neighbors iteration**
  - Direct access to the first neighbor
  - Then iteration is linear:
    - If a vertex has N_n neighbors → $O(N_n)$
    - In general → $O(|V|)$

How does CSR work?

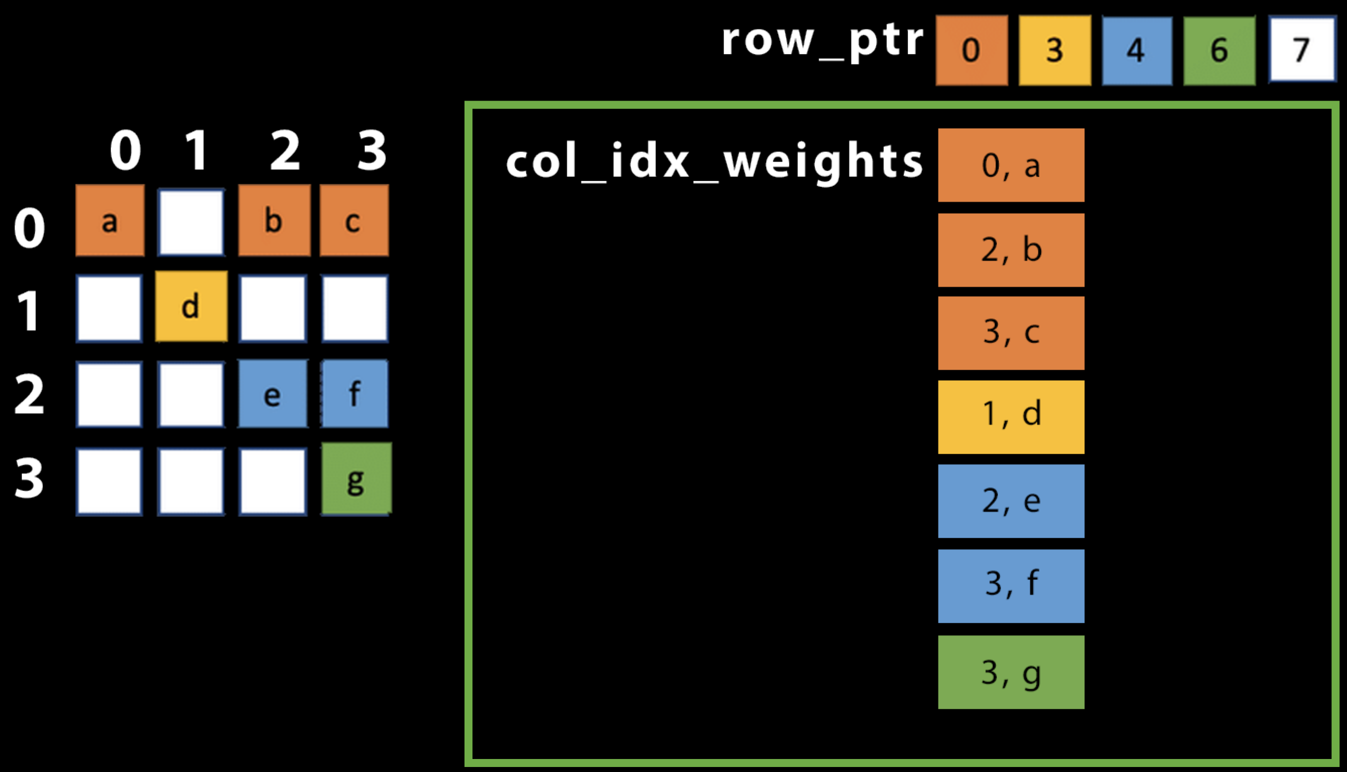


What we need to store?

	0	1	2	3	4			
row_ptr	0	3	4	6	7			$O(\lvert V \rvert)$
col_idx	0	2	3	1	2	3	3	$O(\lvert E \rvert)$
weights/value	a	b	c	d	e	f	g	$O(\lvert E \rvert)$

TOT: $O(|V|+|E|)$

A first optimization



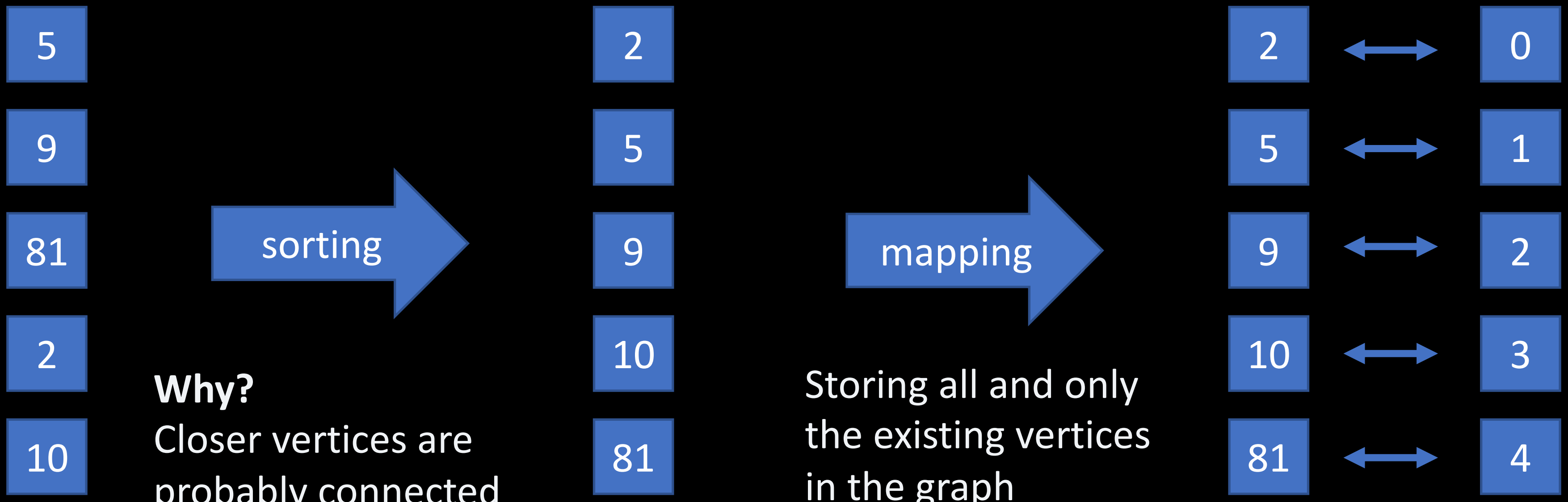
Why?
`col_idx` and `weights` are always accessed together:

- Reducing conflicts
- Improve spatial locality (operate on data in the same cache block)

→ Introducing `col_idx_weights` pair

Data pre-processing in `load_graph()`

Handling graphs with no sequential vertex indexes and "isolated" vertices



Why?
Closer vertices are probably connected → cache hits

Storing all and only the existing vertices in the graph

Why bidirectional?
Printing `dist[i]` with `i` of the original "domain"

https://github.com/andreabosisio/hpgda2022_data-structures/blob/main/include/utils.h

Populating the CSR
first problem

The edge list in input is NOT sorted and the CSR assumes that edges are sorted row by row from an adjacency matrix

We NEED to sort the edge list w.r.t. the first element of the tuple <node_from, node_to, weight>

We need to find a proper sorting algorithm…

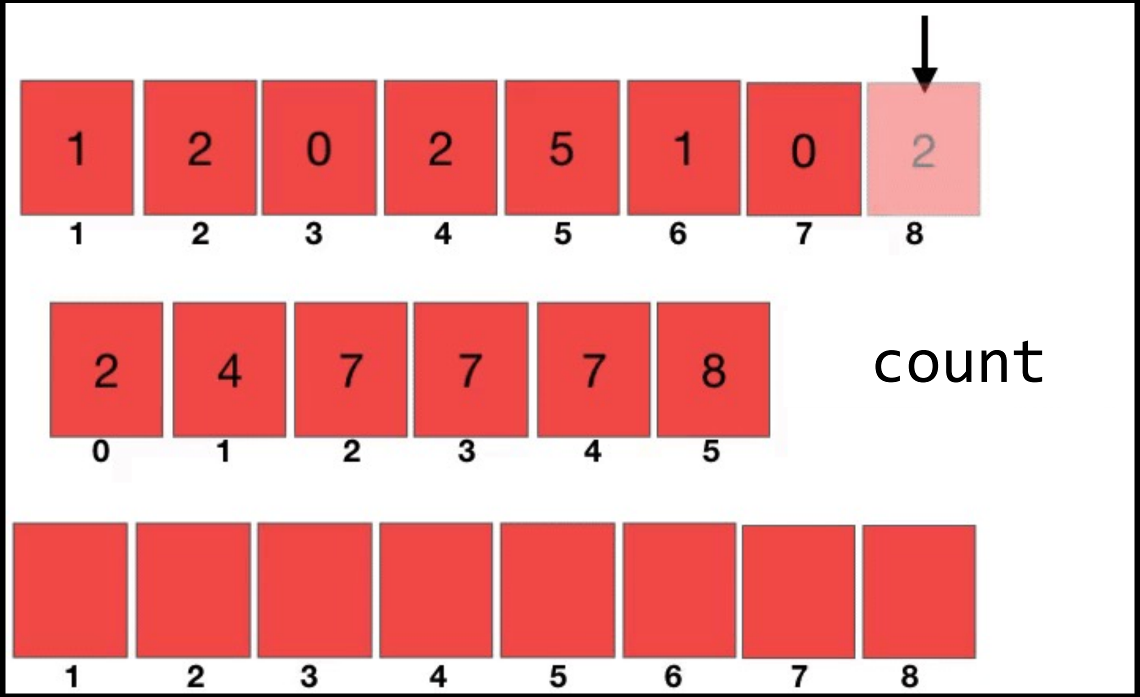
Populating the CSR
a smart solution

row_ptr	0	3	4	6	7

This looks like something we have already seen in a famous sorting algorithm…

The `count` array of the *counting sort* does the same thing!

Counting Sort



We exploit this algorithm to:

- Find the `row_ptr` array that is basically the `count` array
- Building the `col_idx_weights` array which is now sorted w.r.t. the starting vertex (i.e. `node_from`)

Let's now see in detail the code...

populate()

#1. Counting the number of neighbors for each vertex

PARALLEL VERSION

```
#pragma omp parallel for reduction
(+:row_ptr[:(num_vertices+2)])

for(uint64_t i = 0; i < num_edges; i++)
    row_ptr[std::get<0>(e_list[i]) + 1]++;
```

Time Complexity: $O\left(\frac{|E|}{P} + P\right)$

where P is the number of partitions
(i.e. threads)

SERIAL VERSION

```
for (uint64_t n = 0; n < num_edges; ++n)
    row_ptr[std::get<0>(e_list[n]) + 1]++;
```

Time Complexity: $O(|E|)$

populate()

#2. Computing the cumulative sum & Copying row_ptr into count

```
// cumulative sum
std::partial_sum(row_ptr, row_ptr + (num_vertices+2), row_ptr);

// copying row_ptr into count which will be used for the counting sort
std::copy(row_ptr + 1, row_ptr + 1 + num_vertices, count);
```

Time Complexity: $O(|V| + |V|) = O(|V|)$

Drawback: we need to duplicate data on the heap
since count will be decremented

populate()

#3. Sorting and filling the `col_idx_weight` array

PARALLEL VERSION

```
initLocks();
#pragma omp parallel {
std::tuple<uint64_t, uint64_t, double> curr_edge;
uint64_t curr_vertex, new_pos;

#pragma omp for
for (uint64_t i = 0; i < num_edges; i++) {
     curr_edge = e_list[i]; curr_vertex = std::get<0>(curr_edge);
     // locking on curr_vertex
     omp_set_lock(&count_locks[curr_vertex]);
     new_pos = count[curr_vertex] – 1; count[curr_vertex]––;
     // unlocking curr_vertex
     omp_unset_lock(&count_locks[curr_vertex]);

     col_idx_weight[new_pos] =
     std::make_pair(std::get<1>(curr_edge),
std::get<2>(curr_edge));
}}

// continuing …
```

SERIAL VERSION

```
std::tuple<uint64_t, uint64_t, double> curr_edge;
uint64_t curr_vertex, new_pos;

for (uint64_t i = 0; i < num_edges; i++)
{
     curr_edge = e_list[i];
     curr_vertex = std::get<0>(curr_edge);

     new_pos = count[curr_vertex] – 1;
     count[curr_vertex]––;

     col_idx_weight[new_pos] =
     std::make_pair(std::get<1>(curr_edge),
     std::get<2>(curr_edge));
}
```

Time Complexity: $\mathrm{O}(|E|)$

populate()

#3. (CONT'D) Sorting and filling the col_idx_weight array

PARALLEL VERSION

```
initLocks();
#pragma omp parallel {
std::tuple<uint64_t, uint64_t, double> curr_edge;
uint64_t curr_vertex, new_pos;

#pragma omp for
for (uint64_t i = 0; i < num_edges; i++) {
     curr_edge = e_list[i]; curr_vertex = std::get<0>(curr_edge);
     // locking on curr_vertex
     omp_set_lock(&count_locks[curr_vertex]);
     new_pos = count[curr_vertex] – 1; count[curr_vertex]--;
     // unlocking curr_vertex
     omp_unset_lock(&count_locks[curr_vertex]);

     col_idx_weight[new_pos] = std::make_pair(std::get<1>(curr_edge),
std::get<2>(curr_edge));
}}

// destroying locks and sorting neighbors
#pragma omp parallel for
for (uint64_t i = 0; i < num_vertices; i++){
     std::sort(col_idx_weight + row_ptr[i], col_idx_weight +
row_ptr[i+1]);
     omp_destroy_lock(&count_locks[i]);
}
```

Problem: in the parallel version of the counting sort the new_pos of a neighbor of curr_vertex depends on the order in which the threads access the edge list. Thus, BFS and DFS sums are different for each iteration.

Solution: sort the neighbors of every vertex in a predefined order (e.g. ascending order)

populate()

#3. (CONT'D) Sorting and filling the col_idx_weight array

PARALLEL VERSION

```
initLocks();
#pragma omp parallel {
std::tuple<uint64_t, uint64_t, double> curr_edge;
uint64_t curr_vertex, new_pos;

#pragma omp for
for (uint64_t i = 0; i < num_edges; i++) {
     curr_edge = e_list[i]; curr_vertex = std::get<0>(curr_edge);
     // locking on curr_vertex
     omp_set_lock(&count_locks[curr_vertex]);
     new_pos = count[curr_vertex] – 1; count[curr_vertex]--;
     // unlocking curr_vertex
     omp_unset_lock(&count_locks[curr_vertex]);

     col_idx_weight[new_pos] = std::make_pair(std::get<1>(curr_edge),
std::get<2>(curr_edge));
}}

// destroying locks and sorting neighbors
#pragma omp parallel for
for (uint64_t i = 0; i < num_vertices; i++){
     std::sort(col_idx_weight + row_ptr[i], col_idx_weight +
row_ptr[i+1]);
     omp_destroy_lock(&count_locks[i]);
}
```

Time Complexity:

$$O\left(\frac{|E|}{P} + \frac{|V|}{P}|V|\cancelto{C}{log(|V|)}\right)$$

But the sort is done on an array that contains the neighbors of a single vertex. In a **sparse graph** we have that the number of neighbors of a vertex is $\ll |V|$. Therefore,

$$O\left(\frac{|E|}{P} + \frac{|V|}{P}\right)$$

where P is the number of partitions (i.e. threads)

populate()

Total time complexity

PARALLEL VERSION

$$\mathbf{O}\left(\frac{|E|}{P}+P\right)+O(|V|)+\mathbf{O}\left(\frac{|E|}{P}+\frac{|V|}{P}\right)$$

$$=\mathbf{O}\left(\frac{|E|}{P}+P+|V|+\frac{|V|}{P}\right)$$

With $|\mathbf{E}| \gg |\mathbf{V}|$ **:** $\mathbf{O}\left(\frac{|E|}{P}+P\right)$

SERIAL VERSION

$$\mathbf{O}(|E|)+\mathbf{O}(|V|)+\mathbf{O}(|E|)$$

$$=O(|V|+|E|)$$

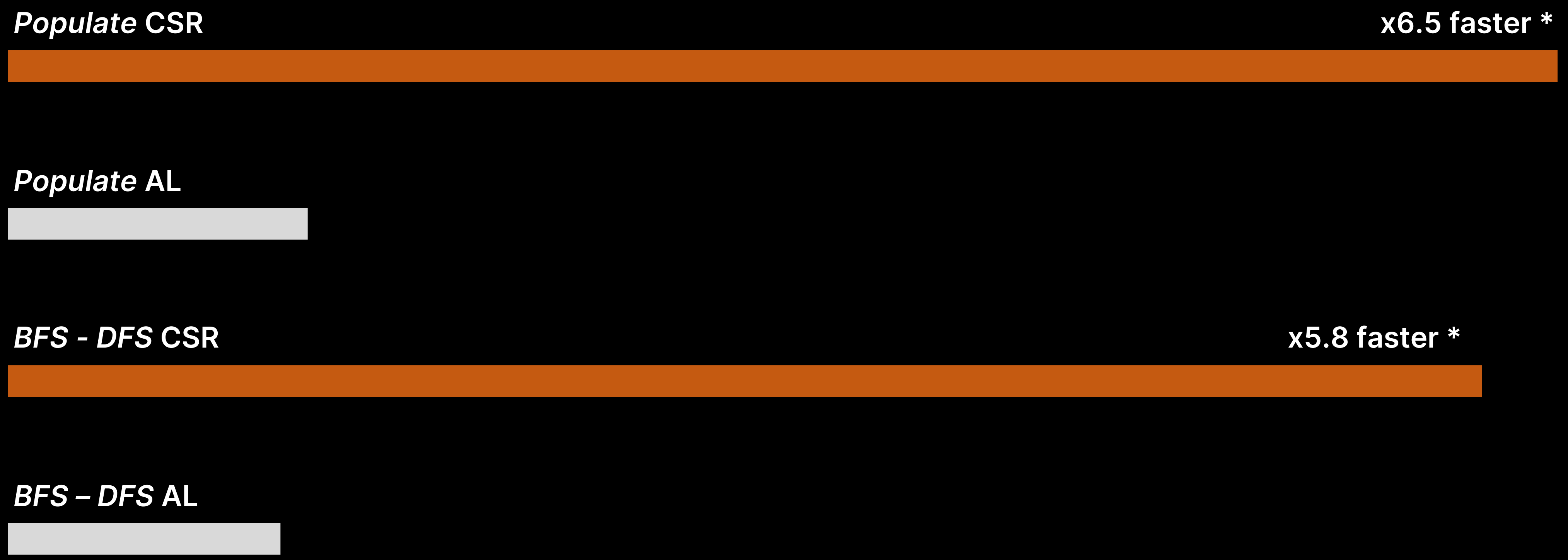
With $|\mathbf{E}| \gg |\mathbf{V}|$ **:** $\mathbf{O}(|E|)$

get_neighbors()

```
CSRIter get_neighbors(uint64_t vertex_idx){
     return CSRIter(col_idx_weight + row_ptr[vertex_idx], col_idx_weight + row_ptr[vertex_idx + 1]);
}


class CSRIter {
     class iterator {
          public:
               iterator(std::pair<uint64_t, double> *ptr) : ptr(ptr) {}
               iterator operator++(){ ++ptr; return *this; }
               bool operator!=(const iterator &other) { return ptr != other.ptr; }
               const std::pair<uint64_t, double> &operator*(){return *ptr;};

          private:
               std::pair<uint64_t, double> *ptr;
     };

     private:
          std::pair<uint64_t, double> *begin_ptr;
          std::pair<uint64_t, double> *end_ptr;

     public:
          CSRIter(std::pair<uint64_t, double> *begin_ptr, std::pair<uint64_t, double> *end_ptr) :
begin_ptr(begin_ptr), end_ptr(end_ptr) {}

          iterator begin() const { return iterator(begin_ptr); }

          iterator end() const { return iterator(end_ptr); }
};
```

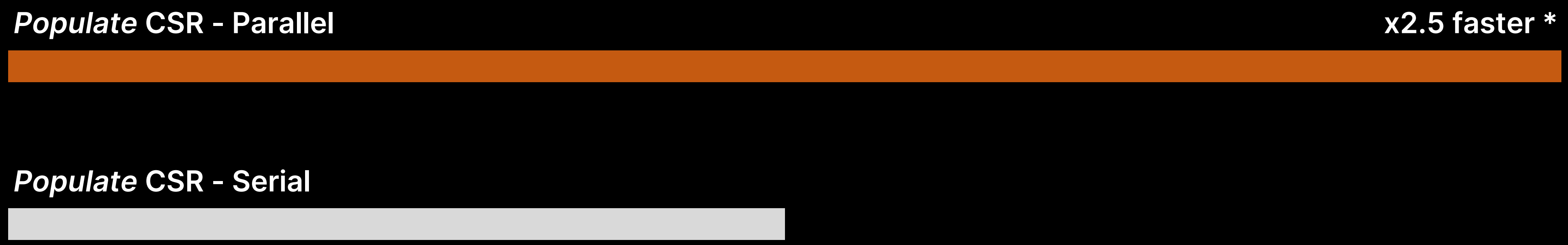
Spatial locality: neighbors of closer vertices are stored continuously

Comparison between serial CSR and the Adjacency List implementations



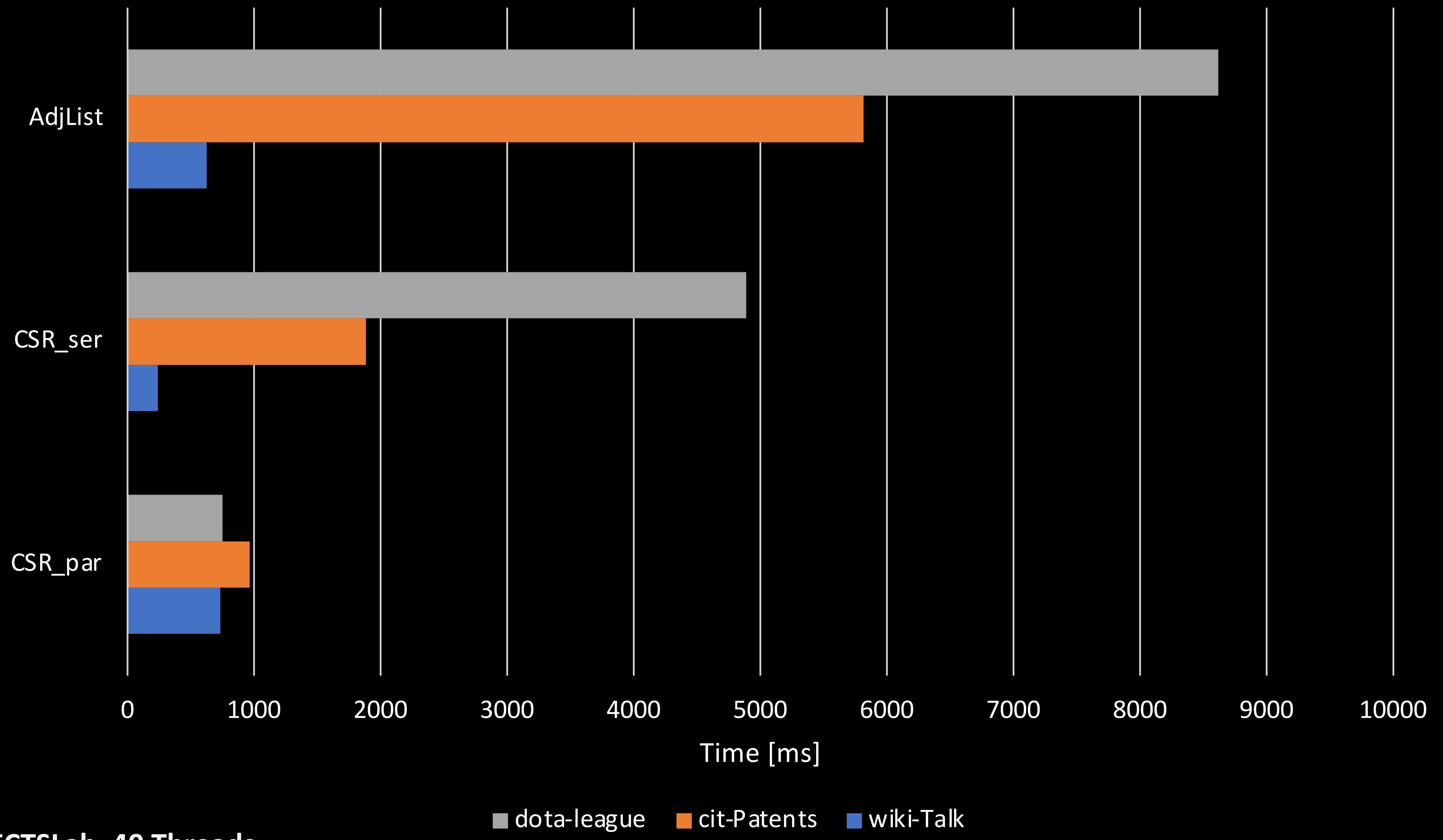
Graph: Wiki_talks (~ 5M edges, ~ 2.5M nodes)
*runned on system with: Intel Core i7-8550U with 1.8GHz clock speed, 4 cores, 8 Threads. The machine has 8GB of RAM, 256KB of L1 cache, 1M of L2 cache, and 8MB of L3 cache.

Comparison between Parallel and Serial Versions



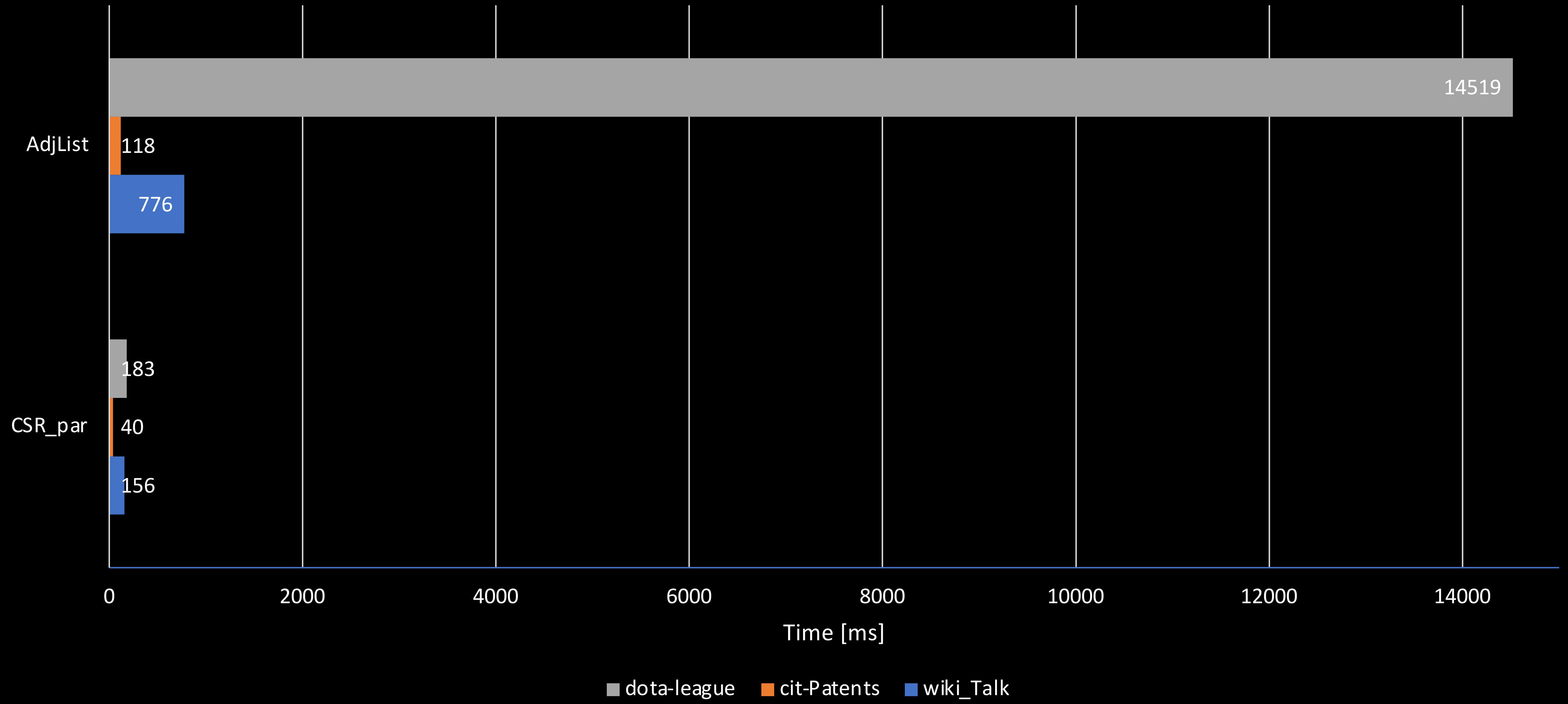
Graph: Wiki_talks (~ 5M edges, ~ 2.5M nodes)
*runned on system with: Intel Core i7-8550U with 1.8GHz clock speed, 4 cores, 8 Threads. The machine has 8GB of RAM, 256KB of L1 cache, 1M of L2 cache, and 8MB of L3 cache.

Populating Time Results



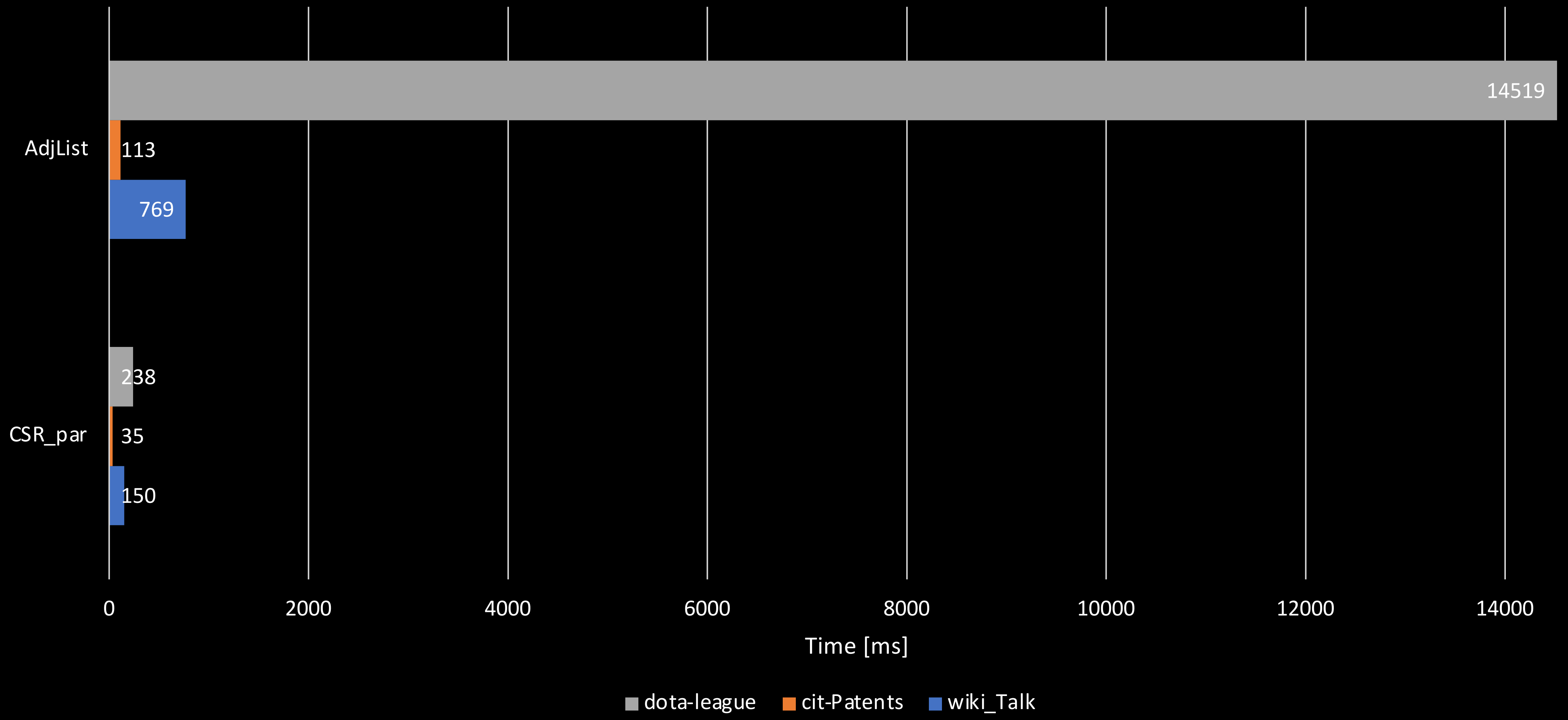
System: nags33 @NECTSLab, 40 Threads

BFS Time Results



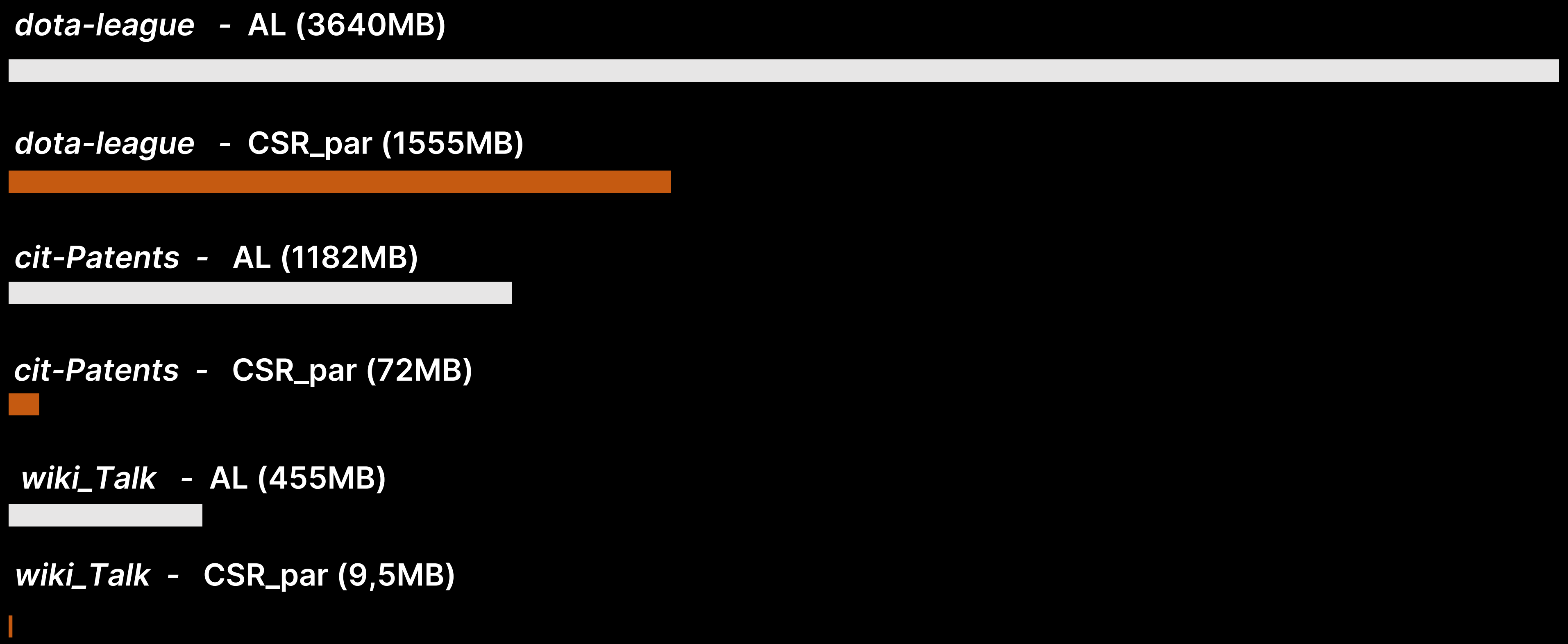
System: nags33 @NECTSLab, 40 Threads

DFS Time Results



System: nags33 @NECTSLab, 40 Threads

Memory results



THANKS