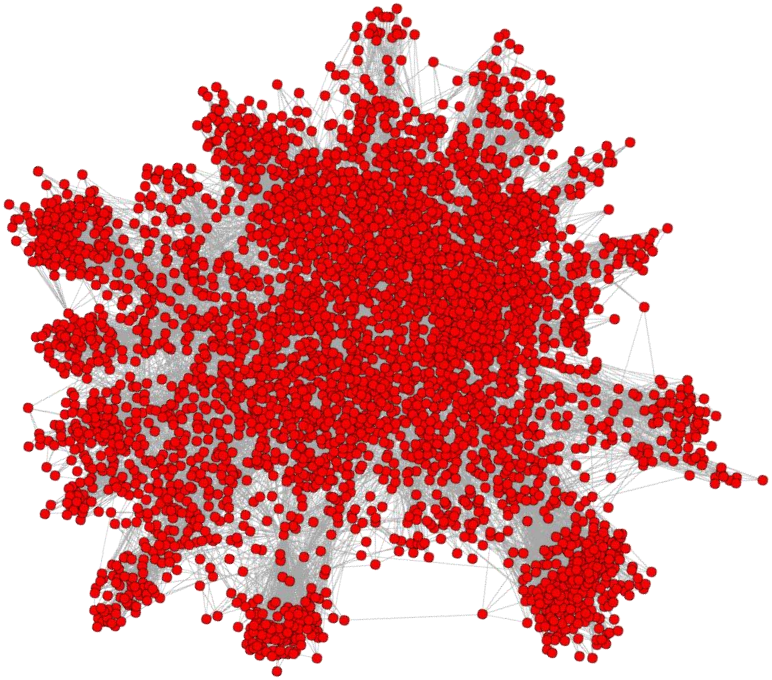
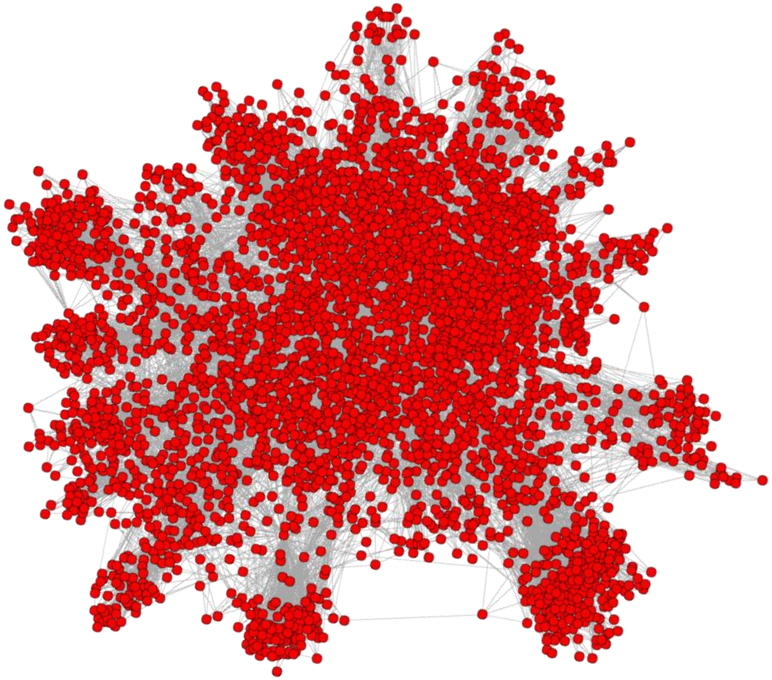
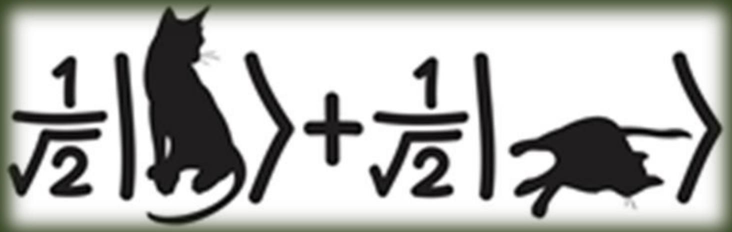
To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations

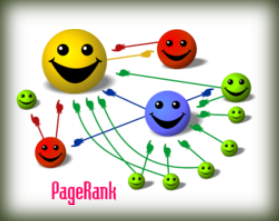
Maciej Besta, Michal Podstawski, Linus Groner, Edgar Solomonik, Torsten Hoefler

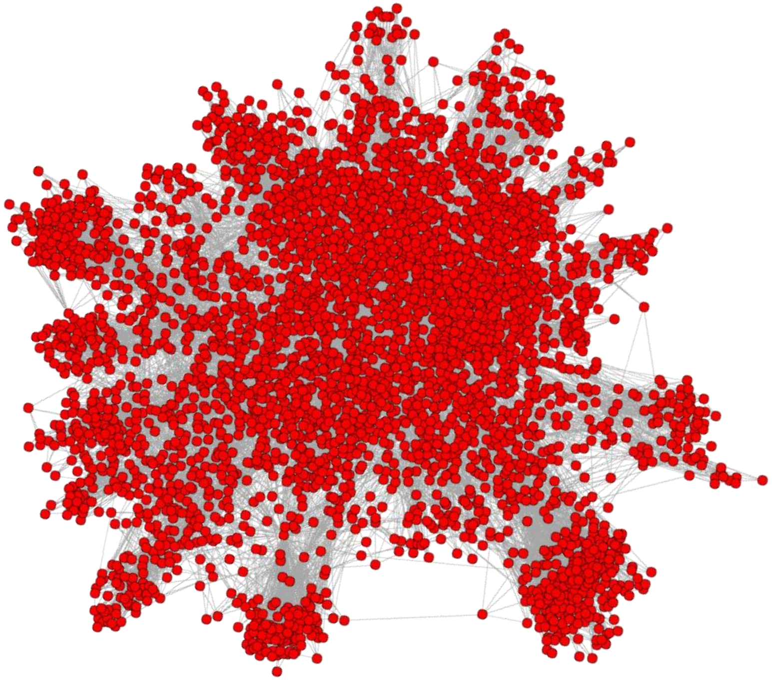




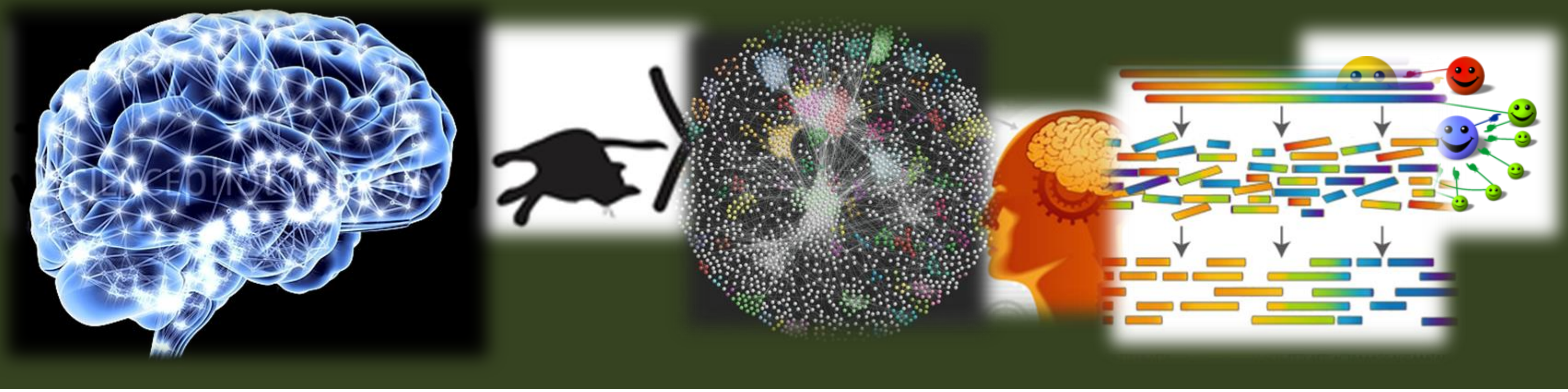
Used in…



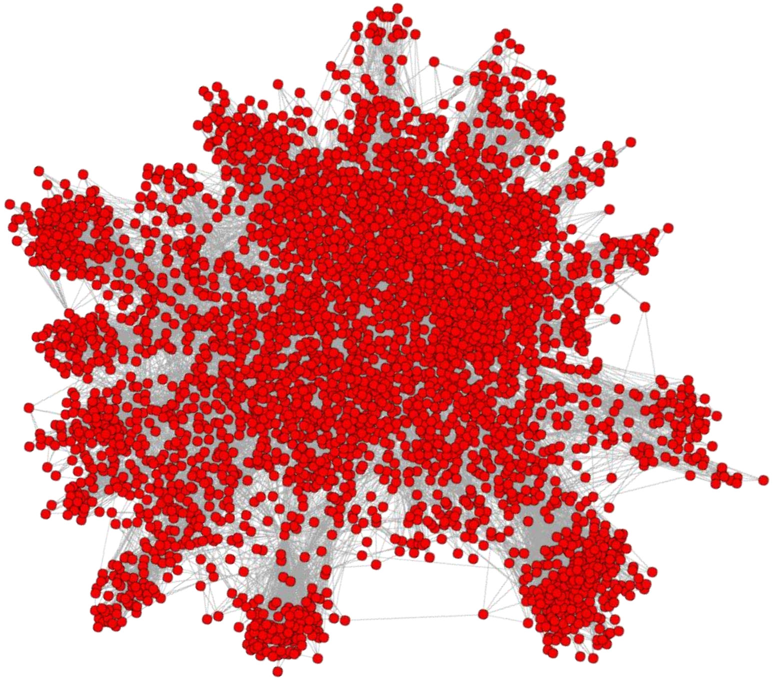




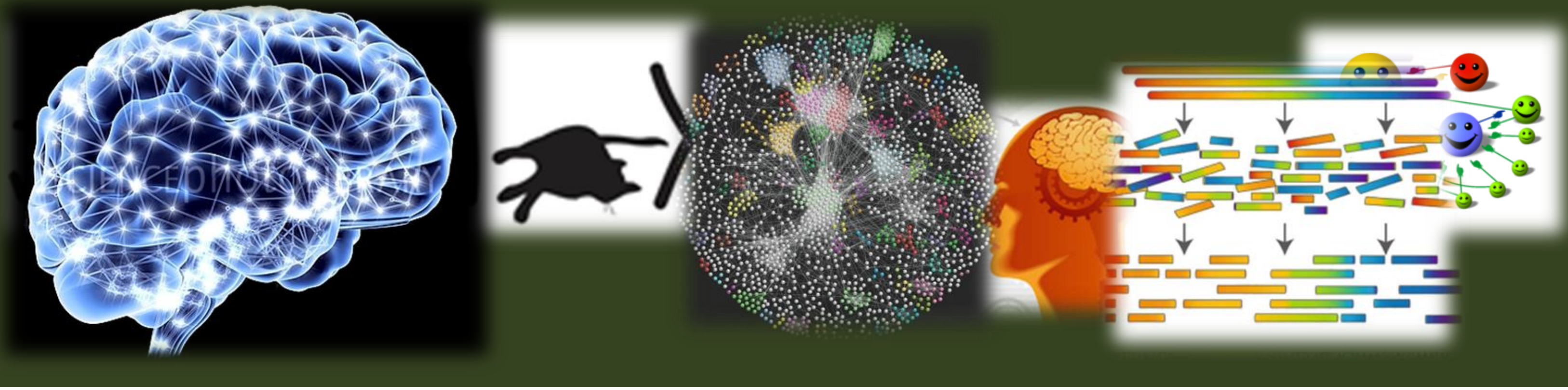
Used in…



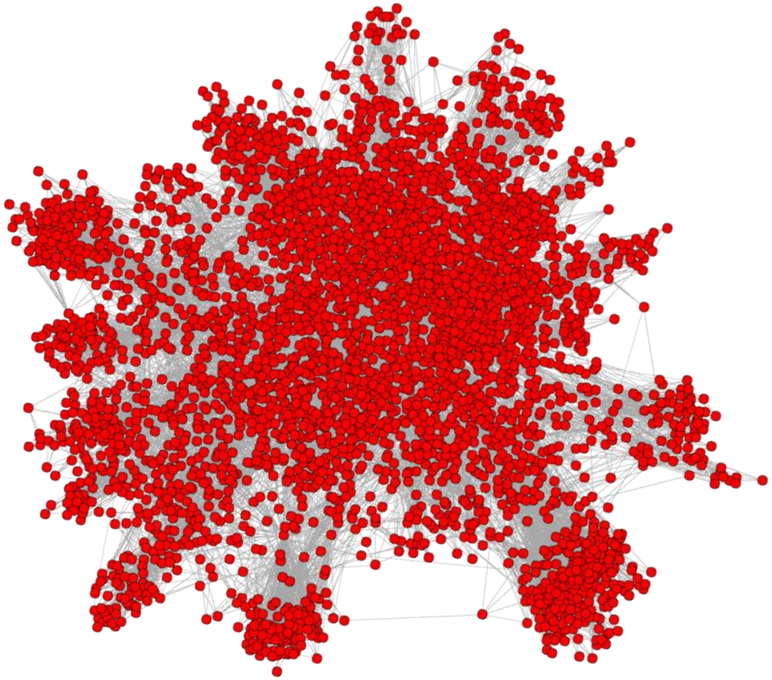
Running on…



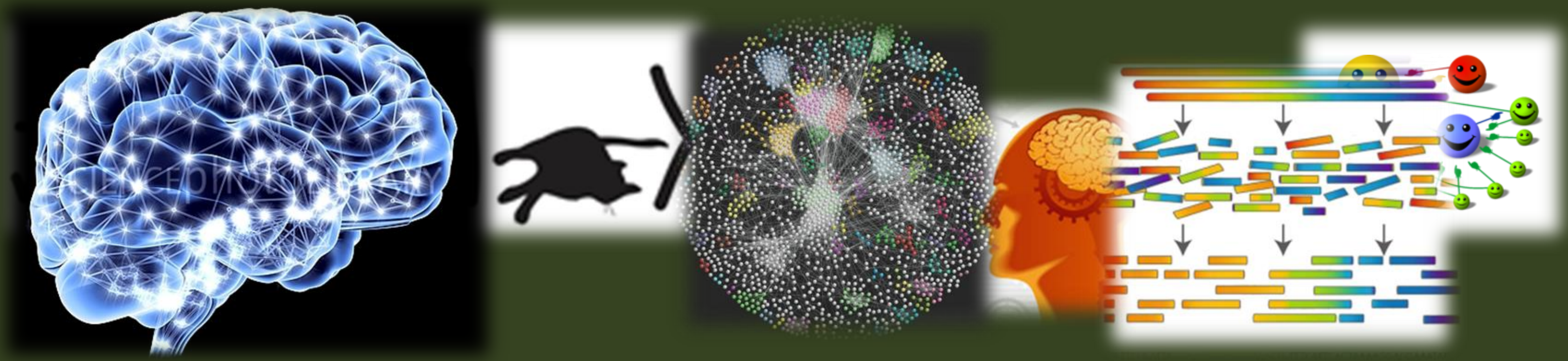
Used in…



Running on…



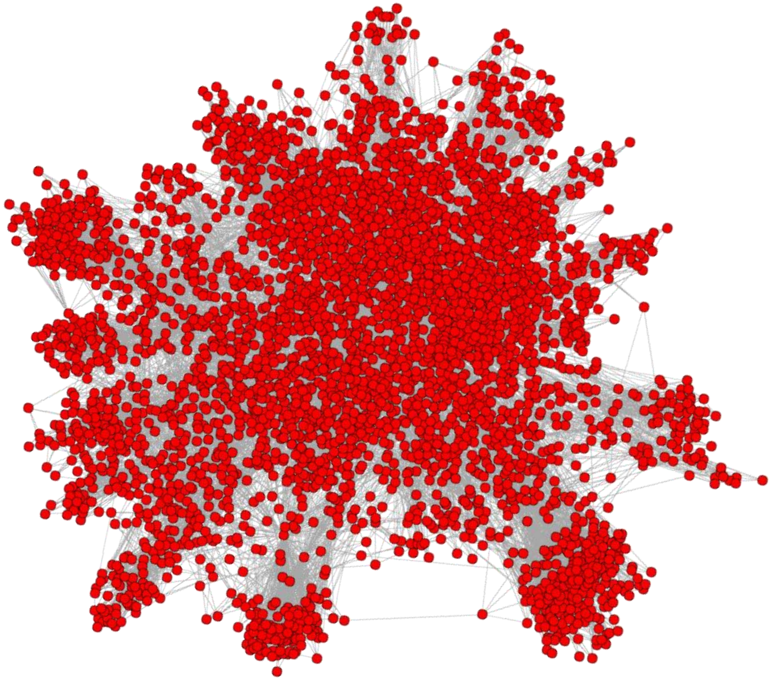
Used in…



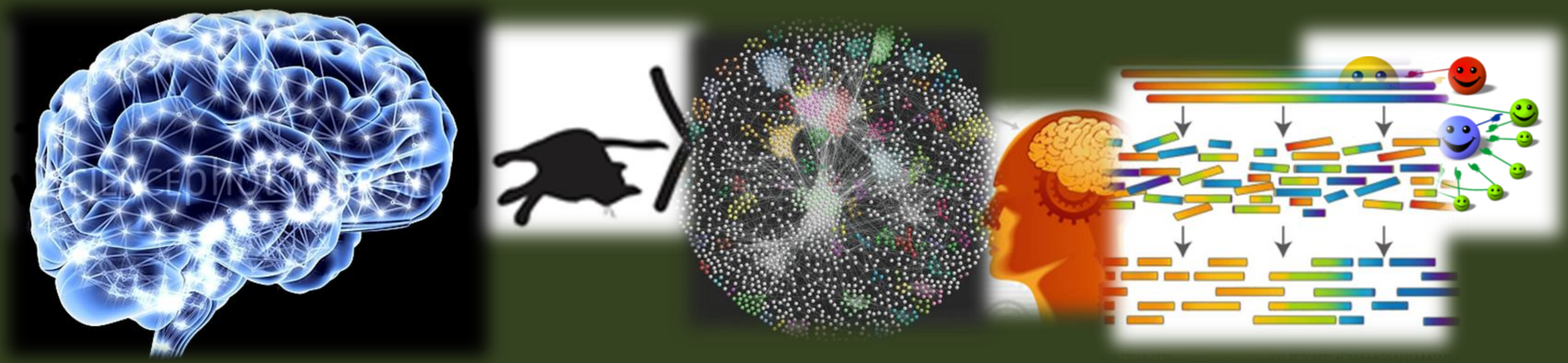
[1] A. Lumsdaine et al. Challenges in Parallel Graph Processing. Parallel Processing Letters. 2007.

Running on…

Irregular



Used in…

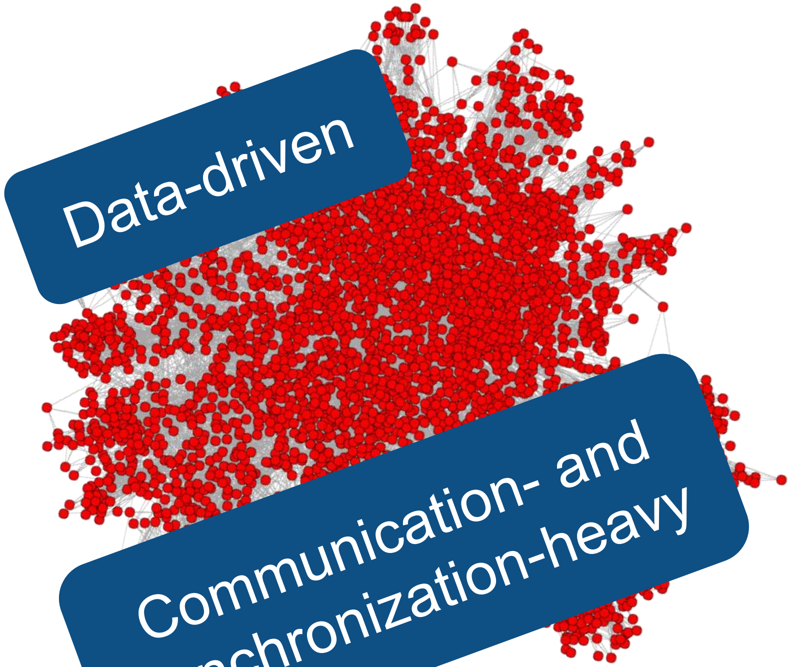


[1] A. Lumsdaine et al. Challenges in Parallel Graph Processing. Parallel Processing Letters. 2007.

Running on…

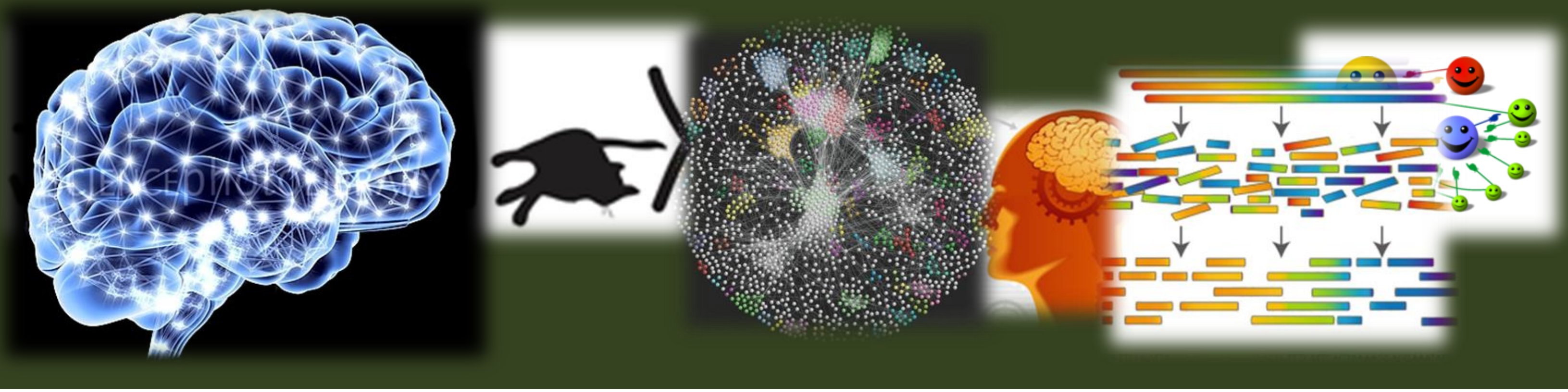
Irregular

Data-driven



Communication- and synchronization-heavy

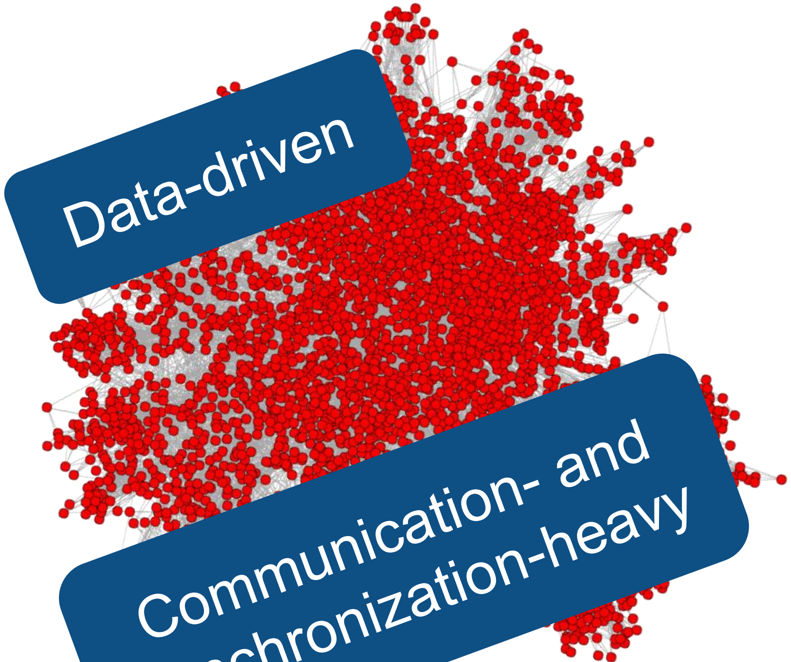
Used in…



[1] A. Lumsdaine et al. Challenges in Parallel Graph Processing. Parallel Processing Letters. 2007.

Running on…

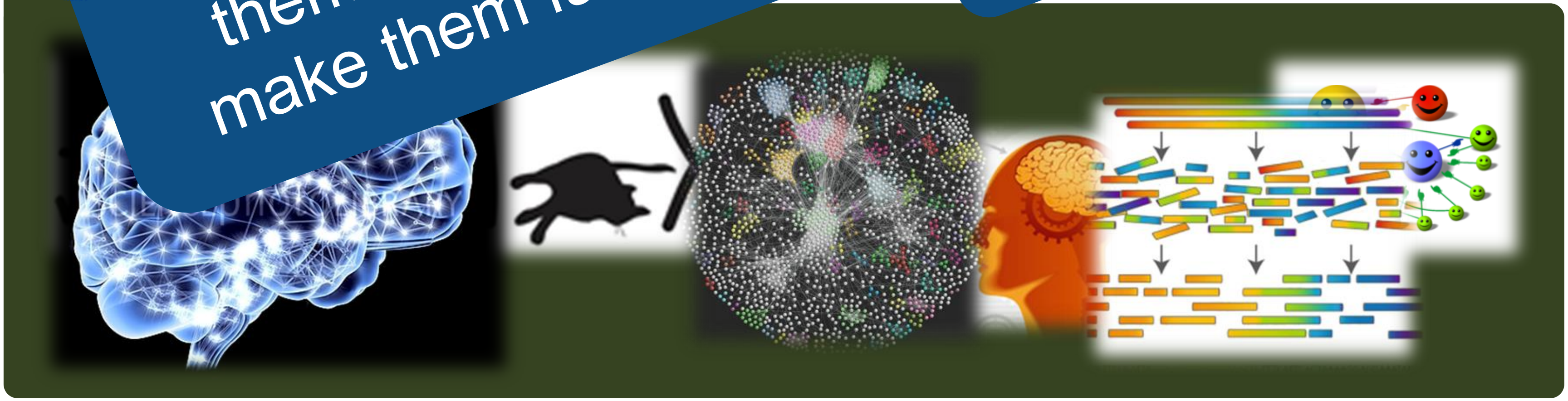
Irregular



Data-driven

Communication- and synchronization-heavy

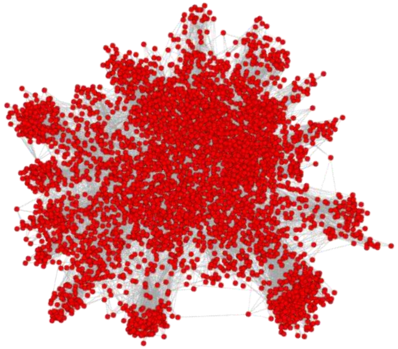
Goal: understand them better and make them faster!



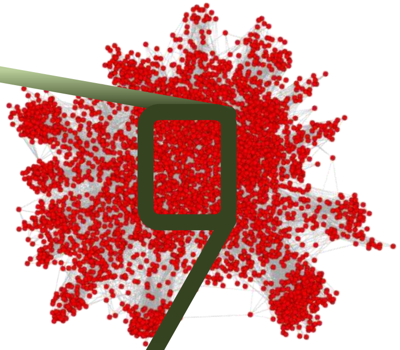
[1] A. Lumsdaine et al. Challenges in Parallel Graph Processing. Parallel Processing Letters. 2007.

PAGERANK

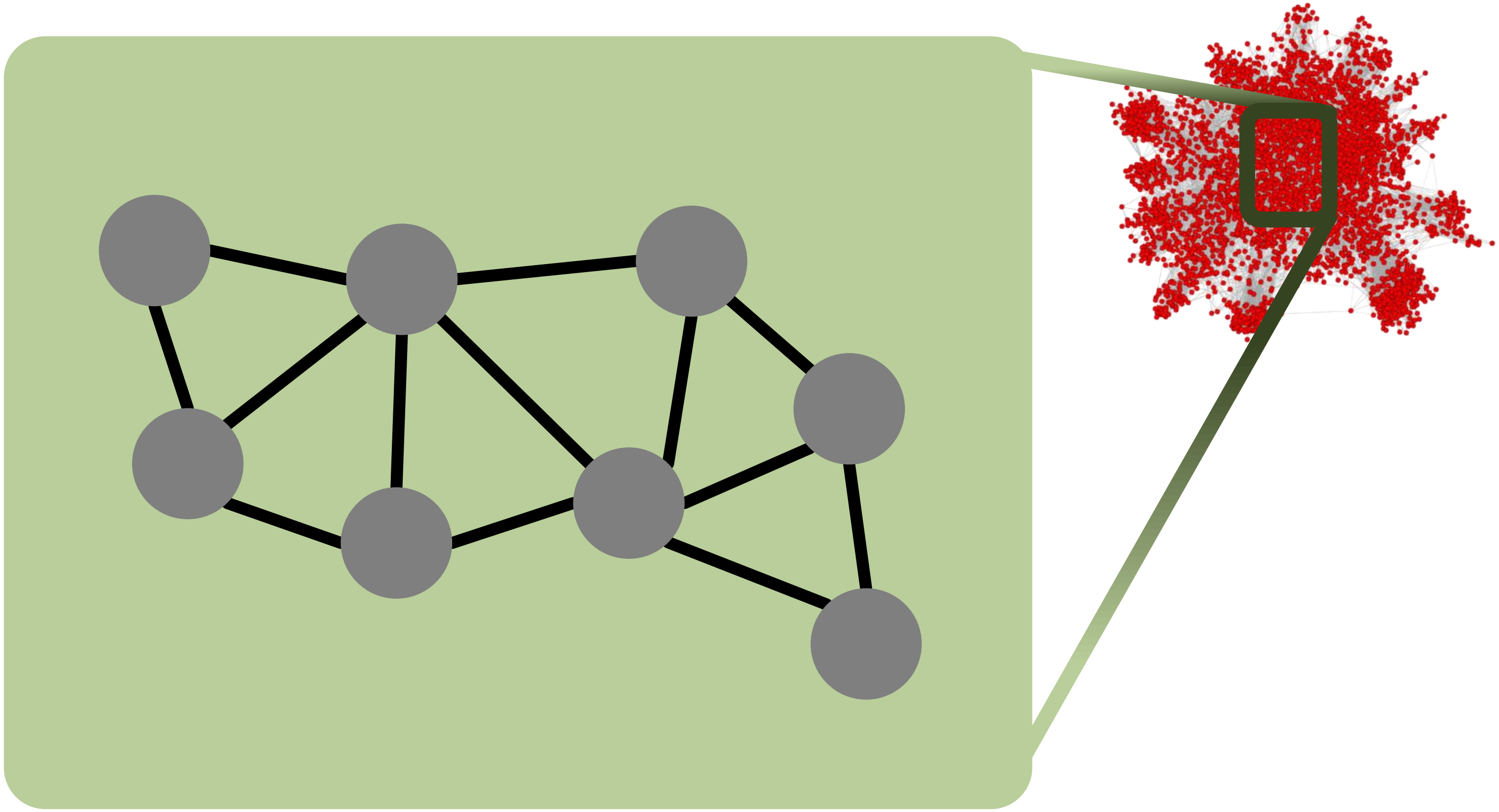
PAGERANK



PAGERANK

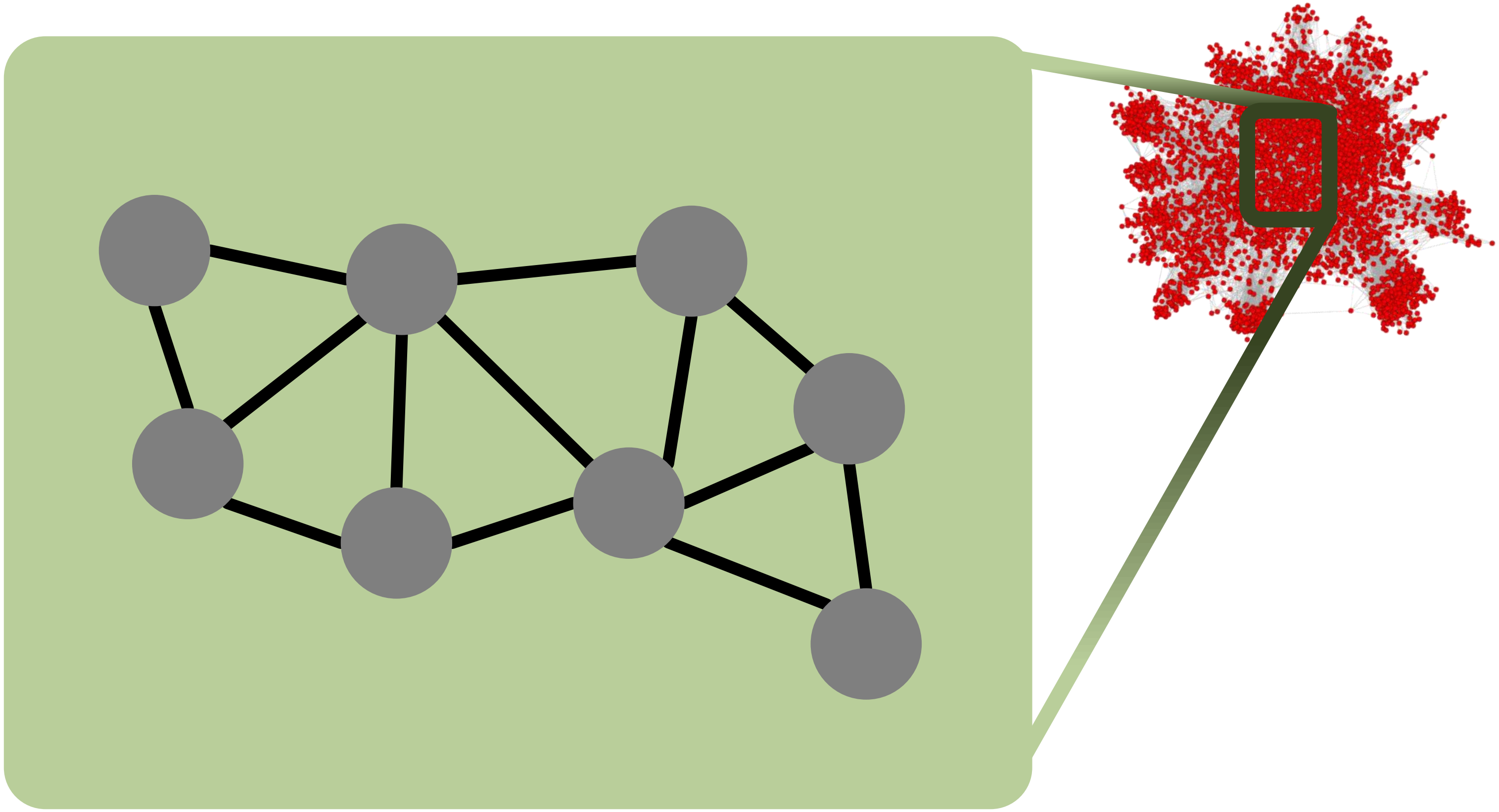


PAGERANK



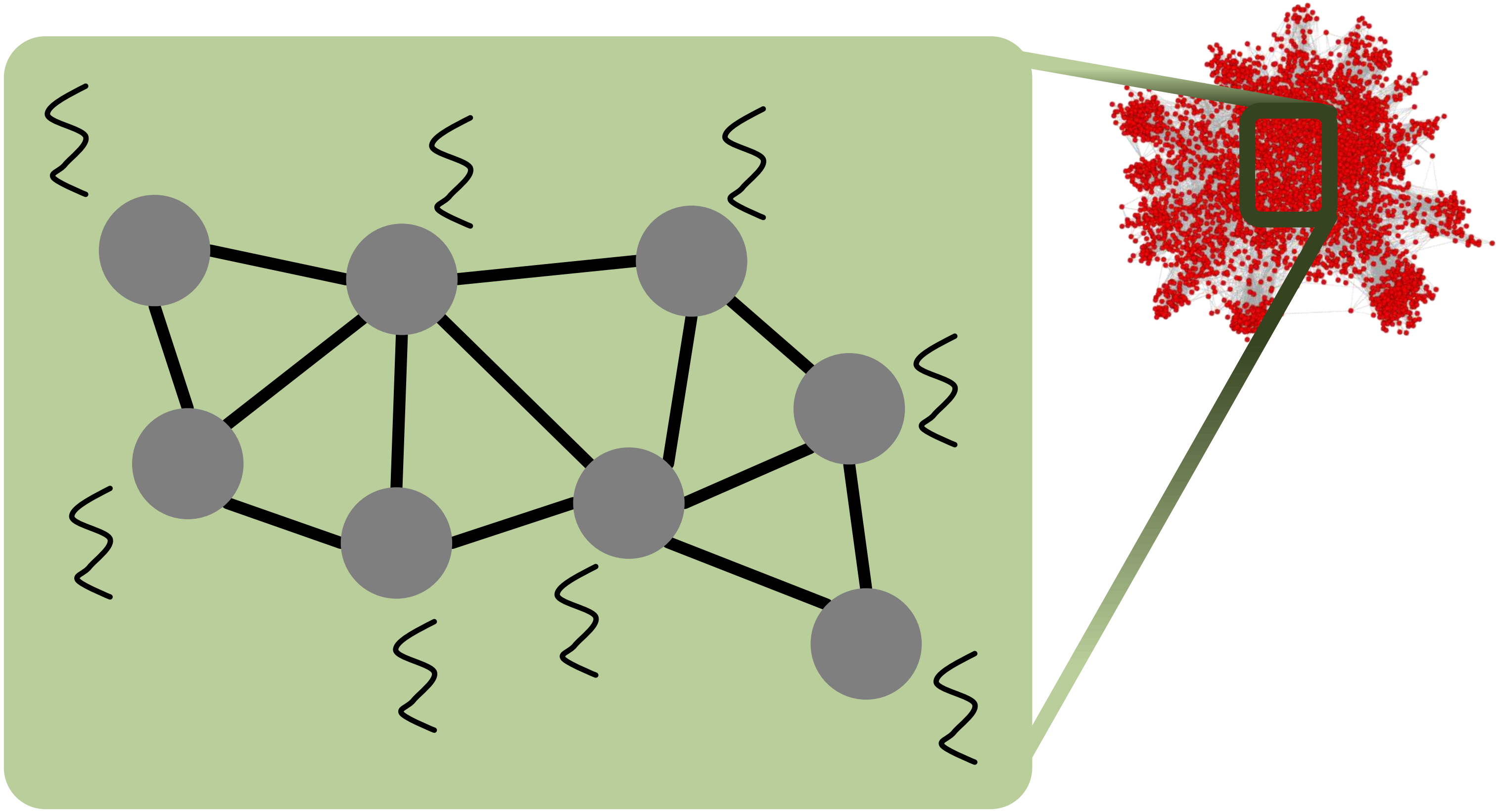
PAGERANK

P threads are used



PAGERANK

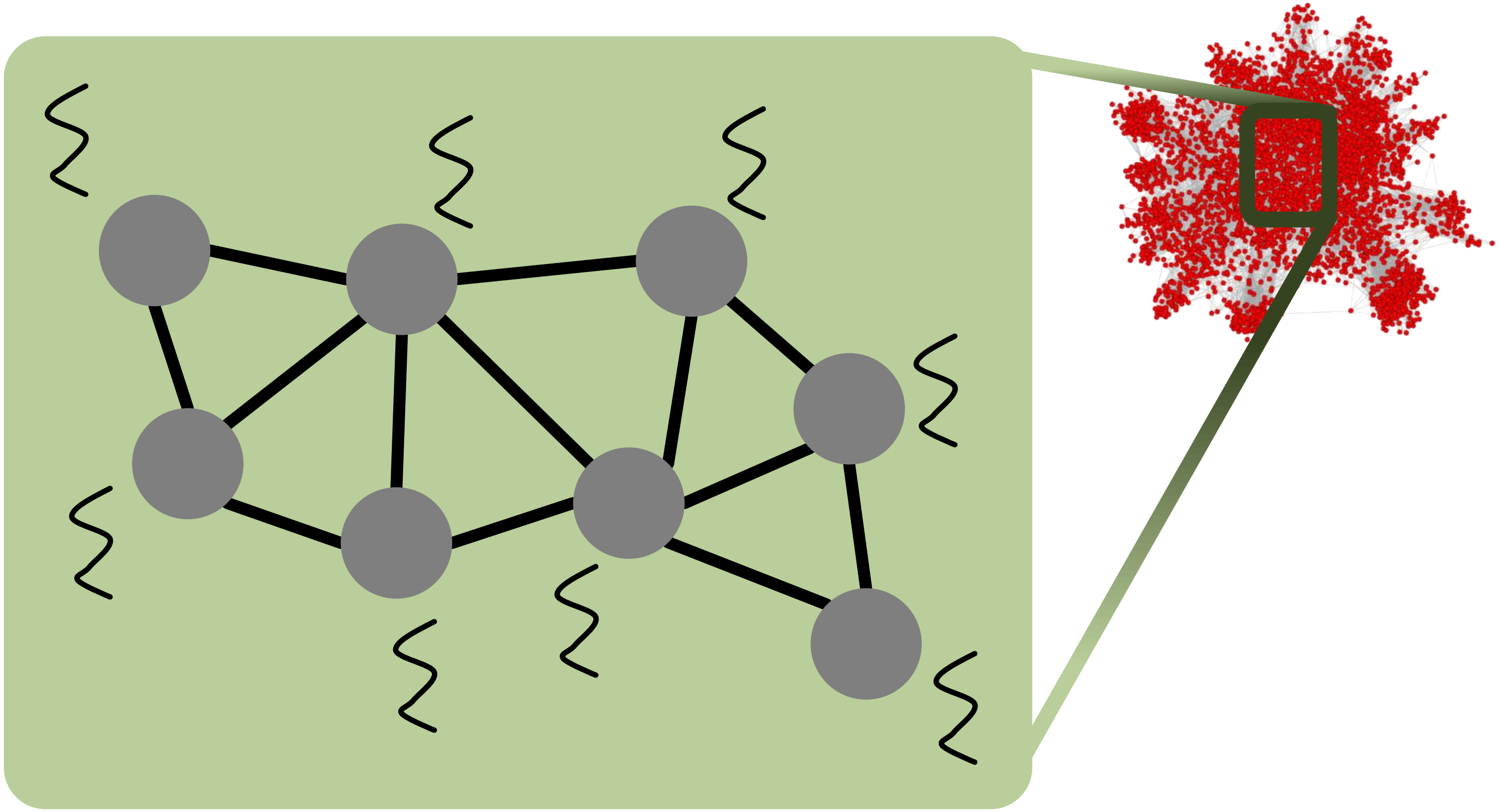
P threads are used



[1] J. J. Whang et al. Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned. Euro-Par 2015.

PAGERANK

P threads are used

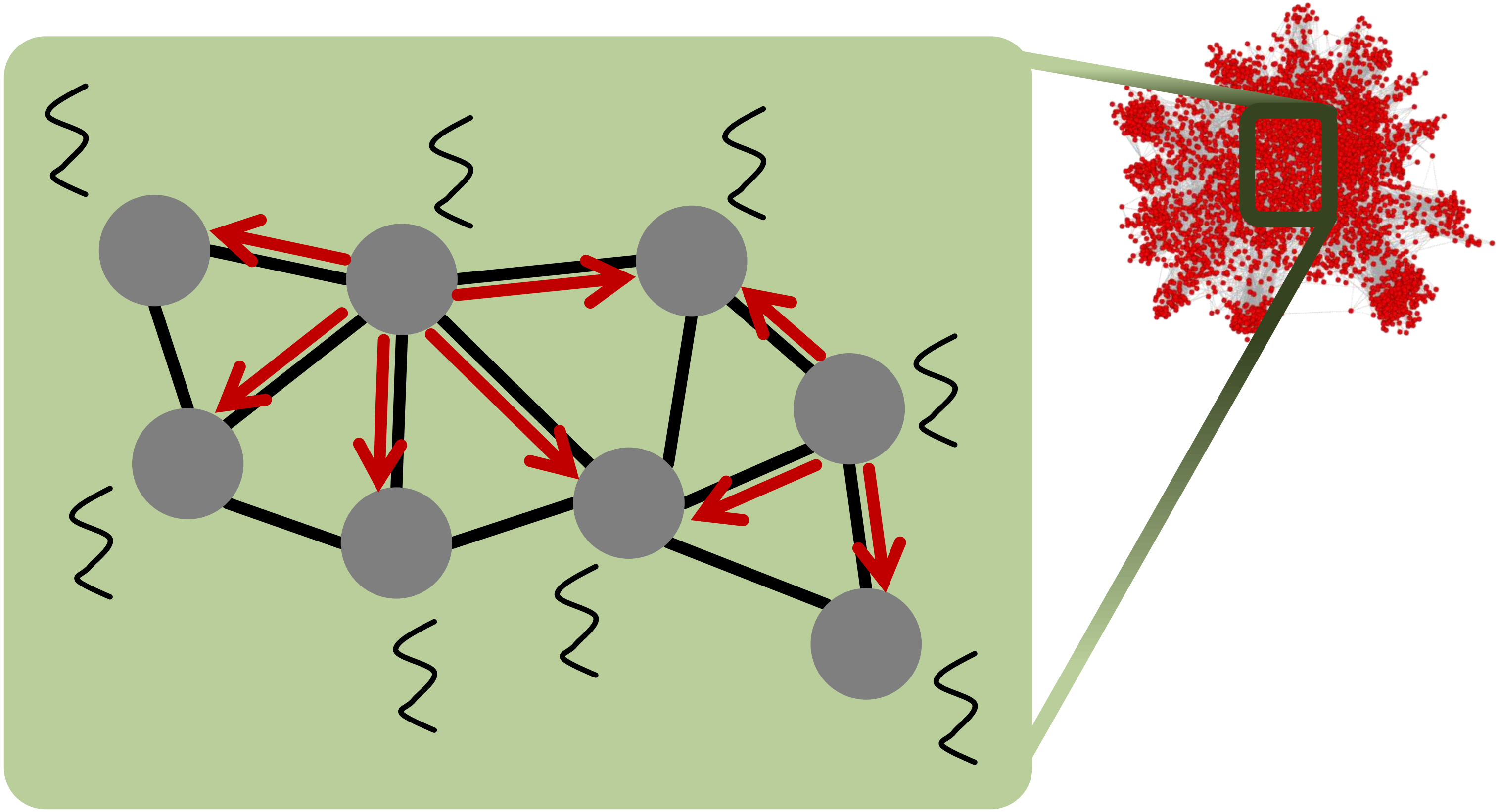


Pushing

[1] J. J. Whang et al. Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned. Euro-Par 2015.

PAGERANK

P threads are used

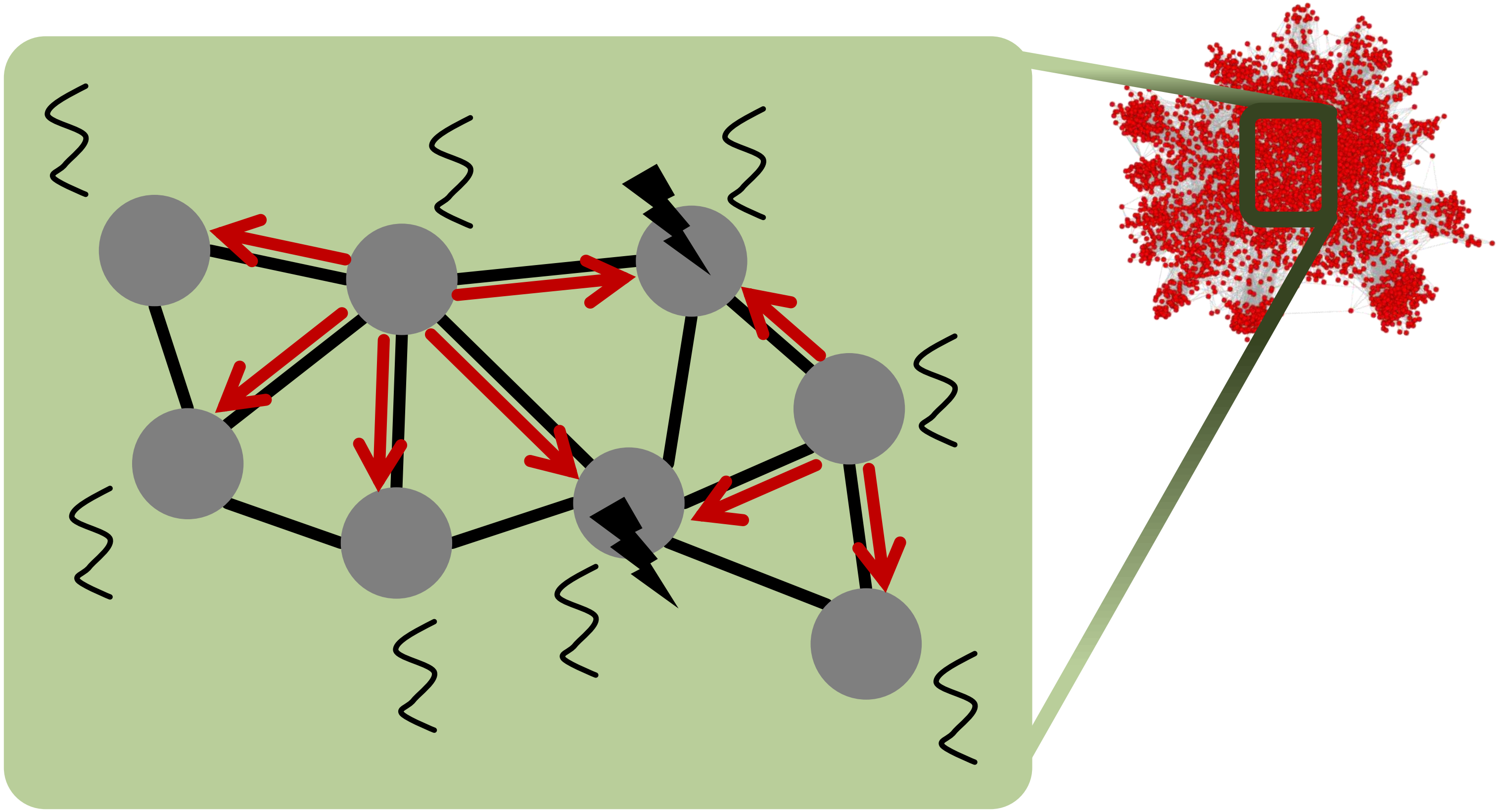


Pushing

[1] J. J. Whang et al. Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned. Euro-Par 2015.

PAGERANK

P threads are used



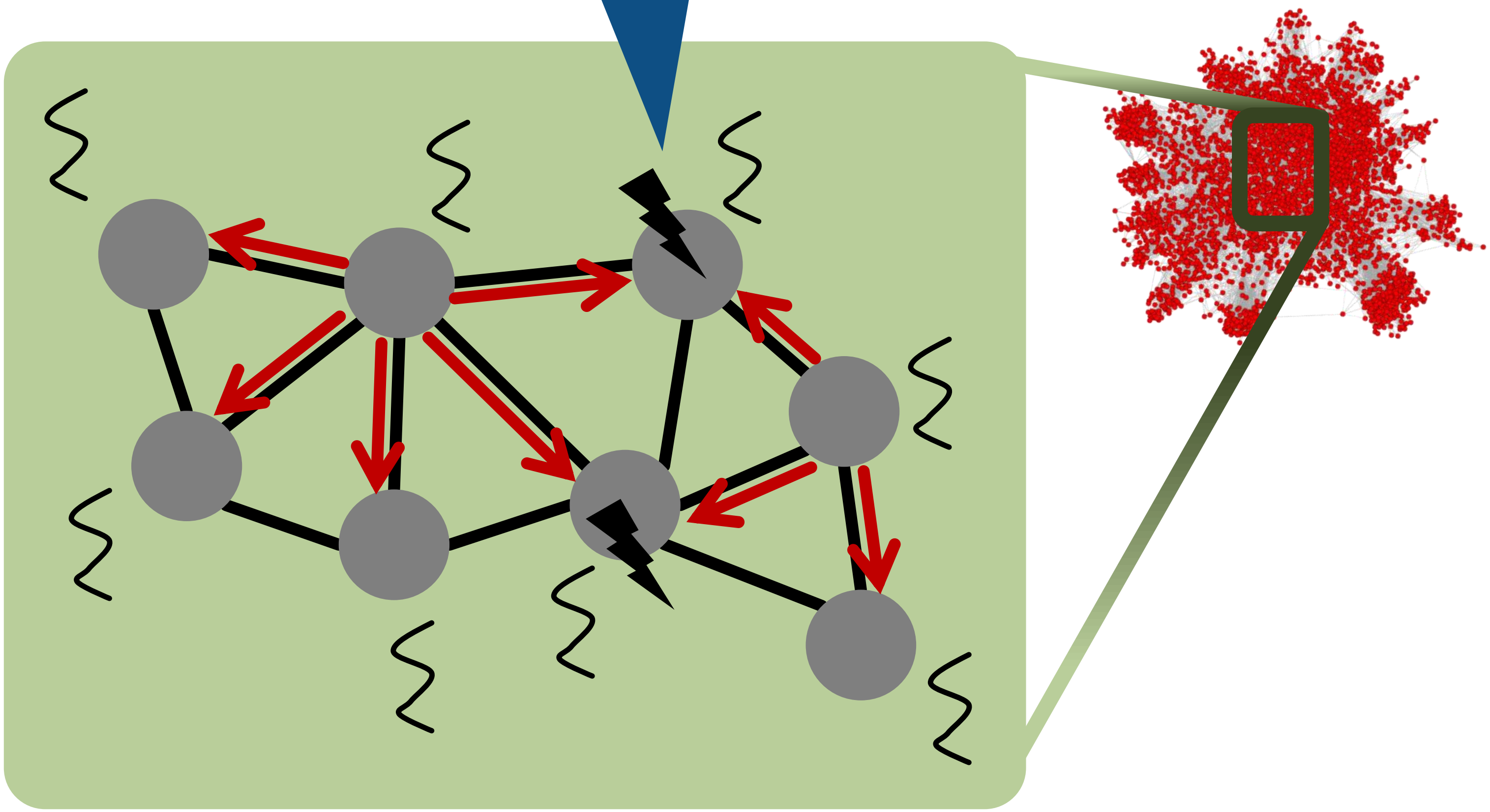
Pushing

[1] J. J. Whang et al. Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned. Euro-Par 2015.

PAGERANK

Write conflicts

P threads are used

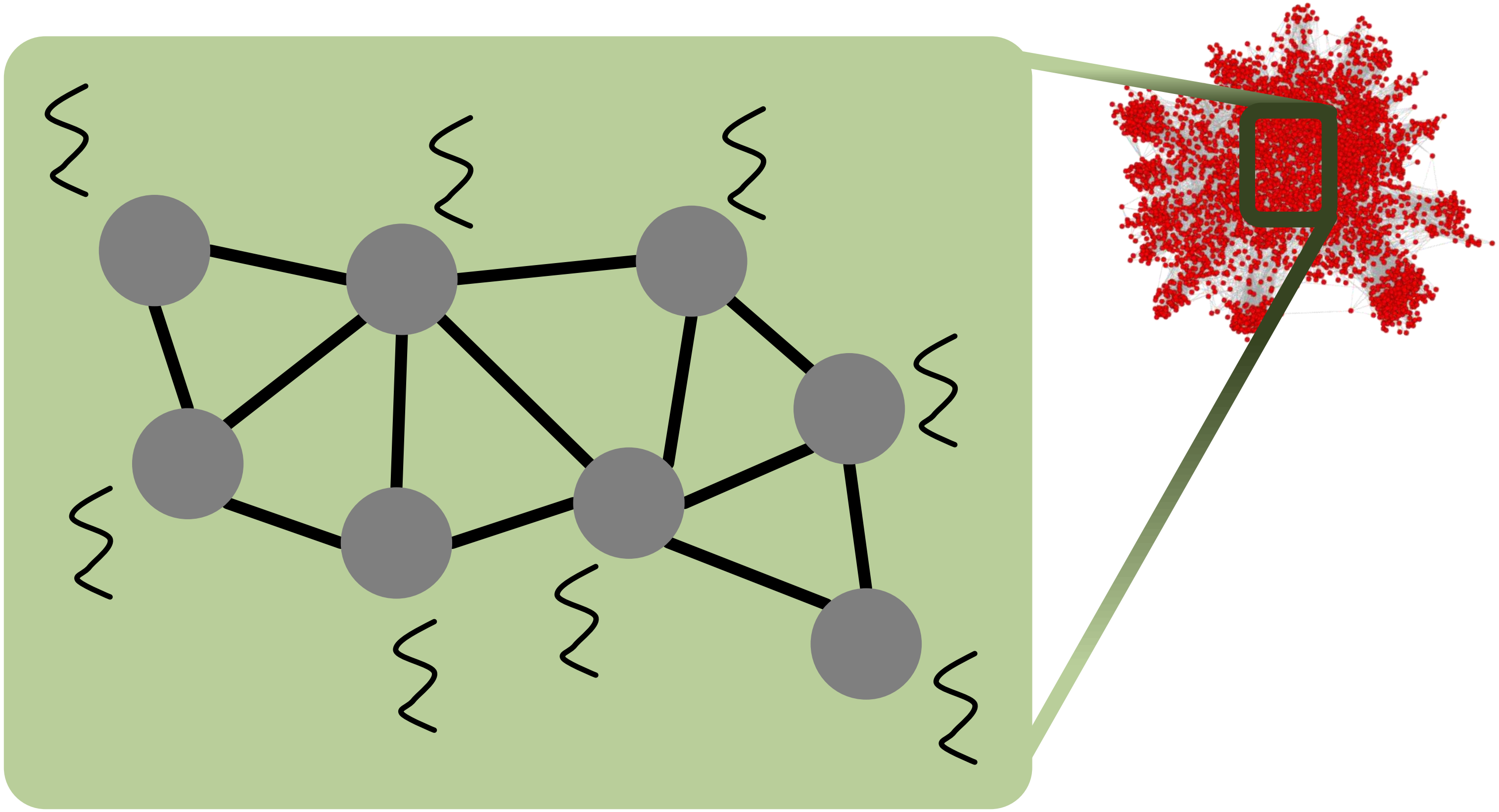


Pushing

[1] J. J. Whang et al. Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned. Euro-Par 2015.

PAGERANK

P threads are used

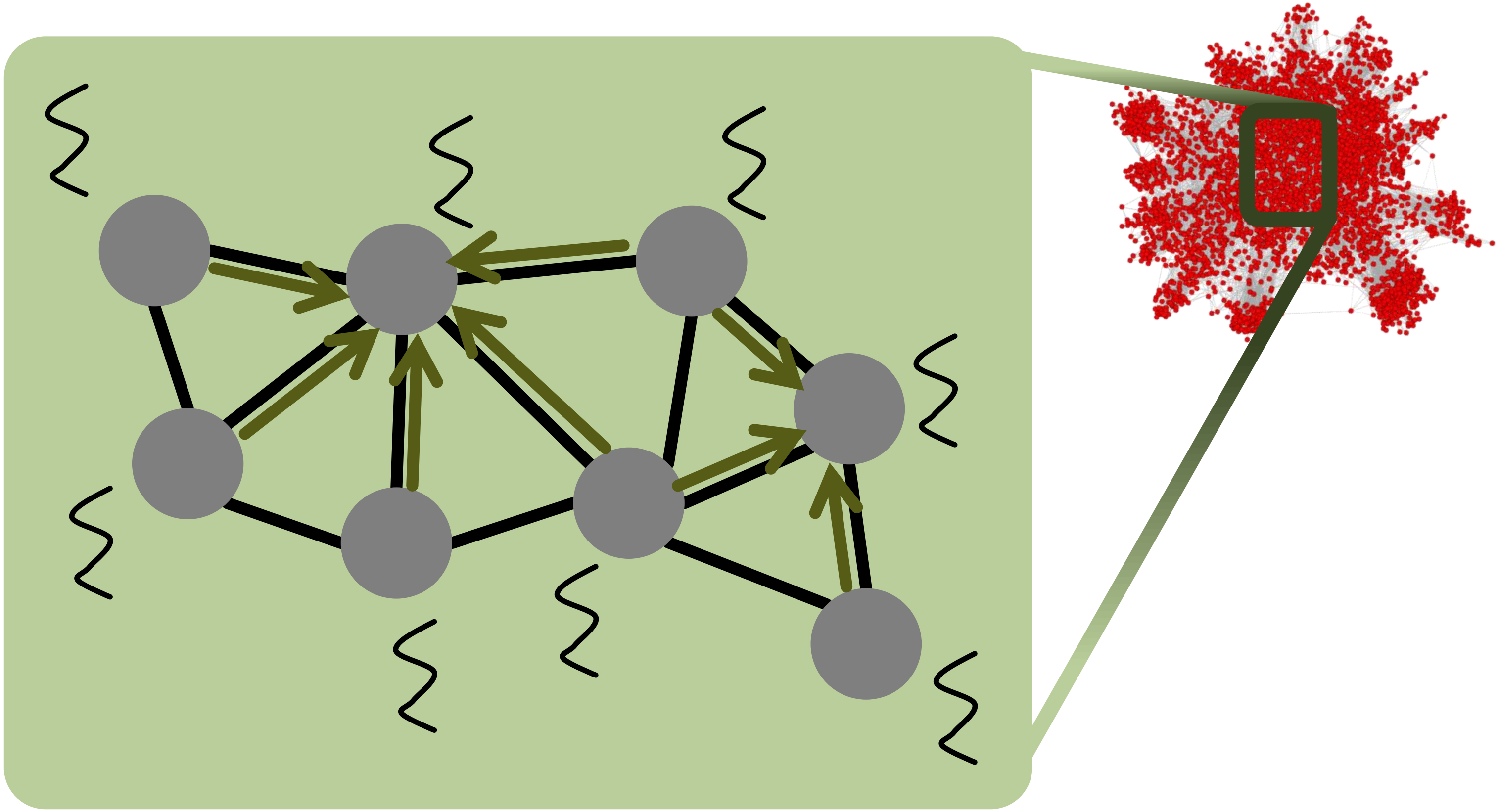


Pulling

[1] J. J. Whang et al. Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned. Euro-Par 2015.

PAGERANK

P threads are used

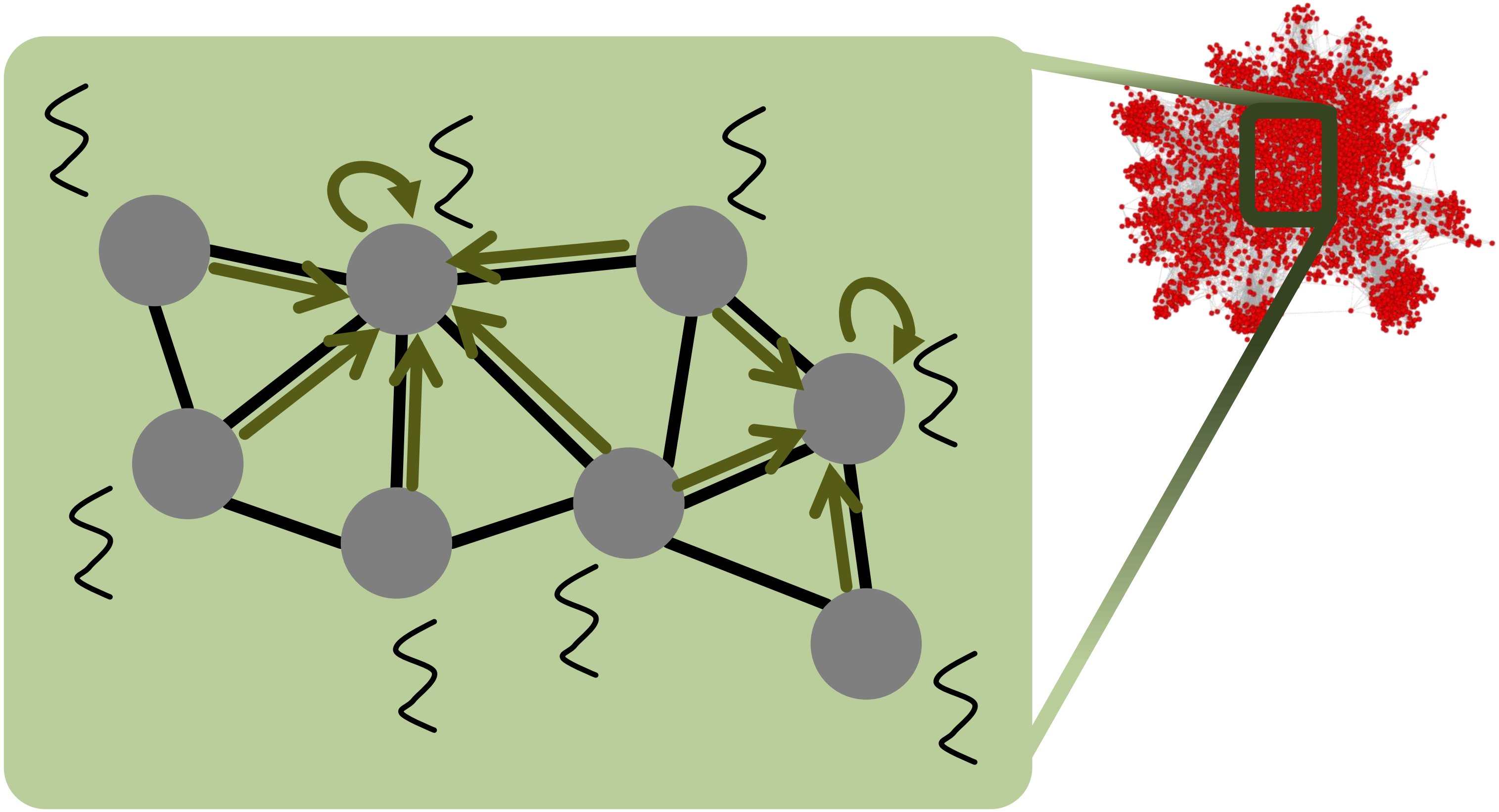


Pulling

[1] J. J. Whang et al. Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned. Euro-Par 2015.

PAGERANK

P threads are used

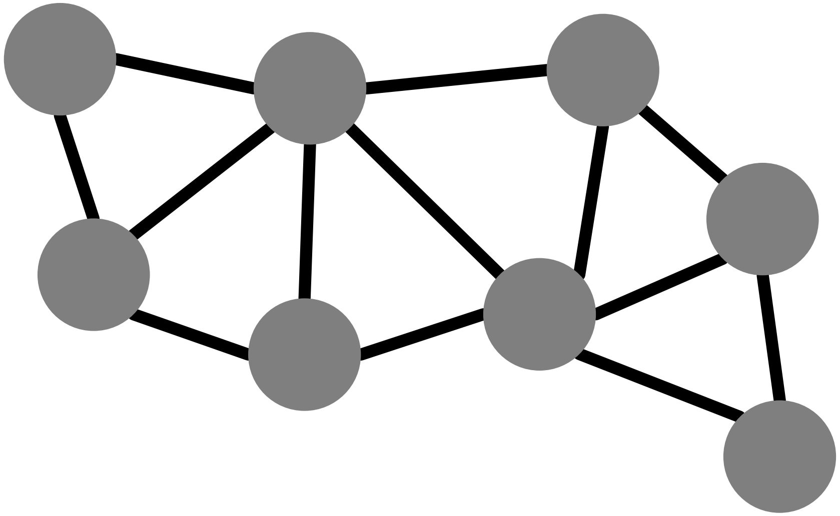


Pulling

[1] J. J. Whang et al. Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned. Euro-Par 2015.

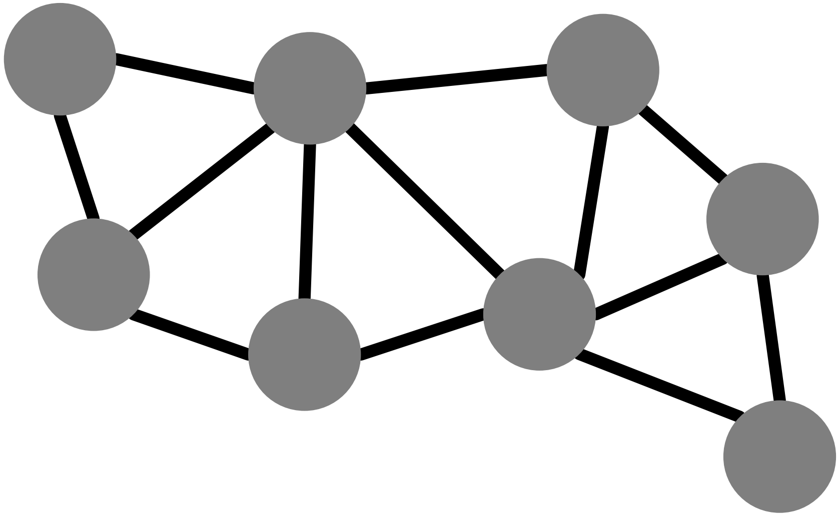
BFS

Top-Down vs. Bottom-Up [1]

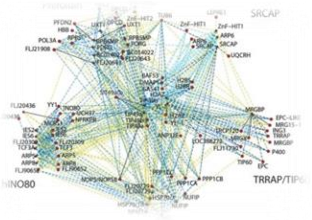


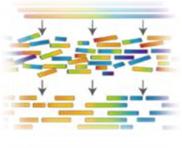
BFS

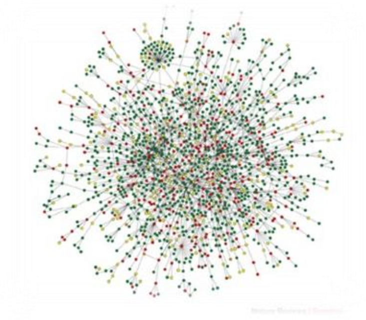
TOP-DOWN VS. BOTTOM-UP [1]

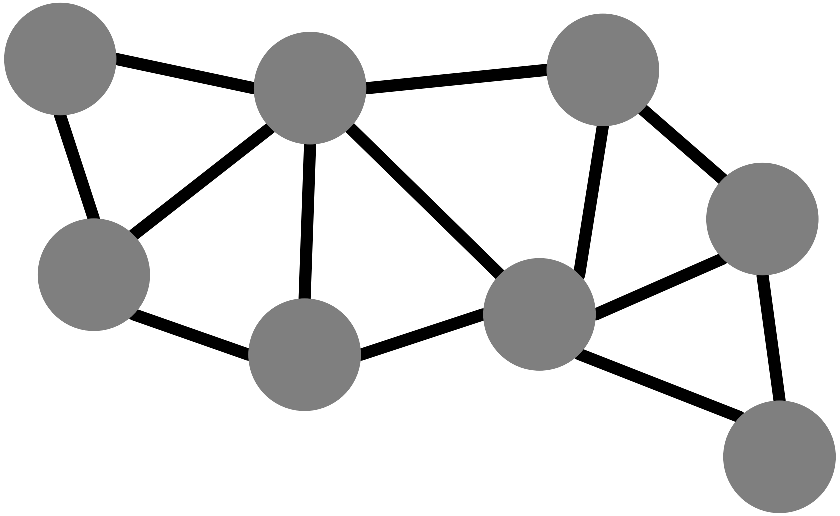


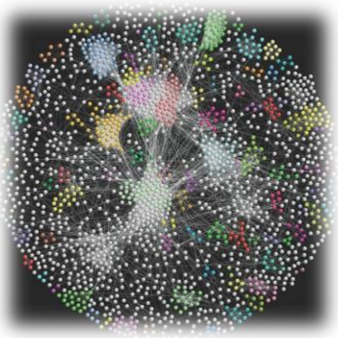
BFS
Top-Down vs. Bottom-Up [1]





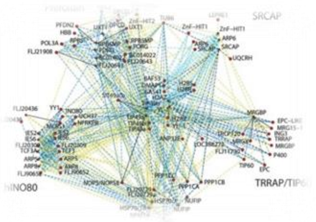


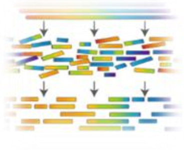


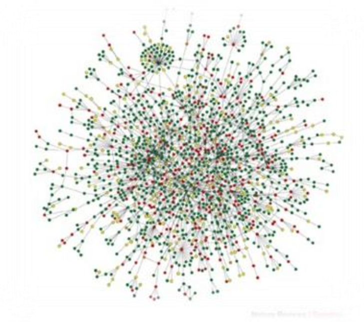


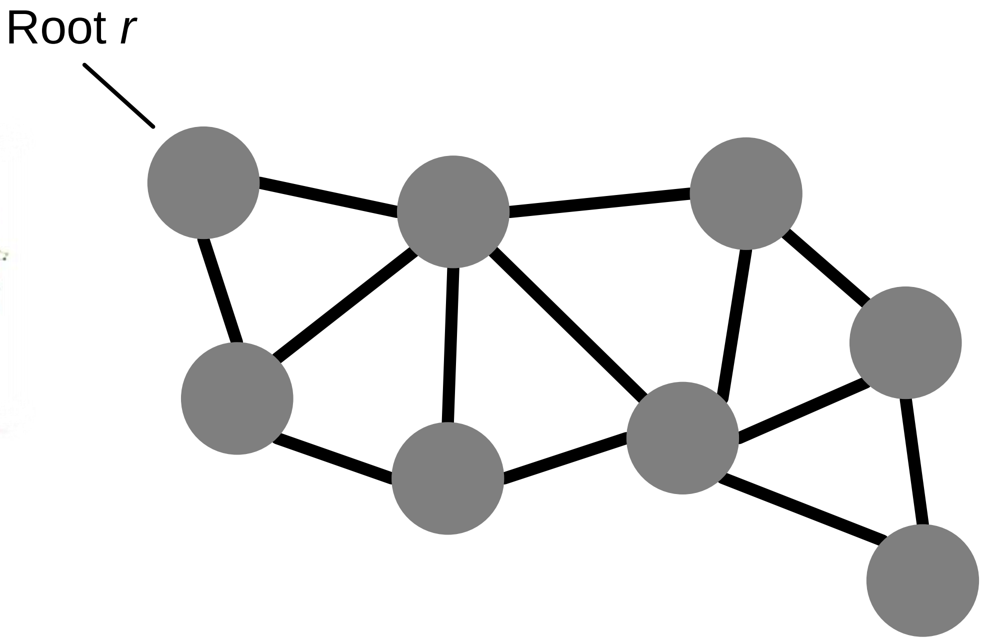
GRAPH 500

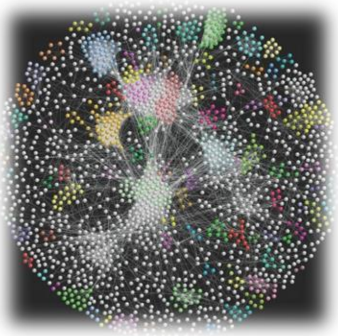
BFS
TOP-DOWN VS. BOTTOM-UP [1]





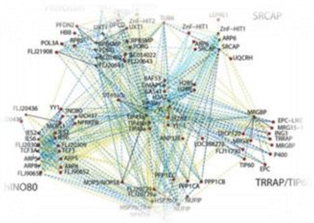


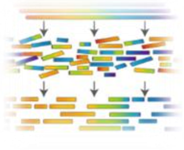


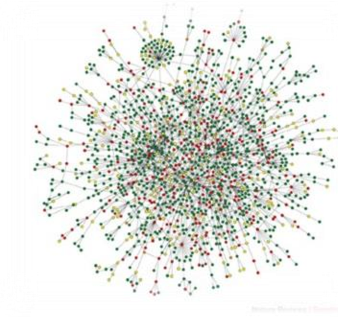


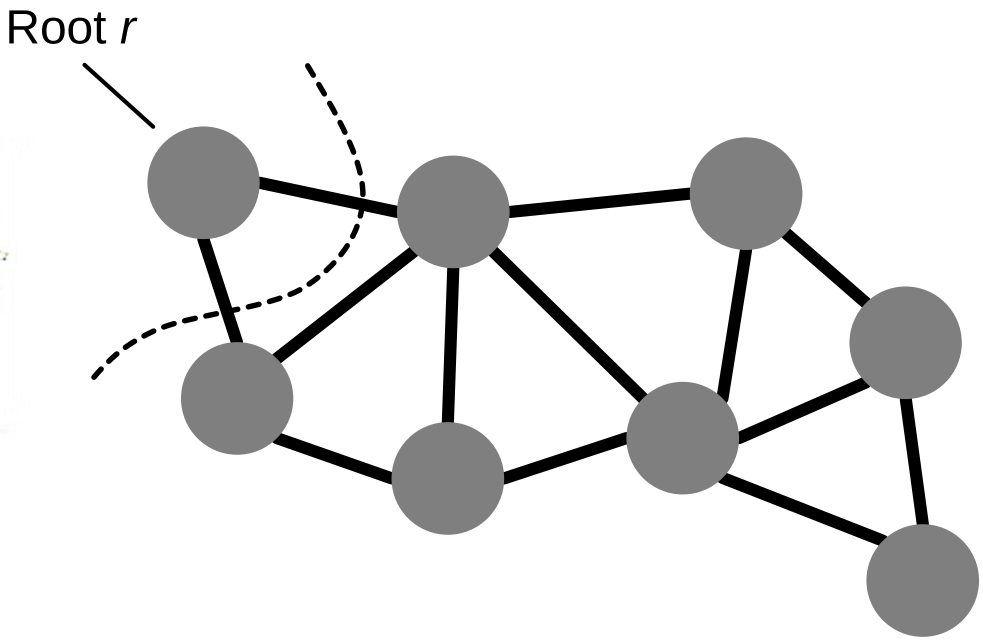
BFS

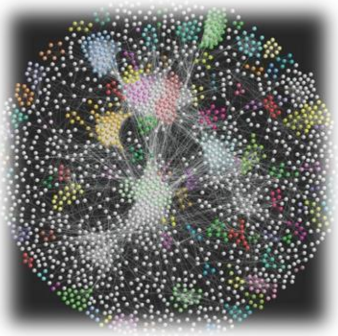
Top-Down vs. Bottom-Up [1]





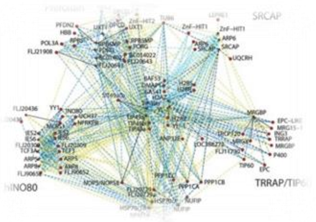


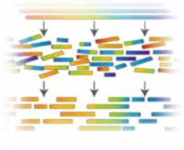


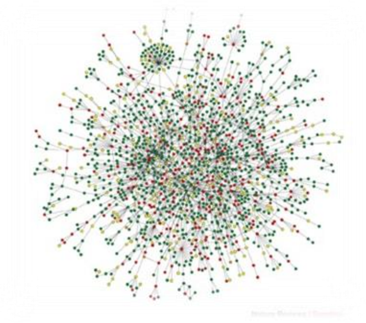


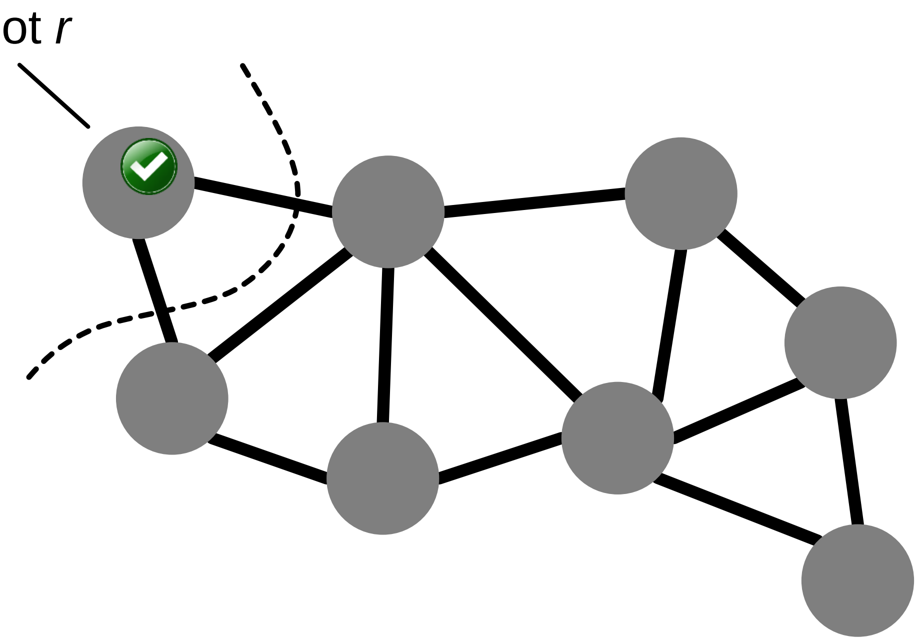
BFS

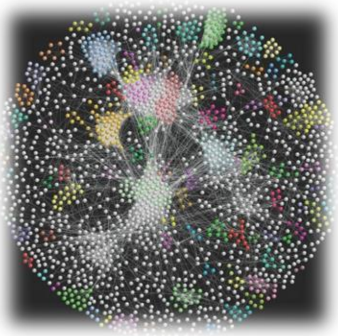
TOP-DOWN VS. BOTTOM-UP [1]





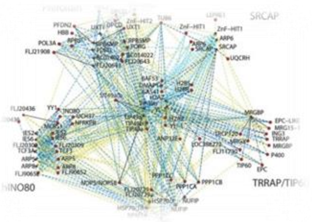


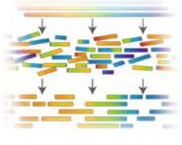


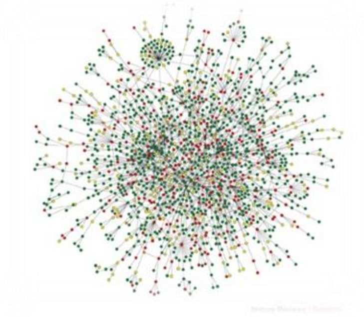


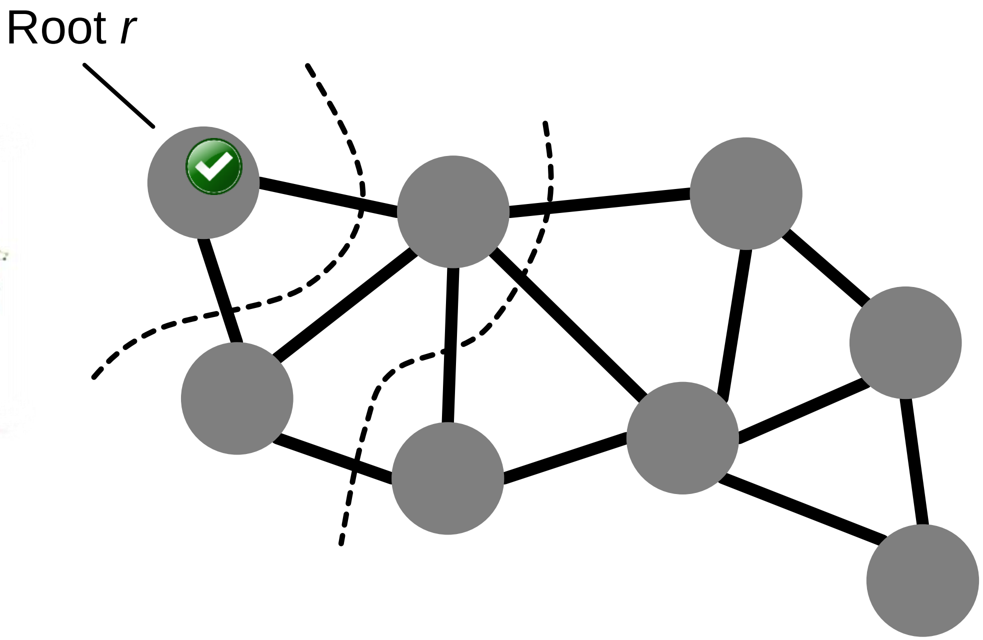
BFS

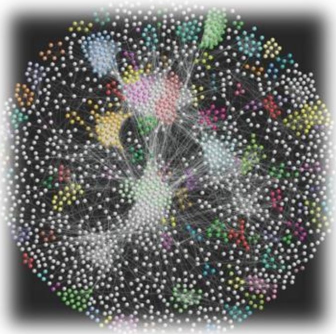
TOP-DOWN VS. BOTTOM-UP [1]





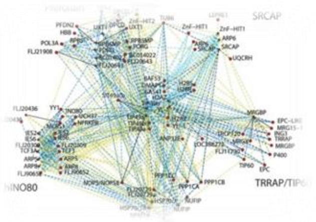


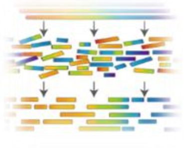


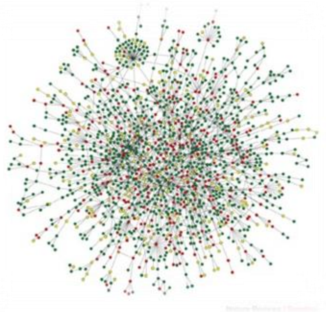


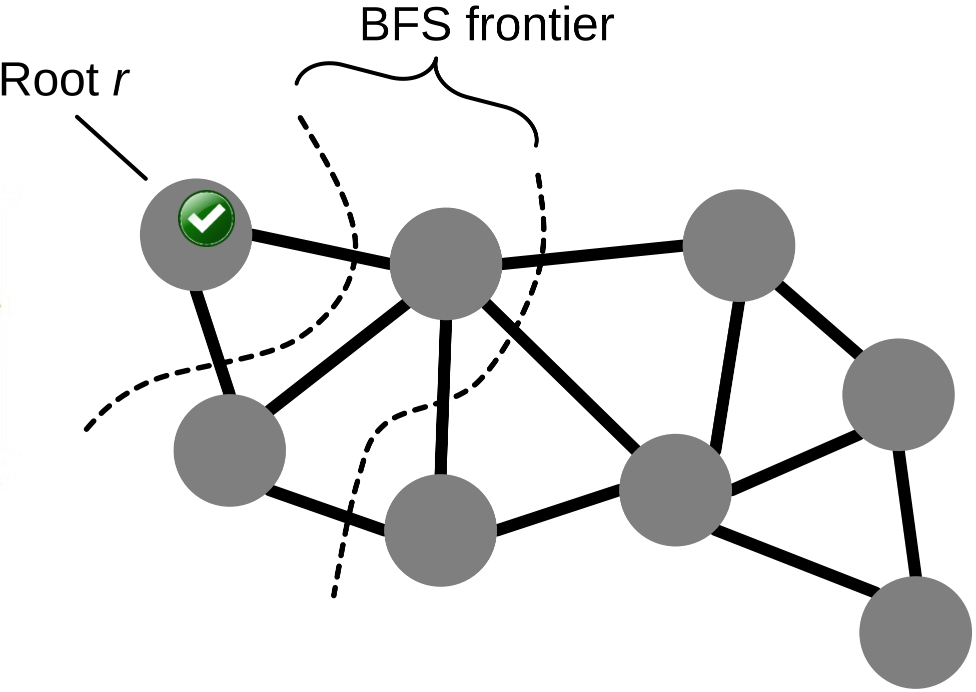
BFS

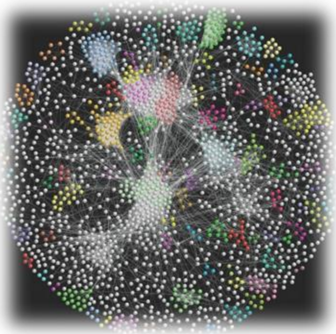
TOP-DOWN VS. BOTTOM-UP [1]





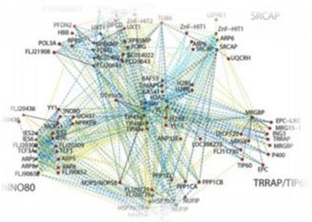


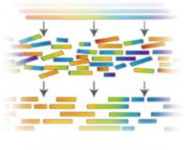


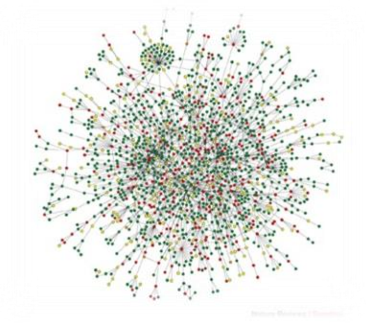


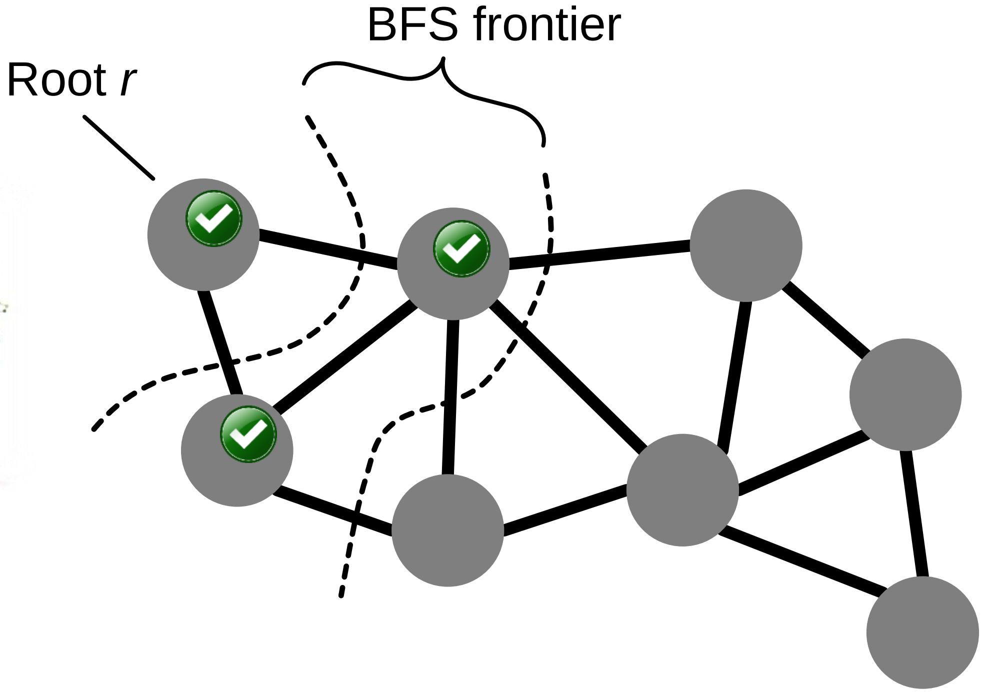
BFS

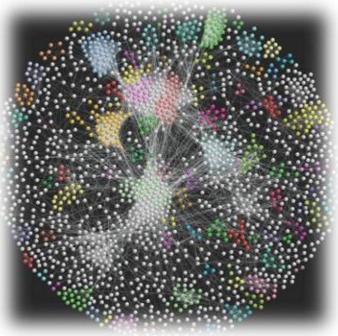
Top-Down vs. Bottom-Up [1]





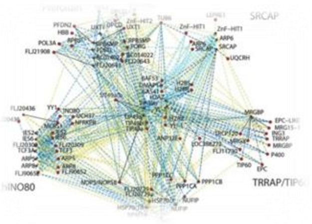


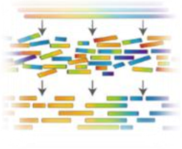


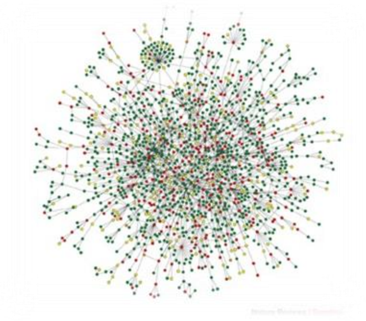


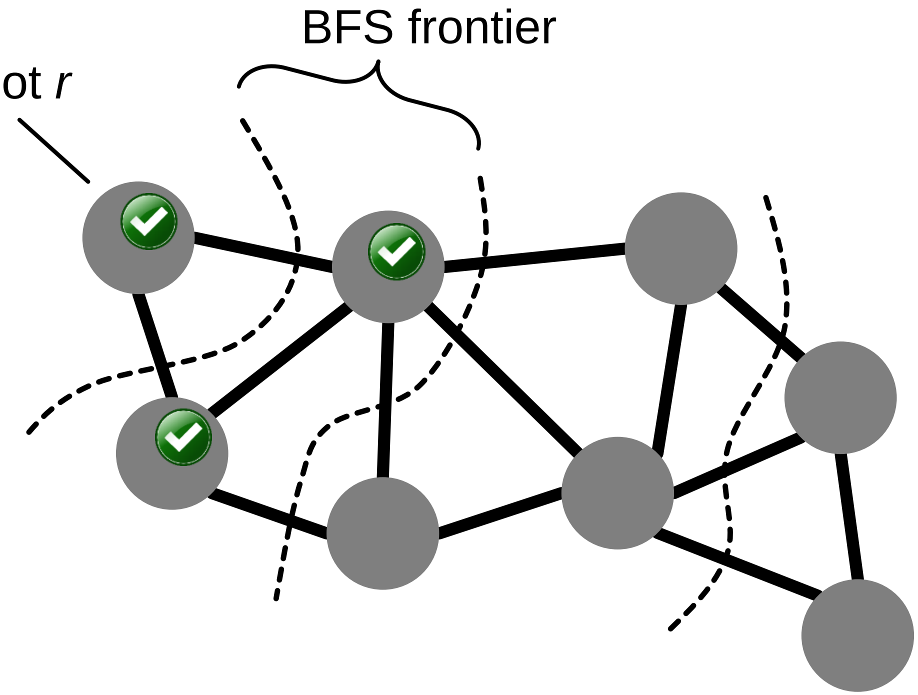
BFS

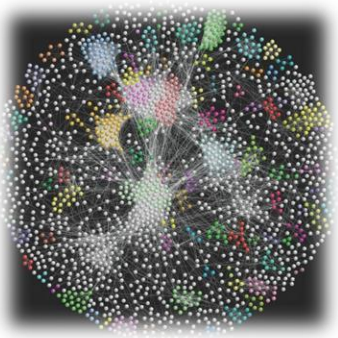
TOP-DOWN VS. BOTTOM-UP [1]





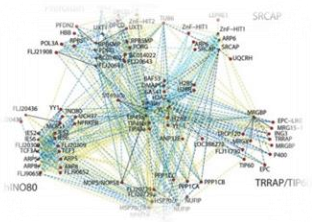


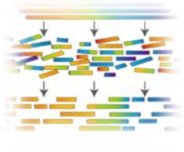


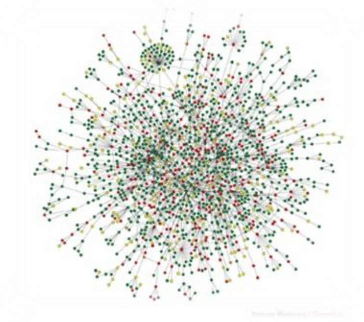


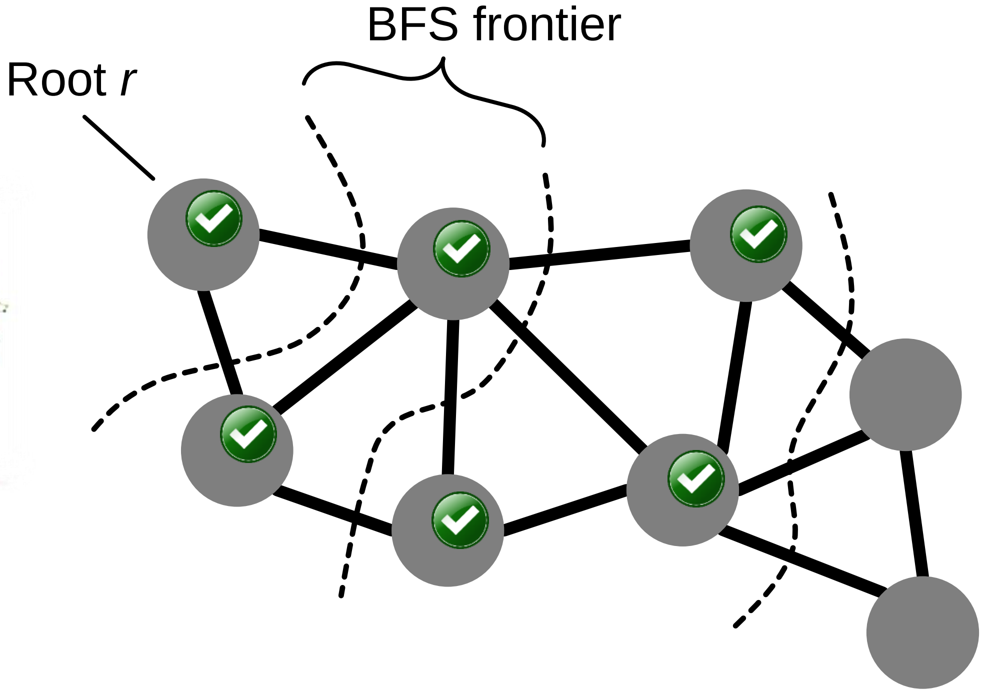
BFS

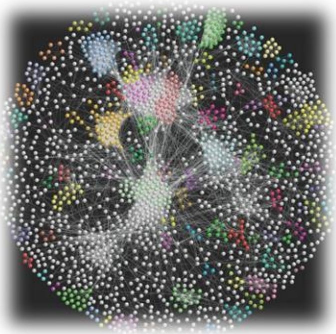
TOP-DOWN VS. BOTTOM-UP [1]



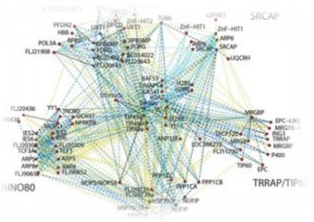


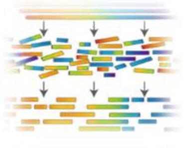


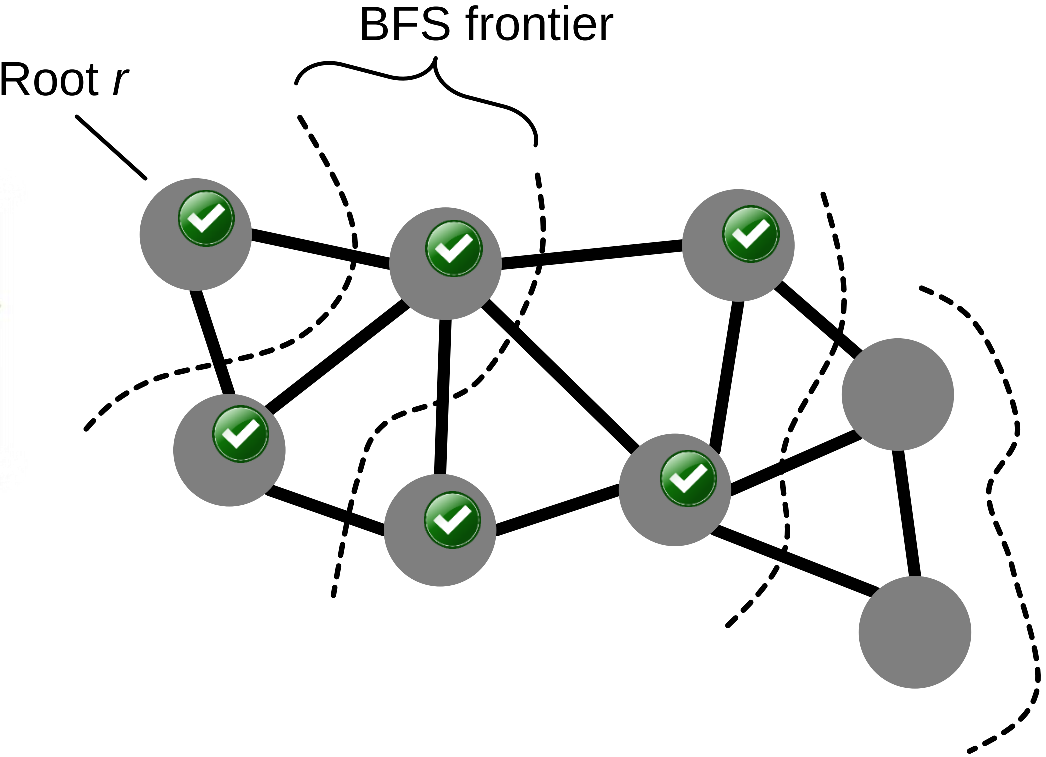


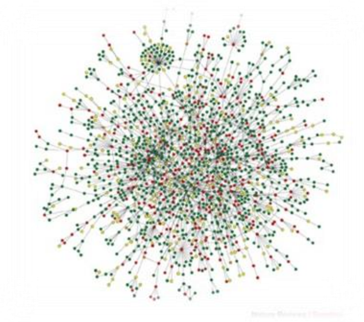


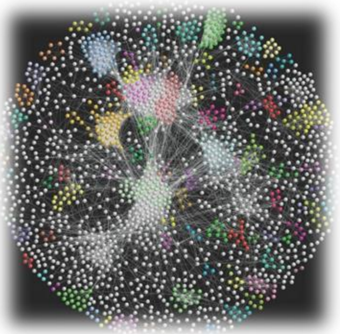
BFS
TOP-DOWN VS. BOTTOM-UP [1]



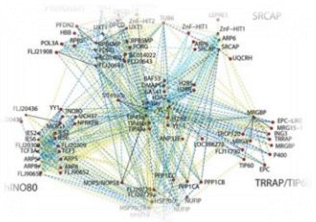


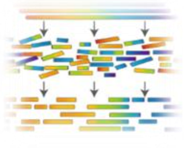


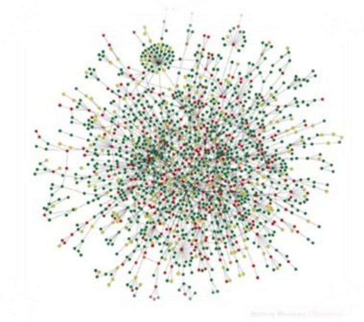


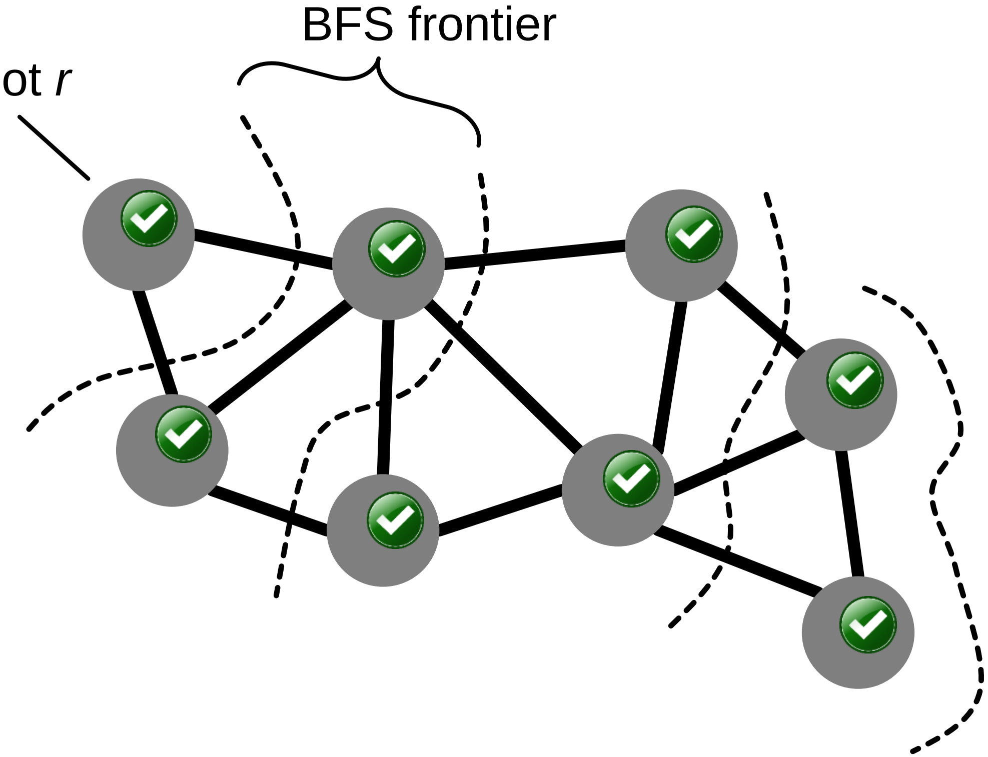


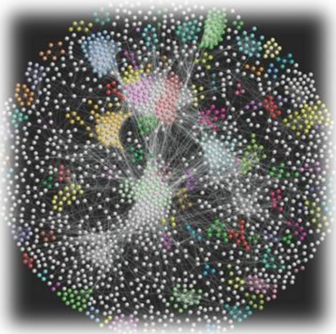
BFS
TOP-DOWN VS. BOTTOM-UP [1]







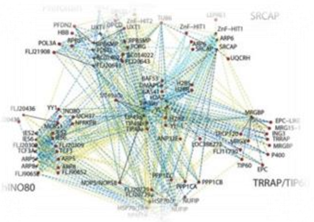


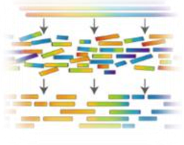


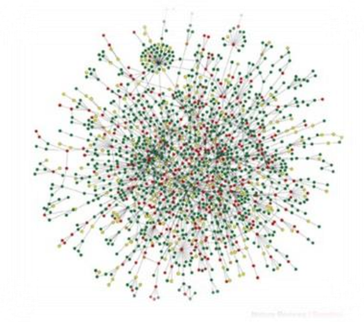
GRAPH 500

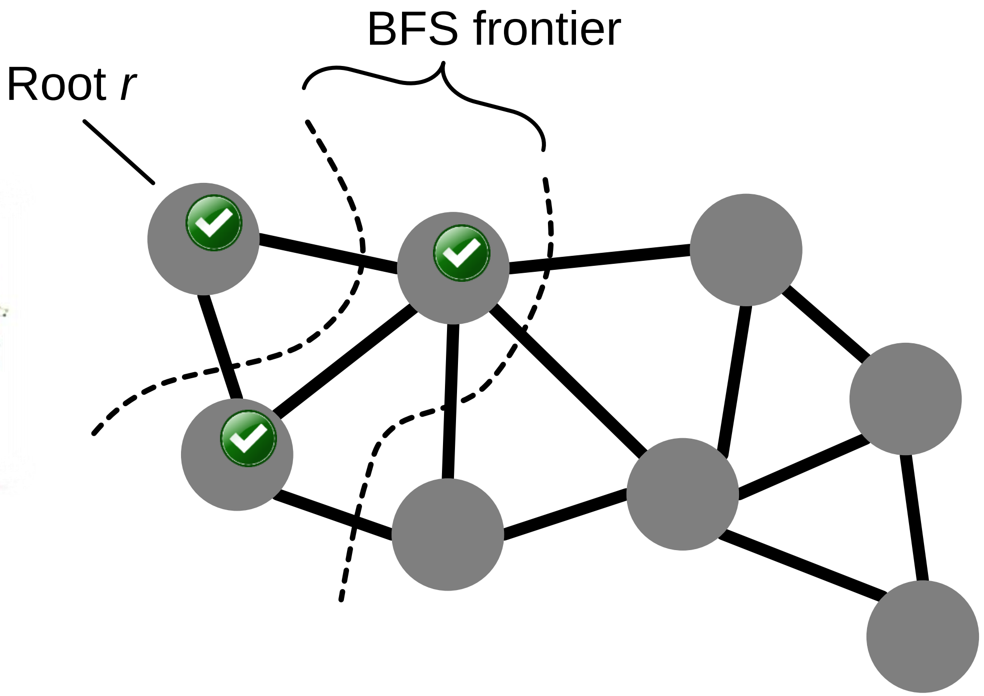
BFS

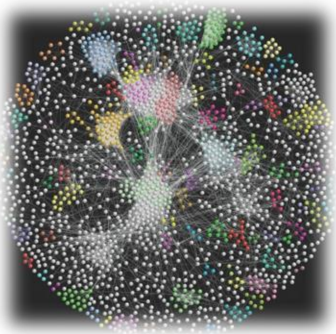
TOP-DOWN VS. BOTTOM-UP [1]



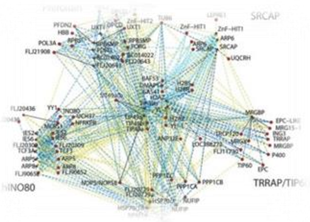


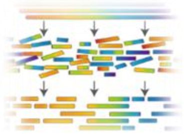




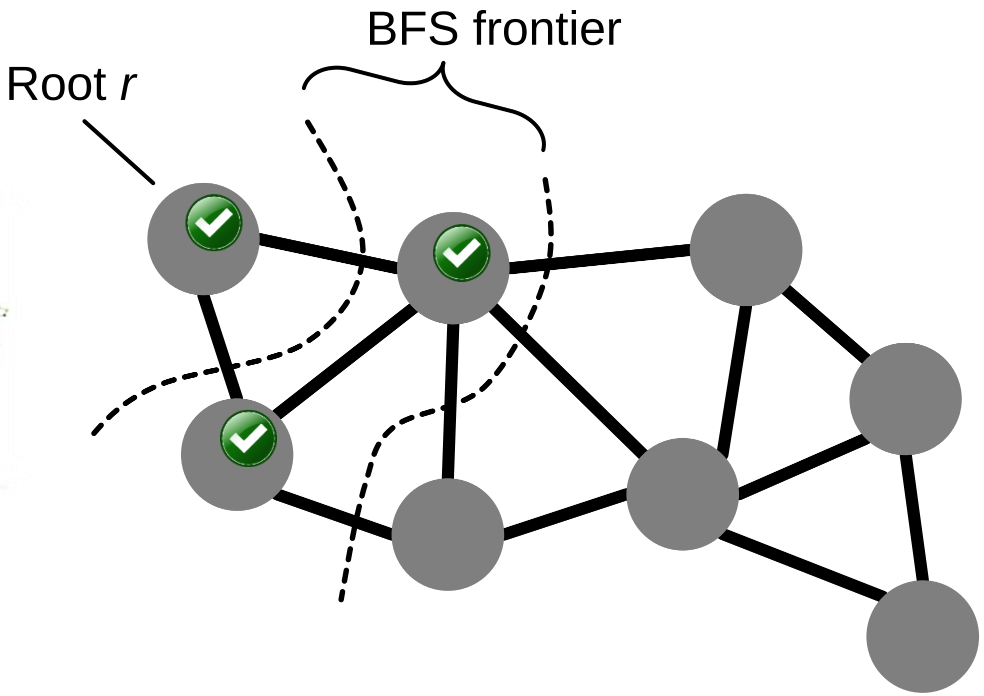


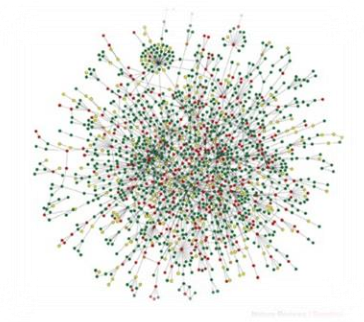
BFS
Top-Down vs. Bottom-Up [1]

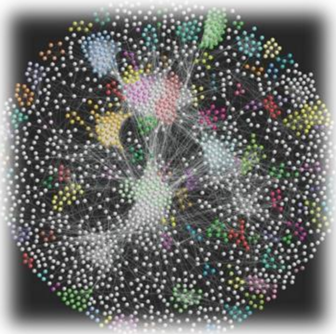




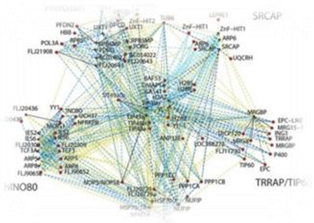
Pushing or pulling when expanding a frontier



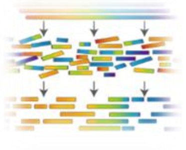


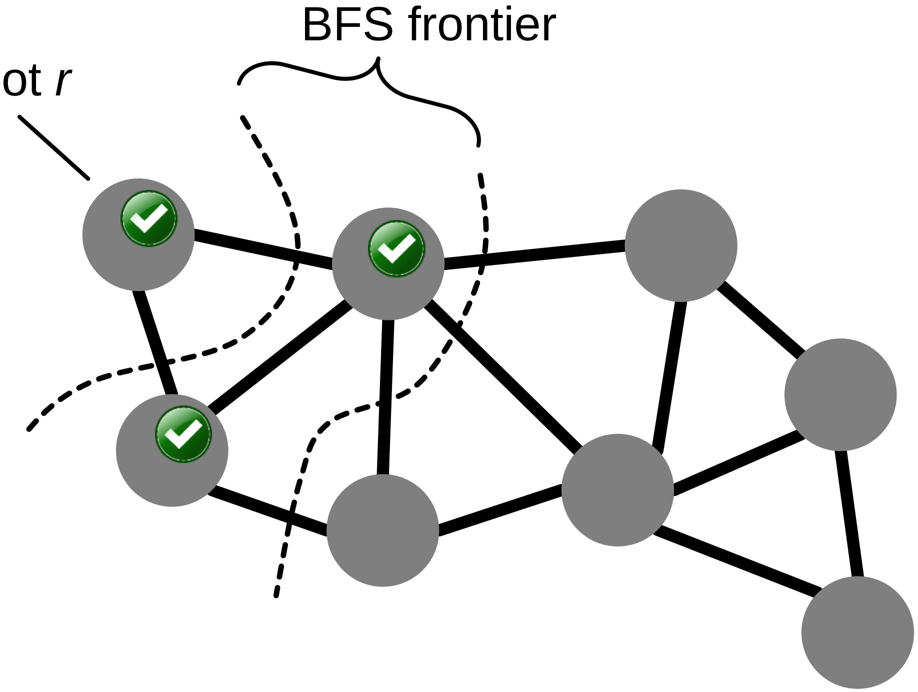


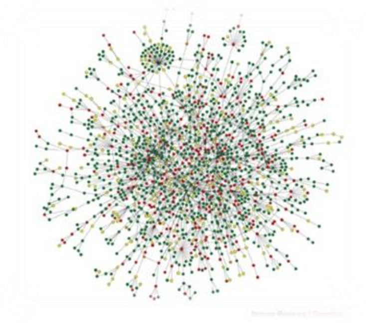
BFS
TOP-DOWN VS. BOTTOM-UP [1]

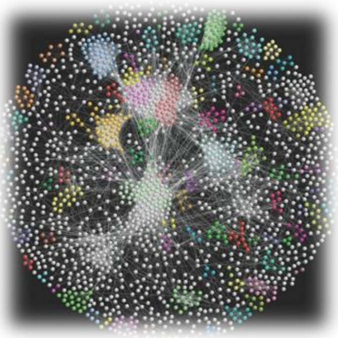


Pushing or pulling when expanding a frontier







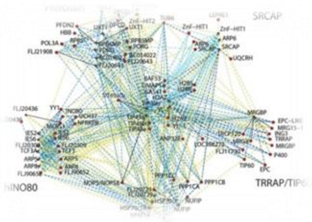


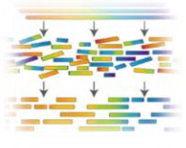
Pushing

[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

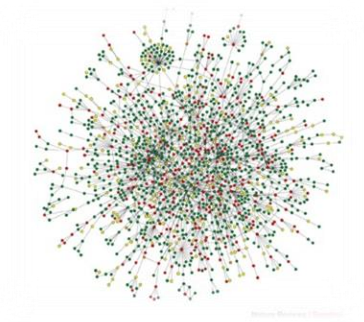
BFS

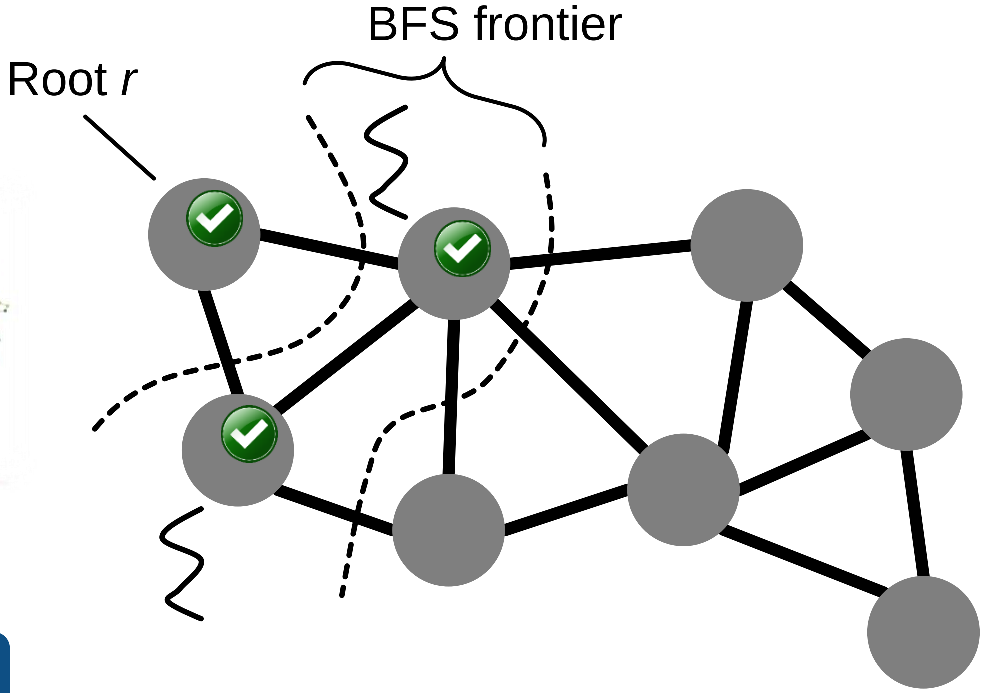
TOP-DOWN VS. BOTTOM-UP [1]

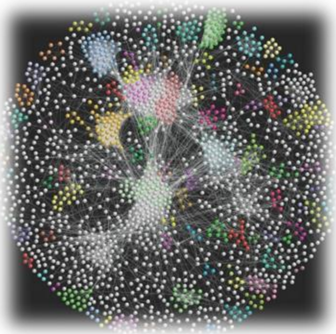




Pushing or pulling when expanding a frontier



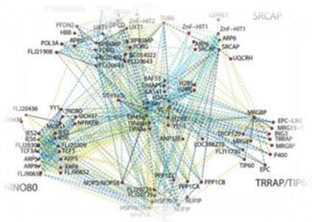


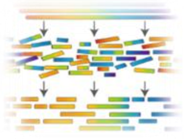


Pushing

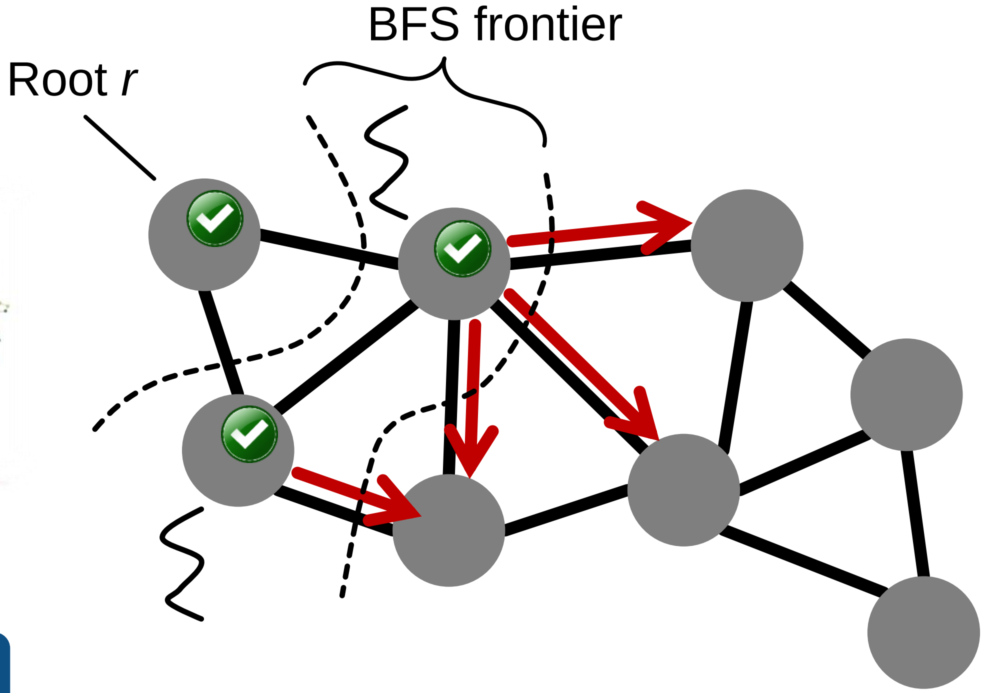
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

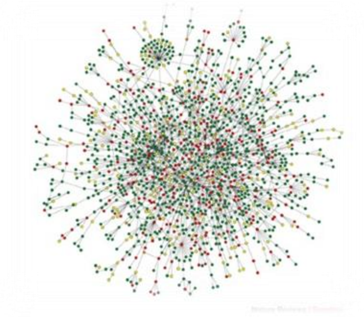
BFS
TOP-DOWN VS. BOTTOM-UP [1]

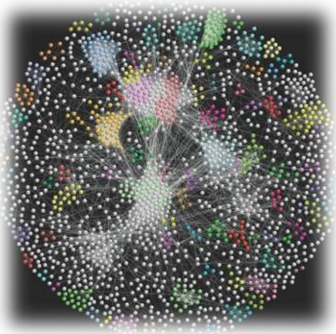




Pushing or pulling when expanding a frontier





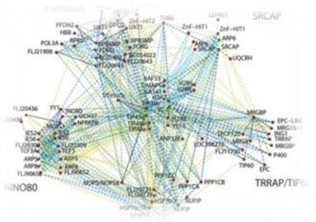


Pushing

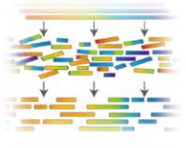
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

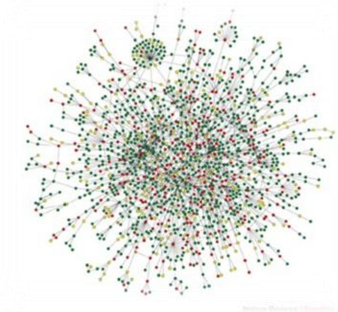
BFS

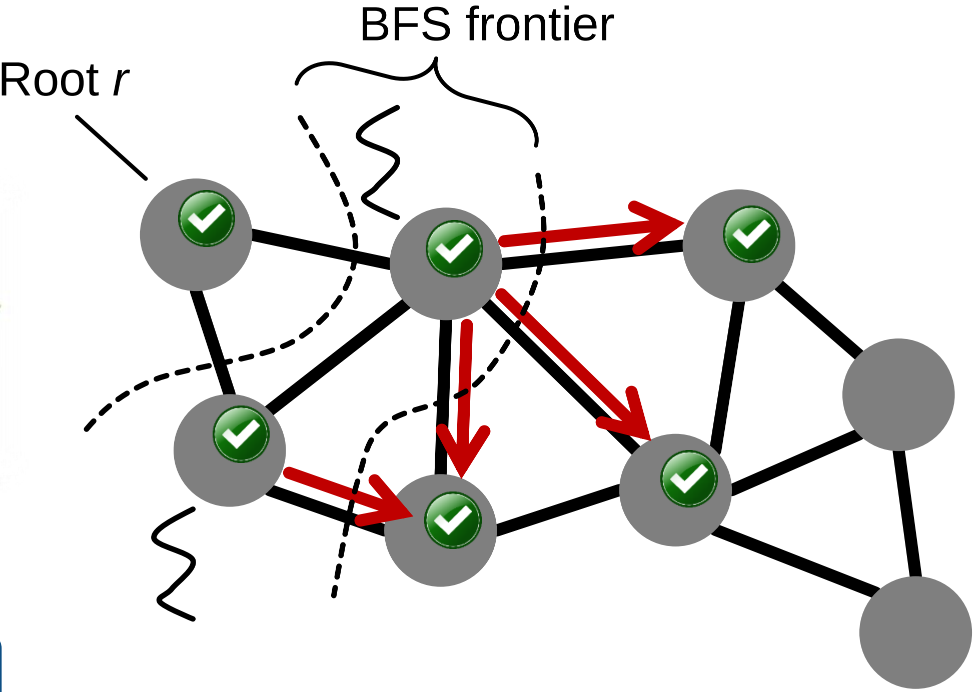
TOP-DOWN VS. BOTTOM-UP [1]

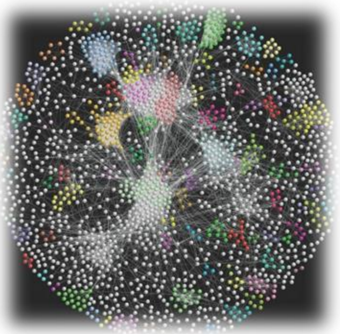


Pushing or pulling when expanding a frontier







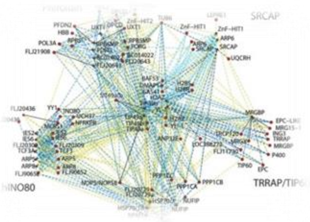


Pushing

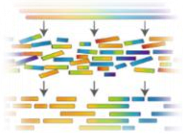
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

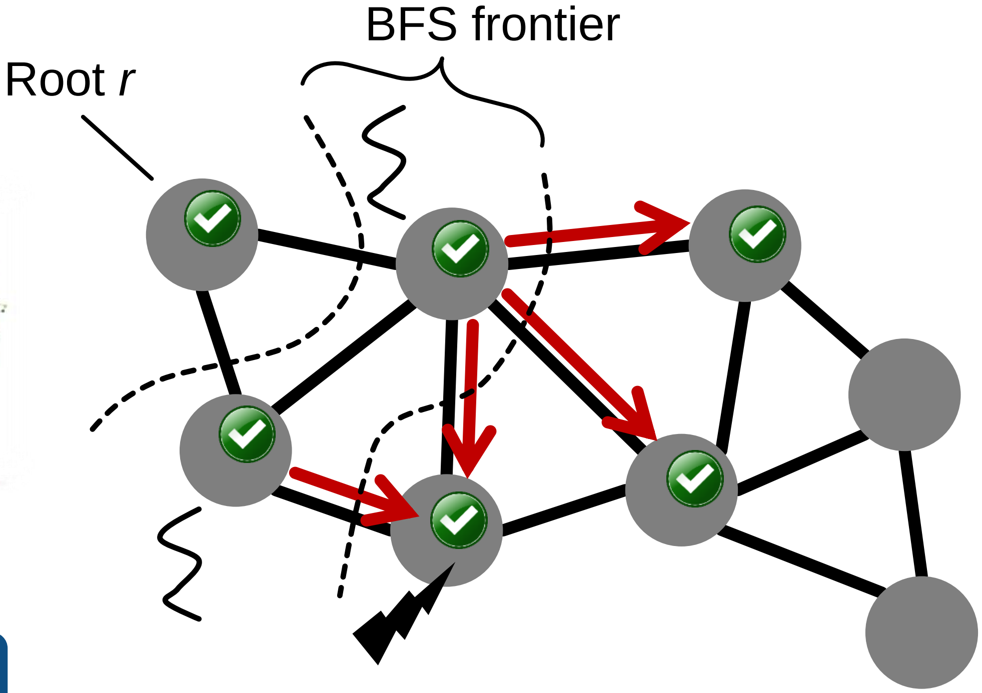
BFS

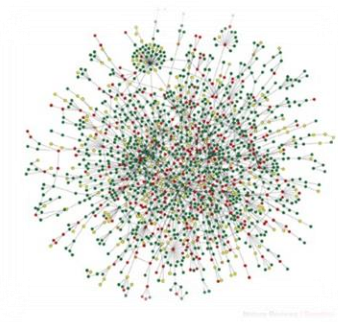
TOP-DOWN VS. BOTTOM-UP [1]

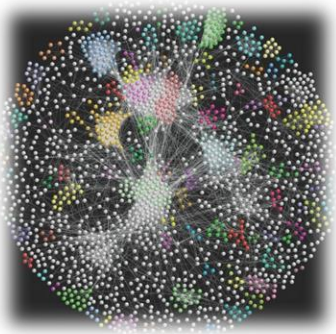


Pushing or pulling when expanding a frontier







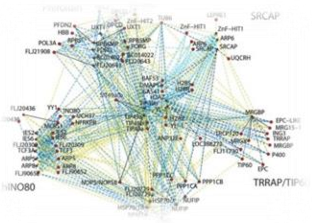


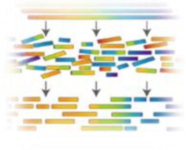
Pushing

[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

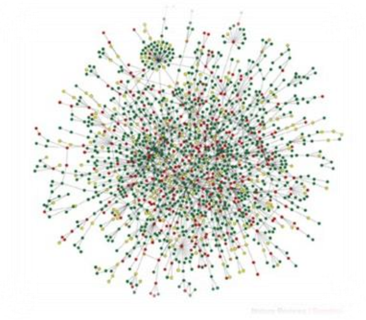
BFS

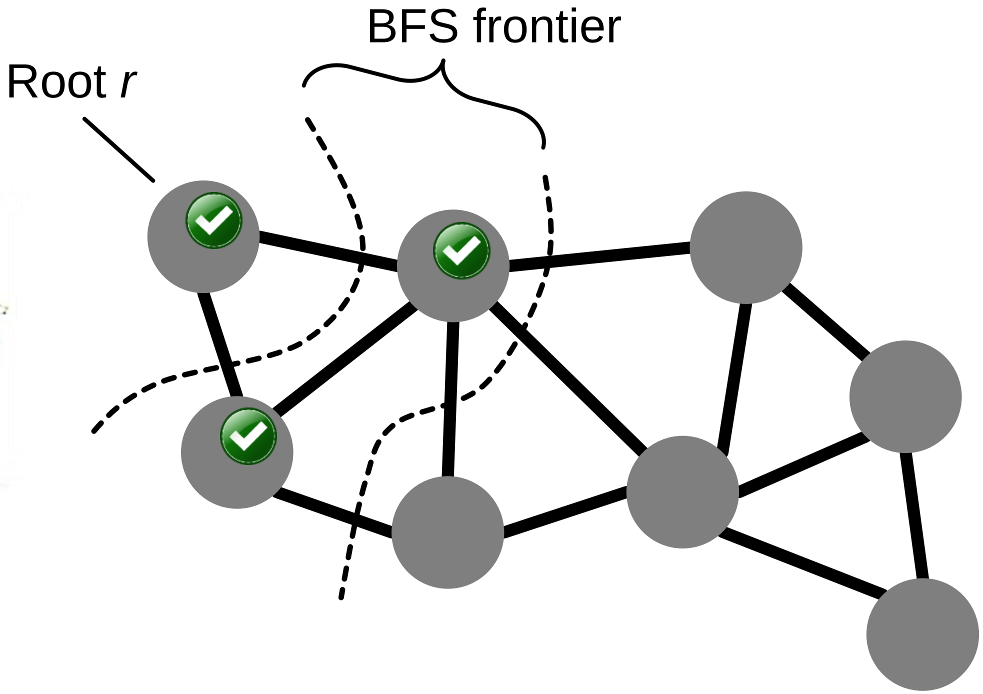
TOP-DOWN VS. BOTTOM-UP [1]

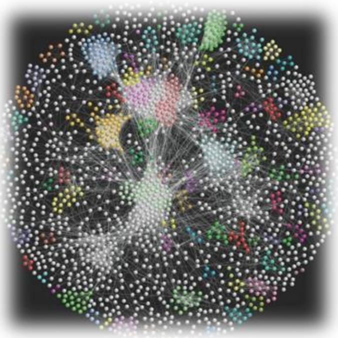




Pushing or pulling when expanding a frontier





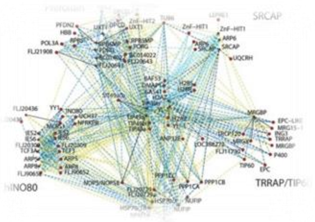


---

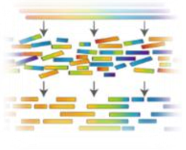
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

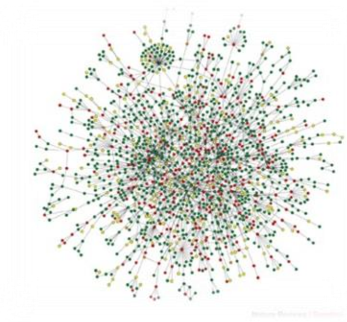
BFS

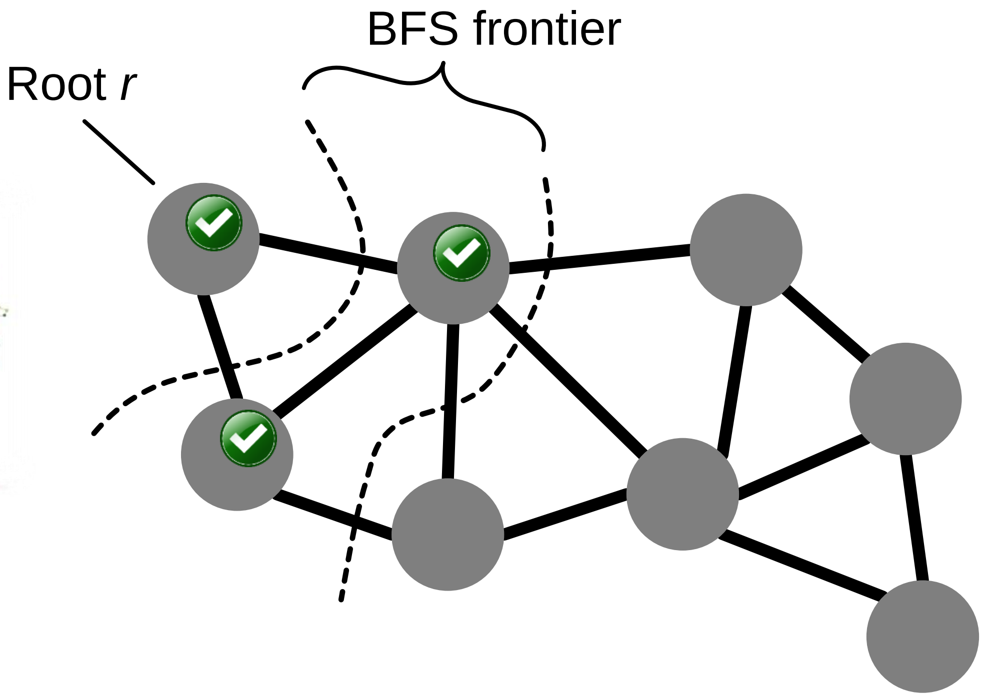
TOP-DOWN VS. BOTTOM-UP [1]

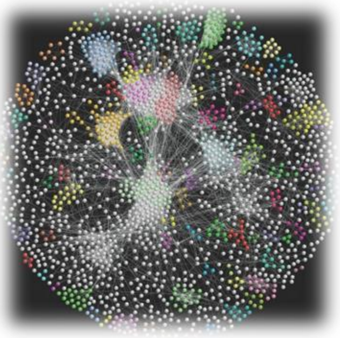


Pushing or pulling when expanding a frontier







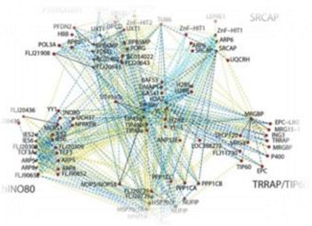


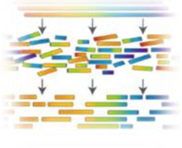
Pulling

[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

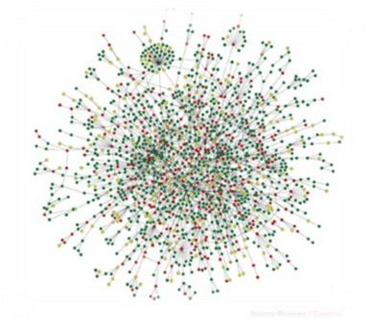
BFS

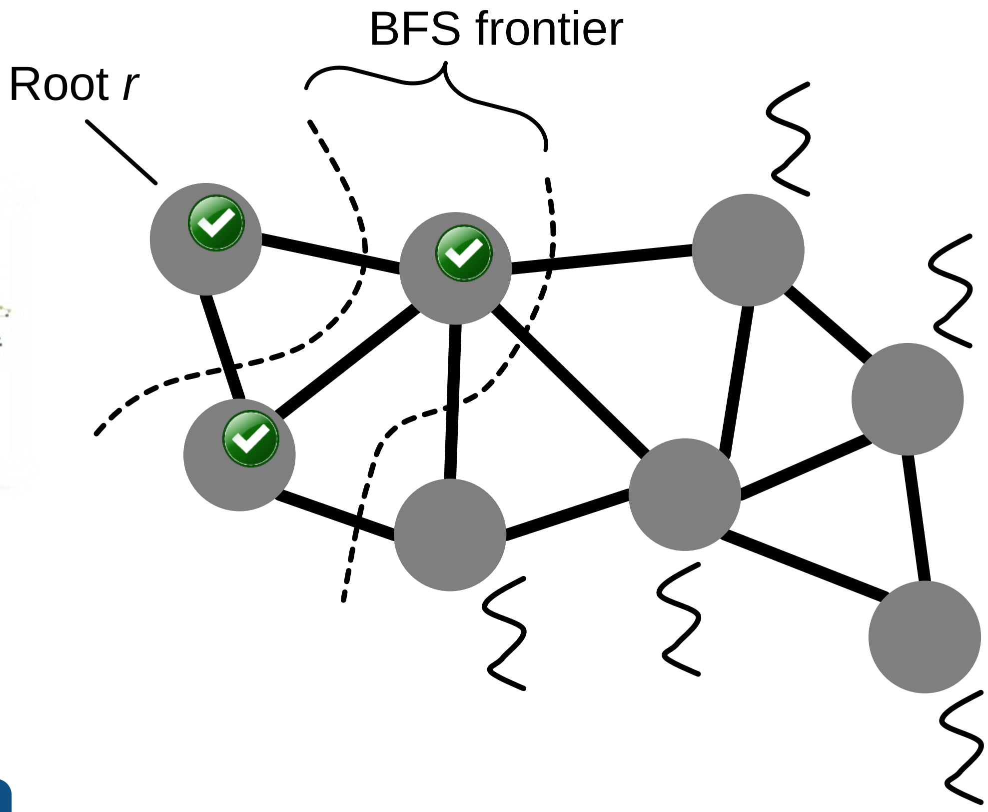
Top-Down vs. Bottom-Up [1]

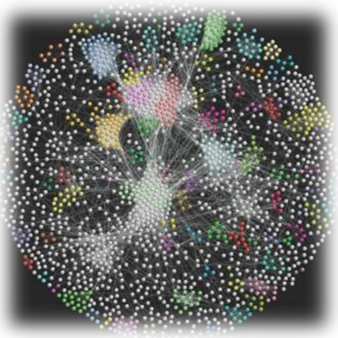




Pushing or pulling when expanding a frontier







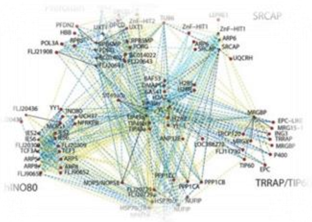
Pulling

---

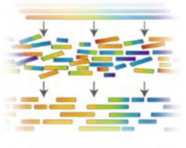
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

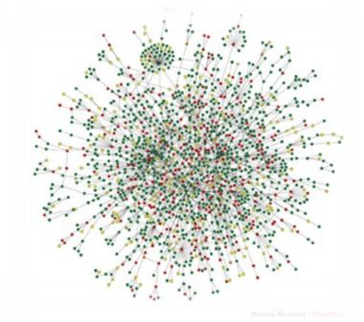
BFS

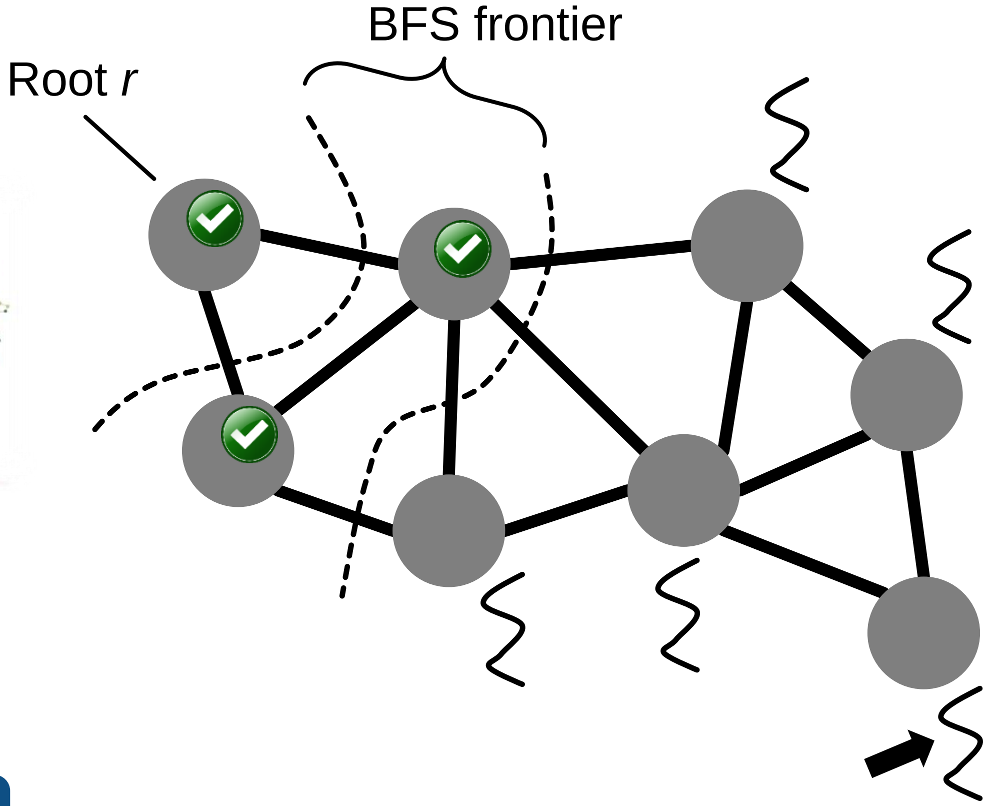
TOP-DOWN VS. BOTTOM-UP [1]

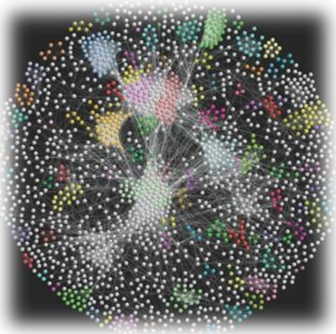


Pushing or pulling when expanding a frontier









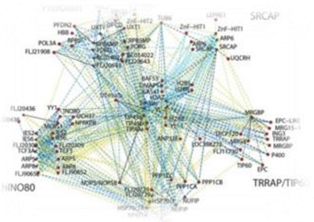
Pulling

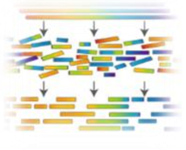
---

[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

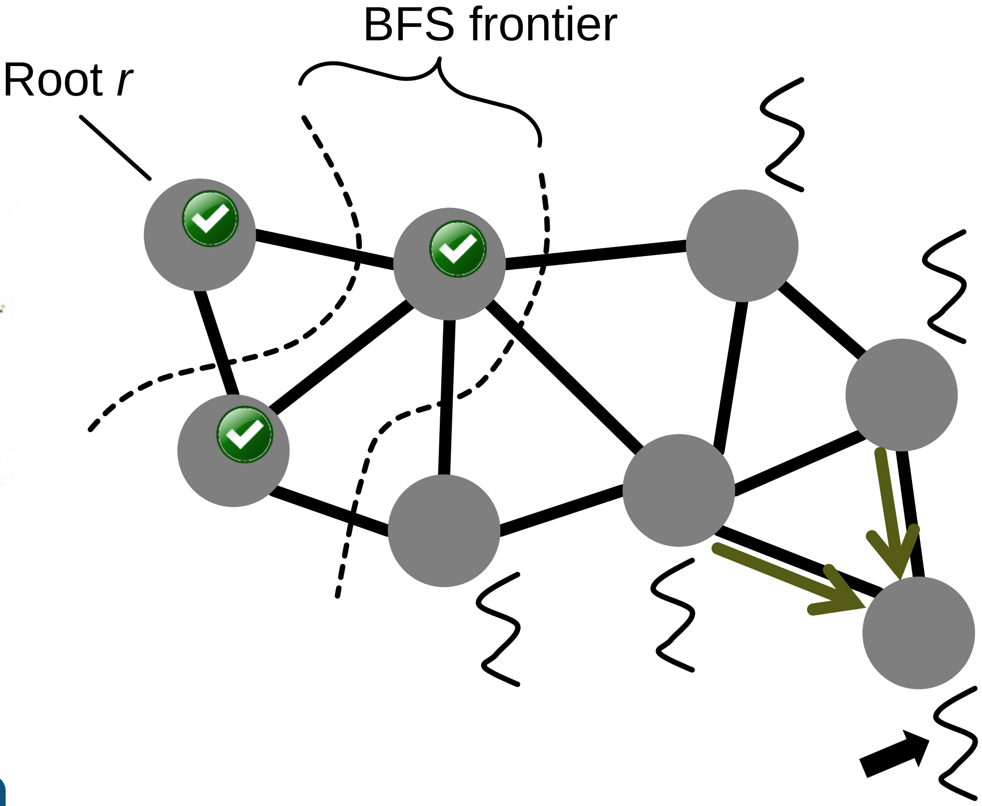
BFS

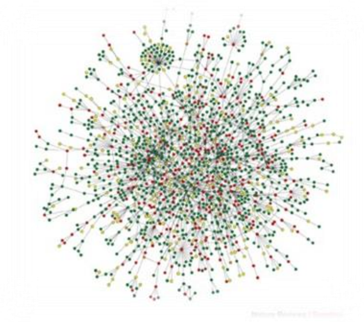
Top-Down vs. Bottom-Up [1]

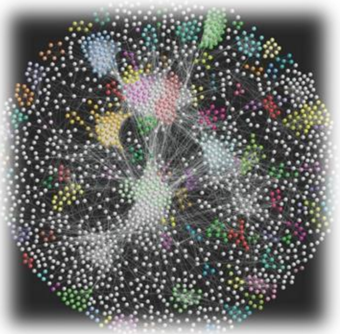




Pushing or pulling when expanding a frontier





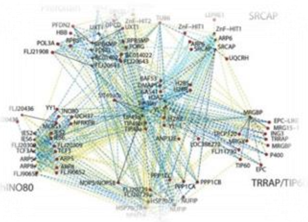


Pulling

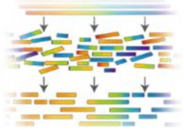
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

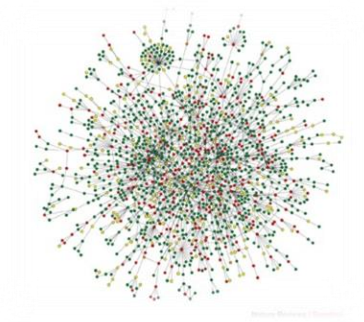
BFS

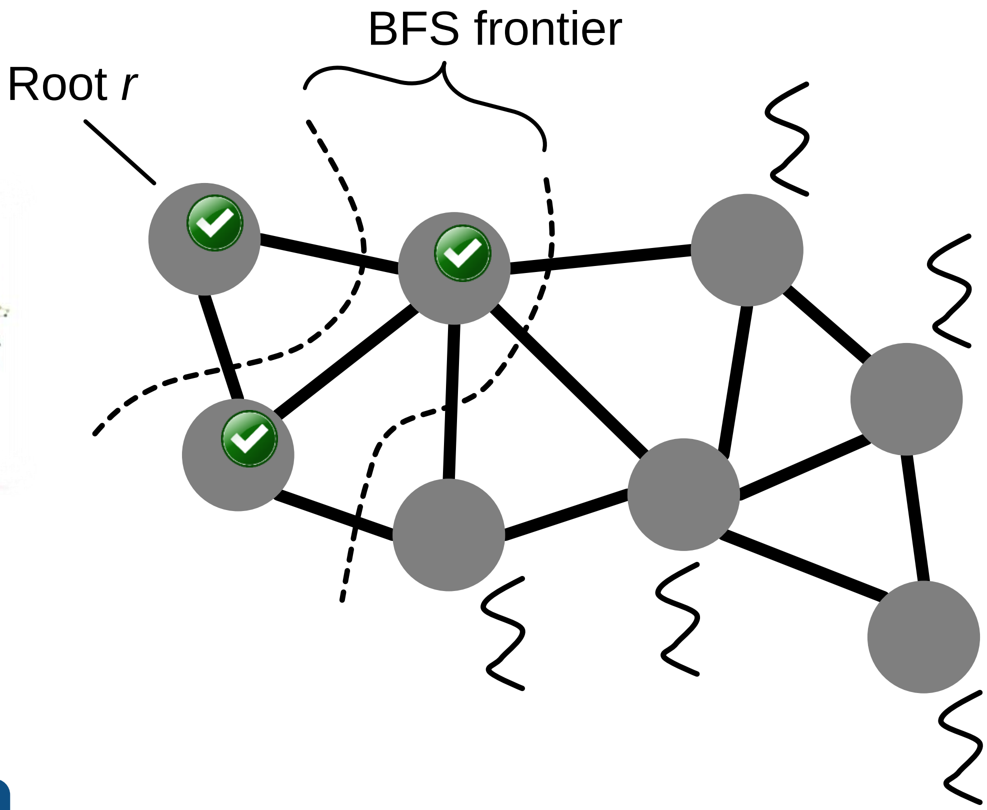
TOP-DOWN VS. BOTTOM-UP [1]

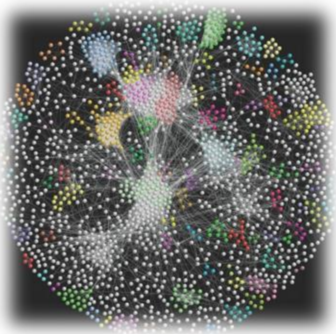


Pushing or pulling when expanding a frontier





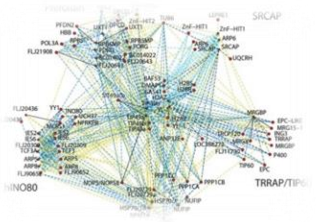


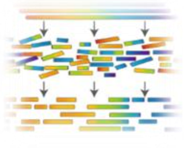


Pulling

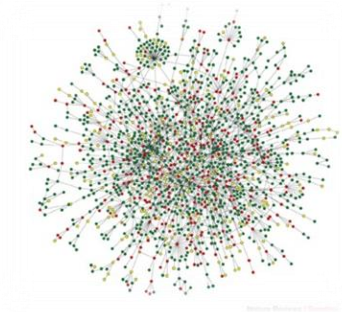
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

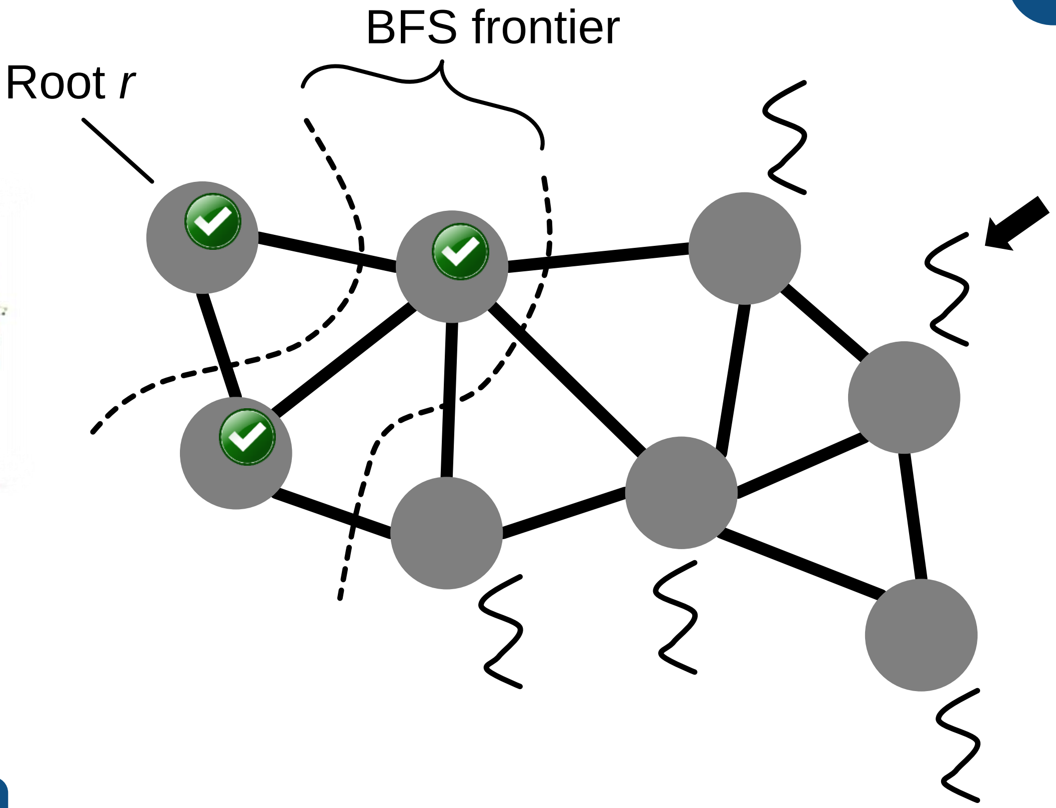
BFS
TOP-DOWN VS. BOTTOM-UP [1]

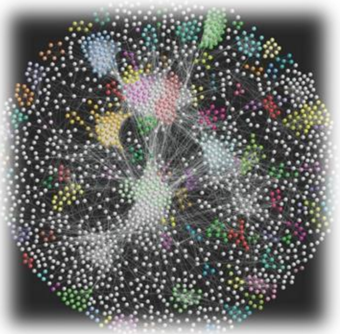




Pushing or pulling when expanding a frontier



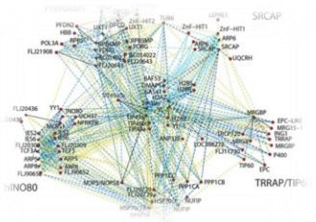




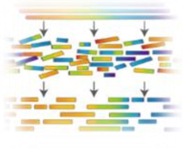
Pulling

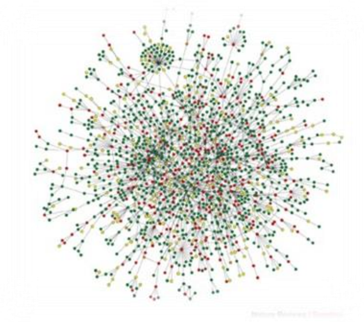
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

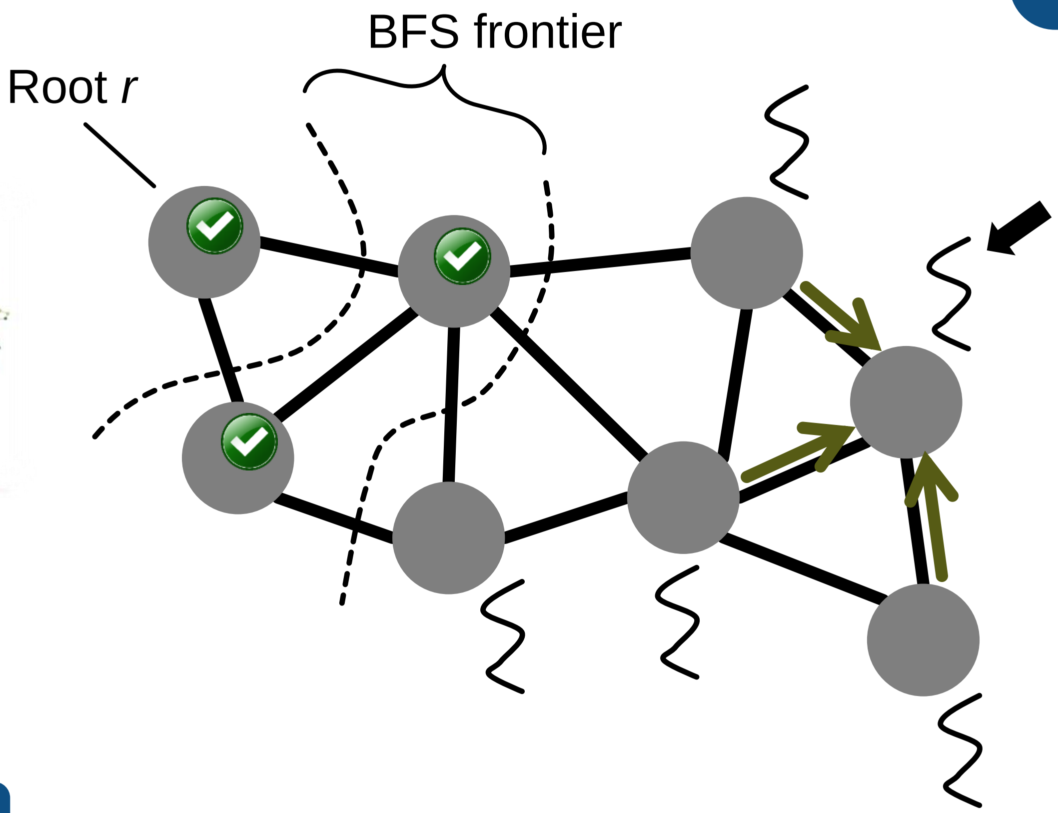
BFS
TOP-DOWN VS. BOTTOM-UP [1]

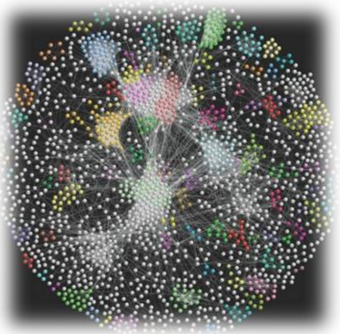


Pushing or pulling when expanding a frontier







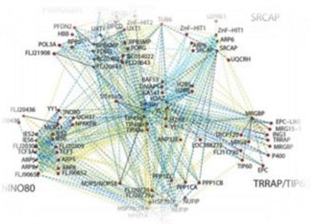


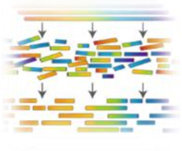
Pulling

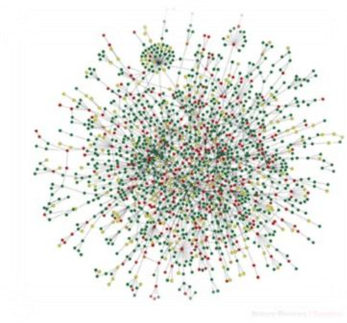
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

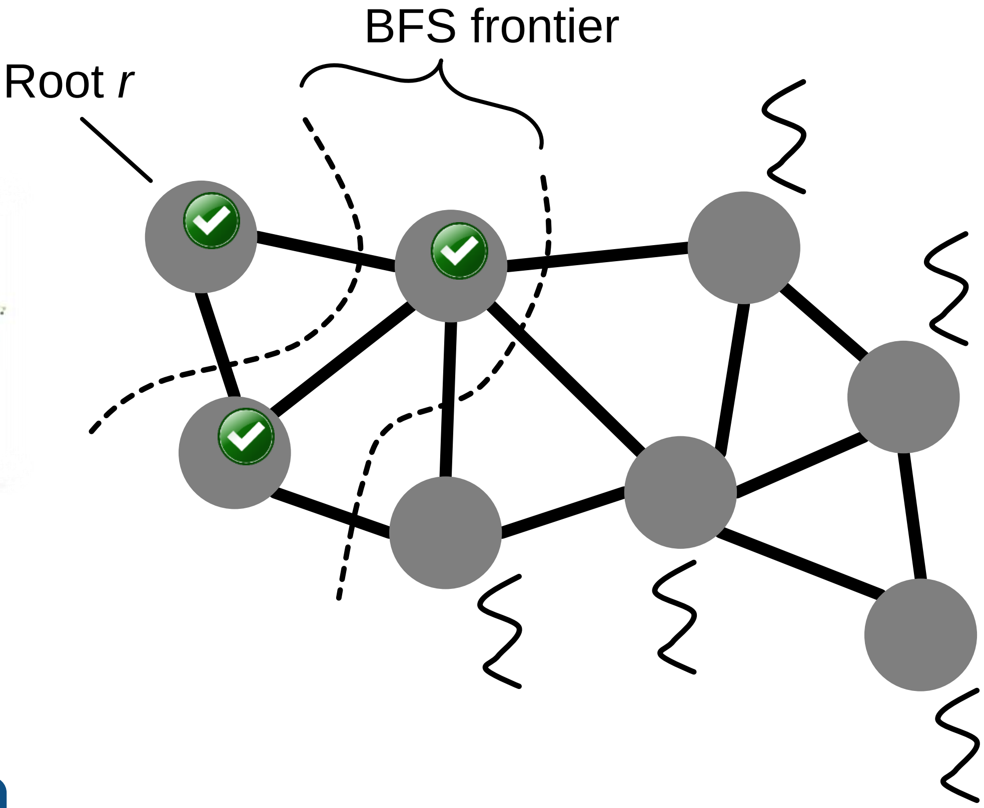
BFS
TOP-DOWN VS. BOTTOM-UP [1]

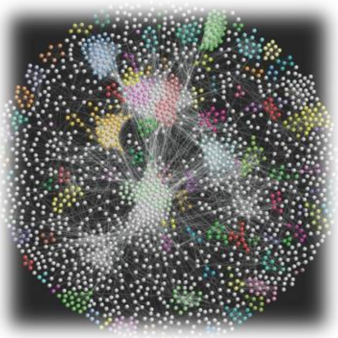
Pushing or pulling when expanding a frontier









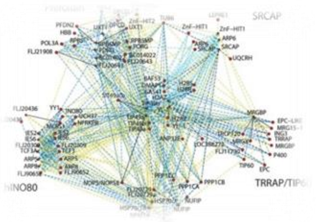


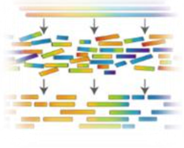
Pulling

---

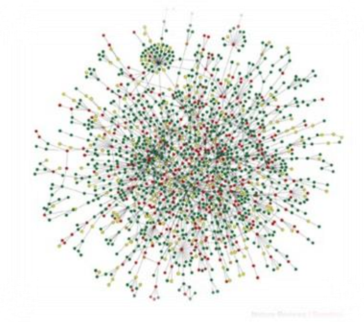
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

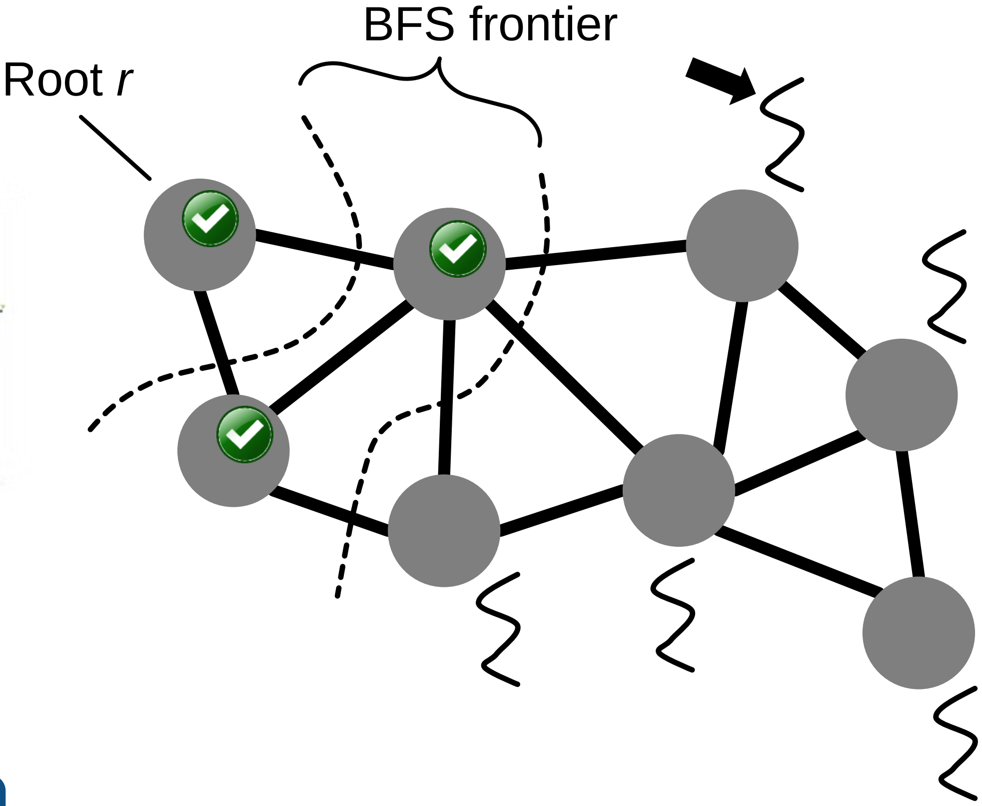
BFS
TOP-DOWN VS. BOTTOM-UP [1]

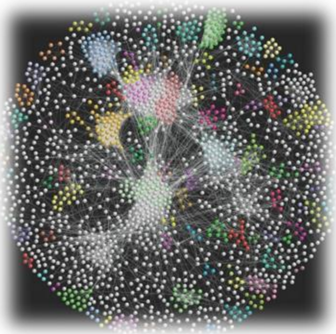




Pushing or pulling when expanding a frontier



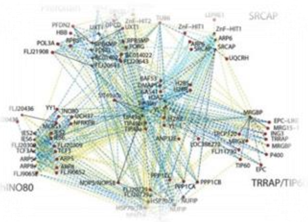


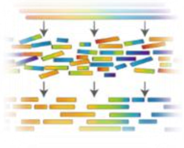


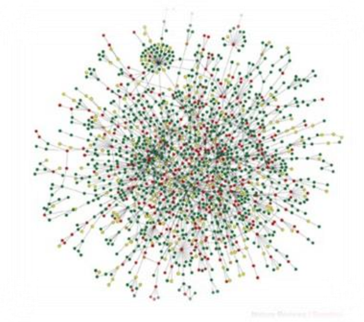
Pulling

[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

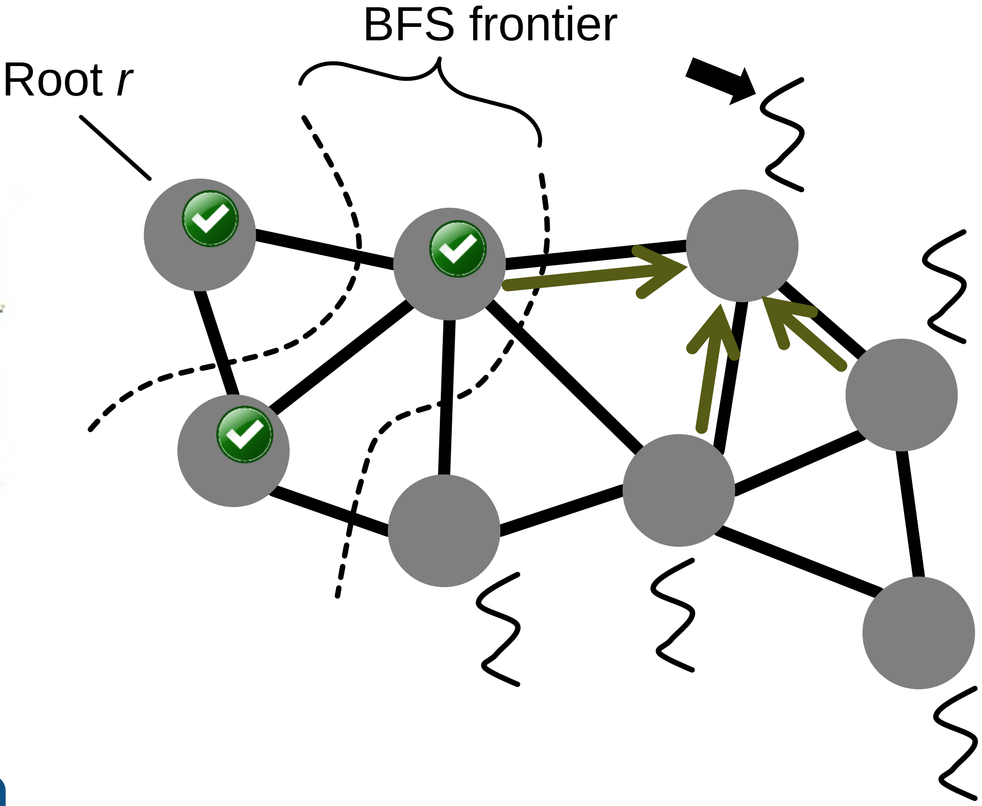
BFS
TOP-DOWN VS. BOTTOM-UP [1]

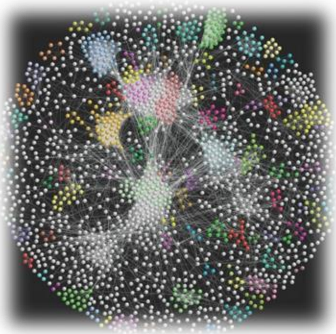






Pushing or pulling when expanding a frontier





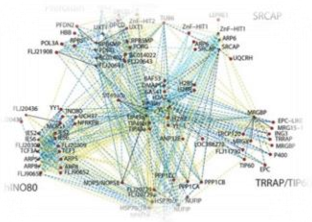
Pulling

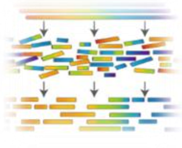
---

[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

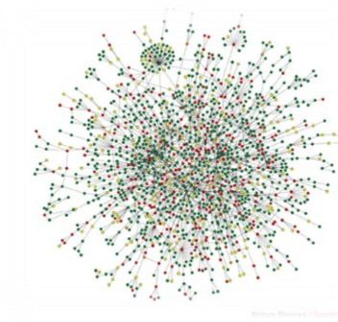
BFS

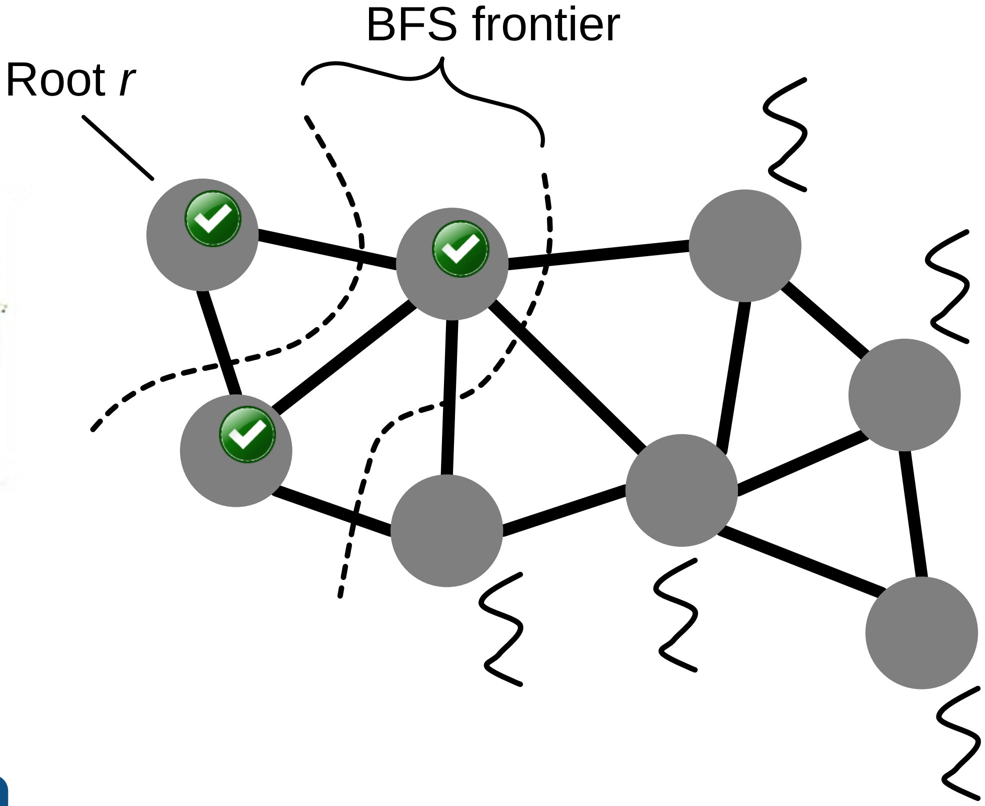
TOP-DOWN VS. BOTTOM-UP [1]

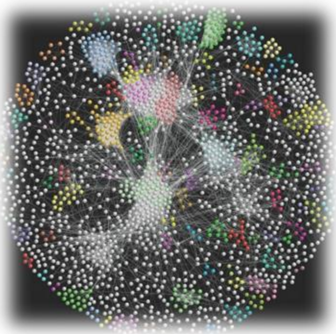




Pushing or pulling when expanding a frontier



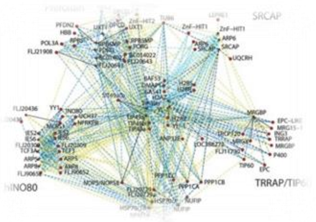


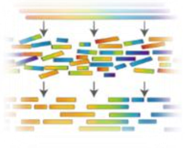


Pulling

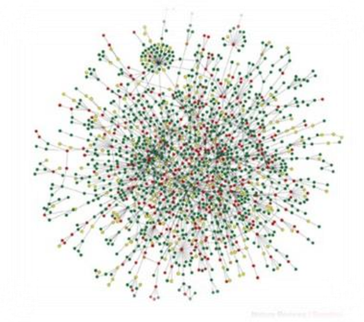
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

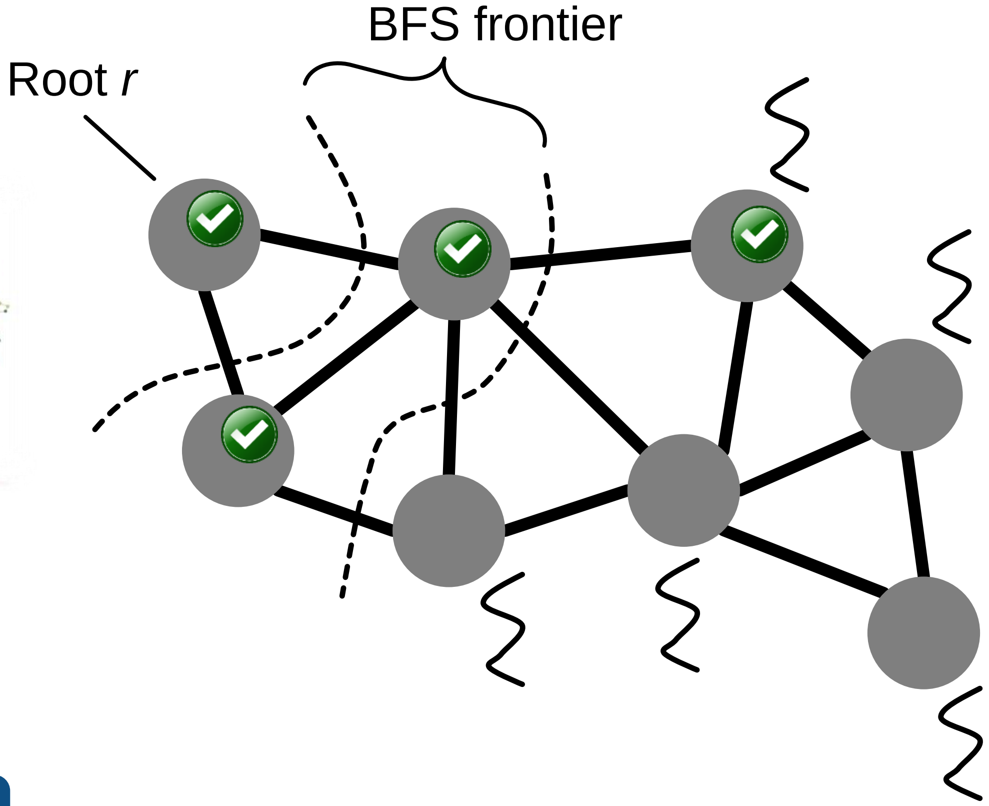
BFS
TOP-DOWN VS. BOTTOM-UP [1]

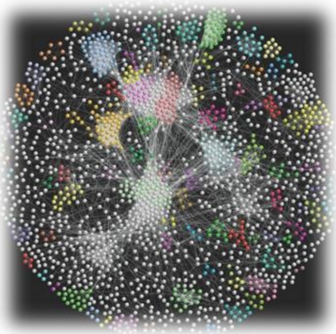




Pushing or pulling when expanding a frontier





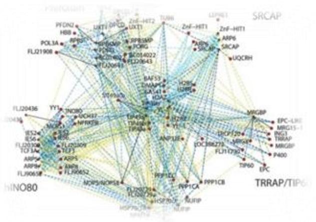


Pulling

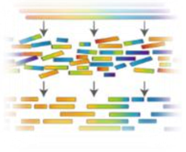
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

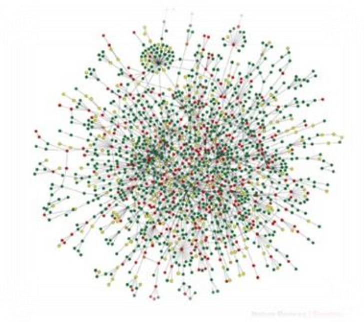
BFS

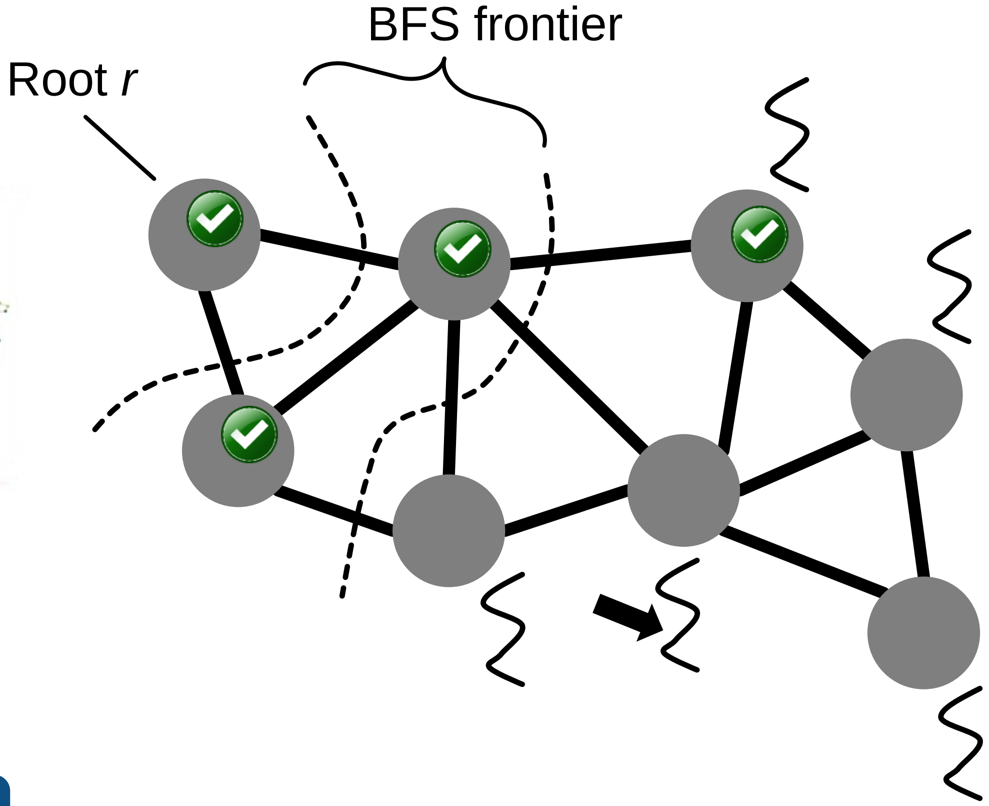
TOP-DOWN VS. BOTTOM-UP [1]

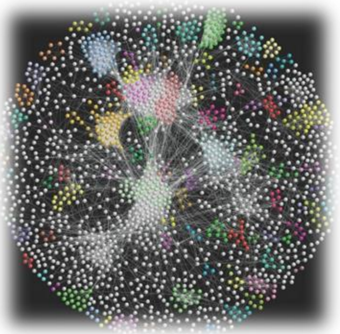


Pushing or pulling when expanding a frontier









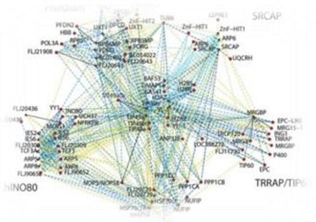
Pulling

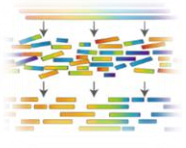
---

[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

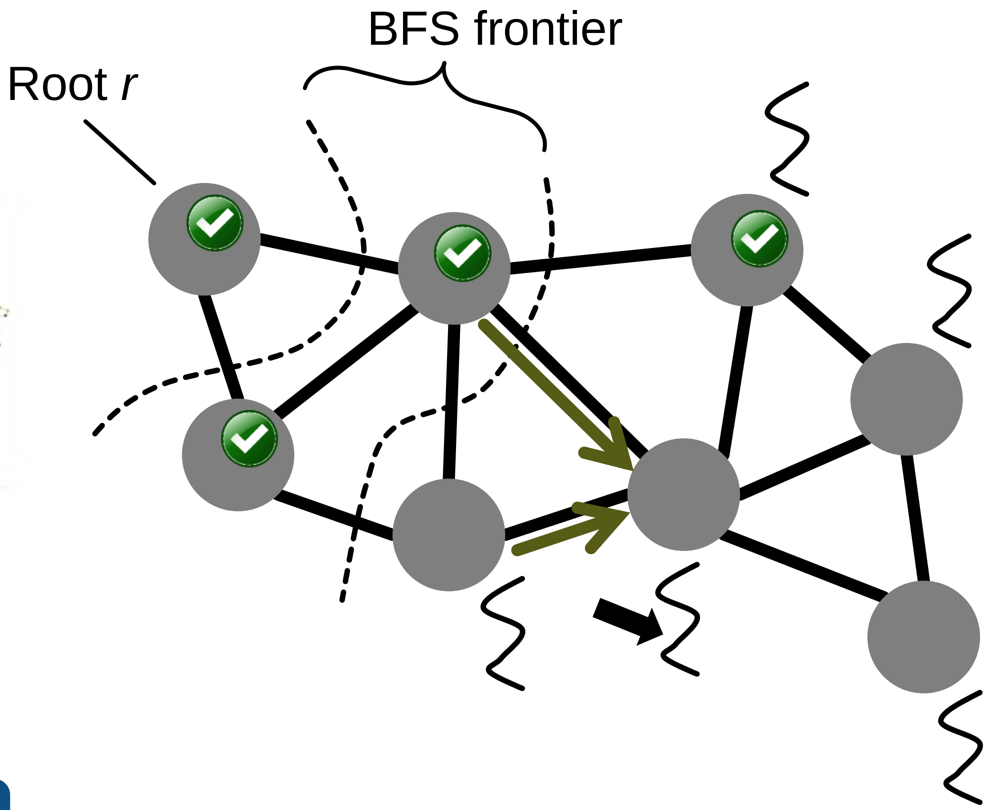
BFS

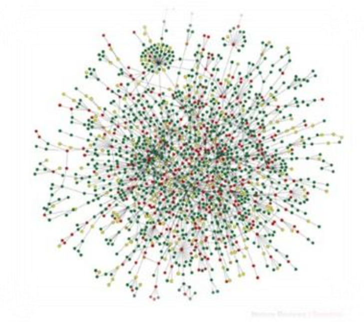
TOP-DOWN VS. BOTTOM-UP [1]

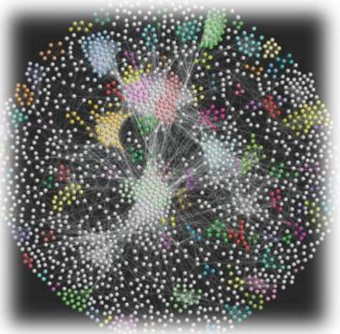




Pushing or pulling when expanding a frontier



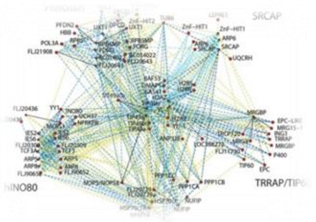


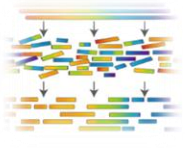


Pulling

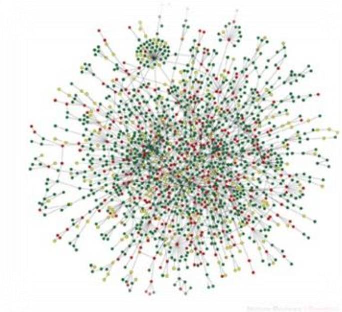
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

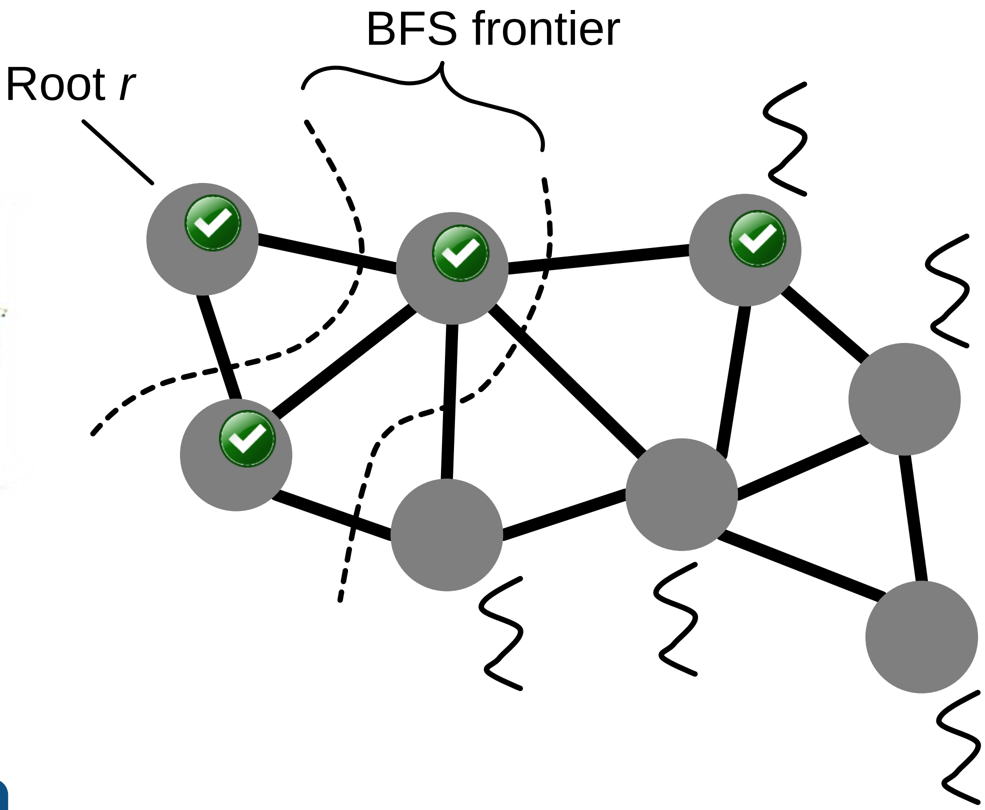
BFS
TOP-DOWN VS. BOTTOM-UP [1]

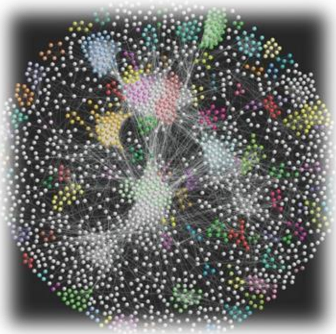




Pushing or pulling when expanding a frontier





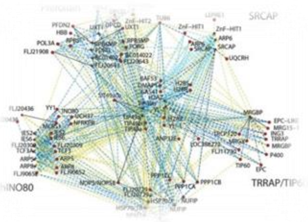


Pulling

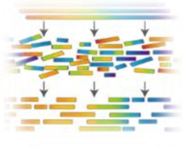
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

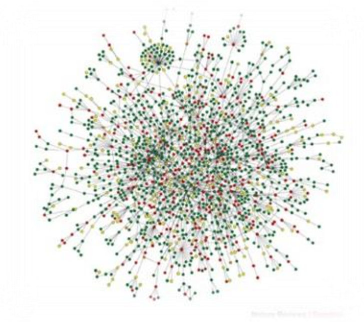
BFS

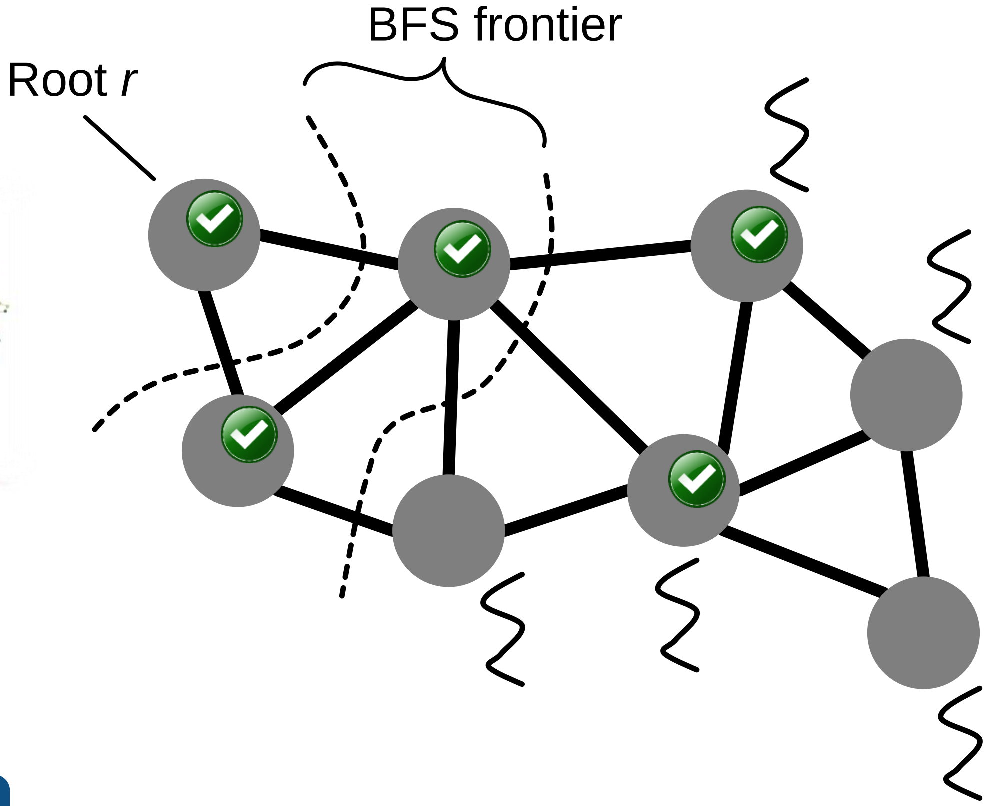
TOP-DOWN VS. BOTTOM-UP [1]

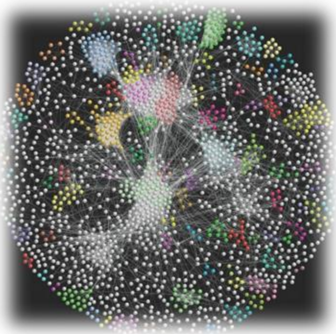


Pushing or pulling when expanding a frontier





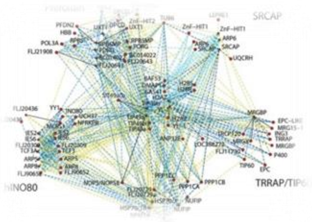


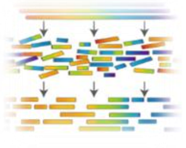


Pulling

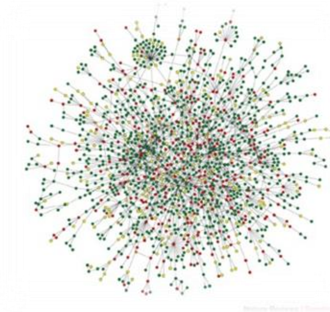
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

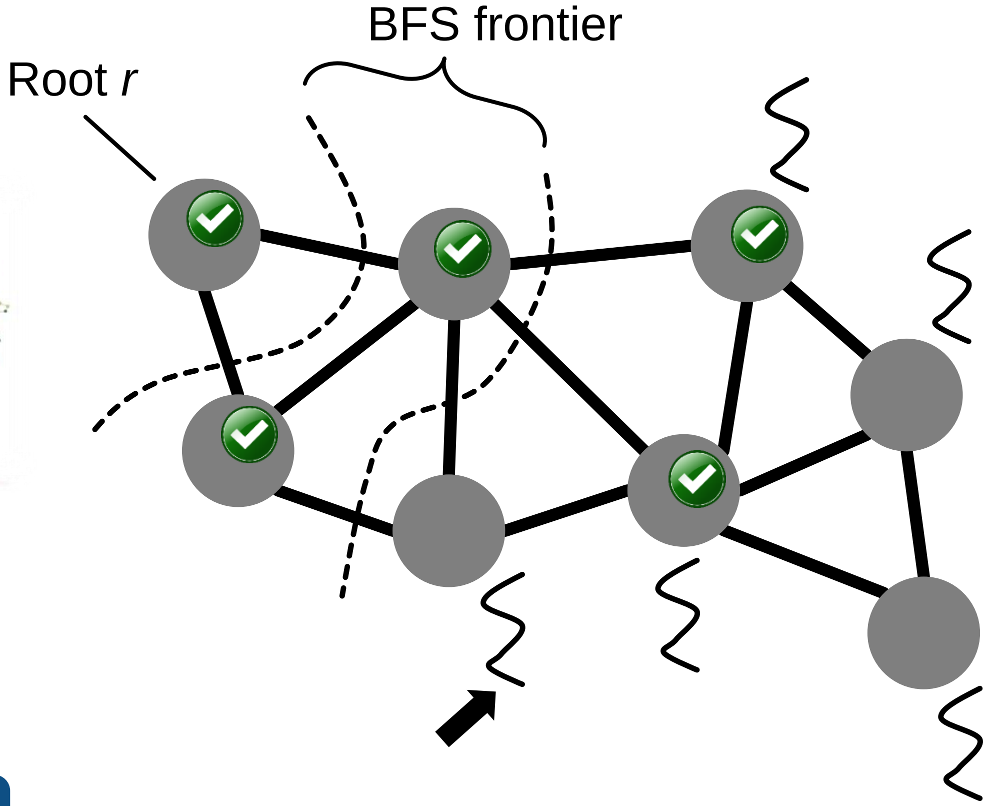
BFS
TOP-DOWN VS. BOTTOM-UP [1]

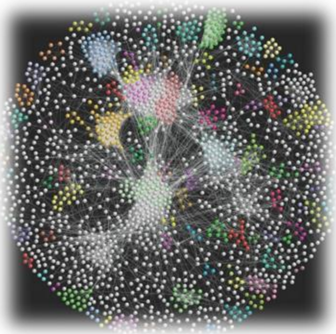




Pushing or pulling when expanding a frontier





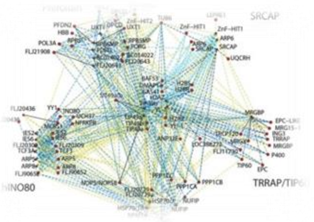


Pulling

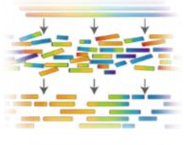
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

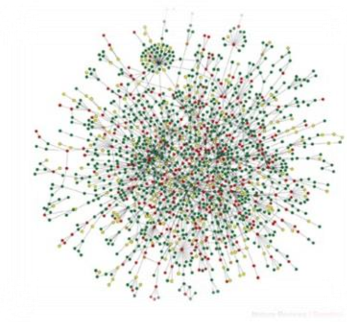
BFS

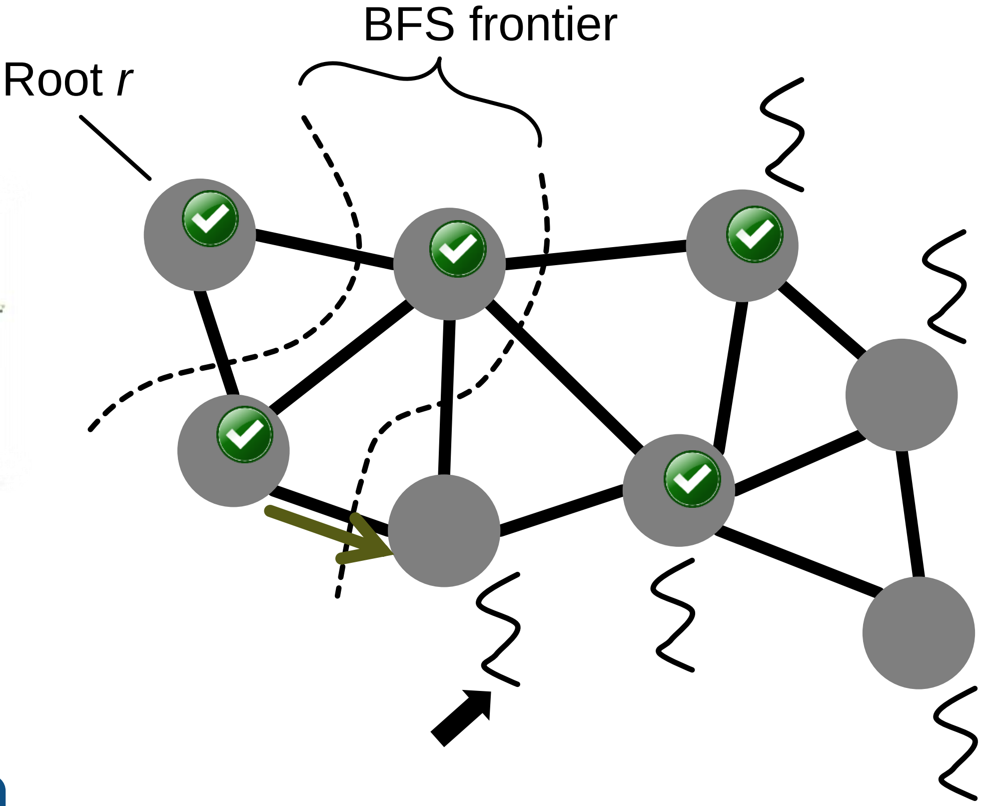
TOP-DOWN VS. BOTTOM-UP [1]

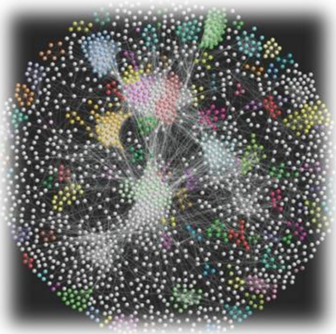


Pushing or pulling when expanding a frontier





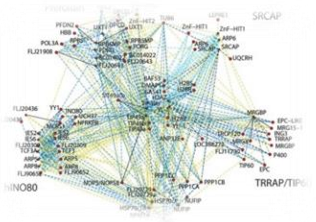




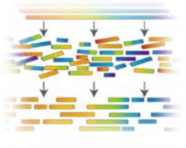
Pulling

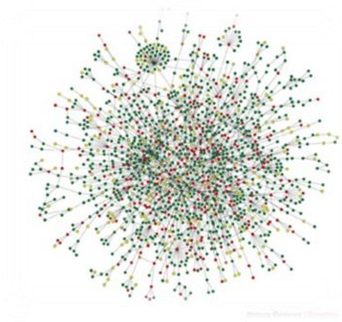
[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

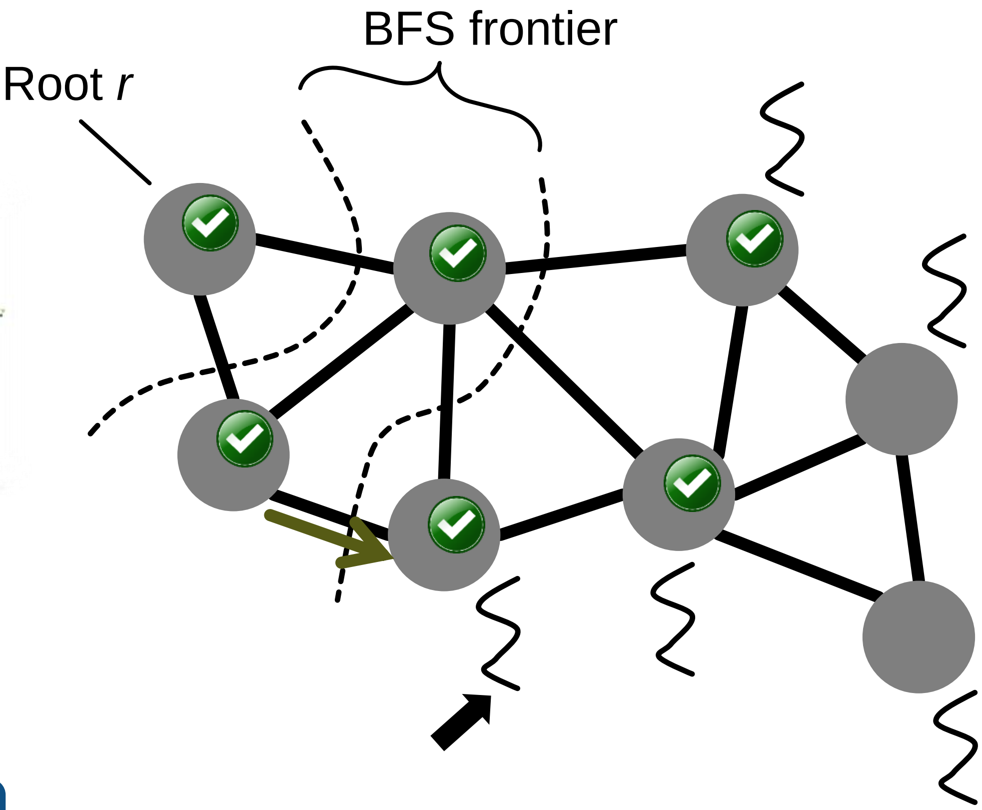
BFS
TOP-DOWN VS. BOTTOM-UP [1]

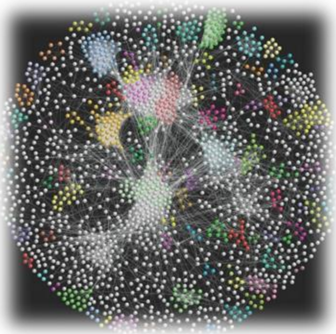


Pushing or pulling when expanding a frontier









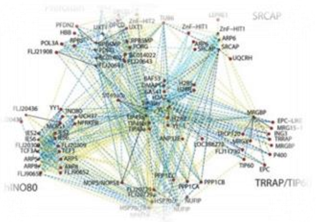
Pulling

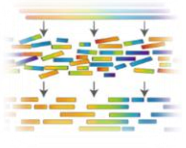
---

[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

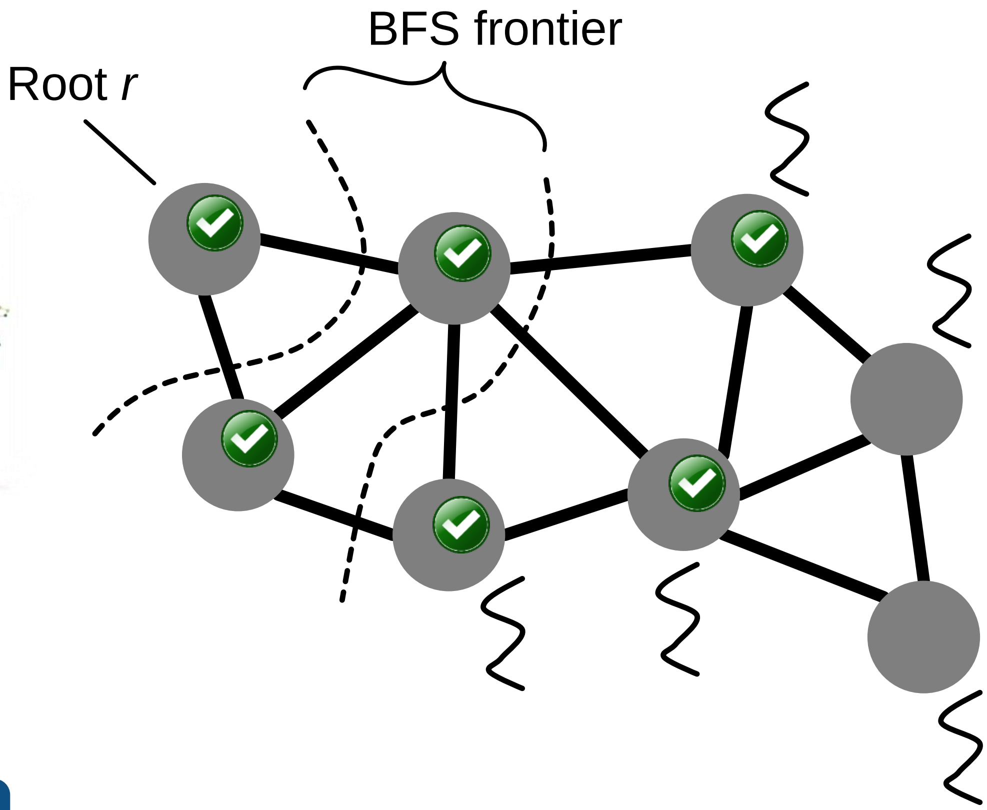
BFS

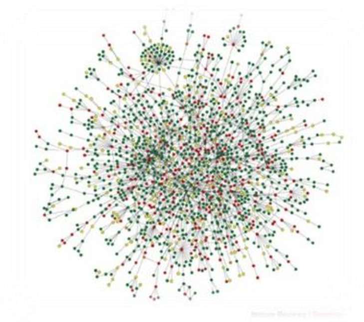
TOP-DOWN VS. BOTTOM-UP [1]

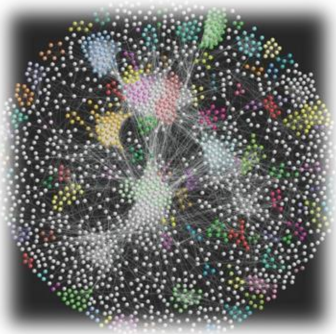




Pushing or pulling when expanding a frontier







Pulling

[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. SC12.

PUSHING VS. PULLING

RESEARCH QUESTIONS

PUSHING VS. PULLING
RESEARCH QUESTIONS

Can we apply the push-pull dichotomy to other graph algorithms?

PUSHING VS. PULLING

RESEARCH QUESTIONS

Can we apply the push-pull dichotomy to other graph algorithms?

What are push-pull formulations of other algorithms?

PUSHING VS. PULLING

RESEARCH QUESTIONS

Can we apply the push-pull dichotomy to other graph algorithms?

What are push-pull formulations of other algorithms?

What pushing vs. pulling *really* is?

PUSHING VS. PULLING
RESEARCH QUESTIONS

What are push-pull formulations of other algorithms?

Can we apply the push-pull dichotomy to other graph algorithms?

What pushing vs. pulling *really* is?

How do they differ in complexity?

PUSHING VS. PULLING

RESEARCH QUESTIONS

What are push-pull formulations of other algorithms?

Can we apply the push-pull dichotomy to other graph algorithms?

What pushing vs. pulling *really* is?

How do they differ in complexity?

What is performance?

PUSHING VS. PULLING
RESEARCH QUESTIONS

What are push-pull formulations of other algorithms?

Can we apply the push-pull dichotomy to other graph algorithms?

What pushing vs. pulling *really* is?

How do they differ in complexity?

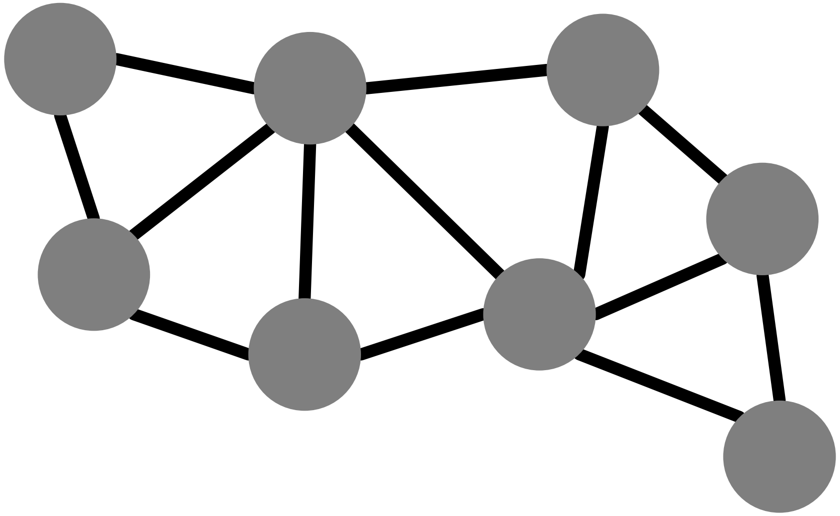
What is performance?

TRIANGLE COUNTING

Vertex importance (#triangles)

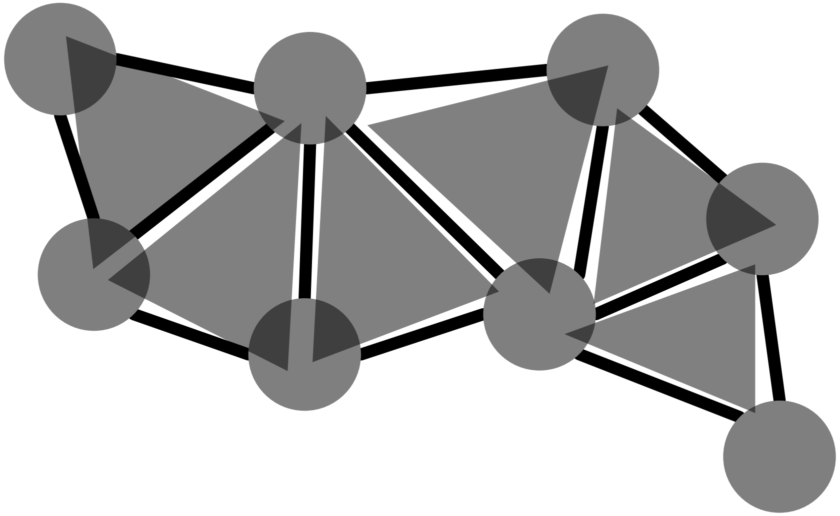
TRIANGLE COUNTING

Vertex importance (#triangles)



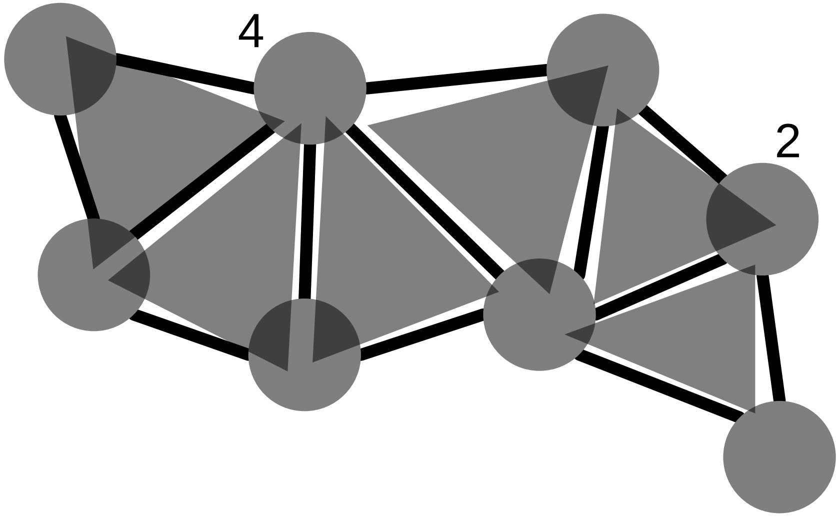
TRIANGLE COUNTING

Vertex importance (#triangles)



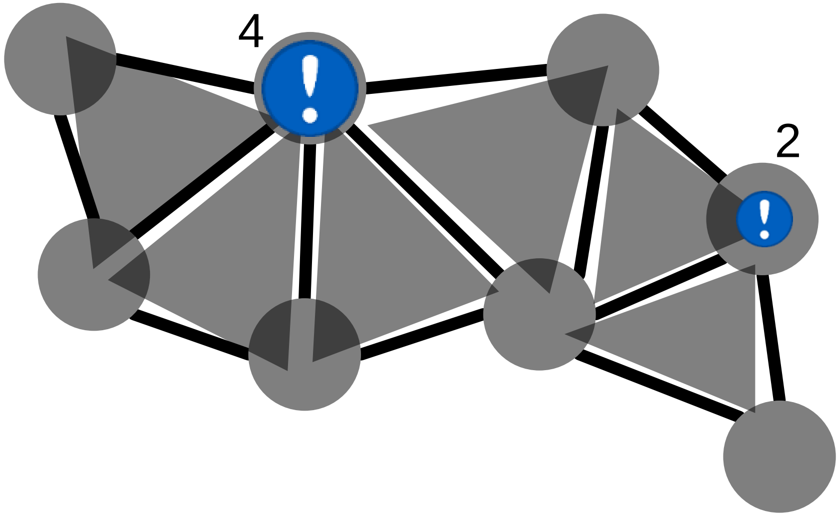
TRIANGLE COUNTING

Vertex importance
(#triangles)



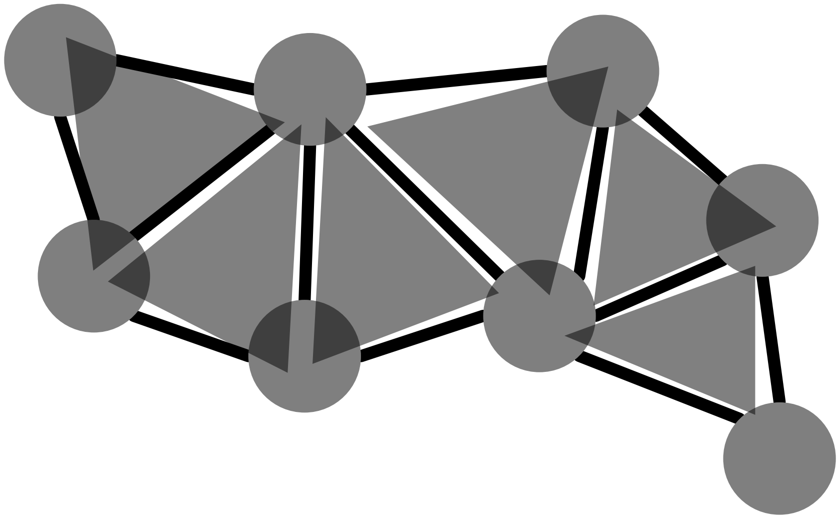
TRIANGLE COUNTING

Vertex importance (#triangles)



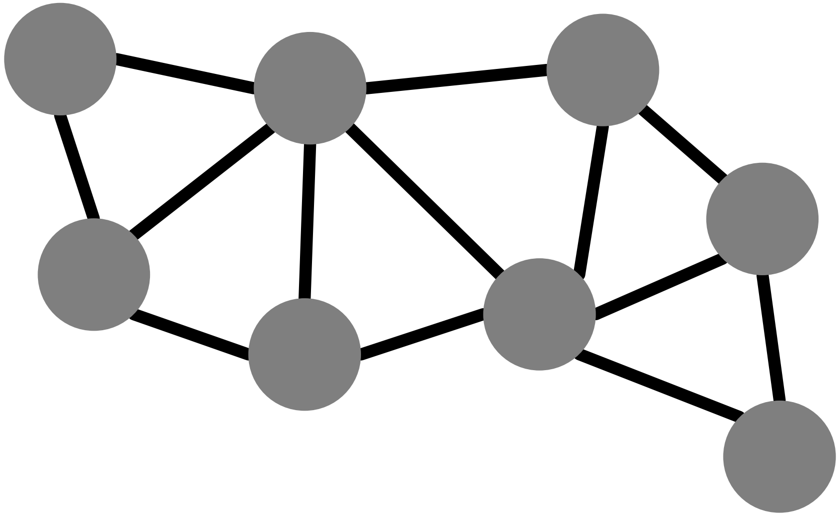
TRIANGLE COUNTING

Vertex importance
(#triangles)



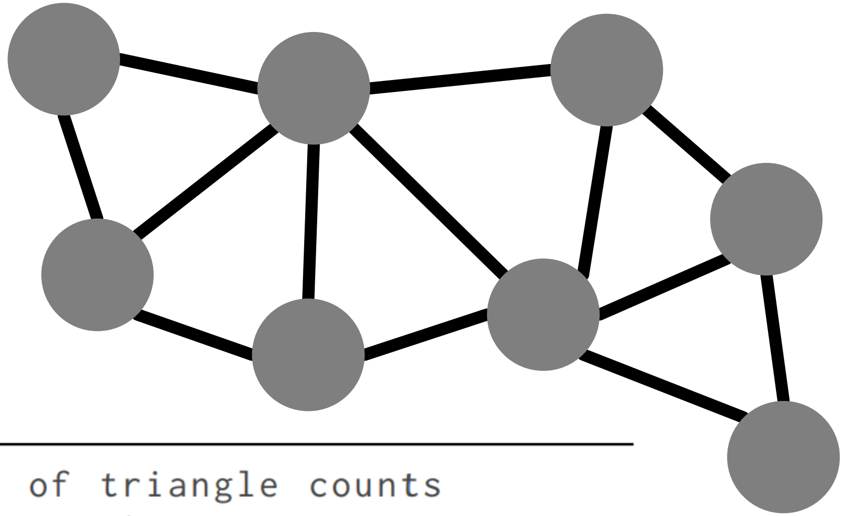
TRIANGLE COUNTING

Vertex importance (#triangles)



TRIANGLE COUNTING

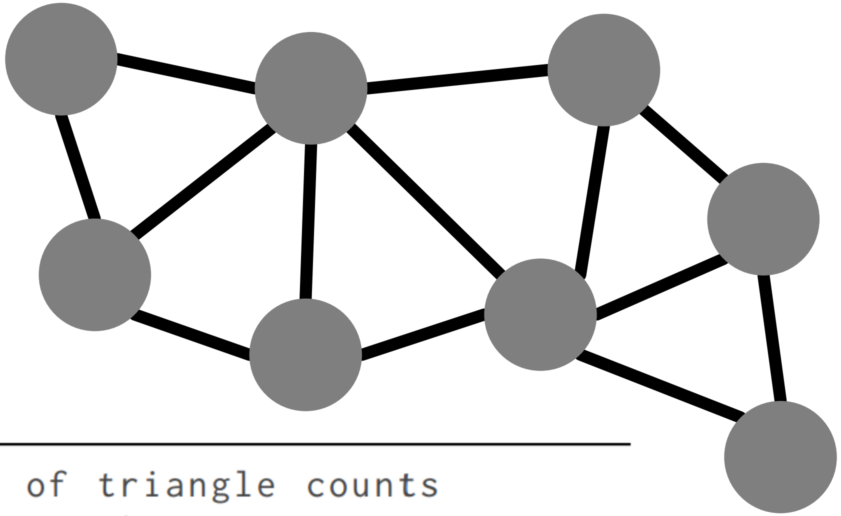
Vertex importance (#triangles)



```
/* Input: a graph G. Output: An array of triangle counts
 * tc[1..n] that each vertex belongs to. */

function TC(G) {




 }
```

TRIANGLE COUNTING

Vertex importance (#triangles)



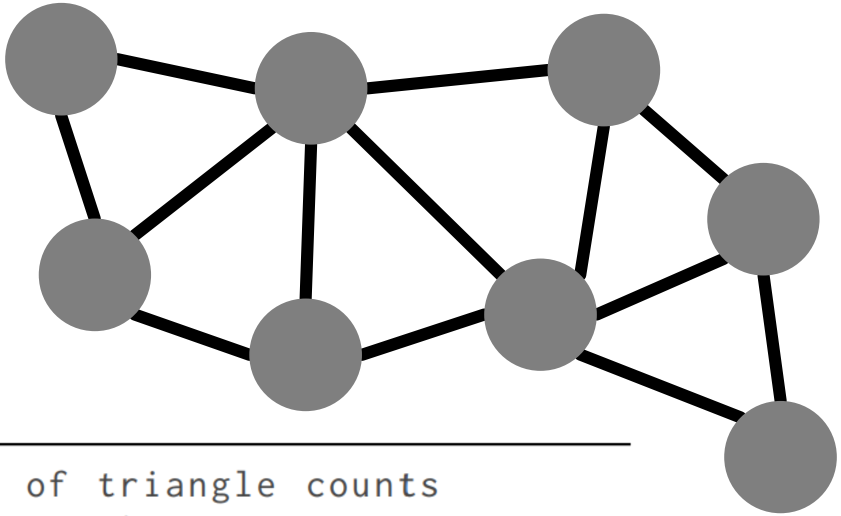
#vertices

```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {




 }
```

TRIANGLE COUNTING

Vertex importance
(#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer

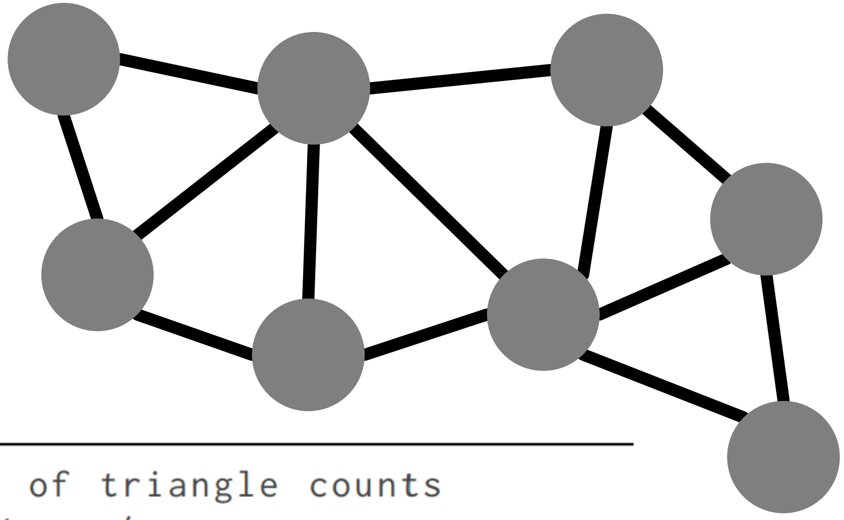


#vertices

```
/* Input: a graph G. Output: An array of triangle counts
 * tc[1..n] that each vertex belongs to. */

function TC(G) {




}





```

TRIANGLE COUNTING

Vertex importance
(#triangles)



Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer

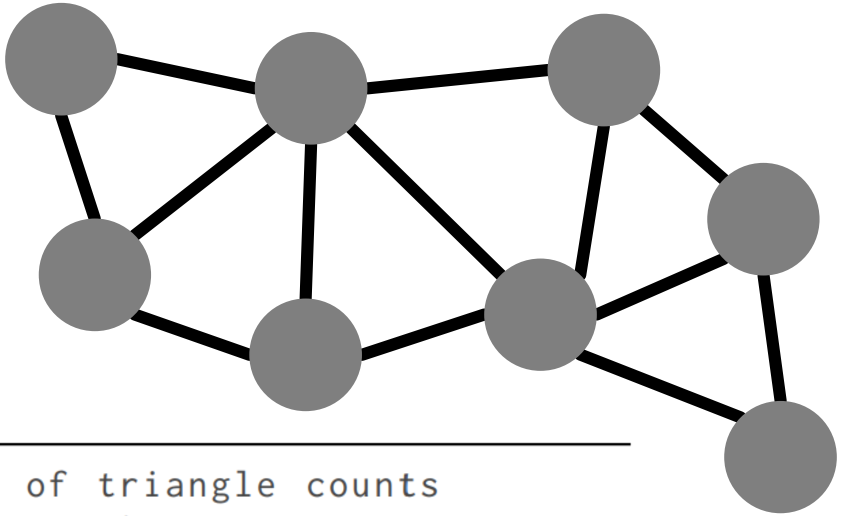
#vertices

```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]




 }
```

TRIANGLE COUNTING

Vertex importance
(#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



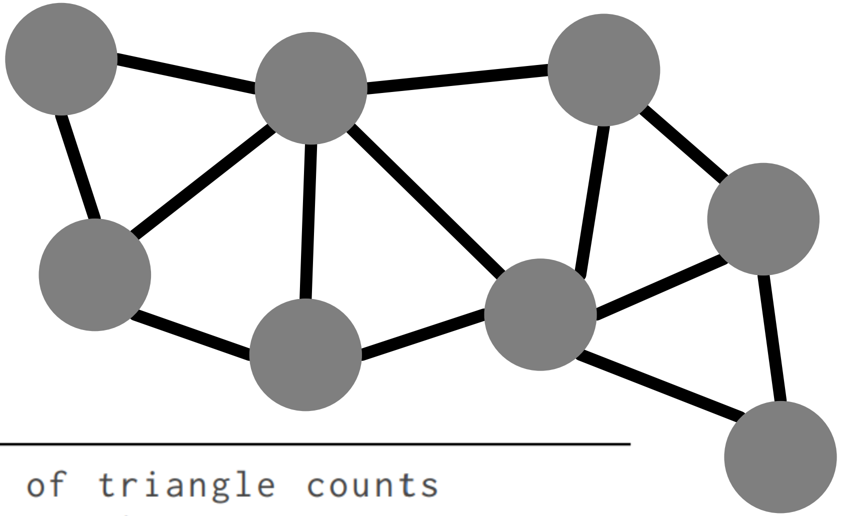
#vertices

```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par



 }





```

TRIANGLE COUNTING

Vertex importance
(#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



#vertices

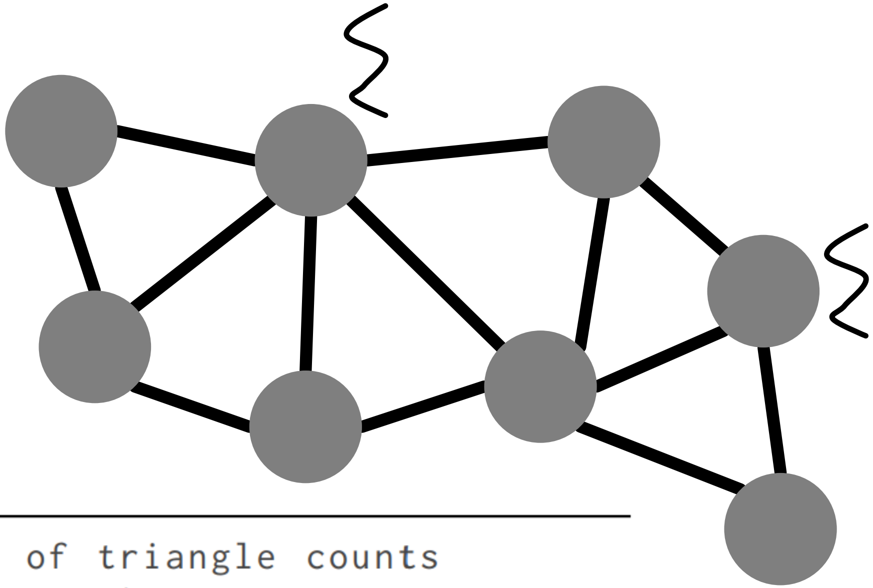
Set of vertices

```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par



}
```

TRIANGLE COUNTING

Vertex importance (#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



#vertices

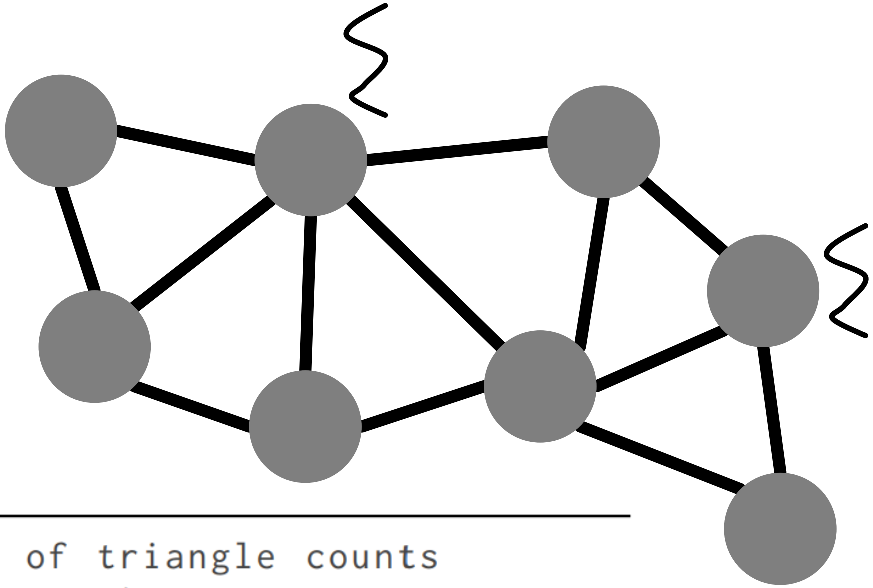
Set of vertices

```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par



}
```

TRIANGLE COUNTING

Vertex importance
(#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



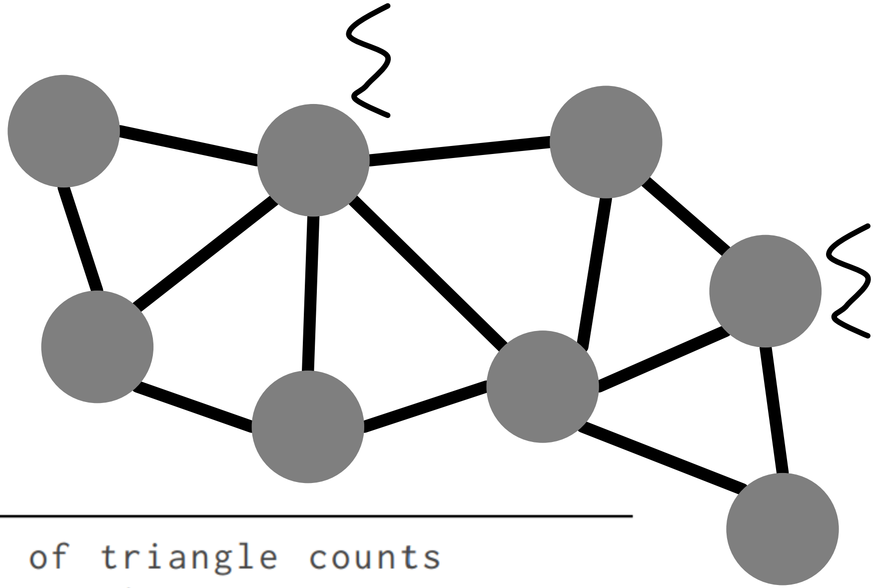
#vertices

```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par
    for $w_1 \in N(v)$ do [in par]
      for $w_2 \in N(v)$ do [in par]

}
```

Set of vertices

TRIANGLE COUNTING

Vertex importance (#triangles)



Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer

#vertices

Set of vertices

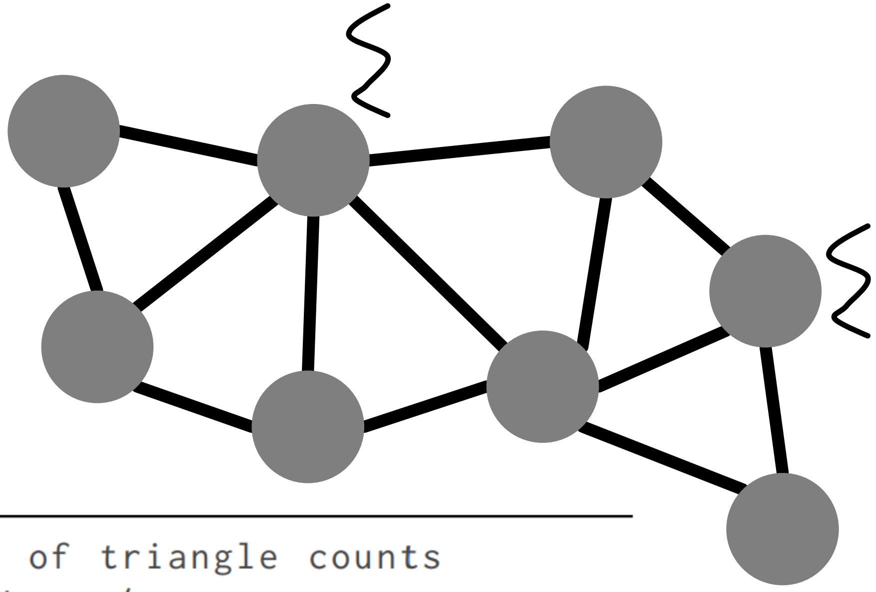
v's neighbors

```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par
    for $w_1 \in N(v)$ do [in par]
      for $w_2 \in N(v)$ do [in par]

}
```

TRIANGLE COUNTING

Vertex importance (#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



#vertices

Set of vertices

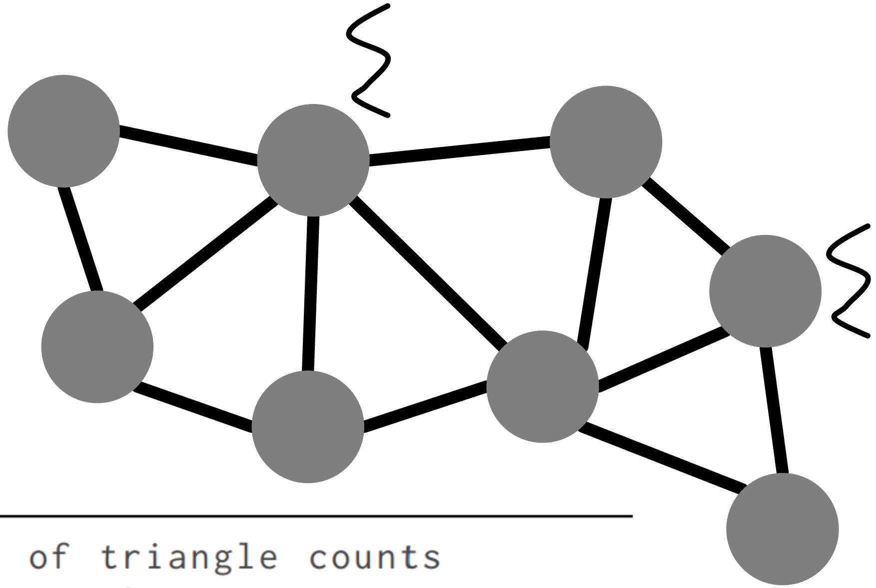
v's neighbors

```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par
    for $w_1 \in N(v)$ do [in par]
      for $w_2 \in N(v)$ do [in par]

}
```

TRIANGLE COUNTING

Vertex importance
(#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



#vertices

```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par
    for $w_1 \in N(v)$ do [in par]
      for $w_2 \in N(v)$ do [in par]
        if adj($w_1$,$w_2$) Ⓡ update_tc();
}
```

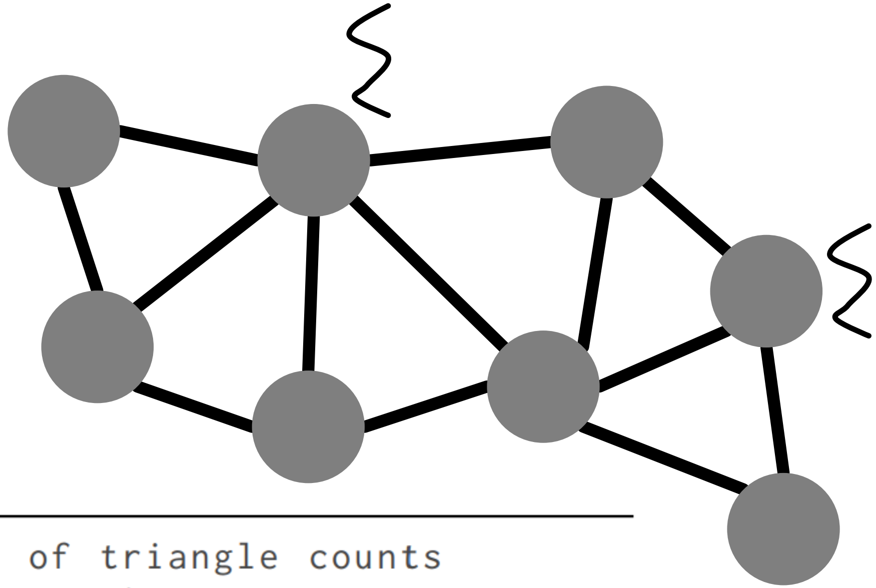
Set of vertices

v's neighbors

TRIANGLE COUNTING

Vertex importance
(#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



#vertices

Set of vertices

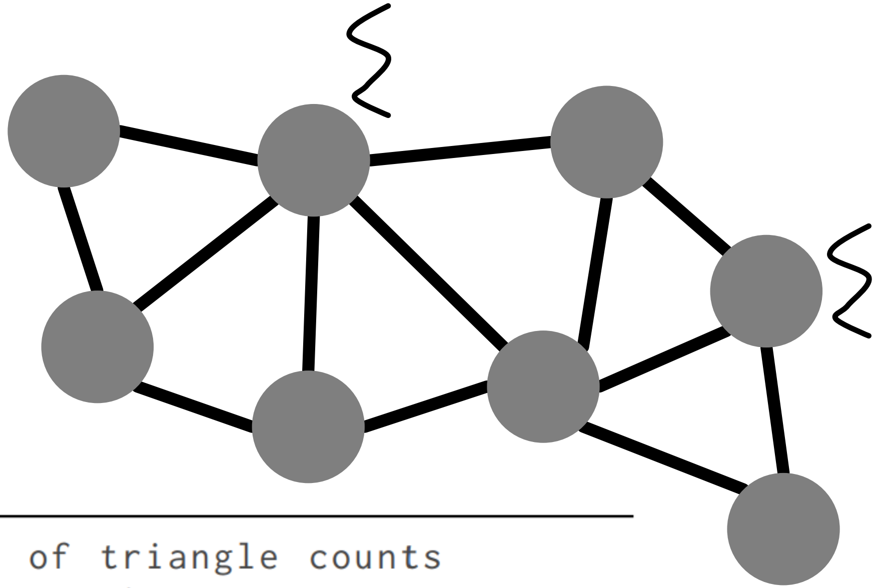
v's neighbors

```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par
    for $w_1 \in N(v)$ do [in par]
      for $w_2 \in N(v)$ do [in par]
        if adj($w_1$,$w_2$) Ⓡ update_tc();
}
function update_tc() {



}
```

TRIANGLE COUNTING

Vertex importance (#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



#vertices

Set of vertices

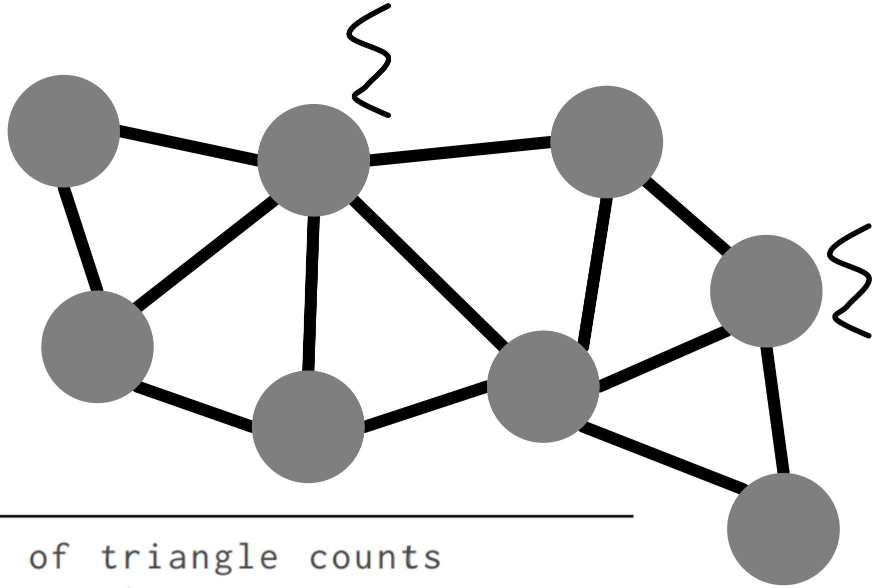
v's neighbors

```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par
    for $w_1 \in N(v)$ do [in par]
      for $w_2 \in N(v)$ do [in par]
        if adj($w_1$,$w_2$) Ⓡ update_tc();
 }
function update_tc() {
  {++tc[$w_1$]; /* or ++tc[$w_2$]. */} Ⓦ i          PUSHING

  {++tc[$v$];}                                      PULLING
}
```

TRIANGLE COUNTING

Vertex importance (#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



#vertices

Set of vertices

v's neighbors

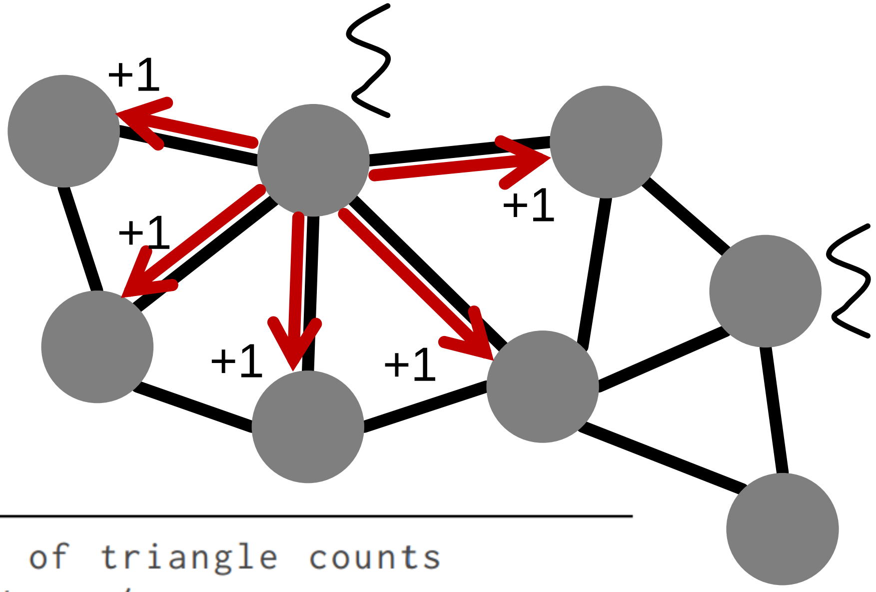
```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par
    for $w_1 \in N(v)$ do [in par]
      for $w_2 \in N(v)$ do [in par]
        if adj($w_1$,$w_2$) Ⓡ update_tc();
}
function update_tc() {
  {++tc[$w_1$]; /* or ++tc[$w_2$]. */} Ⓦ i          PUSHING

  {++tc[$v$];}                                    PULLING
}
```

Pushing

TRIANGLE COUNTING

Vertex importance (#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



#vertices

Set of vertices

v's neighbors

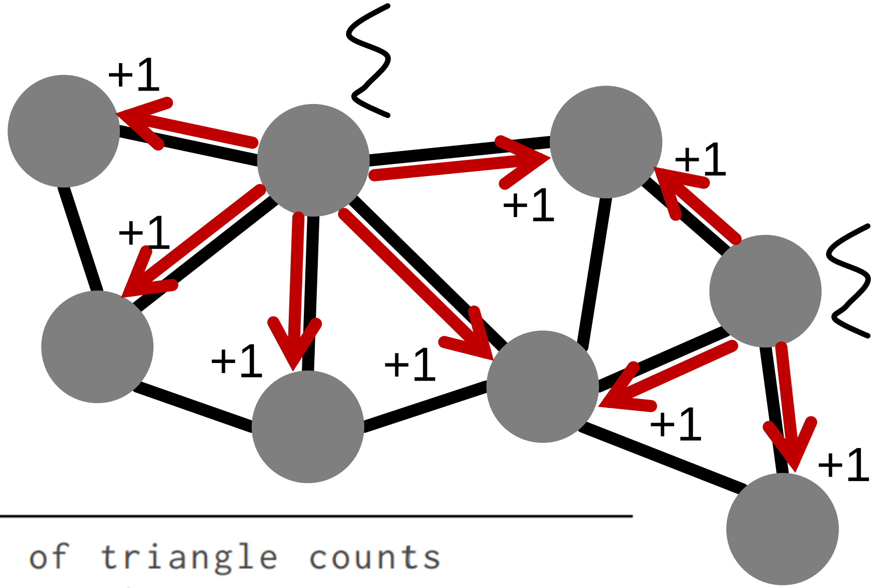
```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par
    for $w_1 \in N(v)$ do [in par]
      for $w_2 \in N(v)$ do [in par]
        if adj($w_1$,$w_2$) Ⓡ update_tc();
 }
function update_tc() {
  {++tc[$w_1$]; /* or ++tc[$w_2$]. */} Ⓦ i          PUSHING

  {++tc[$v$];}                                        PULLING
}
```

Pushing

TRIANGLE COUNTING

Vertex importance (#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



#vertices

Set of vertices

v's neighbors

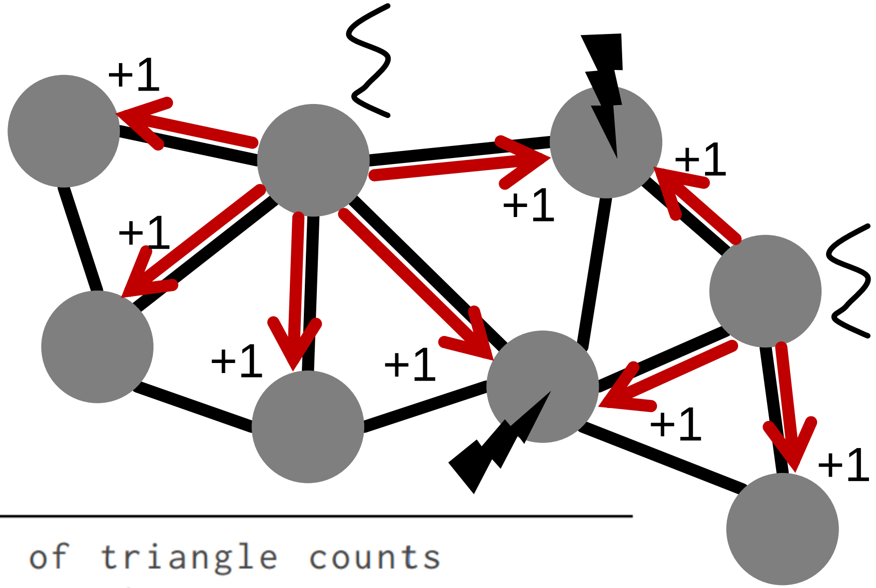
```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par
    for $w_1 \in N(v)$ do [in par]
      for $w_2 \in N(v)$ do [in par]
        if adj($w_1$,$w_2$) Ⓡ update_tc();
}
function update_tc() {
  {++tc[$w_1$]; /* or ++tc[$w_2$]. */} Ⓦ i          PUSHING

  {++tc[$v$];}                                      PULLING
}
```

Pushing

TRIANGLE COUNTING

Vertex importance
(#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



#vertices

Set of vertices

v's neighbors

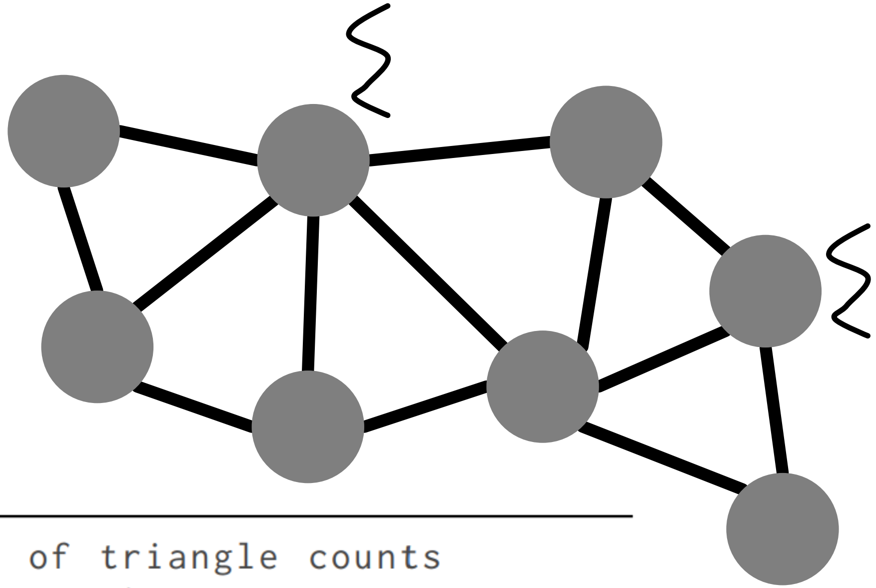
```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par
    for $w_1 \in N(v)$ do [in par]
      for $w_2 \in N(v)$ do [in par]
        if adj($w_1$,$w_2$) Ⓡ update_tc();
}
function update_tc() {
  {++tc[$w_1$]; /* or ++tc[$w_2$]. */} Ⓦ i                PUSHING

  {++tc[$v$];}                                            PULLING
}
```

Pushing

TRIANGLE COUNTING

Vertex importance (#triangles)

W : a write conflict
R : a read conflict
i : integer



#vertices

Set of vertices

v's neighbors

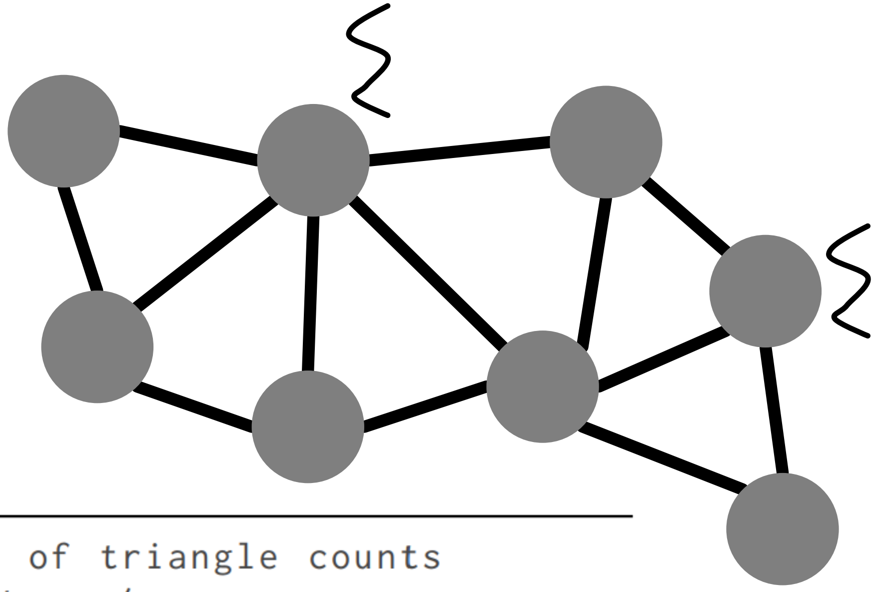
```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par
    for $w_1 \in N(v)$ do [in par]
      for $w_2 \in N(v)$ do [in par]
        if adj($w_1$,$w_2$) R update_tc();
}
function update_tc() {
  {++tc[$w_1$]; /* or ++tc[$w_2$]. */} W i          PUSHING

  {++tc[$v$];}                                      PULLING
}
```

Pushing

TRIANGLE COUNTING

Vertex importance (#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



#vertices

Set of vertices

v's neighbors

```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par
    for $w_1 \in N(v)$ do [in par]
      for $w_2 \in N(v)$ do [in par]
        if adj($w_1$,$w_2$) Ⓡ update_tc();
 }
function update_tc() {
  {++tc[$w_1$]; /* or ++tc[$w_2$]. */} Ⓦ i                PUSHING

  {++tc[$v$];}                                            PULLING
}
```

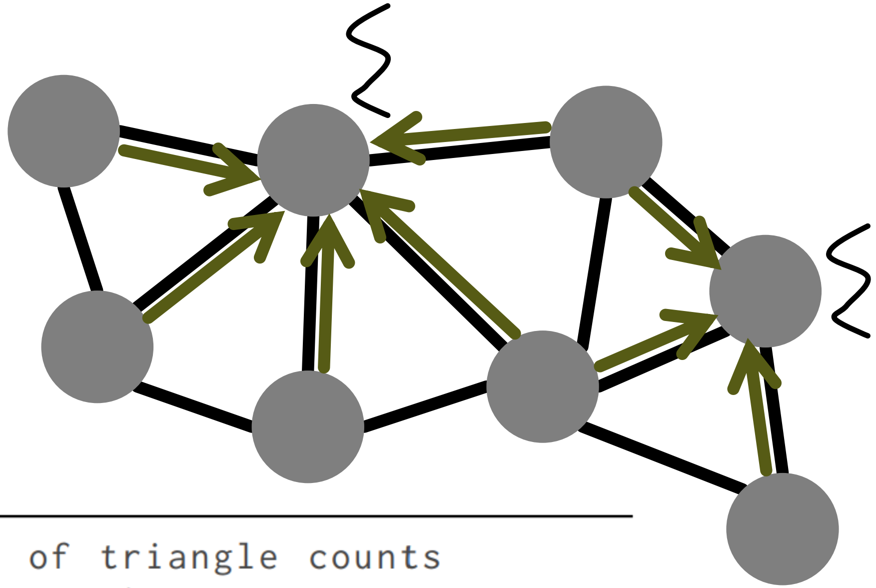
Pushing

Pulling

TRIANGLE COUNTING

Vertex importance (#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



#vertices

Set of vertices

v's neighbors

```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par
    for $w_1 \in N(v)$ do [in par]
      for $w_2 \in N(v)$ do [in par]
        if adj($w_1$,$w_2$) Ⓡ update_tc();
}
function update_tc() {
  {++tc[$w_1$]; /* or ++tc[$w_2$]. */} Ⓦ i        PUSHING

  {++tc[$v$];}                                    PULLING
}
```

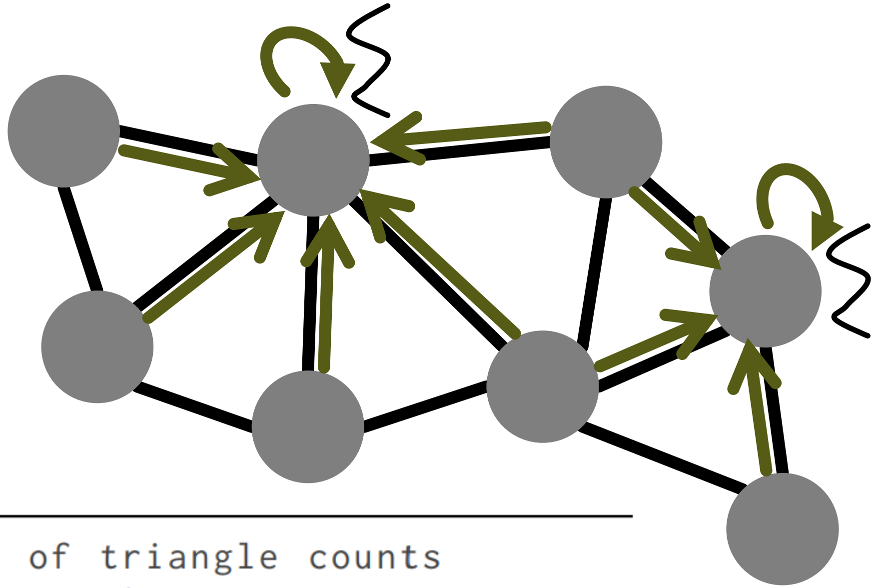
Pushing

Pulling

TRIANGLE COUNTING

Vertex importance (#triangles)

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



#vertices

```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par
    for $w_1 \in N(v)$ do [in par]
      for $w_2 \in N(v)$ do [in par]
        if adj($w_1$,$w_2$) Ⓡ update_tc();
}
function update_tc() {
  {++tc[$w_1$]; /* or ++tc[$w_2$]. */} Ⓦ i          PUSHING

  {++tc[$v$];}                                       PULLING
}
```

Set of vertices

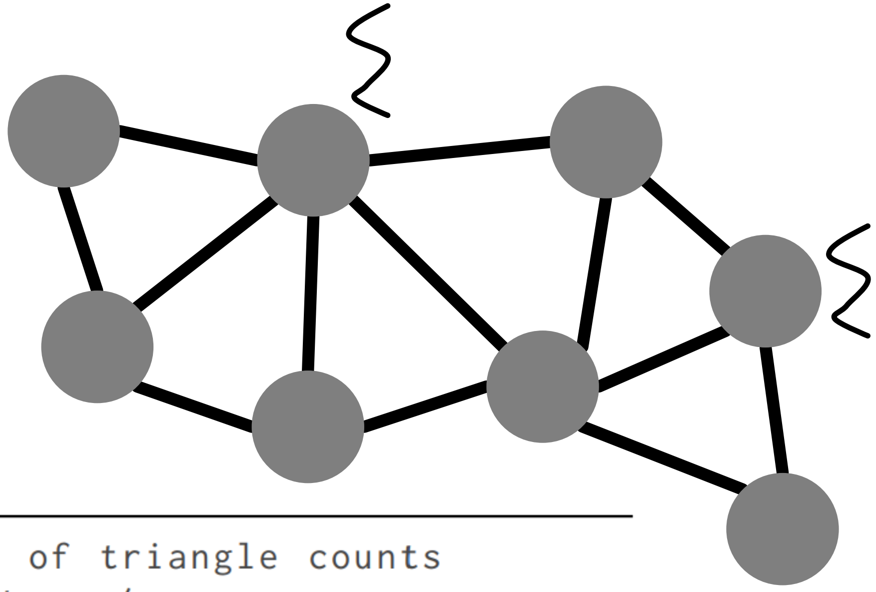
v's neighbors

Pushing

Pulling

TRIANGLE COUNTING

Vertex importance (#triangles)



Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer

#vertices

Set of vertices

v's neighbors

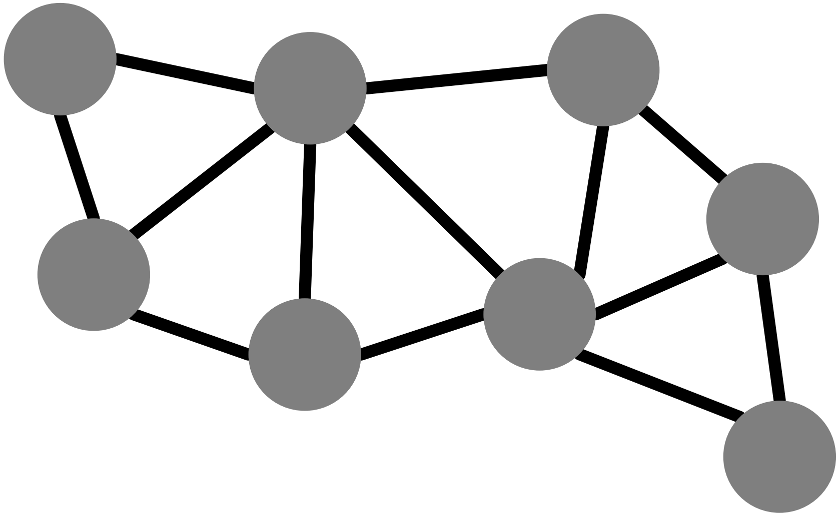
```
/* Input: a graph $G$. Output: An array of triangle counts
 * tc[1..$n$] that each vertex belongs to. */

function TC($G$) {tc[1..$n$] = [0..0]
  for $v \in V$ do in par
    for $w_1 \in N(v)$ do [in par]
      for $w_2 \in N(v)$ do [in par]
        if adj($w_1$,$w_2$) Ⓡ update_tc();
}
function update_tc() {
  {++tc[$w_1$]; /* or ++tc[$w_2$]. */} Ⓦ i          PUSHING

  {++tc[$v$];}                                       PULLING
}
```

Pushing

Pulling

BETWEENNESS CENTRALITY

BRANDES [1]

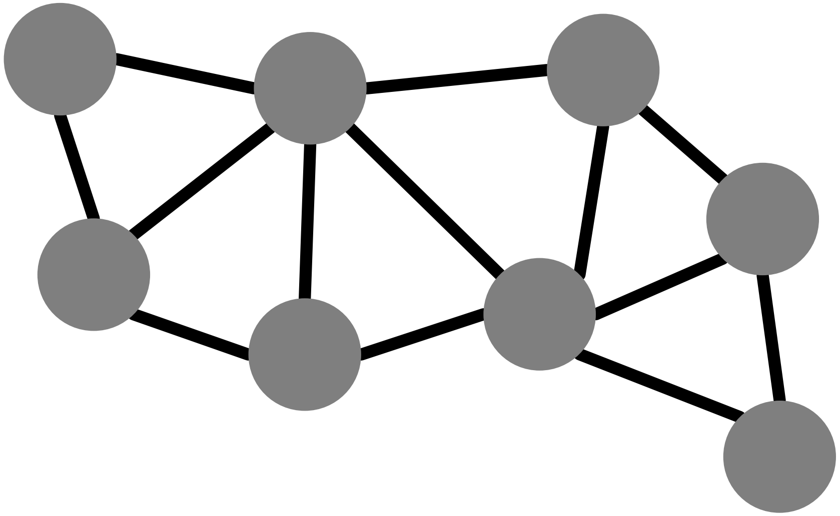


[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

BRANDES [1]

! Vertex importance (#shortest paths)

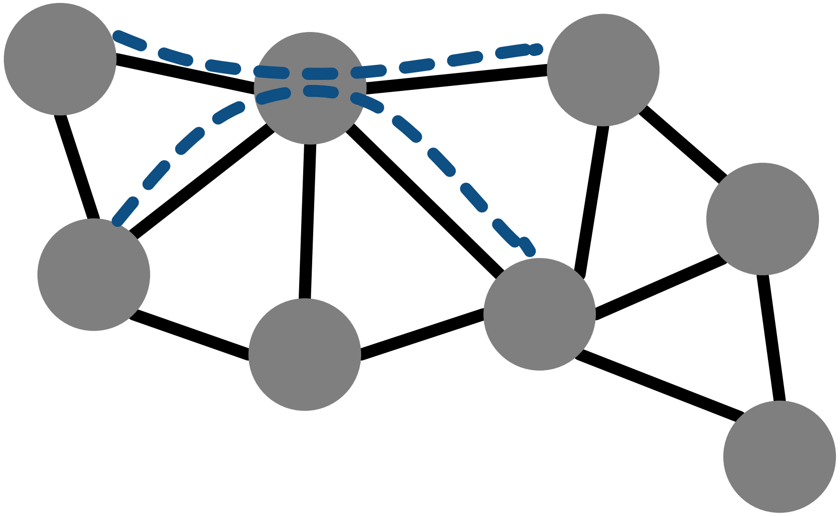


[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

BRANDES [1]

Vertex importance (#shortest paths)

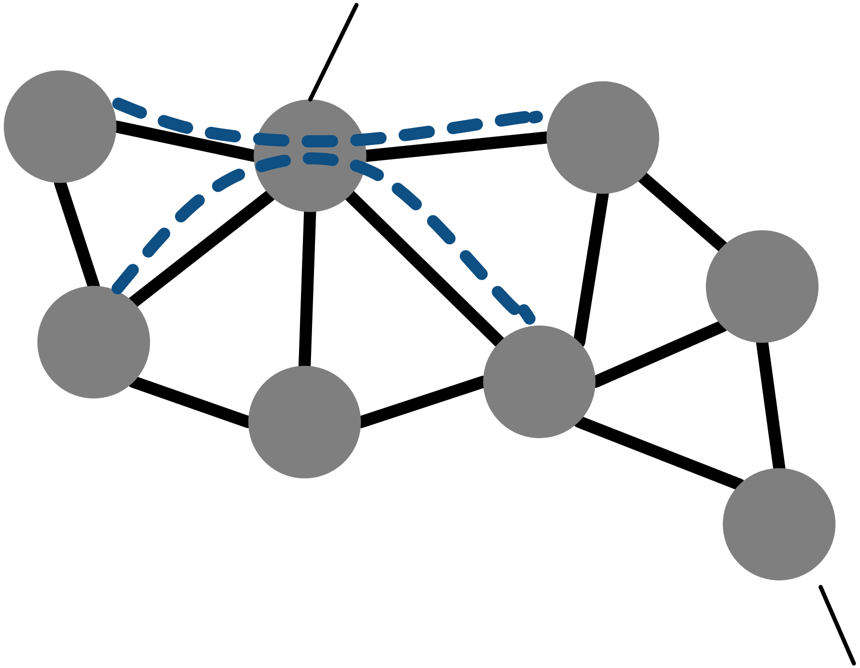


---

[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY
BRANDES [1]

Vertex importance
(#shortest paths)



At least two paths
(this one is relevant)

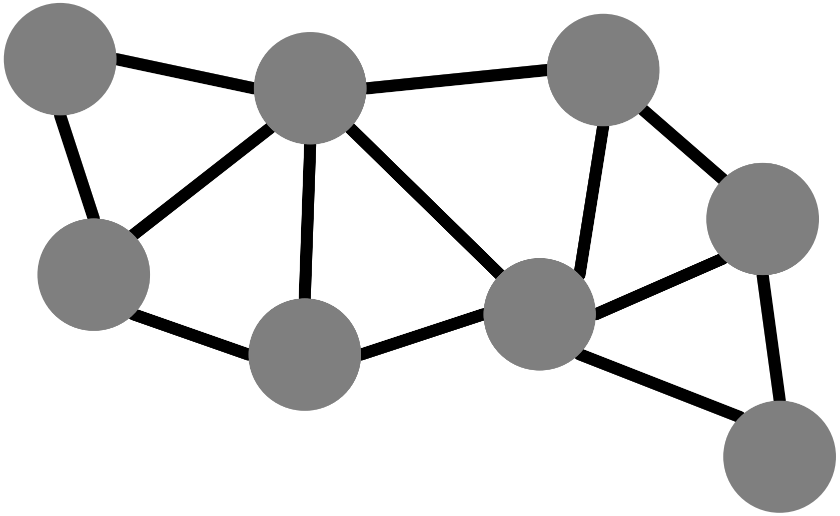
This poor
one has 0

[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

BRANDES [1]

! Vertex importance (#shortest paths)

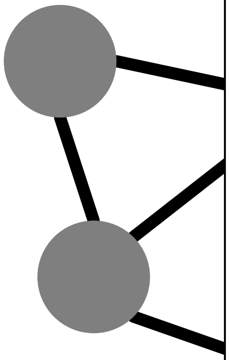


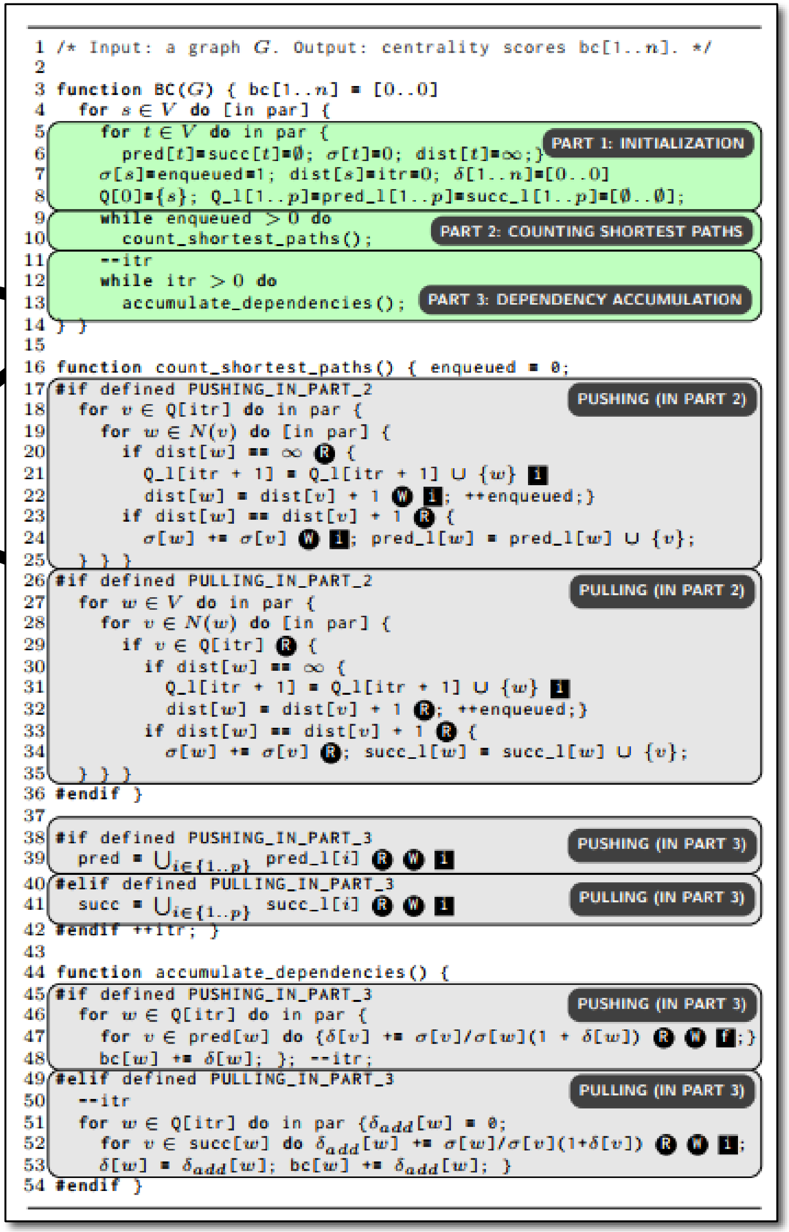
[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

BRANDES [1]

! Vertex importance (#shortest paths)





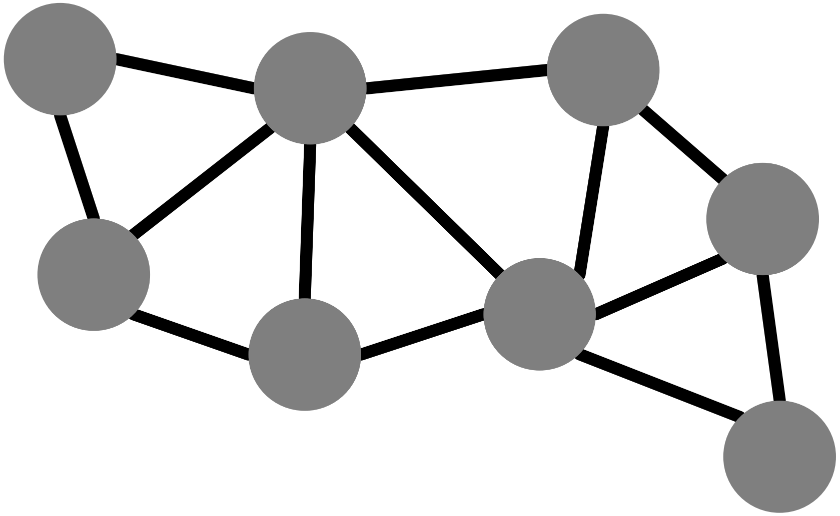
```
/* Input: a graph $G$. Output: centrality scores bc[1..$n$]. */

function BC($G$) { bc[1..$n$] = [0..0]
  for $s \in V$ do [in par] {
    for $t \in V$ do in par {
      pred[$t$]=succ[$t$]=$\emptyset$; $\sigma$[$t$]=0; dist[$t$]=$\infty$;}
    $\sigma$[$s$]=enqueued=1; dist[$s$]=itr=0; $\delta$[1..$n$]=[0..0]
    Q[0]={$s$}; Q_l[1..$p$]=pred_l[1..$p$]=succ_l[1..$p$]=[$\emptyset$..$\emptyset$];
    while enqueued $>$ 0 do
      count_shortest_paths();
    --itr
    while itr $>$ 0 do
      accumulate_dependencies();
} }

function count_shortest_paths() { enqueued = 0;
#if defined PUSHING_IN_PART_2
  for $v \in$ Q[itr] do in par {
    for $w \in N(v)$ do [in par] {
      if dist[$w$] == $\infty$ Ⓡ {
        Q_l[itr + 1] = Q_l[itr + 1] $\cup$ {$w$} 🅸
        dist[$w$] = dist[$v$] + 1 Ⓦ 🅸; ++enqueued;}
      if dist[$w$] == dist[$v$] + 1 Ⓡ {
        $\sigma$[$w$] += $\sigma$[$v$] Ⓦ 🅸; pred_l[$w$] = pred_l[$w$] $\cup$ {$v$};
  } } }
#if defined PULLING_IN_PART_2
  for $w \in V$ do in par {
    for $v \in N(w)$ do [in par] {
      if $v \in$ Q[itr] Ⓡ {
        if dist[$w$] == $\infty$ {
          Q_l[itr + 1] = Q_l[itr + 1] $\cup$ {$w$} 🅸
          dist[$w$] = dist[$v$] + 1 Ⓡ; ++enqueued;}
        if dist[$w$] == dist[$v$] + 1 Ⓡ {
          $\sigma$[$w$] += $\sigma$[$v$] Ⓡ; succ_l[$w$] = succ_l[$w$] $\cup$ {$v$};
  } } }
#endif }

#if defined PUSHING_IN_PART_3
  pred = $\bigcup_{i \in \{1..p\}}$ pred_l[$i$] Ⓡ Ⓦ 🅸
#elif defined PULLING_IN_PART_3
  succ = $\bigcup_{i \in \{1..p\}}$ succ_l[$i$] Ⓡ Ⓦ 🅸
#endif ++itr; }

function accumulate_dependencies() {
#if defined PUSHING_IN_PART_3
  for $w \in$ Q[itr] do in par {
    for $v \in$ pred[$w$] do {$\delta$[$v$] += $\sigma$[$v$]/$\sigma$[$w$](1 + $\delta$[$w$]) Ⓡ Ⓦ 🅵;}
    bc[$w$] += $\delta$[$w$]; }; --itr;
#elif defined PULLING_IN_PART_3
  --itr
  for $w \in$ Q[itr] do in par {$\delta_{add}$[$w$] = 0;
    for $v \in$ succ[$w$] do $\delta_{add}$[$w$] += $\sigma$[$w$]/$\sigma$[$v$](1+$\delta$[$v$]) Ⓡ Ⓦ 🅸;
    $\delta$[$w$] = $\delta_{add}$[$w$]; bc[$w$] += $\delta_{add}$[$w$]; }
#endif }
```

[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

BRANDES [1]

! Vertex importance (#shortest paths)



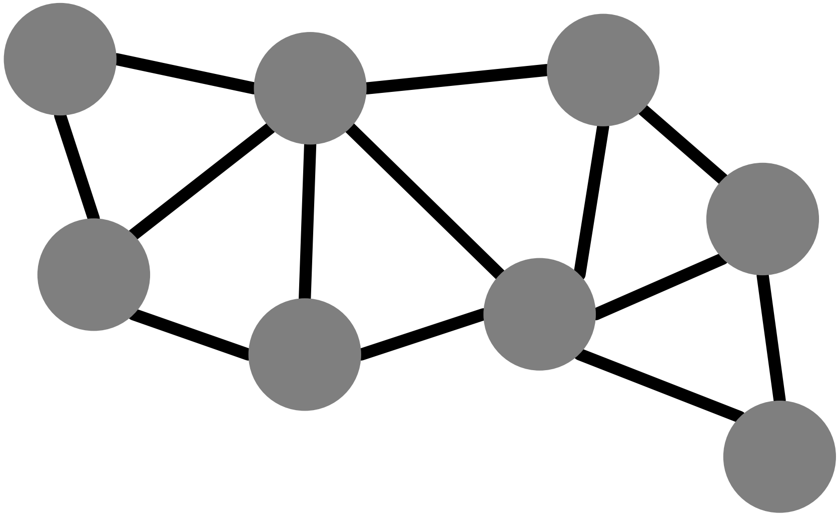
[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

BRANDES [1]

Vertex importance
(#shortest paths)

1. Forward traversals



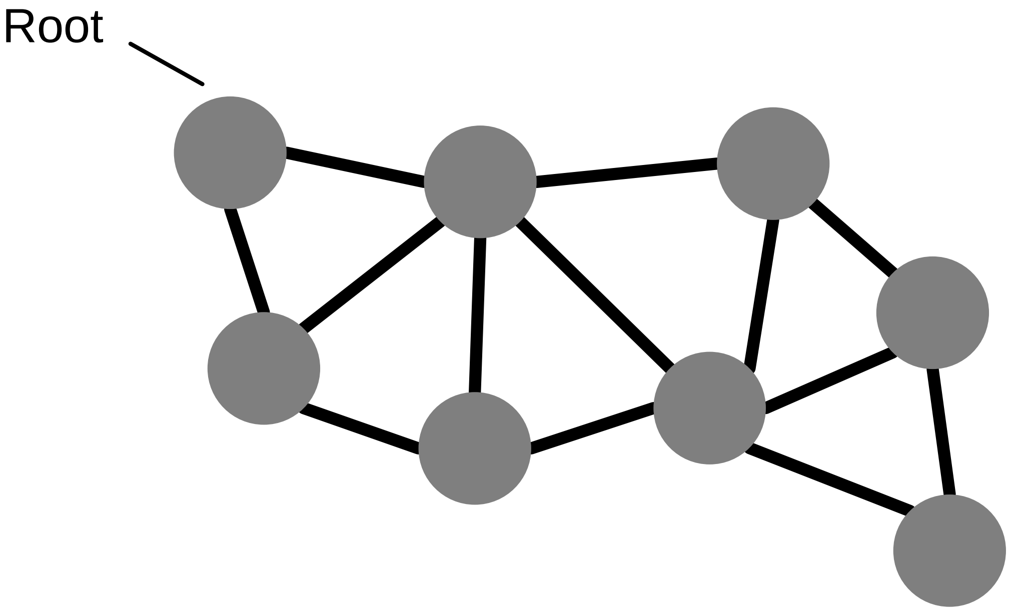
[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

BRANDES [1]

Vertex importance (#shortest paths)

1. Forward traversals



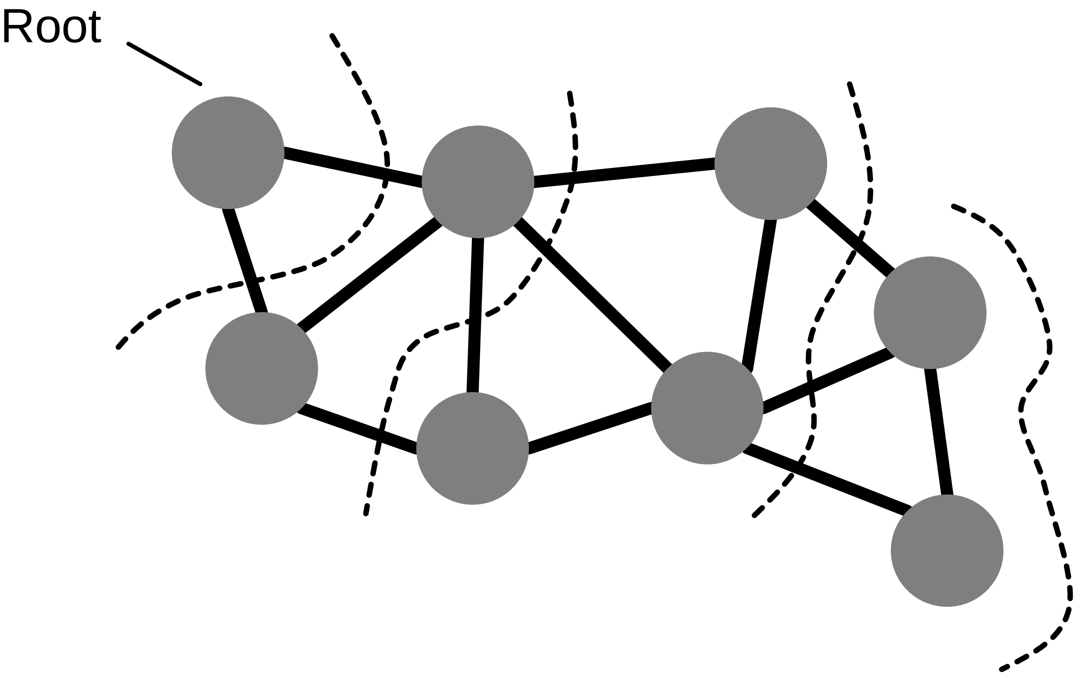
[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

BRANDES [1]

Vertex importance (#shortest paths)

1. Forward traversals



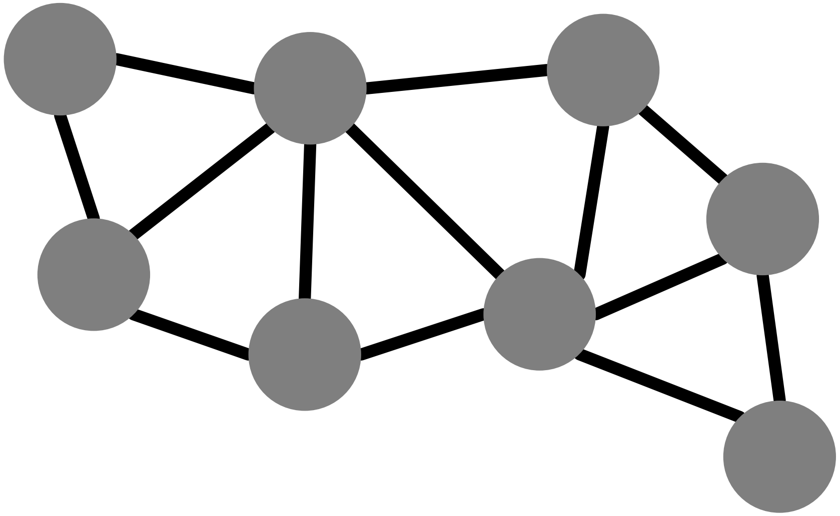
[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

BRANDES [1]

! Vertex importance (#shortest paths)

1. Forward traversals



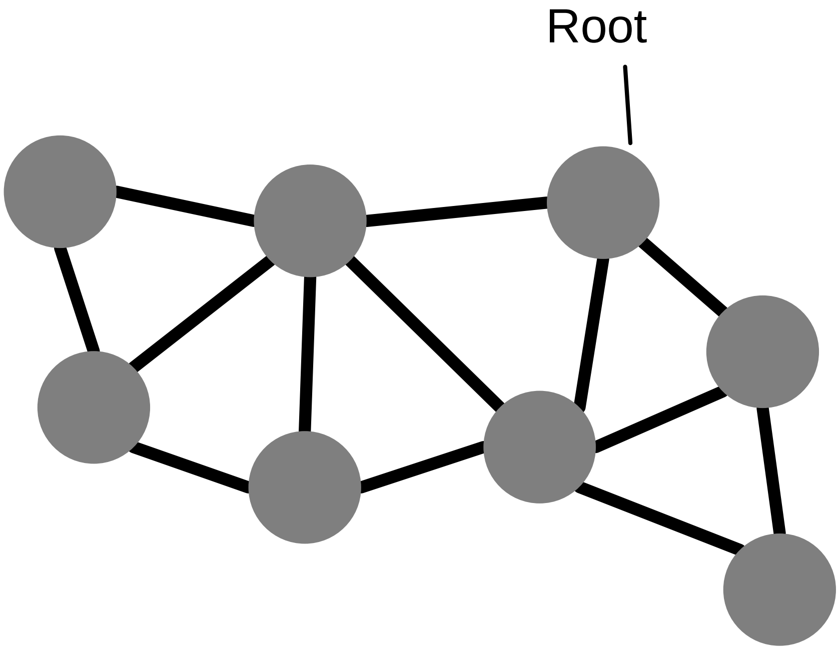
[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

BRANDES [1]

Vertex importance (#shortest paths)

1. Forward traversals



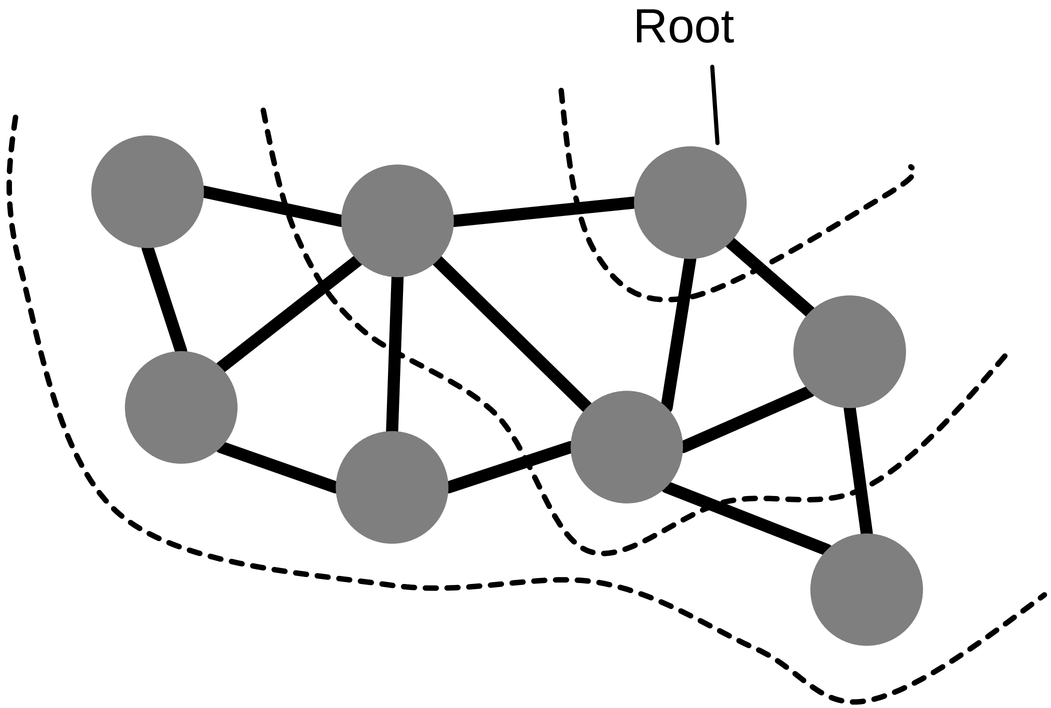
[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

BRANDES [1]

Vertex importance
(#shortest paths)

1. Forward traversals



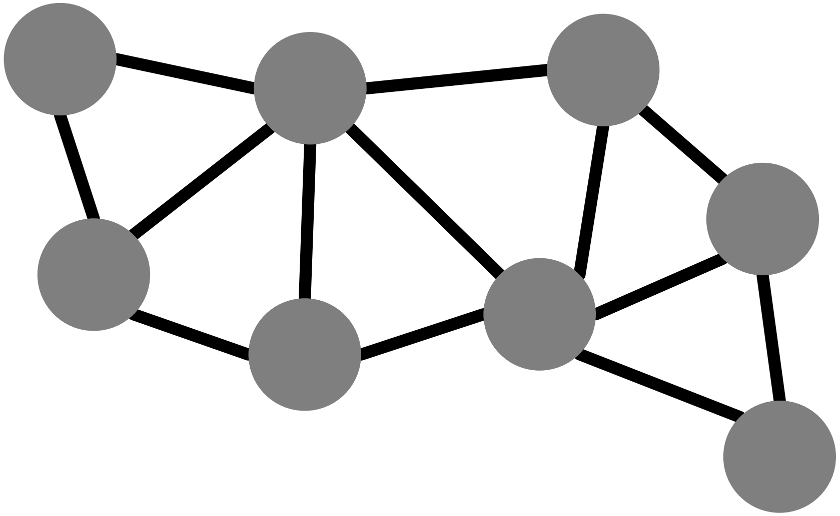
[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

BRANDES [1]

! Vertex importance (#shortest paths)

1. Forward traversals



[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

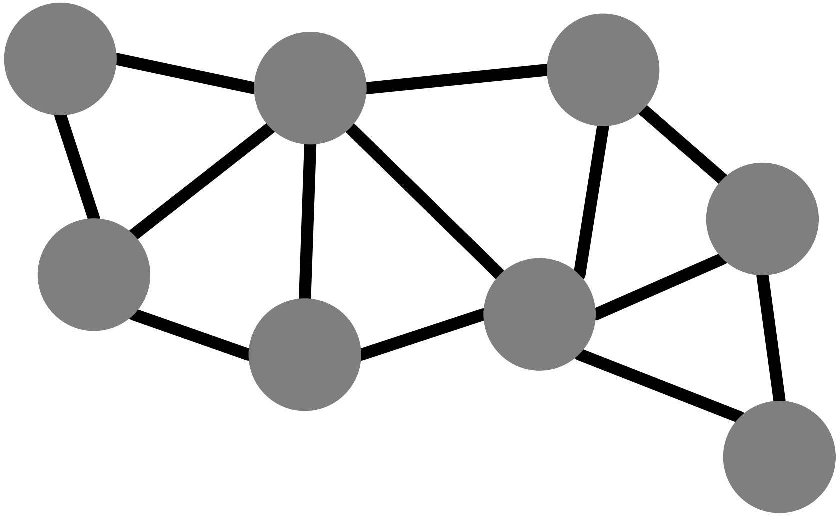
BRANDES [1]

Vertex importance
(#shortest paths)

1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices



[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

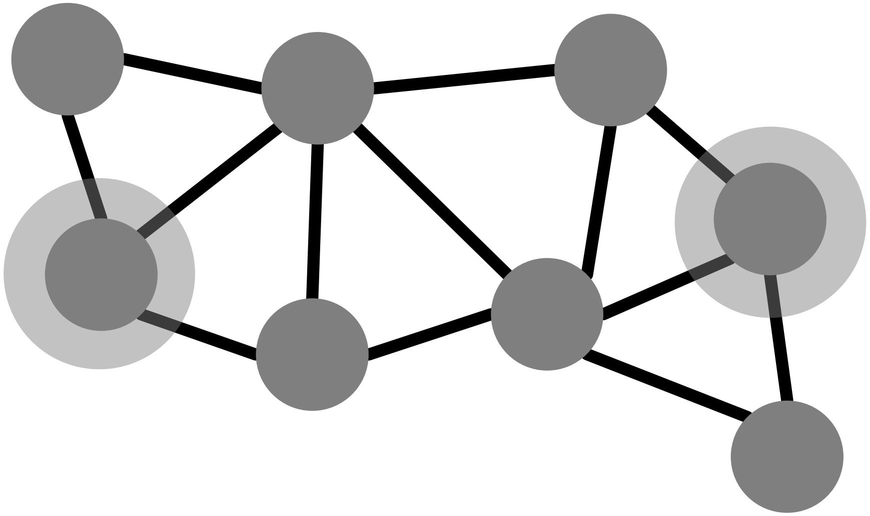
BRANDES [1]

!
Vertex importance (#shortest paths)

1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices



[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

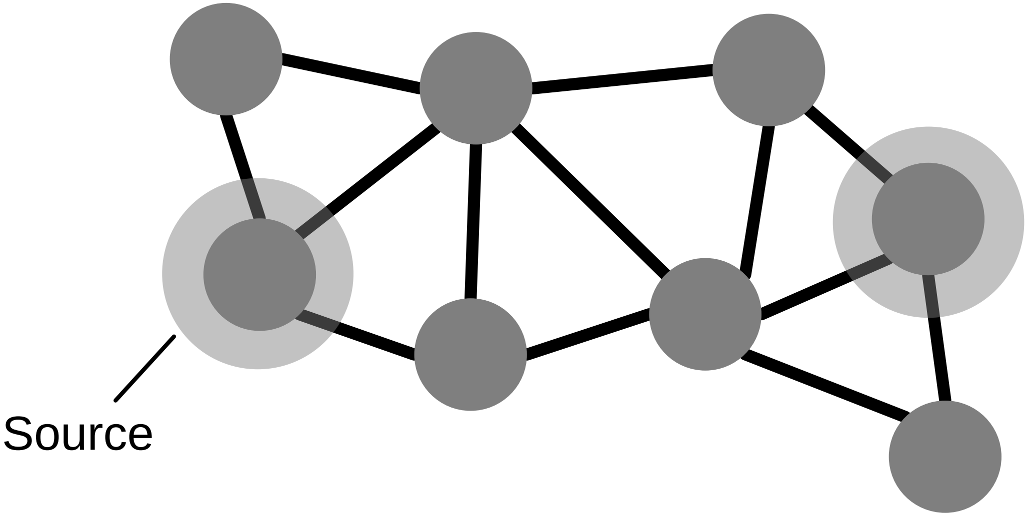
BRANDES [1]

Vertex importance (#shortest paths)

1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices



[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

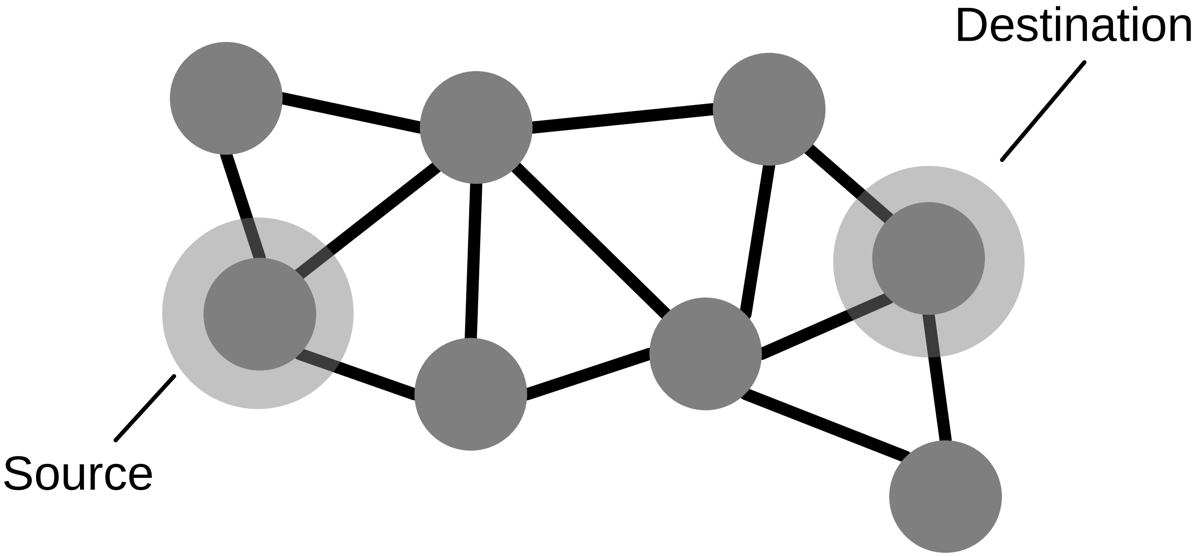
BRANDES [1]

Vertex importance (#shortest paths)

1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices



[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

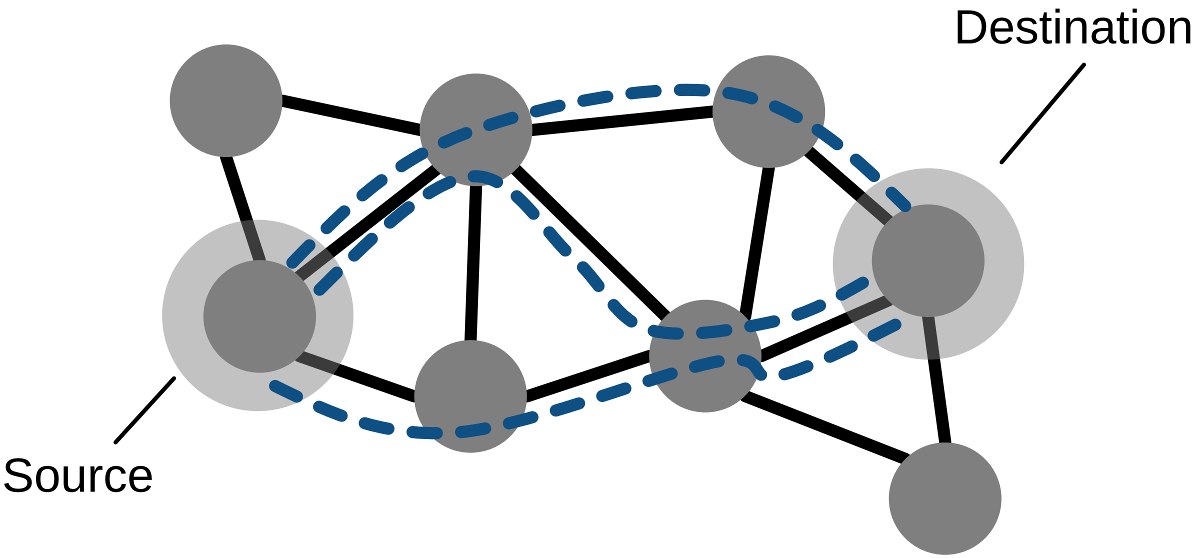
BRANDES [1]

Vertex importance (#shortest paths)

1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices



[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

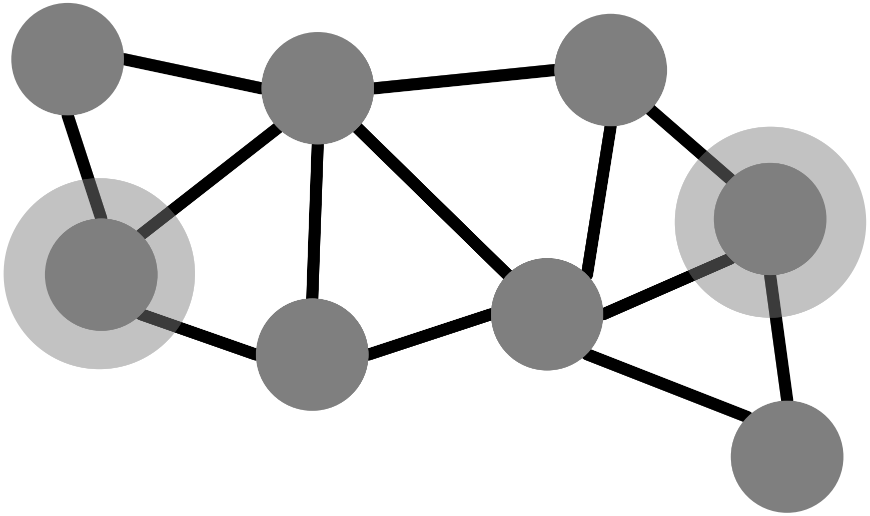
BRANDES [1]

Vertex importance (#shortest paths)

1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices

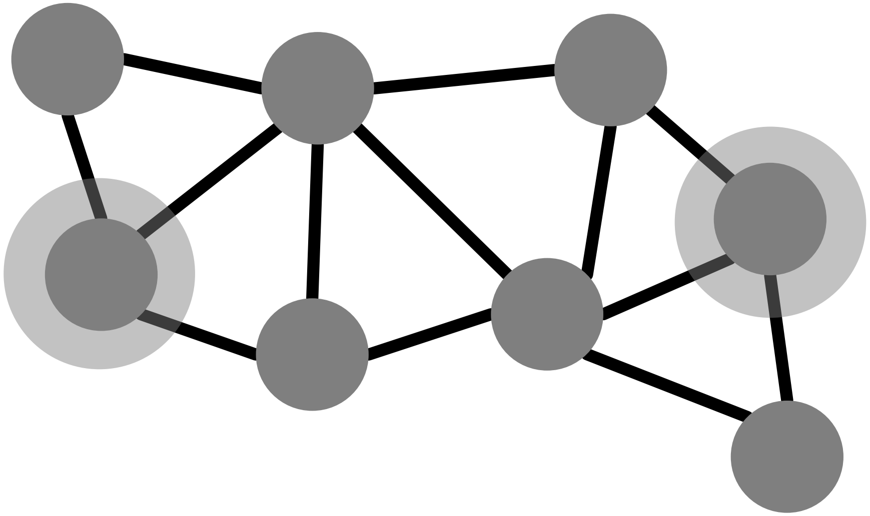


[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

BRANDES [1]

! Vertex importance (#shortest paths)



1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

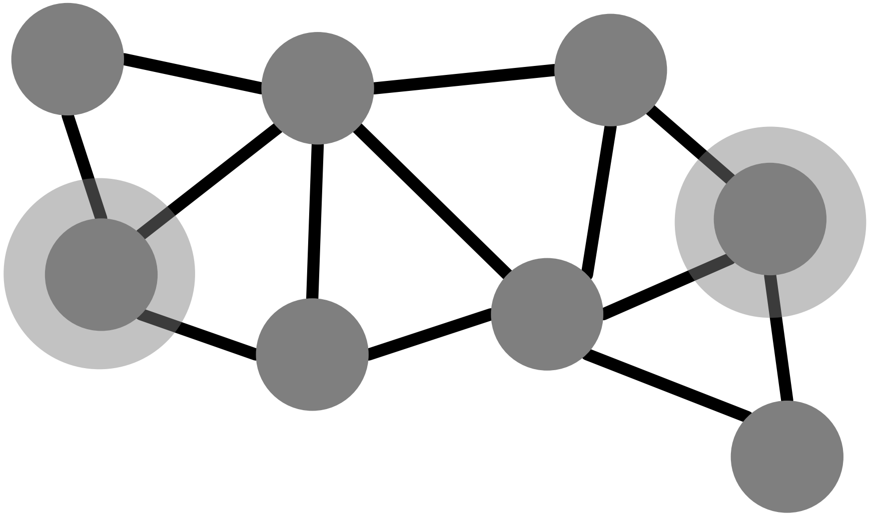
Compute #shortest paths between any two vertices

2. Backward traversals

[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY
BRANDES [1]

Vertex importance (#shortest paths)



1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices

2. Backward traversals

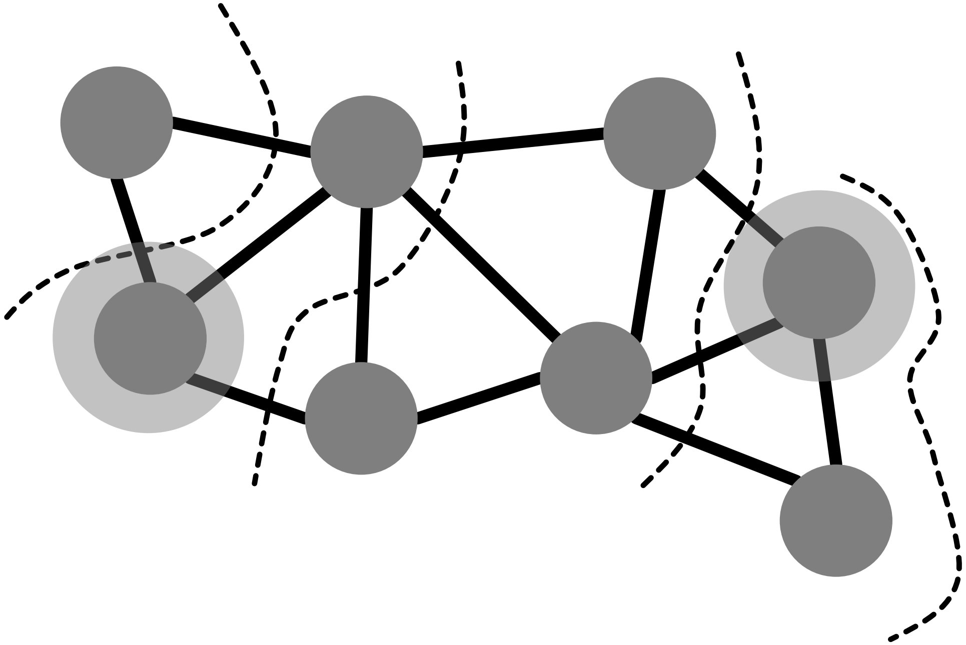
Accumulate centrality scores during backward traversals [1].

[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

BRANDES [1]

! Vertex importance (#shortest paths)



1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices

2. Backward traversals

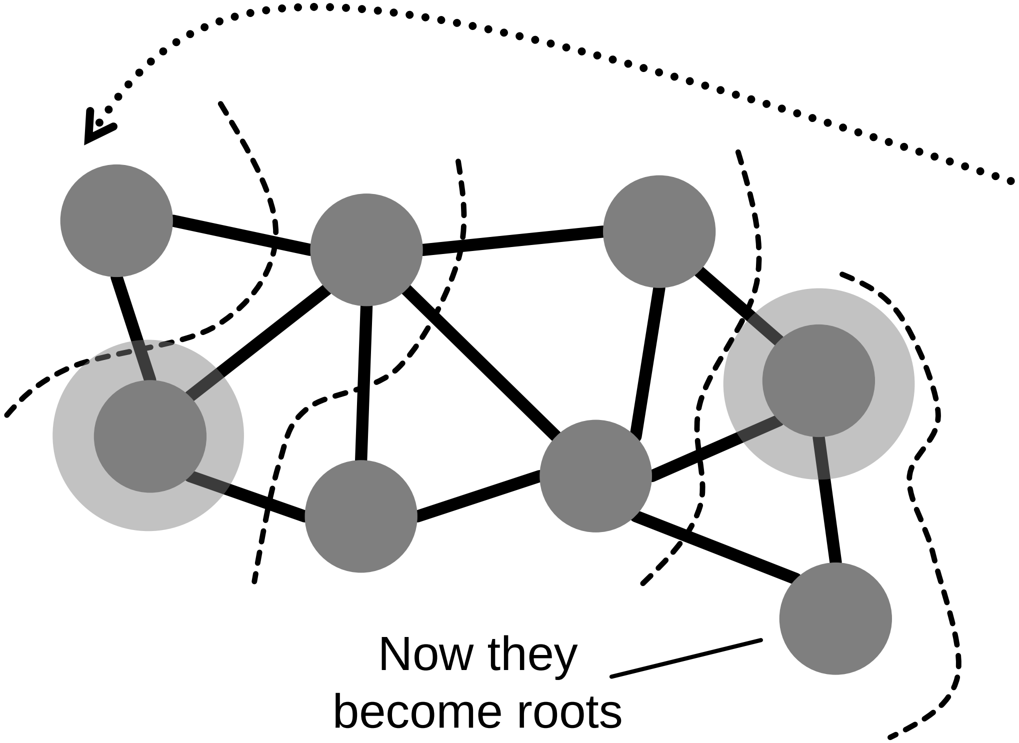
Accumulate centrality scores during backward traversals [1].

[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

BRANDES [1]

! Vertex importance (#shortest paths)



Now they become roots

1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices

2. Backward traversals

Accumulate centrality scores during backward traversals [1].

---

[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

BETWEENNESS CENTRALITY

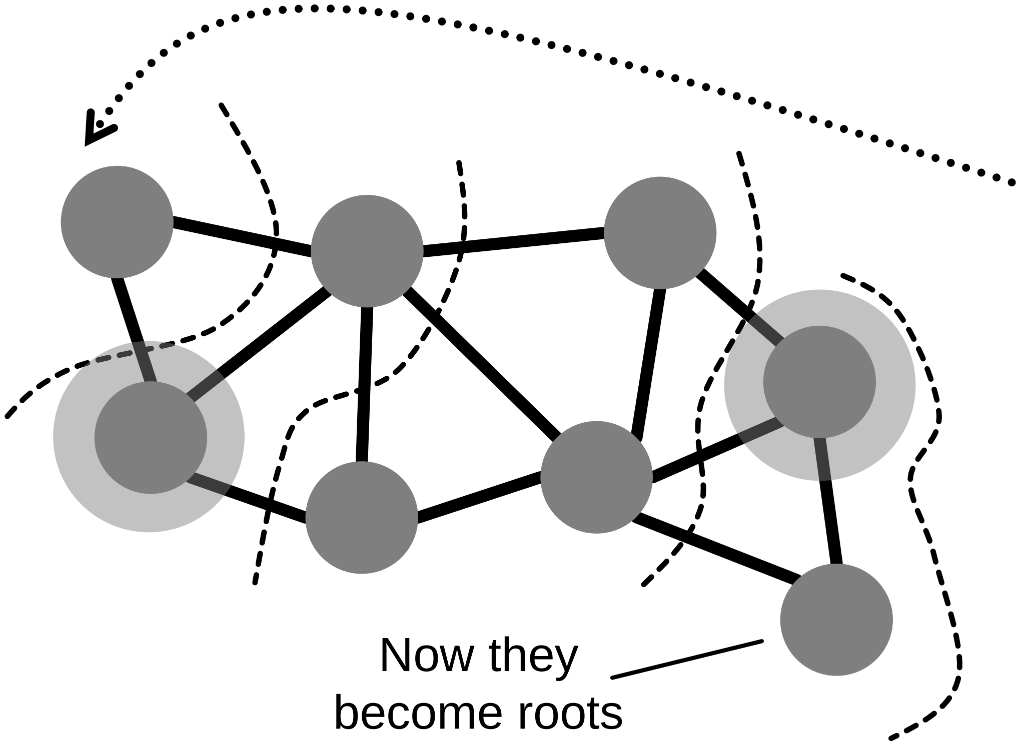
BRANDES [1]

Vertex importance (#shortest paths)

1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices



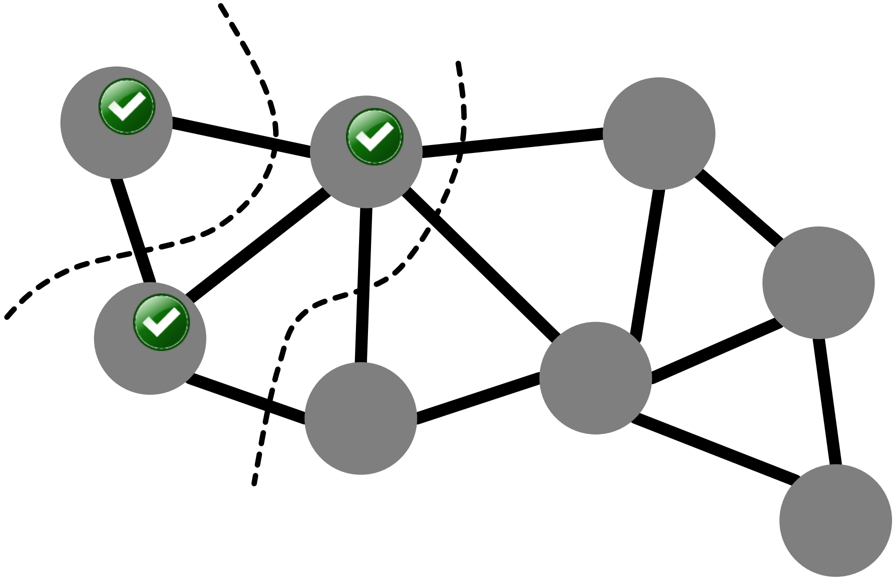
Now they become roots

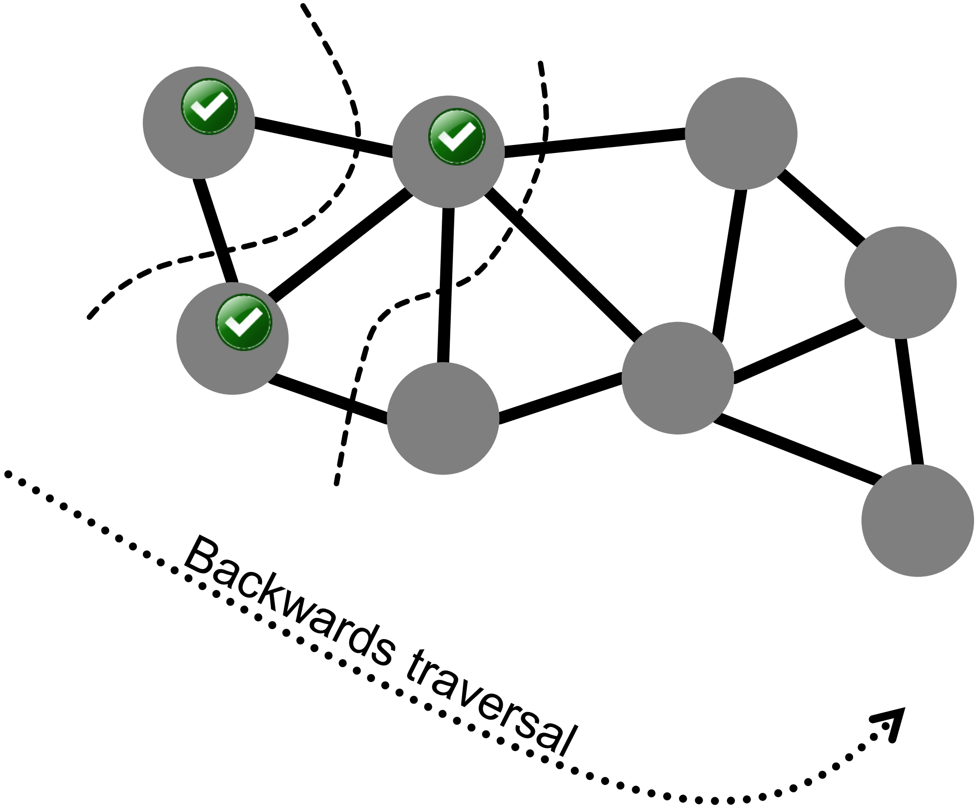
2. Backward traversals

Accumulate centrality scores during backward traversals [1].

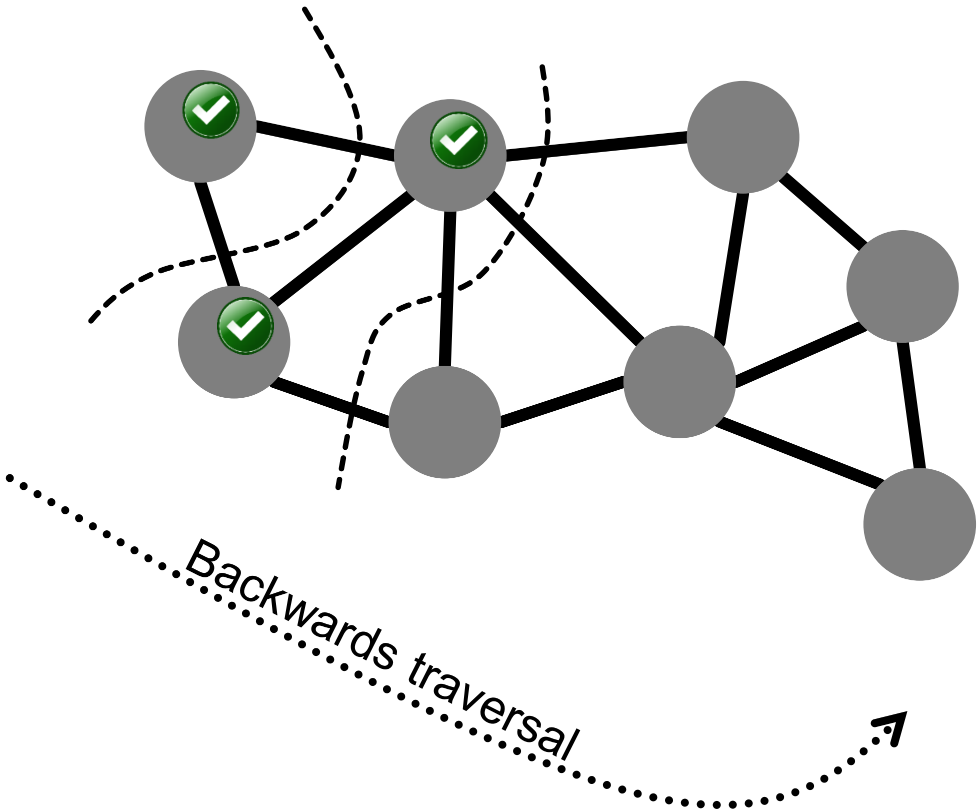
We can do pushing or pulling in both phases

[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

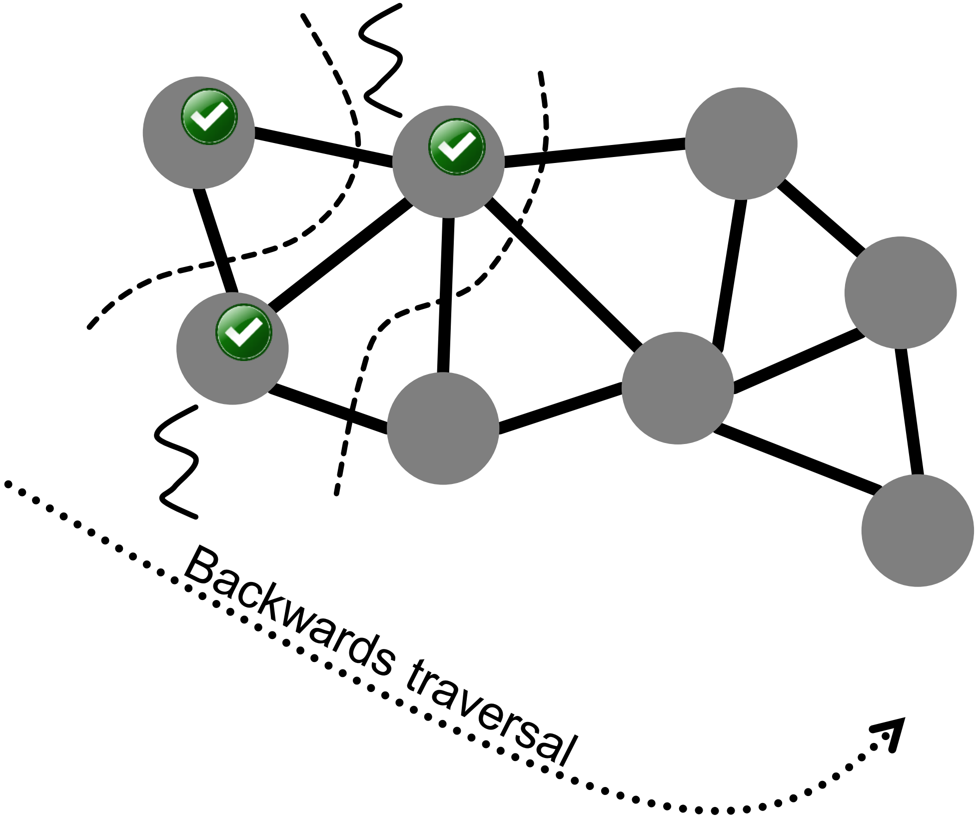



Backwards traversal

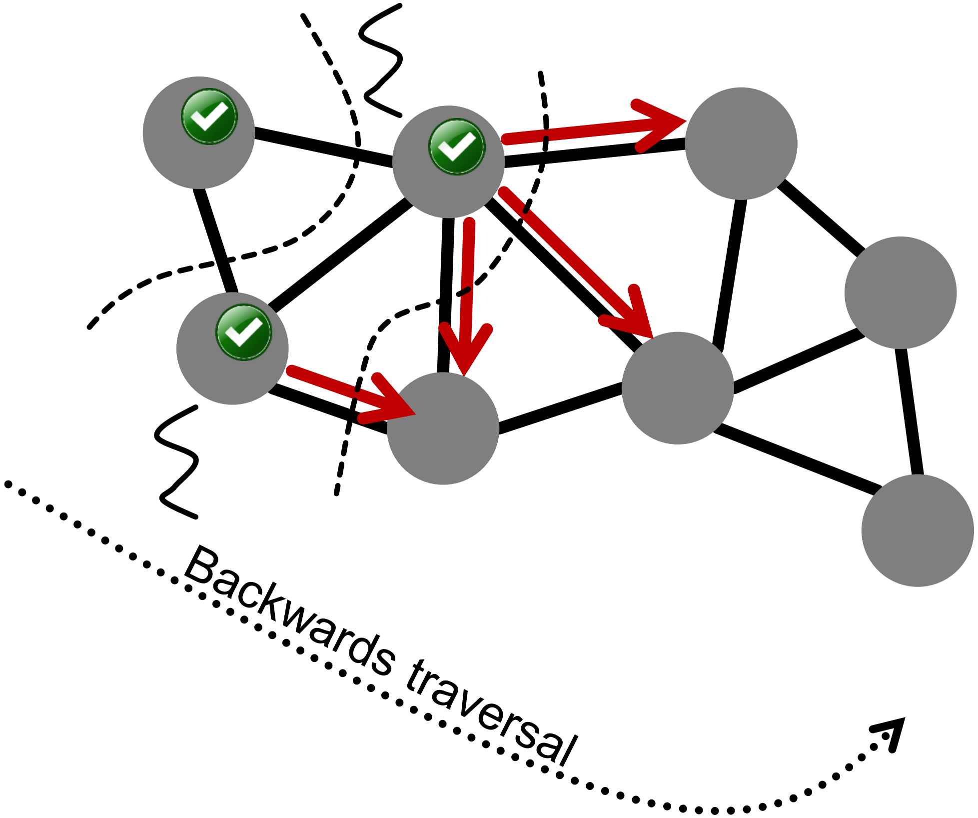
Pushing… like before



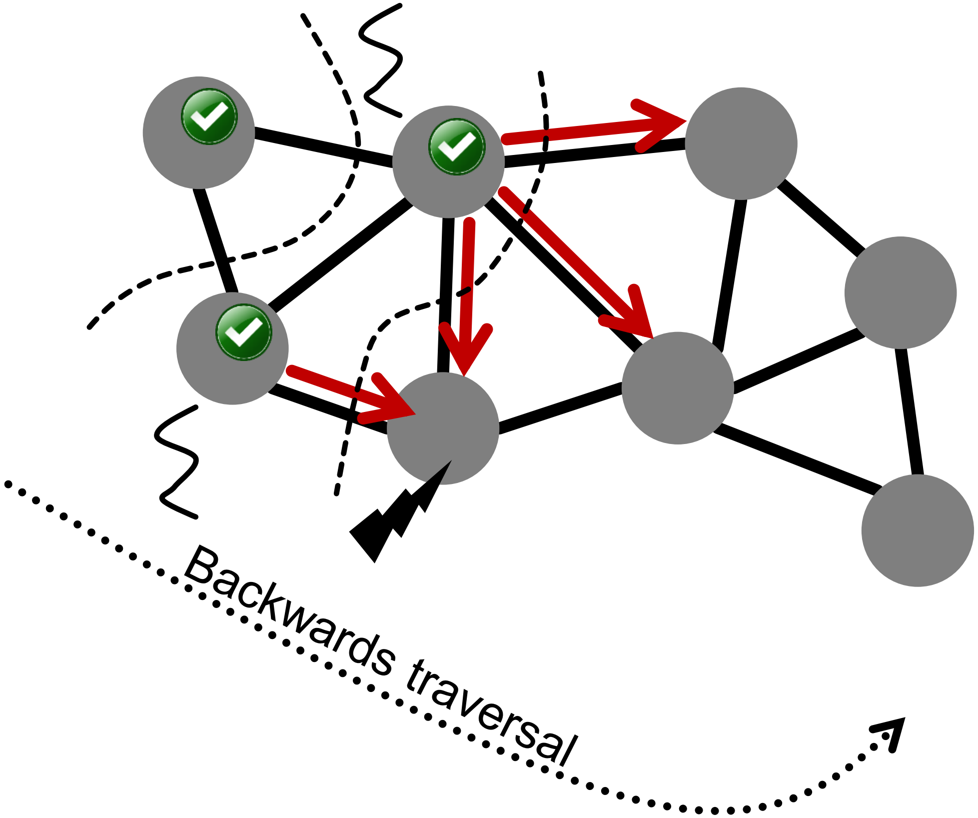
Pushing… like before



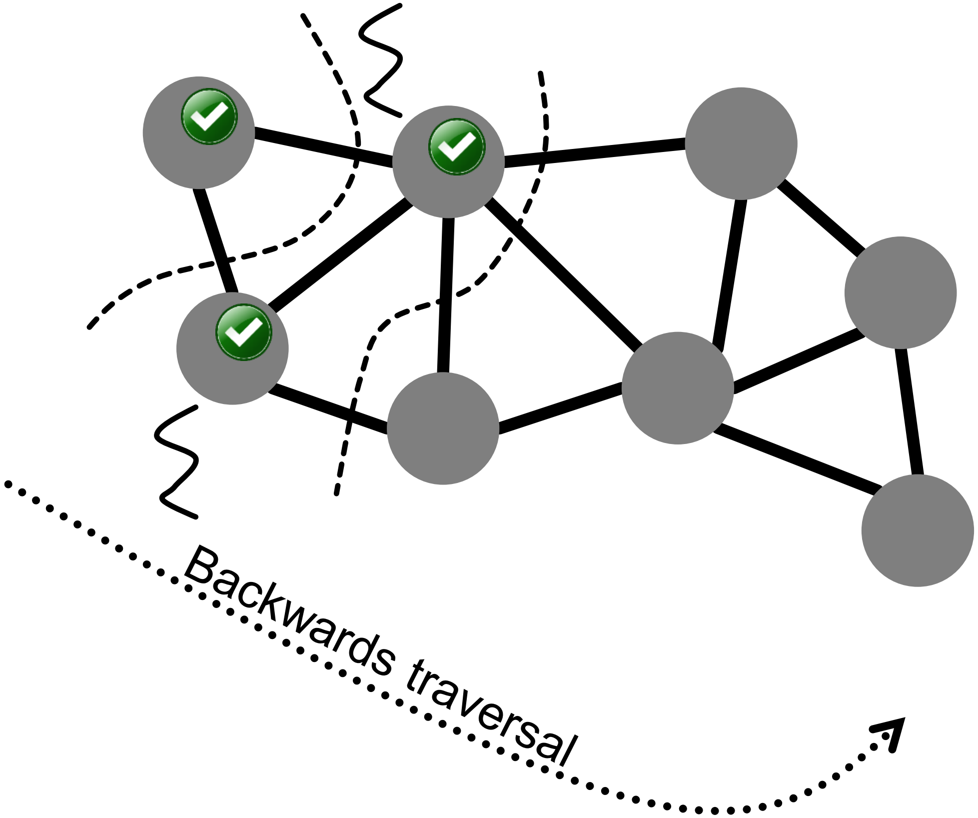
Pushing… like before



Pushing… like before

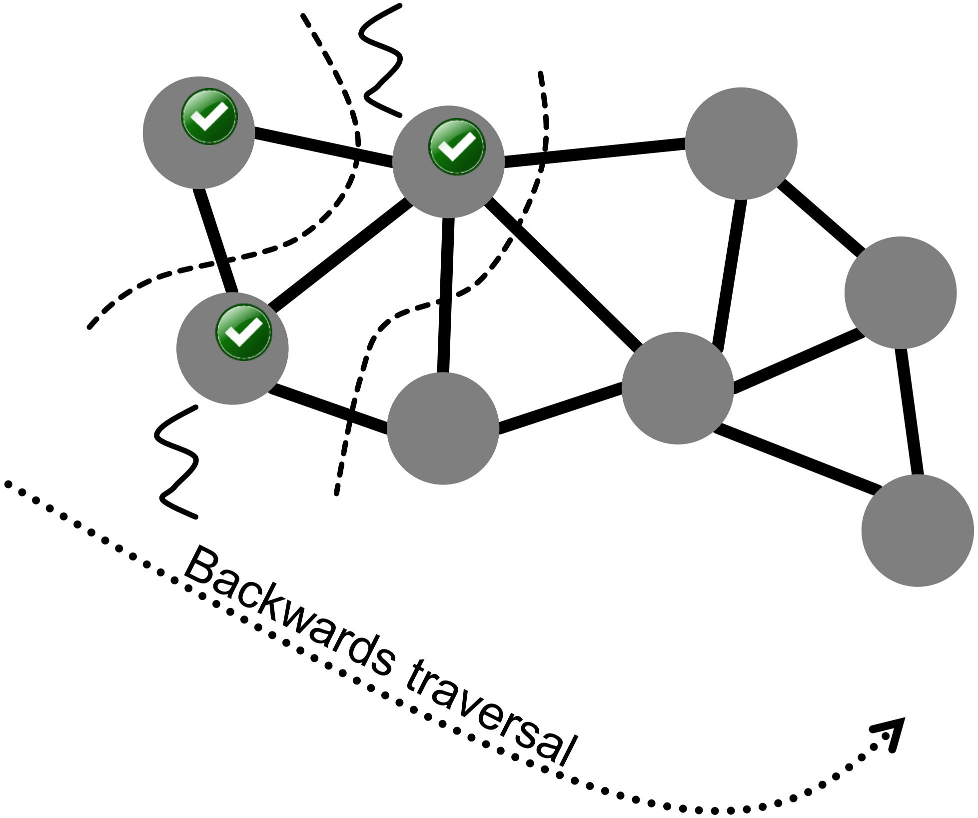


Pushing… like before



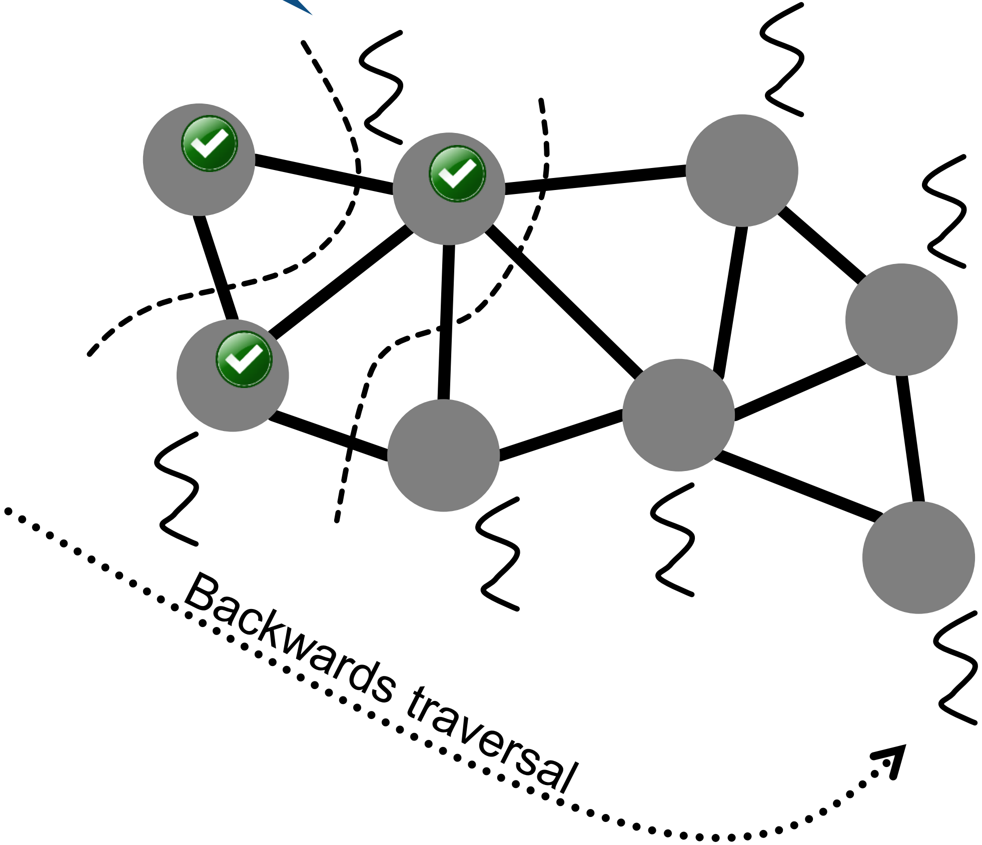
Pushing… like before

Pulling… lower complexity (more performance!)



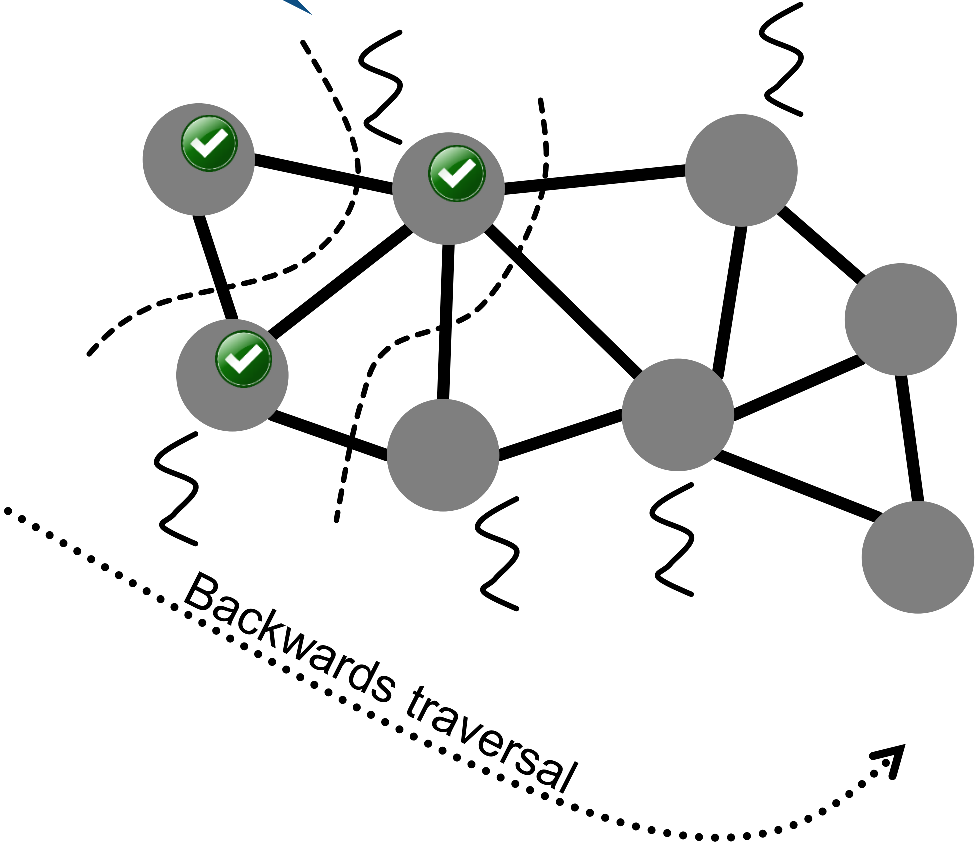
Pushing… like before

Pulling… lower complexity (more performance!)



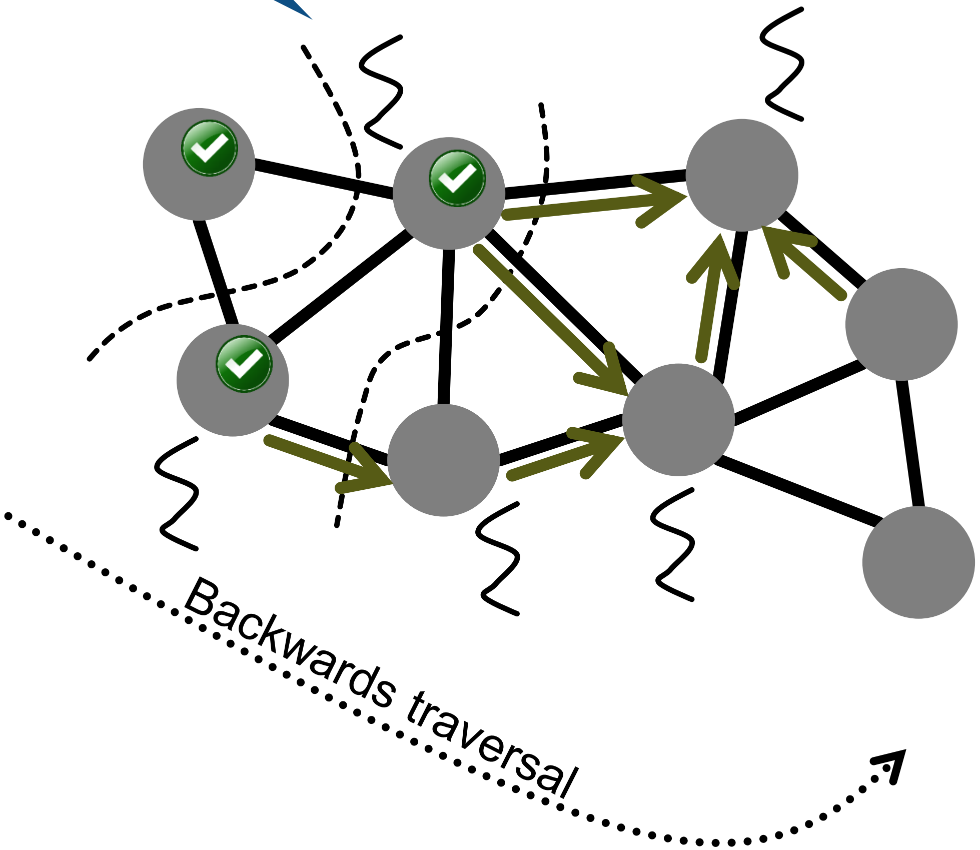
Pushing… like before

Pulling… lower complexity (more performance!)

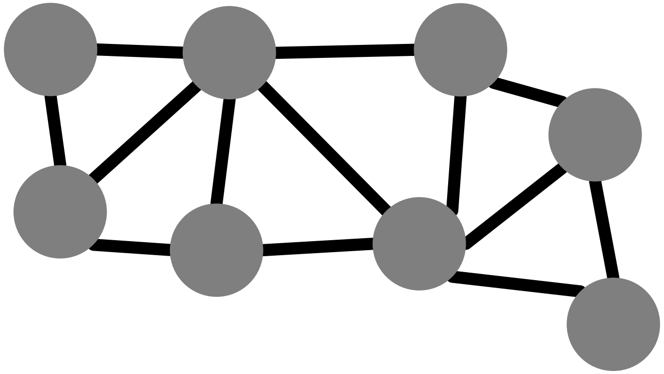


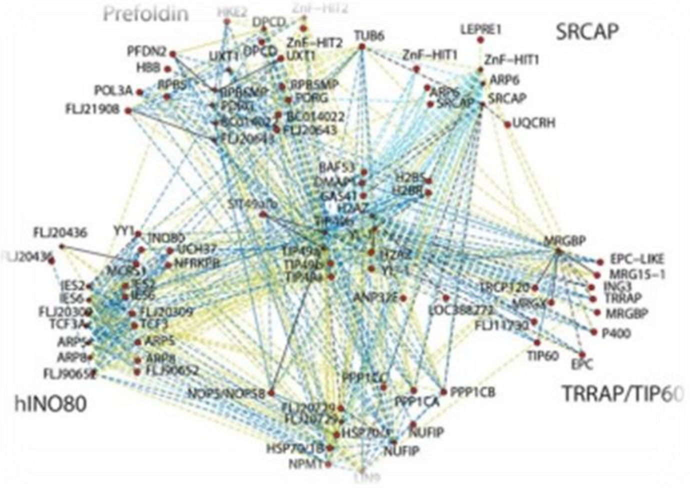
Pushing… like before

Pulling… lower complexity (more performance!)

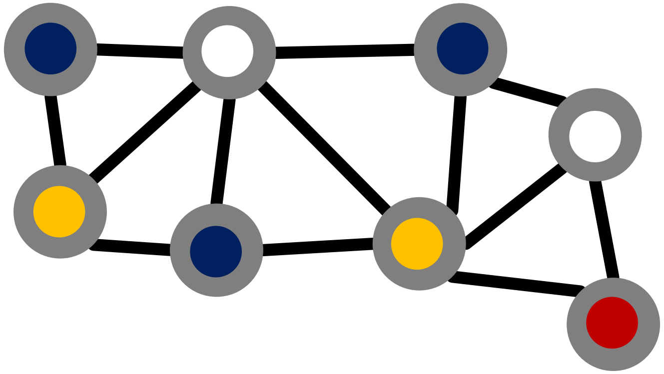


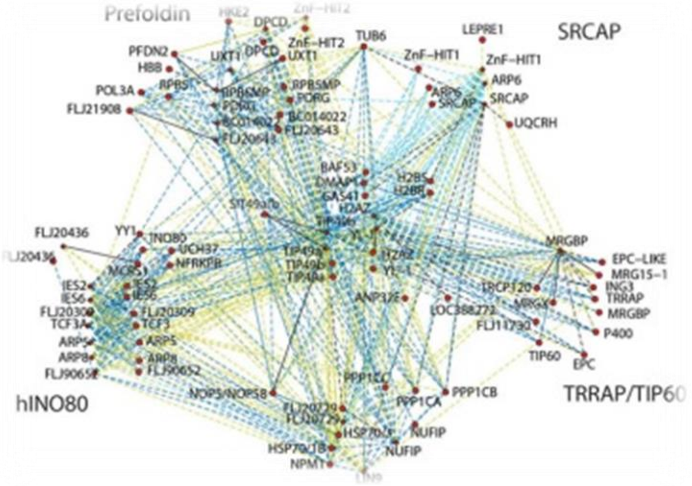
GRAPH COLORING



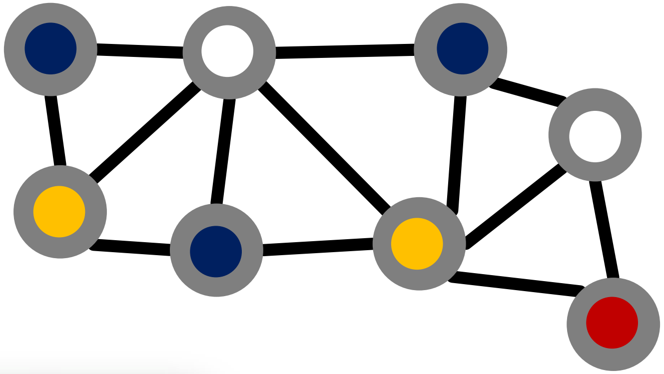


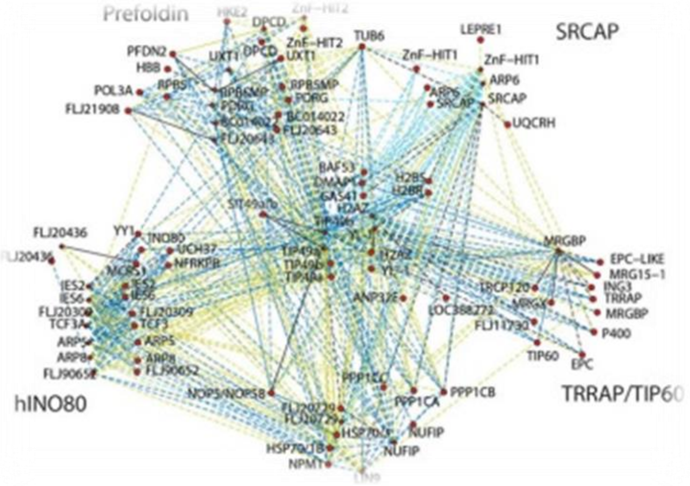
GRAPH COLORING





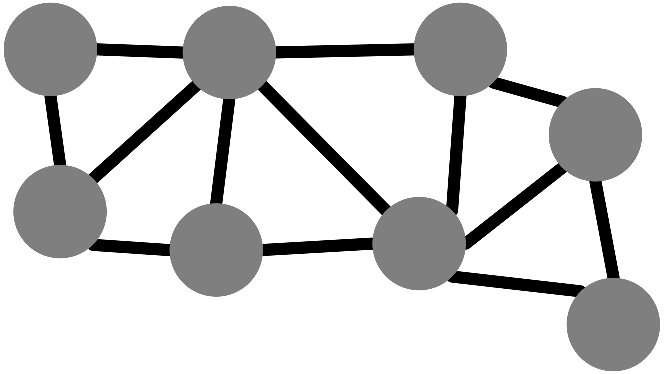
GRAPH COLORING





GRAPH COLORING

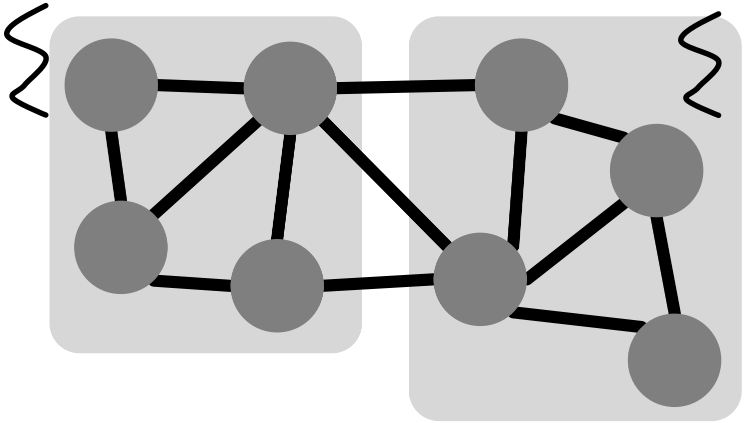
BOMAN ET AL. [1]



[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING

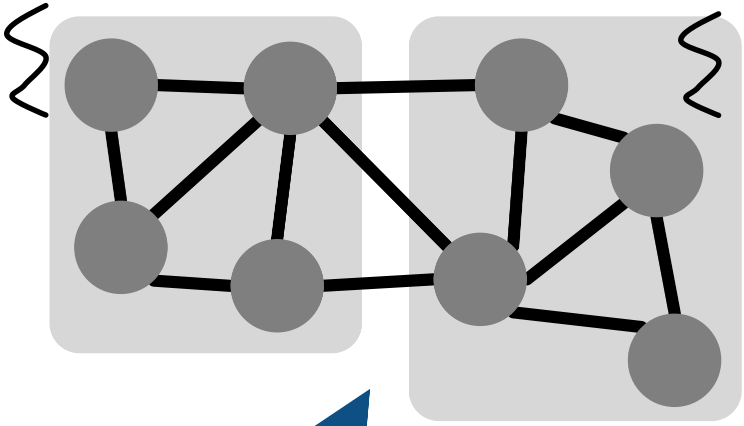
BOMAN ET AL. [1]



[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING

BOMAN ET AL. [1]

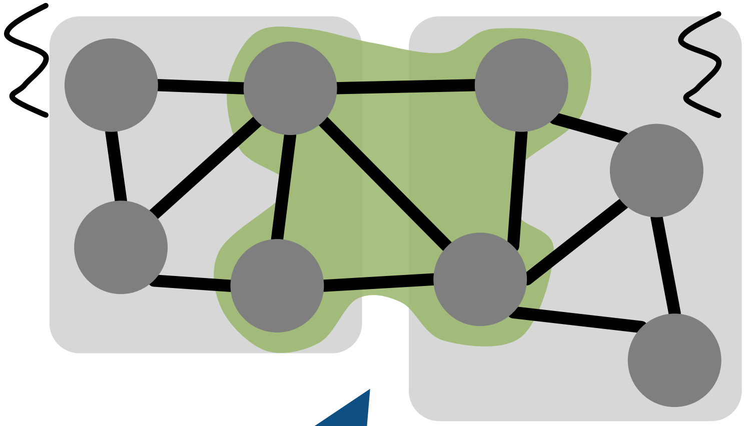


We care explicitly about partitioning now

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING

BOMAN ET AL. [1]

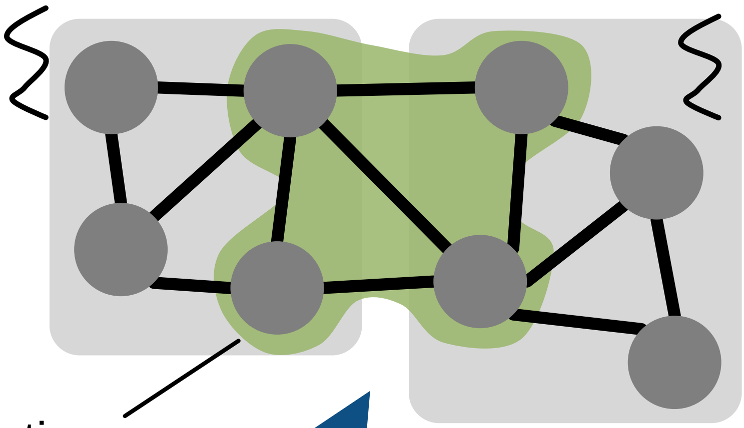


We care explicitly about partitioning now

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING

BOMAN ET AL. [1]



Border vertices

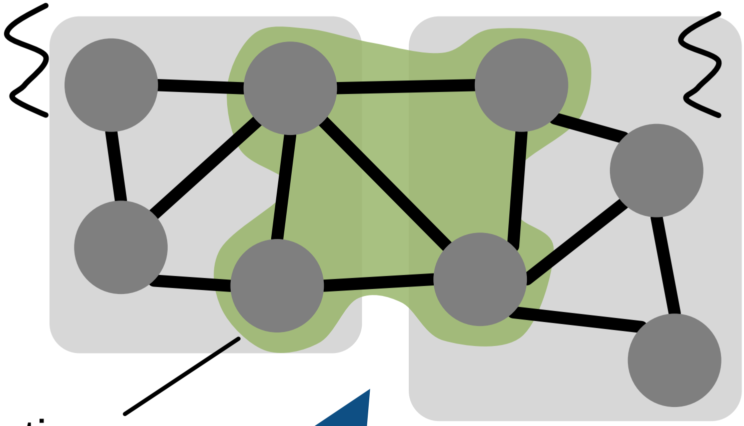
We care explicitly about partitioning now

---

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Iterate until converge (convergence == no color conflicts)



Border vertices

We care explicitly about partitioning now

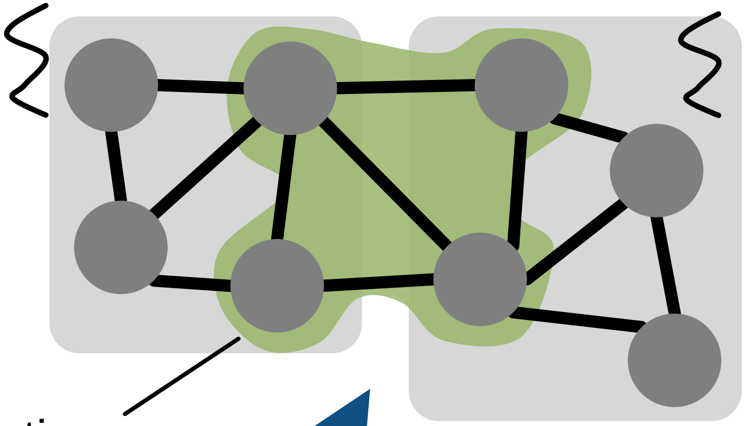
---

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING

BOMAN ET AL. [1]

Iterate until converge (convergence == no color conflicts)



Border vertices

We care explicitly about partitioning now

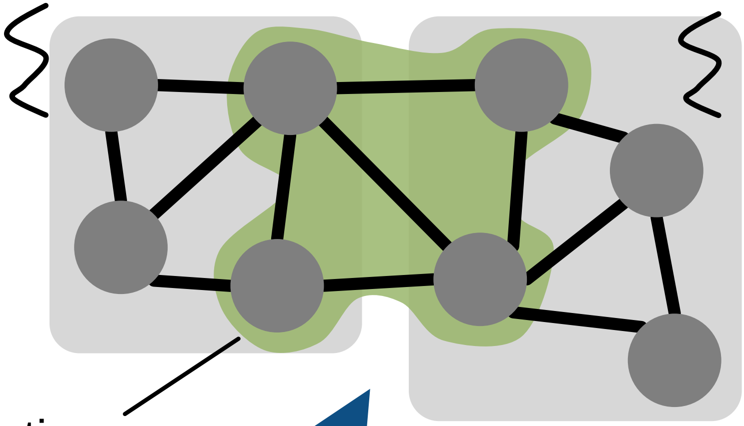
In each iteration:

---

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Iterate until converge (convergence == no color conflicts)



Border vertices

We care explicitly about partitioning now

In each iteration:

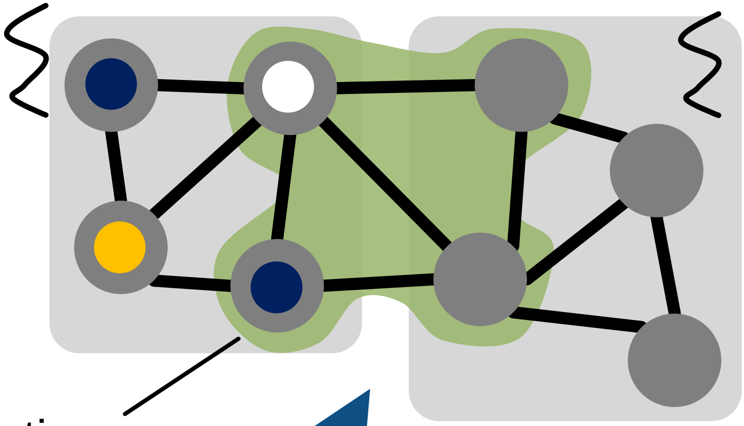
1 Color each partition independently

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING

BOMAN ET AL. [1]

Iterate until converge (convergence == no color conflicts)



Border vertices

We care explicitly about partitioning now

In each iteration:

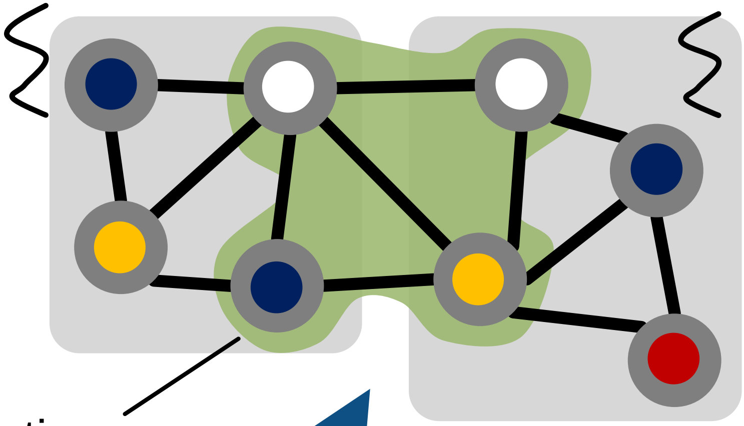
1 Color each partition independently

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING

BOMAN ET AL. [1]

Iterate until converge (convergence == no color conflicts)



Border vertices

We care explicitly about partitioning now

In each iteration:

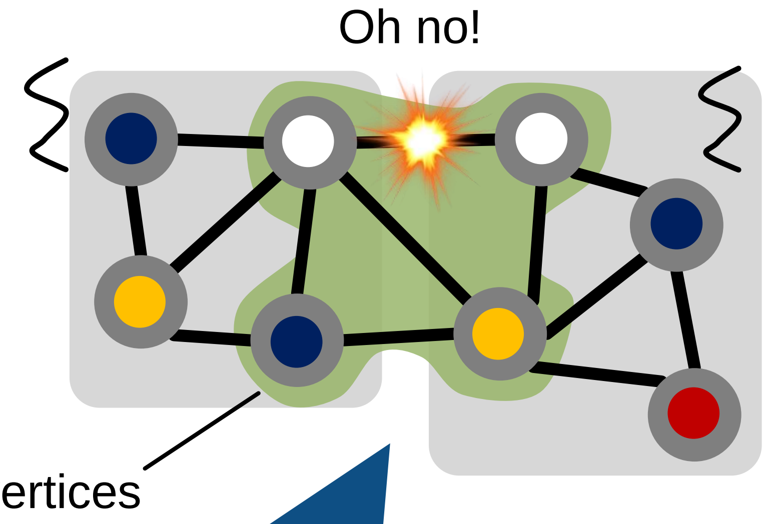
1 Color each partition independently

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING

BOMAN ET AL. [1]

Iterate until converge (convergence == no color conflicts)



We care explicitly about partitioning now

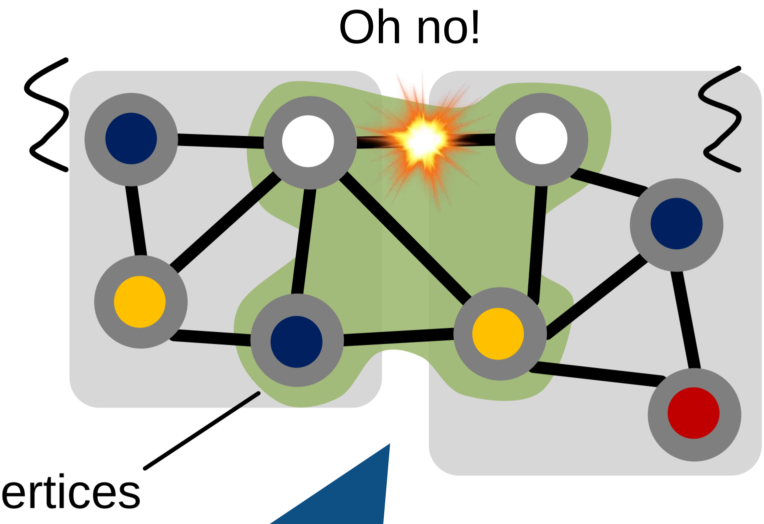
In each iteration:

1 Color each partition independently

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Iterate until converge (convergence == no color conflicts)



We care explicitly about partitioning now

In each iteration:

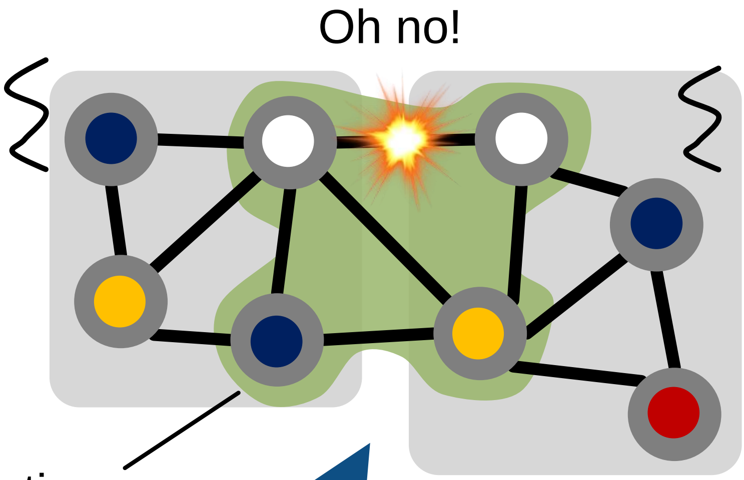
1 Color each partition independently

2 Fix the conflicts

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Iterate until converge (convergence == no color conflicts)



We care explicitly about partitioning now

In each iteration:

1 Color each partition independently

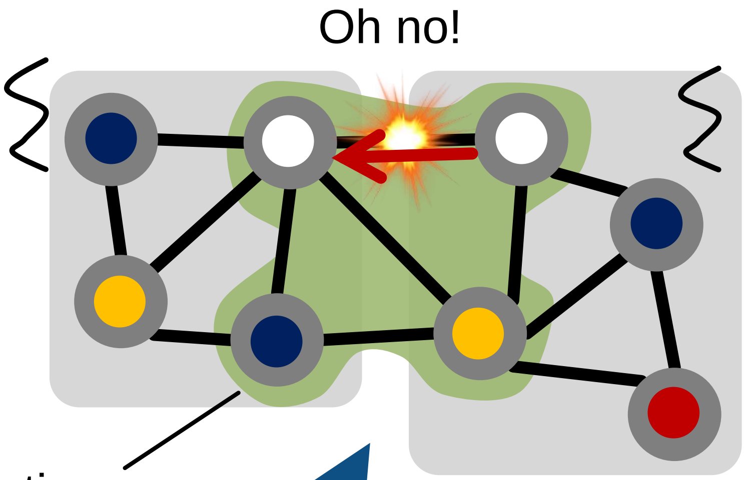
2 Fix the conflicts

Pushing

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Iterate until converge (convergence == no color conflicts)



Oh no!

Border vertices

We care explicitly about partitioning now

In each iteration:

1 Color each partition independently

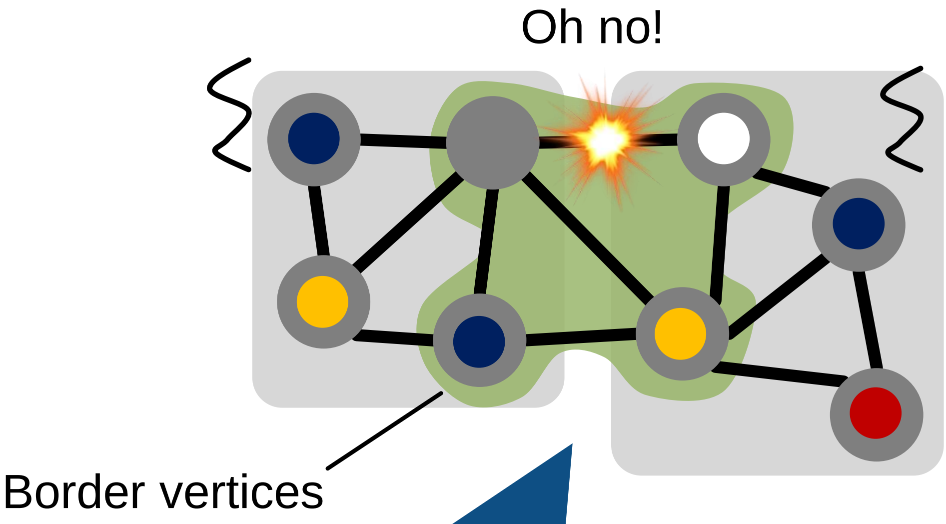
2 Fix the conflicts

Pushing

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Iterate until converge (convergence == no color conflicts)



We care explicitly about partitioning now

In each iteration:

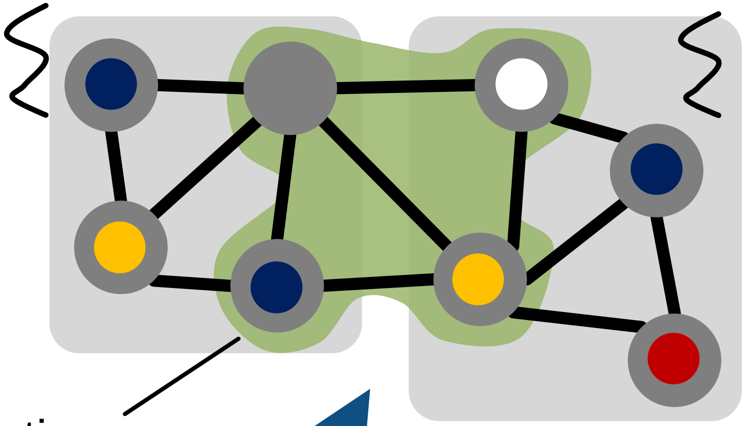
1 Color each partition independently

2 Fix the conflicts

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Iterate until converge (convergence == no color conflicts)



Border vertices

We care explicitly about partitioning now

In each iteration:

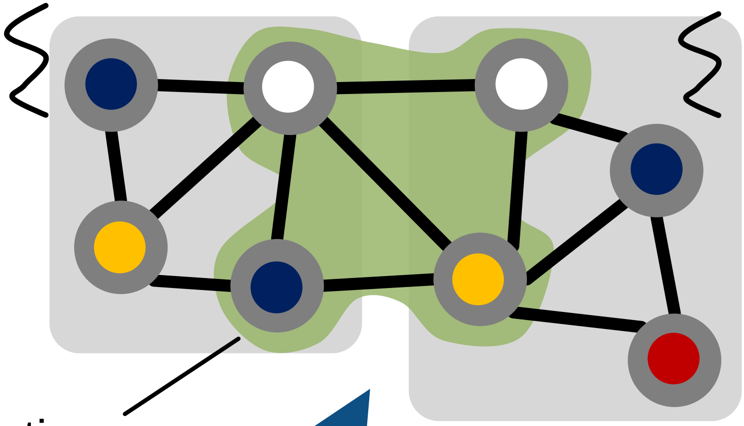
1 Color each partition independently

2 Fix the conflicts

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Iterate until converge (convergence == no color conflicts)



Border vertices

We care explicitly about partitioning now

In each iteration:

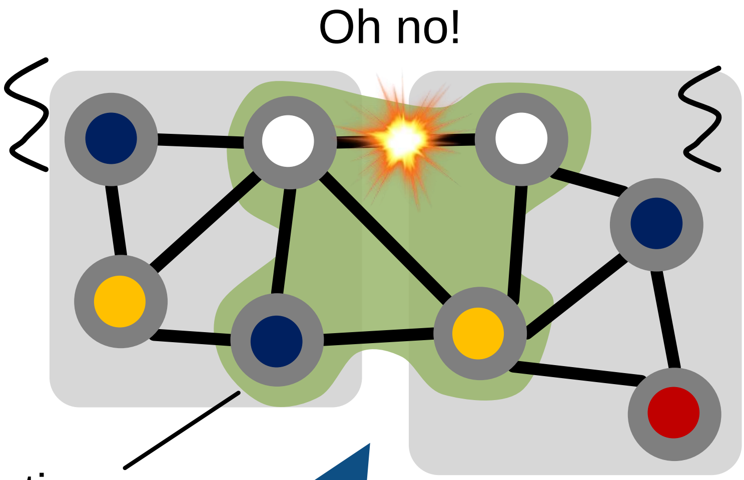
1 Color each partition independently

2 Fix the conflicts

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Iterate until converge (convergence == no color conflicts)



We care explicitly about partitioning now

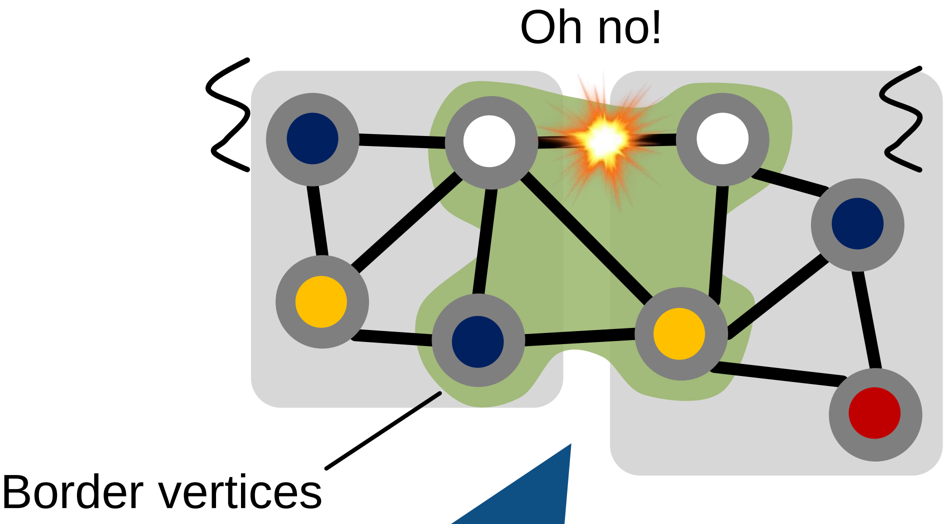
In each iteration:

1 Color each partition independently

2 Fix the conflicts

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]



Iterate until converge (convergence == no color conflicts)

We care explicitly about partitioning now

In each iteration:

1 Color each partition independently

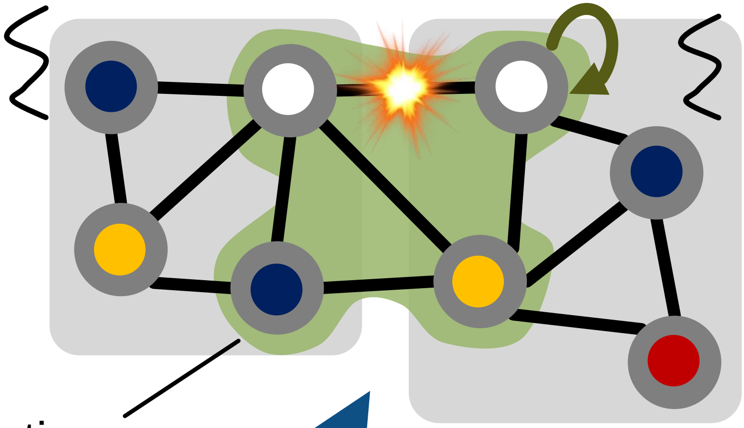
2 Fix the conflicts

Pulling

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Iterate until converge (convergence == no color conflicts)



We care explicitly about partitioning now

In each iteration:

1 Color each partition independently

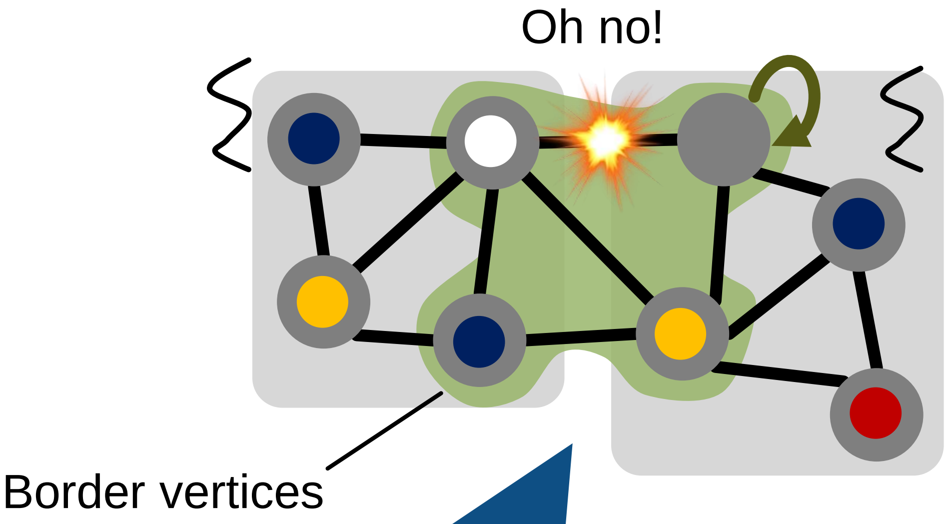
2 Fix the conflicts

Pulling

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Iterate until converge (convergence == no color conflicts)



We care explicitly about partitioning now

In each iteration:

1 Color each partition independently

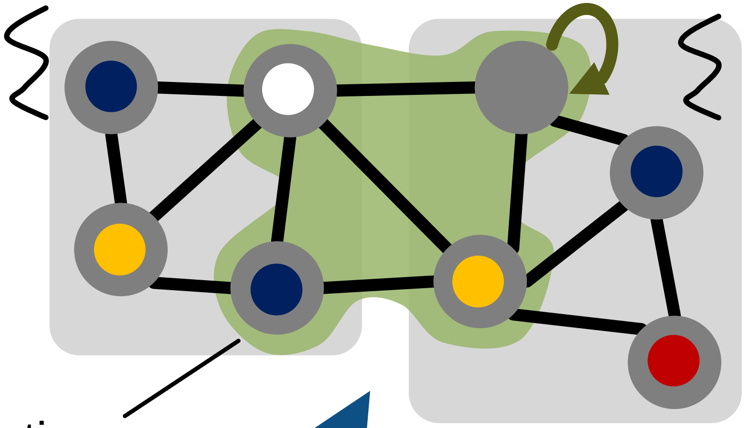
2 Fix the conflicts

Pulling

---

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]



Border vertices

Iterate until converge (convergence == no color conflicts)

We care explicitly about partitioning now

In each iteration:

1 Color each partition independently

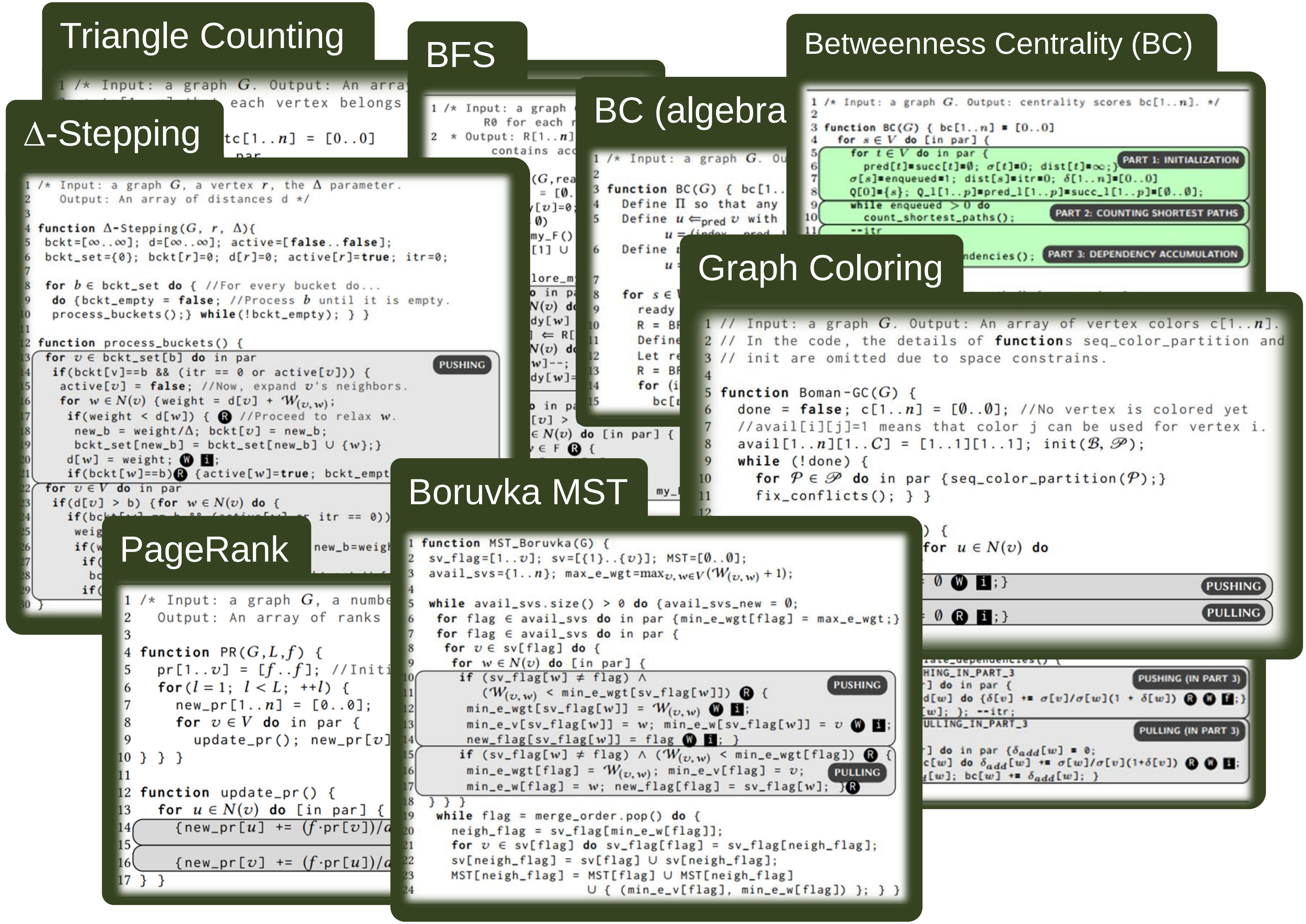
2 Fix the conflicts

Pulling

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

OTHER ALGORITHMS & FORMULATIONS

OTHER ALGORITHMS & FORMULATIONS



Triangle Counting

```
/* Input: a graph G. Output: An array
            each vertex belongs
            tc[1..n] = [0..0]
```

BFS

```
/* Input: a graph
       R0 for each
 * Output: R[1..n]
       contains acc
```

BC (algebraic)

```
/* Input: a graph G. Ou

function BC(G) { bc[1..
  Define Π so that any
  Define u ⇐pred v with
        u = (index  pred
  Define
        u

  for s ∈ V
    ready
    R = BF
    Define
    Let re
    R = BF
    for (i
      bc[
```

Δ-Stepping

```
/* Input: a graph G, a vertex r, the Δ parameter.
   Output: An array of distances d */

function Δ-Stepping(G, r, Δ){
 bckt=[∞..∞]; d=[∞..∞]; active=[false..false];
 bckt_set={0}; bckt[r]=0; d[r]=0; active[r]=true; itr=0;

 for b ∈ bckt_set do { //For every bucket do...
  do {bckt_empty = false; //Process b until it is empty.
  process_buckets();} while(!bckt_empty); } }

function process_buckets() {
 for v ∈ bckt_set[b] do in par
  if(bckt[v]==b && (itr == 0 or active[v])) {
   active[v] = false; //Now, expand v's neighbors.
   for w ∈ N(v) {weight = d[v] + W(v,w);
    if(weight < d[w]) { Ⓡ //Proceed to relax w.
     new_b = weight/Δ; bckt[v] = new_b;
     bckt_set[new_b] = bckt_set[new_b] ∪ {w};}
    d[w] = weight; Ⓦ 🅸;
    if(bckt[w]==b)Ⓡ {active[w]=true; bckt_empt
 for v ∈ V do in par
  if(d[v] > b) {for w ∈ N(v) do {
   if(bc                         itr == 0))
    weig                          new_b=weig
    if(w
     if(
      bc
     if(
}
```

Betweenness Centrality (BC)

```
/* Input: a graph G. Output: centrality scores bc[1..n]. */

function BC(G) { bc[1..n] = [0..0]
  for s ∈ V do [in par] {
    for t ∈ V do in par {
      pred[t]=succ[t]=∅; σ[t]=0; dist[t]=∞;}
    σ[s]=enqueued=1; dist[s]=itr=0; δ[1..n]=[0..0]
    Q[0]={s}; Q_1[1..p]=pred_1[1..p]=succ_1[1..p]=[∅..∅];
    while enqueued > 0 do
      count_shortest_paths();
    --itr
              ndencies();
```

```
HING_IN_PART_3
] do in par {
d[w] do {δ[v] += σ[v]/σ[w](1 + δ[w]) Ⓡ Ⓦ 🅵;}
w]; }; --itr;
ULLING_IN_PART_3

] do in par {δadd[w] = 0;
c[w] do δadd[w] += σ[w]/σ[v](1+δ[v]) Ⓡ Ⓦ 🅸;
[w]; bc[w] += δadd[w]; }
```

Graph Coloring

```
// Input: a graph G. Output: An array of vertex colors c[1..n].
// In the code, the details of functions seq_color_partition and
// init are omitted due to space constrains.

function Boman-GC(G) {
  done = false; c[1..n] = [0..0]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..n][1..C] = [1..1][1..1]; init(ℬ, 𝒫);
  while (!done) {
    for 𝒫 ∈ 𝒫 do in par {seq_color_partition(𝒫);}
    fix_conflicts(); } }

) {
for u ∈ N(v) do
∅ Ⓦ 🅸;}
∅ Ⓡ 🅸;}
```

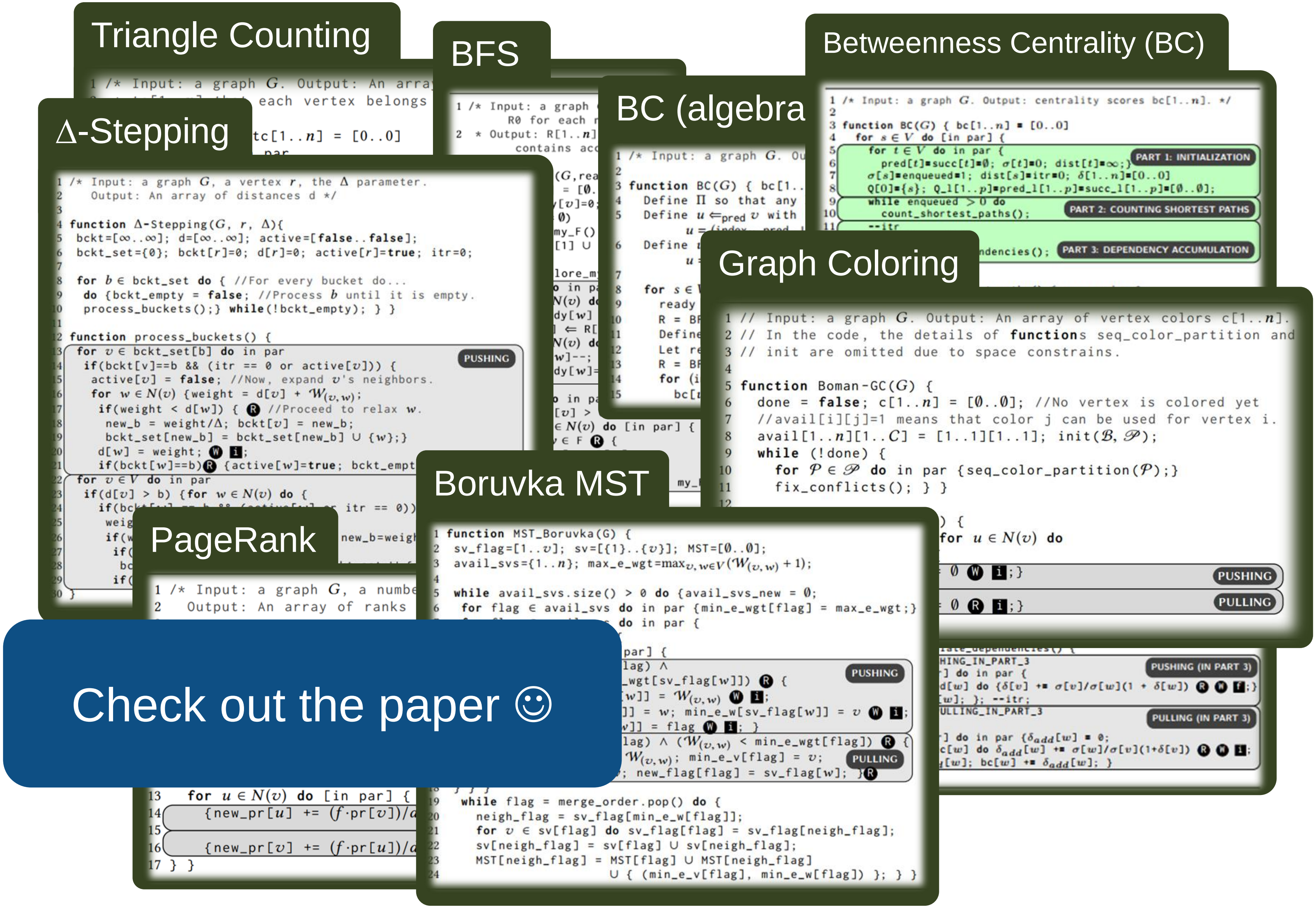
PageRank

```
/* Input: a graph G, a numbe
  Output: An array of ranks

function PR(G,L,f) {
  pr[1..v] = [f..f]; //Initi
  for(l = 1; l < L; ++l) {
    new_pr[1..n] = [0..0];
    for v ∈ V do in par {
      update_pr(); new_pr[v]
} } }

function update_pr() {
  for u ∈ N(v) do [in par] {
      {new_pr[u] += (f·pr[v])/d

      {new_pr[v] += (f·pr[u])/d
} }
```

Boruvka MST

```
function MST_Boruvka(G) {
 sv_flag=[1..v]; sv=[{1}..{v}]; MST=[∅..∅];
 avail_svs={1..n}; max_e_wgt=max_{v,w∈V}(W(v,w) + 1);

 while avail_svs.size() > 0 do {avail_svs_new = ∅;
  for flag ∈ avail_svs do in par {min_e_wgt[flag] = max_e_wgt;}
  for flag ∈ avail_svs do in par {
   for v ∈ sv[flag] do {
    for w ∈ N(v) do [in par] {
     if (sv_flag[w] ≠ flag) ∧
        (W(v,w) < min_e_wgt[sv_flag[w]]) Ⓡ {
      min_e_wgt[sv_flag[w]] = W(v,w) Ⓦ 🅸;
      min_e_v[sv_flag[w]] = w; min_e_w[sv_flag[w]] = v Ⓦ 🅸;
      new_flag[sv_flag[w]] = flag Ⓦ 🅸; }
     if (sv_flag[w] ≠ flag) ∧ (W(v,w) < min_e_wgt[flag]) Ⓡ {
      min_e_wgt[flag] = W(v,w); min_e_v[flag] = v;
      min_e_w[flag] = w; new_flag[flag] = sv_flag[w]; }Ⓡ
 } } }
  while flag = merge_order.pop() do {
    neigh_flag = sv_flag[min_e_w[flag]];
    for v ∈ sv[flag] do sv_flag[flag] = sv_flag[neigh_flag];
    sv[neigh_flag] = sv[flag] ∪ sv[neigh_flag];
    MST[neigh_flag] = MST[flag] ∪ MST[neigh_flag]
                      ∪ { (min_e_v[flag], min_e_w[flag]) }; } }
```

OTHER ALGORITHMS & FORMULATIONS



Triangle Counting

```
/* Input: a graph G. Output: An array
   each vertex belongs
 tc[1..n] = [0..0]
 par
```

BFS

```
/* Input: a graph
   R0 for each r
 * Output: R[1..n]
   contains acc
```

BC (algebra

```
/* Input: a graph G. Ou

function BC(G) { bc[1..
 Define Π so that any
 Define u ⇐pred v with
   u = (index  pred
 Define
   u
 for s ∈
  ready
  R = BF
  Define
  Let re
  R = BF
  for (i
   bc[
```

Betweenness Centrality (BC)

```
/* Input: a graph G. Output: centrality scores bc[1..n]. */

function BC(G) { bc[1..n] = [0..0]
 for s ∈ V do [in par] {
  for t ∈ V do in par {
   pred[t]=succ[t]=∅; σ[t]=0; dist[t]=∞;}        PART 1: INITIALIZATION
  σ[s]=enqueued=1; dist[s]=itr=0; δ[1..n]=[0..0]
  Q[0]={s}; Q_1[1..p]=pred_1[1..p]=succ_1[1..p]=[∅..∅];
  while enqueued > 0 do
   count_shortest_paths();                        PART 2: COUNTING SHORTEST PATHS
  --itr
  ndencies();                                     PART 3: DEPENDENCY ACCUMULATION

HING_IN_PART_3                                    PUSHING (IN PART 3)
] do in par {
d[w] do {δ[v] += σ[v]/σ[w](1 + δ[w]) Ⓡ Ⓦ 🅵;}
w]; }; --itr;
ULLING_IN_PART_3                                  PULLING (IN PART 3)
] do in par {δ_add[w] = 0;
c[w] do δ_add[w] += σ[w]/σ[v](1+δ[v]) Ⓡ Ⓦ 🅸;
[w]; bc[w] += δ_add[w]; }
```

Δ-Stepping

```
/* Input: a graph G, a vertex r, the Δ parameter.
   Output: An array of distances d */

function Δ-Stepping(G, r, Δ){
 bckt=[∞..∞]; d=[∞..∞]; active=[false..false];
 bckt_set={0}; bckt[r]=0; d[r]=0; active[r]=true; itr=0;

 for b ∈ bckt_set do { //For every bucket do...
  do {bckt_empty = false; //Process b until it is empty.
  process_buckets();} while(!bckt_empty); } }

function process_buckets() {
 for v ∈ bckt_set[b] do in par                         PUSHING
  if(bckt[v]==b && (itr == 0 or active[v])) {
   active[v] = false; //Now, expand v's neighbors.
   for w ∈ N(v) {weight = d[v] + W(v,w);
    if(weight < d[w]) { Ⓡ //Proceed to relax w.
     new_b = weight/Δ; bckt[v] = new_b;
     bckt_set[new_b] = bckt_set[new_b] ∪ {w};}
    d[w] = weight; Ⓦ 🅸;
    if(bckt[w]==b)Ⓡ {active[w]=true; bckt_empt
 for v ∈ V do in par
  if(d[v] > b) {for w ∈ N(v) do {
   if(bc           == b && (active[w] or itr == 0))
    weig
    if(w                    new_b=weigh
     if(
      bc
     if(
}
```

Graph Coloring

```
// Input: a graph G. Output: An array of vertex colors c[1..n].
// In the code, the details of functions seq_color_partition and
// init are omitted due to space constrains.

function Boman-GC(G) {
 done = false; c[1..n] = [0..0]; //No vertex is colored yet
 //avail[i][j]=1 means that color j can be used for vertex i.
 avail[1..n][1..C] = [1..1][1..1]; init(ℬ, 𝒫);
 while (!done) {
  for 𝒫 ∈ 𝒫 do in par {seq_color_partition(𝒫);}
  fix_conflicts(); } }

) {
 for u ∈ N(v) do
  ∅ Ⓦ 🅸;}                                           PUSHING
  ∅ Ⓡ 🅸;}                                           PULLING
```

PageRank

```
/* Input: a graph G, a numbe
   Output: An array of ranks

 for u ∈ N(v) do [in par] {
  {new_pr[u] += (f·pr[v])/d
  {new_pr[v] += (f·pr[u])/d
} }
```

Boruvka MST

```
function MST_Boruvka(G) {
 sv_flag=[1..v]; sv=[{1}..{v}]; MST=[∅..∅];
 avail_svs={1..n}; max_e_wgt=max_{v,w∈V}(W(v,w) + 1);

 while avail_svs.size() > 0 do {avail_svs_new = ∅;
  for flag ∈ avail_svs do in par {min_e_wgt[flag] = max_e_wgt;}
  for flag ∈ do in par {
  par] {
  lag) ∧
  _wgt[sv_flag[w]]) Ⓡ {                                PUSHING
  w]] = W(v,w) Ⓦ 🅸;
  ]] = w; min_e_w[sv_flag[w]] = v Ⓦ 🅸;
  v]] = flag Ⓦ 🅸; }
  lag) ∧ (W(v,w) < min_e_wgt[flag]) Ⓡ {
  W(v,w); min_e_v[flag] = v;                           PULLING
  ; new_flag[flag] = sv_flag[w]; }Ⓡ
 while flag = merge_order.pop() do {
  neigh_flag = sv_flag[min_e_w[flag]];
  for v ∈ sv[flag] do sv_flag[flag] = sv_flag[neigh_flag];
  sv[neigh_flag] = sv[flag] ∪ sv[neigh_flag];
  MST[neigh_flag] = MST[flag] ∪ MST[neigh_flag]
                   ∪ { (min_e_v[flag], min_e_w[flag]) }; } }
```

Check out the paper ☺

PUSHING VS. PULLING

RESEARCH QUESTIONS

What are push-pull formulations of other algorithms?

Can we apply the push-pull dichotomy to other graph algorithms?

What pushing vs. pulling *really* is?

How do they differ in complexity?

What is performance?

PUSHING VS. PULLING

RESEARCH QUESTIONS

Yes (developed 7 algorithms and the total of 11 variants)

Check the paper ☺

What pushing vs. pulling *really* is?

How do they differ in complexity?

What is performance?

PUSHING VS. PULLING
RESEARCH QUESTIONS

Yes (developed 7 algorithms and the total of 11 variants)

Check the paper ☺

What pushing vs. pulling *really* is?

How do they differ in complexity?

What is performance?

PUSHING VS. PULLING

GENERIC DIFFERENCES

PUSHING VS. PULLING

GENERIC DIFFERENCES

What pushing vs. pulling *really* is?

PUSHING VS. PULLING
GENERIC DIFFERENCES

- Vertices: $v \in V$
- $t \rightsquigarrow v \Leftrightarrow t$ modifies v
- $t[v]$: a thread that owns v

What pushing vs. pulling *really* is?

PUSHING VS. PULLING

GENERIC DIFFERENCES

- Vertices: $v \in V$
- $t \leadsto v \Leftrightarrow t$ modifies v
- $t[v]$: a thread that owns v

What pushing vs. pulling *really* is?

Algorithm uses pushing ⇔

$$(\exists t\ \exists v \in V: \ t \leadsto v \land t \neq t[v])$$

PUSHING VS. PULLING

GENERIC DIFFERENCES

- Vertices: $v \in V$
- $t \rightsquigarrow v \Leftrightarrow t$ modifies v
- $t[v]$: a thread that owns v

What pushing vs. pulling *really* is?

Algorithm uses pushing ⇔

$$(\exists t\ \exists v \in V:\ \ t \rightsquigarrow v \wedge t \neq t[v])$$

Algorithm uses pulling ⇔

$$(\forall t\ \forall v \in V:\ \ t \rightsquigarrow v \Rightarrow t = t[v])$$

PUSHING VS. PULLING
GENERIC DIFFERENCES

- Vertices: $v \in V$
- $t \rightsquigarrow v \Leftrightarrow t$ modifies v
- $t[v]$: a thread that owns v

What pushing vs. pulling *really* is?

Algorithm uses pushing ⇔

$$(\exists t\ \exists v \in V\colon\ t \rightsquigarrow v \wedge t \neq t[v])$$

This *is* the actual dichotomy

Algorithm uses pulling ⇔

$$(\forall t\ \forall v \in V\colon\ t \rightsquigarrow v \Rightarrow t = t[v])$$

PUSHING VS. PULLING

GENERIC DIFFERENCES

- Vertices: $v \in V$
- $t \rightsquigarrow v \Leftrightarrow t$ modifies v
- $t[v]$: a thread that owns v

What pushing vs. pulling *really* is?

Algorithm uses pushing ⇔

$$\sim\left[(\exists t\ \exists v \in V:\ \ t \rightsquigarrow v \wedge t \neq t[v])\right]$$

This *is* the actual dichotomy

Algorithm uses pulling ⇔

$$(\forall t\ \forall v \in V:\ \ t \rightsquigarrow v \Rightarrow t = t[v])$$

PUSHING VS. PULLING

GENERIC DIFFERENCES

- Vertices: $v \in V$
- $t \rightsquigarrow v \Leftrightarrow t$ modifies v
- $t[v]$: a thread that owns v

What pushing vs. pulling *really* is?

$$\sim\left[\begin{array}{c}\text{Algorithm uses pushing} \Leftrightarrow \\ (\exists t\ \exists v \in V\colon\ t \rightsquigarrow v \land t \neq t[v])\end{array}\right]$$

$$||$$

$$\begin{array}{c}\text{Algorithm uses pulling} \Leftrightarrow \\ (\forall t\ \forall v \in V\colon\ t \rightsquigarrow v \Rightarrow t = t[v])\end{array}$$

This *is* the actual dichotomy

PUSHING VS. PULLING

RESEARCH QUESTIONS

Yes (developed 7 algorithms and the total of 11 variants)

Check the paper ☺

What pushing vs. pulling *really* is?

How do they differ in complexity?

What is performance?

PUSHING VS. PULLING

RESEARCH QUESTIONS

Yes (developed 7 algorithms and the total of 11 variants)

Check the paper ☺

Can be described with the actual dichotomy

How do they differ in complexity?

What is performance?

PUSHING VS. PULLING
RESEARCH QUESTIONS

Yes (developed 7 algorithms and the total of 11 variants)

Check the paper ☺

Can be described with the actual dichotomy

How do they differ in complexity?

What is performance?

❗ Before we move to the complexity analysis…

Before we move to the complexity analysis…

…a brief recap on PRAM models.

PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

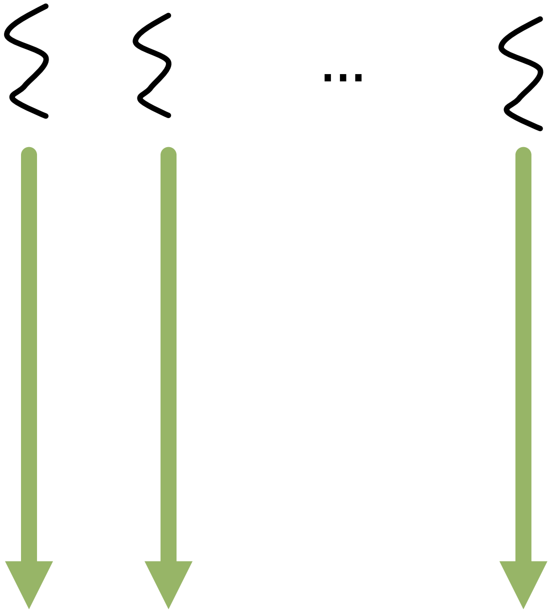
All processes process in lock-steps, communicate by reading from & writing to a shared memory.

PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

All processes process in lock-steps, communicate by reading from & writing to a shared memory.

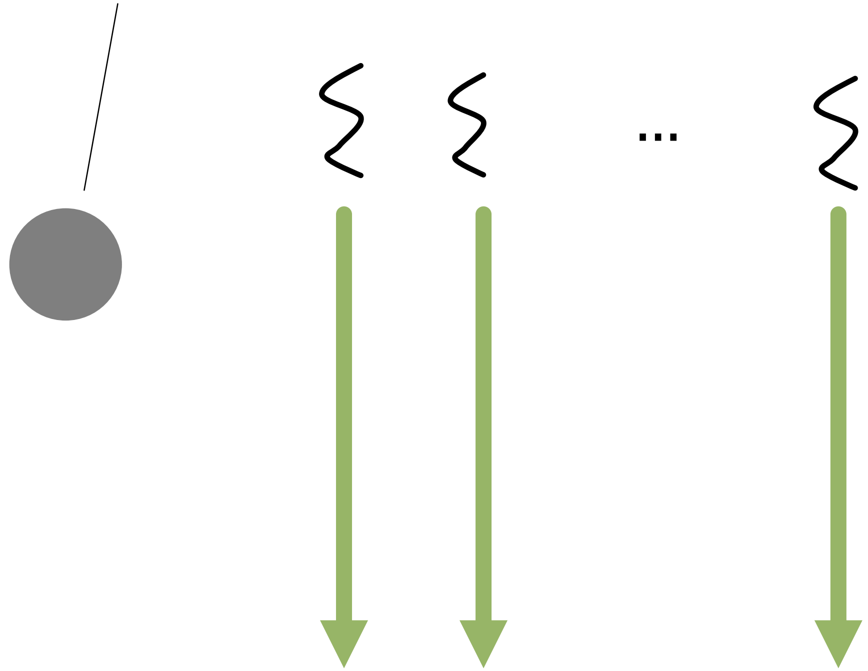
PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

All processes process in lock-steps, communicate by reading from & writing to a shared memory.



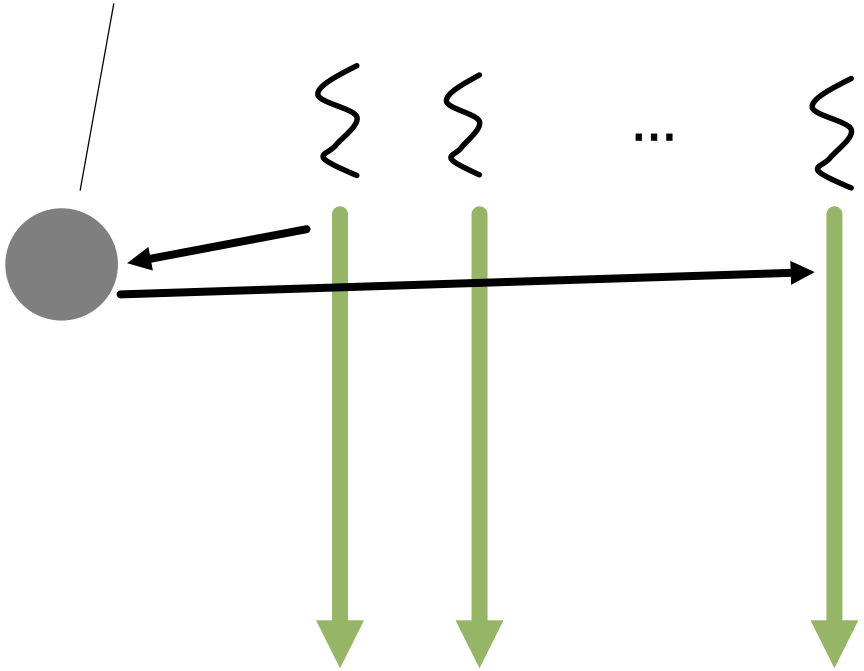
PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

All processes process in lock-steps, communicate by reading from & writing to a shared memory.



PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

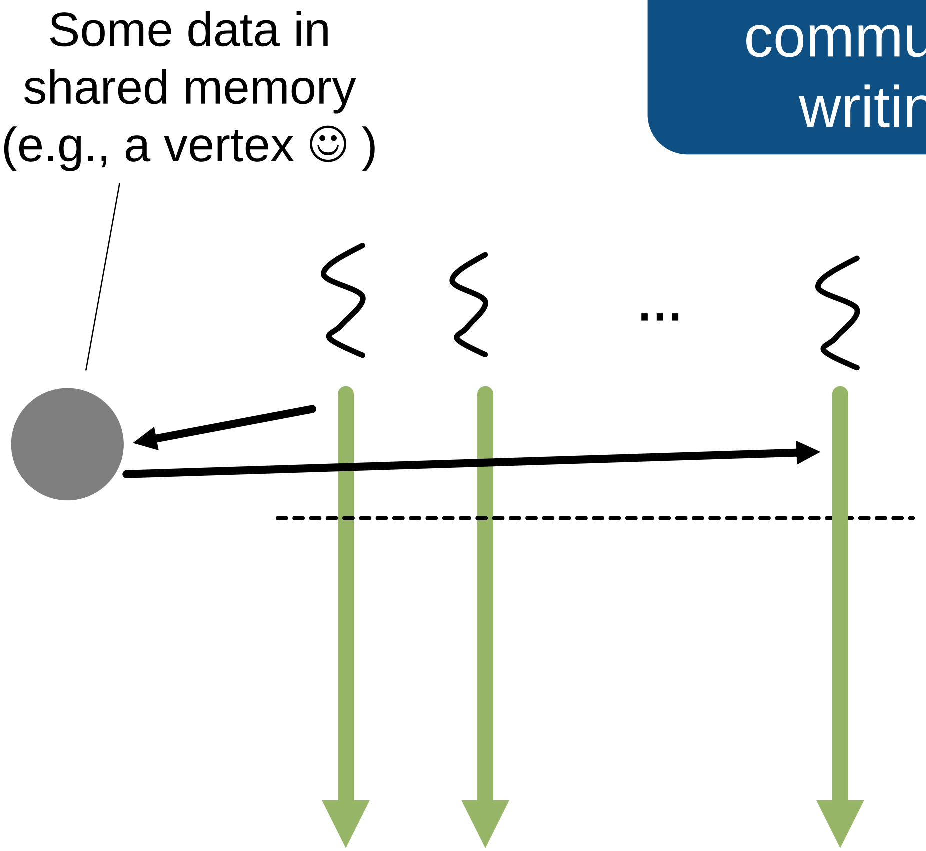
All processes process in lock-steps, communicate by reading from & writing to a shared memory.



Some data in shared memory (e.g., a vertex ☺)

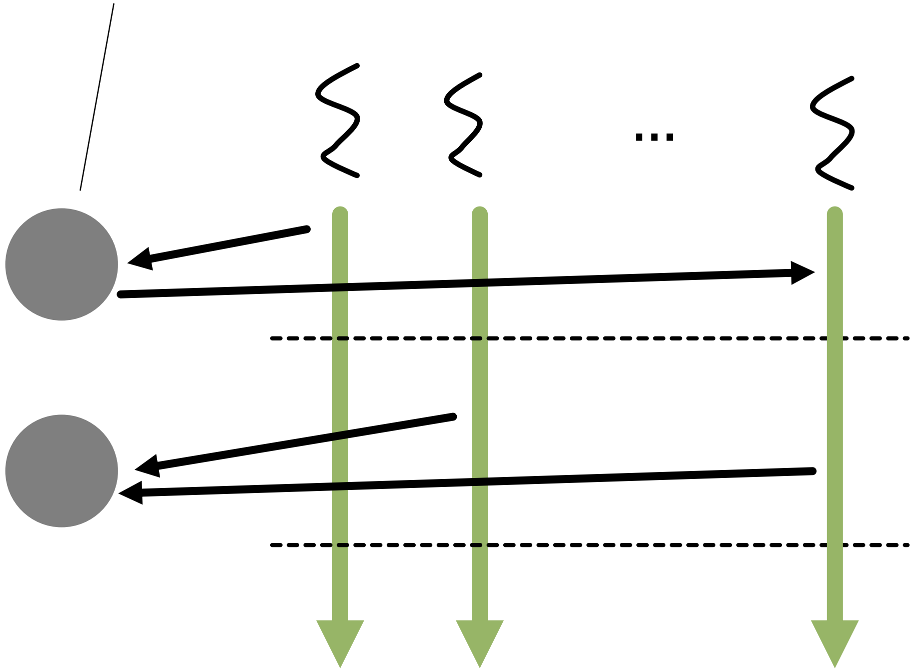
PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

All processes process in lock-steps, communicate by reading from & writing to a shared memory.


Some data in shared memory (e.g., a vertex ☺)

PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

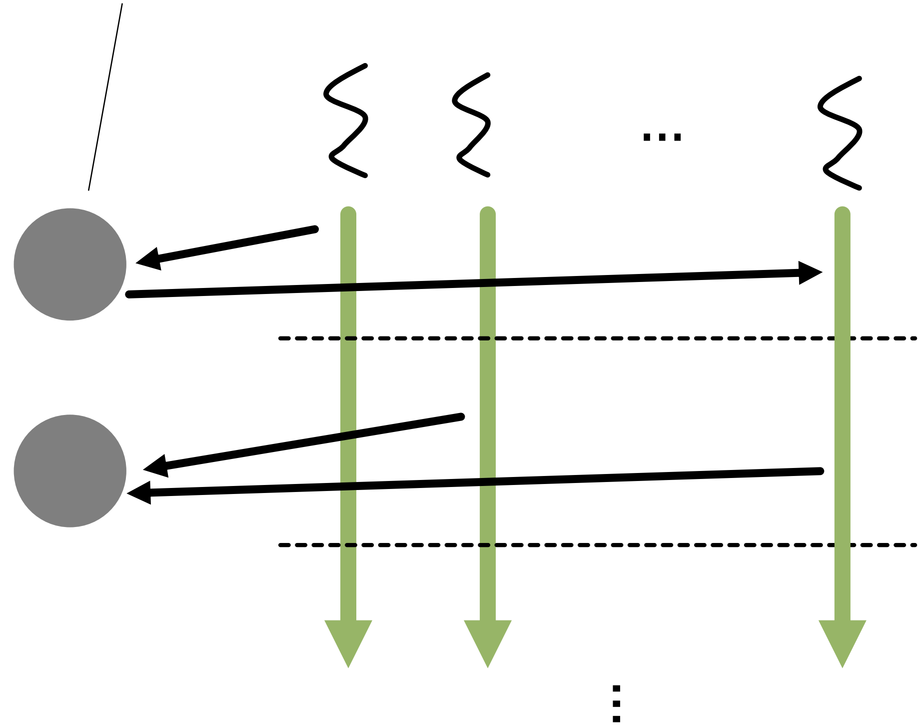
All processes process in lock-steps, communicate by reading from & writing to a shared memory.



Some data in shared memory (e.g., a vertex ☺)

PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

All processes process in lock-steps, communicate by reading from & writing to a shared memory.



Some data in shared memory (e.g., a vertex ☺)

PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

All processes process in lock-steps, communicate by reading from & writing to a shared memory.

PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

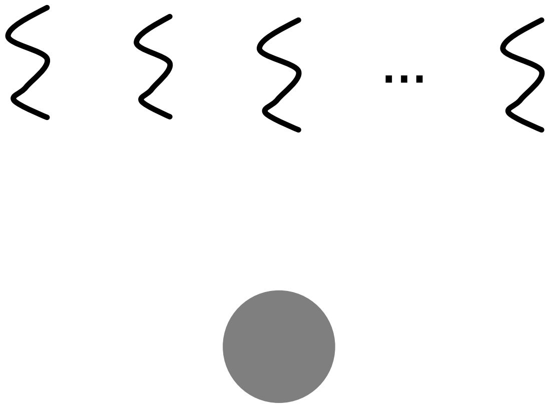
All processes process in lock-steps, communicate by reading from & writing to a shared memory.

CRCW PRAM: concurrent reads and concurrent writes to the same cell take O(1) time.

PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

All processes process in lock-steps, communicate by reading from & writing to a shared memory.

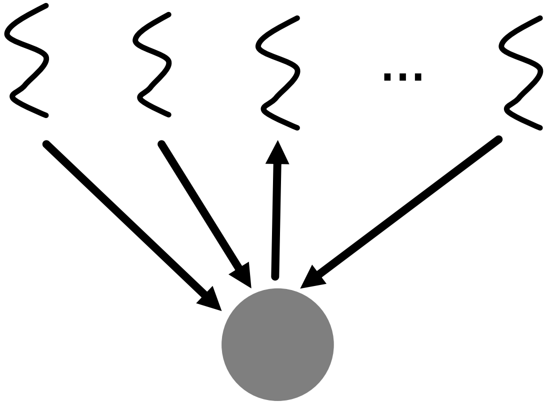
CRCW PRAM: concurrent reads and concurrent writes to the same cell take O(1) time.



PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

All processes process in lock-steps, communicate by reading from & writing to a shared memory.

CRCW PRAM: concurrent reads and concurrent writes to the same cell take O(1) time.

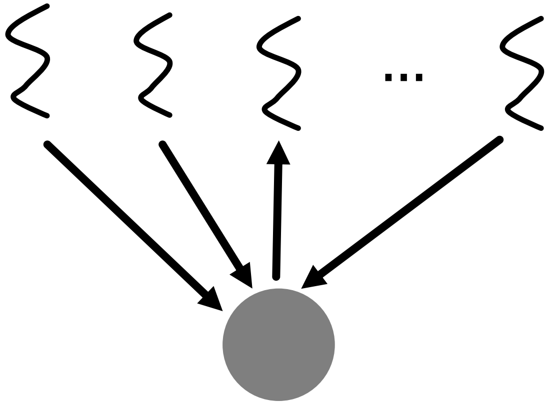


PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

All processes process in lock-steps, communicate by reading from & writing to a shared memory.

CRCW PRAM: concurrent reads and concurrent writes to the same cell take O(1) time.

CREW PRAM: concurrent writes to the same cell are forbidden

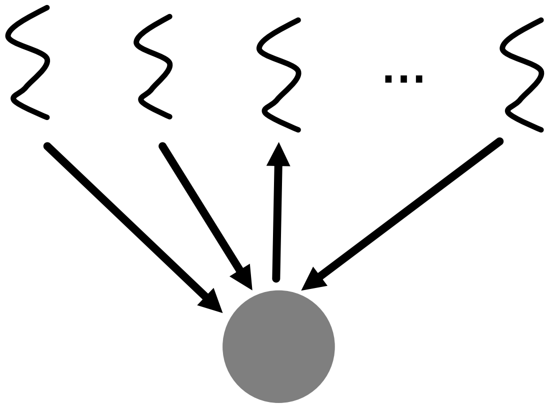


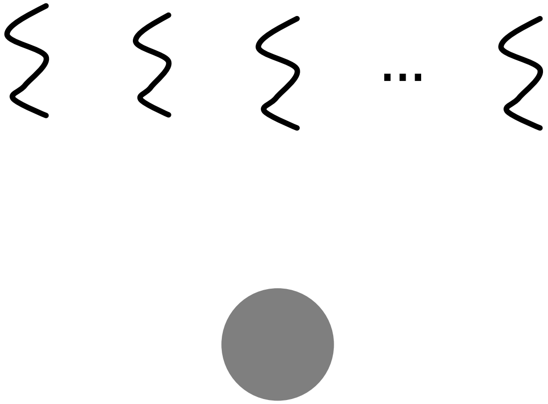
PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

All processes process in lock-steps, communicate by reading from & writing to a shared memory.

CRCW PRAM: concurrent reads and concurrent writes to the same cell take O(1) time.

CREW PRAM: concurrent writes to the same cell are forbidden

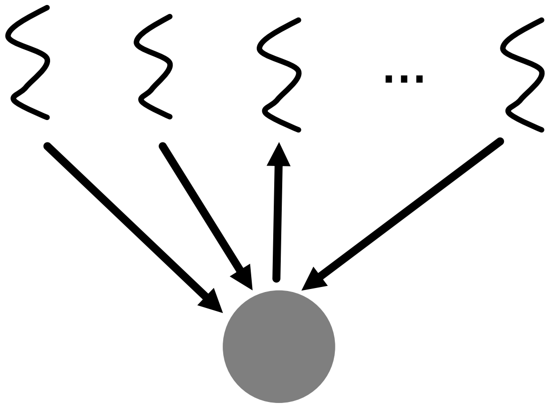




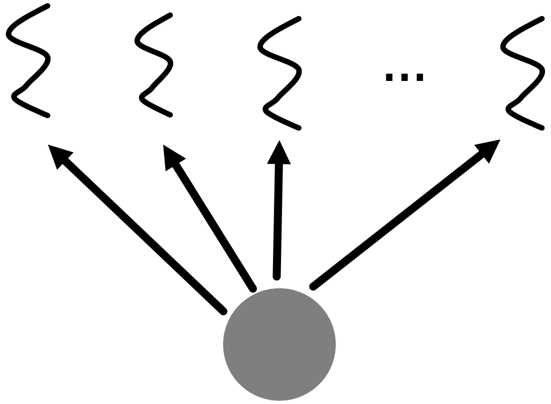
PRAM (Parallel Random Access Machine): a model used to reason about the performance of parallel algorithms

All processes process in lock-steps, communicate by reading from & writing to a shared memory.

CRCW PRAM: concurrent reads and concurrent writes to the same cell take O(1) time.

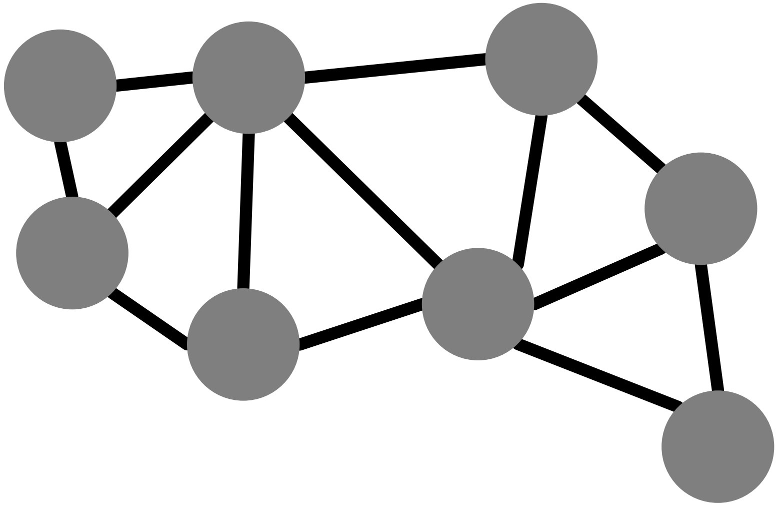


CREW PRAM: concurrent writes to the same cell are forbidden



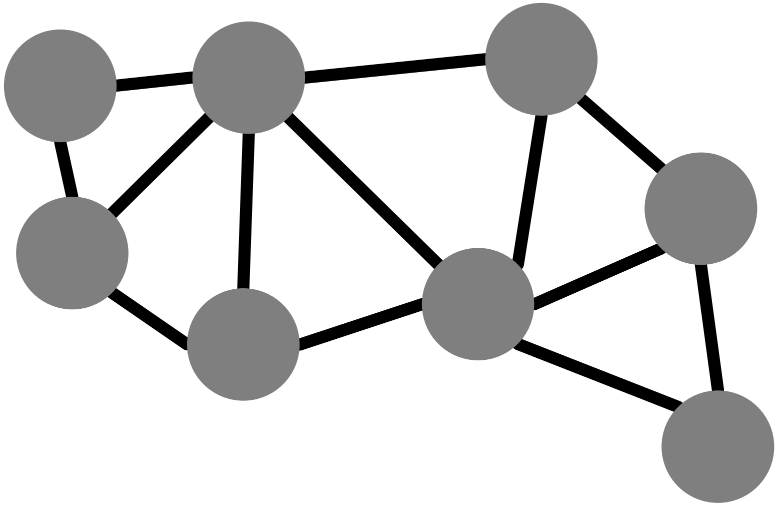
BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER

$k = 4$



BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER

$k = 4$

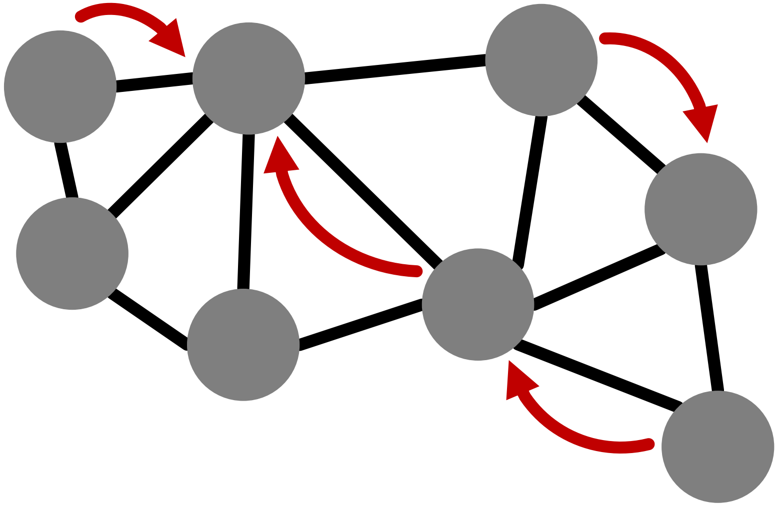


k-RELAXATION

Simultaneous propagation of updates: (pushing) from *k* vertices to one of their neighbors, and (pulling) to *k* vertices from one of their neighbors

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER

$k = 4$

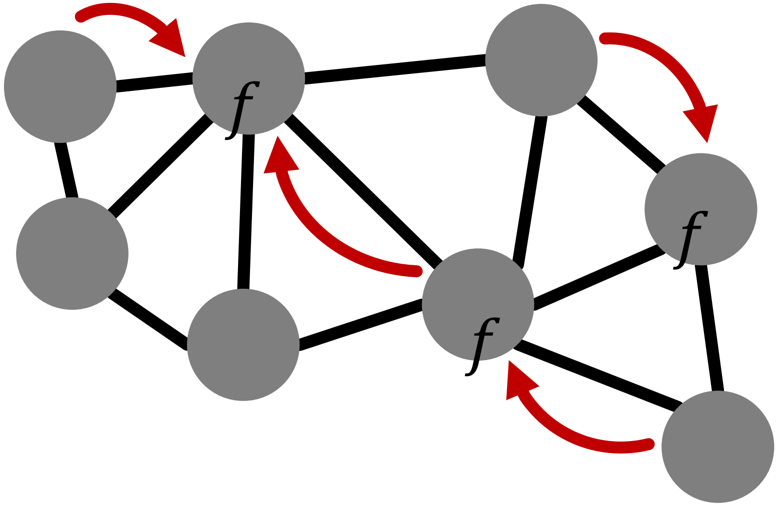


k-RELAXATION

Simultaneous propagation of updates: (pushing) from *k* vertices to one of their neighbors, and (pulling) to *k* vertices from one of their neighbors

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER

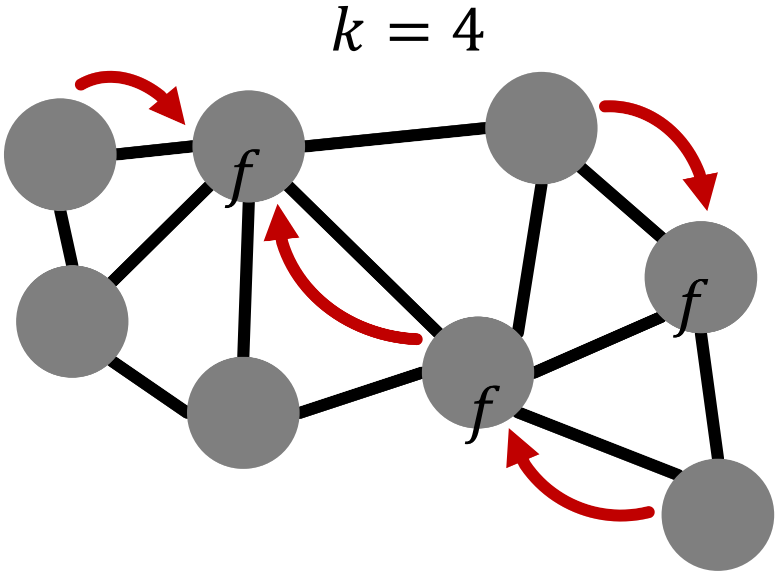
$k = 4$



k-RELAXATION

Simultaneous propagation of updates: (pushing) from *k* vertices to one of their neighbors, and (pulling) to *k* vertices from one of their neighbors

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER

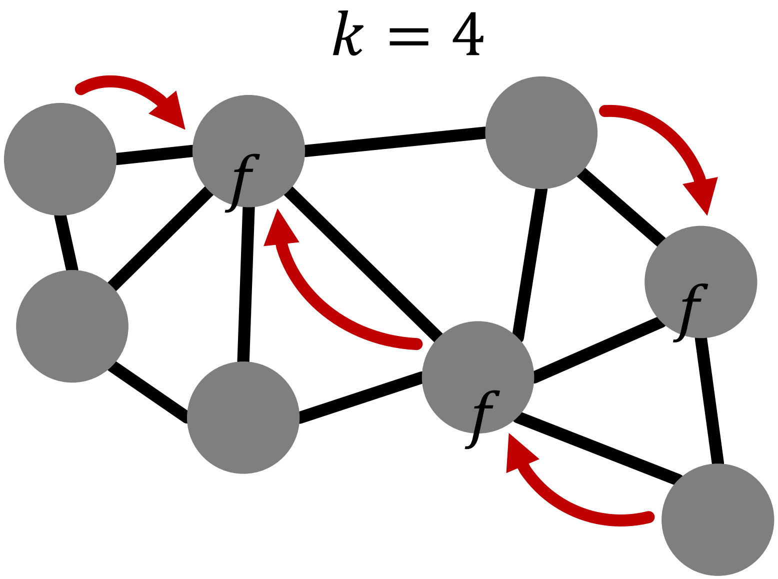


k-RELAXATION

Simultaneous propagation of updates: (pushing) from *k* vertices to one of their neighbors, and (pulling) to *k* vertices from one of their neighbors

Can be thought of a binary tree reduction

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER



k-RELAXATION

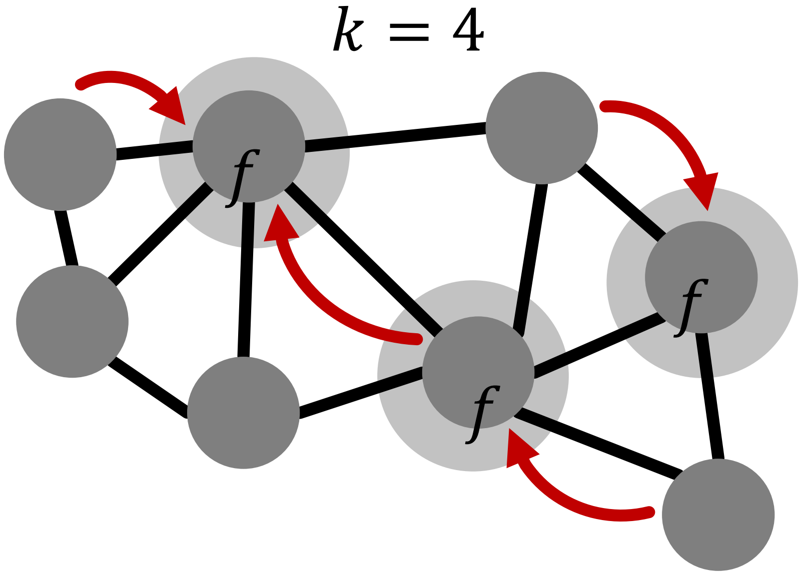
Simultaneous propagation of updates: (pushing) from *k* vertices to one of their neighbors, and (pulling) to *k* vertices from one of their neighbors

Can be thought of a binary tree reduction

k-FILTER

Extract vertices updated in one or more *k*-RELAXATIONs

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER



k-RELAXATION

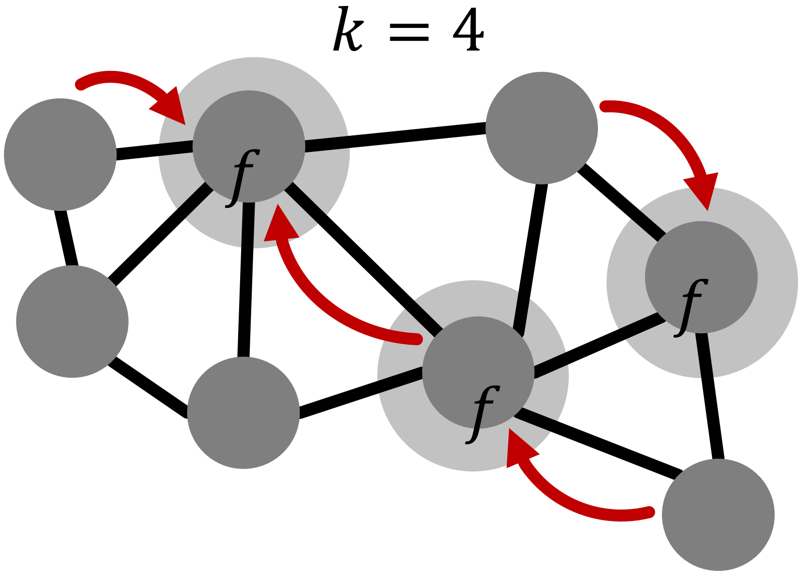
Simultaneous propagation of updates: (pushing) from *k* vertices to one of their neighbors, and (pulling) to *k* vertices from one of their neighbors

Can be thought of a binary tree reduction

k-FILTER

Extract vertices updated in one or more *k*-RELAXATIONs

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER



k-RELAXATION

Simultaneous propagation of updates: (pushing) from *k* vertices to one of their neighbors, and (pulling) to *k* vertices from one of their neighbors

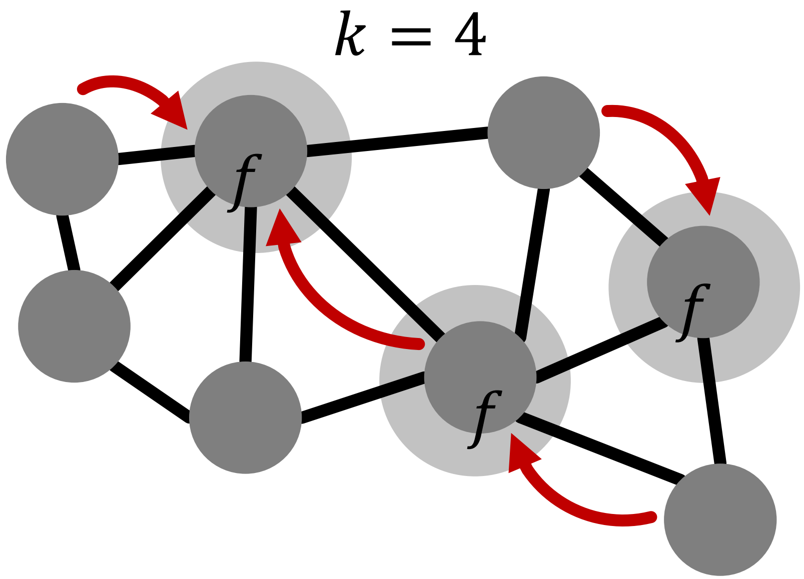
Can be thought of a binary tree reduction

k-FILTER

Extract vertices updated in one or more *k*-RELAXATIONs

Can be thought of a prefix sum

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER



k-RELAXATION

Simultaneous propagation of updates: (pushing) from *k* vertices to one of their neighbors, and (pulling) to *k* vertices from one of their neighbors

Can be thought of a binary tree reduction

k-FILTER

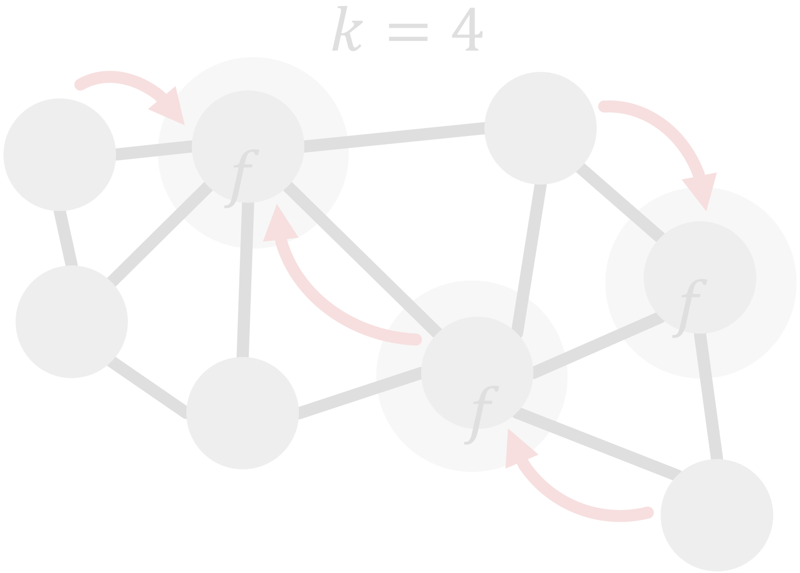
Extract vertices updated in one or more *k*-RELAXATIONs

Can be thought of a prefix sum

!

We can use *k*-RELAXATIONs and *k*-FILTERs to derive all the complexities

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER



k-RELAXATION

Simultaneous propagation of updates: (pushing) from k vertices to one of their neighbors, and (pulling) to k vertices from one of their neighbors

Can be thought of a binary tree reduction

k-FILTER

Extract vertices updated in one or more k-RELAXATIONs

Can be thought of a prefix sum

!

We can use k-RELAXATIONs and k-FILTERs to derive all the complexities

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER

k-RELAXATION

Simultaneous propagation of updates: (pushing) from *k*

We want complexities for (the Cartesian product of):

Can be thought of a prefix sum

FILTERs to derive all the complexities

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER

$k = 4$

k-RELAXATION

Simultaneous propagation of updates: (pushing) from *k*

We want complexities for (the Cartesian product of):

- Time
- work

FILTERs to derive all the complexities

Can be thought of a prefix sum

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER

$k = 4$

k-RELAXATION

Simultaneous propagation of updates: (pushing) from *k*

We want complexities for (the Cartesian product of):

- Time
- work

X

FILTERs to derive all the complexities

Can be thought of a prefix sum

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER


$k = 4$

k-RELAXATION

Simultaneous propagation of updates: (pushing) from *k*

We want complexities for (the Cartesian product of):

- Time
- work

X

- Pushing
- Pulling

FILTERs to derive all the complexities

Can be thought of a prefix sum

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER

k-RELAXATION

Simultaneous propagation of updates: (pushing) from *k*

We want complexities for (the Cartesian product of):

- Time
- work

X

- Pushing
- Pulling

X

Can be thought of a prefix sum

FILTERs to derive all the complexities

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER

k-RELAXATION

Simultaneous propagation of updates: (pushing) from *k*

$k = 4$

We want complexities for (the Cartesian product of):

- Time
- work

X

- Pushing
- Pulling

X

- CRCW PRAM
- CREW PRAM

FILTERs to derive all the complexities

Can be thought of a prefix sum

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER

$k = 4$

k-RELAXATION

Simultaneous propagation of updates: (pushing) from *k*

We want complexities for (the Cartesian product of):

- Time
- work

X

- Pushing
- Pulling

X

- CRCW PRAM
- CREW PRAM

X

Can be thought of a prefix sum

FILTERs to derive all the complexities

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER

k-RELAXATION

Simultaneous propagation of updates: (pushing) from *k*

We want complexities for (the Cartesian product of):

- Time
- work

X

- Pushing
- Pulling

X

- CRCW PRAM
- CREW PRAM

X

- BFS
- PageRank
- Triangle Counting
- Betweenness Centrality
- Graph Coloring
- Δ-Stepping
- MST Boruvka

FILTERs to derive all the complexities

Can be thought of a prefix sum

BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER

$k = 4$

k-RELAXATION

Simultaneous propagation of updates: (pushing) from *k*

We want complexities for (the Cartesian product of):

- Time
- work

X

- Pushing
- Pulling

X

- CRCW PRAM
- CREW PRAM

X

- BFS
- PageRank
- Triangle Counting
- Betweenness Centrality
- Graph Coloring
- Δ-Stepping
- MST Boruvka

+ some others ☺

FILTERs to derive all the complexities

Can be thought of a prefix sum

COMPLEXITY ANALYSES

		PageRank	Triangle Counting	BFS
Pulling	Time	$O(L(m/P+\hat{d}))$	$O(\hat{d}m/P+\hat{d}^2)$	$O(Dm/P+D\hat{d})$
	Work	$O(Lm)$	$O(m\hat{d})$	$O(Dm)$
Pushing	Time (CRCW)	$O\left(L(m/P+\hat{d})\right)$	$O(\hat{d}m/P+\hat{d}^2)$	$O(Dm/P+D\hat{d}+D\log P)$
	Work (CRCW)	$O(Lm)$	$O(m\hat{d})$	$O(m)$
	Time (CREW)	$O\left(L\log(\hat{d})\,(m/P+\hat{d})\right)$	$O\left(\log\hat{d}\,(\hat{d}m/P+\hat{d}^2)\right)$	$O\left(\log\hat{d}\,(Dm/P+D\hat{d})\right)$
	Work (CREW)	$O(Lm\log\hat{d})$	$O(m\hat{d}\log\hat{d})$	$O(m\log\hat{d})$

		Δ-Stepping	Boman Graph Coloring	MST	BC
Pulling	Time	$O\left((L/\Delta)l_\Delta(m/P+\hat{d})\right)$	$O(Lm/P+L\hat{d})$	$O(n^2/P)$	Derived straightforwardly from BFS
	Work	$O((L/\Delta)ml_\Delta)$	$O(Lm)$	$O(n^2)$	
Pushing	Time (CRCW)	$O\left((L/\Delta)l_\Delta\hat{d}+ml_\Delta/P\right)$	$O\left(\log\hat{d}\,(Lm/P+L\hat{d})\right)$	$O(n^2/P)$	
	Work (CRCW)	$O(ml_\Delta)$	$O(Lm)$	$O(n^2)$	
	Time (CREW)	$O\left(\log(\hat{d})\left((L/\Delta)l_\Delta\hat{d}+ml_\Delta/P\right)\right)$	$O\left(\log\hat{d}\,(Lm/P+L\hat{d})\right)$	$O(\log(n)\,n^2/P)$	
	Work (CREW)	$O(\log(\hat{d})\,ml_\Delta)$	$O(Lm\log\hat{d})$	$O(\log(n)\,n^2)$	

COMPLEXITY ANALYSES

No worries, we won’t go over all these details here ☺

		PageRank	Triangle Co	
Pulling	Time	$O(L(m/P + \hat{d}))$	$O(\hat{d}m/P + \hat{d}^2)$	$O(Dm/P + D\hat{d})$
	Work	$O(Lm)$	$O(m\hat{d})$	$O(Dm)$
Pushing	Time (CRCW)	$O\left(L(m/P + \hat{d})\right)$	$O(\hat{d}m/P + \hat{d}^2)$	$O(Dm/P + D\hat{d} + D\log P)$
	Work (CRCW)	$O(Lm)$	$O(m\hat{d})$	$O(m)$
	Time (CREW)	$O\left(L\log(\hat{d})\,(m/P + \hat{d})\right)$	$O\left(\log\hat{d}\,(\hat{d}m/P + \hat{d}^2)\right)$	$O\left(\log\hat{d}\,(Dm/P + D\hat{d})\right)$
	Work (CREW)	$O(Lm\log\hat{d})$	$O(m\widehat{d}\,\log\hat{d})$	$O(m\log\hat{d})$

		Δ-Stepping	Boman Graph Coloring	MST	BC
Pulling	Time	$O\left((L/\Delta)l_\Delta(m/P + \hat{d})\right)$	$O(Lm/P + L\hat{d})$	$O(n^2/P)$	Derived straightforwardly from BFS
	Work	$O((L/\Delta)ml_\Delta)$	$O(Lm)$	$O(n^2)$	
Pushing	Time (CRCW)	$O\left((L/\Delta)l_\Delta\hat{d} + ml_\Delta/P\right)$	$O\left(\log\hat{d}\,(Lm/P + L\hat{d})\right)$	$O(n^2/P)$	
	Work (CRCW)	$O(ml_\Delta)$	$O(Lm)$	$O(n^2)$	
	Time (CREW)	$O\left(\log(\hat{d})\left((L/\Delta)l_\Delta\hat{d} + ml_\Delta/P\right)\right)$	$O\left(\log\hat{d}\,(Lm/P + L\hat{d})\right)$	$O(\log(n)\,n^2/P)$	
	Work (CREW)	$O(\log(\hat{d})\,ml_\Delta)$	$O(Lm\log\hat{d})$	$O(\log(n)\,n^2)$	

COMPLEXITY ANALYSES

Let’s only see the PageRank comparisons (others are similar)

No worries, we won’t go over all these details here ☺

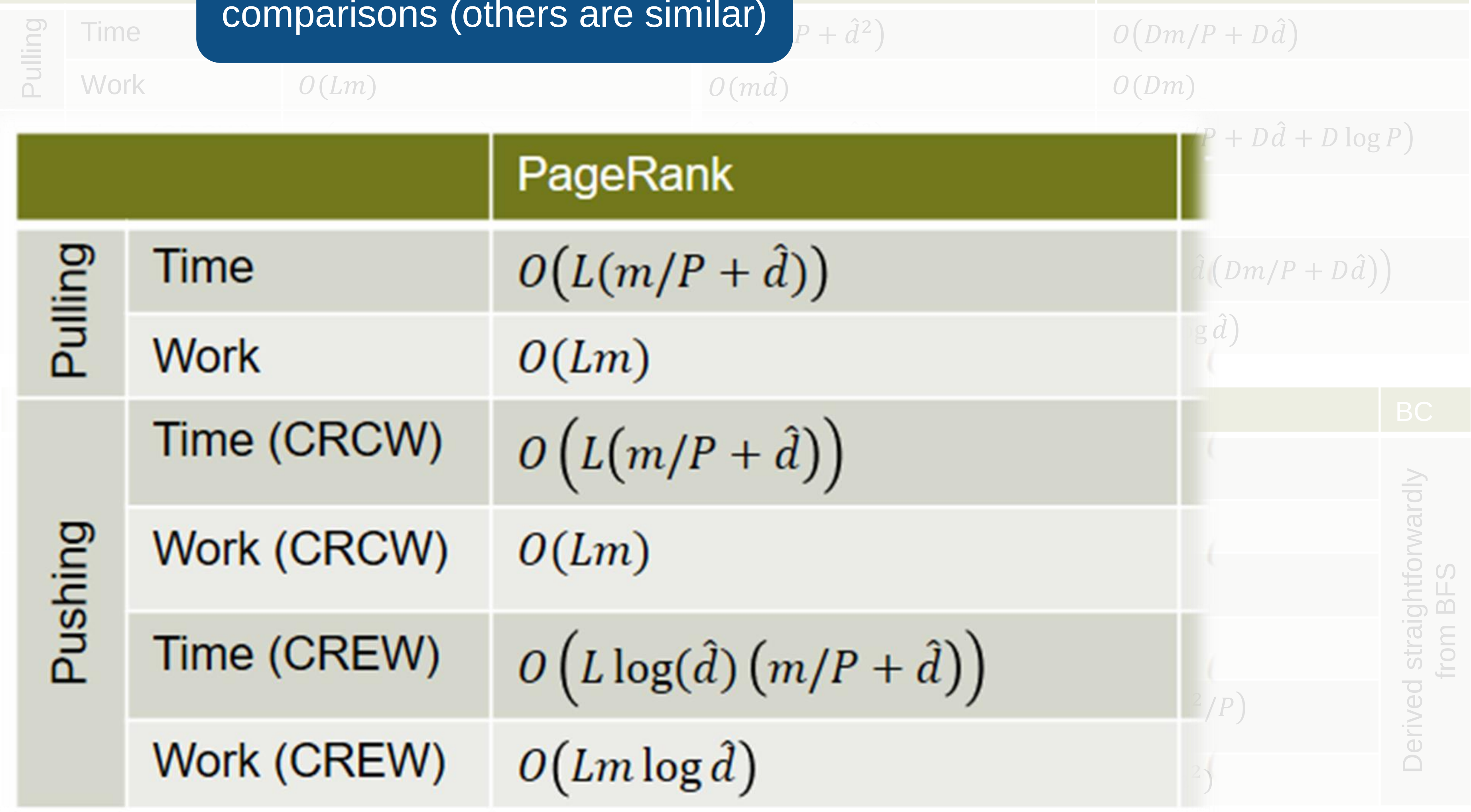
			e Co	
Pulling	Time		$P + \hat{d}^2)$	$O(Dm/P + D\hat{d})$
	Work	$O(Lm)$	$O(m\hat{d})$	$O(Dm)$
Pushing	Time (CRCW)	$O\left(L(m/P + \hat{d})\right)$	$O(\hat{d}m/P + \hat{d}^2)$	$O(Dm/P + D\hat{d} + D\log P)$
	Work (CRCW)	$O(Lm)$	$O(m\hat{d})$	$O(m)$
	Time (CREW)	$O\left(L\log(\hat{d})\,(m/P + \hat{d})\right)$	$O\left(\log\hat{d}\,(\hat{d}m/P + \hat{d}^2)\right)$	$O\left(\log\hat{d}\,(Dm/P + D\hat{d})\right)$
	Work (CREW)	$O(Lm\log\hat{d})$	$O(m\hat{d}\,\log\hat{d})$	$O(m\log\hat{d})$

		Δ-Stepping	Boman Graph Coloring	MST	BC
Pulling	Time	$O\left((L/\Delta)l_\Delta(m/P + \hat{d})\right)$	$(Lm/P + L\hat{d})$	$O(n^2/P)$	Derived straightforwardly from BFS
	Work	$O((L/\Delta)ml_\Delta)$	(Lm)	$O(n^2)$	
Pushing	Time (CRCW)	$O\left((L/\Delta)l_\Delta\hat{d} + ml_\Delta/P\right)$	$\left(\log\hat{d}\,(Lm/P + L\hat{d})\right)$	$O(n^2/P)$	
	Work (CRCW)	$O(ml_\Delta)$	(Lm)	$O(n^2)$	
	Time (CREW)	$O\left(\log(\hat{d})\left((L/\Delta)l_\Delta\hat{d} + ml_\Delta/P\right)\right)$	$\left(\log\hat{d}\,(Lm/P + L\hat{d})\right)$	$O(\log(n)\,n^2/P)$	
	Work (CREW)	$O(\log(\hat{d})\,ml_\Delta)$	$(Lm\log\hat{d})$	$O(\log(n)\,n^2)$	

COMPLEXITY ANALYSES

Let’s only see the PageRank comparisons (others are similar)

No worries, we won’t go over all these details here ☺

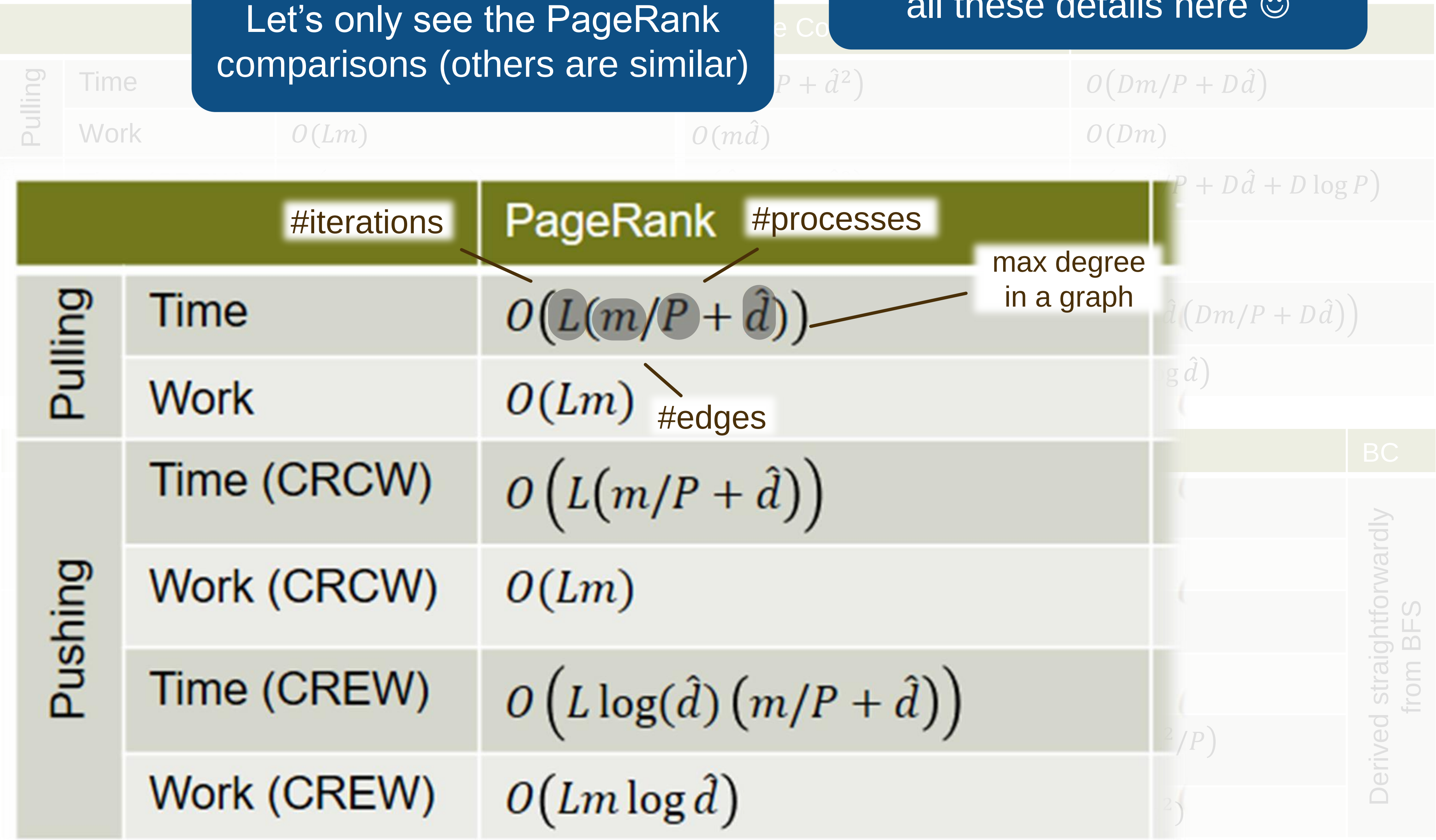


		PageRank
Pulling	Time	$O(L(m/P + \hat{d}))$
	Work	$O(Lm)$
Pushing	Time (CRCW)	$O\left(L(m/P + \hat{d})\right)$
	Work (CRCW)	$O(Lm)$
	Time (CREW)	$O\left(L\log(\hat{d})\,(m/P + \hat{d})\right)$
	Work (CREW)	$O(Lm\log\hat{d})$

COMPLEXITY ANALYSES

No worries, we won't go over all these details here ☺

Let's only see the PageRank comparisons (others are similar)



		PageRank
Pulling	Time	$O(L(m/P + \hat{d}))$
	Work	$O(Lm)$
Pushing	Time (CRCW)	$O\left(L(m/P + \hat{d})\right)$
	Work (CRCW)	$O(Lm)$
	Time (CREW)	$O\left(L \log(\hat{d})\,(m/P + \hat{d})\right)$
	Work (CREW)	$O(Lm \log \hat{d})$

#iterations

#processes

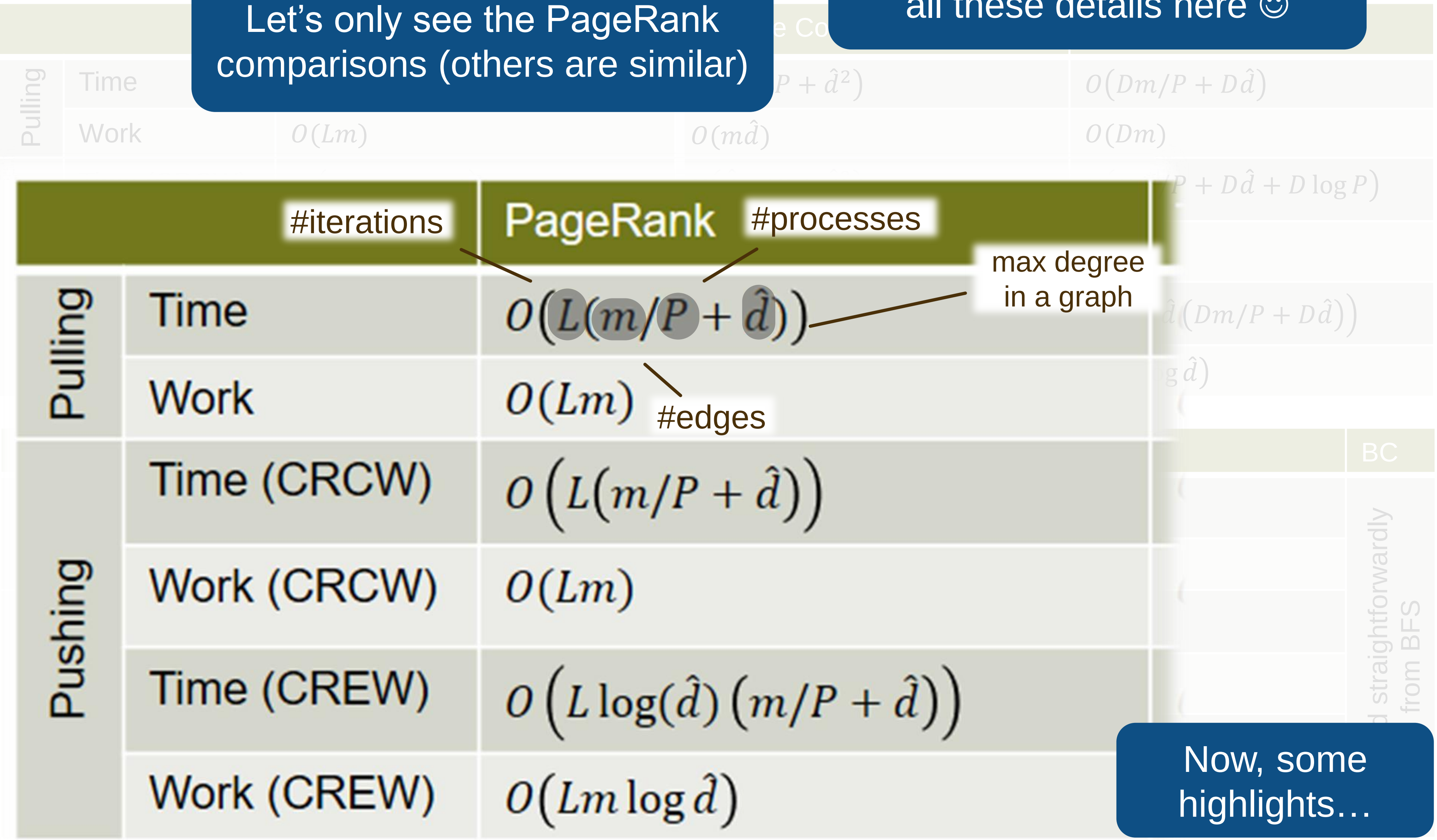
max degree in a graph

#edges

COMPLEXITY ANALYSES

Let's only see the PageRank comparisons (others are similar)

No worries, we won't go over all these details here ☺



		PageRank
Pulling	Time	$O(L(m/P+\hat{d}))$
	Work	$O(Lm)$
Pushing	Time (CRCW)	$O\left(L(m/P+\hat{d})\right)$
	Work (CRCW)	$O(Lm)$
	Time (CREW)	$O\left(L\log(\hat{d})\,(m/P+\hat{d})\right)$
	Work (CREW)	$O(Lm\log\hat{d})$

Now, some highlights…

COMPLEXITY ANALYSES

HIGHLIGHTS

COMPLEXITY ANALYSES

HIGHLIGHTS

Write conflicts Ⓦ

❗

Pushing entails more write conflicts (must be resolved with locks or atomics.

COMPLEXITY ANALYSES

HIGHLIGHTS

Write conflicts Ⓦ

Pushing entails more write conflicts (must be resolved with locks or atomics.

Atomics/Locks

Pulling removes atomics or locks completely (TC, PR, BFS, Δ-Stepping, MST) or it changes the type of conflicts from `f` to `i` (BC).

COMPLEXITY ANALYSES

HIGHLIGHTS

Write conflicts Ⓦ

!

Pushing entails more write conflicts (must be resolved with locks or atomics.

Memory accesses

!

Pulling in traversals (BFS, BC, SSSP-Δ) entails more time and work.

Atomics/Locks

!

Pulling removes atomics or locks completely (TC, PR, BFS, Δ-Stepping, MST) or it changes the type of conflicts from `f` to `i` (BC).

PUSHING VS. PULLING
RESEARCH QUESTIONS

Check the paper ☺

Yes (developed 7 algorithms and the total of 11 variants)

Can be described with the actual dichotomy

How do they differ in complexity?

What is performance?

PUSHING VS. PULLING
RESEARCH QUESTIONS

Yes (developed 7 algorithms and the total of 11 variants)

Answered ☺

Check the paper ☺

Can be described with the actual dichotomy

What is performance?

PUSHING VS. PULLING
RESEARCH QUESTIONS

Yes (developed 7 algorithms and the total of 11 variants)

Check the paper ☺

Can be described with the actual dichotomy

Answered ☺

What is performance?

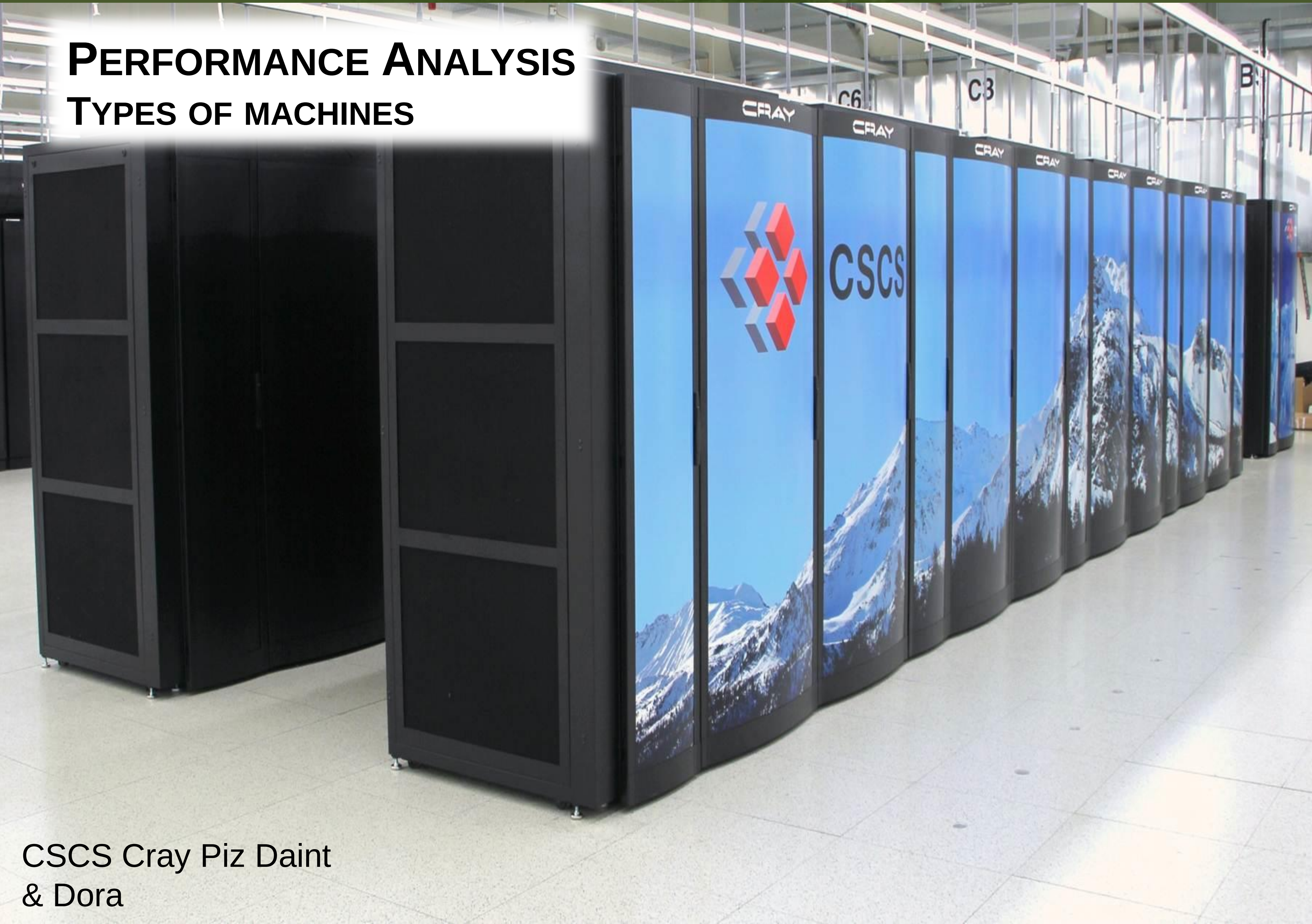
How effective are the incorporated strategies?

Is pushing or pulling faster? When and why?

What is the impact of the programming model? environment?

PERFORMANCE ANALYSIS

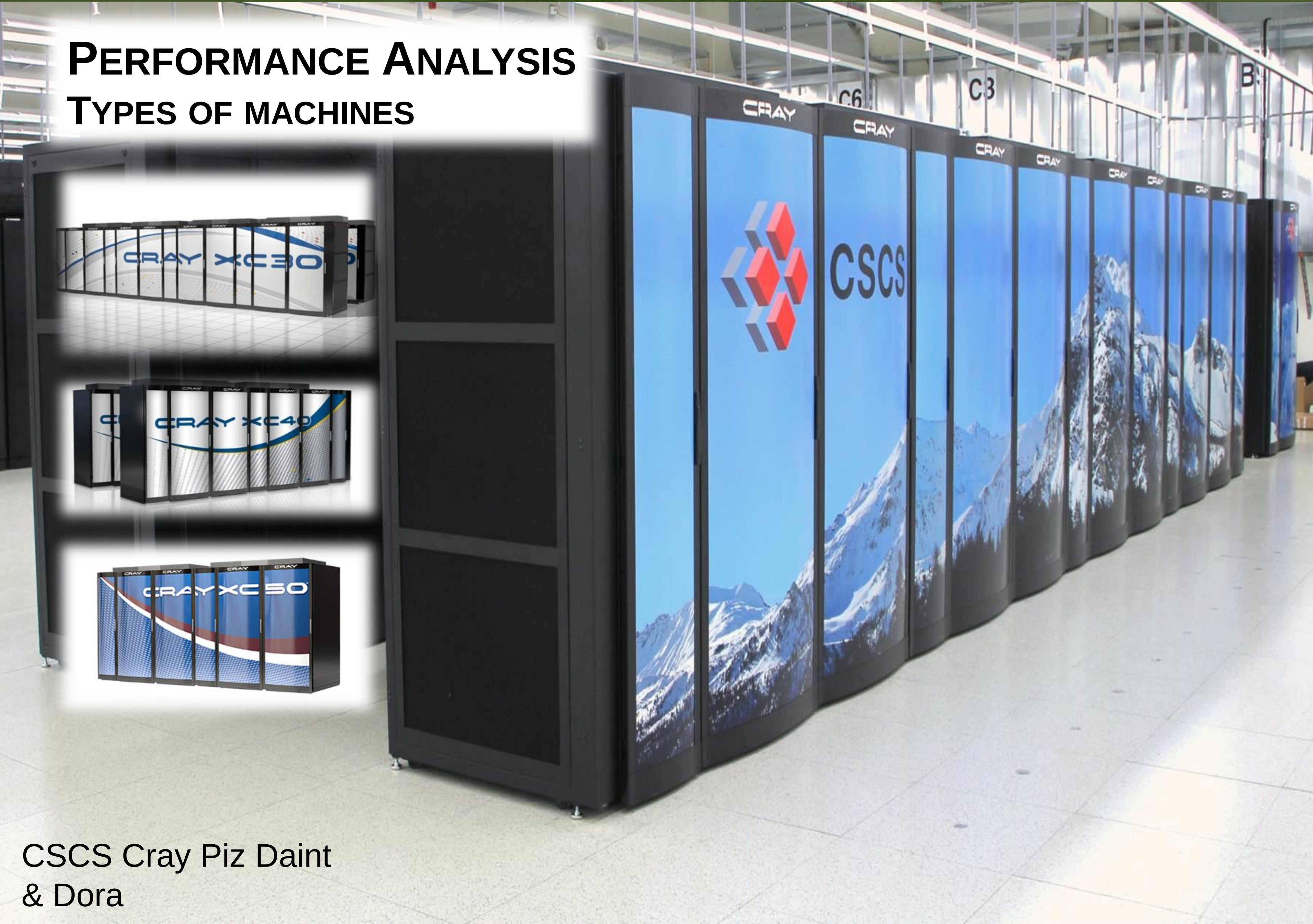
TYPES OF MACHINES



CSCS Cray Piz Daint
& Dora

PERFORMANCE ANALYSIS

TYPES OF MACHINES



CSCS Cray Piz Daint
& Dora

PERFORMANCE ANALYSIS

TYPES OF MACHINES

CSCS Cray Piz Daint
& Dora

Trivium Intel Server

PERFORMANCE ANALYSIS

TYPES OF GRAPHS

PERFORMANCE ANALYSIS

TYPES OF GRAPHS

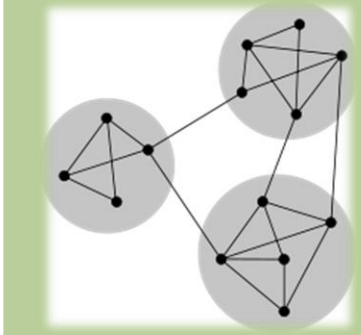
Synthetic graphs

PERFORMANCE ANALYSIS

TYPES OF GRAPHS

Synthetic graphs

Kronecker [1]



---

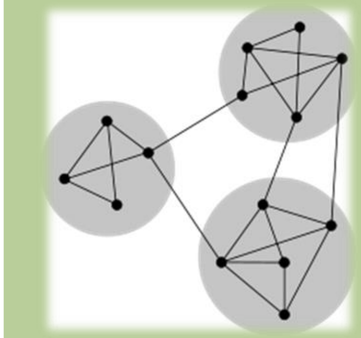
[1] J. Leskovec et al. Kronecker Graphs: An Approach to Modeling Networks. J. Mach. Learn. Research. 2010.

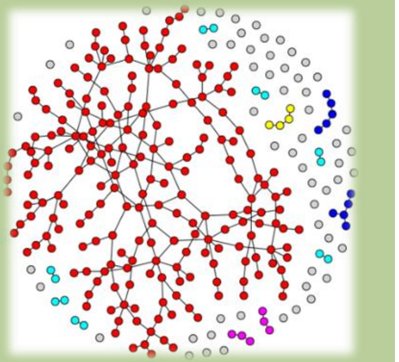
PERFORMANCE ANALYSIS

TYPES OF GRAPHS

Synthetic graphs

Kronecker [1]





Erdös-Rényi [2]

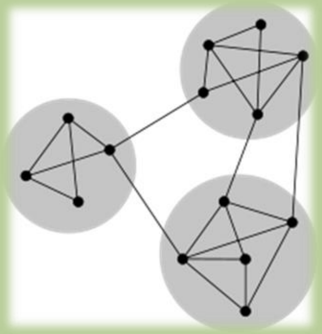
[1] J. Leskovec et al. Kronecker Graphs: An Approach to Modeling Networks. J. Mach. Learn. Research. 2010.
[2] P. Erdos and A. Renyi. On the evolution of random graphs. Pub. Math. Inst. Hun. A. Science. 1960.

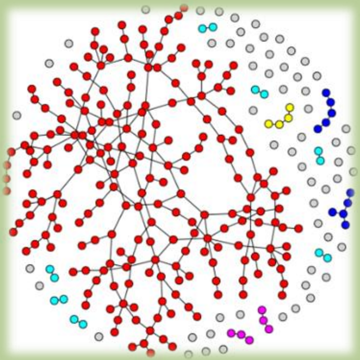
PERFORMANCE ANALYSIS

TYPES OF GRAPHS

Synthetic graphs

Kronecker [1]





Erdös-Rényi [2]

Real-world SNAP graphs [3]

---

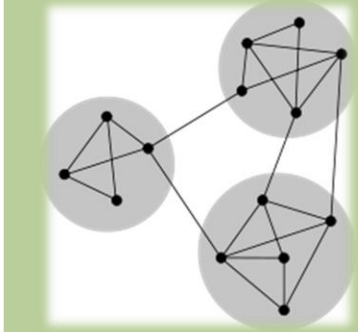
[1] J. Leskovec et al. Kronecker Graphs: An Approach to Modeling Networks. J. Mach. Learn. Research. 2010.
[2] P. Erdos and A. Renyi. On the evolution of random graphs. Pub. Math. Inst. Hun. A. Science. 1960.
[3] https://snap.stanford.edu

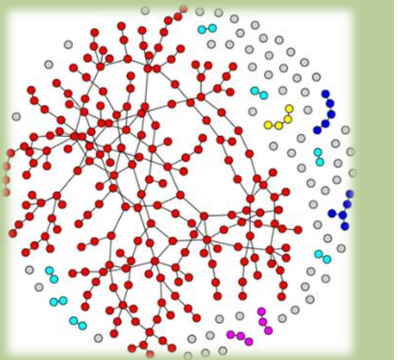
PERFORMANCE ANALYSIS

TYPES OF GRAPHS

Synthetic graphs

Kronecker [1]





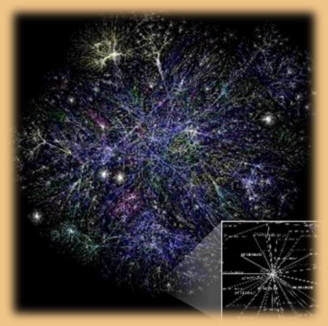
Erdös-Rényi [2]

Real-world SNAP graphs [3]

Road networks

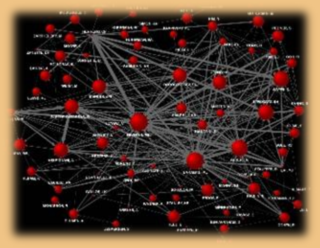
Social networks

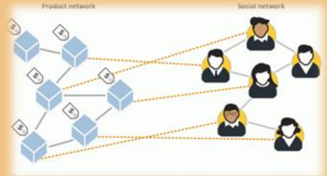
Comm. graphs



Web graphs

Citation graphs





Purchase networks

[1] J. Leskovec et al. Kronecker Graphs: An Approach to Modeling Networks. J. Mach. Learn. Research. 2010.
[2] P. Erdos and A. Renyi. On the evolution of random graphs. Pub. Math. Inst. Hun. A. Science. 1960.
[3] https://snap.stanford.edu

PERFORMANCE ANALYSIS

COUNTED EVENTS

PERFORMANCE ANALYSIS

COUNTED EVENTS

Counted PAPI events

Cache misses (L1, L2, L3)
Reads, writes
Branches (conditional, unconditional)
TLB misses (data, instruction)

PERFORMANCE ANALYSIS

COUNTED EVENTS

Counted PAPI events

Cache misses (L1, L2, L3)
Reads, writes
Branches (conditional, unconditional)
TLB misses (data, instruction)

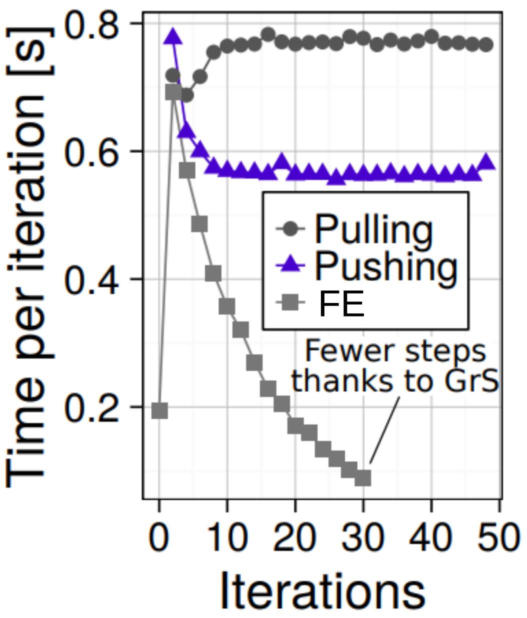
Other counted events

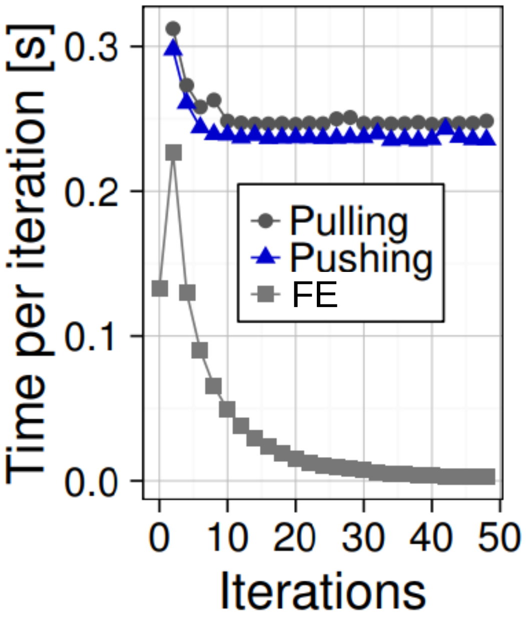
Issued atomics
Acquired locks
Messages (sent, received)
RMA accesses (reads, writes, atomics)

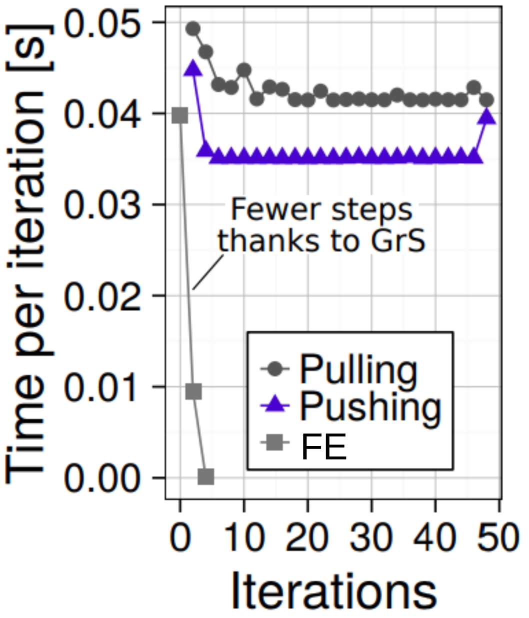
PERFORMANCE ANALYSIS
BOMAN GRAPH COLORING

SNAP
orc, ljn: social networks
rca: road network

Shared-Memory



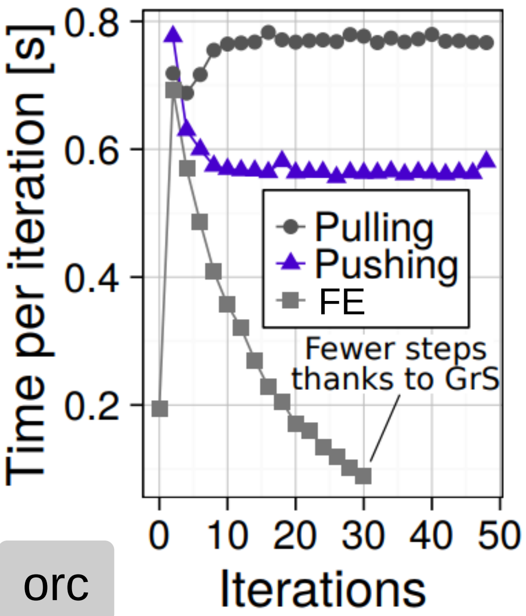




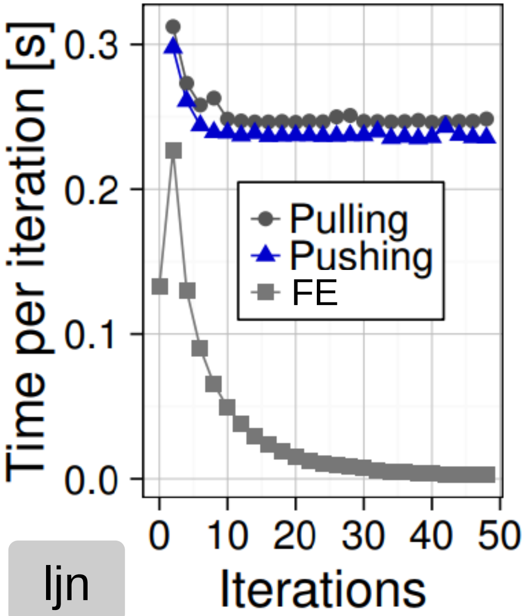
PERFORMANCE ANALYSIS
BOMAN GRAPH COLORING

SNAP
orc, ljn: social networks
rca: road network

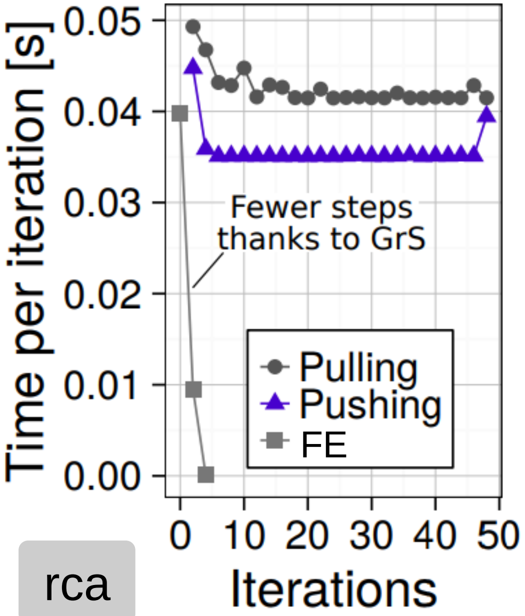
Shared-Memory



orc



ljn



rca

PERFORMANCE ANALYSIS
BOMAN GRAPH COLORING

orc, ljn: social networks
rca: road network

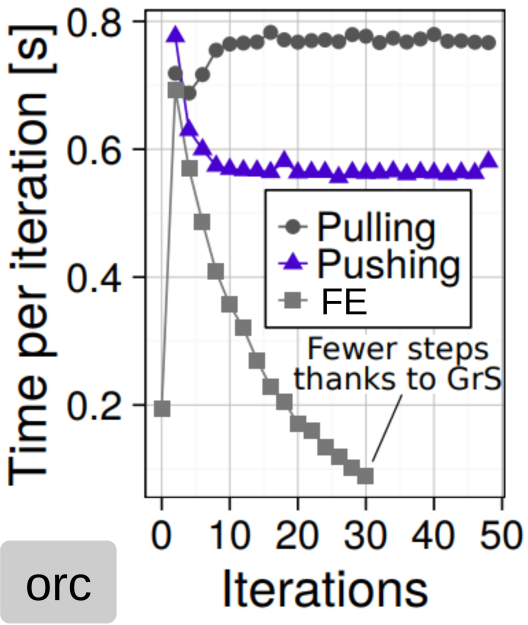
Shared-Memory

!

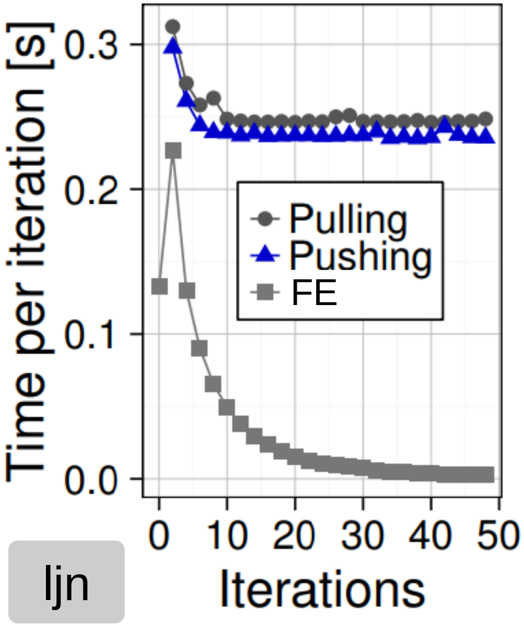
Pushing faster

Fewer cache/TLB misses

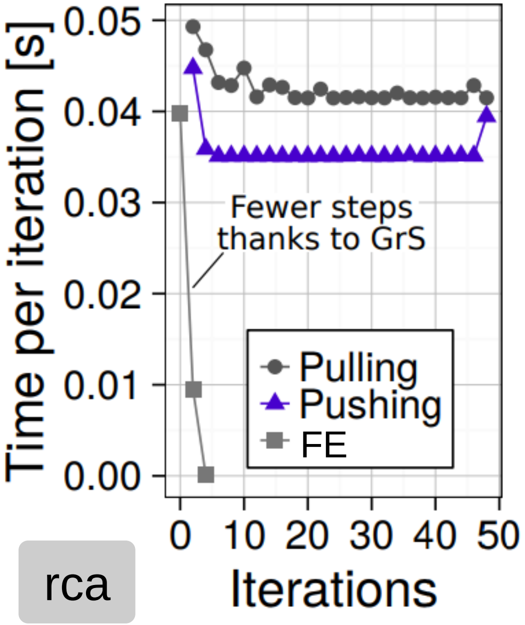
Fewer reads/writes



orc



ljn



rca

PERFORMANCE ANALYSIS
BOMAN GRAPH COLORING

SNAP

orc, ljn: social networks

rca: road network

Shared-Memory

Pushing faster

Fewer cache/TLB misses

Fewer reads/writes

orc

ljn

rca

PERFORMANCE ANALYSIS
BOMAN GRAPH COLORING

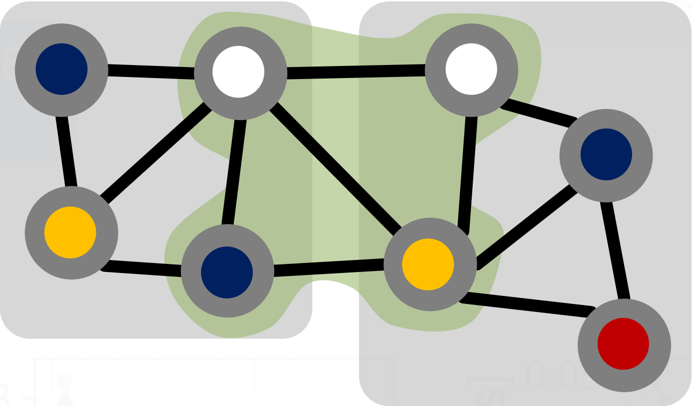
SNAP
orc, ljn: social networks
rca: road network

Shared-Memory

!
Pushing faster

Fewer cache/TL
misses

Fewer reads/writes



orc

ljn

rca

PERFORMANCE ANALYSIS
BOMAN GRAPH COLORING

SNAP

orc, ljn: social networks

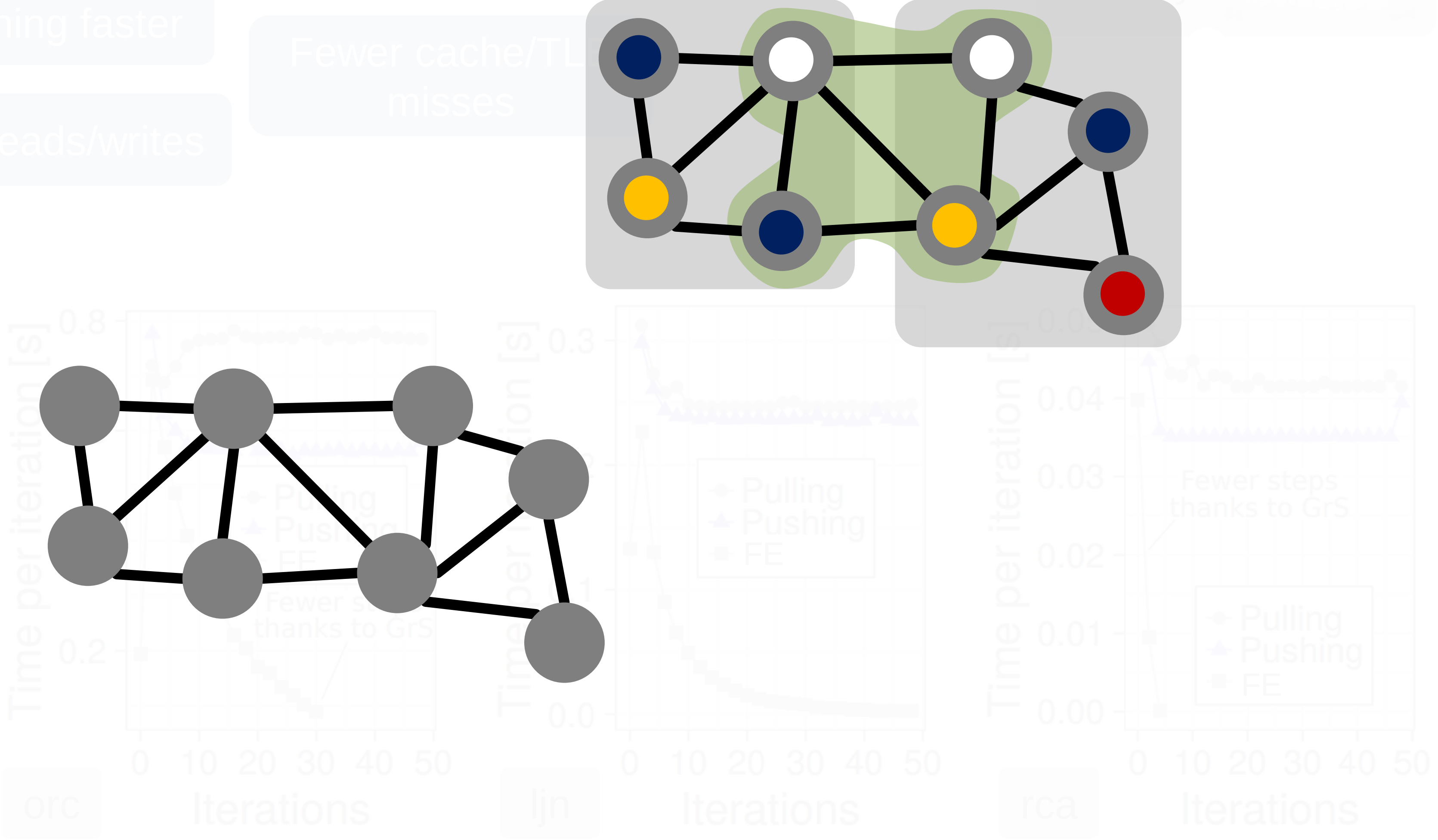
rca: road network

Shared-Memory

Pushing faster

Fewer reads/writes

Fewer cache/TLB misses



PERFORMANCE ANALYSIS
BOMAN GRAPH COLORING

SNAP
orc, ljn: social networks
rca: road network

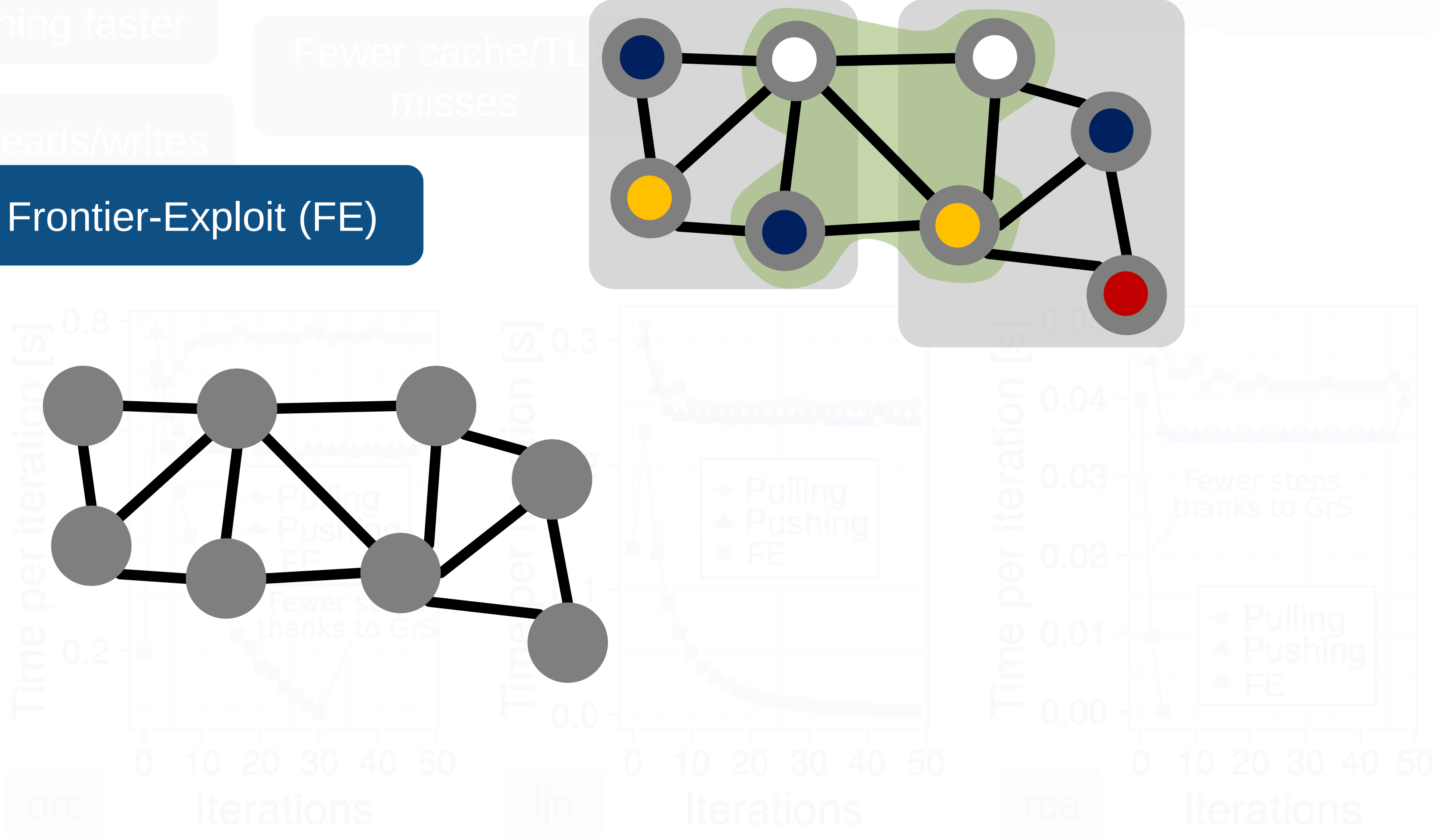
Shared-Memory

Pushing faster

Fewer cache/TLB misses

Fewer reads/writes

Frontier-Exploit (FE)



PERFORMANCE ANALYSIS
BOMAN GRAPH COLORING

SNAP

orc, ljn: social networks
rca: road network

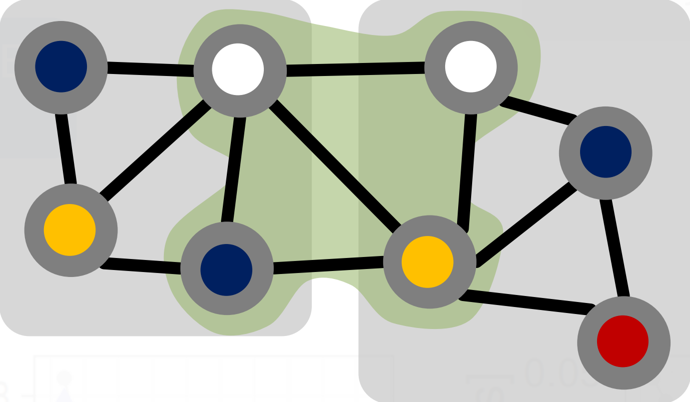
Shared-Memory

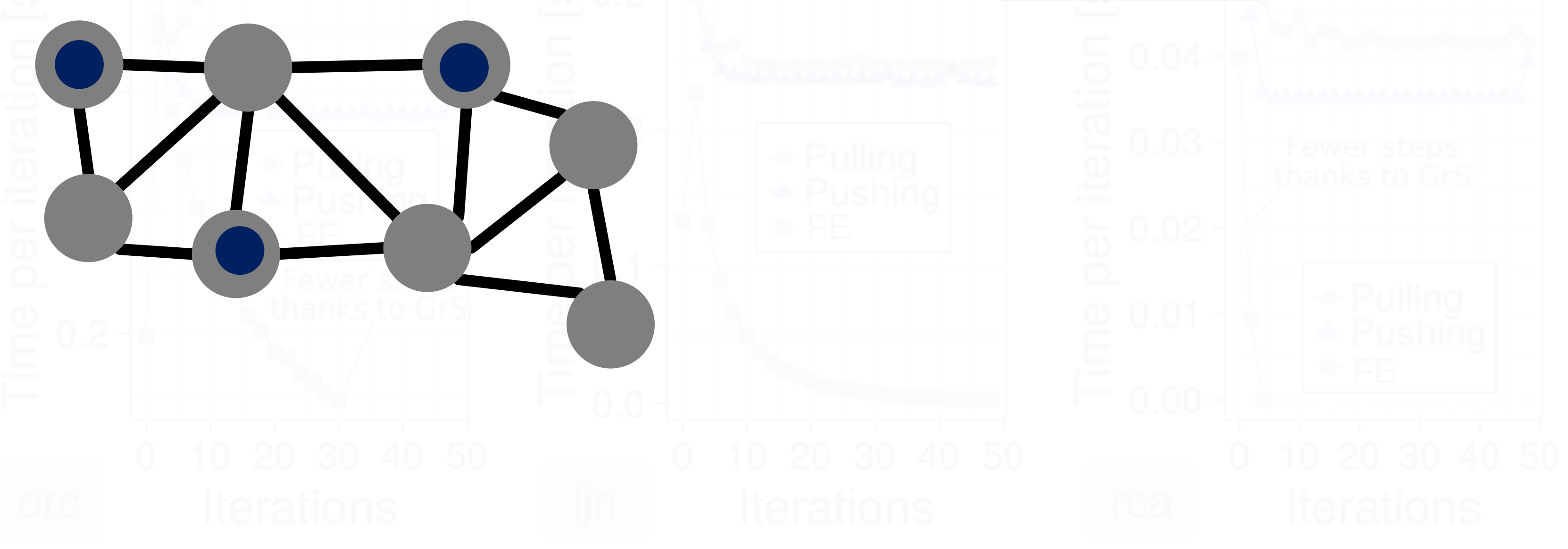
Pushing faster

Fewer cache/TLB misses

Fewer reads/writes

Frontier-Exploit (FE)





PERFORMANCE ANALYSIS
BOMAN GRAPH COLORING

orc, ljn: social networks
rca: road network
SNAP

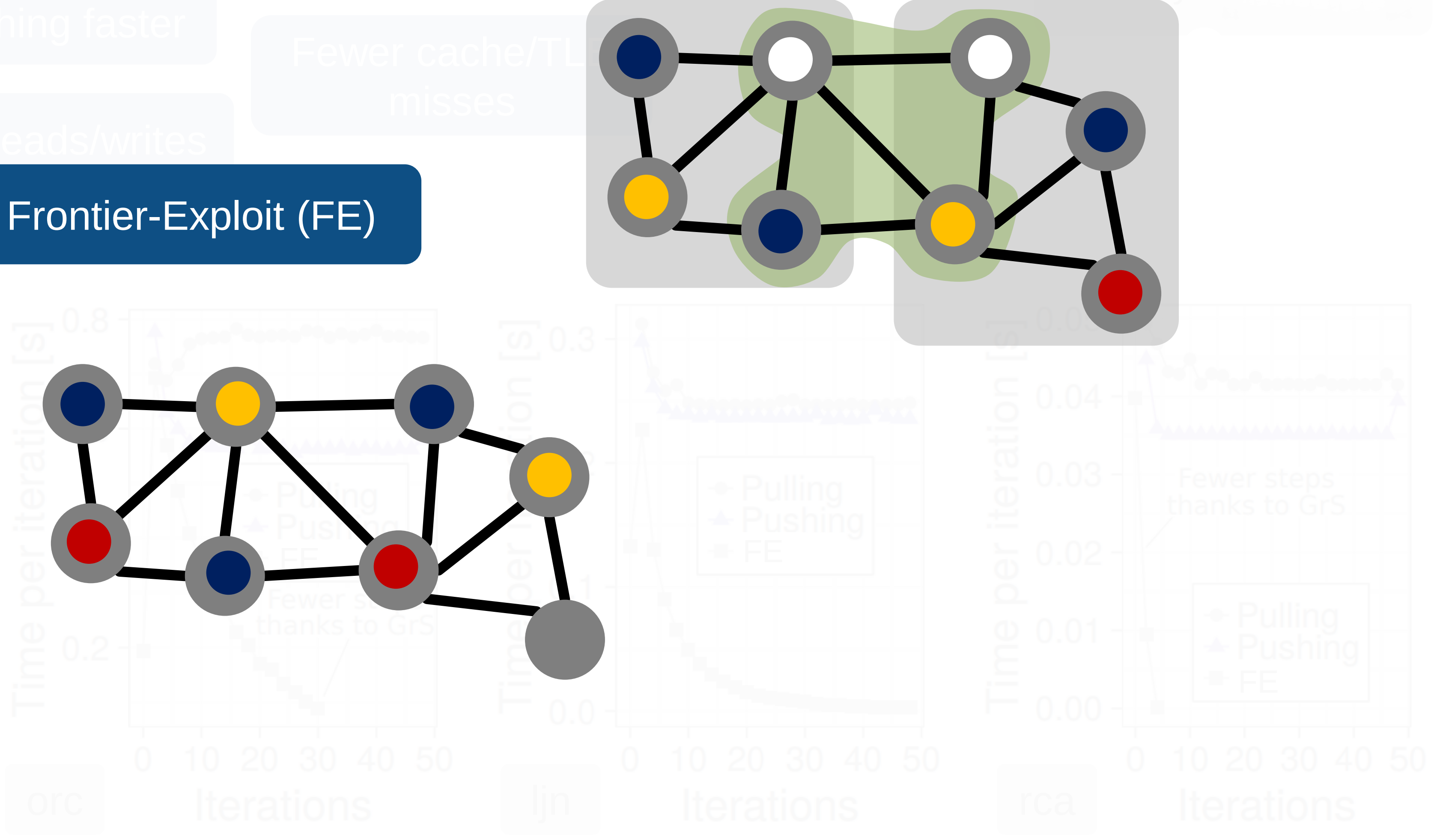
Shared-Memory

Pushing faster

Fewer cache/TL
misses

Fewer reads/writes

Frontier-Exploit (FE)



PERFORMANCE ANALYSIS
BOMAN GRAPH COLORING

SNAP

orc, ljn: social networks
rca: road network

Shared-Memory

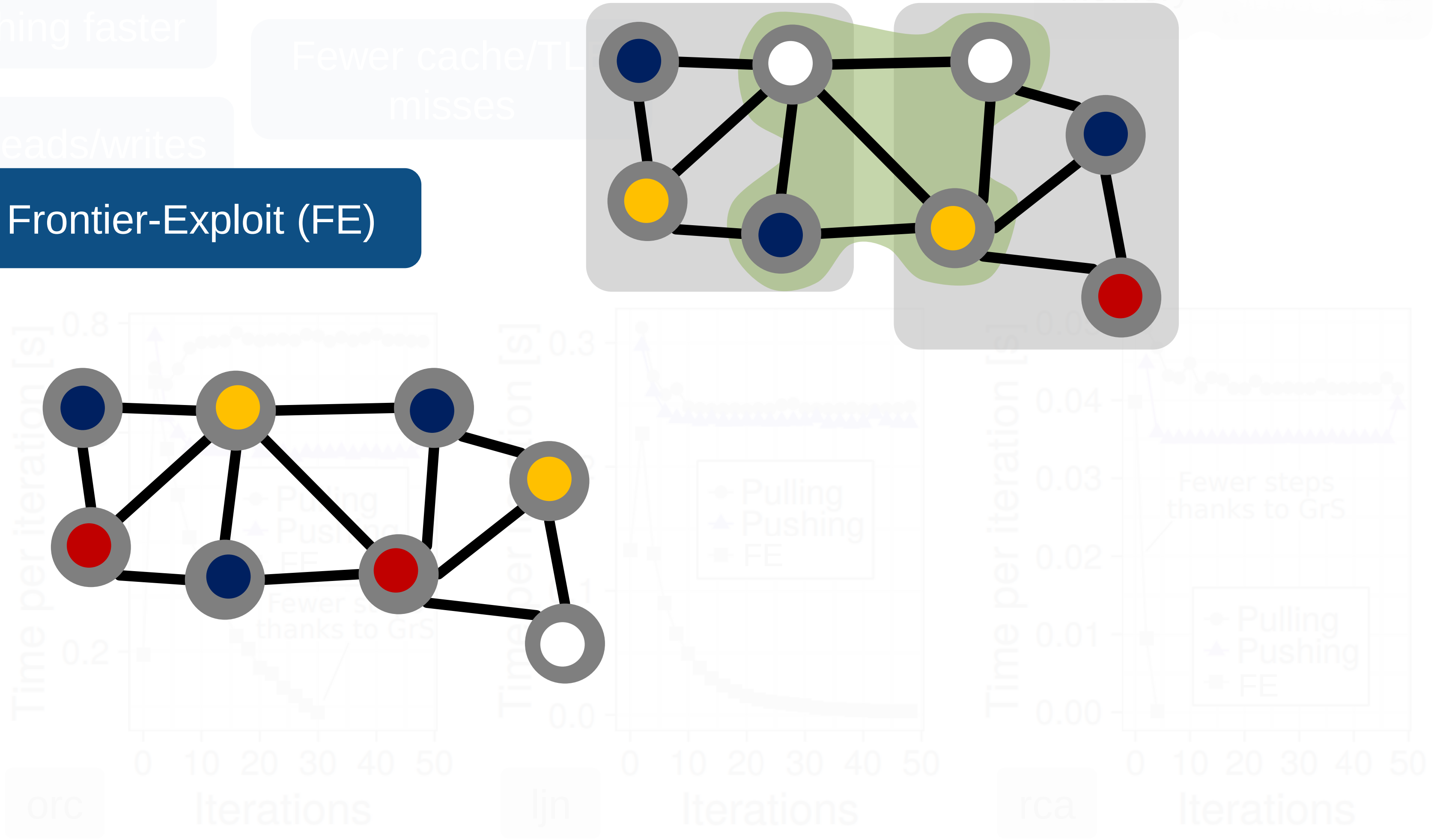
!

Pushing faster

Fewer cache/TLB misses

Fewer reads/writes

Frontier-Exploit (FE)

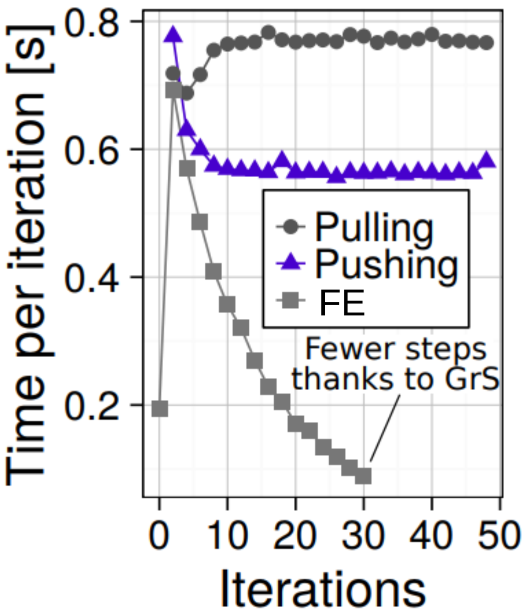


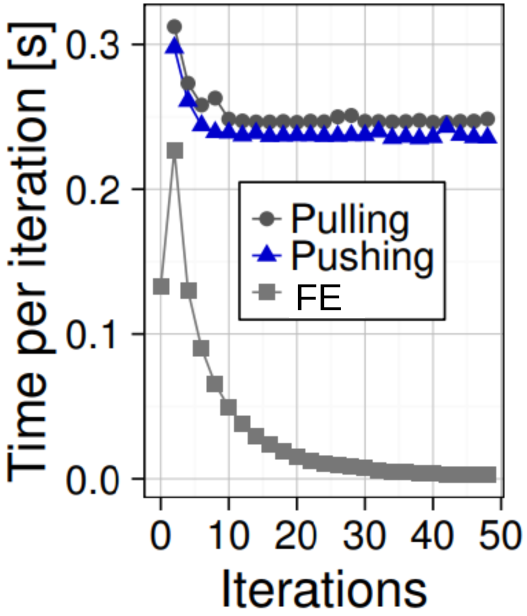
PERFORMANCE ANALYSIS
BOMAN GRAPH COLORING + FE

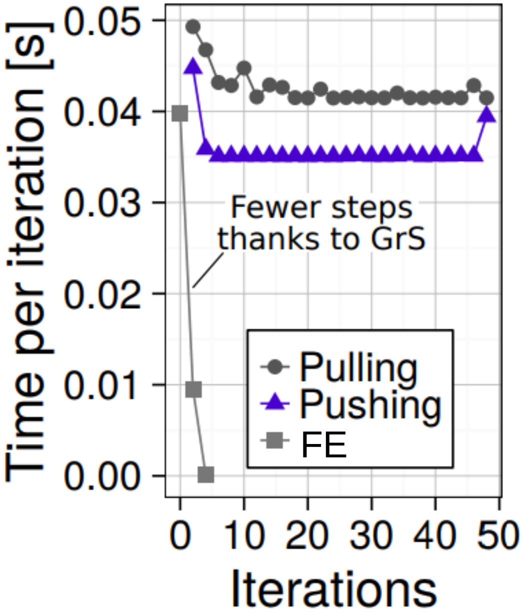
SNAP orc, ljn: social networks
rca: road network

Shared-Memory

FE: Frontier-Exploit (+ more, check the paper☺)







PERFORMANCE ANALYSIS
BOMAN GRAPH COLORING + FE

orc, ljn: social networks

rca: road network

Shared-Memory

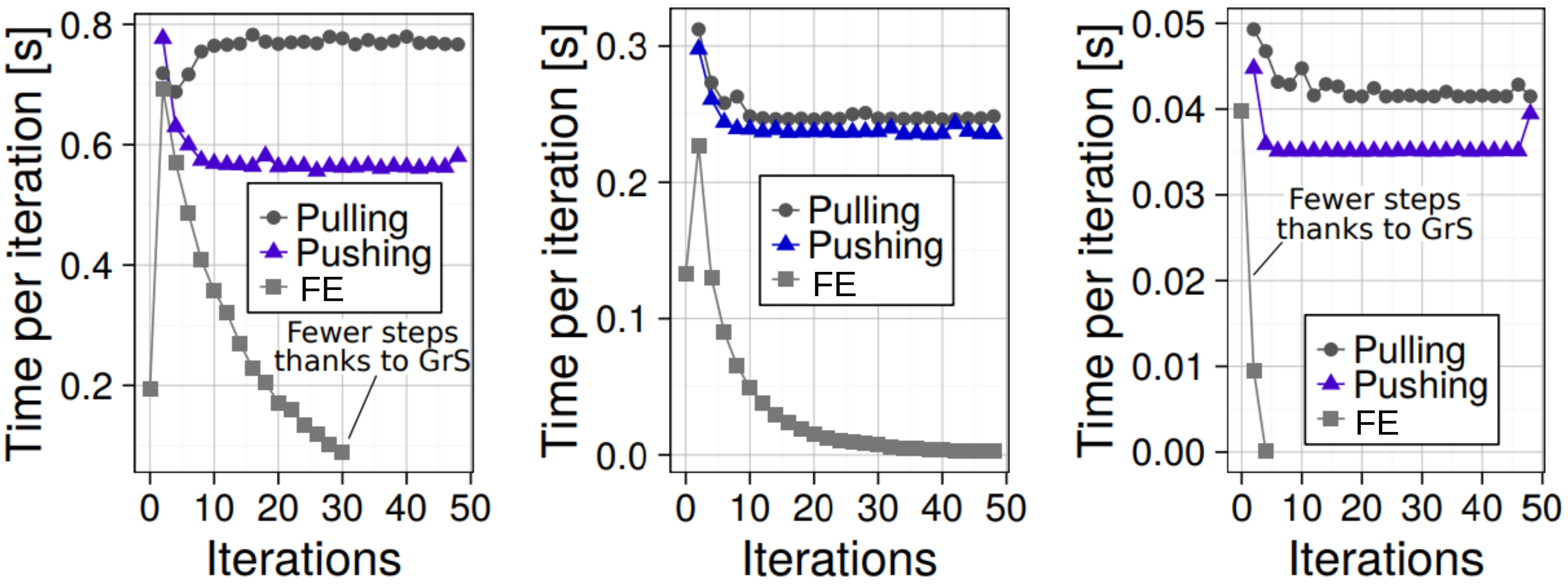
FE: Frontier-Exploit (+ more, check the paper☺)

!

Performance improvements

Fewer iterations

Fewer reads/writes



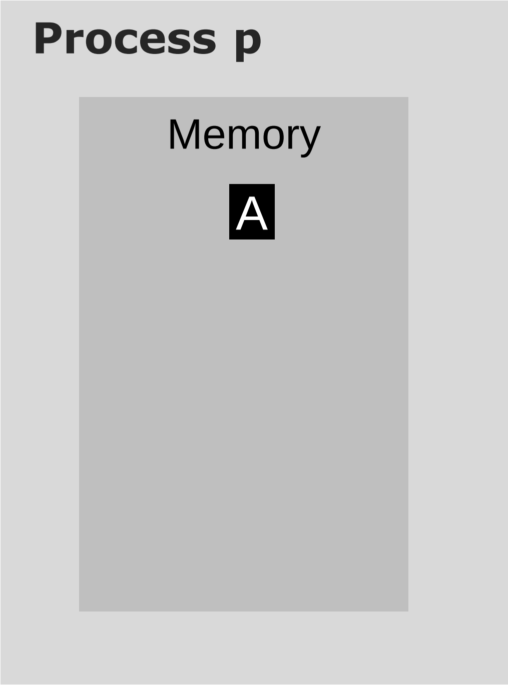
Before we move to Distributed-Memory analyses…

Before we move to Distributed-Memory analyses…

…a brief recap on Remote Memory Access (RMA)

REMOTE MEMORY ACCESS (RMA) PROGRAMMING

REMOTE MEMORY ACCESS (RMA) PROGRAMMING



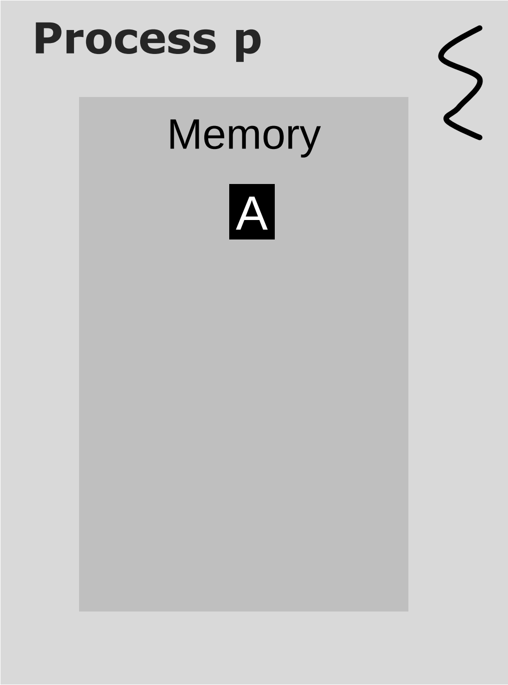
Process p

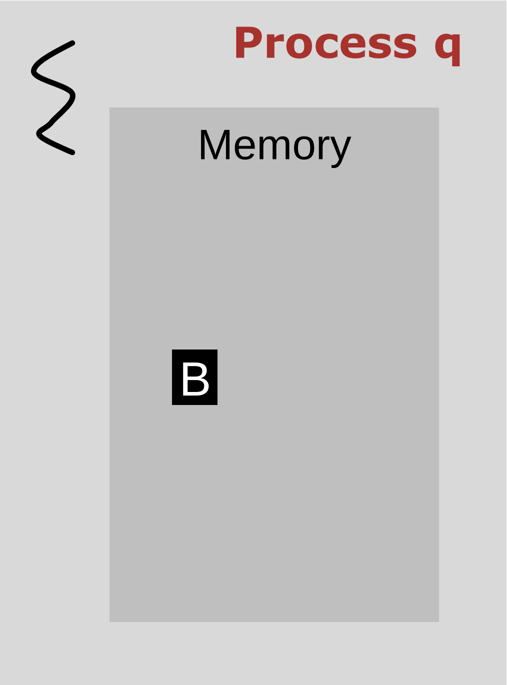
Memory

A

REMOTE MEMORY ACCESS (RMA) PROGRAMMING

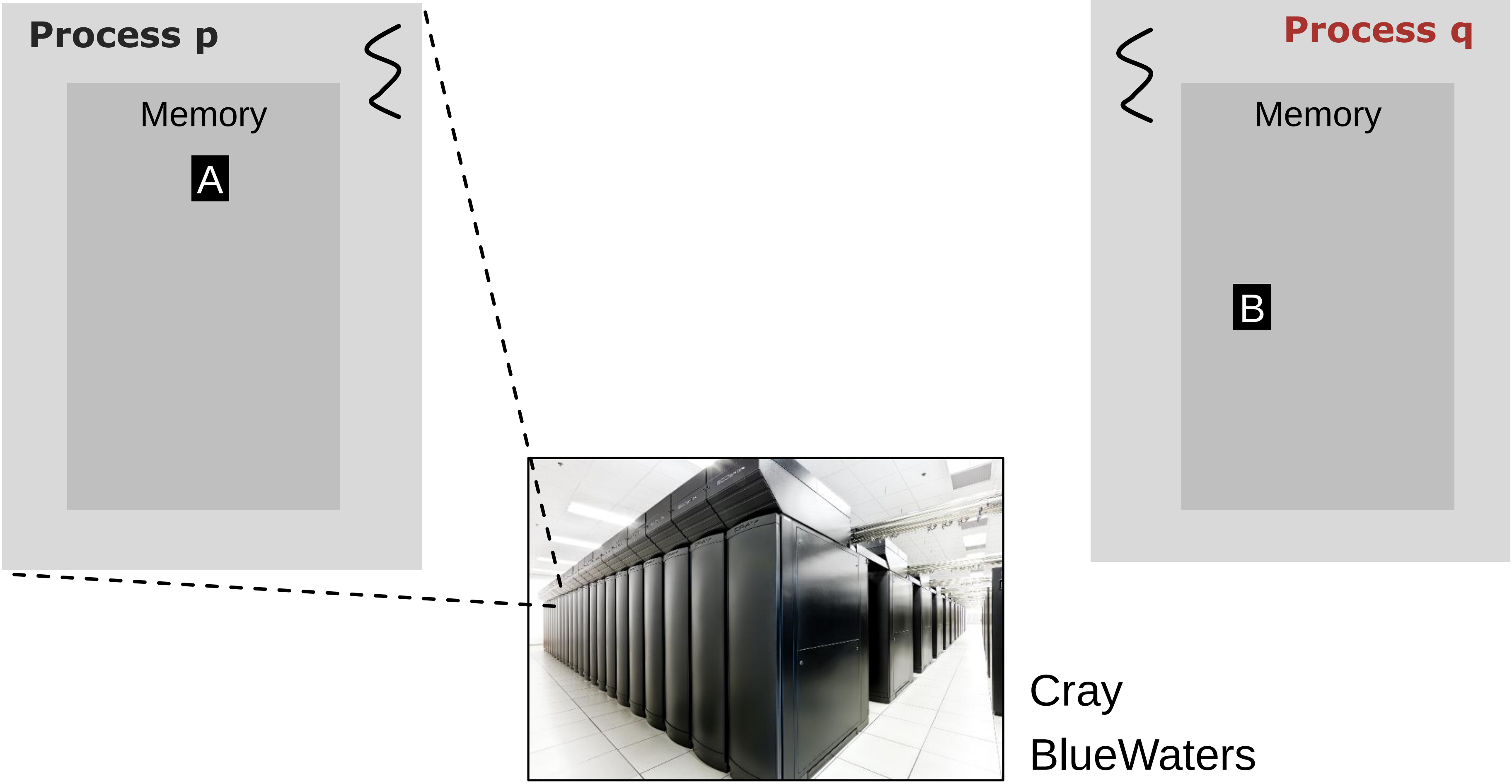
REMOTE MEMORY ACCESS (RMA) PROGRAMMING





Cray
BlueWaters

REMOTE MEMORY ACCESS (RMA) PROGRAMMING



Cray
BlueWaters

REMOTE MEMORY ACCESS (RMA) PROGRAMMING

Process p

Memory

A

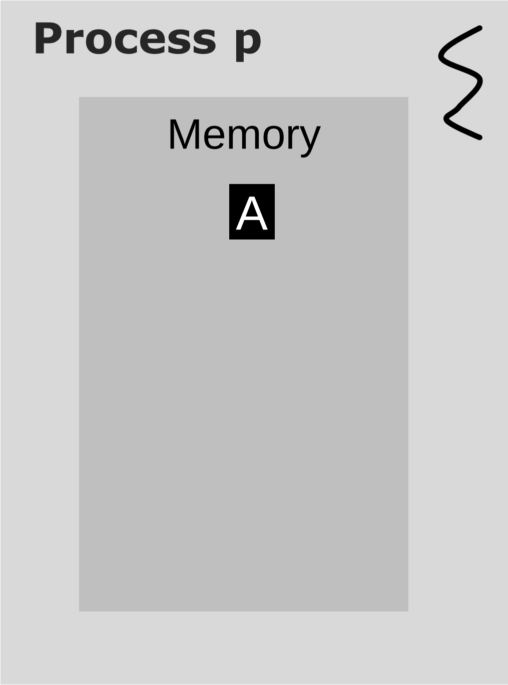
Process q

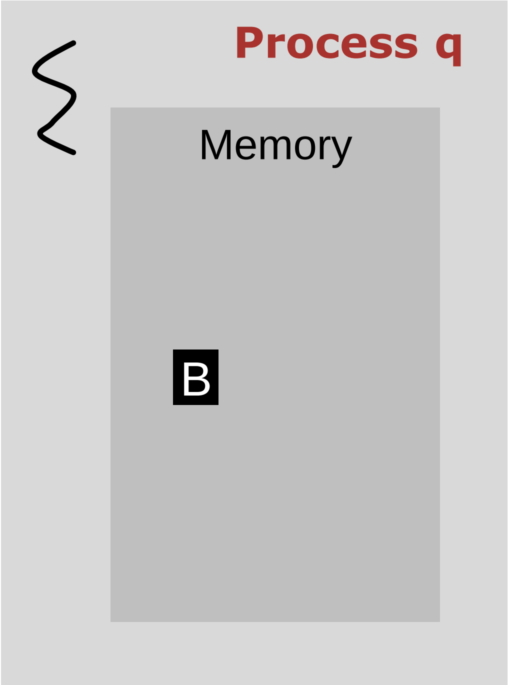
Memory

B

Cray
BlueWaters

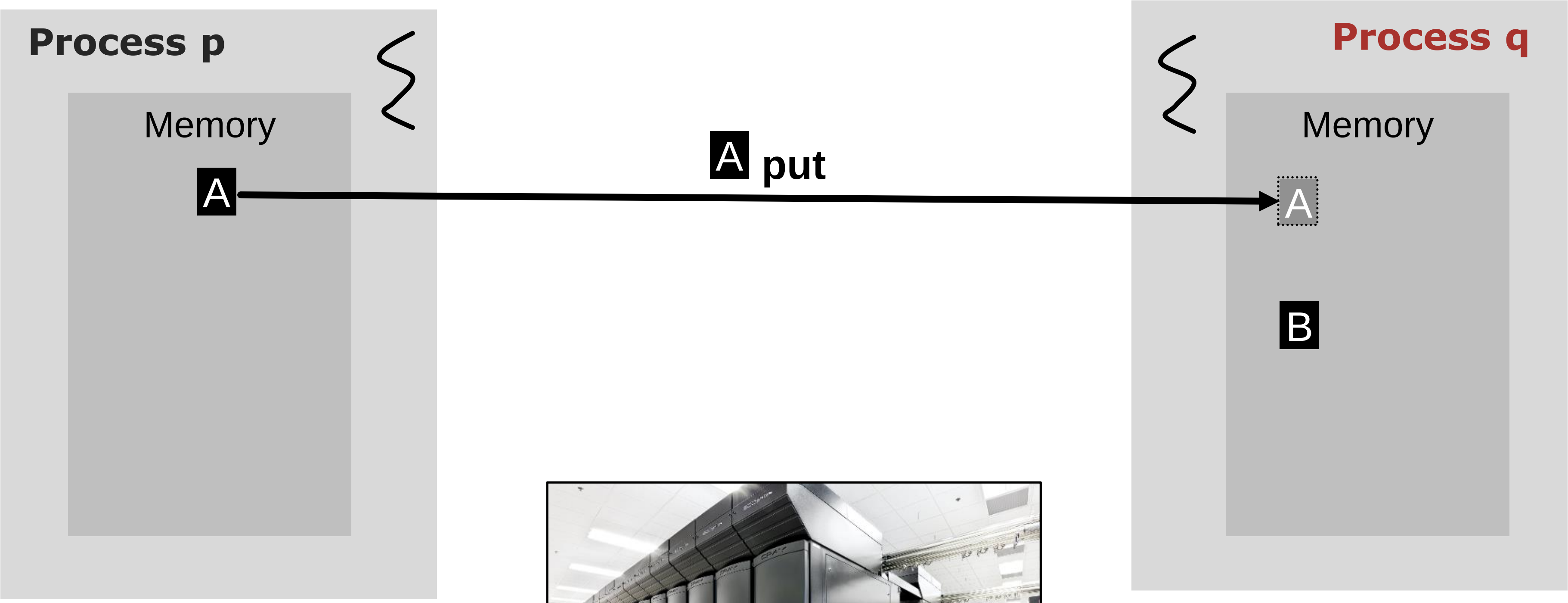
REMOTE MEMORY ACCESS (RMA) PROGRAMMING





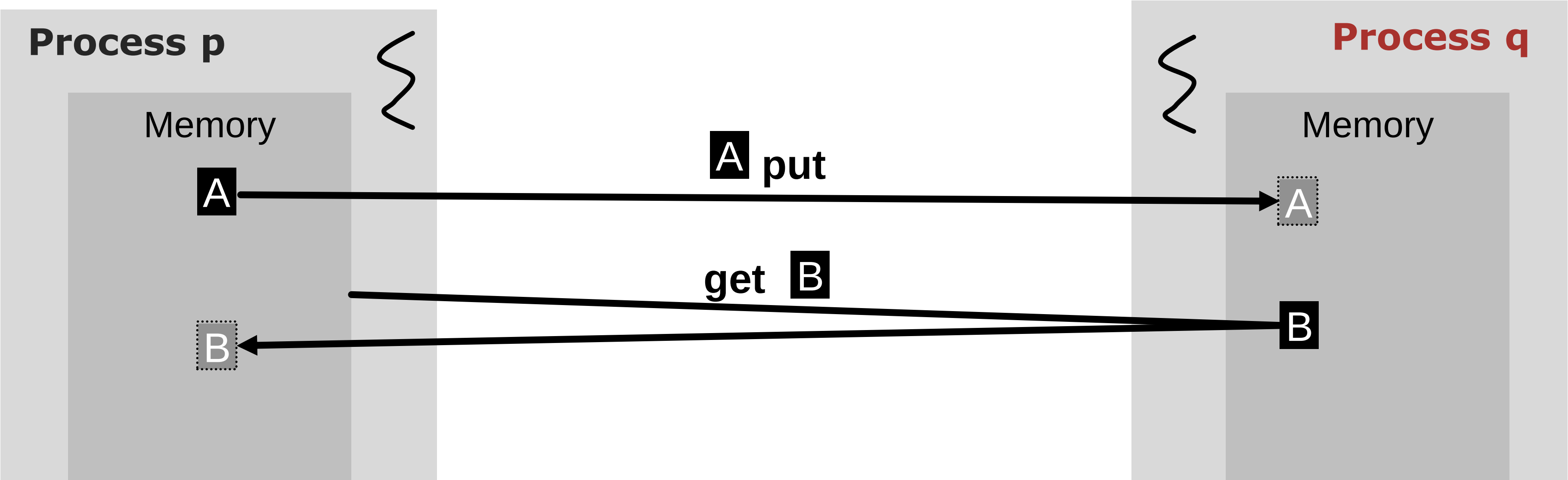
Cray
BlueWaters

REMOTE MEMORY ACCESS (RMA) PROGRAMMING



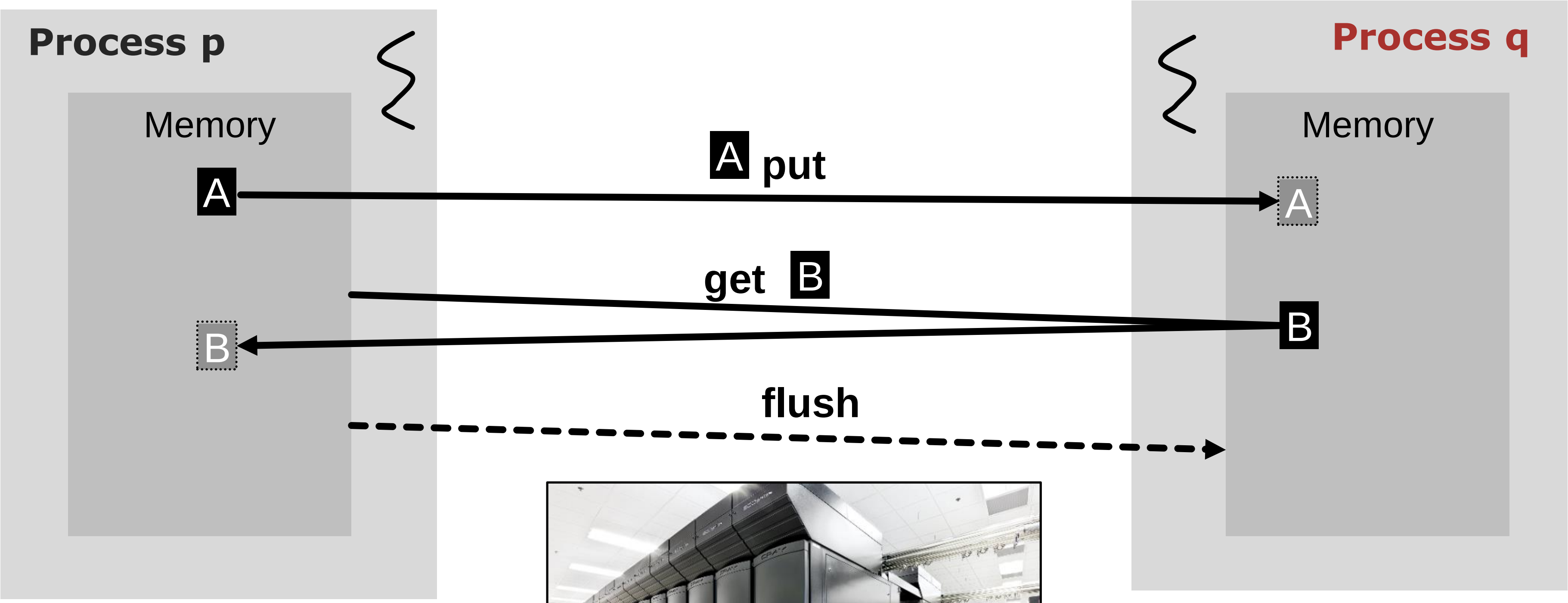
Cray
BlueWaters

REMOTE MEMORY ACCESS (RMA) PROGRAMMING



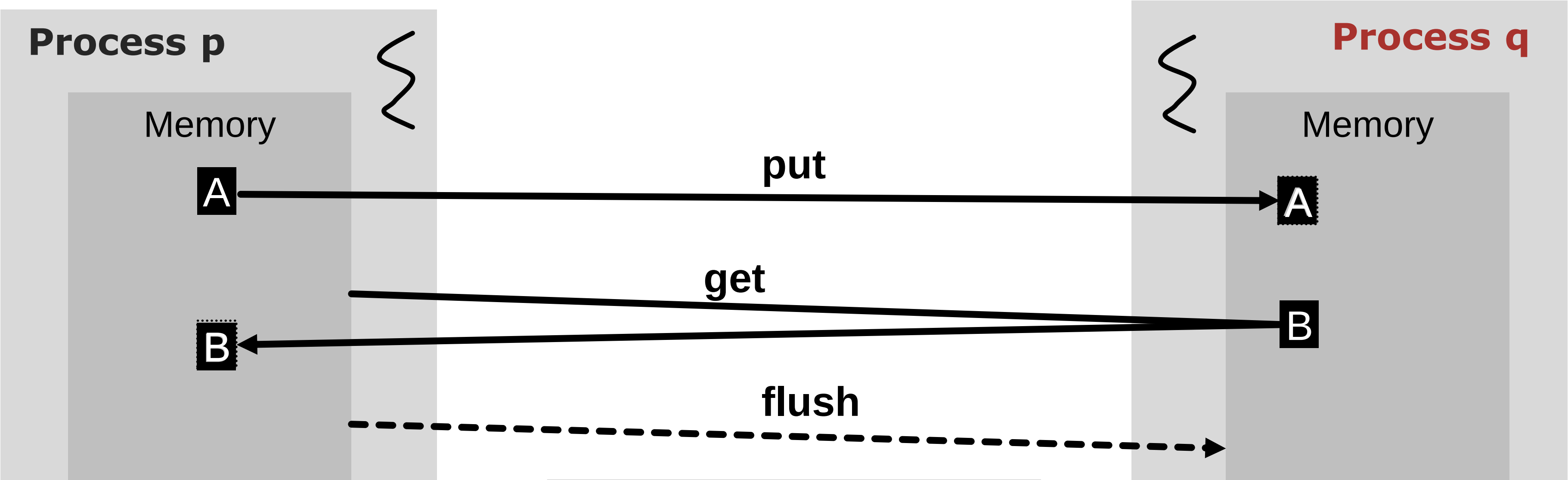
Cray
BlueWaters

REMOTE MEMORY ACCESS (RMA) PROGRAMMING



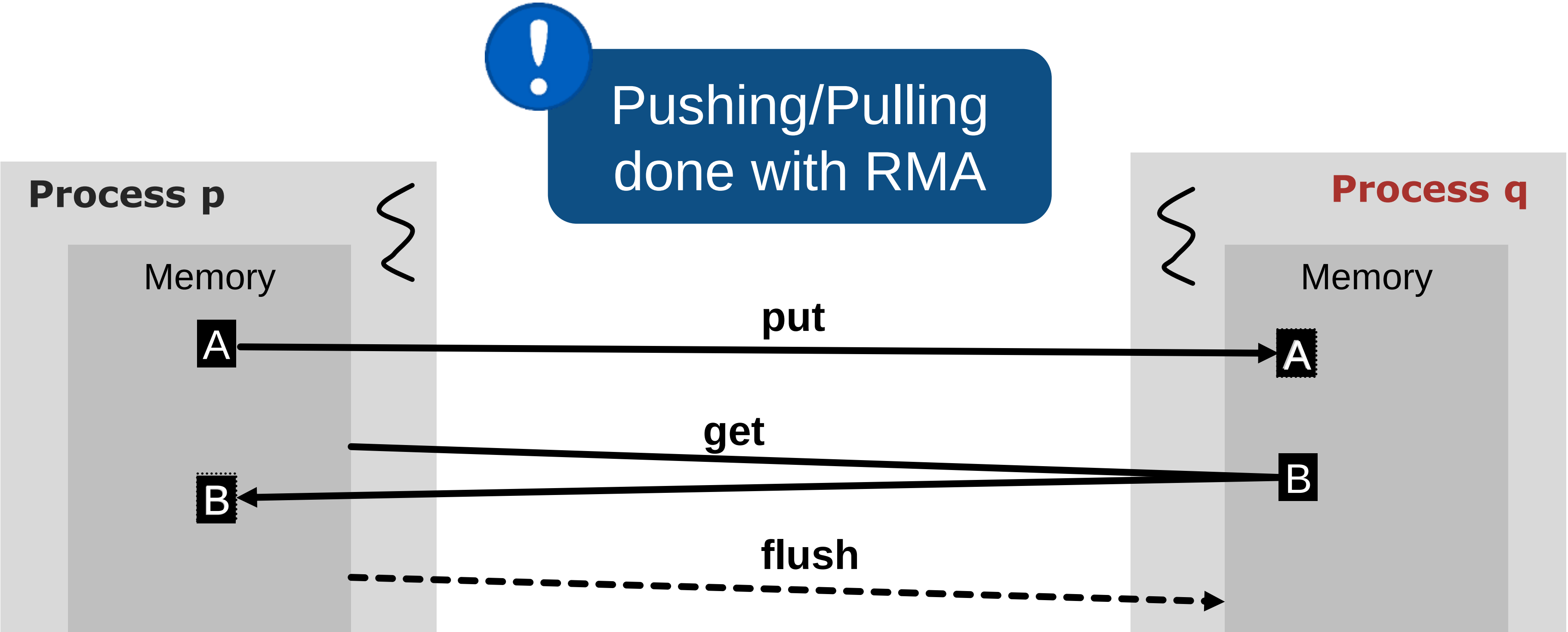
Cray
BlueWaters

REMOTE MEMORY ACCESS (RMA) PROGRAMMING



Cray
BlueWaters

REMOTE MEMORY ACCESS (RMA) PROGRAMMING



Pushing/Pulling done with RMA

Process p

Memory

A

B

put

get

flush

Process q

Memory

A

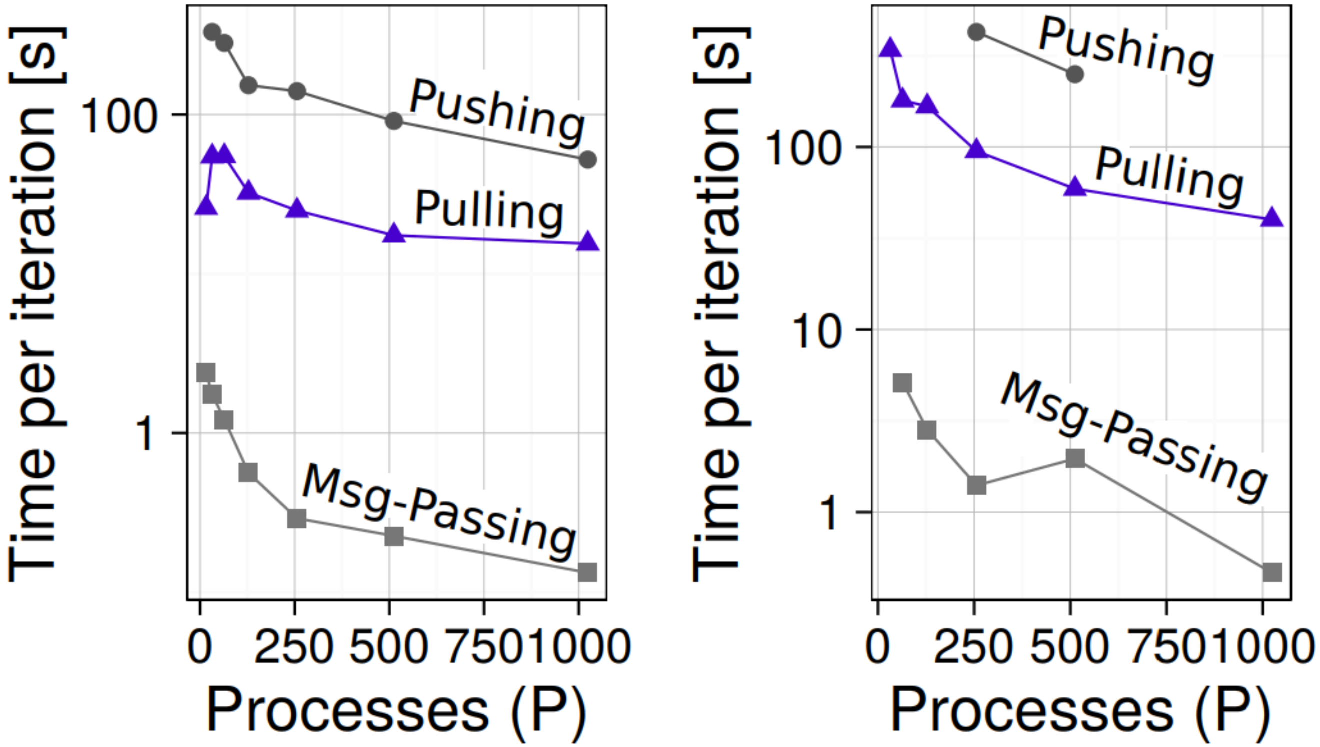
B

Cray
BlueWaters

PERFORMANCE ANALYSIS
PAGERANK

Kronecker graphs

Distributed
-Memory



PERFORMANCE ANALYSIS

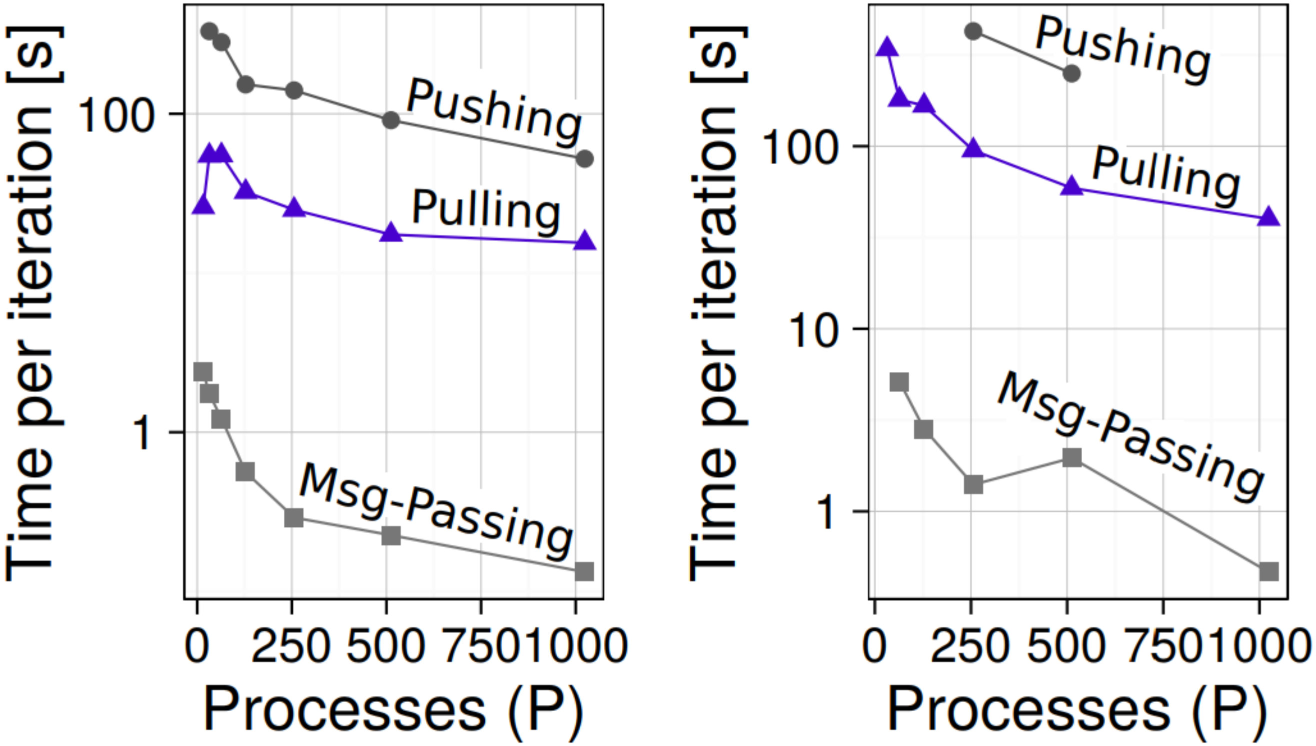
PAGERANK

Kronecker graphs

Distributed
-Memory

$n = 2^{25}, m = 2^{27}$

$n = 2^{27}, m = 2^{29}$



PERFORMANCE ANALYSIS

PAGERANK

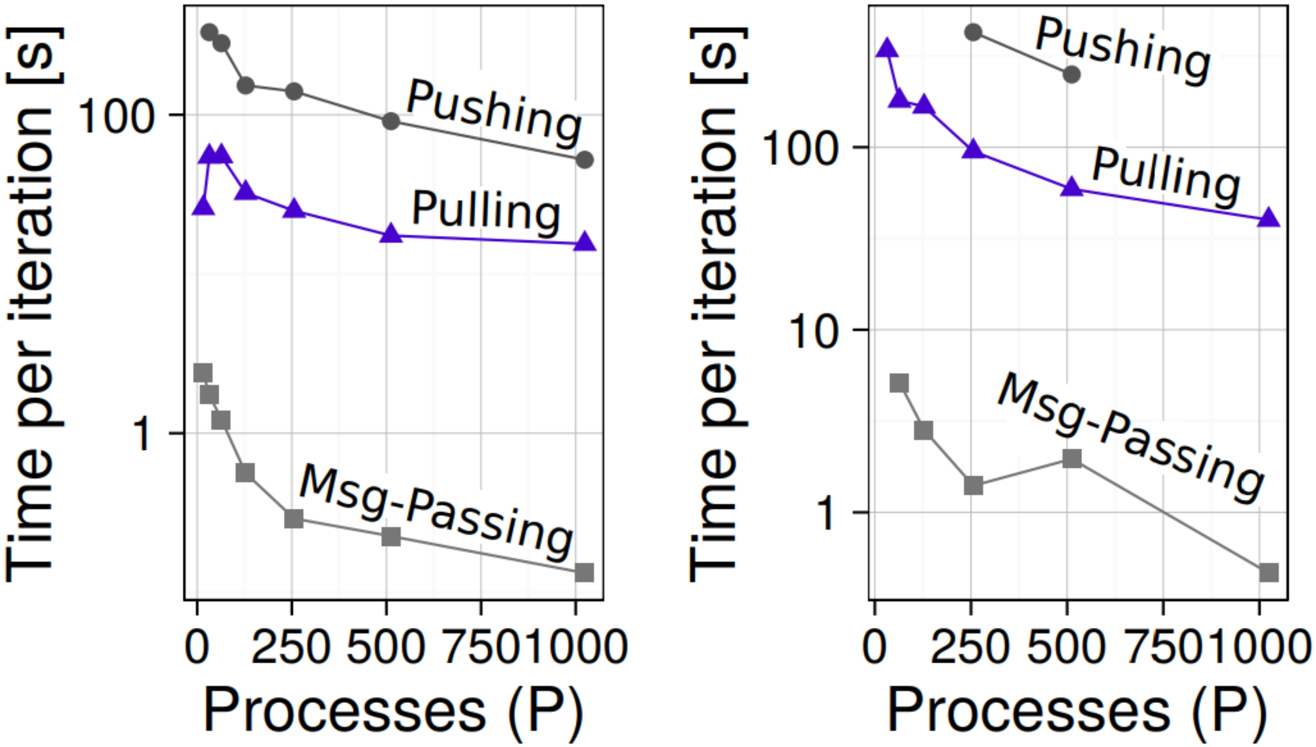
Kronecker graphs

Distributed
-Memory

! Msg-Passing fastest

$n = 2^{25}, m = 2^{27}$

$n = 2^{27}, m = 2^{29}$



PERFORMANCE ANALYSIS

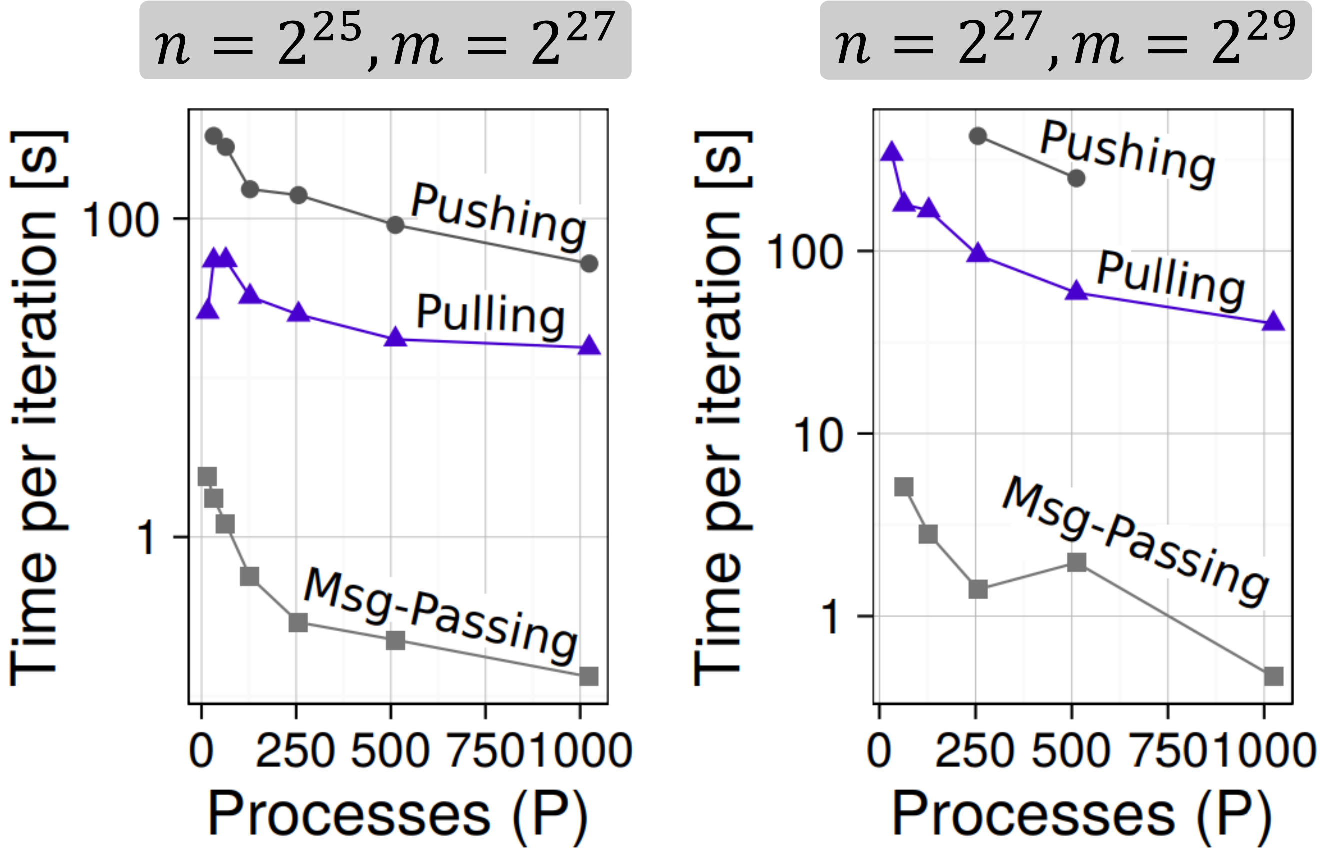
PAGERANK

Kronecker graphs

Distributed -Memory

! Msg-Passing fastest

Pulling incurs more communication while pushing expensive underlying locking



$n = 2^{25}, m = 2^{27}$

$n = 2^{27}, m = 2^{29}$

PERFORMANCE ANALYSIS

PAGERANK

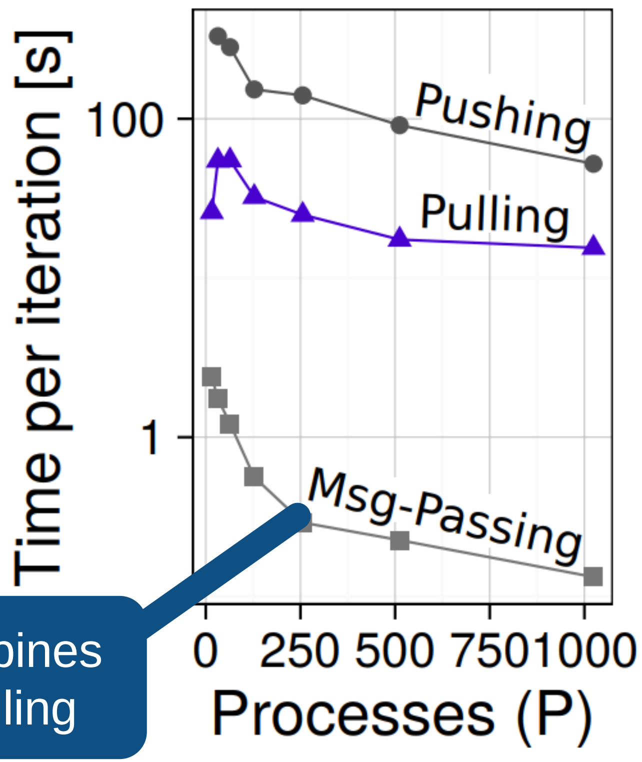
! Msg-Passing fastest

Kronecker graphs

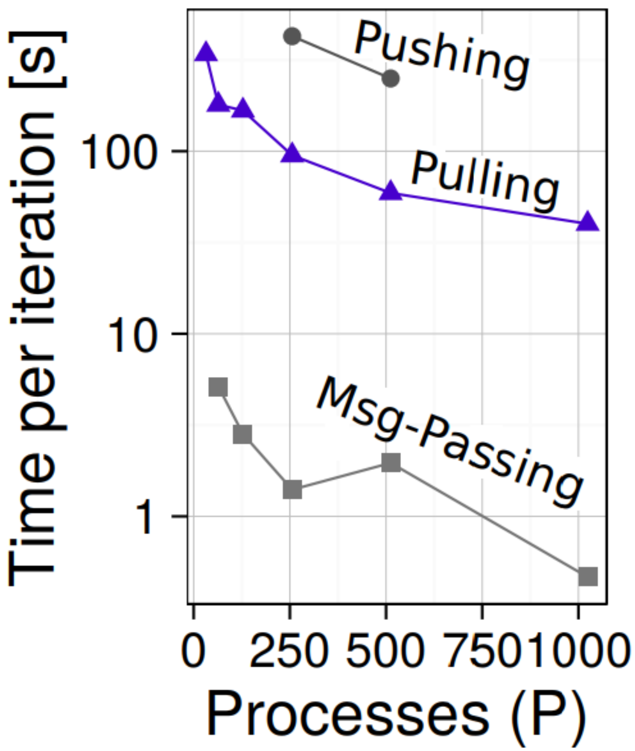
Distributed -Memory

Pulling incurs more communication while pushing expensive underlying locking

$n = 2^{25}, m = 2^{27}$



$n = 2^{27}, m = 2^{29}$



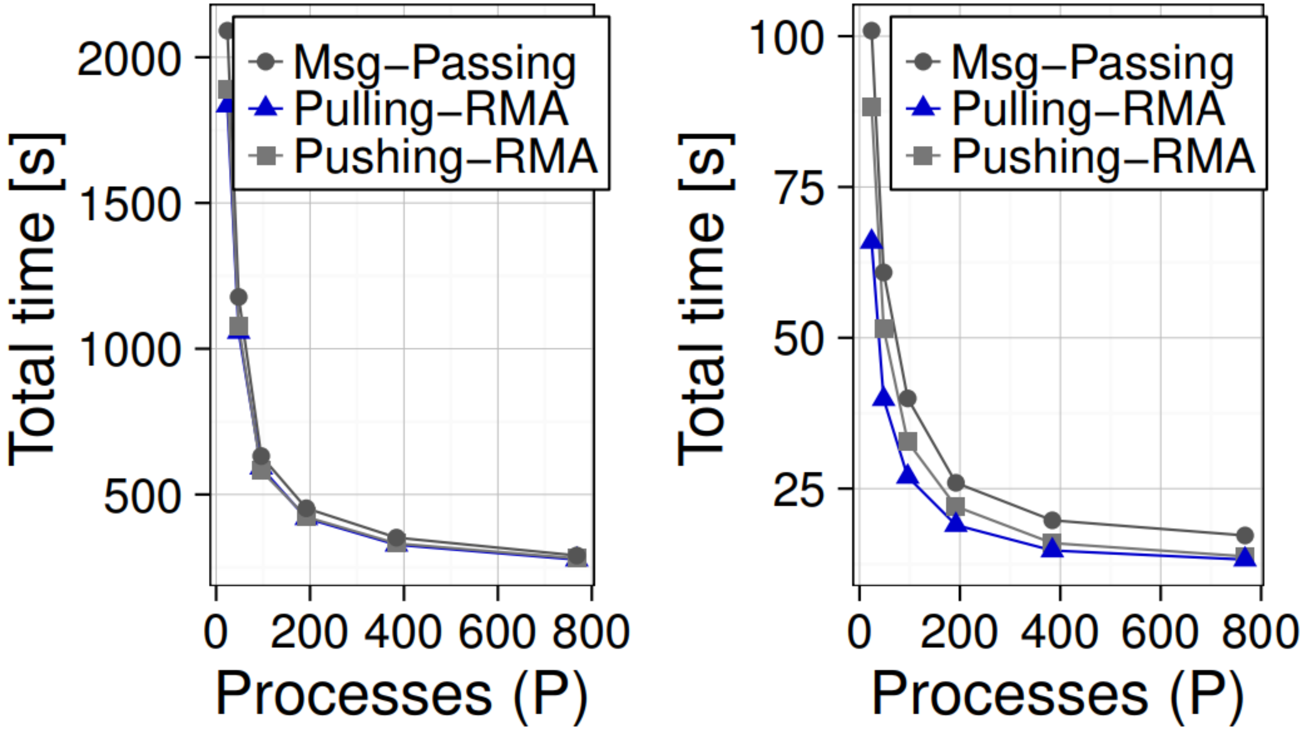
! Collectives: combines pushing and pulling

PERFORMANCE ANALYSIS

TRIANGLE COUNTING

orc, ljn: social networks

Distributed -Memory



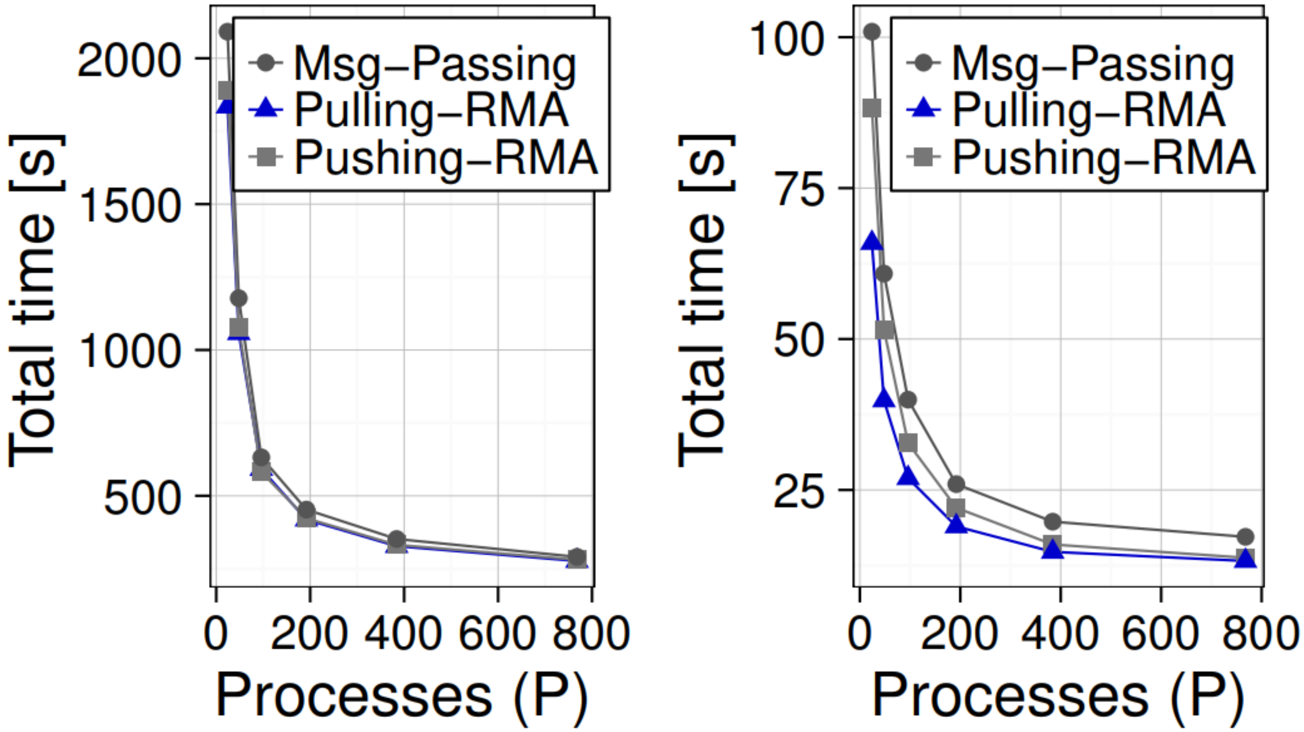
PERFORMANCE ANALYSIS

TRIANGLE COUNTING

SNAP orc, ljn: social networks

Distributed
-Memory

RMA fastest



PERFORMANCE ANALYSIS

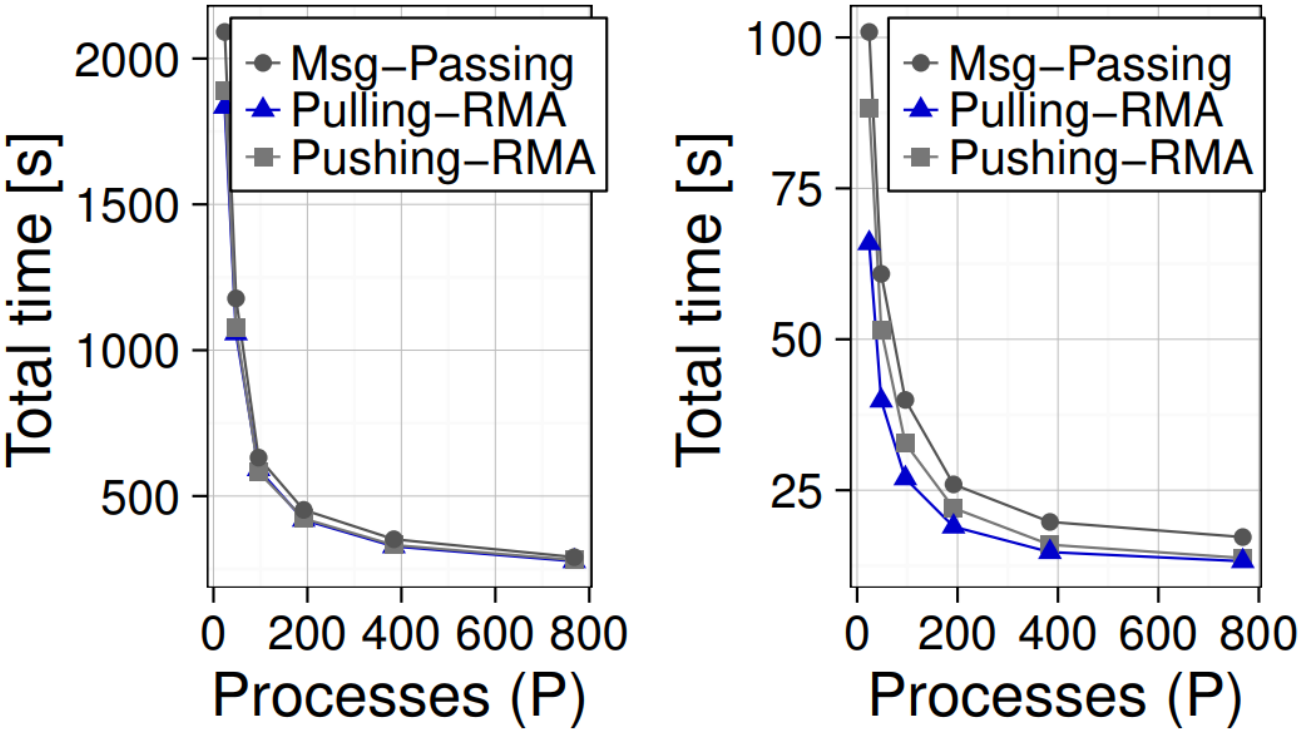
TRIANGLE COUNTING

SNAP orc, ljn: social networks

Msg-Passing now incurs more communication

Distributed -Memory

RMA fastest



PERFORMANCE ANALYSIS

TRIANGLE COUNTING

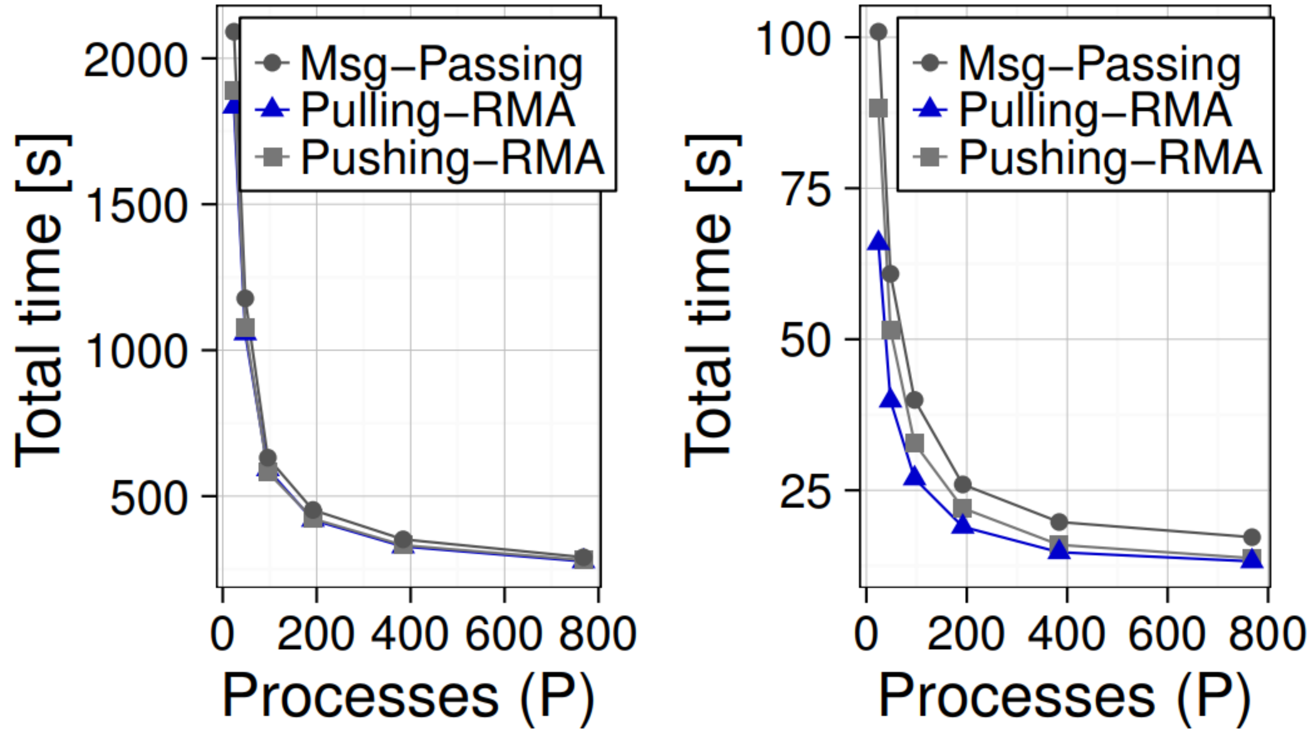
orc, ljn: social networks

Distributed -Memory

Msg-Passing now incurs more communication

RMA fastest

Pushing does not require the expensive locking protocol (Cray offers fast remote atomics for integers)



PUSHING VS. PULLING

RESEARCH QUESTIONS

Yes (developed 7 algorithms and the total of 11 variants)

Answered ☺

Check the paper ☺

Can be described with the actual dichotomy

What is performance?

How effective are the incorporated strategies?

Is pushing or pulling faster? When and why?

What is the impact of the programming model? environment?

PUSHING VS. PULLING

RESEARCH QUESTIONS

Pushing faster if its complexity lower

Pulling faster when their complexities match.

What is performance?

How effective are the incorporated strategies?

Is pushing or pulling faster? When and why?

What is the impact of the programming model? environment?

PUSHING VS. PULLING

RESEARCH QUESTIONS

What is performance?

How effective are the incorporated strategies?

Is pushing or pulling faster? When and why?

What is the impact of the programming model? environment?

Pushing faster if its complexity lower

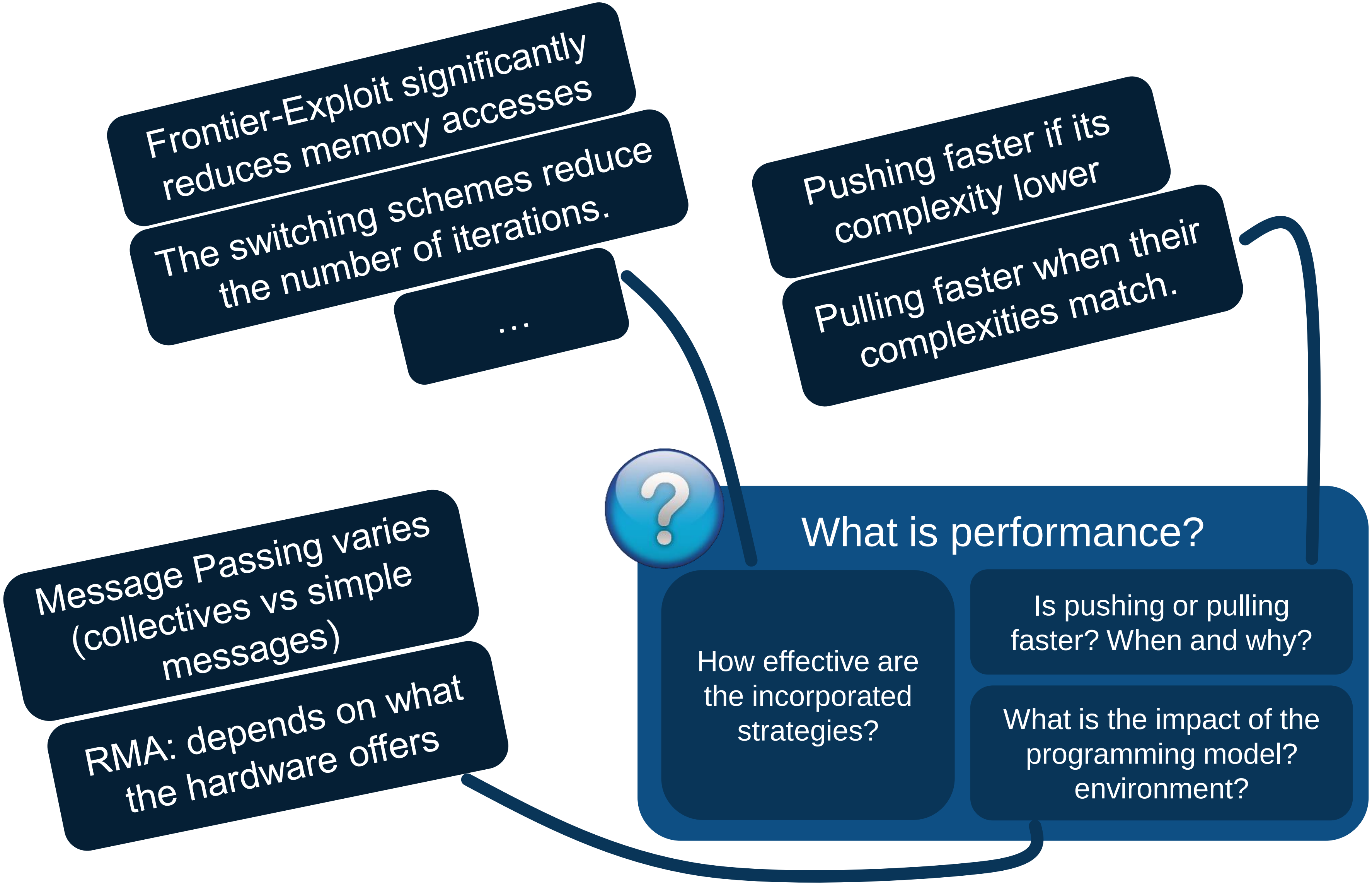
Pulling faster when their complexities match.

Message Passing varies (collectives vs simple messages)

RMA: depends on what the hardware offers

PUSHING VS. PULLING

RESEARCH QUESTIONS



Frontier-Exploit significantly reduces memory accesses

The switching schemes reduce the number of iterations.

…

Pushing faster if its complexity lower

Pulling faster when their complexities match.

Message Passing varies (collectives vs simple messages)

RMA: depends on what the hardware offers

What is performance?

How effective are the incorporated strategies?

Is pushing or pulling faster? When and why?

What is the impact of the programming model? environment?

PUSHING VS. PULLING

RESEARCH QUESTIONS

ignificantly

M

RMA: depend
the hardware offers

strategies?

?

What is the impact of the
programming model?
environment?

To Push or To Pull?

PUSHING VS. PULLING

RESEARCH QUESTIONS

ignificantly

To Push or To Pull?

If the complexities match: pull

M

RMA: depende
the hardware offers

?

strategies?

What is the impact of the programming model? environment?

PUSHING VS. PULLING
RESEARCH QUESTIONS

ignificantly

!

To Push or To Pull?

If the complexities match: pull

Otherwise: push

N

?

RMA: depend
the hardware offers

strategies?

What is the impact of the
programming model?
environment?

PUSHING VS. PULLING
RESEARCH QUESTIONS

ignificantly

M

RMA: depends
the hardware offers

strategies?

?

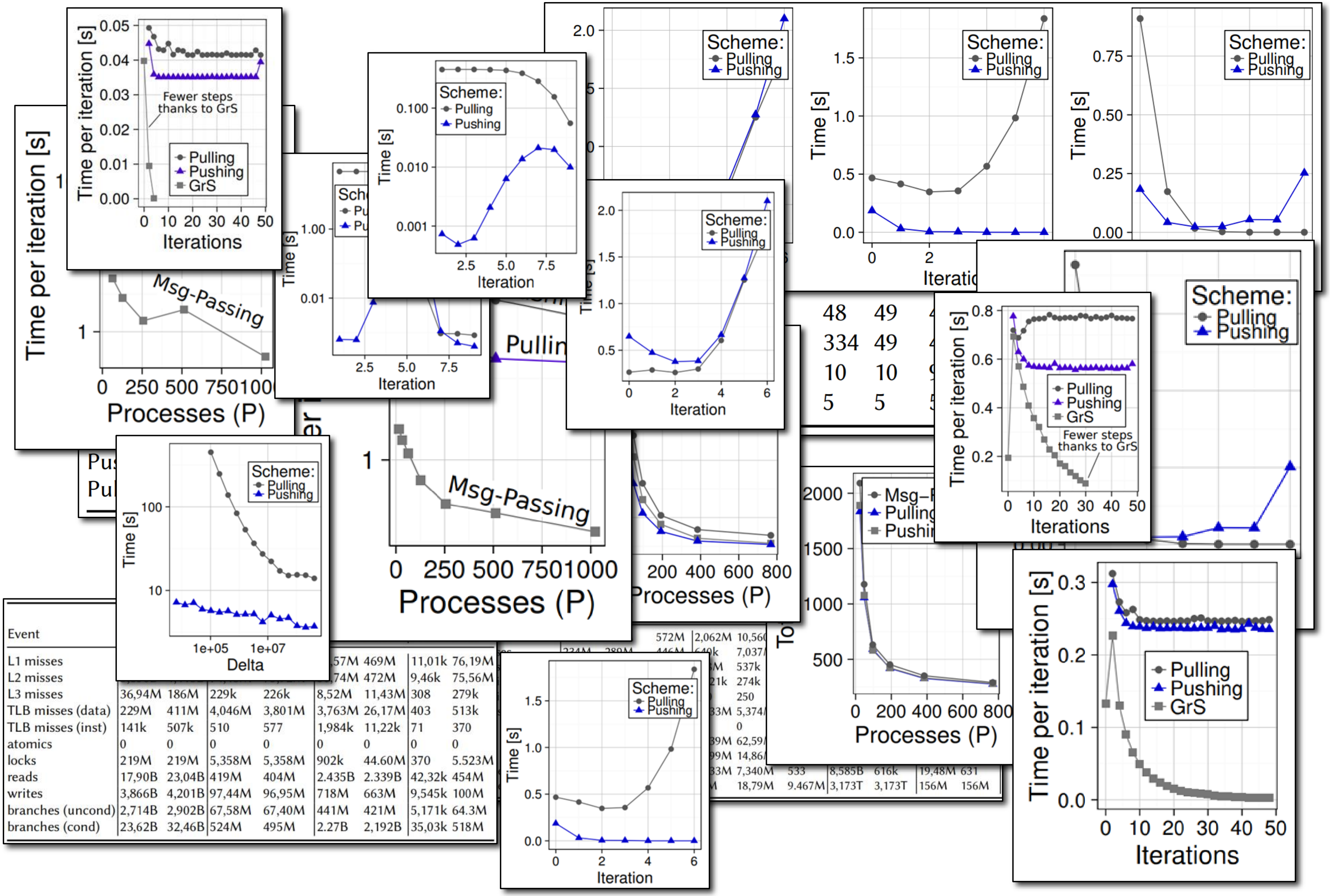
What is the impact of the
programming model?
environment?

To Push or To Pull?

If the complexities match: pull

Otherwise: push

+ check your hardware ☺

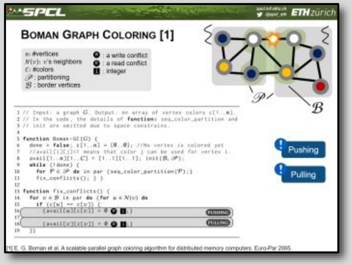


Event								
L1 misses					57M	469M	11,01k	76,19M
L2 misses					74M	472M	9,46k	75,56M
L3 misses	36,94M	186M	229k	226k	8,52M	11,43M	308	279k
TLB misses (data)	229M	411M	4,046M	3,801M	3,763M	26,17M	403	513k
TLB misses (inst)	141k	507k	510	577	1,984k	11,22k	71	370
atomics	0	0	0	0	0	0	0	0
locks	219M	219M	5,358M	5,358M	902k	44.60M	370	5.523M
reads	17,90B	23,04B	419M	404M	2.435B	2.339B	42,32k	454M
writes	3,866B	4,201B	97,44M	96,95M	718M	663M	9,545k	100M
branches (uncond)	2,714B	2,902B	67,58M	67,40M	441M	421M	5,171k	64.3M
branches (cond)	23,62B	32,46B	524M	495M	2.27B	2,192B	35,03k	518M

CONCLUSIONS

CONCLUSIONS

Push vs. Pull: Applicability



BOMAN GRAPH COLORING [1]

n: #vertices
$N(v)$: v's neighbors
C: #colors
$\mathscr{P}$: partitioning
$\mathscr{B}$: border vertices

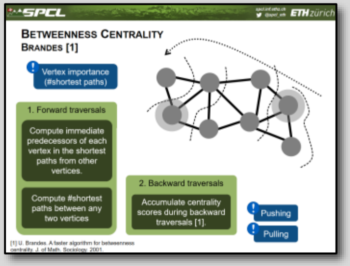
Ⓦ : a write conflict
Ⓡ : a read conflict
I : integer

```
// Input: a graph G. Output: An array of vertex colors c[1..n].
// In the code, the details of functions seq_color_partition and
// init are omitted due to space constrains.

function Boman-GC(G) {
  done = false; c[1..n] = [0..0]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..n][1..C] = [1..1][1..1]; init(B, P);
  while (!done) {
    for P ∈ P do in par {seq_color_partition(P);}
    fix_conflicts(); } }

function fix_conflicts() {
  for v ∈ B in par do {for u ∈ N(v) do
    if (c[u] == c[v]) {
      {avail[u][c[v]] = 0 Ⓦ I;}                PUSHING

      {avail[v][c[v]] = 0 Ⓡ I;}                PULLING
  }}
```

Pushing

Pulling

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.



BETWEENNESS CENTRALITY
BRANDES [1]

Vertex importance (#shortest paths)

1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices

2. Backward traversals

Accumulate centrality scores during backward traversals [1].

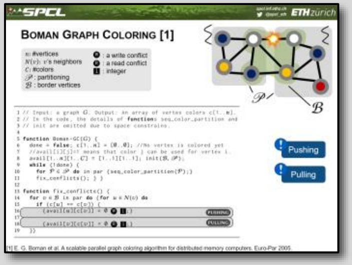
Pushing

Pulling

[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

CONCLUSIONS

Push vs. Pull: Applicability



BOMAN GRAPH COLORING [1]

n: #vertices
$N(v)$: v's neighbors
C: #colors
$\mathscr{P}$: partitioning
$\mathscr{B}$: border vertices

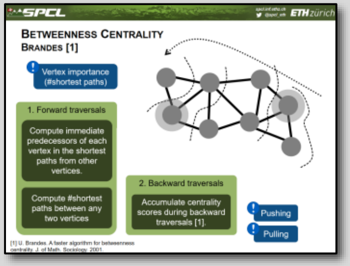
W : a write conflict
R : a read conflict
I : integer

```
// Input: a graph G. Output: An array of vertex colors c[1..n].
// In the code, the details of functions seq_color_partition and
// init are omitted due to space constrains.

function Boman-GC(G) {
  done = false; c[1..n] = [0..0]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..n][1..C] = [1..1][1..1]; init(B, P);
  while (!done) {
    for P ∈ P do in par {seq_color_partition(P);}
    fix_conflicts(); } }

function fix_conflicts() {
  for v ∈ B in par do {for u ∈ N(v) do
    if (c[u] == c[v]) {
      {avail[u][c[v]] = 0 W I;}                    PUSHING
      {avail[v][c[u]] = 0 R I;}                    PULLING
  }}
```

Pushing

Pulling

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.



BETWEENNESS CENTRALITY
BRANDES [1]

Vertex importance (#shortest paths)

1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices

2. Backward traversals

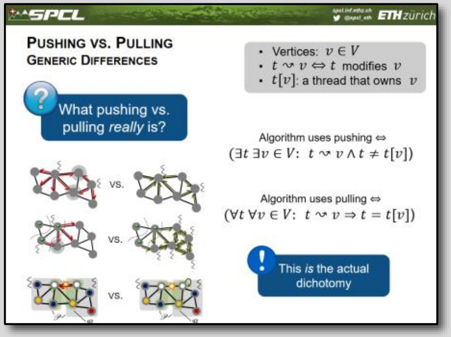
Accumulate centrality scores during backward traversals [1].

Pushing

Pulling

[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

Push vs. Pull: Dichotomy



PUSHING VS. PULLING
GENERIC DIFFERENCES

- Vertices: $v \in V$
- $t \rightsquigarrow v \Leftrightarrow t$ modifies v
- $t[v]$: a thread that owns v

What pushing vs. pulling *really* is?

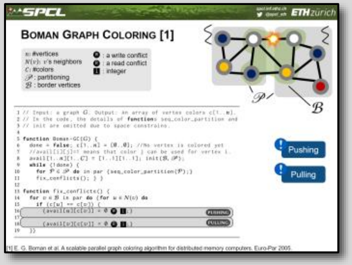
Algorithm uses pushing ⇔
$(\exists t\ \exists v \in V:\ t \rightsquigarrow v \land t \neq t[v])$

Algorithm uses pulling ⇔
$(\forall t\ \forall v \in V:\ t \rightsquigarrow v \Rightarrow t = t[v])$

This *is* the actual dichotomy

CONCLUSIONS

Push vs. Pull: Applicability



BOMAN GRAPH COLORING [1]

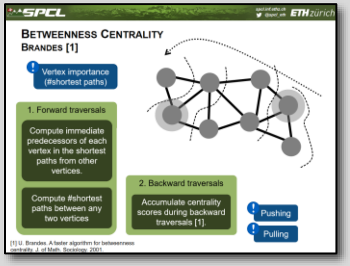
n: #vertices
N(v): v's neighbors
C: #colors
$\mathcal{P}$: partitioning
$\mathcal{B}$: border vertices

a write conflict
a read conflict
integer

Pushing

Pulling

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.



BETWEENNESS CENTRALITY
BRANDES [1]

Vertex importance (#shortest paths)

1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices

2. Backward traversals

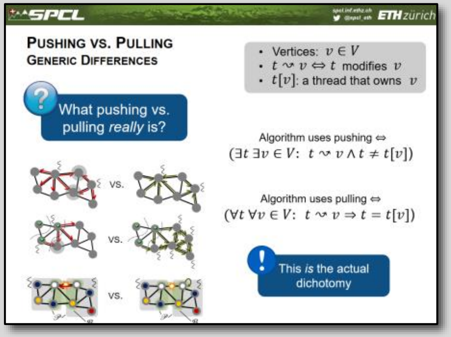
Accumulate centrality scores during backward traversals [1].

Pushing

Pulling

[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

Push vs. Pull: Dichotomy



PUSHING VS. PULLING
GENERIC DIFFERENCES

- Vertices: $v \in V$
- $t \rightsquigarrow v \Leftrightarrow t$ modifies v
- $t[v]$: a thread that owns v

What pushing vs. pulling *really* is?

Algorithm uses pushing ⇔
$(\exists t\ \exists v \in V\colon\ t \rightsquigarrow v \wedge t \neq t[v])$

Algorithm uses pulling ⇔
$(\forall t\ \forall v \in V\colon\ t \rightsquigarrow v \Rightarrow t = t[v])$

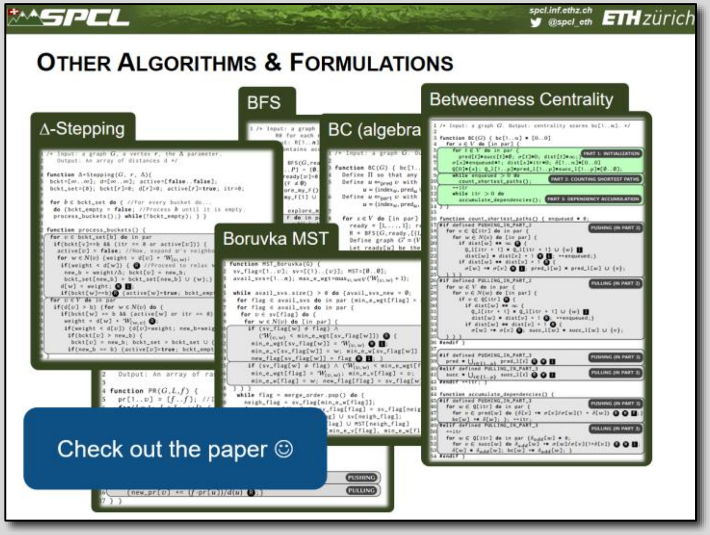
vs.

vs.

vs.

This *is* the actual dichotomy

Push vs. Pull: Formulations



OTHER ALGORITHMS & FORMULATIONS

Δ-Stepping

BFS

BC (algebra

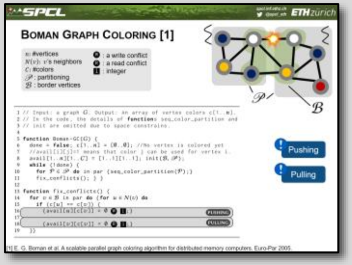
Betweenness Centrality

Boruvka MST

Check out the paper ☺

CONCLUSIONS

Push vs. Pull: Applicability



BOMAN GRAPH COLORING [1]

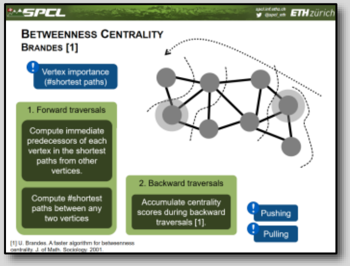
n: #vertices
$N(v)$: v's neighbors
C: #colors
$\mathscr{P}$: partitioning
$\mathscr{B}$: border vertices

● : a write conflict
● : a read conflict
■ : integer

Pushing

Pulling

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.



BETWEENNESS CENTRALITY
BRANDES [1]

Vertex importance (#shortest paths)

1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices

2. Backward traversals

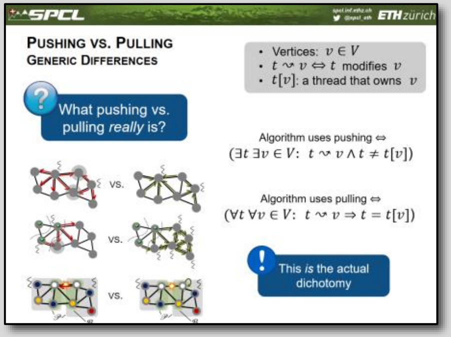
Accumulate centrality scores during backward traversals [1].

Pushing

Pulling

[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

Push vs. Pull: Dichotomy



PUSHING VS. PULLING
GENERIC DIFFERENCES

- Vertices: $v \in V$
- $t \rightsquigarrow v \Leftrightarrow t$ modifies v
- $t[v]$: a thread that owns v

What pushing vs. pulling *really* is?

vs.

vs.

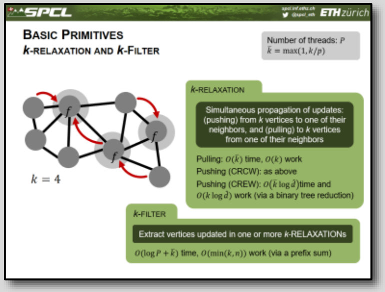
vs.

Algorithm uses pushing ⇔
$(\exists t\, \exists v \in V:\ t \rightsquigarrow v \wedge t \neq t[v])$

Algorithm uses pulling ⇔
$(\forall t\, \forall v \in V:\ t \rightsquigarrow v \Rightarrow t = t[v])$

This *is* the actual dichotomy

Push vs. Pull: Complexity



BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER

Number of threads: P
$\bar{k} = \max(1, k/p)$

$k = 4$

k-RELAXATION

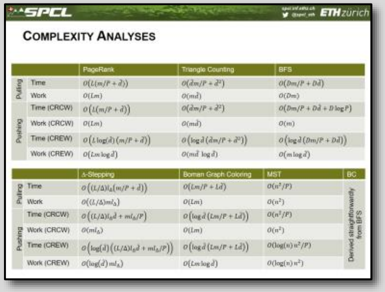
Simultaneous propagation of updates: (pushing) from *k* vertices to one of their neighbors, and (pulling) to *k* vertices from one of their neighbors

Pulling: $O(\bar{k})$ time, $O(k)$ work
Pushing (CRCW): as above
Pushing (CREW): $O(\bar{k}\log\hat{d})$time and $O(k\log\hat{d})$ work (via a binary tree reduction)

k-FILTER

Extract vertices updated in one or more *k*-RELAXATIONs

$O(\log P + \bar{k})$ time, $O(\min(k,n))$ work (via a prefix sum)

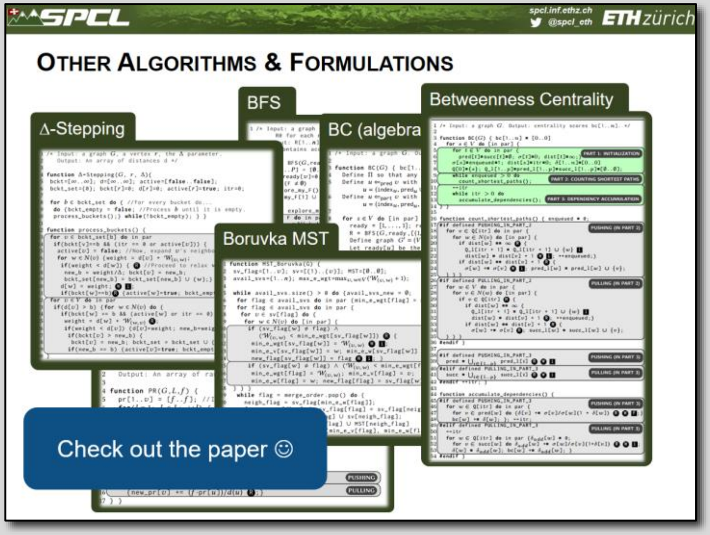


COMPLEXITY ANALYSES

		PageRank	Triangle Counting	BFS
Pulling	Time	$O(L(m/P + \hat{d}))$	$O(\hat{d}m/P + \hat{d}^2)$	$O(Dm/P + D\hat{d})$
	Work	$O(Lm)$	$O(m\hat{d})$	$O(Dm)$
Pushing	Time (CRCW)	$O\left(L(m/P + \hat{d})\right)$	$O(\hat{d}m/P + \hat{d}^2)$	$O(Dm/P + D\hat{d} + D\log P)$
	Work (CRCW)	$O(Lm)$	$O(m\hat{d})$	$O(m)$
	Time (CREW)	$O\left(L\log(\hat{d})(m/P + \hat{d})\right)$	$O\left(\log\hat{d}\,(\hat{d}m/P + \hat{d}^2)\right)$	$O\left(\log\hat{d}\,(Dm/P + D\hat{d})\right)$
	Work (CREW)	$O(Lm\log\hat{d})$	$O(m\hat{d}\log\hat{d})$	$O(m\log\hat{d})$

		Δ-Stepping	Boman Graph Coloring	MST	BC
Pulling	Time	$O\left((L/\Delta)l_\Delta(m/P + \hat{d})\right)$	$O(Lm/P + L\hat{d})$	$O(n^2/P)$	Derived straightforwardly from BFS
	Work	$O((L/\Delta)ml_\Delta)$	$O(Lm)$	$O(n^2)$	
Pushing	Time (CRCW)	$O\left((L/\Delta)l_\Delta\hat{d} + ml_\Delta/P\right)$	$O\left(\log\hat{d}\,(Lm/P + L\hat{d})\right)$	$O(n^2/P)$	
	Work (CRCW)	$O(ml_\Delta)$	$O(Lm)$	$O(n^2)$	
	Time (CREW)	$O\left(\log(\hat{d})\left((L/\Delta)l_\Delta\hat{d} + ml_\Delta/P\right)\right)$	$O\left(\log\hat{d}\,(Lm/P + L\hat{d})\right)$	$O(\log(n)n^2/P)$	
	Work (CREW)	$O(\log(\hat{d})ml_\Delta)$	$O(Lm\log\hat{d})$	$O(\log(n)n^2)$	

Push vs. Pull: Formulations



OTHER ALGORITHMS & FORMULATIONS

Δ-Stepping

BFS

BC (algebra

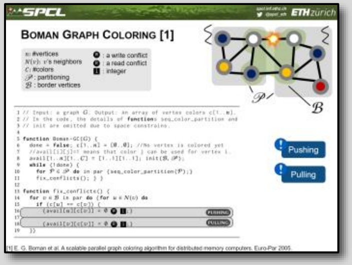
Betweenness Centrality

Boruvka MST

Check out the paper ☺

CONCLUSIONS

Push vs. Pull: Applicability



BOMAN GRAPH COLORING [1]

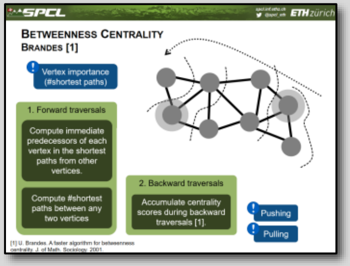
- n: #vertices
- $N(v)$: v's neighbors
- C: #colors
- $\mathcal{P}$: partitioning
- $\mathcal{B}$: border vertices

- ● : a write conflict
- ● : a read conflict
- ■ : integer

Pushing

Pulling

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.



BETWEENNESS CENTRALITY
BRANDES [1]

Vertex importance (#shortest paths)

1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices

2. Backward traversals

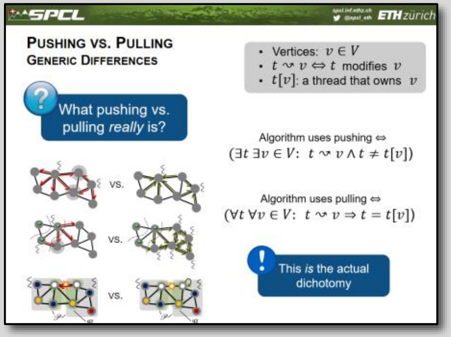
Accumulate centrality scores during backward traversals [1].

Pushing

Pulling

[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

Push vs. Pull: Dichotomy



PUSHING VS. PULLING
GENERIC DIFFERENCES

- Vertices: $v \in V$
- $t \rightsquigarrow v \Leftrightarrow t$ modifies v
- $t[v]$: a thread that owns v

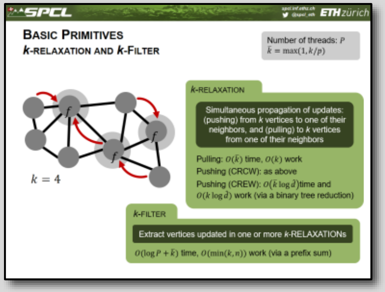
What pushing vs. pulling *really* is?

Algorithm uses pushing ⇔
$(\exists t\ \exists v \in V\colon\ t \rightsquigarrow v \land t \neq t[v])$

Algorithm uses pulling ⇔
$(\forall t\ \forall v \in V\colon\ t \rightsquigarrow v \Rightarrow t = t[v])$

This *is* the actual dichotomy

Push vs. Pull: Complexity



BASIC PRIMITIVES
k-RELAXATION AND *k*-FILTER

Number of threads: P
$\bar{k} = \max(1, k/p)$

k-RELAXATION

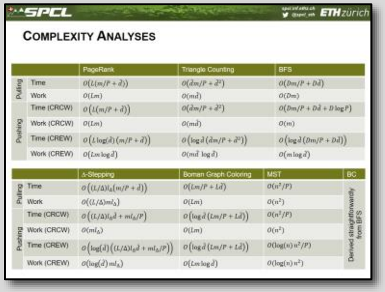
Simultaneous propagation of updates: (pushing) from k vertices to one of their neighbors, and (pulling) to k vertices from one of their neighbors

Pulling: $O(\bar{k})$ time, $O(k)$ work
Pushing (CRCW): as above
Pushing (CREW): $O(\bar{k} \log \hat{d})$time and $O(k \log \hat{d})$ work (via a binary tree reduction)

k-FILTER

Extract vertices updated in one or more *k*-RELAXATIONs

$O(\log P + \bar{k})$ time, $O(\min(k, n))$ work (via a prefix sum)

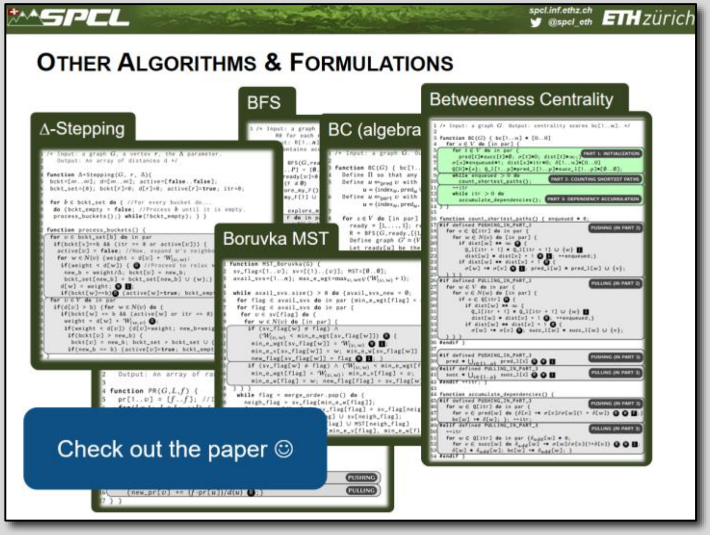


COMPLEXITY ANALYSES

		PageRank	Triangle Counting	BFS
Pulling	Time	$O(L(m/P+\hat{d}))$	$O(\hat{d}m/P+\hat{d}^2)$	$O(Dm/P+D\hat{d})$
	Work	$O(Lm)$	$O(m\hat{d})$	$O(Dm)$
Pushing	Time (CRCW)	$O\left(L(m/P+\hat{d})\right)$	$O(\hat{d}m/P+\hat{d}^2)$	$O(Dm/P+D\hat{d}+D\log P)$
	Work (CRCW)	$O(Lm)$	$O(m\hat{d})$	$O(m)$
	Time (CREW)	$O\left(L\log(\hat{d})\,(m/P+\hat{d})\right)$	$O\left(\log\hat{d}\,(\hat{d}m/P+\hat{d}^2)\right)$	$O\left(\log\hat{d}\,(Dm/P+D\hat{d})\right)$
	Work (CREW)	$O(Lm\log\hat{d})$	$O(m\hat{d}\log\hat{d})$	$O(m\log\hat{d})$

		Δ-Stepping	Boman Graph Coloring	MST	BC
Pulling	Time	$O\left((L/\Delta)l_\Delta(m/P+\hat{d})\right)$	$O(Lm/P+L\hat{d})$	$O(n^2/P)$	Derived straightforwardly from BFS
	Work	$O((L/\Delta)ml_\Delta)$	$O(Lm)$	$O(n^2)$	
Pushing	Time (CRCW)	$O\left((L/\Delta)l_\Delta\hat{d}+ml_\Delta/P\right)$	$O\left(\log\hat{d}\,(Lm/P+L\hat{d})\right)$	$O(n^2/P)$	
	Work (CRCW)	$O(ml_\Delta)$	$O(Lm)$	$O(n^2)$	
	Time (CREW)	$O\left(\log(\hat{d})\left((L/\Delta)l_\Delta\hat{d}+ml_\Delta/P\right)\right)$	$O\left(\log\hat{d}\,(Lm/P+L\hat{d})\right)$	$O(\log(n)\,n^2/P)$	
	Work (CREW)	$O(\log(\hat{d})\,ml_\Delta)$	$O(Lm\log\hat{d})$	$O(\log(n)\,n^2)$	

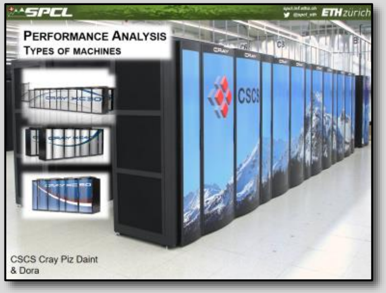
Push vs. Pull: Formulations



OTHER ALGORITHMS & FORMULATIONS

Check out the paper ☺

Performance & space analysis + guidelines

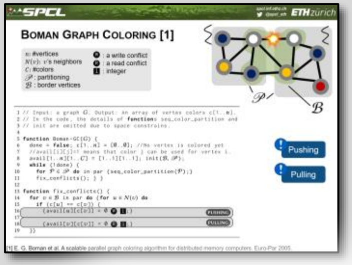


PERFORMANCE ANALYSIS
TYPES OF MACHINES

CSCS Cray Piz Daint & Dora

CONCLUSIONS

Push vs. Pull: Applicability



BOMAN GRAPH COLORING [1]

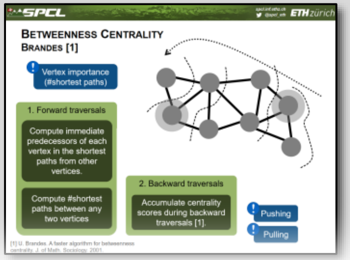
n: #vertices
$N(v)$: v's neighbors
C: #colors
$\mathscr{P}$: partitioning
$\mathscr{B}$: border vertices

W : a write conflict
R : a read conflict
I : integer

Pushing

Pulling

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.



BETWEENNESS CENTRALITY
BRANDES [1]

Vertex importance (#shortest paths)

1. Forward traversals

Compute immediate predecessors of each vertex in the shortest paths from other vertices.

Compute #shortest paths between any two vertices

2. Backward traversals

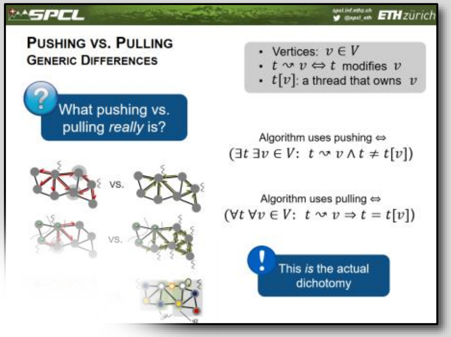
Accumulate centrality scores during backward traversals [1].

Pushing

Pulling

[1] U. Brandes. A faster algorithm for betweenness centrality. J. of Math. Sociology. 2001.

Push vs. Pull: Dichotomy



PUSHING VS. PULLING
GENERIC DIFFERENCES

- Vertices: $v \in V$
- $t \rightsquigarrow v \Leftrightarrow t$ modifies v
- $t[v]$: a thread that owns v

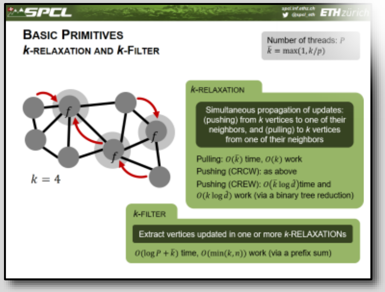
What pushing vs. pulling *really* is?

Algorithm uses pushing ⇔
$(\exists t\ \exists v \in V{:}\ \ t \rightsquigarrow v \wedge t \neq t[v])$

Algorithm uses pulling ⇔
$(\forall t\ \forall v \in V{:}\ \ t \rightsquigarrow v \Rightarrow t = t[v])$

This *is* the actual dichotomy

Push vs. Pull: Complexity



BASIC PRIMITIVES
***k*-RELAXATION AND *k*-FILTER**

Number of threads: P
$\bar{k} = \max(1, k/p)$

$k = 4$

k-RELAXATION

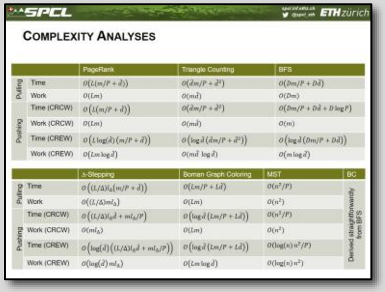
Simultaneous propagation of updates: (pushing) from *k* vertices to one of their neighbors, and (pulling) to *k* vertices from one of their neighbors

Pulling: $O(\bar{k})$ time, $O(k)$ work
Pushing (CRCW): as above
Pushing (CREW): $O(\bar{k}\log\hat{d})$time and $O(k\log\hat{d})$ work (via a binary tree reduction)

k-FILTER

Extract vertices updated in one or more *k*-RELAXATIONs

$O(\log P + \bar{k})$ time, $O(\min(k,n))$ work (via a prefix sum)

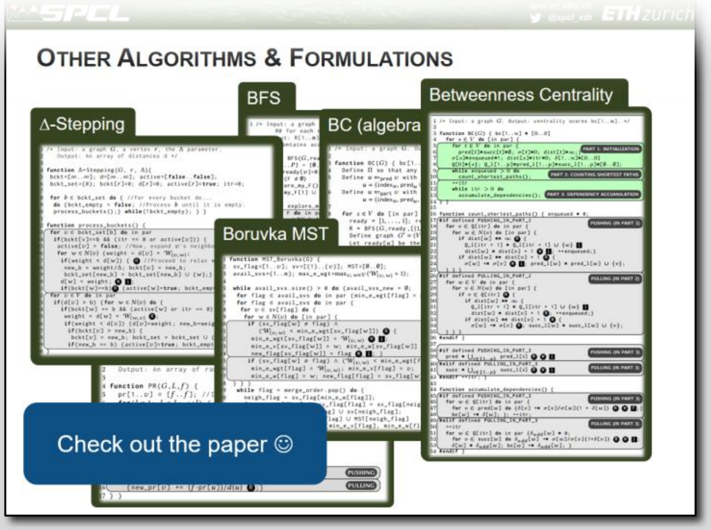


COMPLEXITY ANALYSES

		PageRank	Triangle Counting	BFS
Pulling	Time	$O(L(m/P+\hat{d}))$	$O(\hat{d}m/P+\hat{d}^2)$	$O(Dm/P+D\hat{d})$
	Work	$O(Lm)$	$O(m\hat{d})$	$O(Dm)$
Pushing	Time (CRCW)	$O(L(m/P+\hat{d}))$	$O(\hat{d}m/P+\hat{d}^2)$	$O(Dm/P+D\hat{d}+D\log P)$
	Work (CRCW)	$O(Lm)$	$O(m\hat{d})$	$O(m)$
	Time (CREW)	$O(L\log(\hat{d})(m/P+\hat{d}))$	$O(\log\hat{d}\,(\hat{d}m/P+\hat{d}^2))$	$O(\log\hat{d}\,(Dm/P+D\hat{d}))$
	Work (CREW)	$O(Lm\log\hat{d})$	$O(m\hat{d}\log\hat{d})$	$O(m\log\hat{d})$

		Δ-Stepping	Boman Graph Coloring	MST	BC
Pulling	Time	$O((L/\Delta)l_\Delta(m/P+\hat{d}))$	$O(Lm/P+L\hat{d})$	$O(n^2/P)$	Derived straightforwardly from BFS
	Work	$O((L/\Delta)ml_\Delta)$	$O(Lm)$	$O(n^2)$	
Pushing	Time (CRCW)	$O((L/\Delta)l_\Delta\hat{d}+ml_\Delta/P)$	$O(\log\hat{d}\,(Lm/P+L\hat{d}))$	$O(n^2/P)$	
	Work (CRCW)	$O(ml_\Delta)$	$O(Lm)$	$O(n^2)$	
	Time (CREW)	$O(\log(\hat{d})((L/\Delta)l_\Delta\hat{d}+ml_\Delta/P))$	$O(\log\hat{d}\,(Lm/P+L\hat{d}))$	$O(\log(n)\,n^2/P)$	
	Work (CREW)	$O(\log(\hat{d})\,ml_\Delta)$	$O(Lm\log\hat{d})$	$O(\log(n)\,n^2)$	

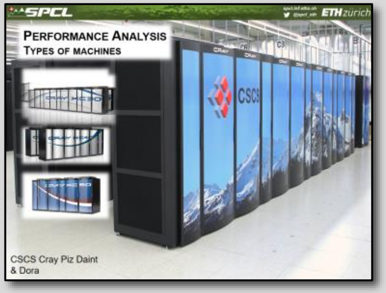
Thank you for your attention



OTHER ALGORITHMS & FORMULATIONS

Check out the paper ☺

Performance & space analysis + guidelines

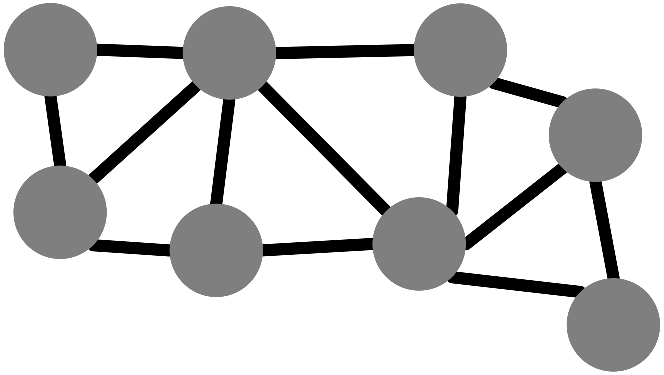


PERFORMANCE ANALYSIS
TYPES OF MACHINES

CSCS Cray Piz Daint & Dora

Backup slides

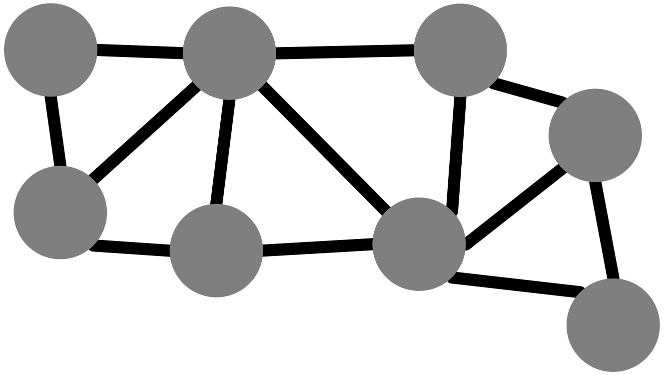
GRAPH COLORING
BOMAN ET AL. [1]



```
// Input: a graph G. Output: An array of vertex colors c[1..n].



function Boman-GC(G) {





                              }
```

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

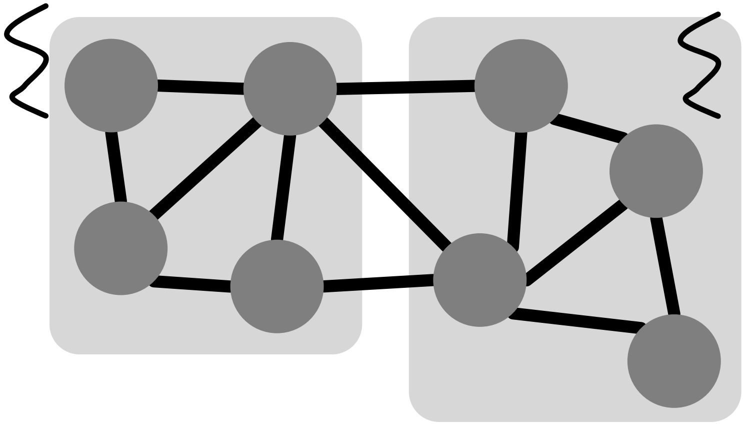


```
// Input: a graph G. Output: An array of vertex colors c[1..n].



function Boman-GC(G) {





                                }








```

#vertices

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

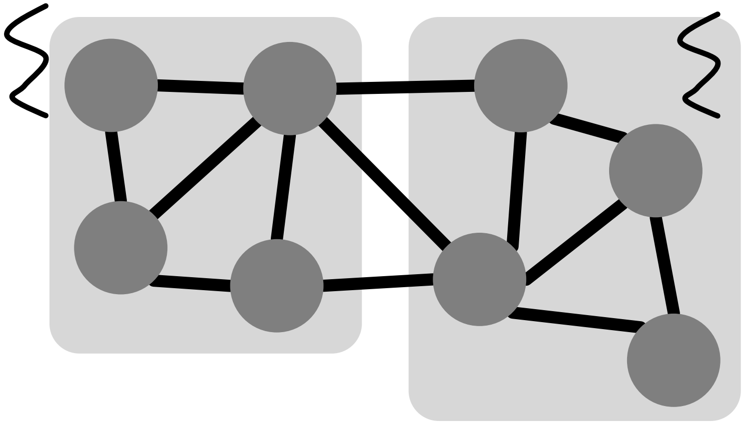


```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {





                        }
```

#vertices

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]



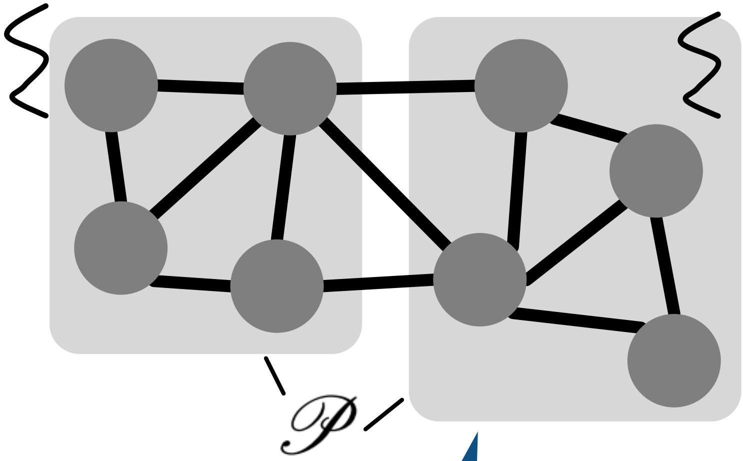
```
// Input: a graph G. Output: An array of vertex colors c[1..n].



function Boman-GC(G) {





                   }
```

#vertices

We care explicitly about partitioning now

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]



```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {





                             }
```

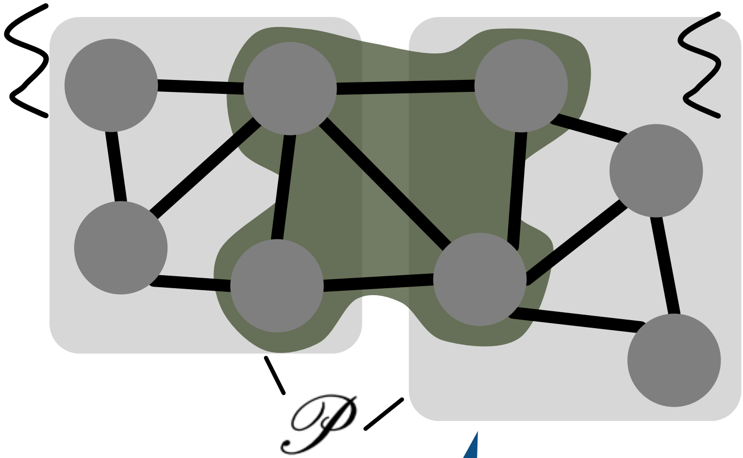
#vertices

We care explicitly about partitioning now

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING

BOMAN ET AL. [1]



```
// Input: a graph G. Output: An array of vertex colors c[1..n].



function Boman-GC(G) {





                        }
```

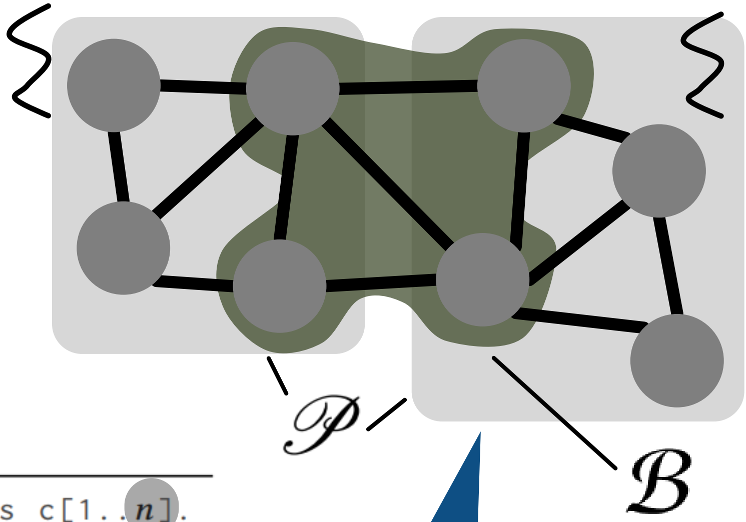
#vertices

We care explicitly about partitioning now

---

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]



```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {





                          }
```

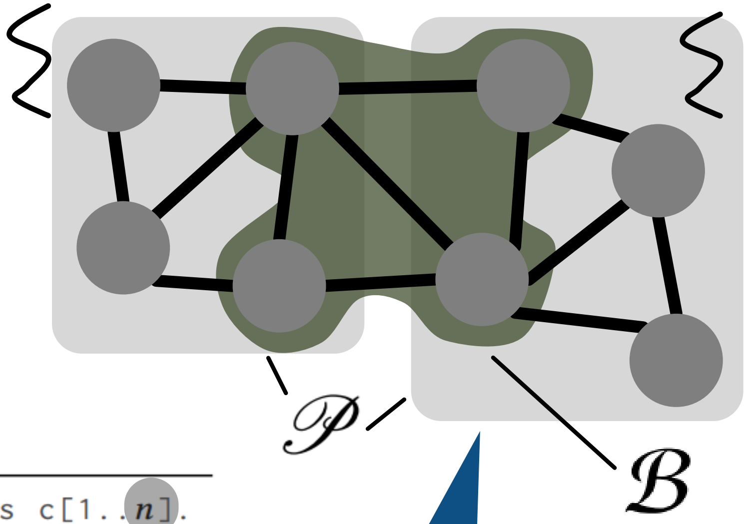
#vertices

We care explicitly about partitioning now

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {





                                }
```

#vertices

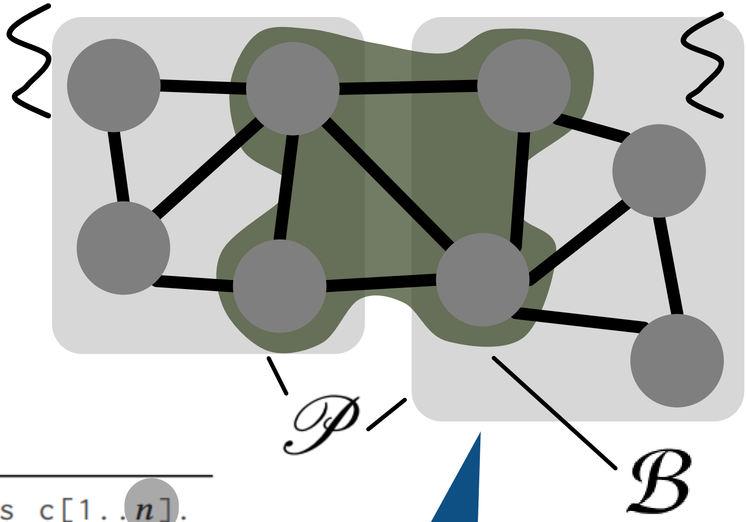
[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer

```
// Input: a graph G. Output: An array of vertex colors c[1..n].



function Boman-GC(G) {
  done = false; c[1..n] = [∅..∅]; //No vertex is colored yet





                     }
```

#vertices



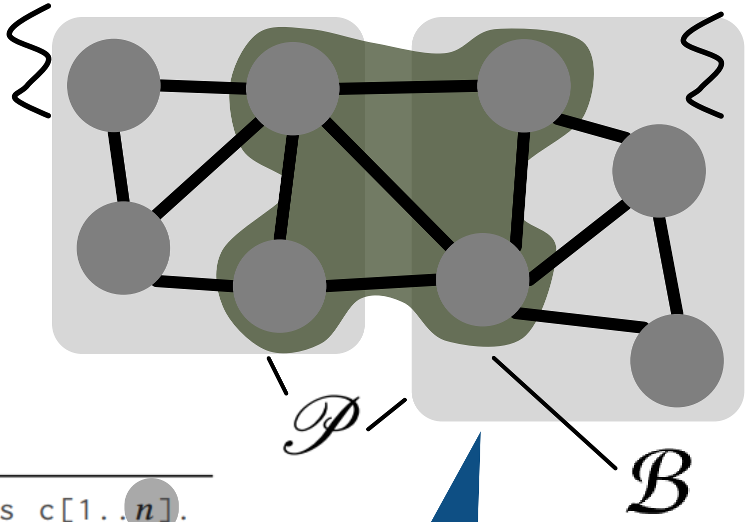
We care explicitly about partitioning now

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING

BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

```
// Input: a graph G. Output: An array of vertex colors c[1..n].



function Boman-GC(G) {
  done = false; c[1..n] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.



                             }
```

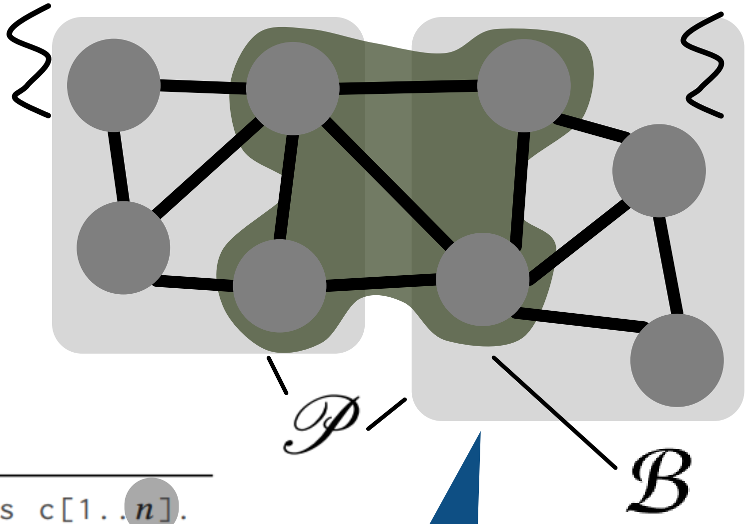
#vertices

---

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

W : a write conflict
R : a read conflict
i : integer



```
// Input: a graph G. Output: An array of vertex colors c[1..n].



function Boman-GC(G) {
  done = false; c[1..n] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..n][1..C] = [1..1][1..1];


                             }
```

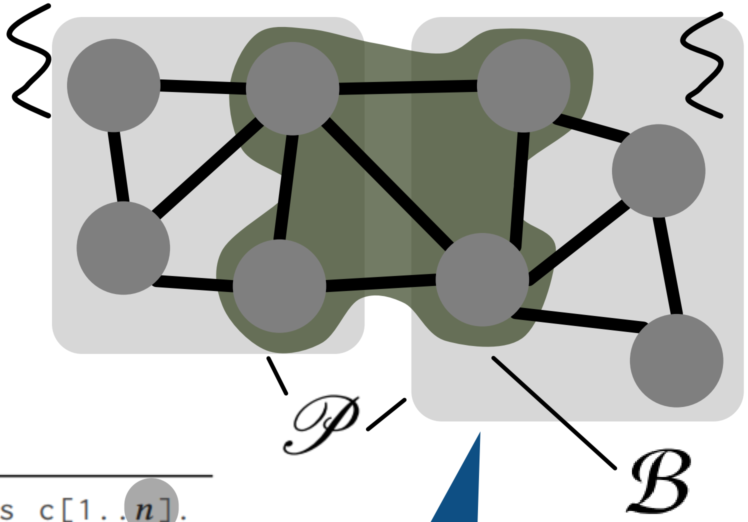
#vertices

We care explicitly about partitioning now

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

- W : a write conflict
- R : a read conflict
- i : integer



We care explicitly about partitioning now

maximum #colors

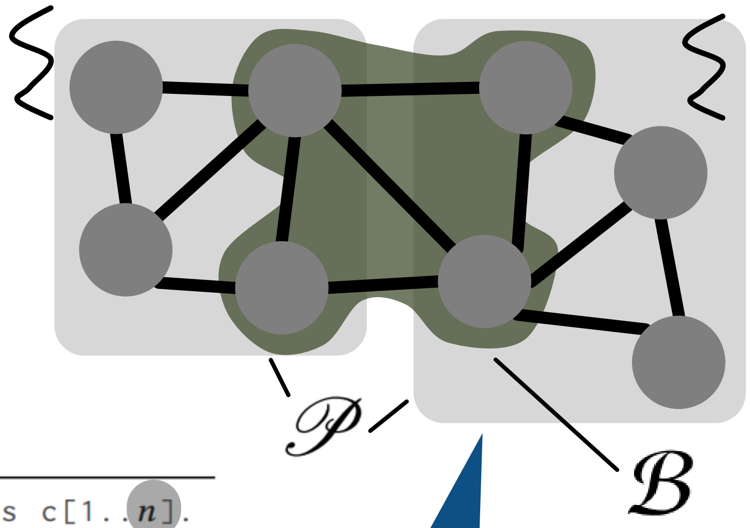
#vertices

```
// Input: a graph G. Output: An array of vertex colors c[1..n].



function Boman-GC(G) {
  done = false; c[1..n] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..n][1..C] = [1..1][1..1];


                              }
```

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathcal{B}$, $\mathscr{P}$);


                         }
```

#vertices

maximum #colors

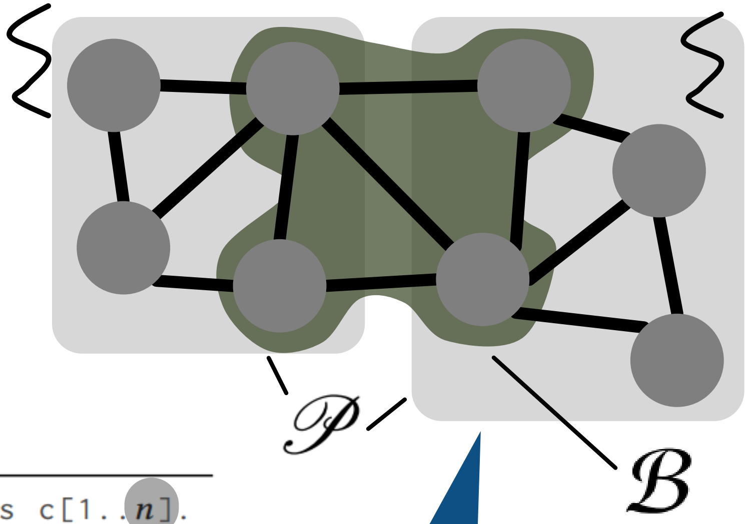
We care explicitly about partitioning now

---

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

```
// Input: a graph G. Output: An array of vertex colors c[1..n].



function Boman-GC(G) {
  done = false; c[1..n] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..n][1..C] = [1..1][1..1]; init(ℬ, 𝒫);
  while (!done) {

                        } }
```

maximum #colors

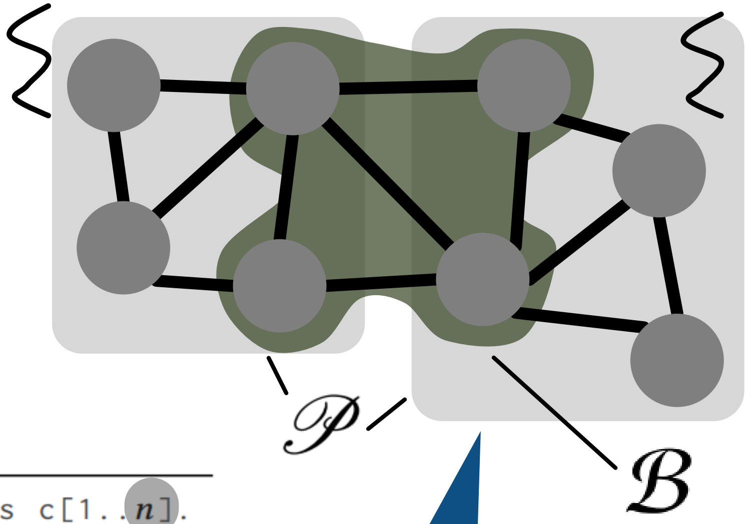
#vertices

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING

BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

maximum #colors

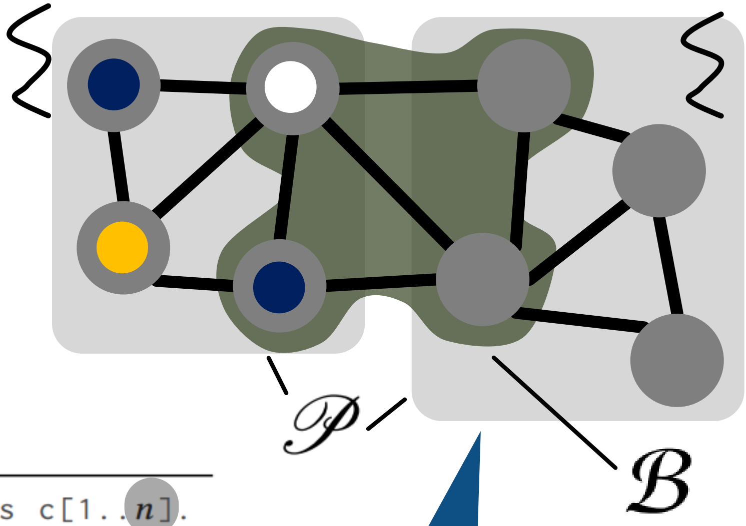
#vertices

```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathcal{B}$, $\mathscr{P}$);
  while (!done) {
    for $\mathcal{P} \in \mathscr{P}$ do in par {seq_color_partition($\mathcal{P}$);}
                         } }
```

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

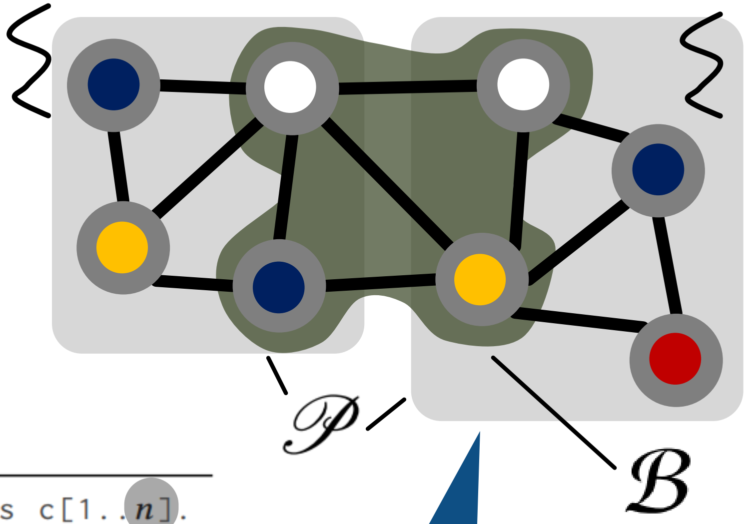
#vertices

maximum #colors

```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [$\emptyset$..$\emptyset$]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathcal{B}$, $\mathscr{P}$);
  while (!done) {
    for $\mathcal{P} \in \mathscr{P}$ do in par {seq_color_partition($\mathcal{P}$);}
                    } }
```

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]



Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer

```
// Input: a graph G. Output: An array of vertex colors c[1..n].



function Boman-GC(G) {
  done = false; c[1..n] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..n][1..C] = [1..1][1..1]; init(ℬ, 𝒫);
  while (!done) {
    for 𝓟 ∈ 𝒫 do in par {seq_color_partition(𝓟);}
                    } }
```

maximum #colors

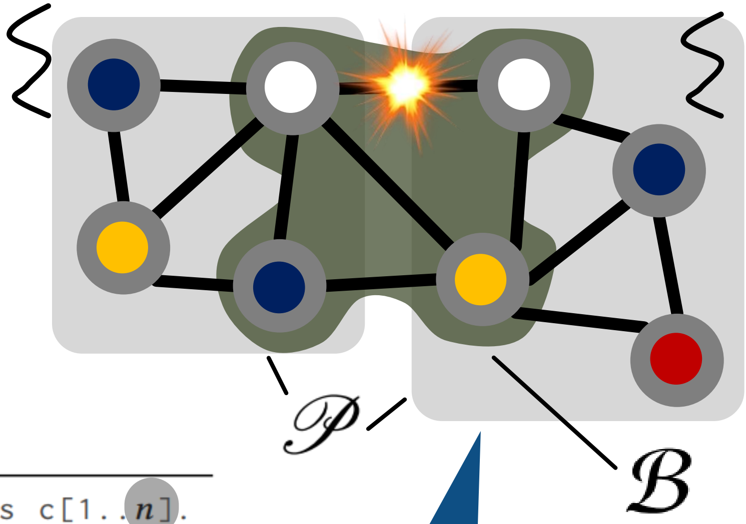
#vertices

We care explicitly about partitioning now

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

```
// Input: a graph G. Output: An array of vertex colors c[1..n].



function Boman-GC(G) {
  done = false; c[1..n] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..n][1..C] = [1..1][1..1]; init(ℬ, 𝒫);
  while (!done) {
    for P ∈ 𝒫 do in par {seq_color_partition(P);}
                        } }
```

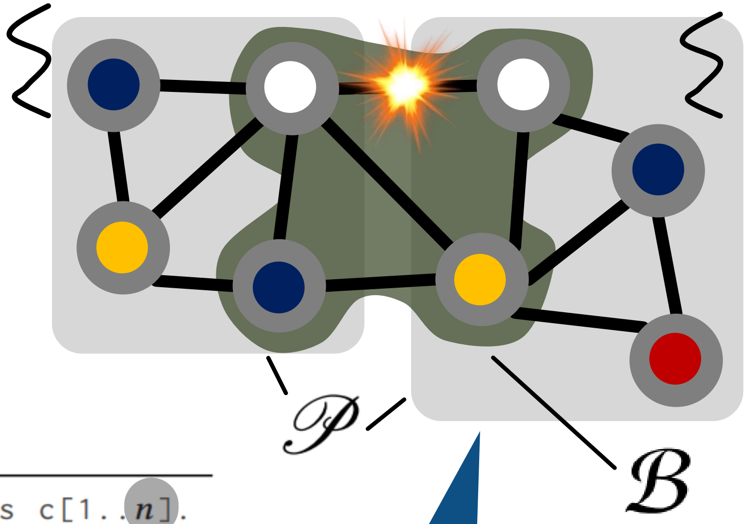
maximum #colors

#vertices

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

W : a write conflict
R : a read conflict
i : integer



We care explicitly about partitioning now

```
// Input: a graph G. Output: An array of vertex colors c[1..n].



function Boman-GC(G) {
  done = false; c[1..n] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..n][1..C] = [1..1][1..1]; init(ℬ, 𝒫);
  while (!done) {
    for 𝑃 ∈ 𝒫 do in par {seq_color_partition(𝑃);}
    fix_conflicts(); } }
```

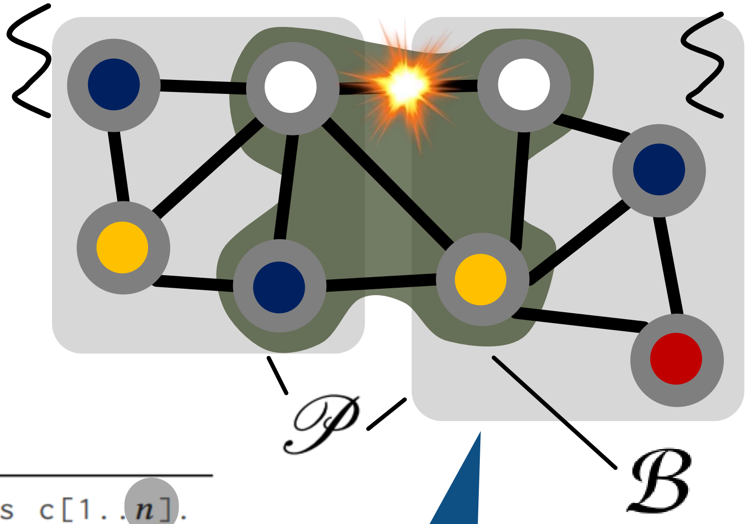
#vertices

maximum #colors

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

W : a write conflict
R : a read conflict
i : integer



We care explicitly about partitioning now

```
// Input: a graph G. Output: An array of vertex colors c[1..n].



function Boman-GC(G) {
  done = false; c[1..n] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..n][1..C] = [1..1][1..1]; init(ℬ, 𝒫);
  while (!done) {
    for P ∈ 𝒫 do in par {seq_color_partition(P);}
    fix_conflicts(); } }

function fix_conflicts() {






   }
```

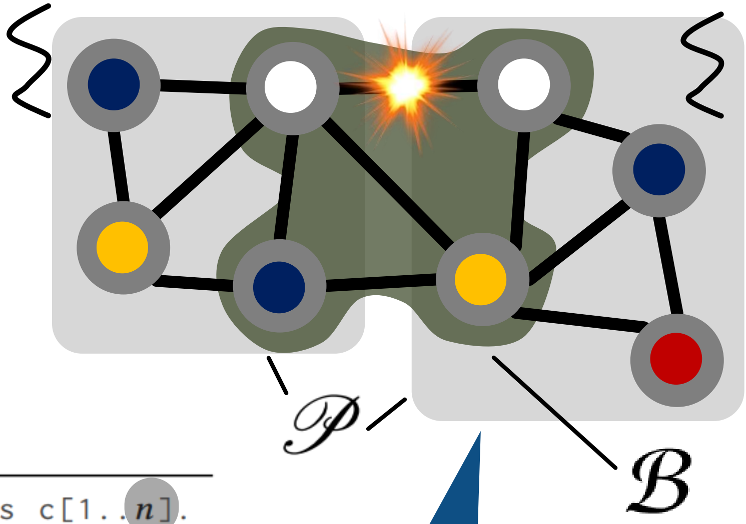
maximum #colors

#vertices

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

- Ⓦ : a write conflict
- Ⓡ : a read conflict
- i : integer



We care explicitly about partitioning now

```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [$\emptyset$..$\emptyset$]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathcal{B}$, $\mathscr{P}$);
  while (!done) {
    for $\mathcal{P} \in \mathscr{P}$ do in par {seq_color_partition($\mathcal{P}$);}
    fix_conflicts(); } }

function fix_conflicts() {
  for $v \in \mathcal{B}$ in par do {





  }}
```

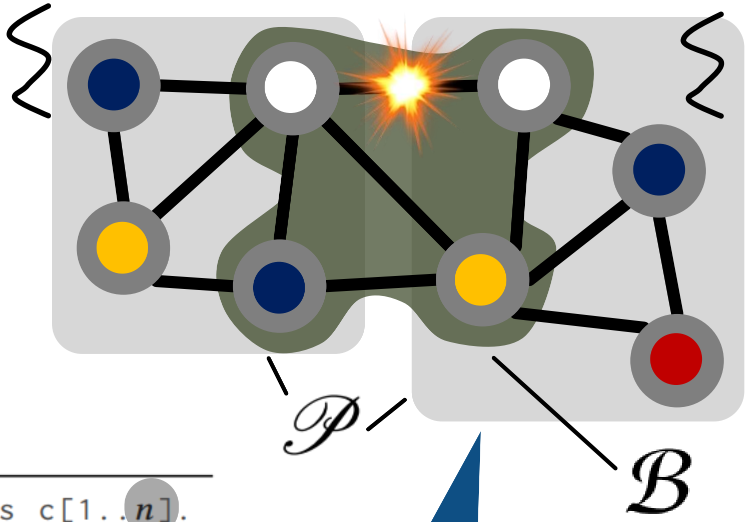
#vertices

maximum #colors

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

maximum #colors

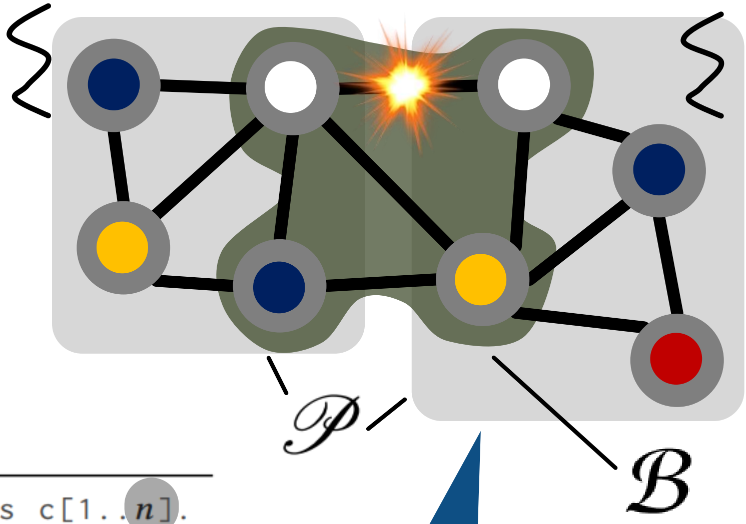
#vertices

```
// Input: a graph G. Output: An array of vertex colors c[1..n].



function Boman-GC(G) {
  done = false; c[1..n] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..n][1..C] = [1..1][1..1]; init(ℬ, 𝒫);
  while (!done) {
    for P ∈ 𝒫 do in par {seq_color_partition(P);}
    fix_conflicts(); } }

function fix_conflicts() {
  for v ∈ ℬ in par do {for u ∈ N(v) do




  }}
```

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

```
// Input: a graph G. Output: An array of vertex colors c[1..n].



function Boman-GC(G) {
  done = false; c[1..n] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..n][1..C] = [1..1][1..1]; init(ℬ, 𝒫);
  while (!done) {
    for P ∈ 𝒫 do in par {seq_color_partition(P);}
    fix_conflicts(); } }

function fix_conflicts() {
  for v ∈ ℬ in par do {for u ∈ N(v) do




  }}
```

#vertices

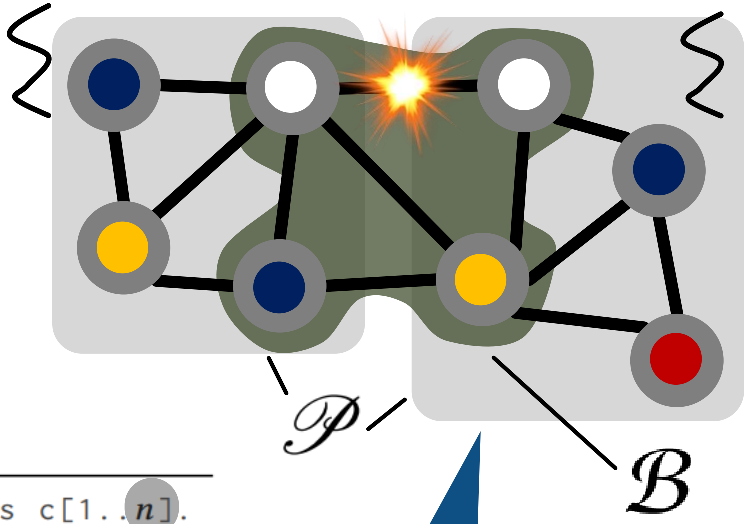
maximum #colors

v's neighbors

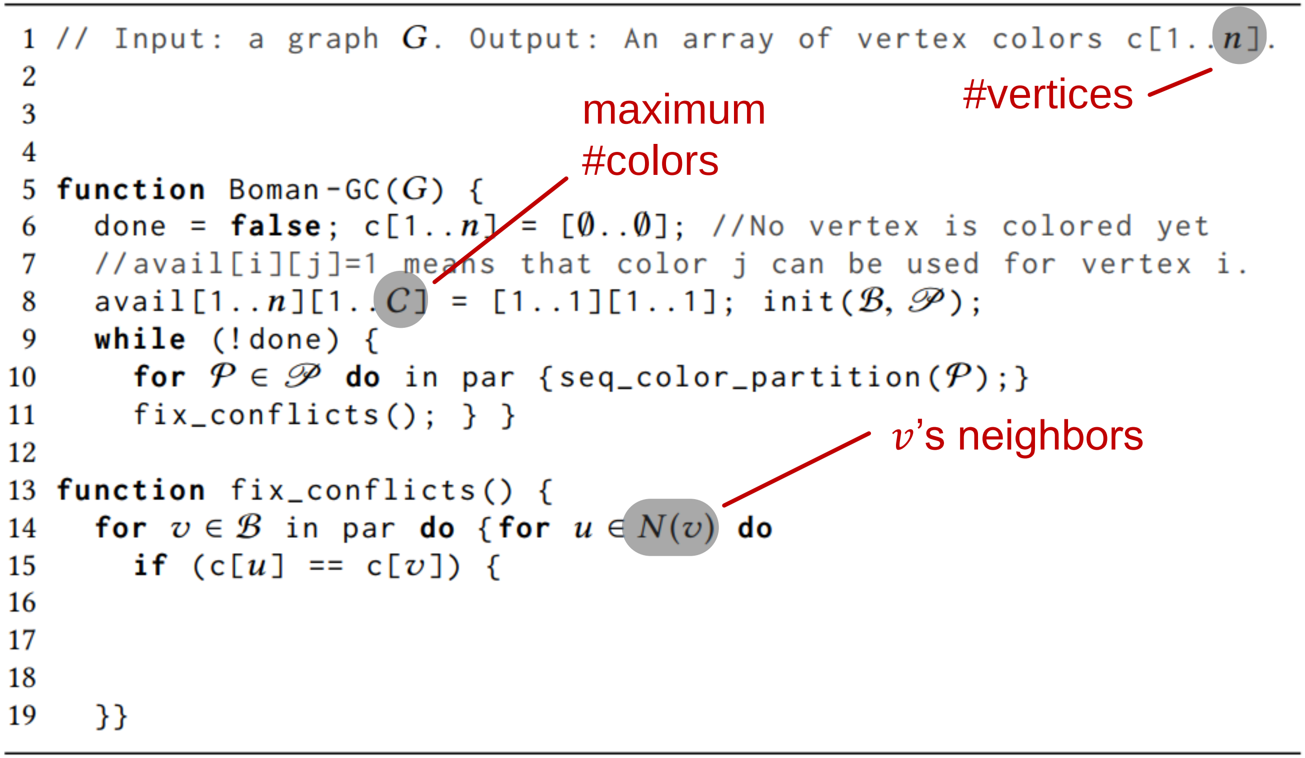
[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

W : a write conflict
R : a read conflict
i : integer



We care explicitly about partitioning now



```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathcal{B}$, $\mathscr{P}$);
  while (!done) {
    for $\mathcal{P} \in \mathscr{P}$ do in par {seq_color_partition($\mathcal{P}$);}
    fix_conflicts(); } }

function fix_conflicts() {
  for $v \in \mathcal{B}$ in par do {for $u \in N(v)$ do
    if (c[$u$] == c[$v$]) {



  }}
```

#vertices

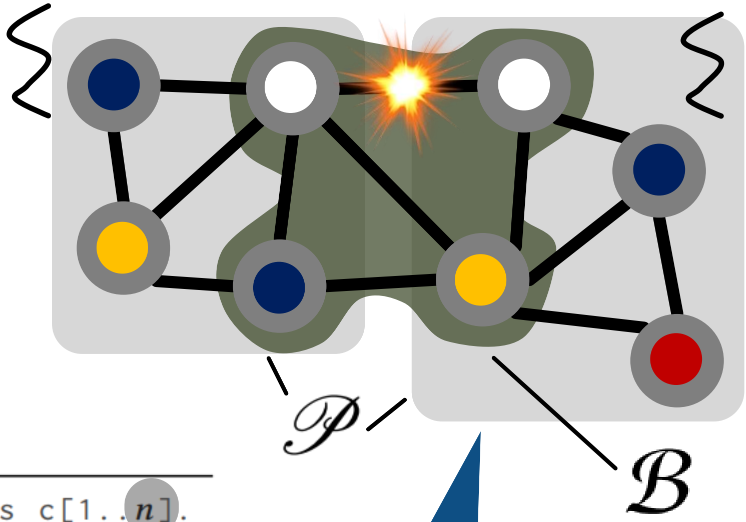
maximum #colors

v's neighbors

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [$\emptyset$..$\emptyset$]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathcal{B}$, $\mathscr{P}$);
  while (!done) {
    for $\mathcal{P} \in \mathscr{P}$ do in par {seq_color_partition($\mathcal{P}$);}
    fix_conflicts(); } }

function fix_conflicts() {
  for $v \in \mathcal{B}$ in par do {for $u \in N(v)$ do
    if (c[$u$] == c[$v$]) {
      {avail[$u$][c[$v$]] = $\emptyset$ Ⓦ i;}                    PUSHING

      {avail[$v$][c[$v$]] = $\emptyset$ Ⓡ i;}                    PULLING
  }}
```

#vertices

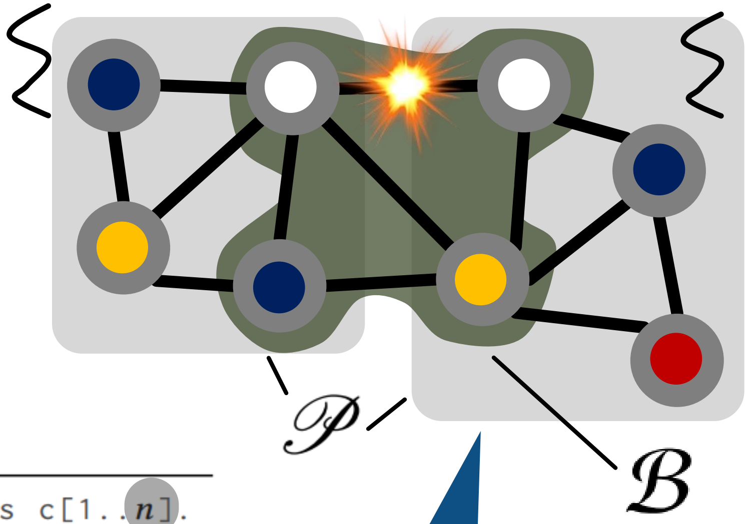
maximum #colors

v's neighbors

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

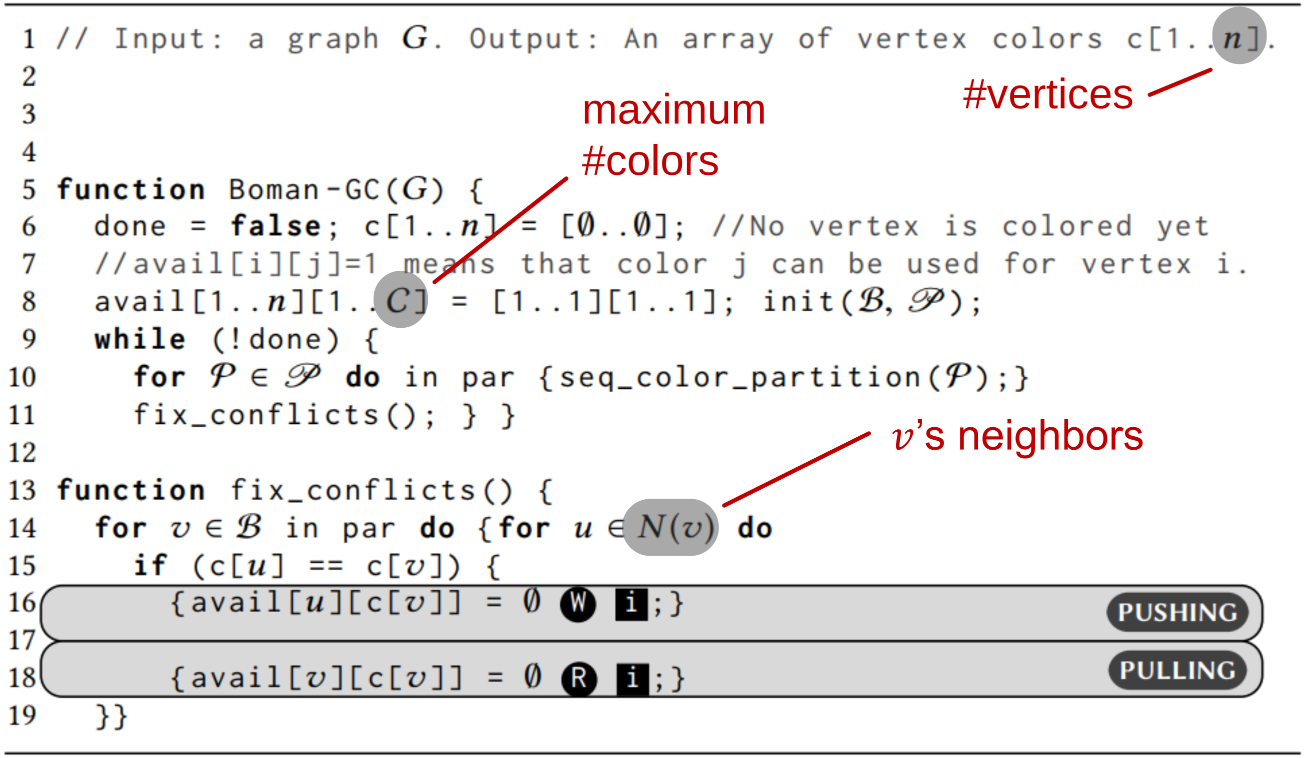
GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

Pushing



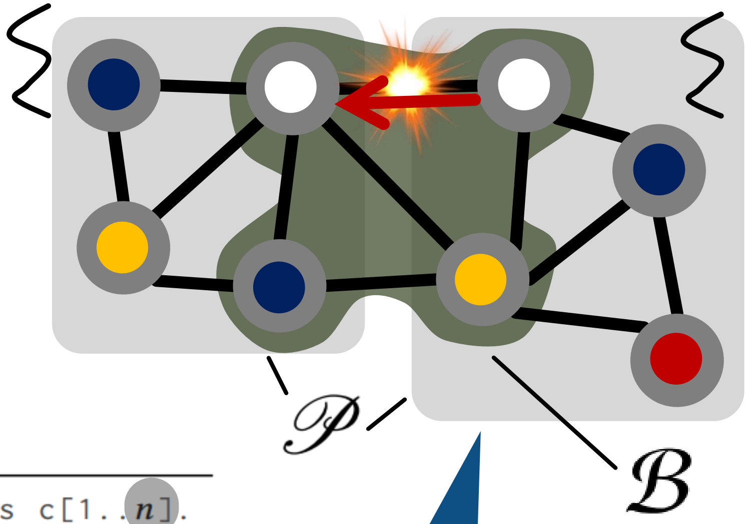
```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathscr{B}$, $\mathscr{P}$);
  while (!done) {
    for $\mathcal{P} \in \mathscr{P}$ do in par {seq_color_partition($\mathcal{P}$);}
    fix_conflicts(); } }

function fix_conflicts() {
  for $v \in \mathscr{B}$ in par do {for $u \in N(v)$ do
    if (c[$u$] == c[$v$]) {
      {avail[$u$][c[$v$]] = ∅ Ⓦ i;}                PUSHING

      {avail[$v$][c[$v$]] = ∅ Ⓡ i;}                PULLING
  }}
```

---

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

Pushing

```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [$\emptyset$..$\emptyset$]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathscr{B}$, $\mathscr{P}$);
  while (!done) {
    for $\mathcal{P} \in \mathscr{P}$ do in par {seq_color_partition($\mathcal{P}$);}
    fix_conflicts(); } }

function fix_conflicts() {
  for $v \in \mathscr{B}$ in par do {for $u \in N(v)$ do
    if (c[$u$] == c[$v$]) {
      {avail[$u$][c[$v$]] = $\emptyset$ Ⓦ i;}                      PUSHING

      {avail[$v$][c[$v$]] = $\emptyset$ Ⓡ i;}                      PULLING
  }}
```

maximum #colors

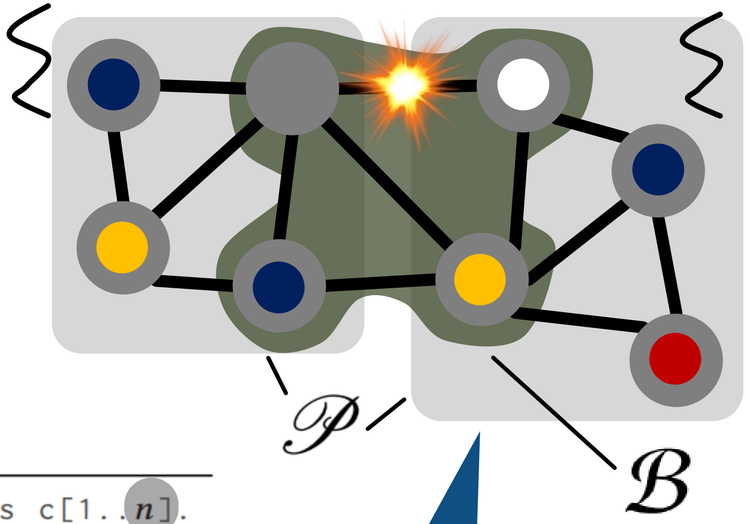
#vertices

v's neighbors

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathcal{B}$, $\mathcal{P}$);
  while (!done) {
    for $P \in \mathcal{P}$ do in par {seq_color_partition($P$);}
    fix_conflicts(); } }

function fix_conflicts() {
  for $v \in \mathcal{B}$ in par do {for $u \in N(v)$ do
    if (c[$u$] == c[$v$]) {
      {avail[$u$][c[$v$]] = ∅ Ⓦ i;}                    PUSHING

      {avail[$v$][c[$v$]] = ∅ Ⓡ i;}                    PULLING
  }}
```

#vertices

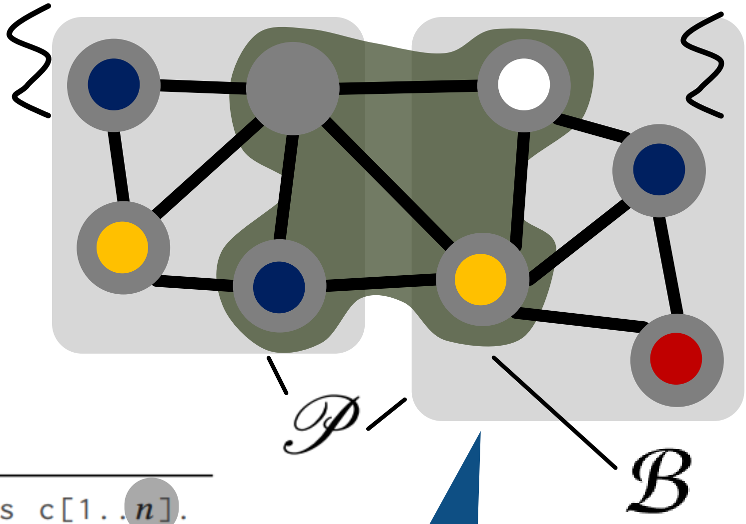
maximum #colors

v's neighbors

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathcal{B}$, $\mathscr{P}$);
  while (!done) {
    for $\mathcal{P}$ ∈ $\mathscr{P}$ do in par {seq_color_partition($\mathcal{P}$);}
    fix_conflicts(); } }

function fix_conflicts() {
  for $v$ ∈ $\mathcal{B}$ in par do {for $u$ ∈ $N(v)$ do
    if (c[$u$] == c[$v$]) {
      {avail[$u$][c[$v$]] = ∅ Ⓦ i;}                    PUSHING

      {avail[$v$][c[$v$]] = ∅ Ⓡ i;}                    PULLING
  }}
```

#vertices

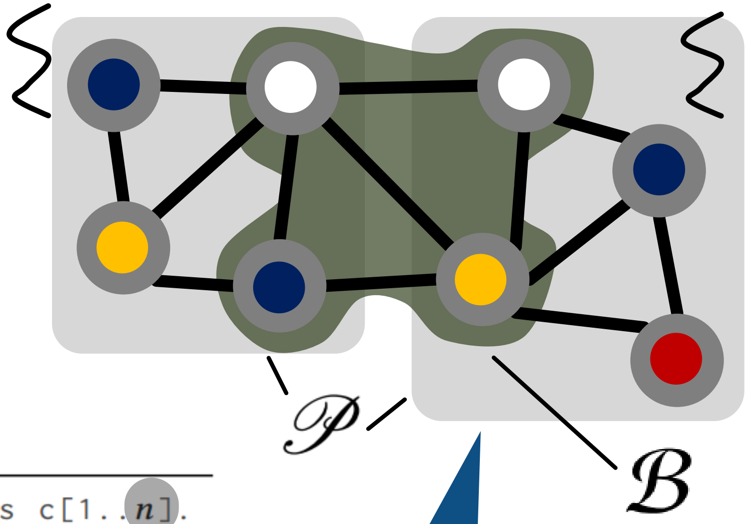
maximum #colors

v's neighbors

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [$\emptyset$..$\emptyset$]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathcal{B}$, $\mathscr{P}$);
  while (!done) {
    for $\mathcal{P} \in \mathscr{P}$ do in par {seq_color_partition($\mathcal{P}$);}
    fix_conflicts(); } }

function fix_conflicts() {
  for $v \in \mathcal{B}$ in par do {for $u \in N(v)$ do
    if (c[$u$] == c[$v$]) {
      {avail[$u$][c[$v$]] = $\emptyset$ Ⓦ i;}                    PUSHING

      {avail[$v$][c[$v$]] = $\emptyset$ Ⓡ i;}                    PULLING
  }}
```

#vertices

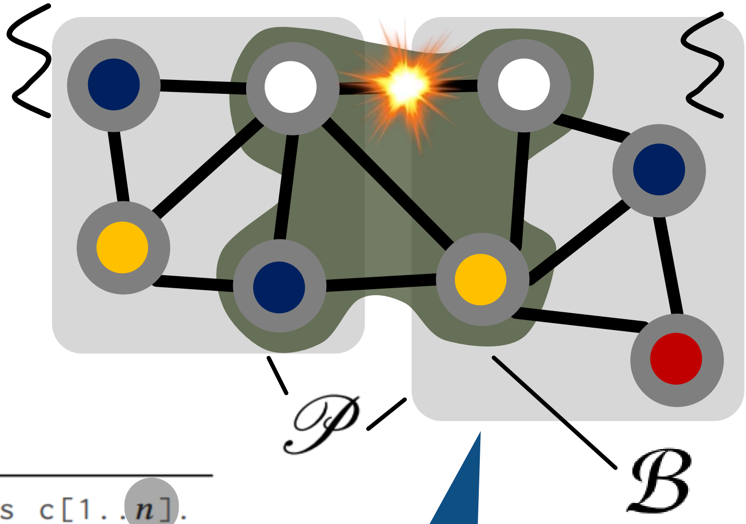
maximum #colors

v's neighbors

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING

BOMAN ET AL. [1]



We care explicitly about partitioning now

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer

```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [∅..∅]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathcal{B}$, $\mathscr{P}$);
  while (!done) {
    for $\mathcal{P} \in \mathscr{P}$ do in par {seq_color_partition($\mathcal{P}$);}
    fix_conflicts(); } }

function fix_conflicts() {
  for $v \in \mathcal{B}$ in par do {for $u \in N(v)$ do
    if (c[$u$] == c[$v$]) {
      {avail[$u$][c[$v$]] = ∅ Ⓦ i;}                                PUSHING

      {avail[$v$][c[$v$]] = ∅ Ⓡ i;}                                PULLING
  }}
```

#vertices

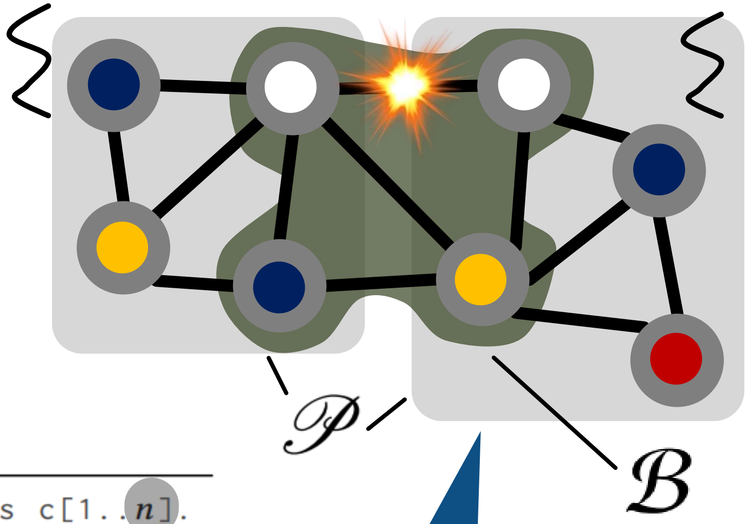
maximum #colors

v's neighbors

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [$\emptyset$..$\emptyset$]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathcal{B}$, $\mathscr{P}$);
  while (!done) {
    for $\mathcal{P} \in \mathscr{P}$ do in par {seq_color_partition($\mathcal{P}$);}
    fix_conflicts(); } }

function fix_conflicts() {
  for $v \in \mathcal{B}$ in par do {for $u \in N(v)$ do
    if (c[$u$] == c[$v$]) {
      {avail[$u$][c[$v$]] = $\emptyset$ Ⓦ i;}          PUSHING

      {avail[$v$][c[$v$]] = $\emptyset$ Ⓡ i;}          PULLING
  }}
```

#vertices

maximum #colors

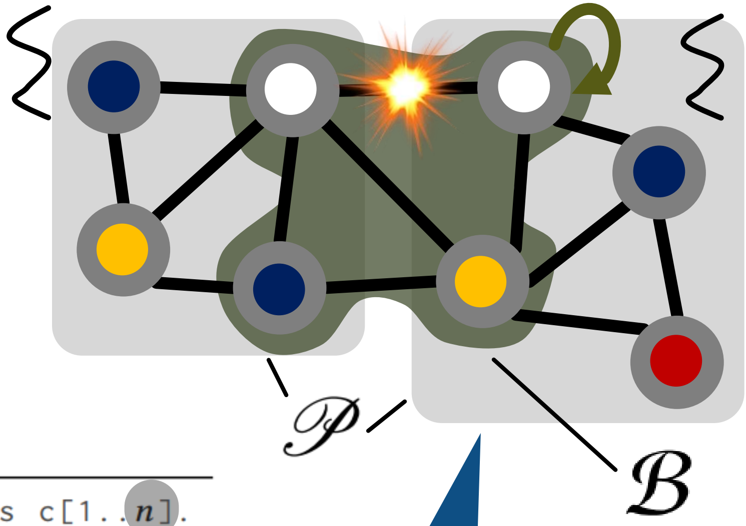
v's neighbors

Pulling

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [$\emptyset$..$\emptyset$]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathcal{B}$, $\mathscr{P}$);
  while (!done) {
    for $\mathcal{P} \in \mathscr{P}$ do in par {seq_color_partition($\mathcal{P}$);}
    fix_conflicts(); } }

function fix_conflicts() {
  for $v \in \mathcal{B}$ in par do {for $u \in N(v)$ do
    if (c[$u$] == c[$v$]) {
      {avail[$u$][c[$v$]] = $\emptyset$ Ⓦ i;}                    PUSHING

      {avail[$v$][c[$v$]] = $\emptyset$ Ⓡ i;}                    PULLING
  }}
```

#vertices

maximum #colors

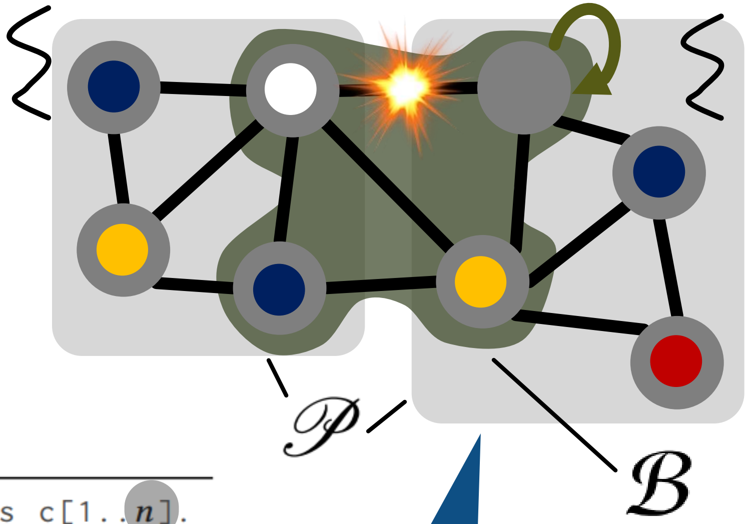
v's neighbors

Pulling

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

W : a write conflict
R : a read conflict
i : integer



We care explicitly about partitioning now

```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [$\emptyset$..$\emptyset$]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathcal{B}$, $\mathscr{P}$);
  while (!done) {
    for $\mathcal{P} \in \mathscr{P}$ do in par {seq_color_partition($\mathcal{P}$);}
    fix_conflicts(); } }

function fix_conflicts() {
  for $v \in \mathcal{B}$ in par do {for $u \in N(v)$ do
    if (c[$u$] == c[$v$]) {
      {avail[$u$][c[$v$]] = $\emptyset$ W i;}                      PUSHING

      {avail[$v$][c[$v$]] = $\emptyset$ R i;}                      PULLING
  }}
```

#vertices

maximum #colors

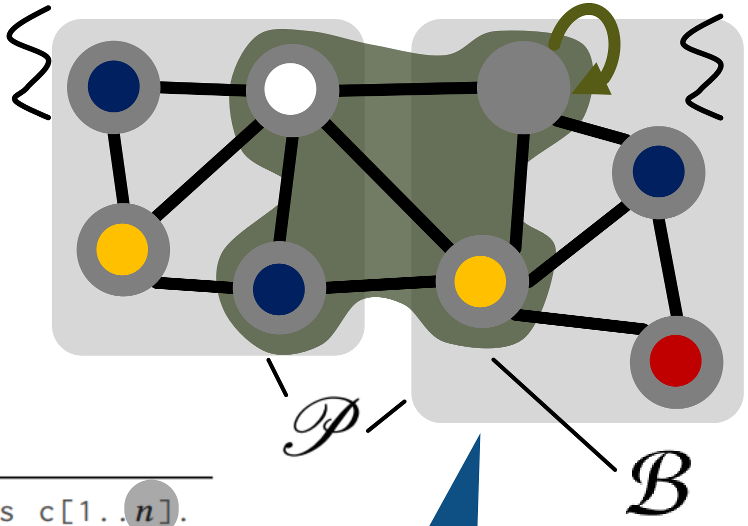
v's neighbors

Pulling

[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

GRAPH COLORING
BOMAN ET AL. [1]

Ⓦ : a write conflict
Ⓡ : a read conflict
i : integer



We care explicitly about partitioning now

```
// Input: a graph $G$. Output: An array of vertex colors c[1..$n$].



function Boman-GC($G$) {
  done = false; c[1..$n$] = [$\emptyset$..$\emptyset$]; //No vertex is colored yet
  //avail[i][j]=1 means that color j can be used for vertex i.
  avail[1..$n$][1..$C$] = [1..1][1..1]; init($\mathcal{B}$, $\mathscr{P}$);
  while (!done) {
    for $\mathcal{P} \in \mathscr{P}$ do in par {seq_color_partition($\mathcal{P}$);}
    fix_conflicts(); } }

function fix_conflicts() {
  for $v \in \mathcal{B}$ in par do {for $u \in N(v)$ do
    if (c[$u$] == c[$v$]) {
      {avail[$u$][c[$v$]] = $\emptyset$ Ⓦ i;}                    PUSHING

      {avail[$v$][c[$v$]] = $\emptyset$ Ⓡ i;}                    PULLING
  }}
```

#vertices

maximum #colors

v's neighbors

Pulling

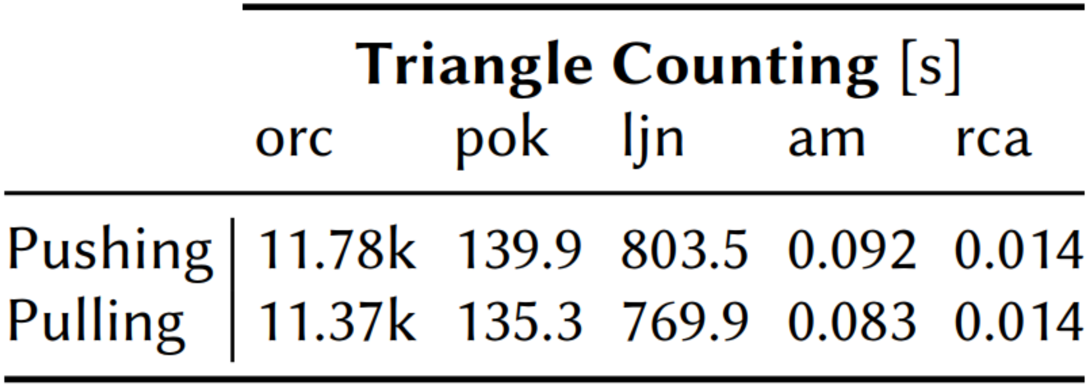
[1] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. Euro-Par 2005.

PERFORMANCE ANALYSIS

TRIANGLE COUNTING

PERFORMANCE ANALYSIS

TRIANGLE COUNTING



	Triangle Counting [s]				
	orc	pok	ljn	am	rca
Pushing	11.78k	139.9	803.5	0.092	0.014
Pulling	11.37k	135.3	769.9	0.083	0.014

PERFORMANCE ANALYSIS

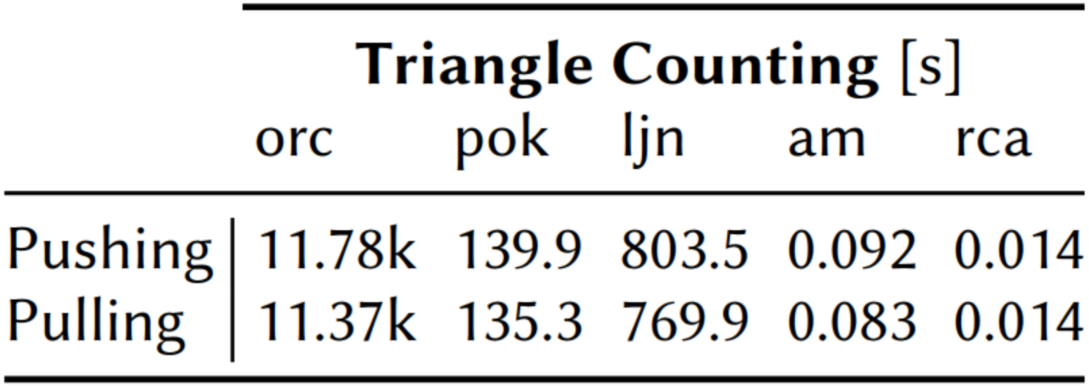
TRIANGLE COUNTING

SNAP

orc, pok, ljn: social networks

rca: road network

am: amazon graph



Triangle Counting [s]

	orc	pok	ljn	am	rca
Pushing	11.78k	139.9	803.5	0.092	0.014
Pulling	11.37k	135.3	769.9	0.083	0.014

PERFORMANCE ANALYSIS

TRIANGLE COUNTING

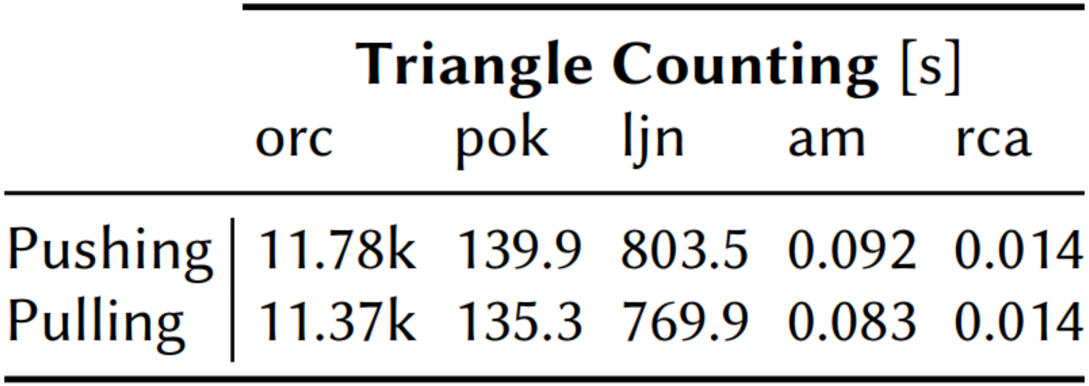
SNAP

orc, pok, ljn: social networks

rca: road network

am: amazon graph

Shared-Memory



	Triangle Counting [s]				
	orc	pok	ljn	am	rca
Pushing	11.78k	139.9	803.5	0.092	0.014
Pulling	11.37k	135.3	769.9	0.083	0.014

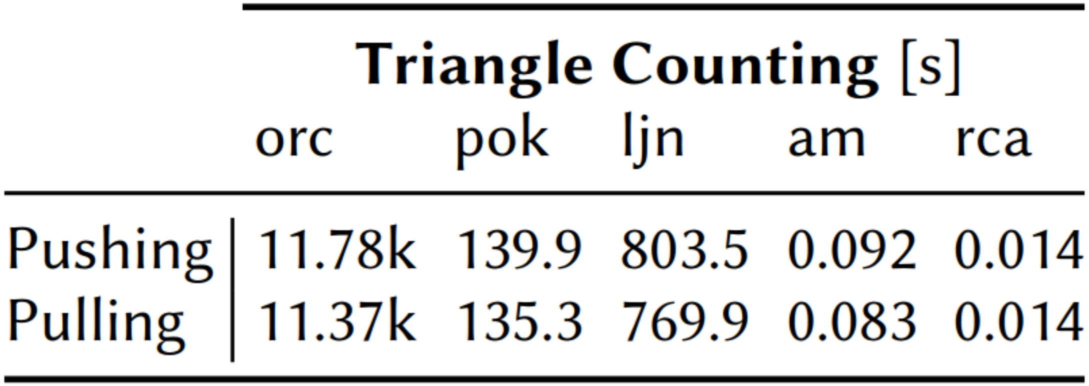
PERFORMANCE ANALYSIS

TRIANGLE COUNTING

SNAP
orc, pok, ljn: social networks
rca: road network
am: amazon graph

Shared-Memory

Pulling faster



	Triangle Counting [s]				
	orc	pok	ljn	am	rca
Pushing	11.78k	139.9	803.5	0.092	0.014
Pulling	11.37k	135.3	769.9	0.083	0.014

PERFORMANCE ANALYSIS

TRIANGLE COUNTING

orc, pok, ljn: social networks
rca: road network
am: amazon graph

Shared-Memory

Pulling faster

Fewer cache misses

No atomics

	Triangle Counting [s]				
	orc	pok	ljn	am	rca
Pushing	11.78k	139.9	803.5	0.092	0.014
Pulling	11.37k	135.3	769.9	0.083	0.014

PERFORMANCE ANALYSIS
BOMAN GRAPH COLORING + GRS + FE

orc, ljn: social networks
rca: road network

Shared-Memory

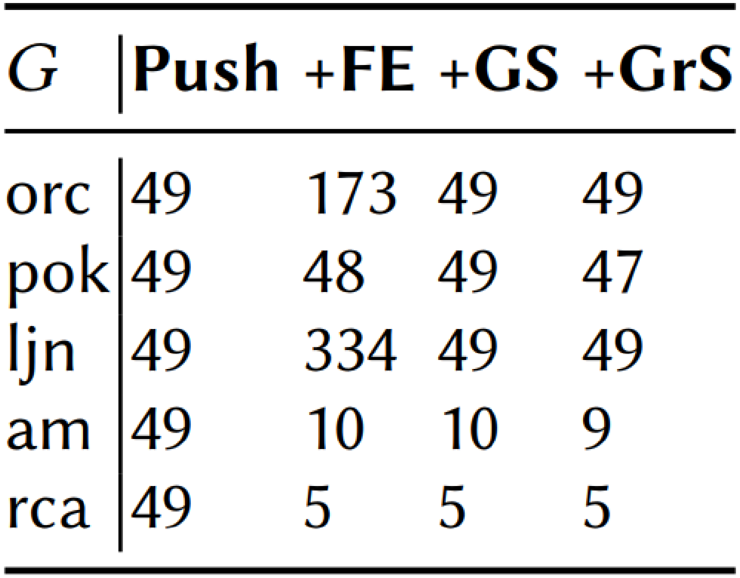
GrS+FE: Greedy-Switch + Frontier-Exploit
GS: Generic-Switch

!

Performance improvements

Fewer iterations

Fewer reads/writes



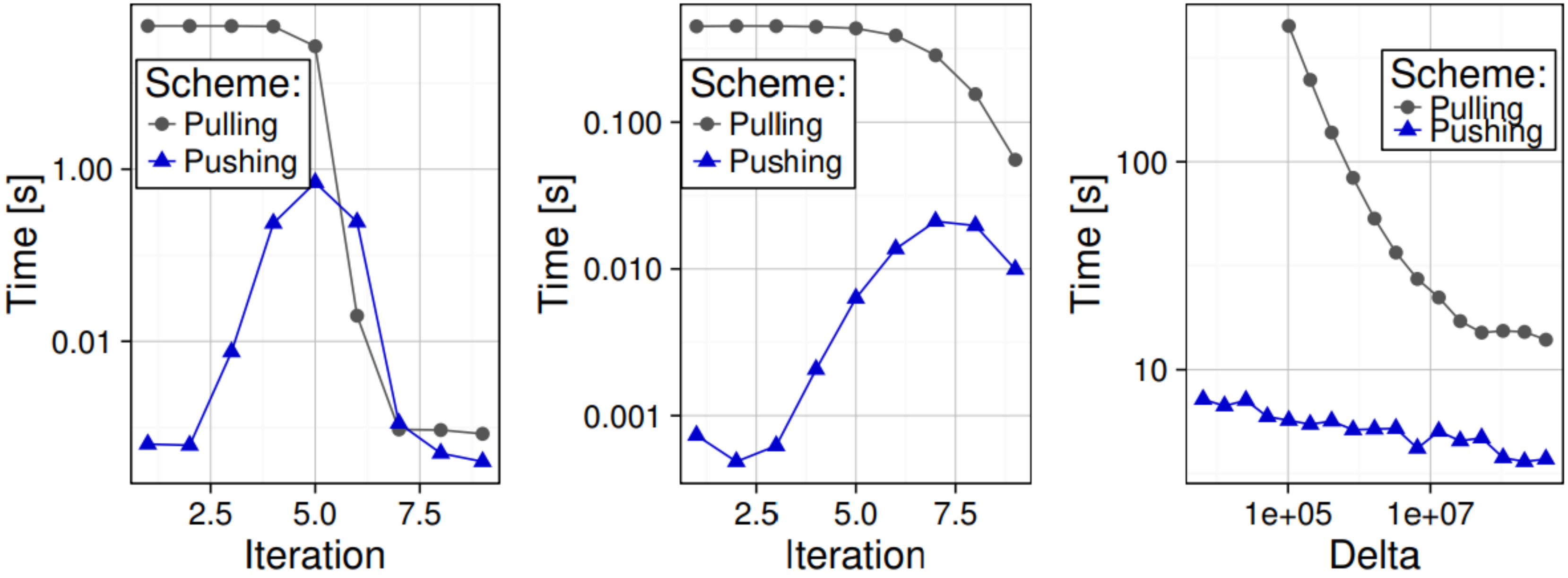
G	**Push**	**+FE**	**+GS**	**+GrS**
orc	49	173	49	49
pok	49	48	49	47
ljn	49	334	49	49
am	49	10	10	9
rca	49	5	5	5

PERFORMANCE ANALYSIS

Δ-STEPPING

SNAP
orc: social network
am: Amazon graph

Shared-Memory



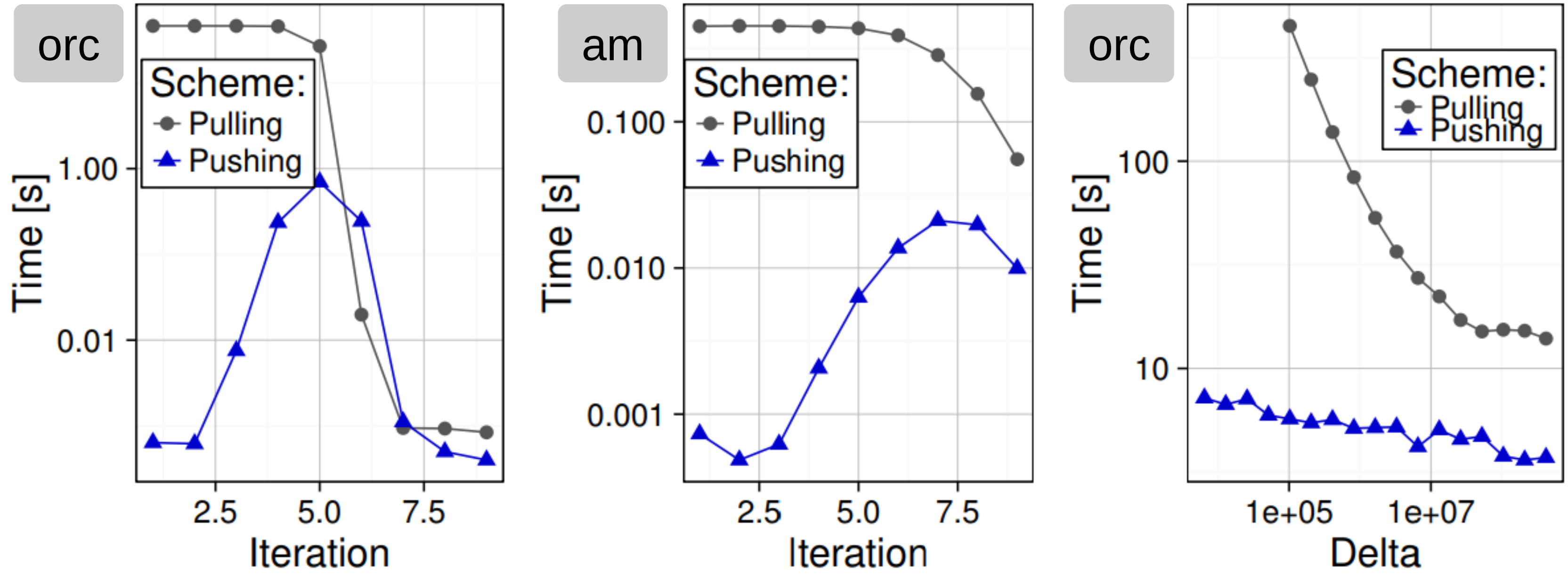
PERFORMANCE ANALYSIS

Δ-STEPPING

orc: social network

am: Amazon graph

Shared-Memory



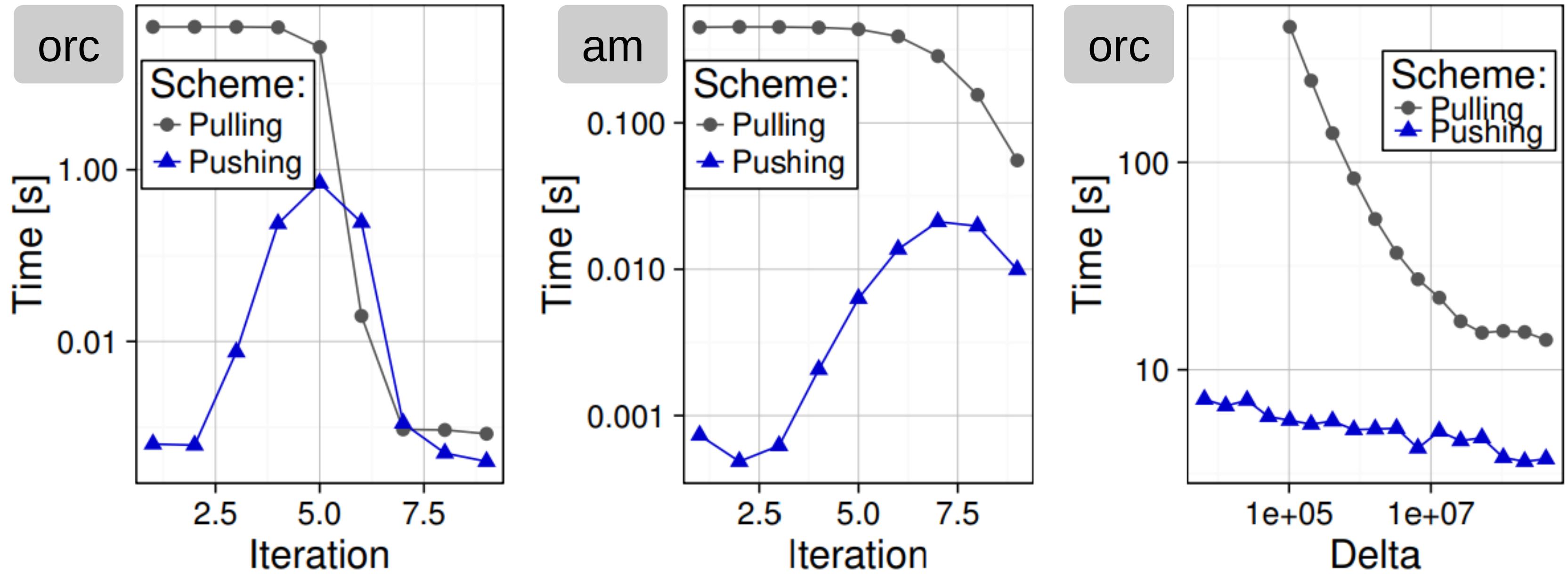
PERFORMANCE ANALYSIS

Δ-STEPPING

orc: social network
am: Amazon graph

Shared-Memory

Pushing faster



PERFORMANCE ANALYSIS

Δ-STEPPING

SNAP

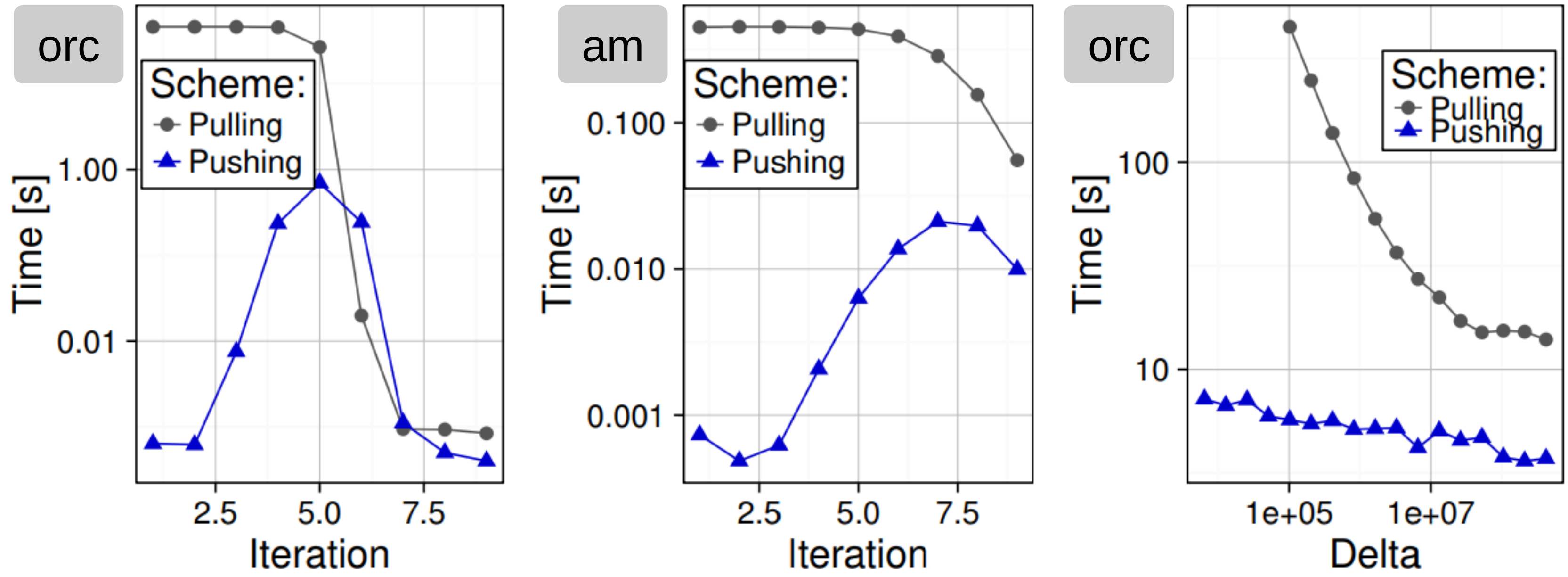
orc: social network

am: Amazon graph

Shared-Memory

! Pushing faster

Fewer reads/writes



PERFORMANCE ANALYSIS

Δ-STEPPING

orc: social network

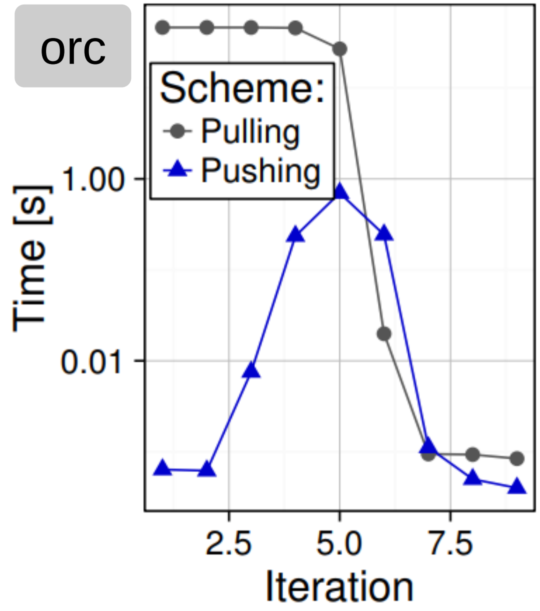
am: Amazon graph

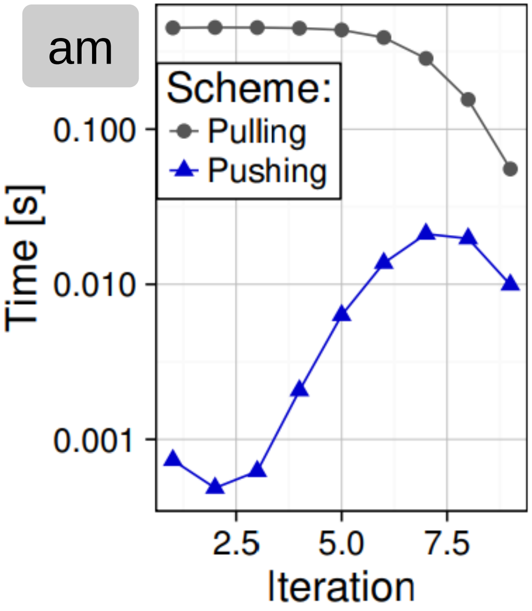
Shared-Memory

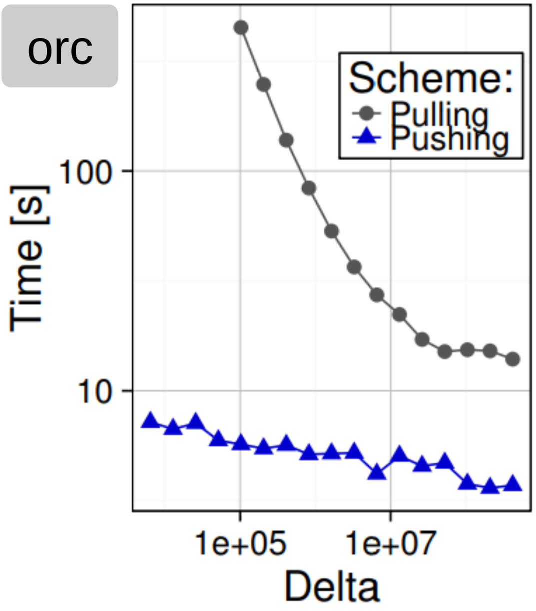
Pushing faster

Fewer reads/writes

The larger Δ, the smaller the difference between pushing and pulling





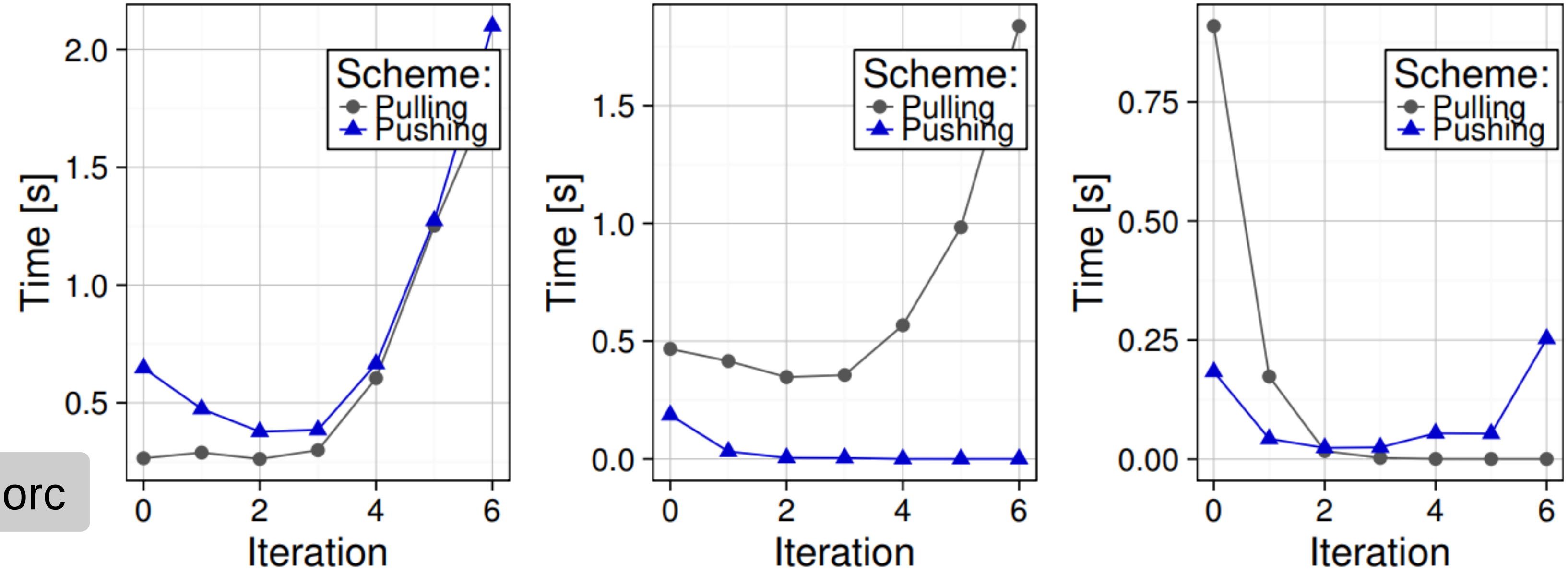


PERFORMANCE ANALYSIS

BORUVKA MST

orc: social network

Shared-Memory



orc

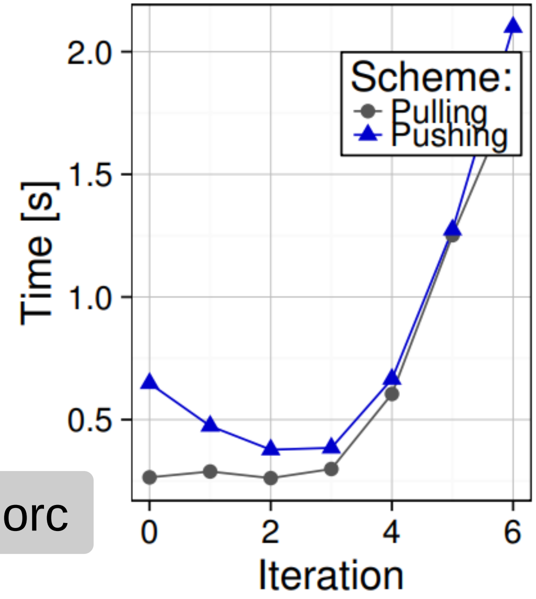
PERFORMANCE ANALYSIS

BORUVKA MST

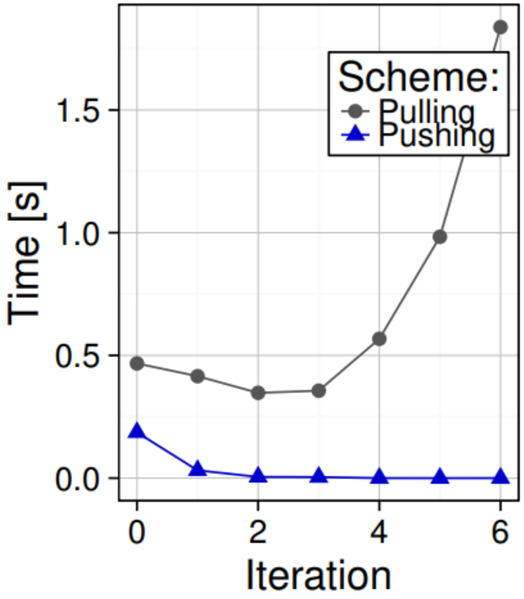
SNAP orc: social network

Shared-Memory

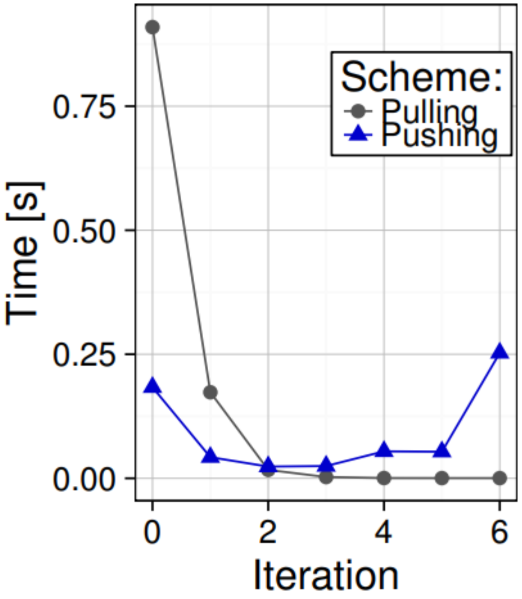
“Find minimum”



“Build merge tree”



“Merge”



orc

PERFORMANCE ANALYSIS
BORUVKA MST

orc: social network

Shared-Memory

! Pulling faster

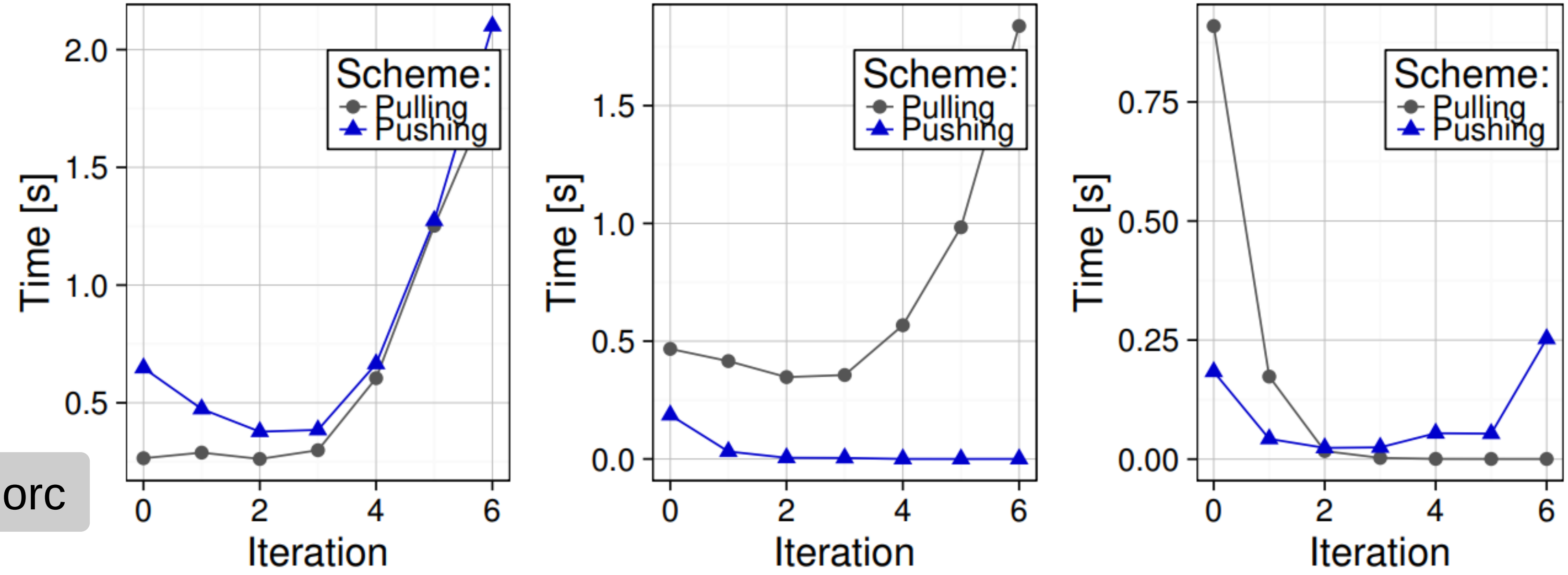
"Find minimum"

! Pushing faster

"Build merge tree"

Pushing ≈ pulling !

"Merge"



orc

PERFORMANCE ANALYSIS
BORUVKA MST

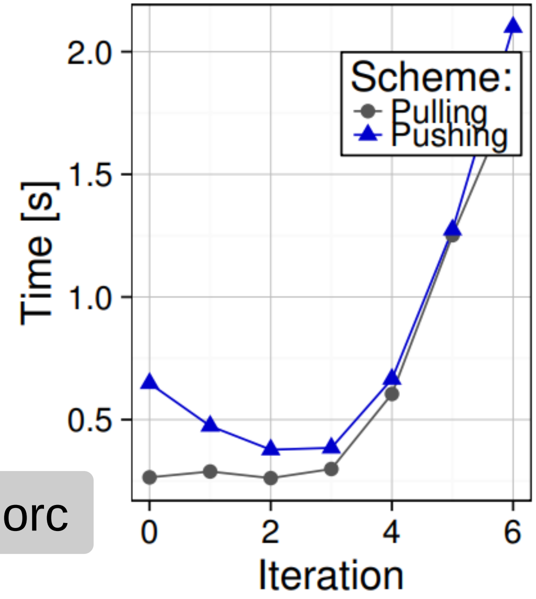
orc: social network

Shared-Memory

Pulling is cumulatively faster

Pulling faster

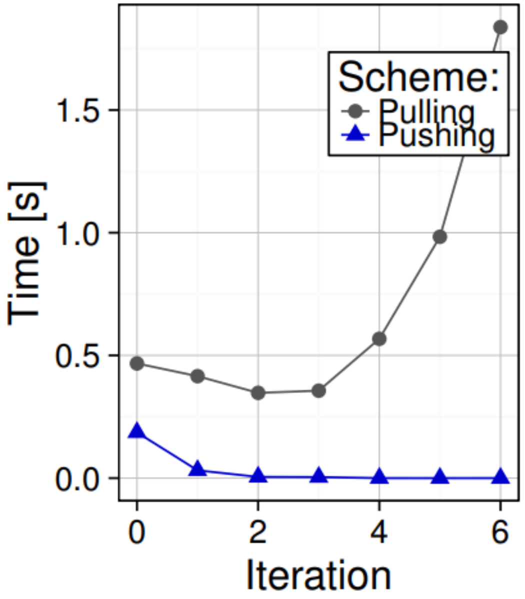
“Find minimum”



orc

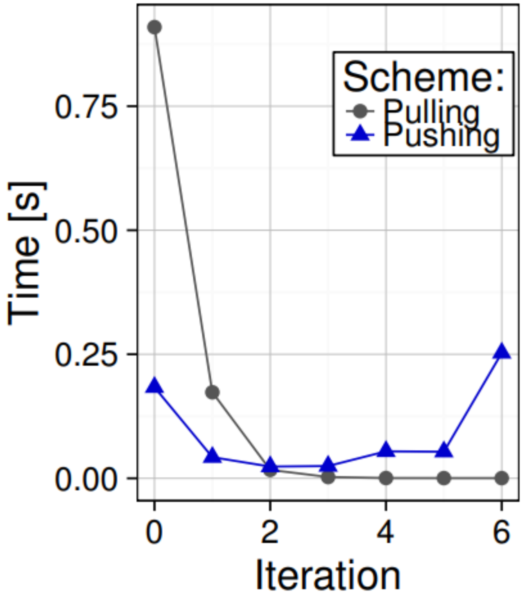
Pushing faster

“Build merge tree”



Pushing ≈ pulling

“Merge”



PERFORMANCE ANALYSIS
BORUVKA MST

orc: social network

Shared-Memory

No expensive write conflicts

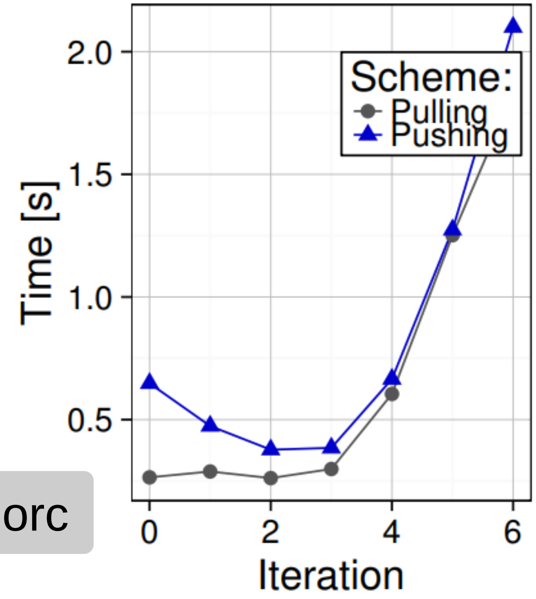
Pulling is cumulatively faster

Pulling faster

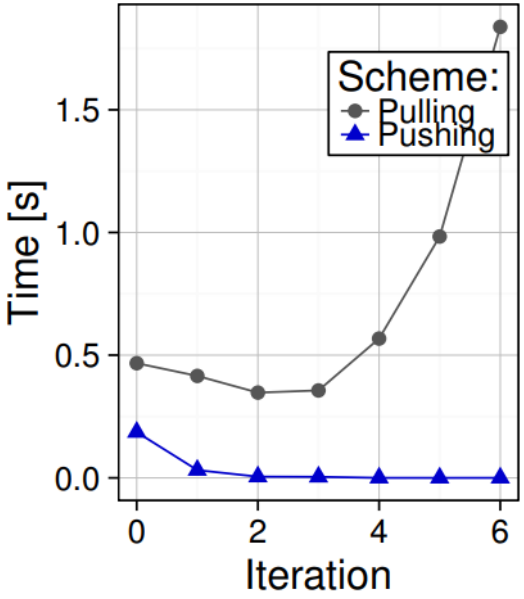
Pushing faster

Pushing ≈ pulling

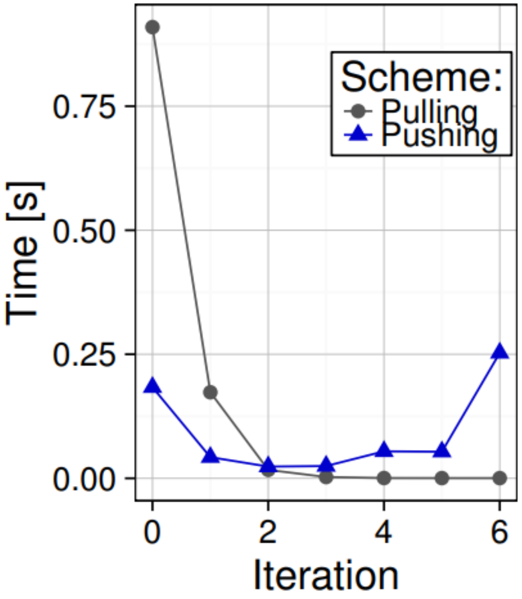
“Find minimum”



“Build merge tree”



“Merge”



orc

PERFORMANCE ANALYSIS

PAGERANK

SNAP

orc, pok, ljn: social networks

rca: road network

am: amazon graph

Shared-Memory

G	PageRank [ms]				
	orc	pok	ljn	am	rca
Pushing	572	129	264	4.62	6.68
Pulling	557	103	240	2.46	5.42

PERFORMANCE ANALYSIS
PAGERANK

SNAP

orc, pok, ljn: social networks

rca: road network

am: amazon graph

Shared-
Memory

Pulling faster in sparse graphs by ≈3%

Many cache misses dominate performance

G	**PageRank** [ms]				
	orc	pok	ljn	am	rca
Pushing	572	129	264	4.62	6.68
Pulling	557	103	240	2.46	5.42

PERFORMANCE ANALYSIS

PAGERANK

SNAP

orc, pok, ljn: social networks

rca: road network

am: amazon graph

Shared-Memory

! Pulling faster in sparse graphs by ≈3%

! Pulling faster in dense graphs by ≈19%

Many cache misses dominate performance

	PageRank [ms]				
G	orc	pok	ljn	am	rca
Pushing	572	129	264	4.62	6.68
Pulling	557	103	240	2.46	5.42

PERFORMANCE ANALYSIS
PAGERANK

SNAP

orc, pok, ljn: social networks

rca: road network

am: amazon graph

Shared-Memory

! Pulling faster in sparse graphs by ≈3%

! Pulling faster in dense graphs by ≈19%

Many cache misses dominate performance

No atomics

G	**PageRank** [ms] orc	pok	ljn	am	rca
Pushing	572	129	264	4.62	6.68
Pulling	557	103	240	2.46	5.42

PERFORMANCE ANALYSIS
PAGERANK + PA

orc, pok, ljn: social networks
rca: road network
am: amazon graph

Shared-Memory

PA: Partition-Awareness

G	**Push**	**+PA**
orc	557.985	425.928
pok	103.907	87.577
ljn	240.943	145.475
am	2.467	5.193
rca	5.422	13.705

PERFORMANCE ANALYSIS

PAGERANK + PA

SNAP

orc, pok, ljn: social networks

rca: road network

am: amazon graph

Shared-Memory

PA: Partition-Awareness

! Pushing now faster in dense graphs by ≈24%

Fewer atomics (thanks to PA) and still fewer cache misses

G	**Push**	**+PA**
orc	557.985	425.928
pok	103.907	87.577
ljn	240.943	145.475
am	2.467	5.193
rca	5.422	13.705

PERFORMANCE ANALYSIS

PAGERANK + PA

SNAP
orc, pok, ljn: social networks
rca: road network
am: amazon graph

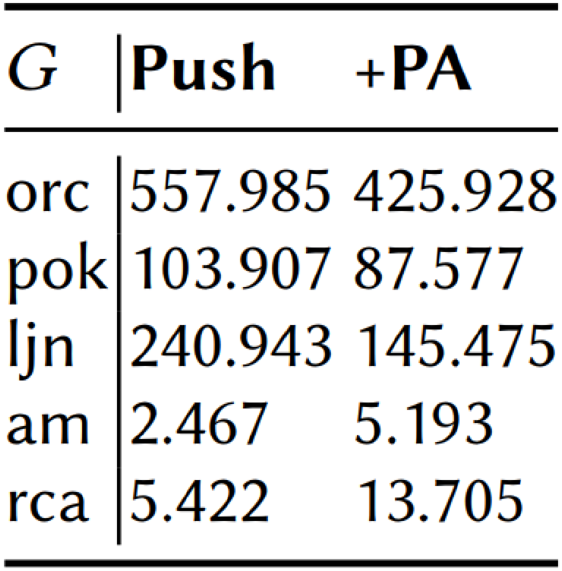
Shared-Memory

PA: Partition-Awareness

! Pushing now faster in dense graphs by ≈24%

Fewer atomics (thanks to PA) and still fewer cache misses

! Pushing+PA the slowest for sparse graphs



G	**Push**	**+PA**
orc	557.985	425.928
pok	103.907	87.577
ljn	240.943	145.475
am	2.467	5.193
rca	5.422	13.705

PERFORMANCE ANALYSIS
PAGERANK + PA

SNAP

orc, pok, ljn: social networks

rca: road network

am: amazon graph

Shared-Memory

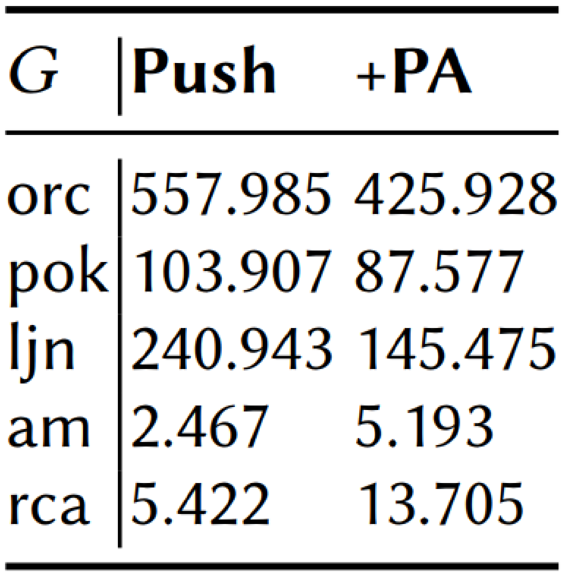
PA: Partition-Awareness

Pushing now faster in dense graphs by ≈24%

Fewer atomics (thanks to PA) and still fewer cache misses

Pushing+PA the slowest for sparse graphs

Fewer atomics dominated by more branches



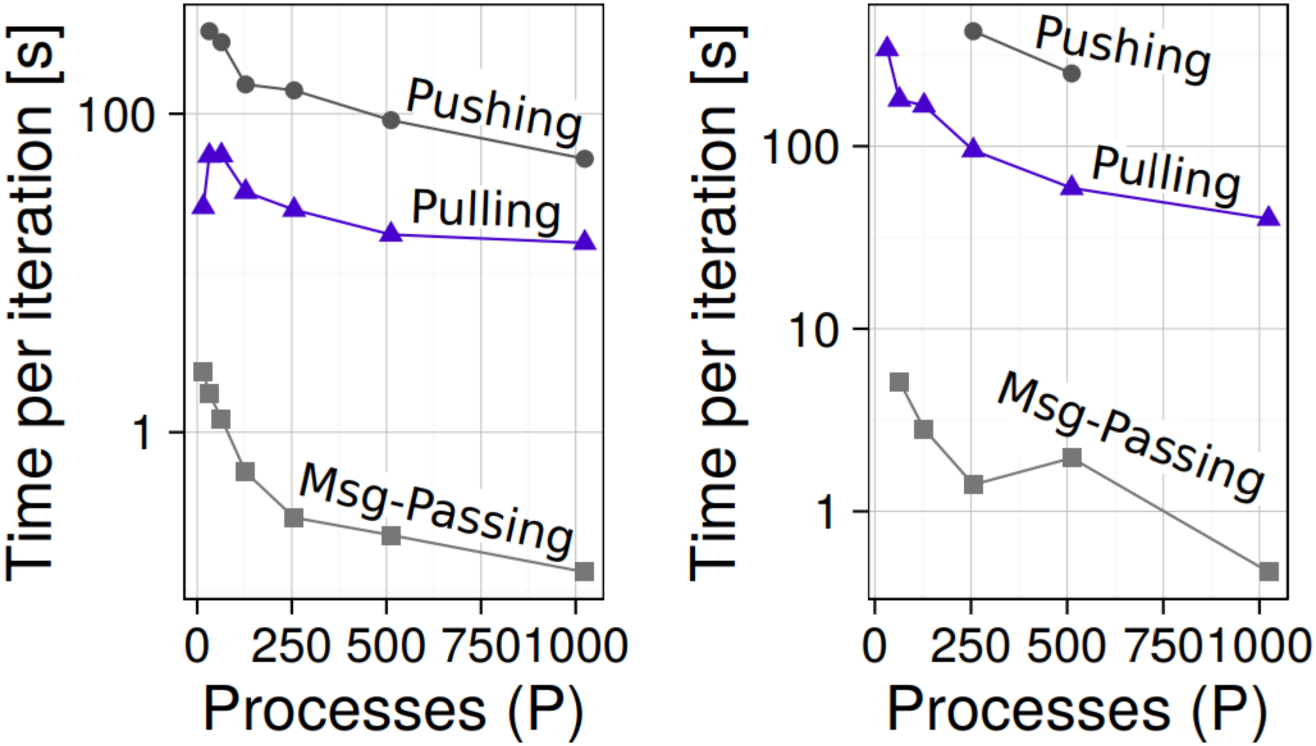
G	**Push**	**+PA**
orc	557.985	425.928
pok	103.907	87.577
ljn	240.943	145.475
am	2.467	5.193
rca	5.422	13.705

PERFORMANCE ANALYSIS

PAGERANK

Kronecker graphs

Distributed -Memory



PERFORMANCE ANALYSIS

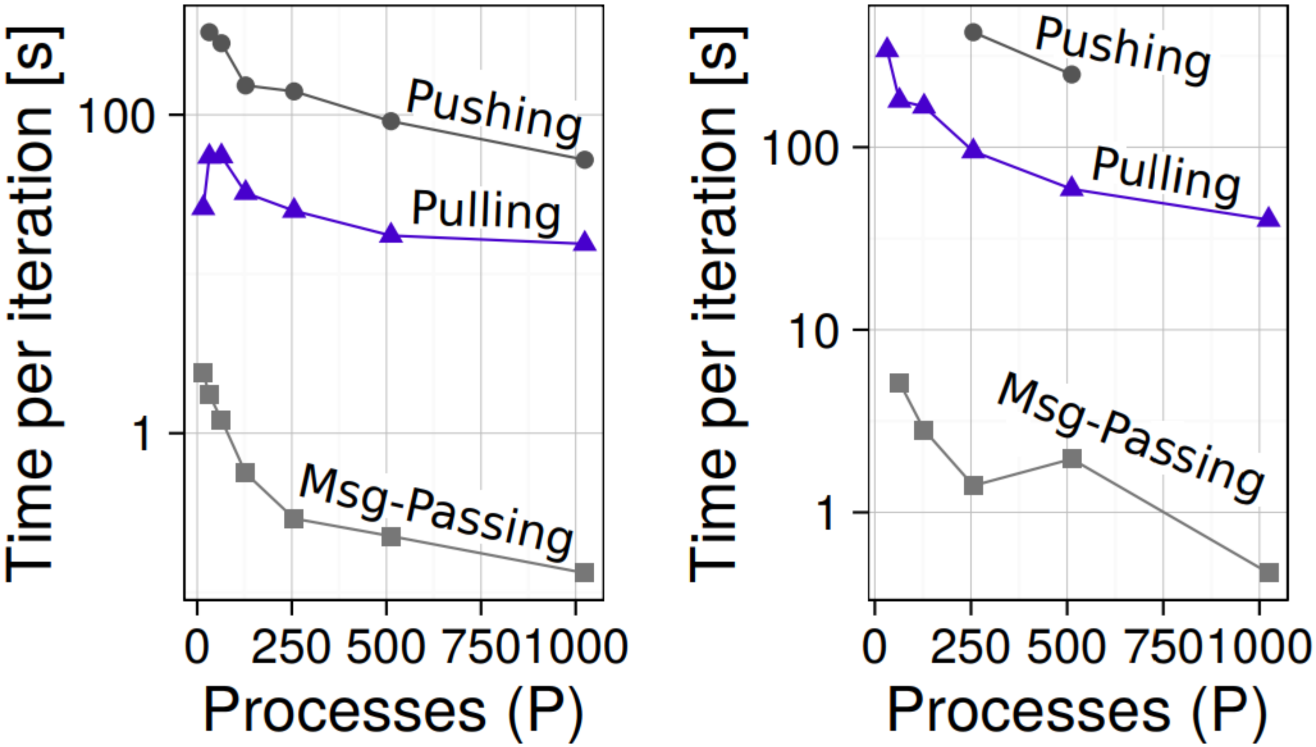
PAGERANK

Kronecker graphs

Distributed -Memory

$n = 2^{25}, m = 2^{27}$

$n = 2^{27}, m = 2^{29}$



PERFORMANCE ANALYSIS

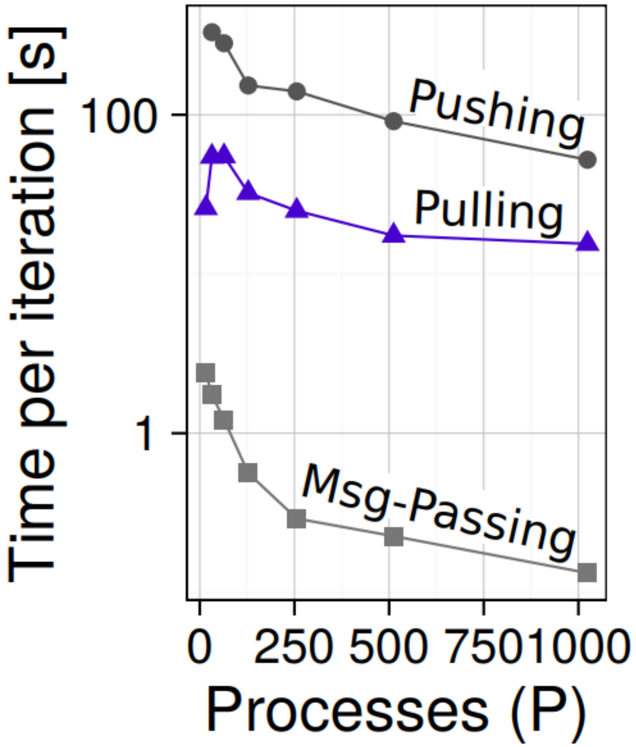
PAGERANK

! Msg-Passing fastest

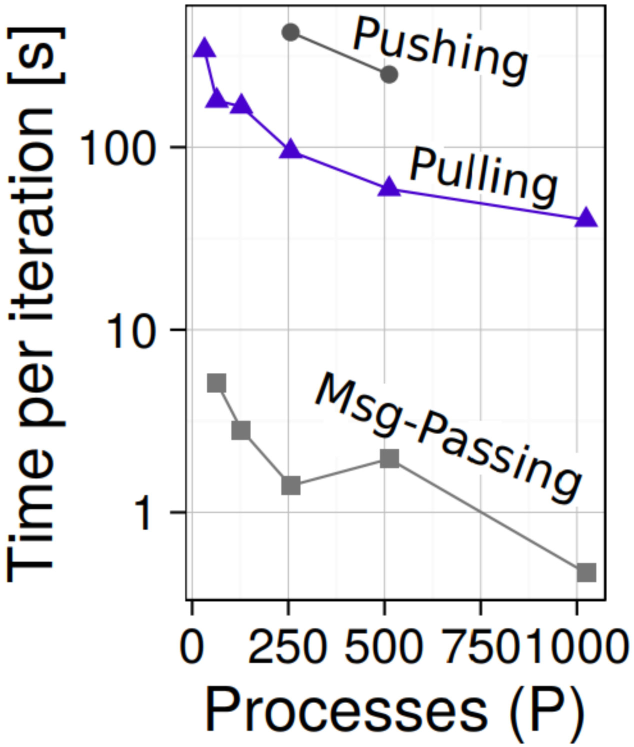
Kronecker graphs

Distributed -Memory

$n = 2^{25}, m = 2^{27}$



$n = 2^{27}, m = 2^{29}$



PERFORMANCE ANALYSIS

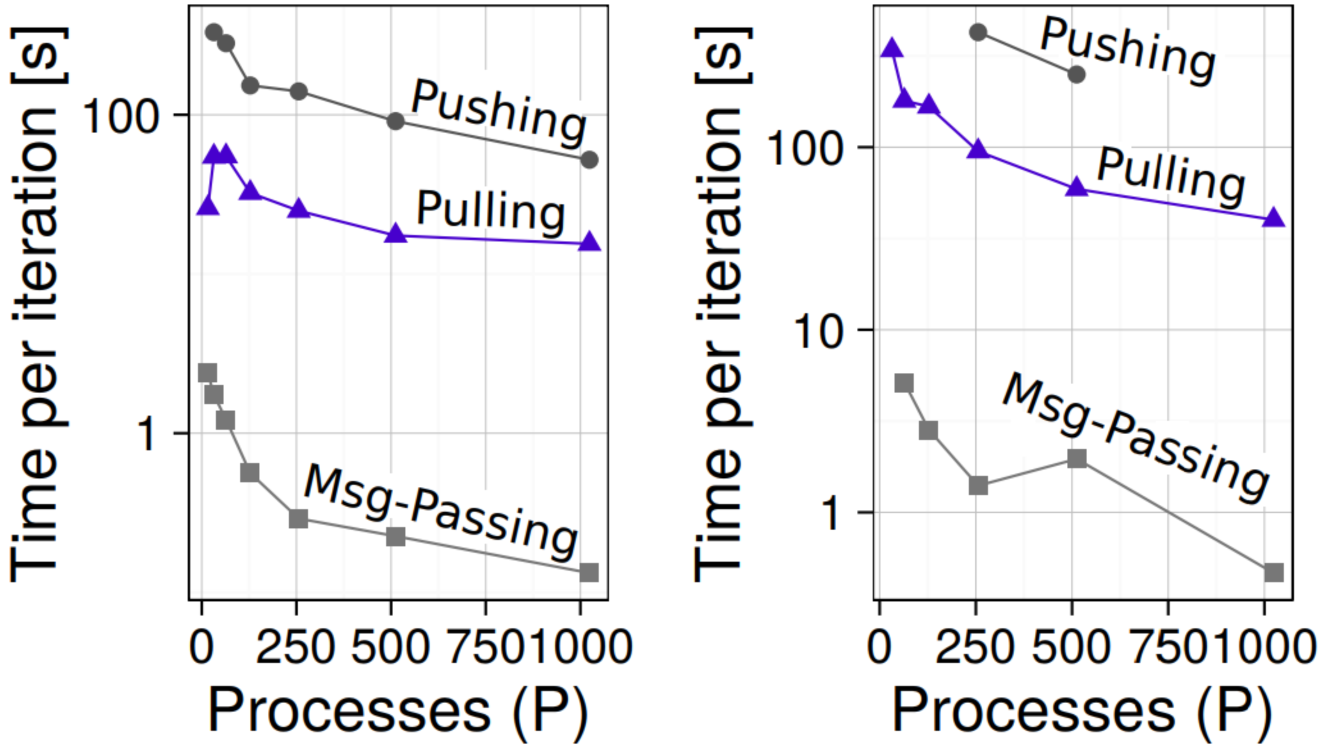
PAGERANK

Kronecker graphs

Distributed -Memory

! Msg-Passing fastest

Overheads from buffer preparation



$n = 2^{25}, m = 2^{27}$

$n = 2^{27}, m = 2^{29}$

PERFORMANCE ANALYSIS

PAGERANK

Kronecker graphs

Distributed -Memory

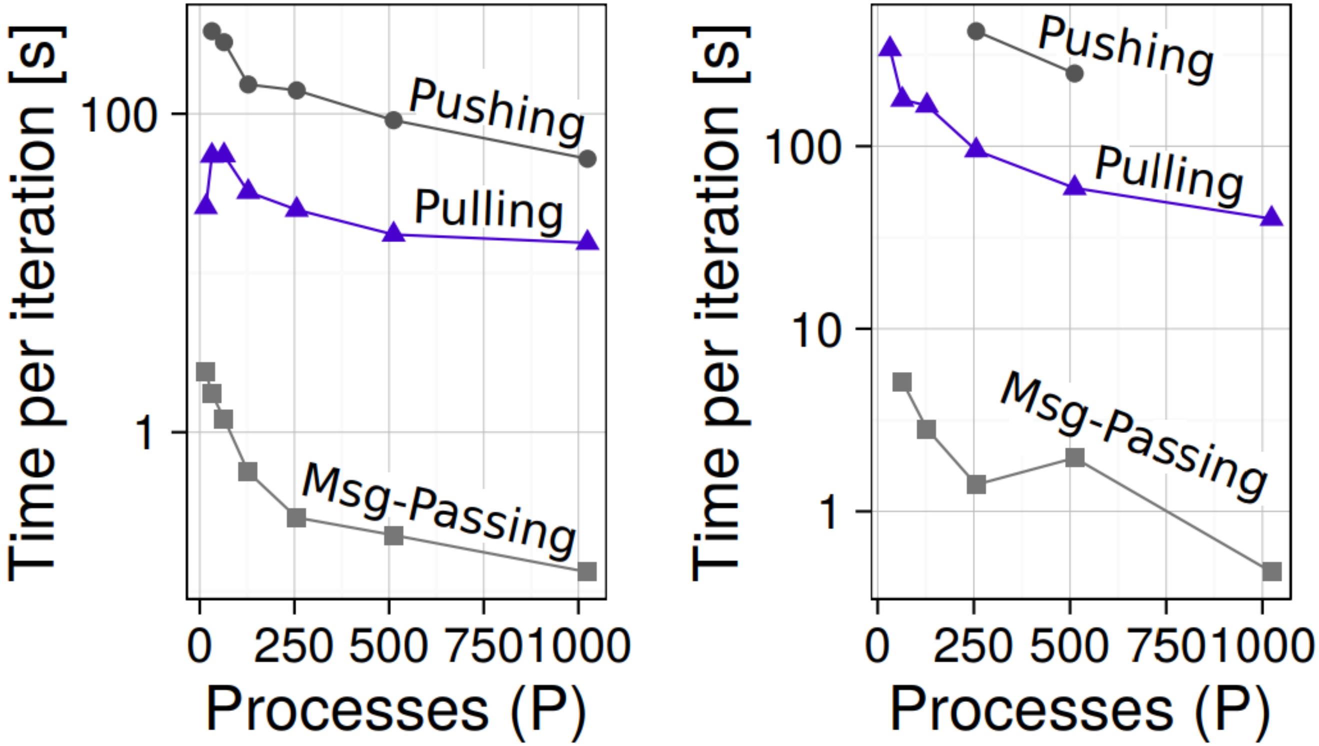
Msg-Passing fastest

Overheads from buffer preparation

…but pulling incurs more communication while pushing expensive underlying locking

$n = 2^{25}, m = 2^{27}$

$n = 2^{27}, m = 2^{29}$



PERFORMANCE ANALYSIS

PAGERANK

Kronecker graphs

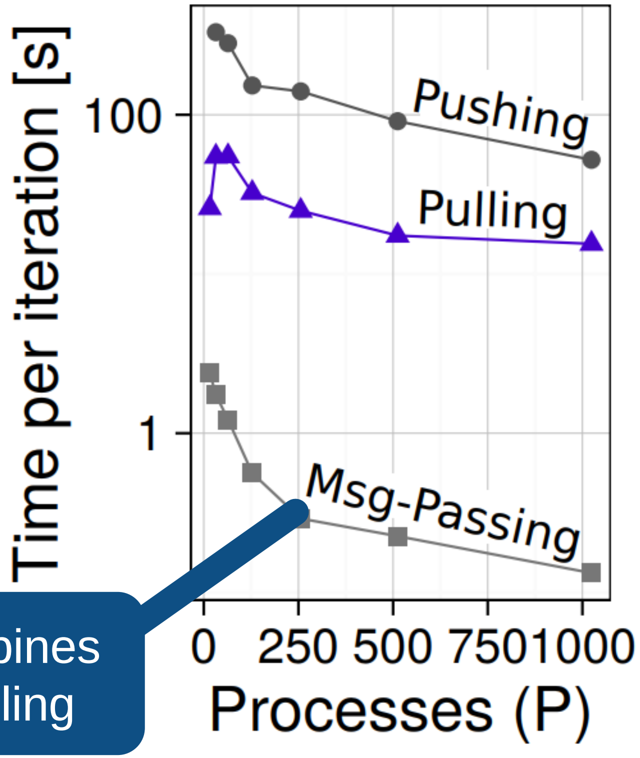
Distributed -Memory

! Msg-Passing fastest

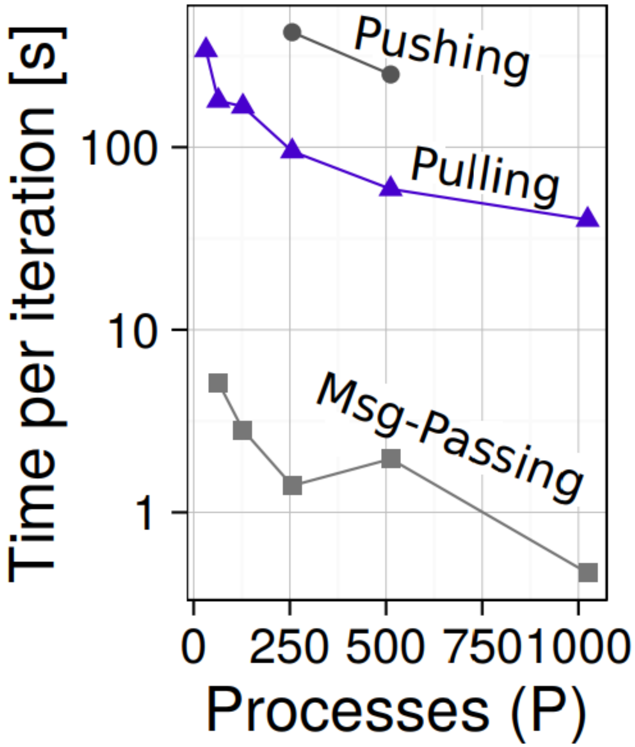
Overheads from buffer preparation

…but pulling incurs more communication while pushing expensive underlying locking

$n = 2^{25}, m = 2^{27}$



$n = 2^{27}, m = 2^{29}$



! Collectives: combines pushing and pulling

PERFORMANCE ANALYSIS

PAGERANK

Kronecker graphs

Distributed -Memory

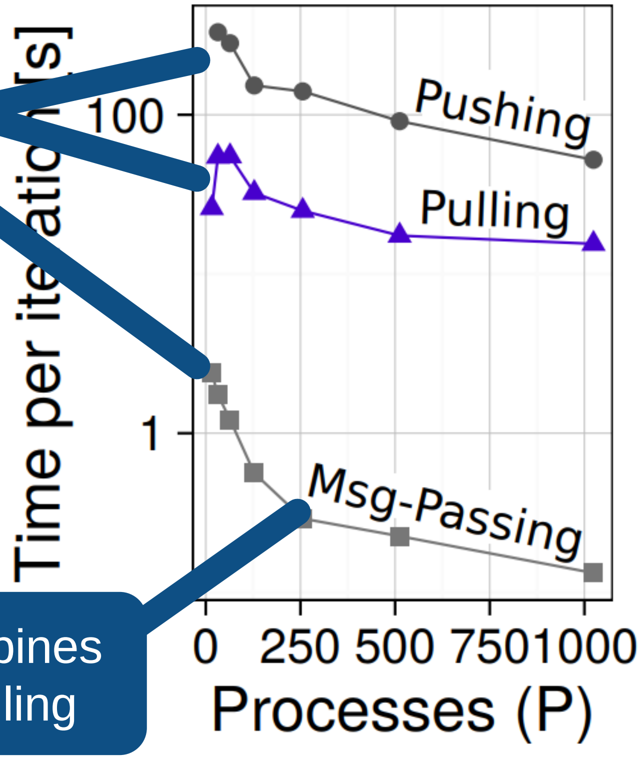
Msg-Passing fastest

Overheads from buffer preparation

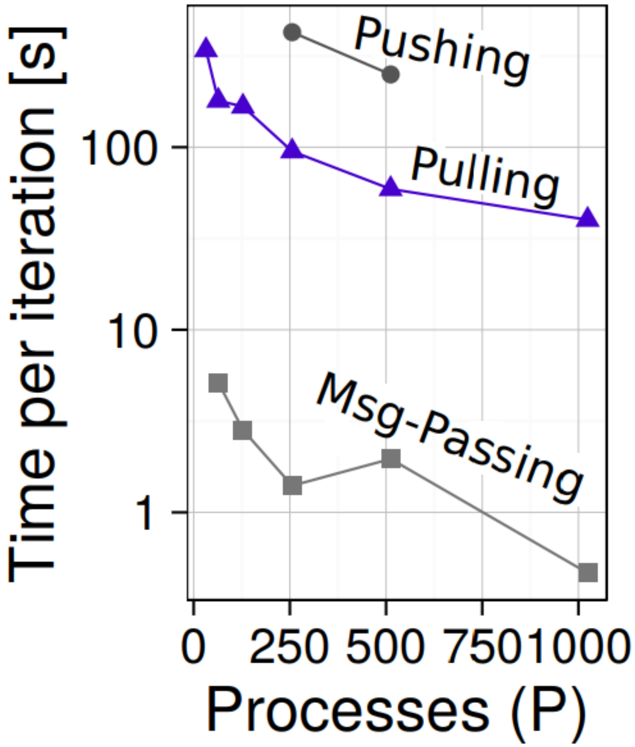
…but pulling incurs more communication while pushing expensive underlying locking

Storage per process
$O(1)$
$O((n\hat{d})/P)$

$n = 2^{25}, m = 2^{27}$



$n = 2^{27}, m = 2^{29}$



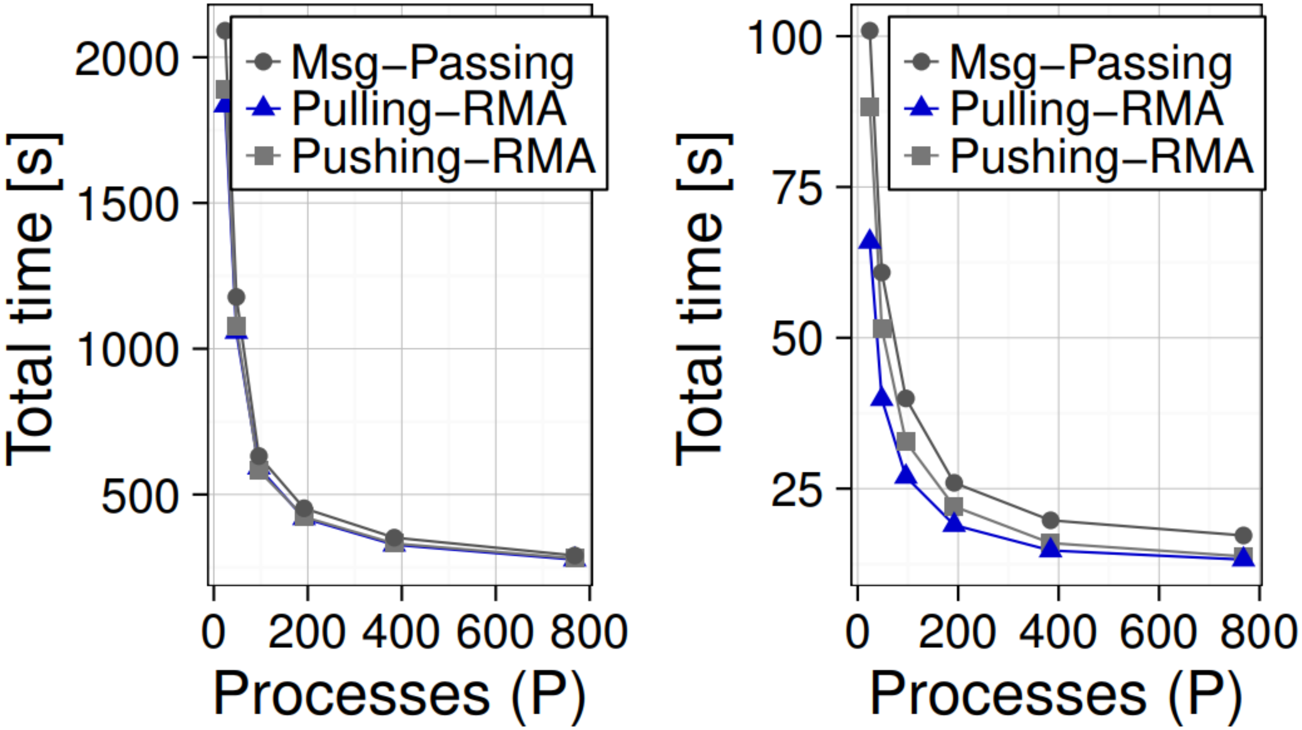
Collectives: combines pushing and pulling

PERFORMANCE ANALYSIS

TRIANGLE COUNTING

SNAP orc, ljn: social networks

Distributed
-Memory



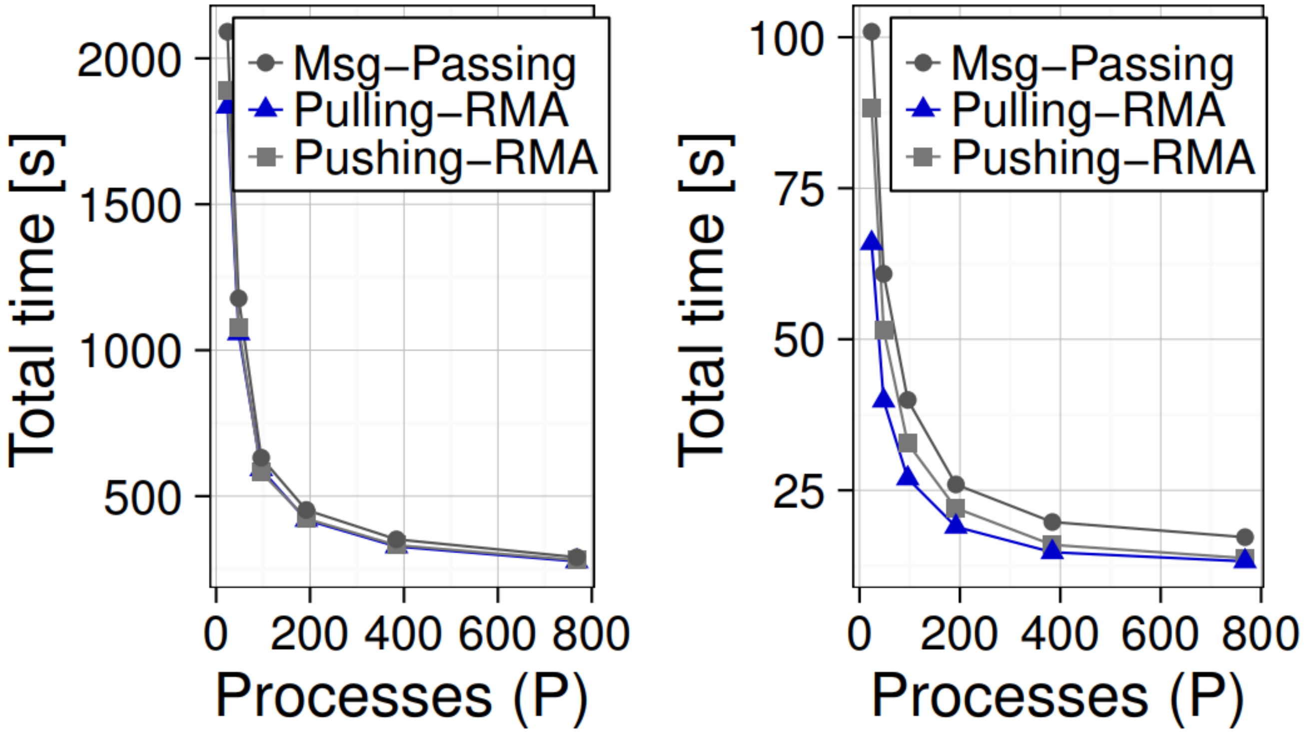
PERFORMANCE ANALYSIS

TRIANGLE COUNTING

SNAP orc, ljn: social networks

Distributed -Memory

RMA fastest



PERFORMANCE ANALYSIS

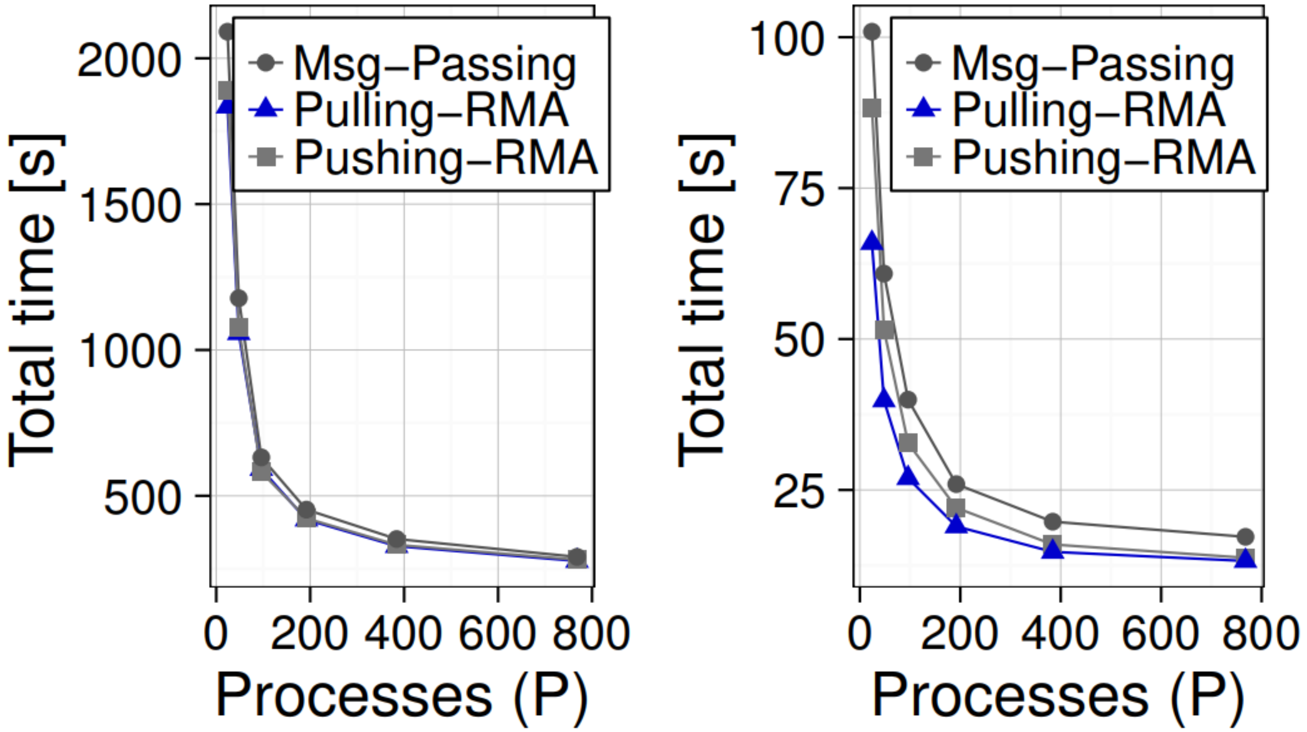
TRIANGLE COUNTING

SNAP orc, ljn: social networks

Distributed -Memory

Msg-Passing incurs now more communication

RMA fastest



PERFORMANCE ANALYSIS

TRIANGLE COUNTING

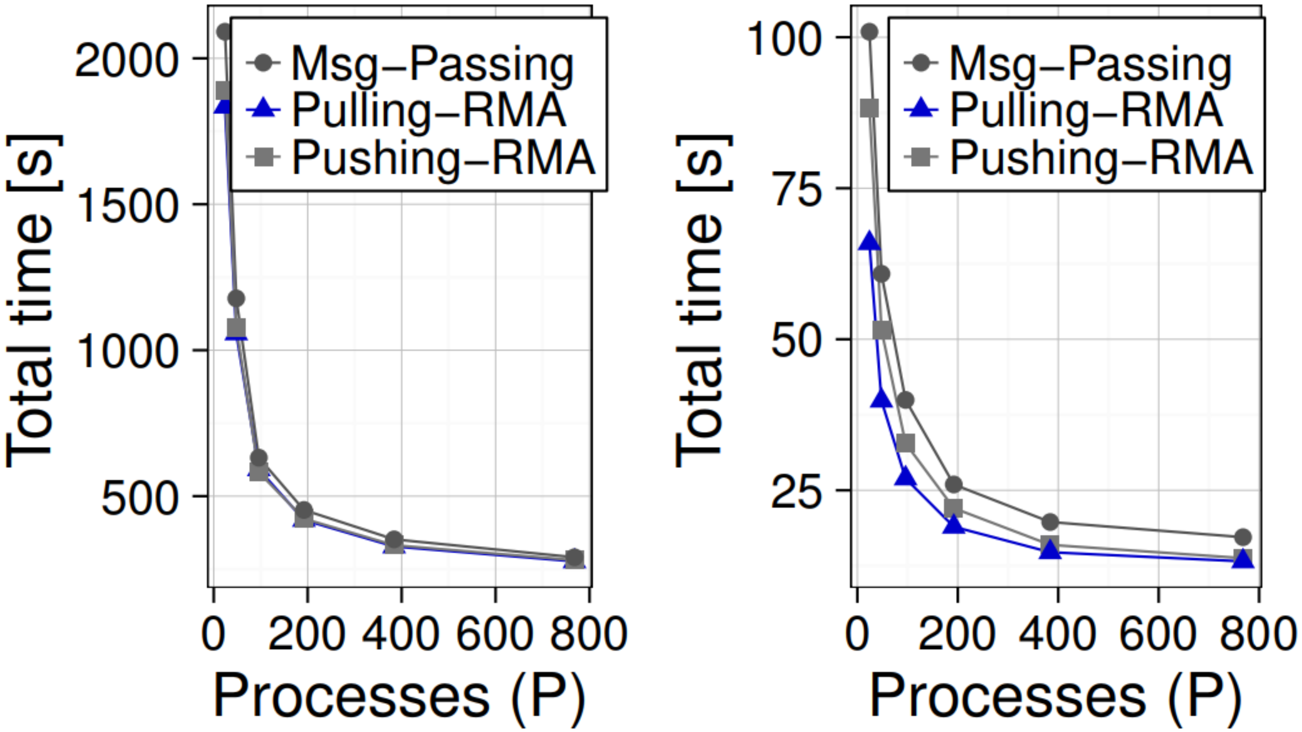
SNAP orc, ljn: social networks

Distributed -Memory

Msg-Passing incurs now more communication

! RMA fastest

Pushing does not require the expensive locking protocol (Cray offers fast remote atomics for integers)



PERFORMANCE ANALYSIS

TRIANGLE COUNTING

SNAP orc, ljn: social networks

Distributed -Memory

Msg-Passing incurs now more communication

! RMA fastest

Pushing does not require the expensive locking protocol (Cray offers fast remote atomics for integers)

Storage per process

$O(\hat{d})$