Advanced Programming Techniques

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

Parallel search: some motivations

- BFS/DFS are fundamental primitives in graph algorithms.

$\Longrightarrow$ We need those searches to be done <u>real</u> fast!

- Unfortunately, standard BFS/DFS implementations are inherently sequential/serial.

$\rightarrow$ sequential: all operations executed one after the other ($\neq$ concurrent).

$\rightarrow$ serial: all operations executed by one computing unit ($\neq$ parallel)

- Today's objective: speeding-up graph searches using parallelism!

A few basics on parallel algorithms

Disclaimer: parallel/distributed/concurrent programming and their relatives are the main topic of other classes!

- Multiple operations can be executed in a given time (often, concurrently) by independent + asynchronous computing units.

→ **processors**, threads, cores, ...

- Processors have private memories, but they can share information using:

  - Shared memory
  - Message passing (Distributed memory)

→ For very large graphs (that do not hold on a single machine), distributed memory is required.

Synchronization

- Concurrent accesses to a resource lead to minor/major races, with consequences on the correctness, and even termination, of the algorithm.
- Various synchronization mechanisms are available:
  - **Lock**/mutex/semaphore: limited access at a given time to a resource
  - **Barrier**: stopping point where all processors are waiting for each other
  - **Reducer**: processors have private (incoherent) views of some variables, which are combined when subcomputations join (using an update function, to be specified).

1	$x = 10$	1	$x = 10$
2	$x{+}{+}$	2	$x{+}{+}$
3	$x \mathrel{+}= 3$	3	$x \mathrel{+}= 3$
4	$x \mathrel{+}= -2$	4	$x \mathrel{+}= -2$
5	$x \mathrel{+}= 6$	5	$x \mathrel{+}= 6$
6	$x{-}{-}$		$x' = 0$
7	$x \mathrel{+}= 4$	6	$x'{-}{-}$
8	$x \mathrel{+}= 3$	7	$x' \mathrel{+}= 4$
9	$x{+}{+}$	8	$x' \mathrel{+}= 3$
10	$x \mathrel{+}= -9$	9	$x'{+}{+}$
		10	$x' \mathrel{+}= -9$
			$x \mathrel{+}= x'$

Performance evaluation

Various (conflicting) optimization criteria:

- Step bound ("execution time")
- Number of processors
- Total work: Number of processors $\times$ step bound ("sequential time")

<u>Dream goal</u>: step bound in $T(n, m)/p$, where $T(n, m)$ is the sequential time complexity, and p the number of processors ("Linear speed-up").

Early work on parallel computing (mostly theoretical) was considering step bound, while neglecting the number of processors (complexity class NC).

OpenMP

- A programming interface that supports shared-memory parallel computing.

→ Available languages: C/C++, Fortran.

- Included in most recent versions of popular compilers, such as gcc.

```
gcc -fopenmp
```

- Requires `<omp.h>` in code dependencies

- Parallelization occurs via a system of annotations + a few primitives (much more programmer friendly than POSIX).

Some essentials of OpenMP (1/n)

→ Parallel blocks of instructions:

```
#pragma omp parallel
{
 ...
}
```

→ The number of threads can be accessed to with `omp_get_num_threads()`. It can be changed at the programmer's convenience:

```
#pragma omp parallel num_threads(5)
```

→ Every thread has an ID, which can be accessed to with `omp_get_thread_num()`, and can be used in order to force threads to execute different codes.

Some essentials of OpenMP (2/n)

→ Parallelization of a for loop:

```
#pragma omp parallel for
for(int i = 0; ...)  {
 ...
}
```

→ Variables outside/inside a parallel blocks are shared/private by default. However, this behaviour can be changed.

```
int a, b;
#pragma omp parallel private(a) shared(b)
{
 ...
}
```

Some essentials of OpenMP (3/n)

→ Synchronization mechanisms

```
int x;
#pragma omp parallel
{
 ...
 #pragma omp critical
 {
  ...
 }
 ...
 #pragma omp barrier
 ...
}
```

Some essentials of OpenMP (4/n)

→ Reductions

```
 double  ave=0.0, A[MAX];    int i;
#pragma omp parallel for reduction (+:ave)
 for (i=0;i< MAX; i++) {
        ave + = A[i];
 }
 ave = ave/MAX;
```

Parallel search by pruning edges

- Classical searches BFS/DFS must visit each vertex only once.
- Once we visit a vertex v, we can isolate it by removing v from the adjacency list of all its neighbours.
- These independent removals can be executed in parallel!

```
procedure isolate(v):
 for every neighbour u in parallel:
  remove v from A[u]
```

<u>Step bound</u>: $\lceil \frac{d(v)}{p} \rceil \leq 1 + d(v)/p$.

Applications to DFS

```
procedure dfs(v):
 visit v
 isolate(v)
 while A[v] is nonempty:
  dfs(A[v].head) //first unvisited neighbour
```

Remark: all visited neighbours are removed from A[v], therefore, every remaining neighbour of A[v] must be unvisited.

Step bound: $\sum_v (1 + d(v)/p) \leq n + 2m/p$.

Applications to BFS

```
procedure bfs(s):
 Q:= {}
 Q.enqueue(s)
 isolate(s)
 while Q is nonempty:
  v := Q.dequeue()
  visit v
  for every neighbour u of A[v]:
   isolate(u)
   Q.enqueue(u)
```

<u>Step bound</u>: $\sum_v (1 + d(v)/p) \leq n + 2m/p$.

Limitations of edge pruning procedures

- The Step bound is always at least in $\Omega(n)$.

$\rightarrow$ mostly relevant for graphs of moderate order (number of nodes) and **dense**.

- The procedure modifies the graph (it iteratively removes all edges).

$\rightarrow$ We need to repair the graph or to work on a copy.

PDFS: State of the art

- There is no known efficient parallelization of DFS.

- In fact, there is strong evidence that we *cannot* parallelize the serial DFS algorithm (unless all polynomial-time problems can be parallelized).

- There exist parallel DFS algorithms in $\log^c n$ steps. They compute a DFS spanning tree, but not necessarily the one computed by serial DFS. Furthermore, this algorithm is:

  - randomized
  - intricate (various reductions to min-cost flow)
  - and it requires a <u>huge</u> (but polynomial) number of processors!

- Some authors have considered heuristics where each processor executes a partial DFS (vertices already visited by another processor are ignored).

PBFS on shared memory

- Most suited for graphs of moderately large size (that can still hold on the machine).

- Higher memory bandwidth, lower latency + no message overhead.

- Vertices are explored one distance layer after another, in any order (which is weaker than what serial BFS does, but anyway...). **We explore in parallel vertices on a same layer.**

- Since layers may have different size and number of incident edges, the work distribution at every level is highly irregular.

Classification of PBFS algorithms

- **Vertex centric**: Every vertex v is owned by one processor, which repeats the same code (while loop) until any neighbor of v get visited. Then, we can visit v, compute its level (distance to the source) and notify in parallel all unvisited v's neighbors.

(+) No need for a queue nor any kind of vertex container

(−) However, we need to synchronize vertices' processors after every loop (e.g., with a barrier).

(−) If the diameter of the graph is D, then the number of steps is in $\mathcal{O}(D)$, and the total work in $\mathcal{O}(Dn)$, which is only interesting for small D.

- **Container centric**: vertices of the current level are stored in a queue (<u>or another container</u>), which is processed in parallel.

The notion of frontier

- The traditional serial implementation of BFS involves only one queue.
- However, most parallel implementations rely on two queues: one for the current distance layer (the frontier), and one for their unvisited neighbours (the next frontier).

```
procedure bfs(s):
 Q := {}, Q' := {}
 Q.enqueue(s)
 while Q is nonempty:
  v := Q.dequeue()
  visit v
  for all unvisited neighbours u:
   if u is not already in Q' then Q'.enqueue(u)
  Q' := Q
```

Simple PBFS

Input: $G(V,E)$, source vertex r

Output: Array $P[1..n]$ with $P[v]$ holding the parent of v

Data: CQ : queue of vertices to be explored in the current level
NQ : queue of vertices to be explored in the next level

```
for all v ∈ V in parallel do
    P[v] ⟵ ∞;
P[r] ⟵ 0;
CQ ⟵ Enqueue r;
while CQ ≠ ϕ do
    NQ ⟵ ϕ;
    for all u ∈ CQ in parallel do
        u ⟵ Dequeue CQ;
        for each v adjacent to u in parallel do
            if P[v] = ∞ then then
                P[v] ⟵ u;
                NQ ⟵ Enqueue v;
    Swap(CQ, NQ);
```

Simple PBFS

Input: $G(V,E)$, source vertex r

Output: Array $P[1..n]$ with $P[v]$ holding the parent of v

Data: CQ : queue of vertices to be explored in the current level
NQ : queue of vertices to be explored in the next level

```
for all v ∈ V in parallel do
    P[v] ⟵ ∞;
P[r] ⟵ 0;
CQ ⟵ Enqueue r;
while CQ ≠ ϕ do
    NQ ⟵ ϕ;
    for all u ∈ CQ in parallel do
        u ⟵ Dequeue CQ;
        for each v adjacent to u in parallel do
            if P[v] = ∞ then then
                P[v] ⟵ u;
                NQ ⟵ Enqueue v;
    Swap(CQ, NQ);
```

Need to be atomic operations (fully executed at once), e.g. by using synchronization such as locks/mutexes.

Optimizations

<u>Goal</u>: avoid synchronization whenever possible.

Reminder:

```
for each v adjacent to u in parallel do
    if P[v] = ∞ then then
        P[v] ⟵ u;
        NQ ⟵ Enqueue v;
```

$\Longrightarrow$ We may check whether v is visited ("dirty read") before entering in the critical section.

$\Longrightarrow$ Faster test if we keep track of all visited vertices in a **bitmap**.

Bitmaps are just binary machine words (low memory storage). So, the space overhead can be neglected.

bitmap PBFS

```
for all v ∈ V in parallel do
    P[v] ⟵ ∞;
for i ⟵ 1..n in parallel do
    Bitmap[i] ⟵ 0;
P[r] ⟵ 0;
CQ ⟵ Enqueue r;

while CQ ≠ ϕ do
    NQ ⟵ ϕ;
    while CQ ≠ ϕ in parallel do
        u ⟵LockedDequeue (CQ);
        for each v adjacent to u do
            a ⟵ Bitmap[v];
            if a = 0 then
                prev ⟵ LockedReadSet (Bitmap[v],1);
                if prev = 0 then
                    P[v] ⟵ u;
                    LockedEnqueue (NQ, v);
    Synchronize;
    Swap(CQ, NQ);
```

The `LockedReadSet` operation succeeds only if no other concurrent thread has already modified v's bit.

BFS containers: beyond queues

Some observations about PBFS algorithms:

- Vertices of the frontier must be dequeued one at a time because of synchronization.

- Similarly, vertices must be enqueued in the next frontier one at a time.

Negative consequence: the number of steps must be at least $\Omega(n)$!

$\rightarrow$ In order to mitigate this major hindrance, processors must work on local containers for the frontier/next frontier, which are reunited every time a level has been processed (using reducers).

$\rightarrow$ Requires efficient union/split operations on containers, in order to ensure load balancing (balanced repartition of the tasks between processors).

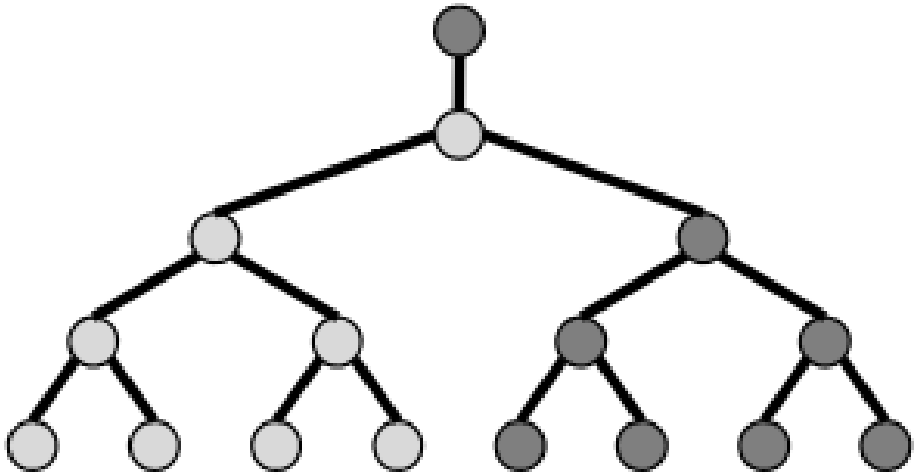
Best achieved with a new data structure, called a bag.

Pennants

Definition

Pennant Tree with 2^k nodes, such that:

- The root has no right child;
- The left subtree is a complete binary rooted tree on k levels.

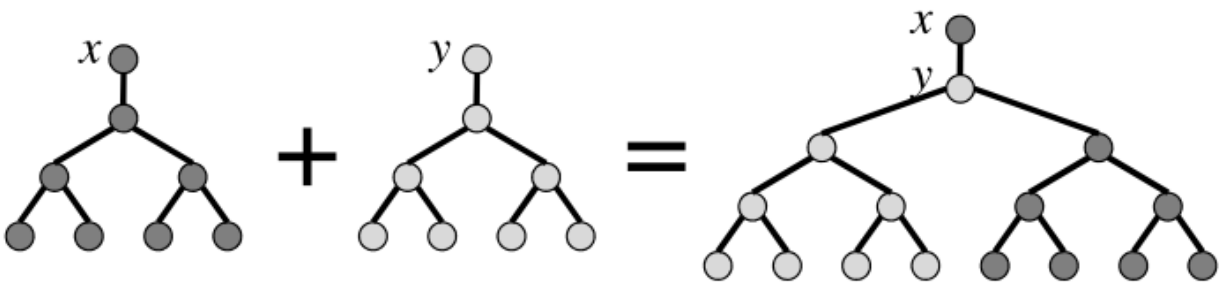


Operation PENNANT-UNION

Input: pennants with 2^k nodes each, and respective roots x and y.

Output: a new pennant with 2^{k+1} nodes and root either x or y.

```
PENNANT-UNION(x,y)
  y.right = x.left
  x.left = y
  return x
```



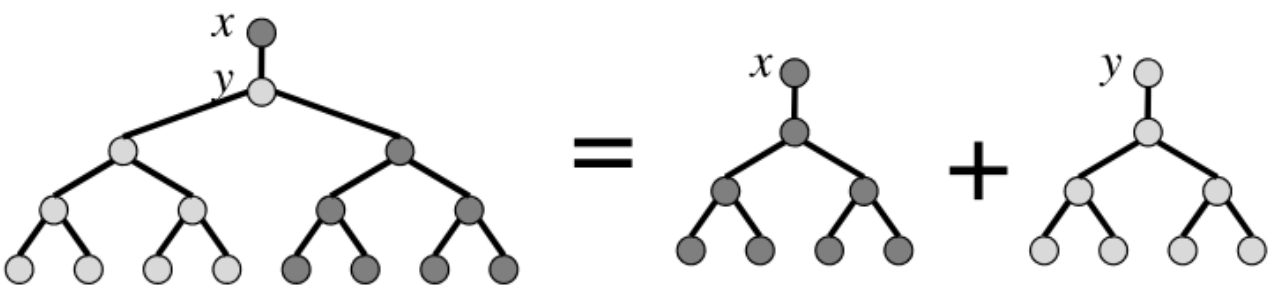
<u>Complexity</u>: $\mathcal{O}(1)$

Operation PENNANT-SPLIT

Input: a pennant with 2^{k+1} nodes and root x.

Output: new pennants with 2^k nodes each.

```
PENNANT-SPLIT(x)
y = x.left
x.left = y.right
y.right = NULL
return y
```

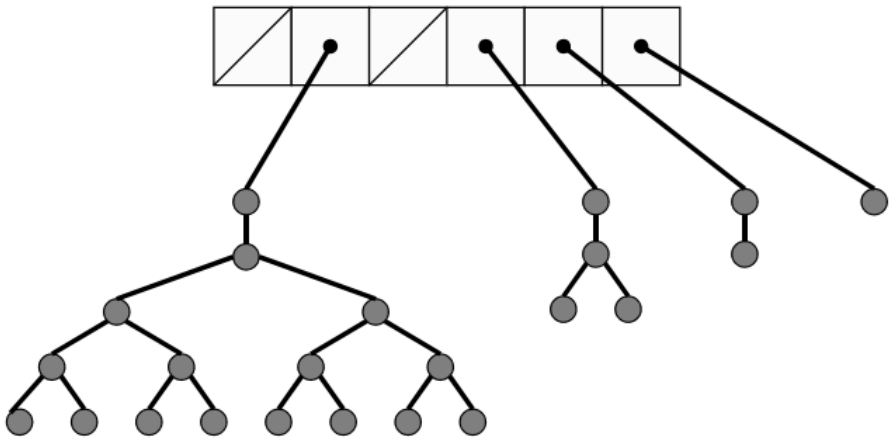


Complexity: $\mathcal{O}(1)$

Bags

- A bag is a vector of (pointers to) pennants. If there are n elements stored in the bag, then for every integer k:

$$\textit{there is a pennant with } 2^k \textit{ nodes} \iff \textit{the } k\textit{th bit of } n \textit{ is set to } 1.$$



A bag with $23 = 010111_2$ elements.

<u>Remark</u>: there is at most one pennant with 2^k nodes.

Insertion

- Mimics incrementation of a binary counter (with carry flag):

```
BAG-INSERT(S,x)
k = 0
while S[k] ≠ NULL
        x = PENNANT-UNION(S[k],x)
        S[k++] = NULL
S[k] = x
```

Insertion

- Mimics incrementation of a binary counter (with carry flag):

```
BAG-INSERT(S, x)
k = 0
while S[k] ≠ NULL
    x = PENNANT-UNION(S[k], x)
    S[k++] = NULL
S[k] = x
```

pennant with 1 node
(new element)

Insertion

- Mimics incrementation of a binary counter (with carry flag):

```
BAG-INSERT(S,x)
k = 0
while S[k] ≠ NULL
    x = PENNANT-UNION(S[k],x)
    S[k++] = NULL
S[k] = x
```

bitwise addition with carry

Insertion

- Mimics incrementation of a binary counter (with carry flag):

```
BAG-INSERT(S,x)
k = 0
while S[k] ≠ NULL
    x = PENNANT-UNION(S[k],x)
    S[k++] = NULL
S[k] = x
```

bitwise addition with carry

<u>Worst-case complexity</u>: $\mathcal{O}(\log n)$

<u>**Amortized** complexity</u>: $\mathcal{O}(1)$ – Potential function: number of pennants

Union

- Mimics addition of two numbers in binary (with carry flag):

```
BAG-UNION(S1, S2)
y = NULL        // The "carry" bit.
for k = 0 to r
        (S1[k], y) = FA(S1[k], S2[k], y)
```

x	y	z	s	c
0	0	0	NULL	NULL
1	0	0	x	NULL
0	1	0	y	NULL
0	0	1	z	NULL
1	1	0	NULL	PENNANT-UNION(x,y)
1	0	1	NULL	PENNANT-UNION(x,z)
0	1	1	NULL	PENNANT-UNION(y,z)
1	1	1	x	PENNANT-UNION(y,z)

Worst-case complexity: $\mathcal{O}(\log n)$

Splitting

- Mimics the division by 2 of a number in binary (right shifting).

$\rightarrow$ every power 2^k is halved in $2^{k-1}, 2^{k-1}$.

```
BAG-SPLIT(S1)
S2 = BAG-CREATE()
y = S1[0]
S1[0] = NULL
for k = 1 to r
    if S1[k] ≠ NULL
        S2[k-1] = PENNANT-SPLIT(S1[k])
        S1[k-1] = S1[k]
        S1[k] = NULL
if y ≠ NULL
    BAG-INSERT(S1, y)
return S2
```

<u>Worst-case complexity</u>: $\mathcal{O}(\log n)$

Splitting

- Mimics the division by 2 of a number in binary (right shifting).

$\rightarrow$ every power 2^k is halved in $2^{k-1}, 2^{k-1}$.

```
BAG-SPLIT(S1)
    S2 = BAG-CREATE()
    y = S1[0]
    S1[0] = NULL
    for k = 1 to r
        if S1[k] ≠ NULL
            S2[k-1] = PENNANT-SPLIT(S1[k])
            S1[k-1] = S1[k]
            S1[k] = NULL
    if y ≠ NULL
        BAG-INSERT(S1, y)
    return S2
```

halving of 2^k

<u>Worst-case complexity</u>: $\mathcal{O}(\log n)$

Splitting

- Mimics the division by 2 of a number in binary (right shifting).

$\rightarrow$ every power 2^k is halved in $2^{k-1}, 2^{k-1}$.

```
BAG-SPLIT(S1)
S2 = BAG-CREATE()
y = S1[0]
S1[0] = NULL
for k = 1 to r
    if S1[k] ≠ NULL
        S2[k-1] = PENNANT-SPLIT(S1[k])
        S1[k-1] = S1[k]
        S1[k] = NULL
if y ≠ NULL
    BAG-INSERT(S1, y)
return S2
```

handling with the unique pennant with 1 node (index k=0)

Worst-case complexity: $\mathcal{O}(\log n)$

Application to PBFS

```
PBFS(G, v0)
1   parallel for each vertex v ∈ V(G) − {v0}
2       v.dist = ∞
3   v0.dist = 0
4   d = 0
5   V0 = BAG-CREATE()
6   BAG-INSERT(V0, v0)
7   while ¬BAG-IS-EMPTY(Vd)
8       Vd+1 = new reducer BAG-CREATE()
9       PROCESS-LAYER(revert Vd, Vd+1, d)
10      d = d + 1
```

```
PROCESS-LAYER(in-bag, out-bag, d)
11  if BAG-SIZE(in-bag) < GRAINSIZE
12      for each u ∈ in-bag
13          parallel for each v ∈ Adj[u]
14              if v.dist == ∞
15                  v.dist = d + 1          // benign race
16                  BAG-INSERT(out-bag, v)
17      return
18  new-bag = BAG-SPLIT(in-bag)
19  spawn PROCESS-LAYER(new-bag, out-bag, d)
20  PROCESS-LAYER(in-bag, out-bag, d)
21  sync
```

<u>Number of steps</u>: n/p (lines 1-2) $+$ m/p (lines 13-16)

$+$ D $\times$ number of steps needed for all splits/unions at a layer

The recursion depth for all union/split operations is in $\mathcal{O}(\log n)$. Each operation takes $\mathcal{O}(\log n)$ steps.

$\Longrightarrow \mathcal{O}((n+m)/p + D \log^2 n)$ rounds.

PBFS with distributed memory

- Suited to very large graphs (that do not hold on a single machine).

- Each processor owns a private partial view of the graph (e.g., vertices/edges may be partitioned between processors).

$\rightarrow$ Since there is no shared memory, processors do not have all information needed to perform a search in the graph.

- Whenever a processor visits some vertices in its view, it must notify to other processors by message passing.

$\rightarrow$ induces communication overhead.

1-D partitioning

- Vertices are partitioned across processors (i.e., every vertex is owned by a unique processor).

- Each processor has private containers for its frontier and next frontier.

- Whenever we process the frontier at a processor, we sometimes insert in the next frontier some neighbors owned by other processors. This must be notified by messages so that they can update their next frontier.

- The algorithm stops when the frontiers of *all* processors are empty.

Implementation

```
define 1_D_distributed_memory_BFS( graph(V,E), source s):
    //normal initialization
    for all v in V do
        d[v] = -1;
    d[s] = 0; level = 0; FS = {}; NS = {};
    //begin BFS traversal
    while True do:
        FS = {the set of local vertices with level}
        //all vertices traversed
        if FS = {} for all processors then:
            terminate the while loop
        //construct the NS based on local vertices in current frontier
        NS = {neighbors of vertices in FS, both local and not local vertices}
        //synchronization: all-to-all communication
        for 0 <= j < p do:
            N_j = {vertices in NS owned by processor j}
            send N_j to processor j
            receive N_j_rcv from processor j
        //combine the received message to form local next vertex frontier then updat
level for them
        NS_rcv = Union(N_j_rcv)
        for v in NS_rcv and d[v] == -1 do
            d[v] = level + 1
```

A* and Parallelism

- Due to its applications to very large graphs, sometimes even infinite, the possibility to parallelize A* has been considered very early.

- The basic strategy consists in each processor handling with private open/closed sets. However, properties of the algorithm and conditions of termination may become more subtle to analyse (e.g., a processor with an empty open set should not necessarily stop the search).

- Various strategies studied in the literature (PLA*,PRA*, etc.).

Hash Distributed A* (HDA*)

- Each vertex is owned by one processor. The processor owning a vertex is chosen according to a **hash function** $h : V \to \{0, 1, \ldots, p-1\}$.

Remark: ensures load balancing if the hash function is uniform (or at least universal).

- Every processor i visits one vertex x_i from its private open set (if it is nonempty). If some neighbor y_i of x_i needs to be (re)open, then we notify processor $h(y_i)$ so that it inserts y_i in its own private open set.

- The algorithm stops whenever all local open sets are empty, or we have reached our target in the graph (requires a synchronization mechanism, such as a barrier).

Questions