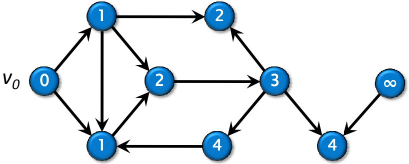
A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducer Hyperobjects)

Charles E. Leiserson   Tao B. Schardl

MIT Computer Science and Artificial Intelligence Laboratory

22nd ACM Symposium on Parallel Algorithms and Architectures
2010

Breadth-first search (BFS)



Problem: Given an unweighted graph $G = (V, E)$ and a designated starting vertex v_0, find the shortest path distance from v_0 to all other $u \in V$.

- Guarantee that the vertices are visited in ***breadth-first order***: For all distances d, all vertices that are d away from v_0 must be visited before any vertex of distance $d + 1$.
- We want a parallel algorithm to solve this problem.

Serial BFS

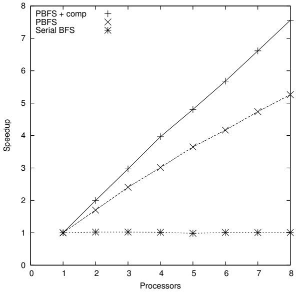
```
SERIAL-BFS(G = (V, E), v0)
1   for each vertex u ∈ V − {v0}
2       u.dist = ∞
3   v0.dist = 0
4   Q = {v0}
5   while Q ≠ ∅
6       u = DEQUEUE(Q)
7       for each v ∈ V such that (u, v) ∈ E
8           if v.dist == ∞
9               v.dist = u.dist + 1
10              ENQUEUE(Q, v)
```

- The queue Q is a FIFO queue.
- The distance of vertex $u = \text{DEQUEUE}(Q)$ in line 6 is monotonically increasing.
- Consequently, vertices are visited in breadth-first order.

This algorithm does not parallelize well.

- FIFO queue is a serial bottleneck.
- Parallelizing the **for** loops gives $O(E/V)$ parallelism, which is puny for sparse graphs.

Summary of results

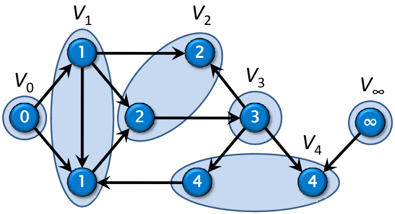


- We have designed a parallel breadth-first search algorithm, called PBFS, and we have implemented PBFS using Cilk++.
- PBFS obtains $5\times$ to $6\times$ speedup on eight processing cores on many real-world benchmark graphs.
- When run serially, PBFS is competitive with SERIAL-BFS.
- The theoretical running time of PBFS on P processors is $O((V + E)/P + D \lg^3(V/D))$.

Outline

Outline

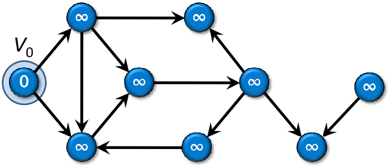
Strategy for parallelizing breadth-first search



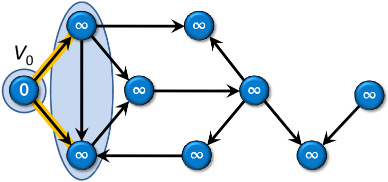
Strategy: consider the graph in layers.

- The dth ***layer*** of G is the set V_d of vertices that are all at distance d from v_0.
- Breadth-first ordering: all vertices in V_d are visited before any vertex in V_{d+1}.
- We shall examine the layers V_d serially, but
- For each layer V_d, we shall process all vertices in V_d *in parallel*.

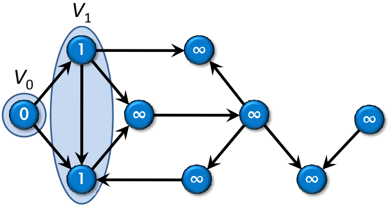
Strategy for parallelizing breadth-first search



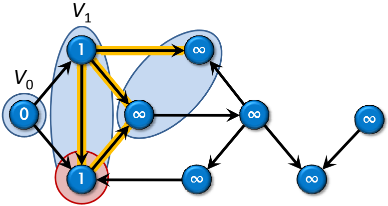
Strategy for parallelizing breadth-first search



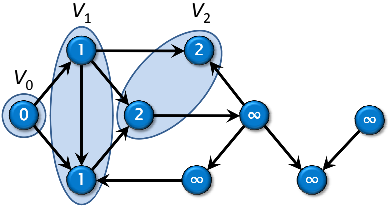
Strategy for parallelizing breadth-first search



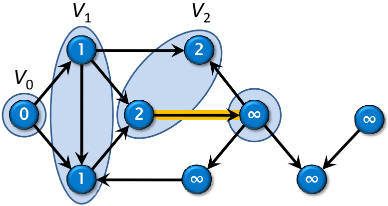
Strategy for parallelizing breadth-first search



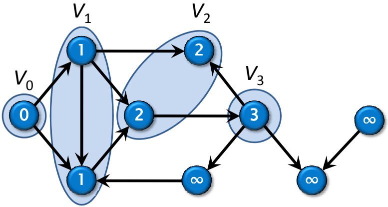
Strategy for parallelizing breadth-first search



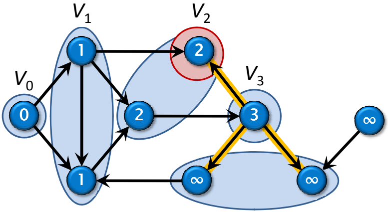
Strategy for parallelizing breadth-first search



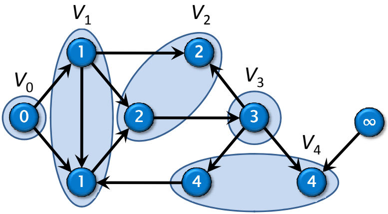
Strategy for parallelizing breadth-first search



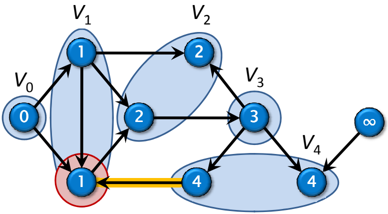
Strategy for parallelizing breadth-first search



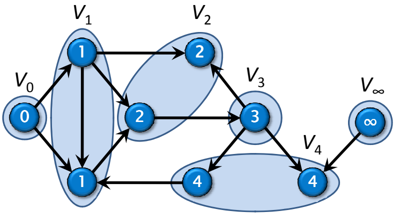
Strategy for parallelizing breadth-first search



Strategy for parallelizing breadth-first search



Strategy for parallelizing breadth-first search



Outline

Storing a layer of the graph

Problem: We need a data structure to handle a single layer. Specifically, we need a data structure that does the following:

- It must store an unordered set of elements.
- It must support efficient parallel traversal of the stored elements.
- It must allow parallel workers to add elements simultaneously to the same structure.

Storing a layer of the graph

Solution: Use a ***bag*** — a multi-set data structure, which supports the following operations:

BAG-CREATE	Create a new, empty bag.
BAG-INSERT	Add an element to a bag.
BAG-SPLIT	Divide a bag into two equal-sized bags.
BAG-UNION	Combine the contents of two bags into a single bag.

Processing a layer

```
PROCESS-LAYER(in-bag, out-bag, d)
if BAG-SIZE(in-bag) < GRAINSIZE
    for each u ∈ in-bag
        parallel for each v ∈ Adj[u]
            if v.dist == ∞
                v.dist = d + 1        // benign race
                BAG-INSERT(out-bag, v)
    return
new-bag = BAG-SPLIT(in-bag)
spawn PROCESS-LAYER(new-bag, out-bag, d)
PROCESS-LAYER(in-bag, out-bag, d)
sync
```

Processing a layer

```
PROCESS-LAYER(in-bag, out-bag, d)
if BAG-SIZE(in-bag) < GRAINSIZE
        for each u ∈ in-bag
                parallel for each v ∈ Adj[u]
                        if v.dist == ∞
                                v.dist = d + 1          // benign race
                                BAG-INSERT(out-bag, v)  // malignant race
        return
new-bag = BAG-SPLIT(in-bag)
spawn PROCESS-LAYER(new-bag, out-bag, d)
PROCESS-LAYER(in-bag, out-bag, d)
sync
```

Cilk++ reducers

Cilk++ supports a type of parallel data structure, called a ***reducer***.

```
x = 10
x++
x += 3
x += -2
x += 6
x--
x += 4
x += 3
x++
x += -9
```

```
x = 10
x++
x += 3
x += -2
x += 6
x' = 0
x'--
x' += 4
x' += 3
x'++
x' += -9
x += x'
```

```
x = 10
x++
x += 3
x' = 0
x' += -2
x' += 6
x'--
x'' = 0
x'' += 4
x'' += 3
x''++
x'' += -9
x += x'
x += x''
```

Cilk++ reducers

Cilk++ supports a type of parallel data structure, called a ***reducer***.

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

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

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

Cilk++ reducers

Cilk++ supports a type of parallel data structure, called a ***reducer***.

$x = 10$
$x{+}{+}$
$x \mathrel{+}= 3$
$x \mathrel{+}= -2$
$x \mathrel{+}= 6$
$x{-}{-}$
$x \mathrel{+}= 4$
$x \mathrel{+}= 3$
$x{+}{+}$
$x \mathrel{+}= -9$

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

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

Cilk++ reducers

We use the bag as a Cilk++ reducer to solve our malignant race.

- After stealing a task, a worker starts executing the task with a local, "identity" copy of a new reducer.
- Each worker freely manipulates its local copy with write-only update operations.
- As tasks return, the workers' local copies are combined together into a single data structure using REDUCE operations.
- If REDUCE is associative, then the program has serial semantics.

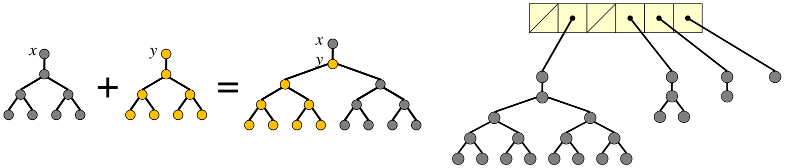
Cilk++ reducers

We use the bag as a Cilk++ reducer to solve our malignant race.

- After stealing a task, a worker starts executing the task with a local, “identity” copy of a new reducer.
  - For bags, the identity is an empty bag.
- Each worker freely manipulates its local copy with write-only update operations.
  - For bags, the update operation is BAG-INSERT.
- As tasks return, the workers’ local copies are combined together into a single data structure using REDUCE operations.
  - For bags, REDUCE = BAG-UNION.
- If REDUCE is associative, then the program has serial semantics.
  - For bags, BAG-UNION is not strictly associative, since the order of elements within a bag is nondeterministic. Bags have a notion of “logical associativity,” which is sufficient for PBFS’s correctness.

Outline

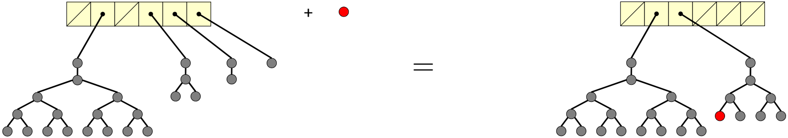
The bag data structure



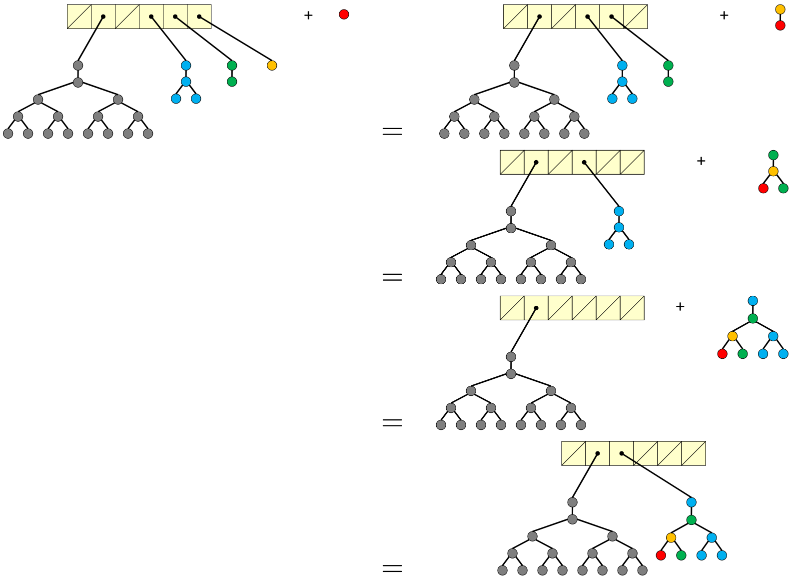
- A bag is made up of ***pennants*** — complete binary trees with extra root nodes, which store the elements.
  - Pennants may be split and combined in $O(1)$ time by changing pointers.
  - A pennant is only combined with another pennant of the same size.
- A bag is an array of pointers to pennants.
  - For all i, the ith entry in the array is either null or points to a pennant of size 2^i.
  - Intuitively, a bag acts much like a binary number.

The bag data structure — BAG-INSERT

Inserting an element works similarly to incrementing a binary number.

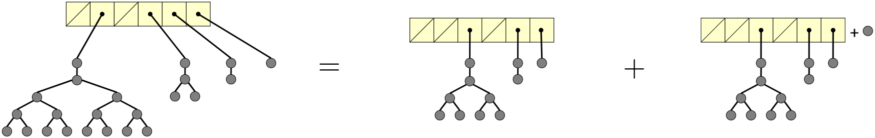


BAG-INSERT runs in $O(1)$ amortized time and $O(\lg n)$ worst-case time.



The bag data structure — BAG-SPLIT

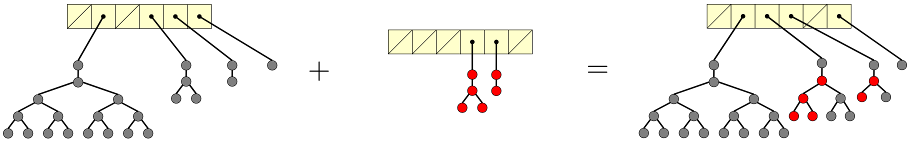
Splitting a bag works similarly to an arithmetic right shift.



BAG-SPLIT runs in $O(\lg n)$ time.

The bag data structure — BAG-UNION

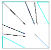
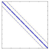
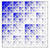
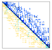
Unioning two bags is works similarly to adding two binary numbers.



BAG-UNION works in $O(\lg n)$ time.

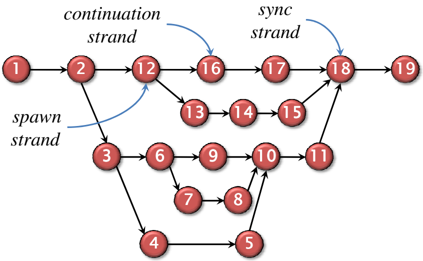
Outline

Empirical Results

Name *Description*	*Spy Plot*	$\lvert V \rvert$ $\lvert E \rvert$ D	Parallelism	$\frac{\text{PBFS } T_1}{\text{SERIAL BFS } T_1}$	PBFS T_1 / T_8
Kkt_power Optimal power flow, nonlinear opt.		2.05 M 12.76 M 31	104.09	0.705	6.102
Freescale1 Circuit simulation		3.43 M 17.1 M 128	153.06	1.120	5.145
Cage14 DNA electrophoresis		1.51 M 27.1 M 43	246.35	1.060	5.442
Wikipedia Links between Wikipedia pages		2.4 M 41.9 M 460	179.02	0.804	6.833
Grid3D200 3D 7-point finite-diff mesh		8 M 55.8 M 598	79.27	0.747	4.902
RMat23 Scale-free graph model		2.3 M 77.9 M 8	93.22	0.835	6.794
Cage15 DNA electrophoresis		5.15 M 99.2 M 50	675.22	1.058	5.486
Nlpkkt160 Nonlinear optimization		8.35 M 225.4 M 163	331.57	1.138	6.096

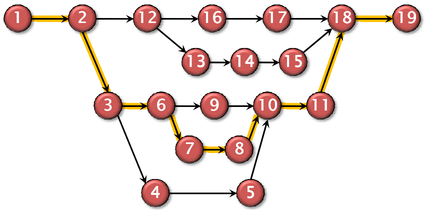
Outline

The dag model of computation



- We can model a Cilk++ program with a dag (directed acyclic graph) *A*.
- Each vertex in *A* corresponds to a ***strand*** — a sequence of serially executed instructions.
- Edges in *A* describe control dependencies between strands.

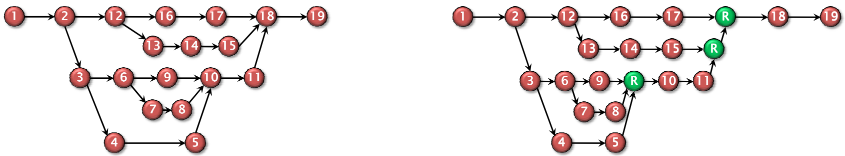
The dag model of computation



- ***Work*** $W(A)$: The sum of the lengths of all of the strands in A.
- ***Span*** $S(A)$: The length of the longest path in A.
- The Cilk++ scheduler guarantees that A runs in $T_P(A) \leq W(A)/P + O(S(A))$.
- ***Parallelism*** of A: $W(A)/S(A)$.
- This model does not accurately represent runtime system operations on reducers.

Outline

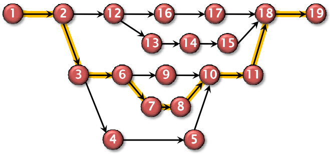
Modeling reducers

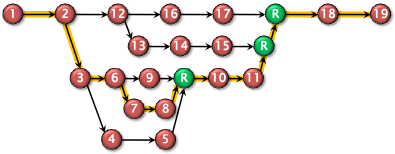


For a computation A:

- Consider the ***user dag*** User(A) — a dag where runtime system operations on reducers are not represented.
  - This dag models the computation as the user understands it.
- Insert REDUCE strands performed by the runtime system into User(A) before the sync strand that requires its completion to get a ***performance dag*** Perf(A).
- A delay-sequence argument proves that the performance of A is $T_P(A) \le W(\mathrm{Perf}(A))/P + O(S(\mathrm{Perf}(A)))$.
- We want a performance bound in terms of User(A).

Modeling reducers





Let τ be the worst-case running time of any REDUCE operation. In the paper, we show that:

- $S(\text{Perf}(A)) = O(\tau \cdot S(\text{User}(A)))$ and
- $W(\text{Perf}(A)) = W(\text{User}(A)) + O(\tau^2 P \cdot S(\text{User}(A)))$.

Outline

Analysis of PBFS

- PBFS's user dag has $O(V + E)$ work and $O(D \lg(V/D))$ span for bounded-degree input graphs.
- The worst-case cost of any BAG-UNION in PBFS is $O(\lg(V/D))$.
- Relating the user and performance dags for PBFS, we have $S(\text{PBFS}) = O(D \lg^2(V/D))$ and $W(\text{PBFS}) = O(V + E) + O(PD \lg^3(V/D))$.
- Consequently, we have $T_P(\text{PBFS}) = O((V + E)/P + D \lg^3(V/D))$.
- If $O((V + E)/P) \gg O(D \lg^3(V/D))$, then we expect linear speedup from PBFS. We define the ***effective parallelism*** of PBFS to be $O\left(\frac{V+E}{D \lg^3(V/D)}\right) \approx O\left(\frac{E}{D}\right)$.

Conclusion

- We have seen a parallel breadth-first search algorithm implemented in Cilk++, which uses a novel Cilk reducer for unordered sets.
- Future work includes:
  - Comparing the performance of PBFS versus an implementation that uses thread-local storage instead of reducers.
  - Augmenting PBFS to return a deterministic BFS tree.
  - Parallelizing other graph algorithms, such as weighted SSSP, max flow, or min-cost flow.
  - Parallelizing other non-numeric algorithms using Cilk technologies.

Thank you

Questions?

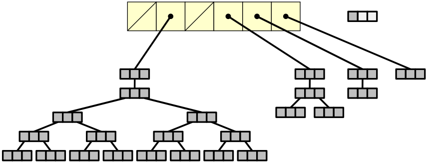
Cilk++ reducers

Reducers are implemented in Cilk++ with one additional optimization.

- After stealing a task, a worker starts executing the task with a local *null* copy of a reducer.
  - For example, this null copy may be a NULL pointer to a bag.
- The first time the worker tries to manipulate its local copy of the reducer after a steal, the runtime system initializes the reducer using a CREATE-IDENTITY operation.
  - For bags, CREATE-IDENTITY = BAG-CREATE.
- Modeling CREATE-IDENTITY calls in theory is similar (and simpler) than modeling REDUCE calls.

Optimizing the bag data structure

We can improve the real-world efficiency of the bag by storing an array of data at each node.



- Each node in a pennant stores a fixed-size array of data, which is guaranteed to be full.
- The bag stores an extra fixed-size array of data, called the ***hopper***, which may not be full.
- Inserts first attempt to insert into the hopper. Once the hopper is full, a new, empty hopper is created while the old hopper is inserted into the bag using the original algorithm.

With this optimization, the common case for BAG-INSERT is exactly like enqueueing a vertex in a FIFO queue.

Locked PBFS

To simplify theoretical analysis, we analyze a locked version of PBFS.

```
PROCESS-LAYER(in-bag, out-bag, d)
if BAG-SIZE(in-bag) < GRAINSIZE
    for each u ∈ in-bag
        parallel for each v ∈ Adj[u]
            if v.dist == ∞
                if TRY-LOCK(v)
                    if v.dist == ∞
                        v.dist = d + 1
                        BAG-INSERT(out-bag, v)
                    RELEASE-LOCK(v)
    return
new-bag = BAG-SPLIT(in-bag)
spawn PROCESS-LAYER(new-bag, out-bag, d)
PROCESS-LAYER(in-bag, out-bag, d)
sync
```

Relating the user and performance dags

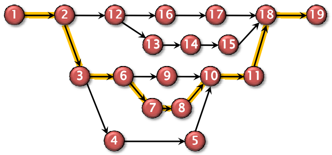
Lemma

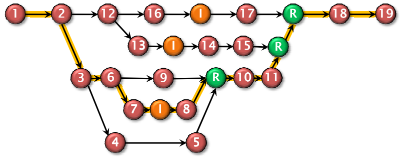
Consider a computation A, and let τ be the worst-case running time of any REDUCE *or* CREATE-IDENTITY *operation in A. We have* $S(\text{Perf}(A)) = O(\tau \cdot S(\text{User}(A)))$ *in expectation.*

Proof.

- Every successful steal may force a CREATE-IDENTITY operation.
- Every successful steal may force a REDUCE operation.

Relating the user and performance dags





Proof cont'd.

- Consider a critical path p in Perf(A).
- This path p corresponds to some path q in User(A), which has length at most $S(\text{User}(A))$.
- Since at most every node in q corresponds to a steal, the length of p is $O(\tau \cdot S(\text{User}(A)))$ in expectation.

□

Relating the user and performance dags

Lemma

Consider a computation A, and let τ be the worst-case running time of any REDUCE *or* CREATE-IDENTITY *operation in A. We have* $W(\mathrm{Perf}(A)) = W(\mathrm{User}(A)) + O(\tau^2 P \cdot S(\mathrm{User}(A)))$ *in expectation.*

Proof.

- The computation A contains all of the strands in $\mathrm{User}(A)$.
- At most $O(P \cdot S(\mathrm{Perf}(A)))$ steals occur during A's execution.
- Consequently, REDUCE and CREATE-IDENTITY strands contribute $O(\tau P \cdot S(\mathrm{Perf}(A)))$ additional work to $\mathrm{User}(A)$.
- From the previous lemma, we have $S(\mathrm{Perf}(A)) = O(\tau \cdot S(\mathrm{User}(A)))$.
- Therefore, we have $W(\mathrm{Perf}(A)) = W(\mathrm{User}(A)) + O(\tau^2 P \cdot S(\mathrm{User}(A)))$ in expectation.

$\square$