Randomized Parallel Marking in Mark-and-Copy Garbage Collectors

Jay Reynolds Freeman

SJSU – CS 262 – Spring 2012

- From my project in Computer Science 262, “Random Algorithms and Applications”, taught by Professor Teng Moh at San Jose (California) State University, spring 2012.
- Text: <u>Probability and Computing</u>, Michael Mitzenmacher and Eli Upfal. Cambridge, 2005.

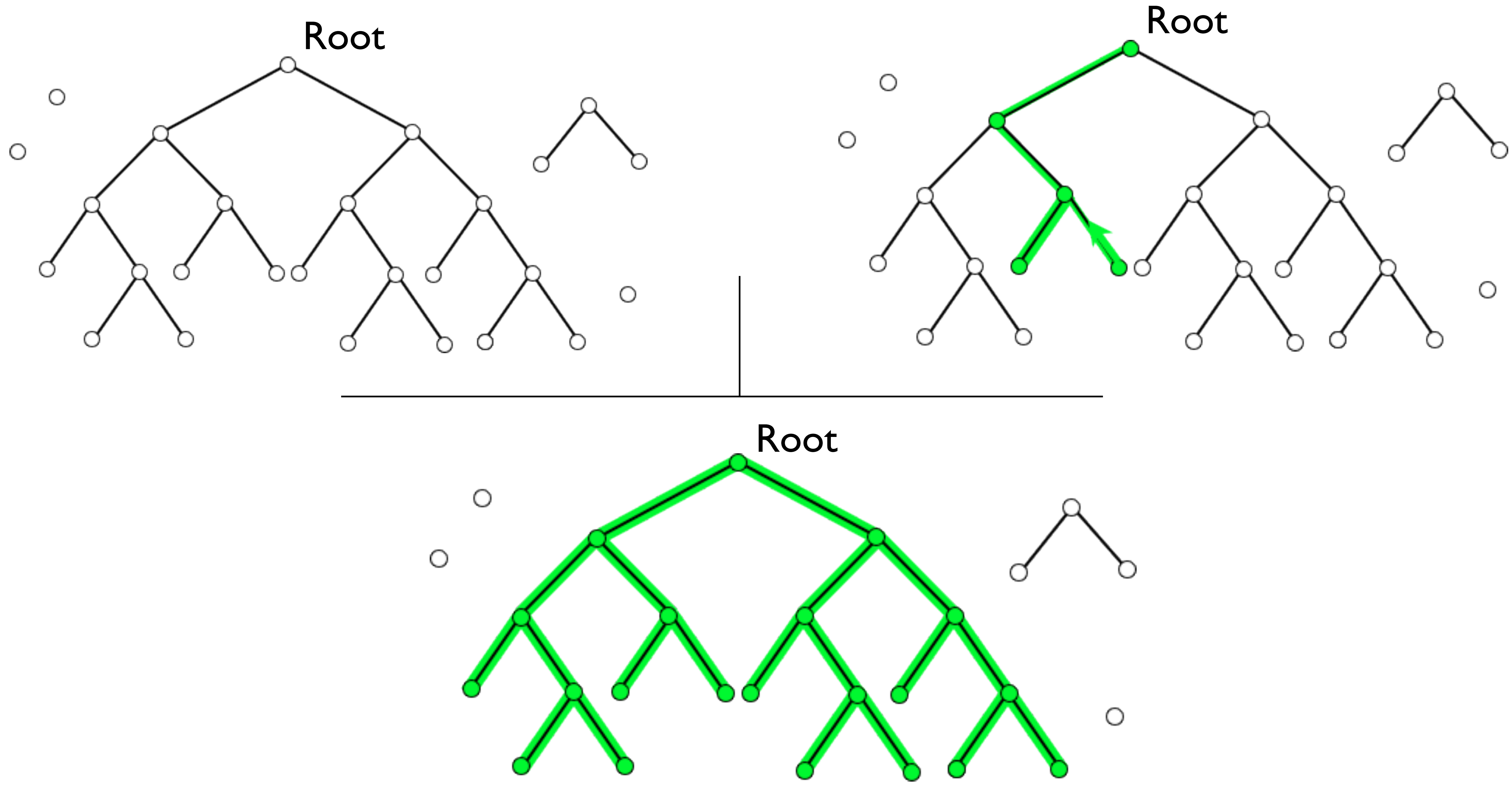
- Underlying problem:

  Parallel search in a complicated linked data structure.

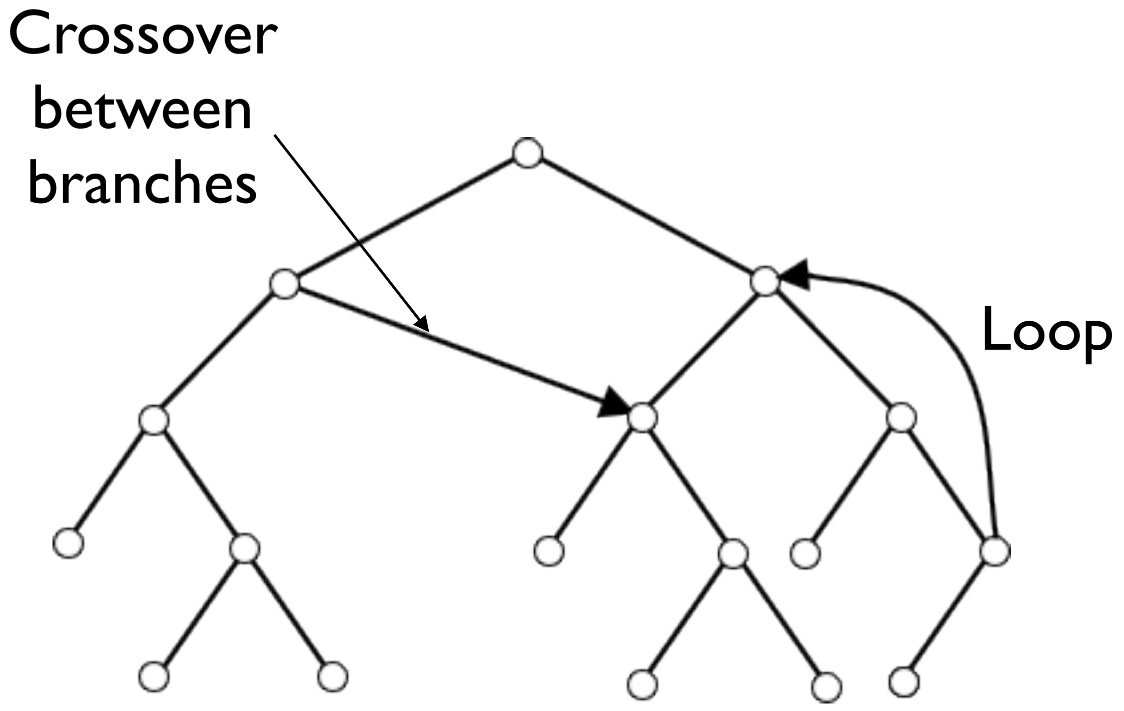
- Personal interest:

  I support an open-source shareware Lisp system (google "Wraith Scheme"), with a garbage collector of this kind.

- *Almost* depth-first traverse of a binary tree with preorder coloring.



- Complications:
  - Not a tree.
  - Loops.
  - Implementation details (messy, I will spare you).



- Not just Lisps:
    - Java
    - Objective C
- How can we use parallel hardware ...
    - In multicore machines?
    - In cluster computing?
    - Efficiently?
    - Robustly?

Can multiple threads cooperate?

- Easy with interprocess communication – <u>but</u> a major time bottleneck.
- What happens if one or more processes crash? (E.g., if using a cluster.)

To exploit parallel hardware efficiently, an algorithm must:

- Complete – find all the non-garbage.
- Avoid interprocess communication.
- Deal well with process failure.

The goal is different from the applications discussed in class.

- There are <u>many</u> O(n) ways to mark (n is the number of non-garbage nodes).
- We seek efficient use of parallel hardware – divide the constant in O(n) by a number approaching the number of parallel processors available.

I found <u>no</u> prior work on the use of randomness in parallel garbage collection.

- Searched the table of contents for "Random Structures and Algorithms".
- Usual Google search, et cetera.
- Searched US Patent Office for patents and patent applications on random garbage collection: None.

- One paper described a random algorithm for a different kind of garbage collection, using reference counting:

  Kaoutar El Maghraoui and Travis Desell, 2003. "Randomized Distributed Garbage Collection", Rensselaer Polytechnic Institute. (http://en.scientificcommons.org/42386895)

  Not relevant to this project.

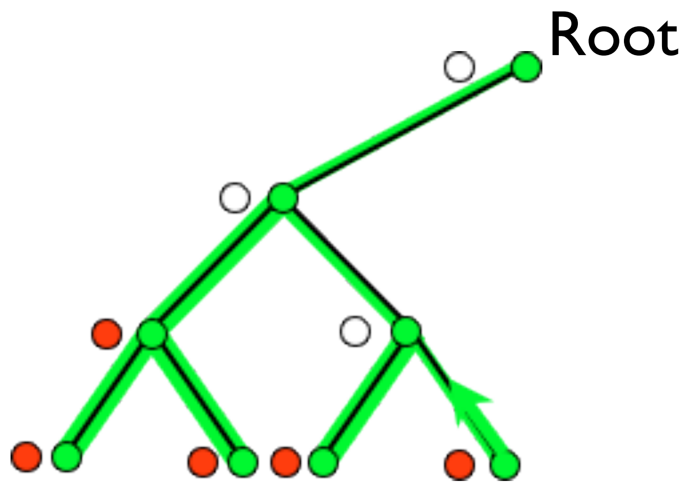
- Nothing relevant in the classic work on garbage collection:

Richard Jones and Rafael Lins, 1996. Garbage Collection: Algorithms for Automatic Dynamic Memory Management, Wiley.

Here’s my plan:

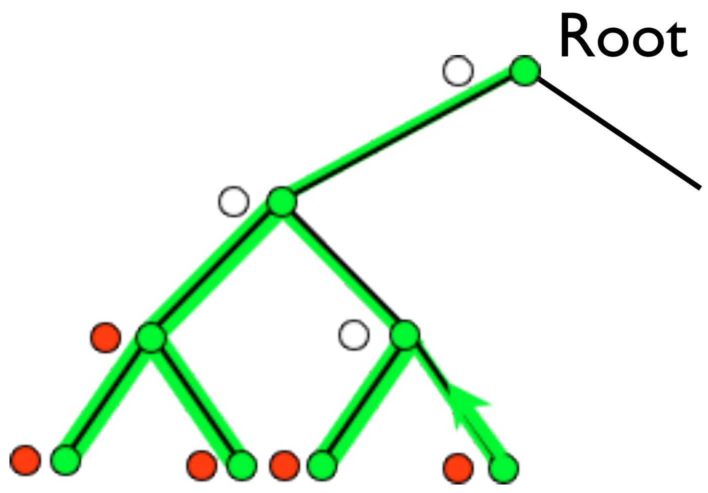
- Do <u>both</u> preorder and postorder coloring during the traverse.
- Preorder coloring marks each node as non-garbage.
- Postorder coloring asserts that the given node and all its descendants have been searched – no work remains to be done on any path through a postcolored node.

Example: Green circles are the preorder coloring; red circles are the postorder coloring.



- No process enters a node that has a postorder-coloring bit set.
- Such nodes I call <u>closed</u>.

Randomness: When a process has a choice – left and right child not closed – it decides randomly, with 50/50 probability, as when it first saw the root and points a, b, and c.

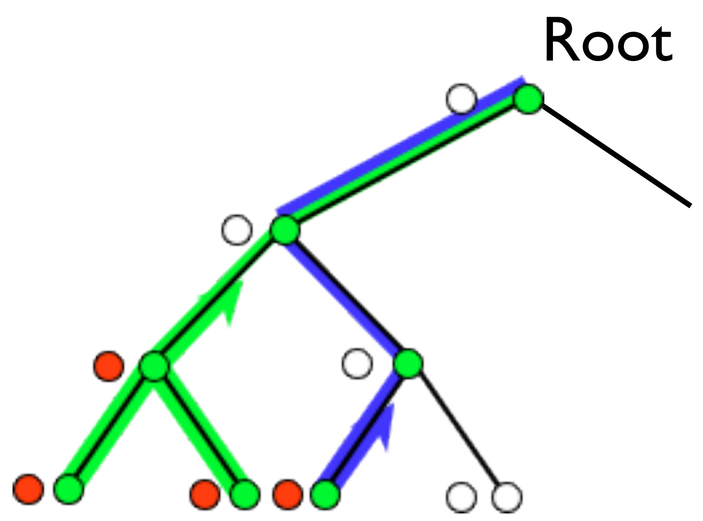


I was prepared for a more complex decision, but – getting ahead of myself – none was required.

Why does simple random choice work?

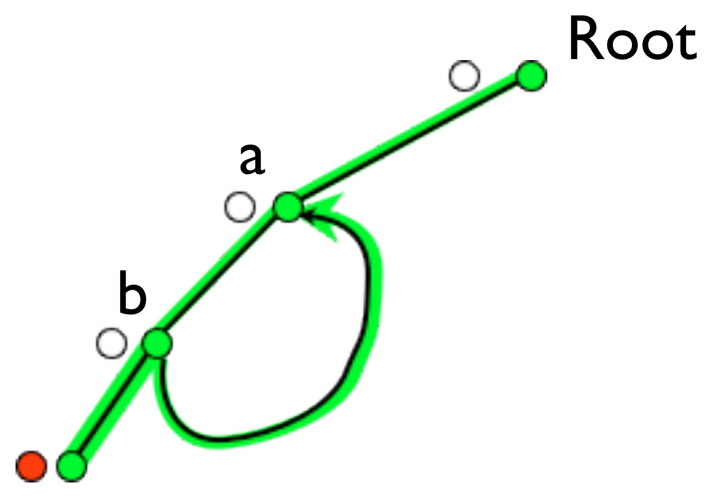
- Intuition: Processes only duplicate work when they randomly choose the same path through the graph.
- Not likely! For two processes, continuing on the same path is a geometric random variable with probability 0.5, and so on.
- Simulation confirms intuition.

Example:



- The blue and green processes each made the same random decision at the root, but at the next point down, they diverged. After diverging, they work in parallel, without duplication of effort.

It doesn't work for loops ...



- The green process is stuck in loop from a to b and back to a again, forever.
- There is a fix ...

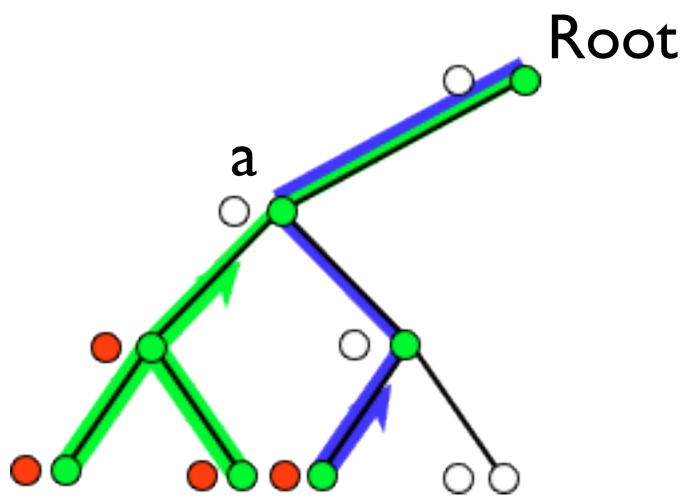
- Memory cost: One pointer per node. Many Lisps already have space for that.
- The first process to reach a node stores a record of where it came from -- a pointer to the node it was last at.
- No other process overwrites this record.
    - In race conditions, the process that writes the record is effectively the first to get there – it doesn't matter who wins.

Only enter marked nodes from the same direction as the first process to get there:

A process X may enter a node N only if ...

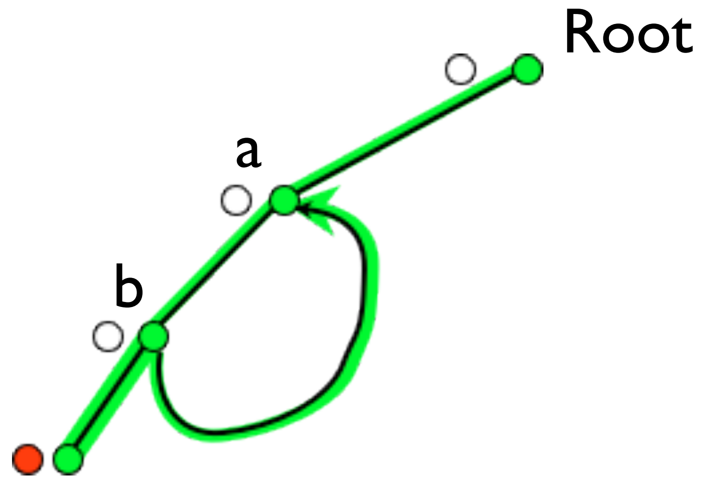
- N is not closed (not postorder colored, red herein), and **either** ...
    - N is not marked non-garbage (not preorder colored, green herein), **or**
    - Whatever process Y reached N first, came from the same node that X is coming from.

- If two processes are proceeding away from the root on the same branch, each can enter any node first reached by the other, since both come from the same parent node.



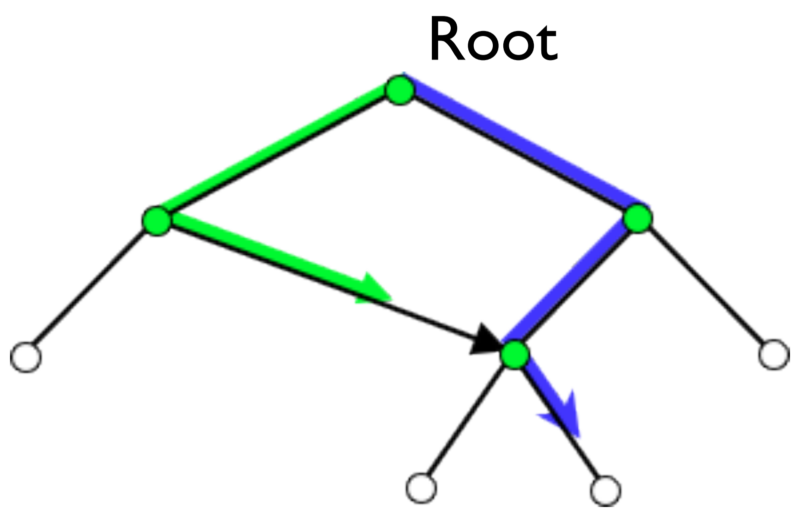
- The blue and green processes can both enter node a, because both approach it from the same parent node -- the root.

- In the case of a loop, when this process tries to enter node a the second time (the arrowhead) ...



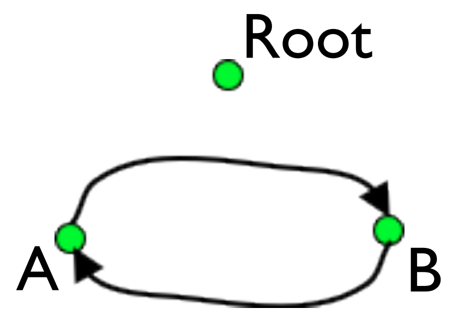
- It finds itself arriving from a different node (b) than the one written in the record (the root): So it cannot enter – it backtracks.

- This mechanism blocks "crossovers":



- But that is perhaps a small price to pay for an algorithm that does not enter an infinite loop.
- And the algorithm will complete.

Can the algorithm create a loop like this?



- We might imagine some sort of race condition involving two processes writing the "where did I come from" records.
- I think not, and have put a proof in an addendum, after the main body of the talk.

What if a process dies?

- Any node it may have been about to mark non-garbage, or mark closed, will have to be revisited.
- No mistake has been made, there is just some extra work to do, and not very much.
- Low cost!
- Fault tolerant!

- I did a little theory and a lot of simulation developing and testing this algorithm.
- Superficial lemmas about the expected times and speedups when two processes traverse a symmetric binary tree.
- “Superficial”: They clearly demonstrated the intuition about random choice efficiently distributing work.

- Simulations of increasing complexity ...
  - Unit of work: Visiting a node.
  - The n-process speedup is the ratio of:
    - The time for an isolated single process to traverse the entire structure, all by itself, to
    - The maximum time required by the slowest of the n cooperating processes.

- 3800 lines of C++, mostly using one simulation as boilerplate for the next.
- Simulated about ten variations of the algorithm.
  - Simplest: Preorder coloring of a symmetric binary tree.
  - Most complicated: Full algorithm, in data structure with loops and crossovers.

- Max tree depth: 28
- Max number of processes: Typically 10-12 less than tree depth.
- Tried OpenMP, but more realistic to list the processes and single-step them through one node at a time in random order.
- The final algorithm worked, and with number of processes at least order of ten less than tree depth, I <u>always got speedup close to the number of processes</u>.

Wraith Scheme:

- Modified the single-process garbage collector to use the new algorithm (still just one process – no parallelism yet).
- Tested extensively – Wraith Scheme has a large regression suite that bangs hard on the garbage collector.
- It worked.

A gotcha:

- Wraith Scheme is already multithreaded.
- Didn't play well with OpenMP.
- Insufficient time to re-architect Wraith Scheme to use multiple garbage collection threads by some other means than OpenMP.

But very promising!

Conclusions:

- Algorithm works.
  - Exploits parallelism well.
  - Costs one bit and one pointer per node.
- Implementable in a real Lisp.

Future Work:

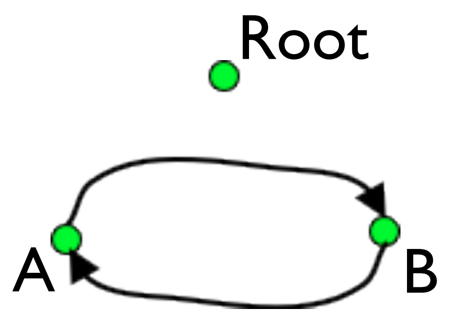
- Finish the Wraith Scheme implementation.
- Test and measure real speedup attained.
- I will probably do that ...

Randomized Parallel Marking in Mark-and-Copy Garbage Collectors

Jay Reynolds Freeman

(But wait, there's more ...)

Addendum



- The proof that this situation cannot occur is a little wordy for a pretty slide, but I have appended it in smaller text hereafter. I submit that the proof given clearly generalizes to any more complicated loop.

Two processes are assumed to start coloring from the root -- one process is described in each column. In the pseudocode, the consequent of each _if_ is indicated by indentation.

The times T1 through T12 are the specific times of the read or write of the global data structure, that is implicit in the given instruction. (Instructions may of course take many clocks, of which the read or write involves but one.) Assume writes are atomic, and also assume that reads and writes done by any one process are accomplished in the same order as the corresponding high-level instructions. I make no assumption about the order of execution of reads and writes done by different processes.

Note that for the unwanted loop to occur, all of the tests in the _if_ statements must pass.

```
      First Process                          Second Process

T1  if( node A is not green)               T7  if( node B is not green )
T2      color node A green                 T8      color node B green
T3      set A->cameFrom to Root            T9      set B->cameFrom to Root
T4      if( node B is not green )          T10     if( node A is not green )
T5          color node B green             T11         color node A green
T6          set B->cameFrom to A           T12         set A->cameFrom to B
```

Times T1 ...T6 are all distinct, and in order:

(1) $T1 < T2 < T3 < T4 < T5 < T6$.

Similarly,

(2) $T7 < T8 < T9 < T10 < T11 < T12$.

For the loop to exist, we must have that the two sets of A->cameFrom occur in the right order, so that the content of A->cameFrom is the unwanted value:

(3) $T3 < T12$,

and similarly for the sets of B->cameFrom:

(4) $T9 < T6$.

So let us **assume** that (3) and (4) are true, and then proceed with a proof by contradiction: For the if-test of T10 to pass, we must have

(5) $T10 < T2$,

and similarly,

(6) $T4 < T8$.

Now from (6) and (2) we conclude

(7) $T4 < T8 < T10$,

but by (5), we can extend (7) to get

(8) $T4 < T8 < T10 < T2$

which means that

(9) $T4 < T2$.

But from (1), we have $T4 > T2$, hence we have a contradiction with (9), so the assumption that both (3) and (4) are true must be false. (Remember that the inequalities are all strict, so no "equals" condition can destroy the contradiction.)

Q. E. D.