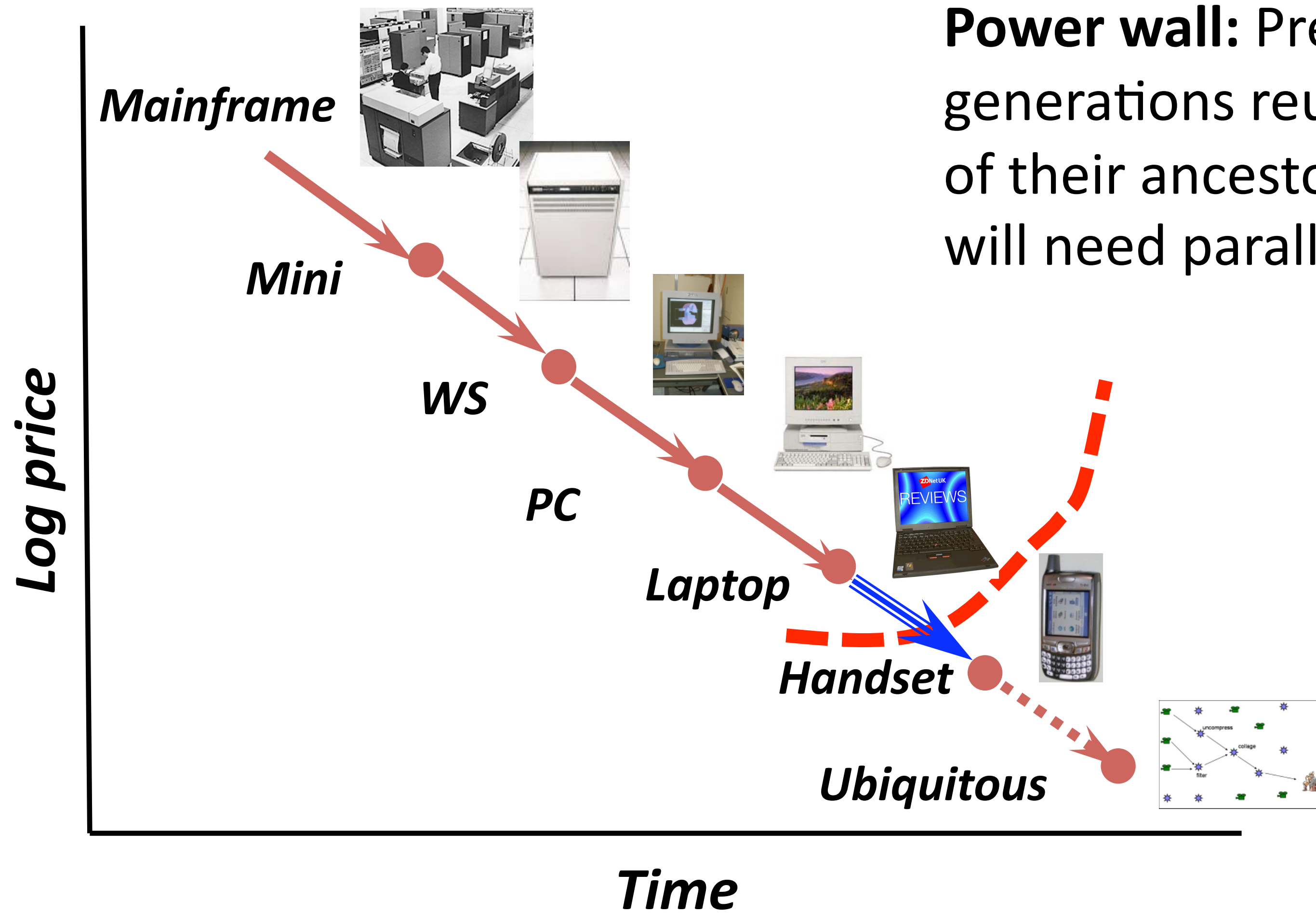
Parallelizing the Web Browser

Chris Jones, Rose Liu, Leo Meyerovich
Krste Asanovic, and Rastislav Bodik

ParLab

UC Berkeley

The Transition to Handhelds



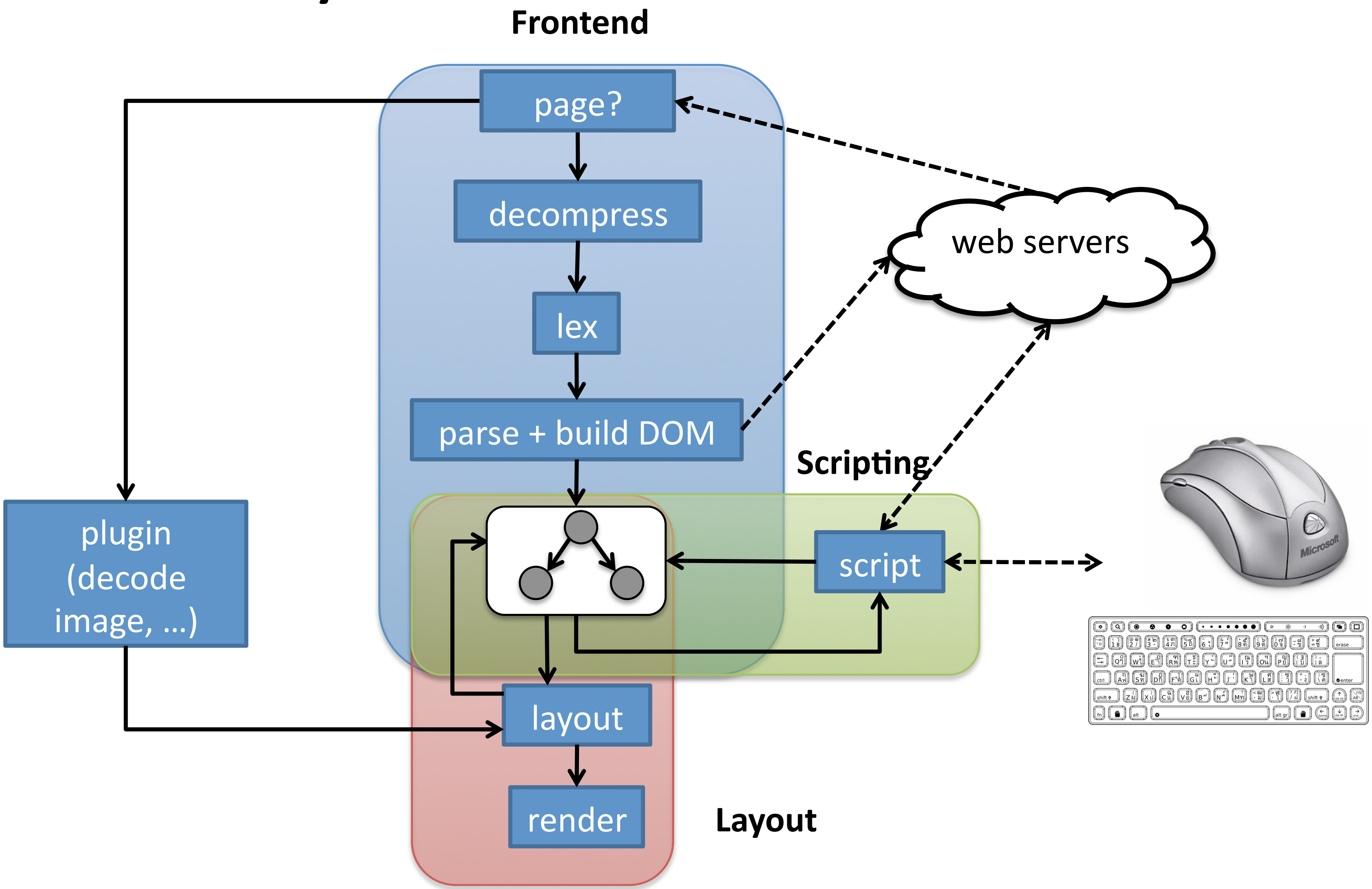
Power wall: Previous generations reused software of their ancestors. Mobiles will need parallel software.

Soon on mobile: 4-cores x 2-threads x 8-SIMD = 64-way parallelism

Why Parallelize a Browser?

- **Dominant application platform**
  - easy deployment: apps downloaded, JS portable
  - productive programming: scripting, layout
- **… but not on handhelds**
  - native frameworks for: iPhone, Google Android
  - slow: for Slashdot, Laptop: 3s => iPhone: 21s
- **Parallel browser may need new architecture**
  - ex: JavaScript relies on “gotos”, is too serial

Anatomy of a Browser



Project Status

1. **Developed *work-efficient* algorithms**

   work-efficient : no more work than sequential algo.

   – *layout:* parallel-map with a tiling optimization

   – *layout*: break up tree traversal into five parallel ones

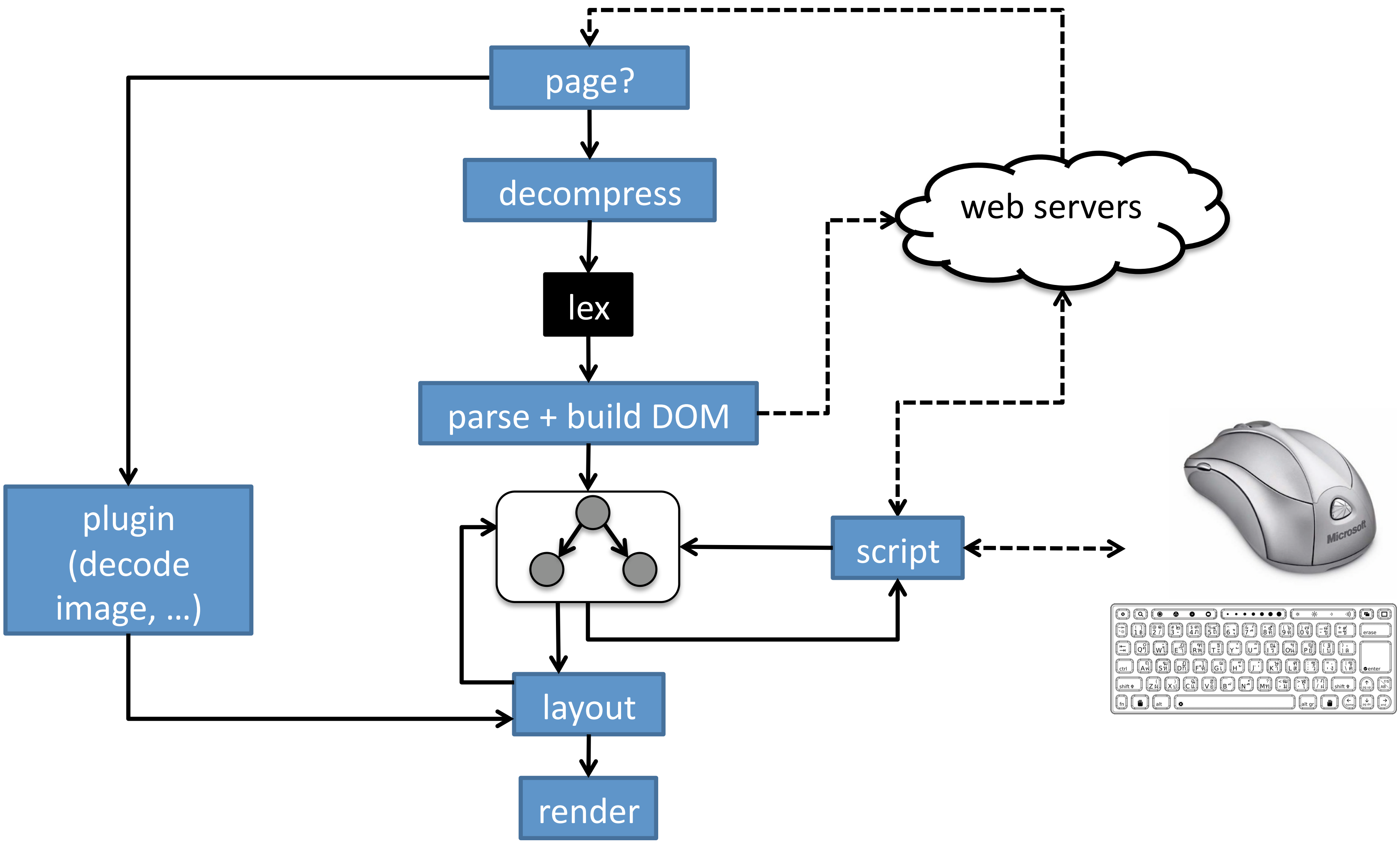
   – *lexing:* speculation to break sequential dependencies

2. **Reexamining the scripting programming model**

   – *programmer productivity*: from callbacks to actors

   – *performance*: adding structure to detect dependences

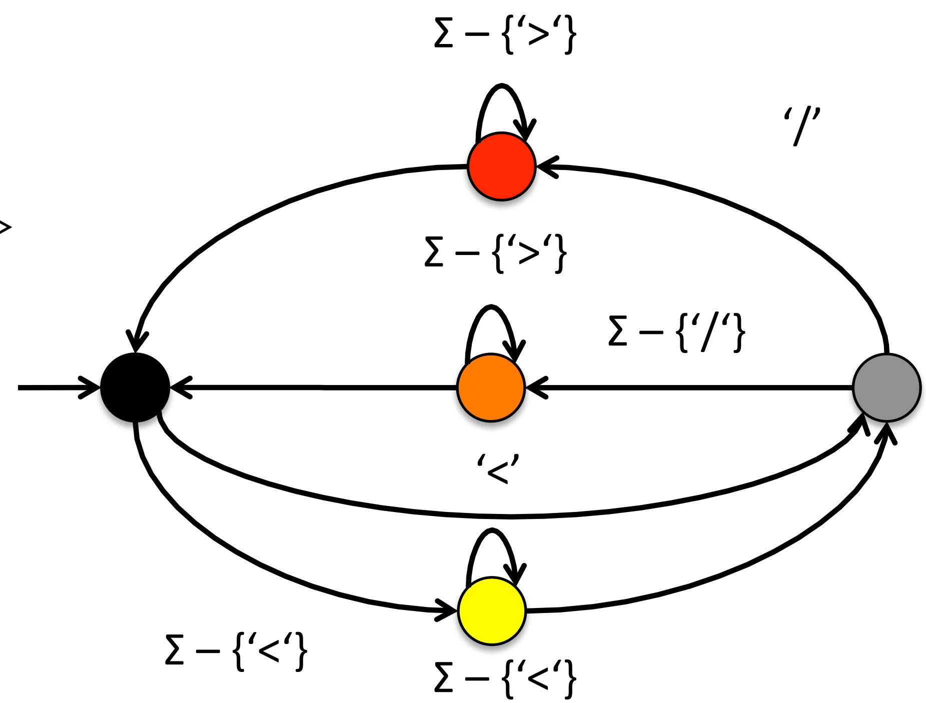
Frontend: Lexing



Lexing, from 10,000 feet

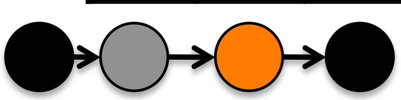
Goal: given lexical spec and input, find lexemes

```
STag     ::= <[^>]*>
Content  ::= [^<]+
ETag     ::= </[^>]*>
```



STag

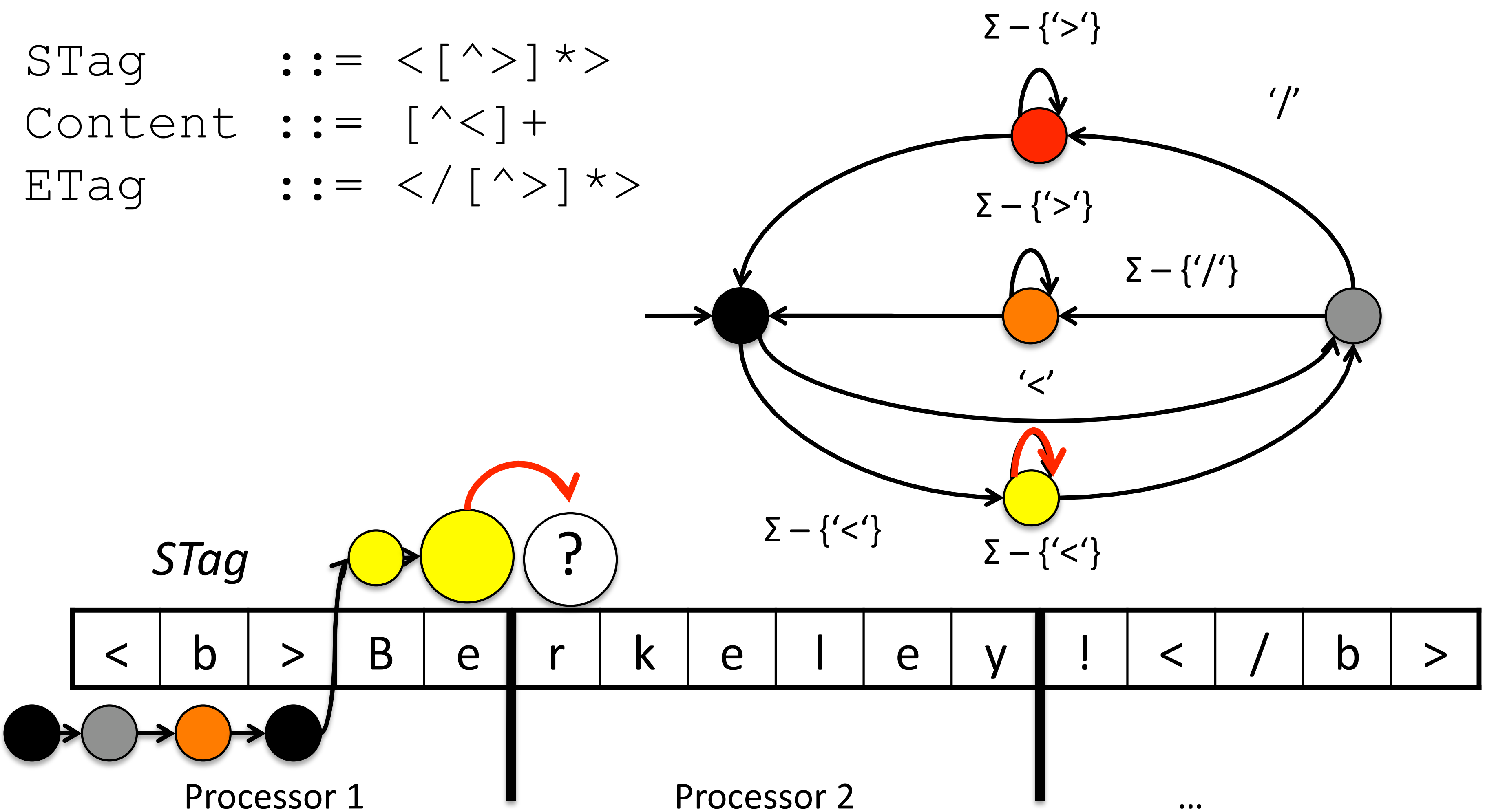
<	b	>	B	e	r	k	e	l	e	y	!	<	/	b	>



(label each character with its state)

Inherently Sequential?

```
STag     ::= <[^>]*>
Content  ::= [^<]+
ETag     ::= </[^>]*>
```

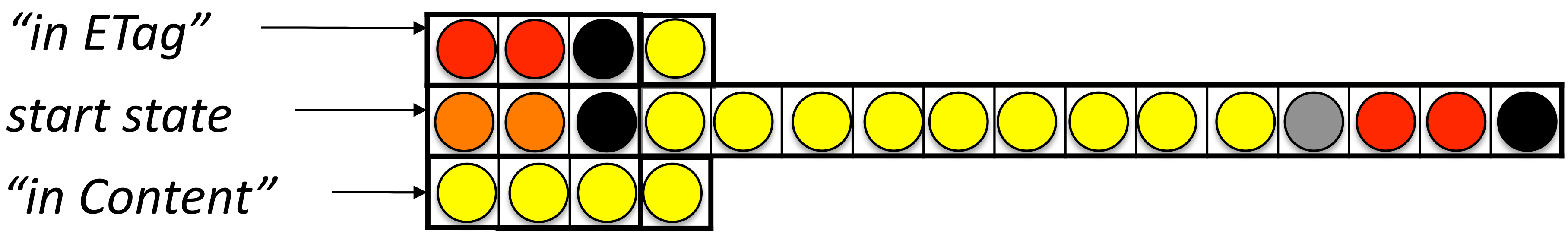


An observation

In lexing, irrespective of where DFA starts, it converges to a *stable, recurring* state

Lexing:

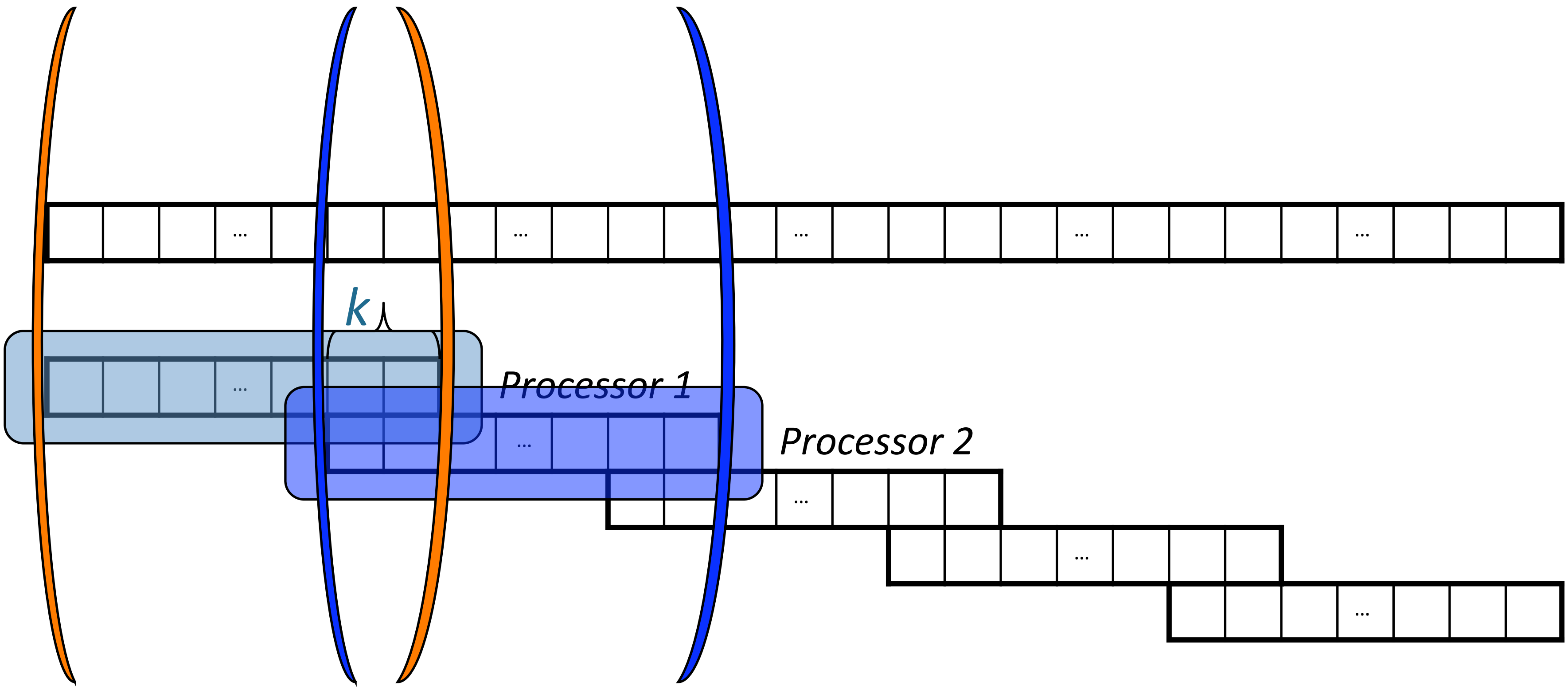
<	b	>	B	e	r	k	e	l	e	y	!	<	/	b	>



Parallel scans thus need not scan from all possible states, just one, yielding a work-efficient algorithm.

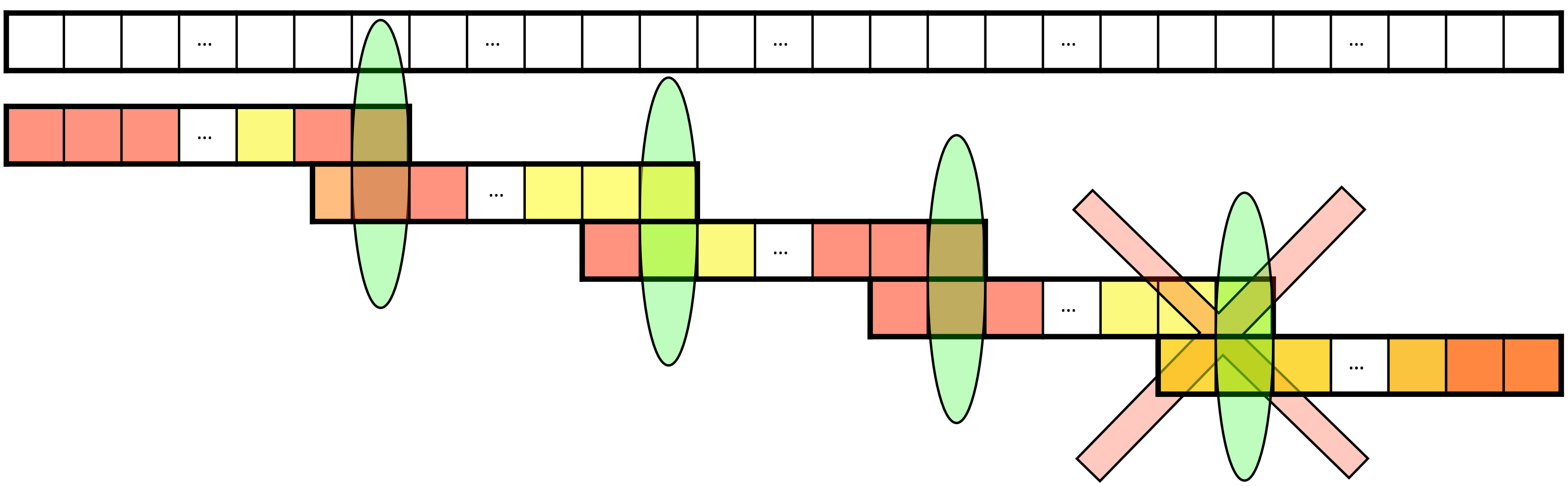
Our solution (1/2): Partition

- split input into blocks with *k*-character overlap
- scan in parallel; start block from a *tolerant* state

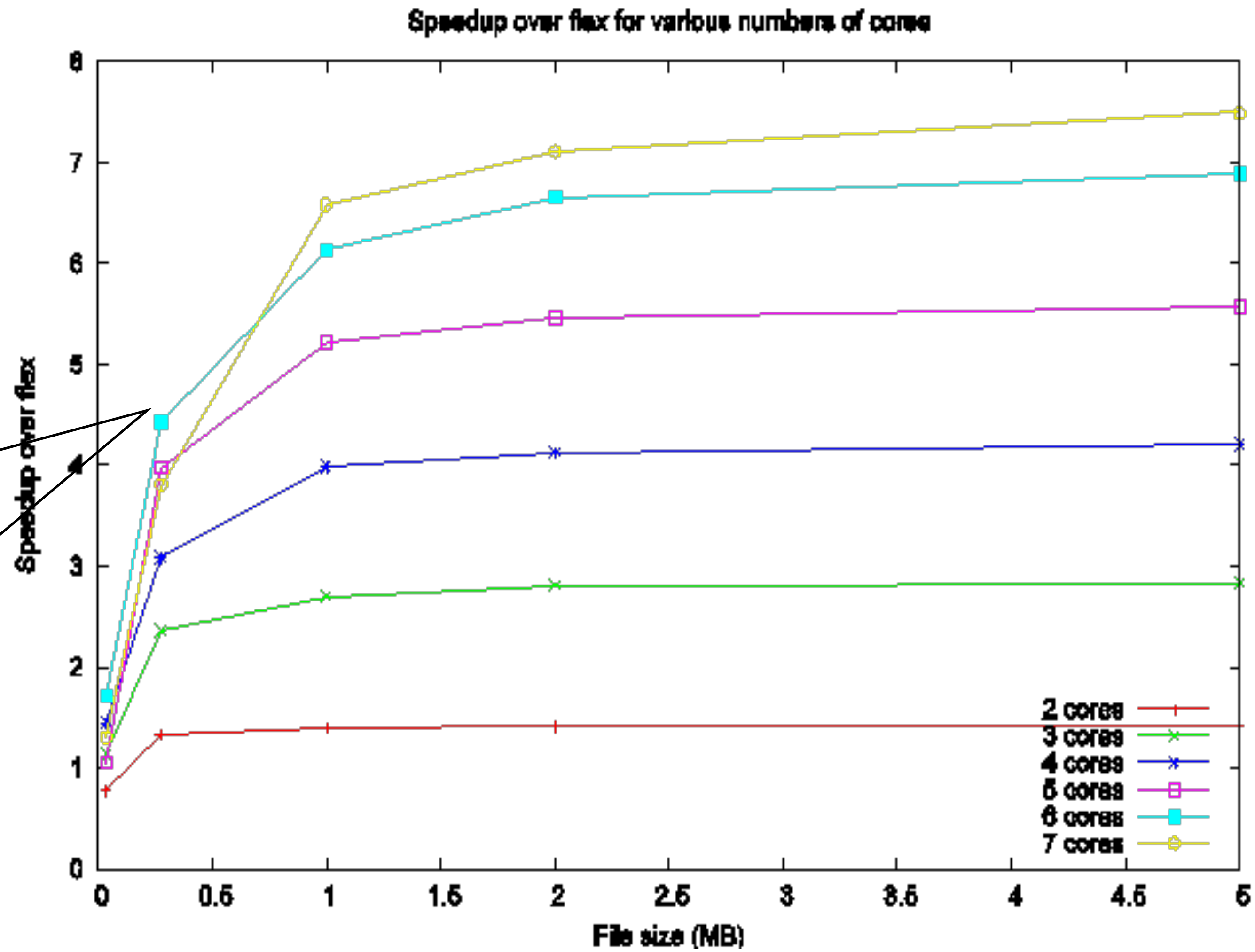


Our solution (2/2): Speculate

- split input into blocks with *k*-character overlap
- scan in parallel; start block from a *tolerant* state
- check if blocks converge: expected in *k*-overlap
- speculation may fail; if so, block is rescanned



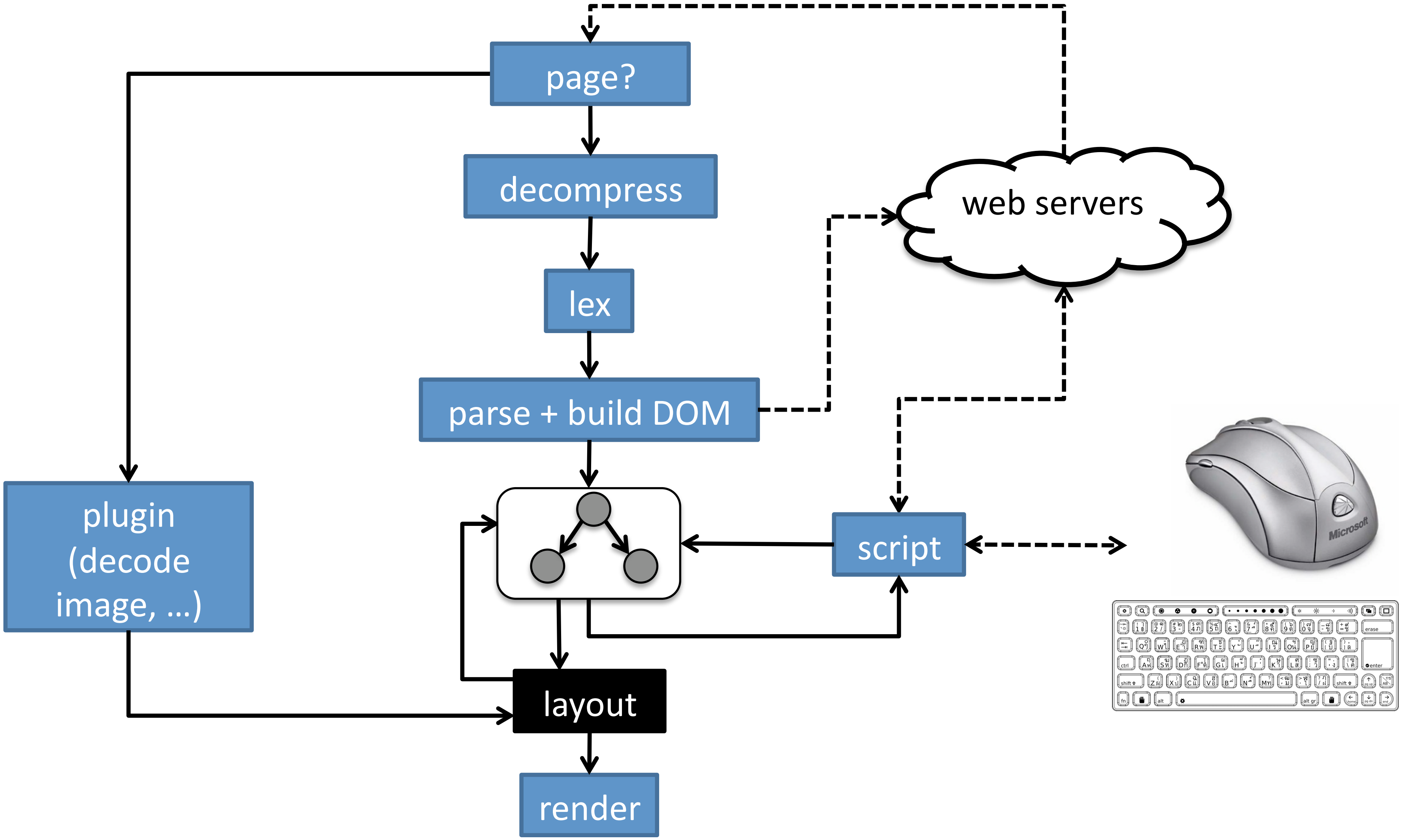
Speedup: Flex vs Cell



***today's page sizes**: 5 cores are 4.5x faster than flex*

baseline: (sequential) flex on the CELL main CPU

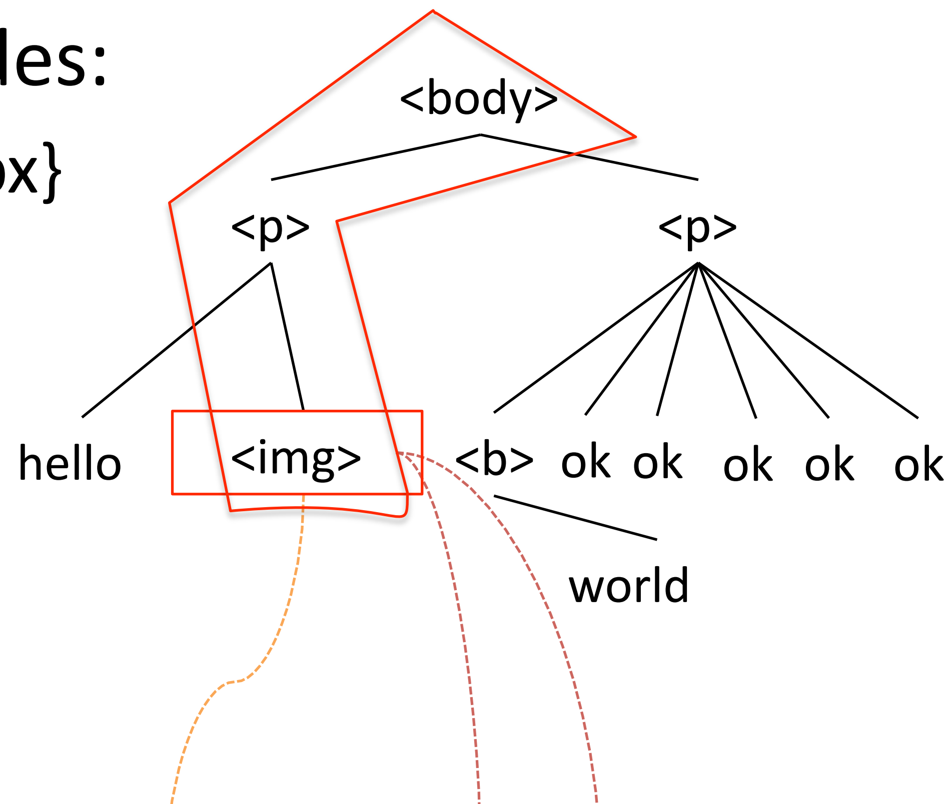
Layout Solving (1/2)



Rule Matching

Goal: Match rules with nodes:

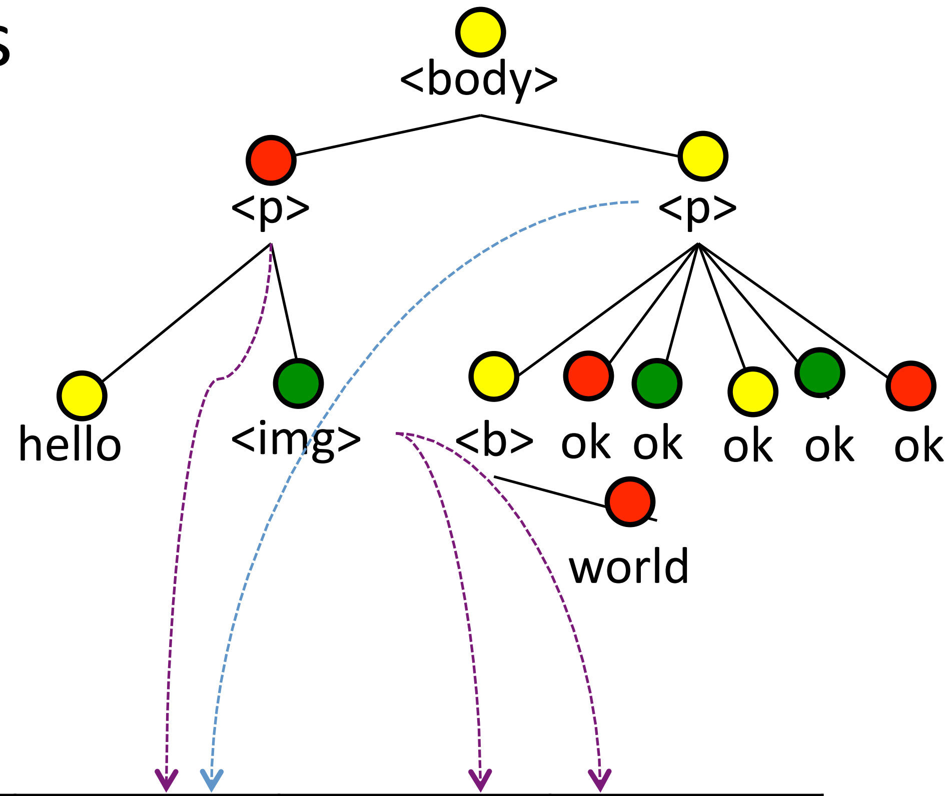
- a rule: p img { fontsize: 7px}
- match tag path
- path-rule matching
  - end with the same node
  - and are a substring



selectors	p	img	p img
properties	height=83%	width=100px float=left	fontsize=7px

Parallelization

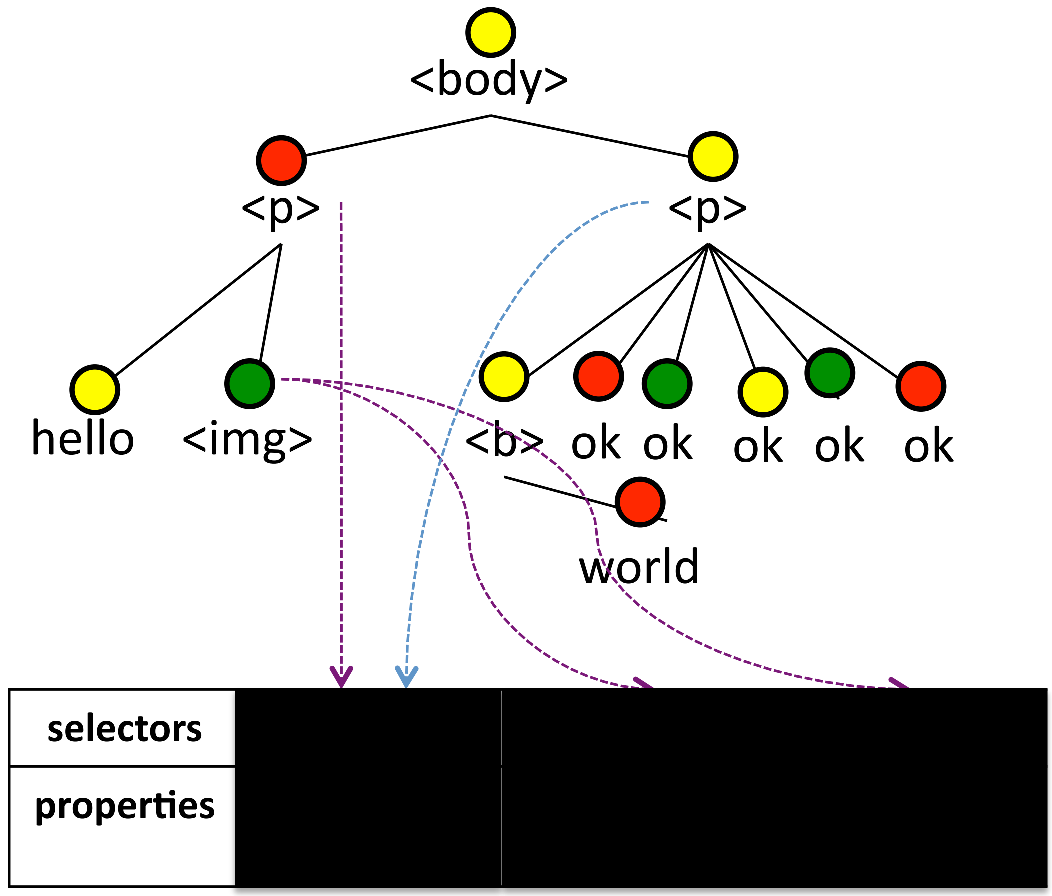
- 1000s nodes, 1000s rules
- Assign nodes to cores



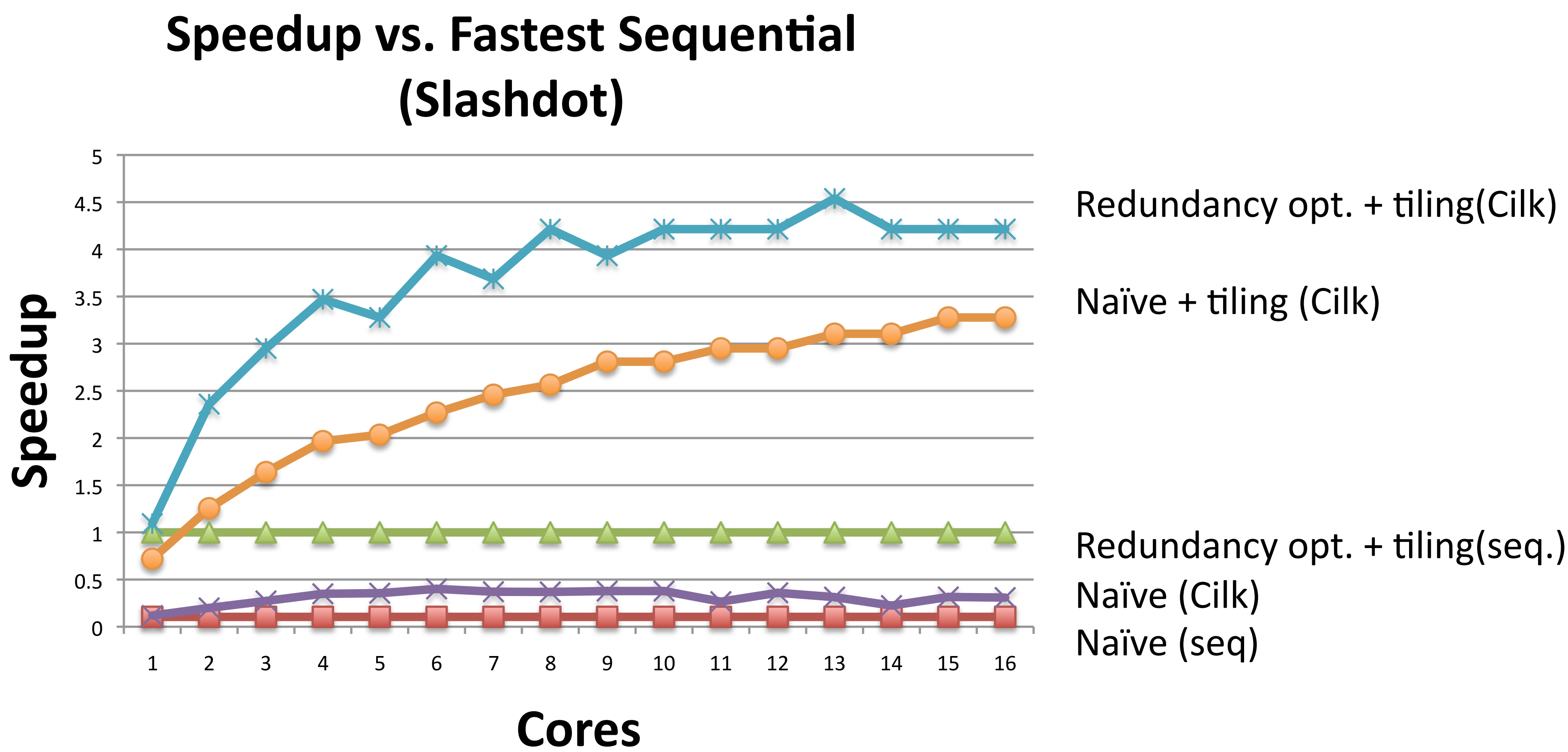
selectors	p	img	p img
properties	height=83%	width=100px float=left	fontsize=7px

Tiling for Caches

Problem: all the nodes + selectors might not fit in cache!

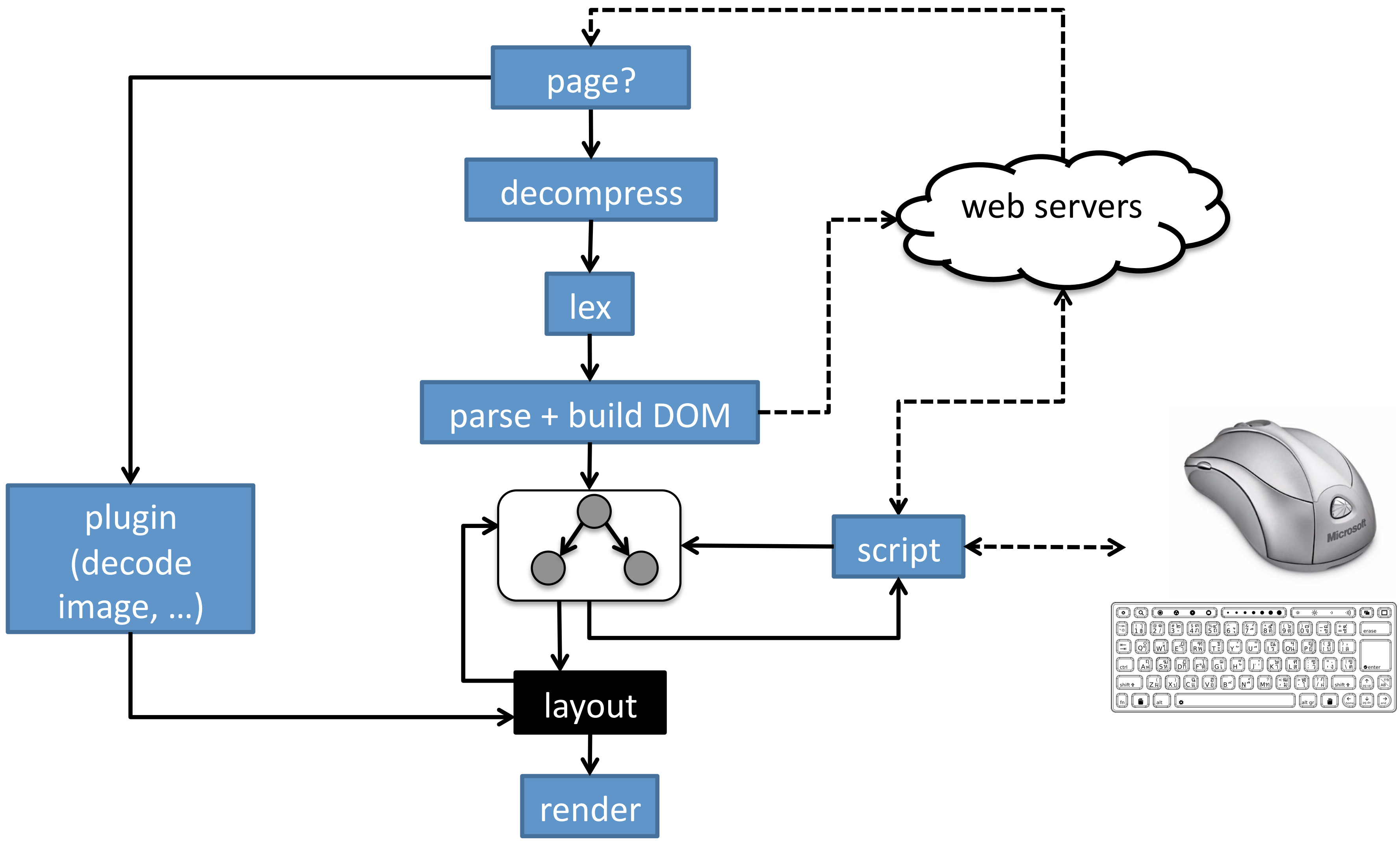


Speedup (Cilk++)

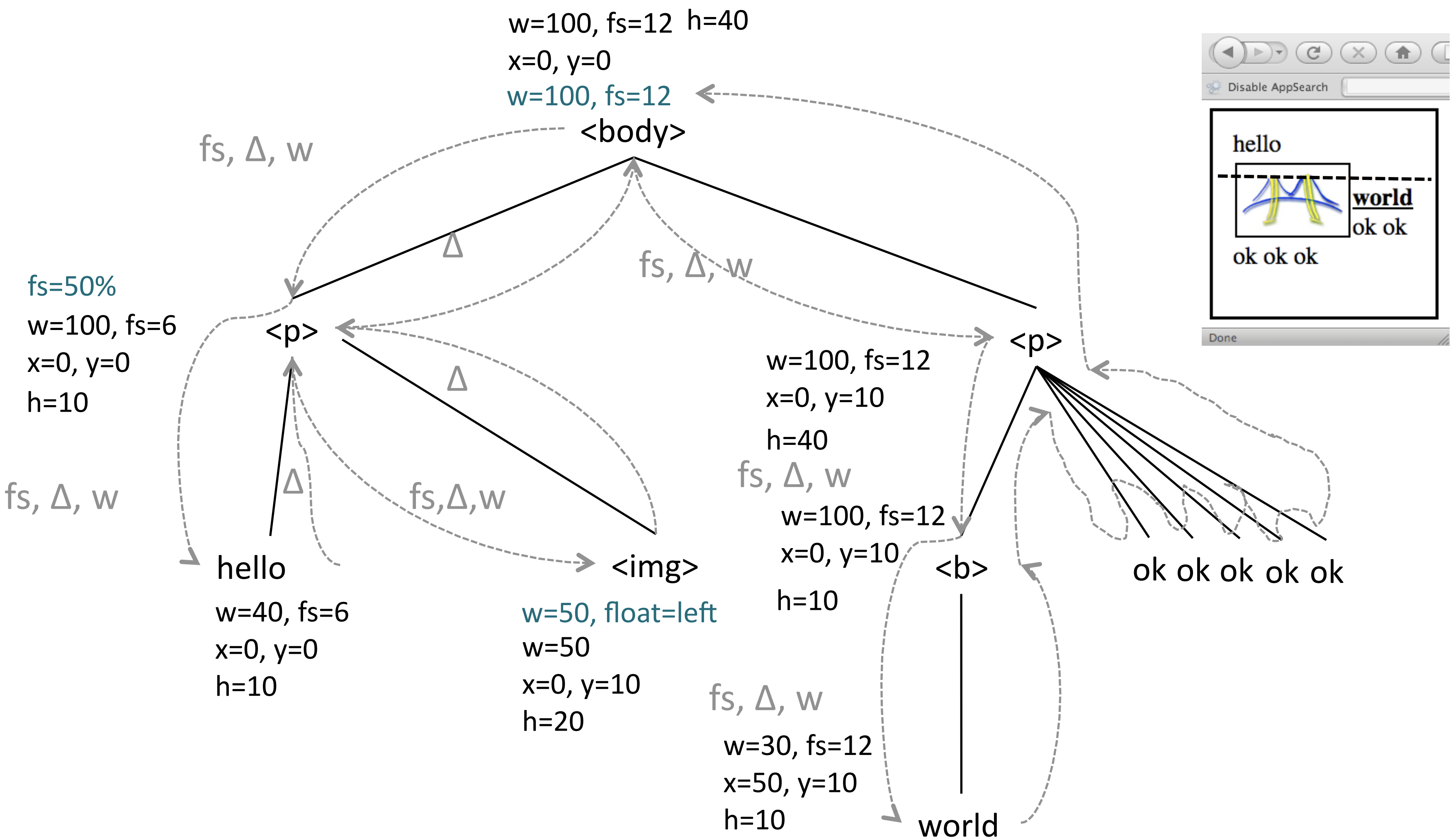


2 socket x 4 core x 2 thread (2.6 Ghz, 12x 1 GB)

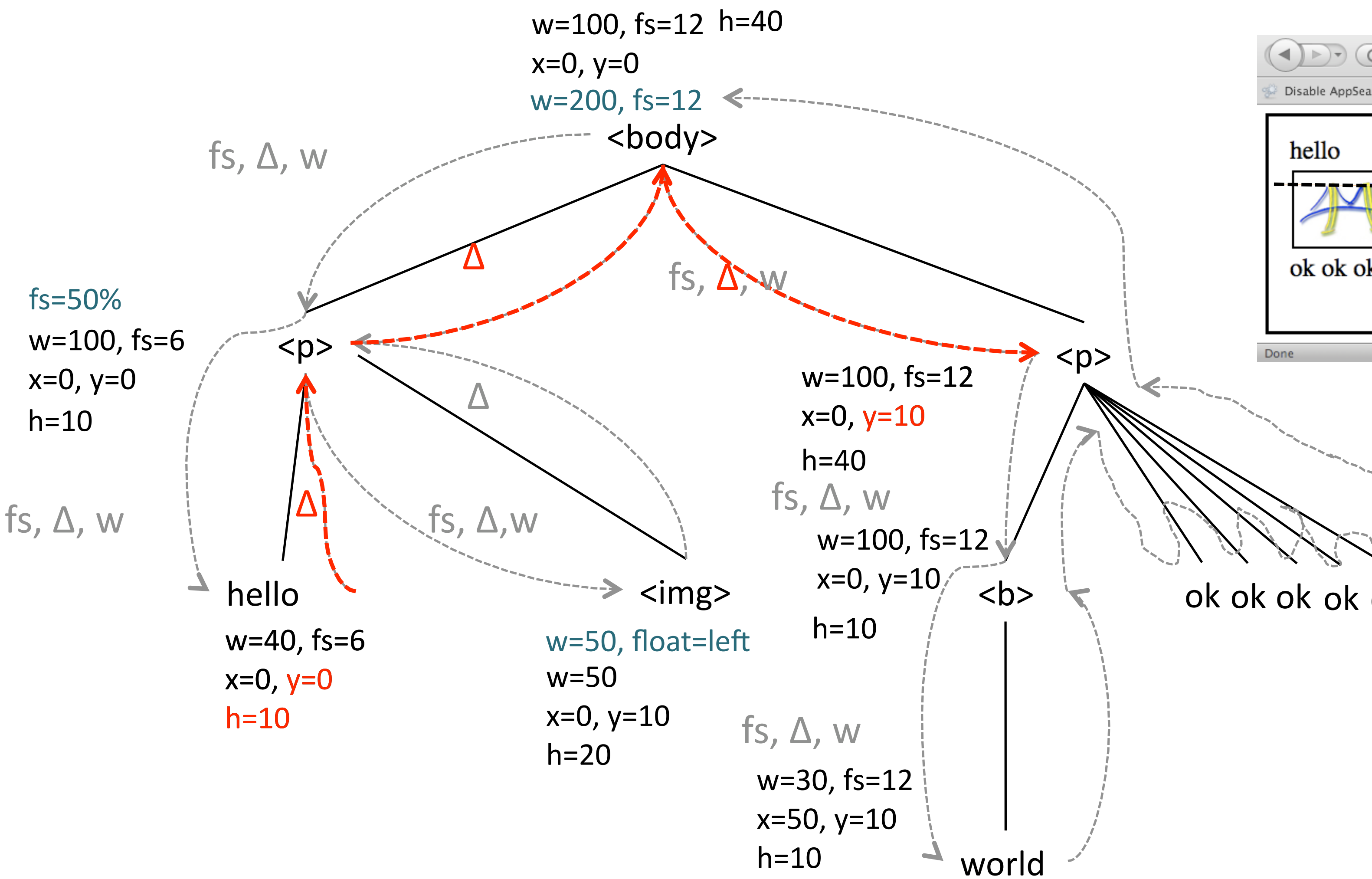
Layout Solving (2/2)

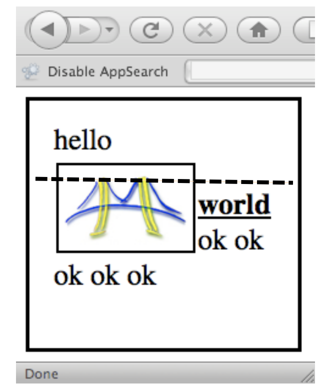


Problem: Layout a Page

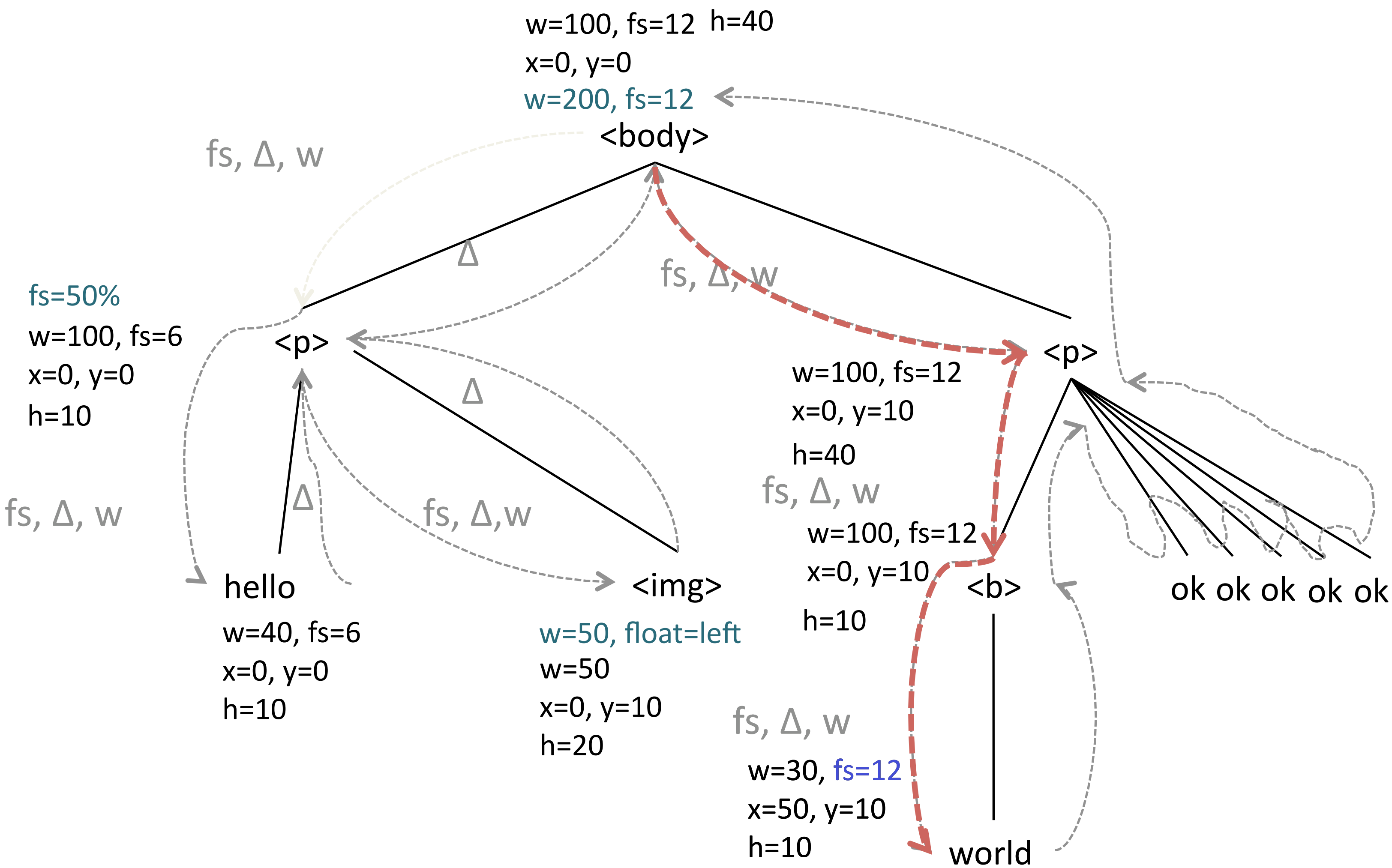


It looks rather sequential..

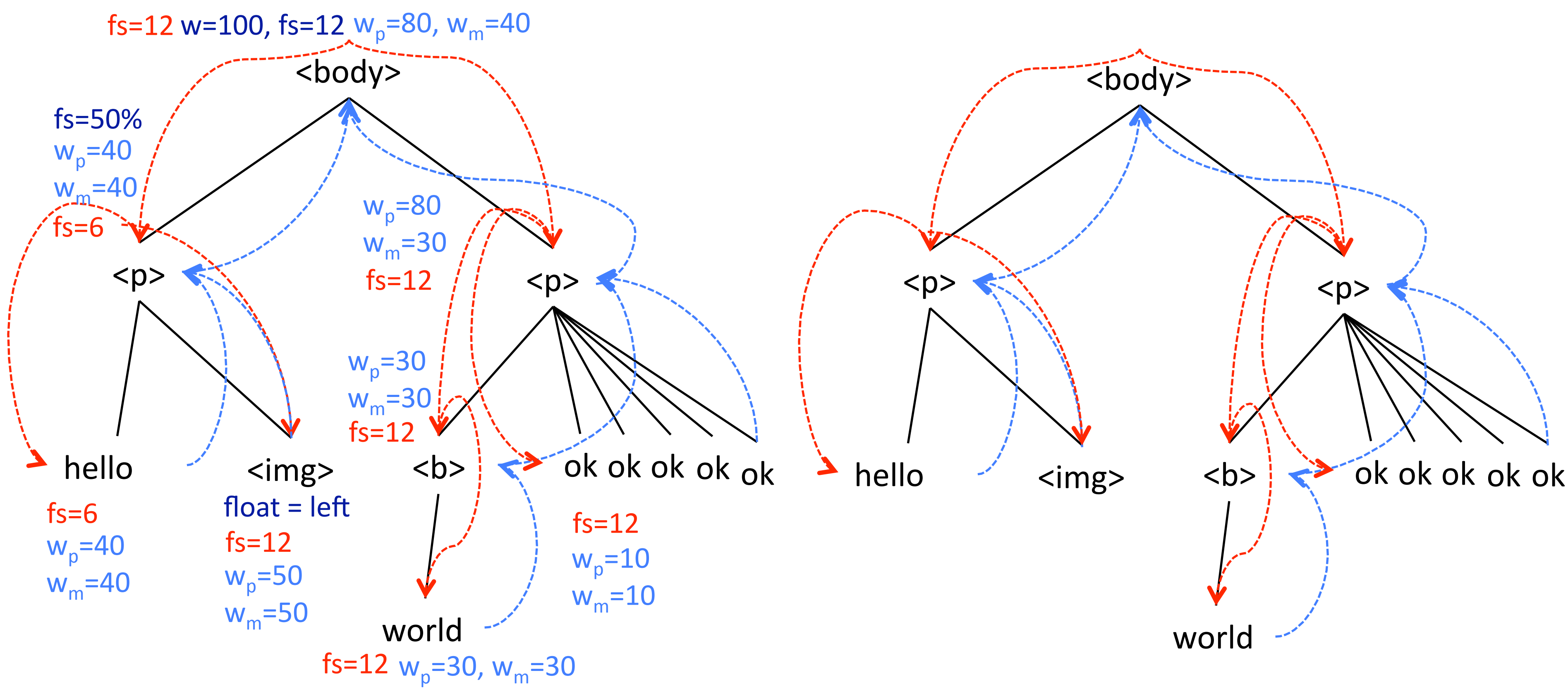




But not entirely



5 Phases: Each Exhibits Tree Parallelism



Phase 1: font size, temporary width

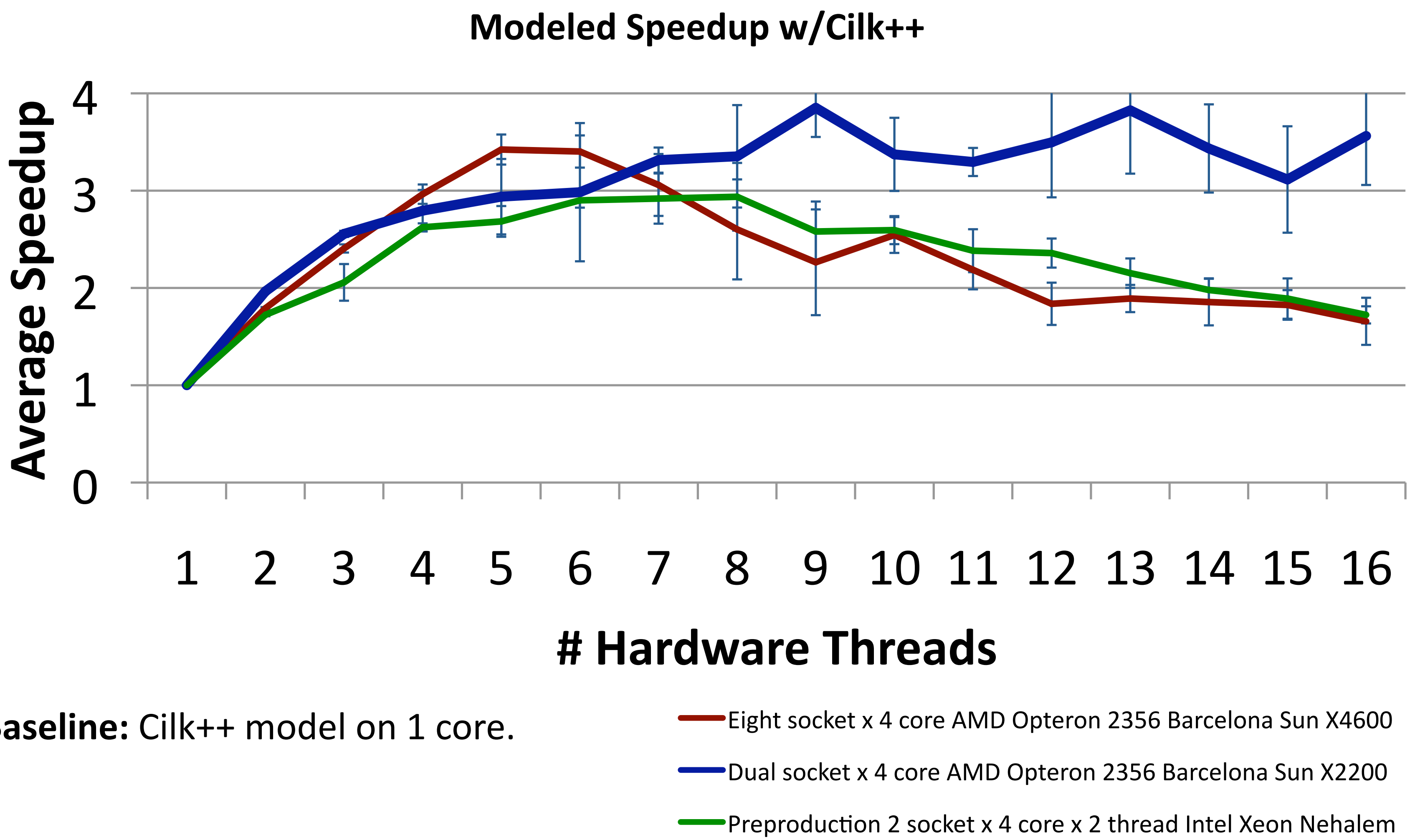
Phase 2: preferred max & min width

Phase 3: solved width

Phase 4: height, relative x/y position

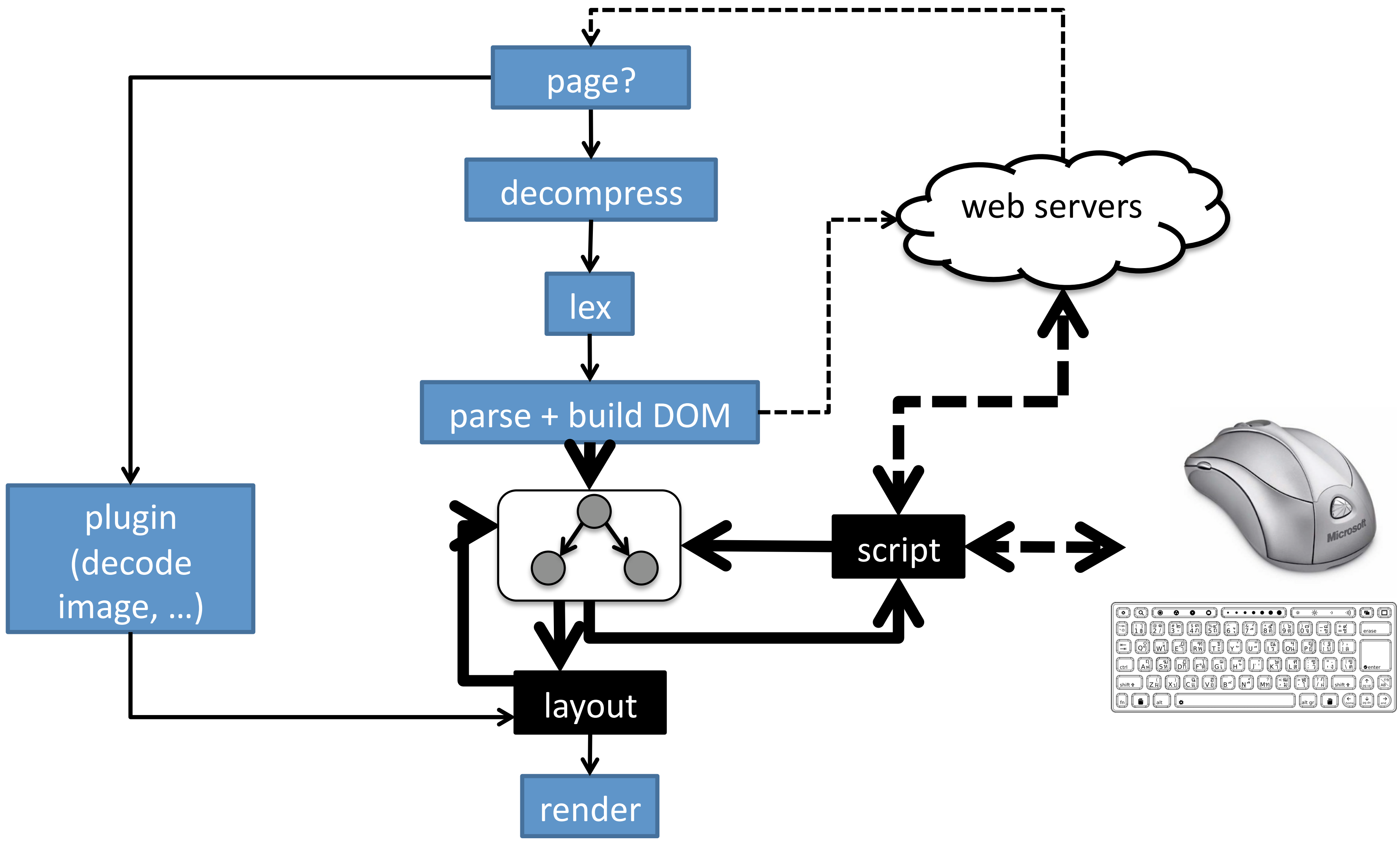
Phase 5: absolute x/y position

Results: layout (*modeled)*



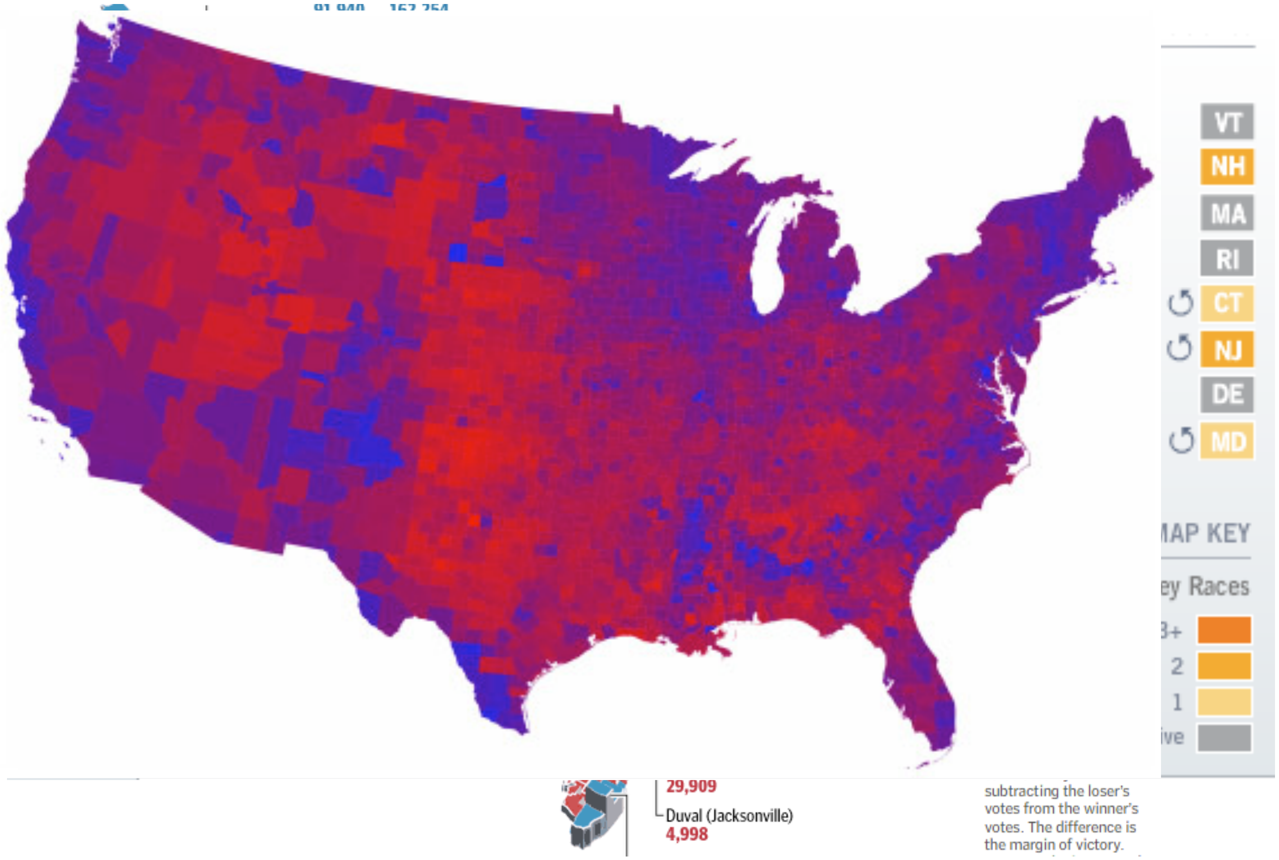
Baseline: Cilk++ model on 1 core.

Scripting



page?
decompress
lex
parse + build DOM
web servers
plugin (decode image, …)
script
layout
render

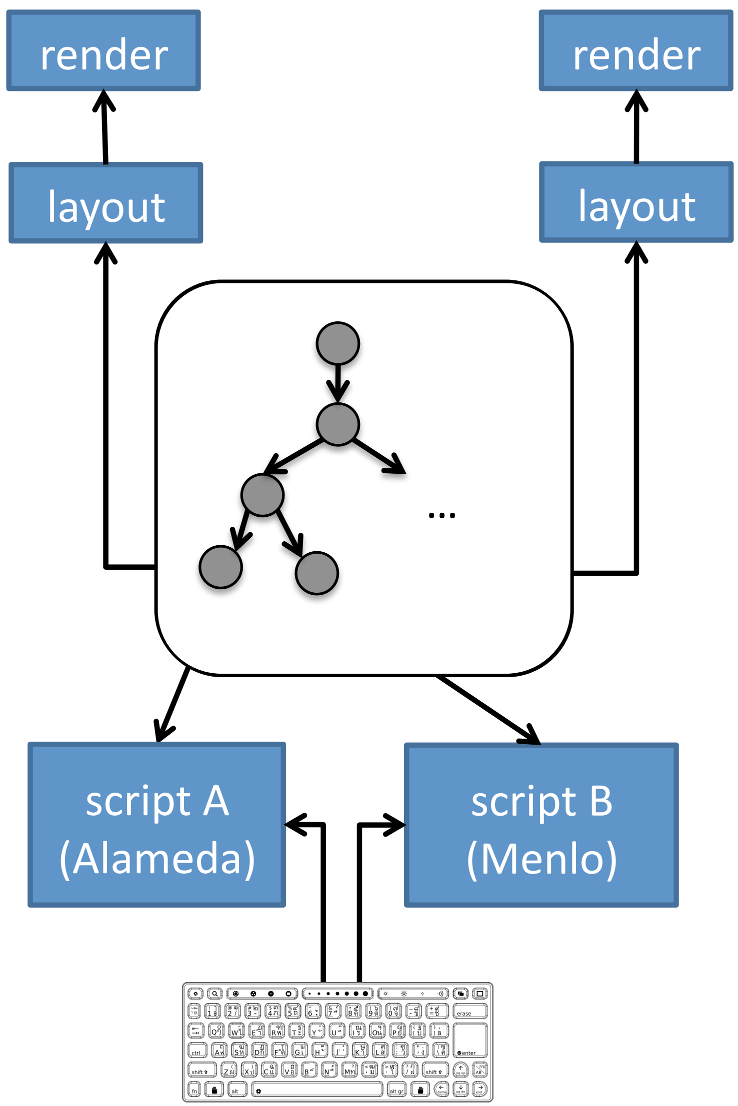
Why parallelize scripting (example)



Example: animate between different views

- each transition: recolor, resize each state or county
- animation rate 30fps => 33ms for 1000s of nodes

The browser programming model



- Nonpreemptive event model
- Handlers respond to events
- Handlers execute atomically
  - document changes cause relayout
  - style changes cause relayout

- To parallelize, must understand how the document is shared
  - document-carried dependencies:
    handler A: california.x =100;
    handler B: var z = california.x;
  - layout-carried dependencies:
    handler A: america.w = 200%;
    layout: california.w = 200%;
    handler B: var z = california.w;

Concurrency bugs

1. GUI animations and interactions
   – several animations modifying an object simultaneously
2. Server interactions
   – responses to requests may be delayed, reordered
3. Eager script loading
   – executing a script on a document before done loading

“Gotos” in JavaScript

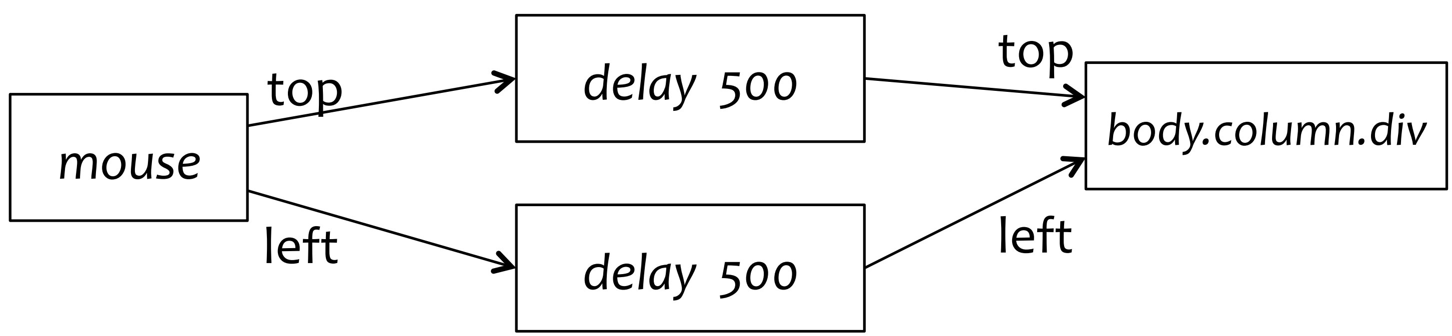
```
<div id="box" style="position:absolute; background: yellow;”>
  My box
</div>

<script>
document.addEventListener (
    'mousemove',
    function (e) {
        var left = e.pageX;
        var top = e.pageY;
        setTimeout(function() {
            document.getElementById("box").style.top = top;
            document.getElementById("box").style.left = left;
        } , 500);
    }, false);
 </script>
```

Preliminary design of our language

Program structure is clearer when data and control is explicit

- in dataflow version: **changing mouse coordinates are streams**
- coordinate streams adjust box position after they are delayed
- **structured names** of document element allow analysis



Summary

1. **Developed *work-efficient* algorithms**
   – *Rule matching:* parallel-map with a tiling optimization
   – *Layout*: break up tree traversal into five parallel ones
   – *Lexing:* speculation to break sequential dependencies
2. **Reexamining the scripting programming model**
   – *programmer productivity*: from callbacks to actors
      • influenced by Flapjax, Ptolemy, Max/MSP, LabVIEW
   – *performance*: adding structure to detect dependences
      • current browsers: JIT compilation, font vectorization, task parallelism eg for image rendering – all these are useful, too.