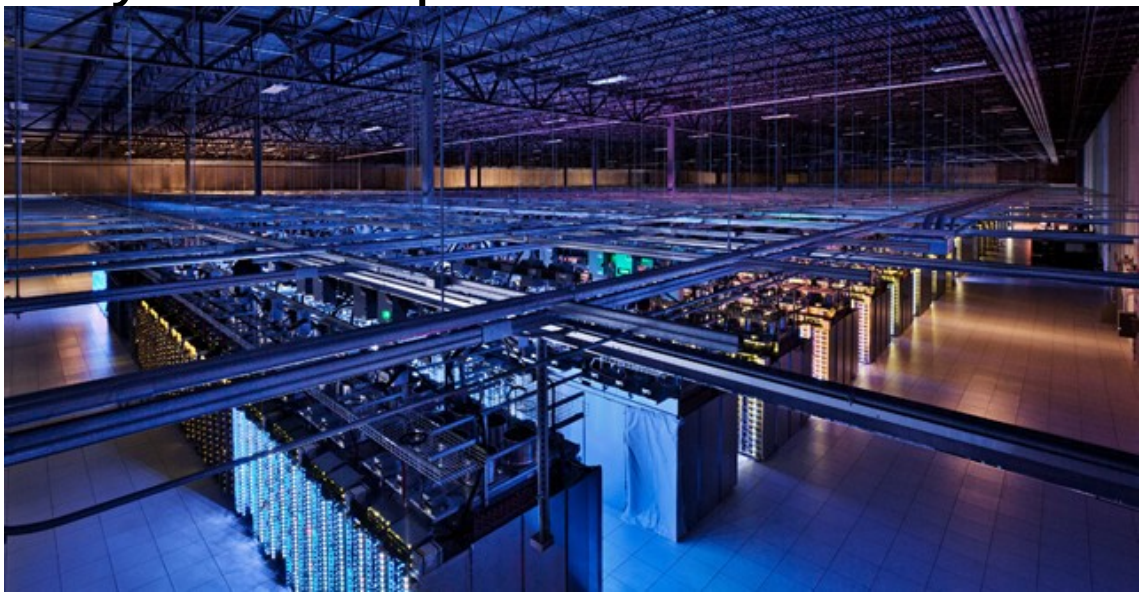
Functional Programming and Performance

Nicholas Chapman
Managing Director,
Glare Technologies Ltd
nick@indigorenderer.com

Is software performance important?

The importance of software performance

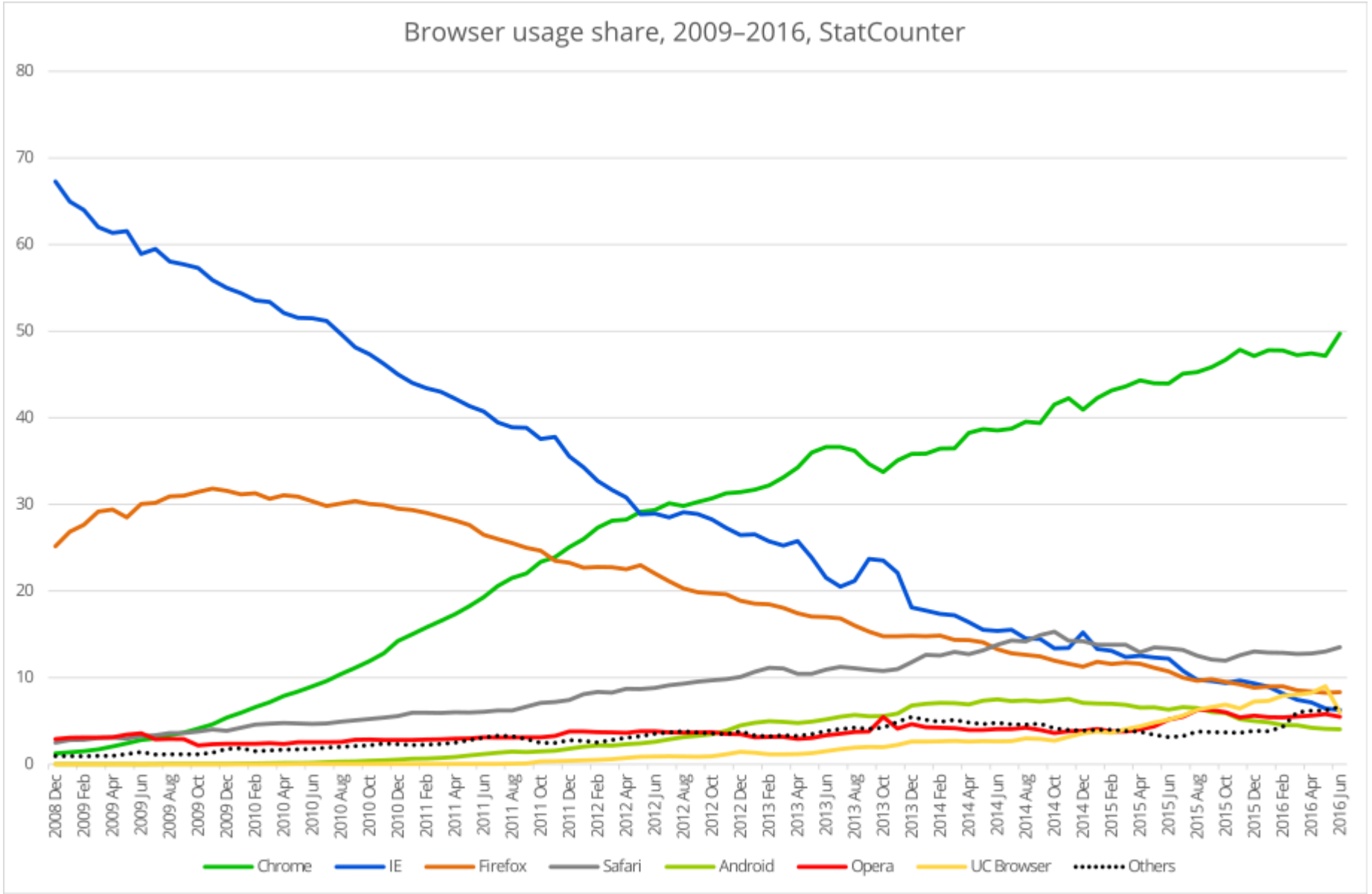
- Power consumption in data centres:
- “US data centers consumed about 70 billion kilowatt-hours of electricity in 2014, the most recent year examined, representing 2 percent of the country’s total energy consumption, according to the study. That’s equivalent to the amount consumed by about 6.4 million average American homes that year.” - http://www.datacenterknowledge.com/



The importance of software performance (2)

- No one likes to wait for their software.
- Low performance software = lots of waiting.
- Better performing software is more popular.
- See e.g. Google's Chrome and their focus on performance. (Nice chapter 'High Performance Networking in Chrome' in 'The performance of Open Source Applications') http://www.aosabook.org/en/posa/high-performance-networking-in-chrome.html

Chrome market share domination – in large part due to performance?



Our area – Computer graphics

Rendered with Indigo Renderer – probably took several hours to render.

Our area – computer graphics (2)

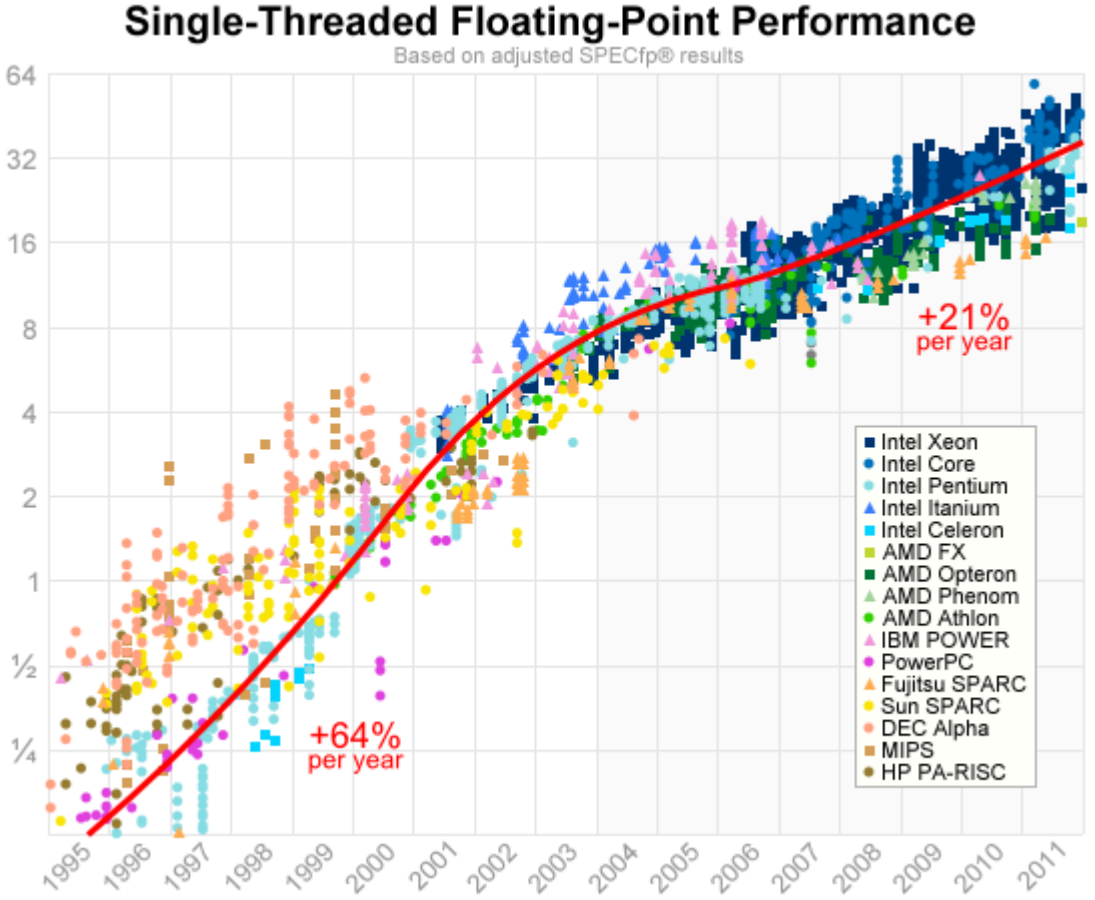
- Performance is crucial
- A single image may take hours to generate on a fast modern computer.
- Customers like to see an image fast, they also often have deadlines to produce renders by

The clock speed plateau

- Intel Pentium 4 – reached 3.06 GHz in 2002.
- My current computer in 2016: 3rd gen Intel Core i7 (Ivy Bridge) – hits 3.7 GHz.

- The free lunch of clock speed increases is definitely over
- ‘The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software’ - http://www.gotw.ca/publications/concurrency-ddj.htm

Overall performance is still increasing though, just more slowly



- From http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/

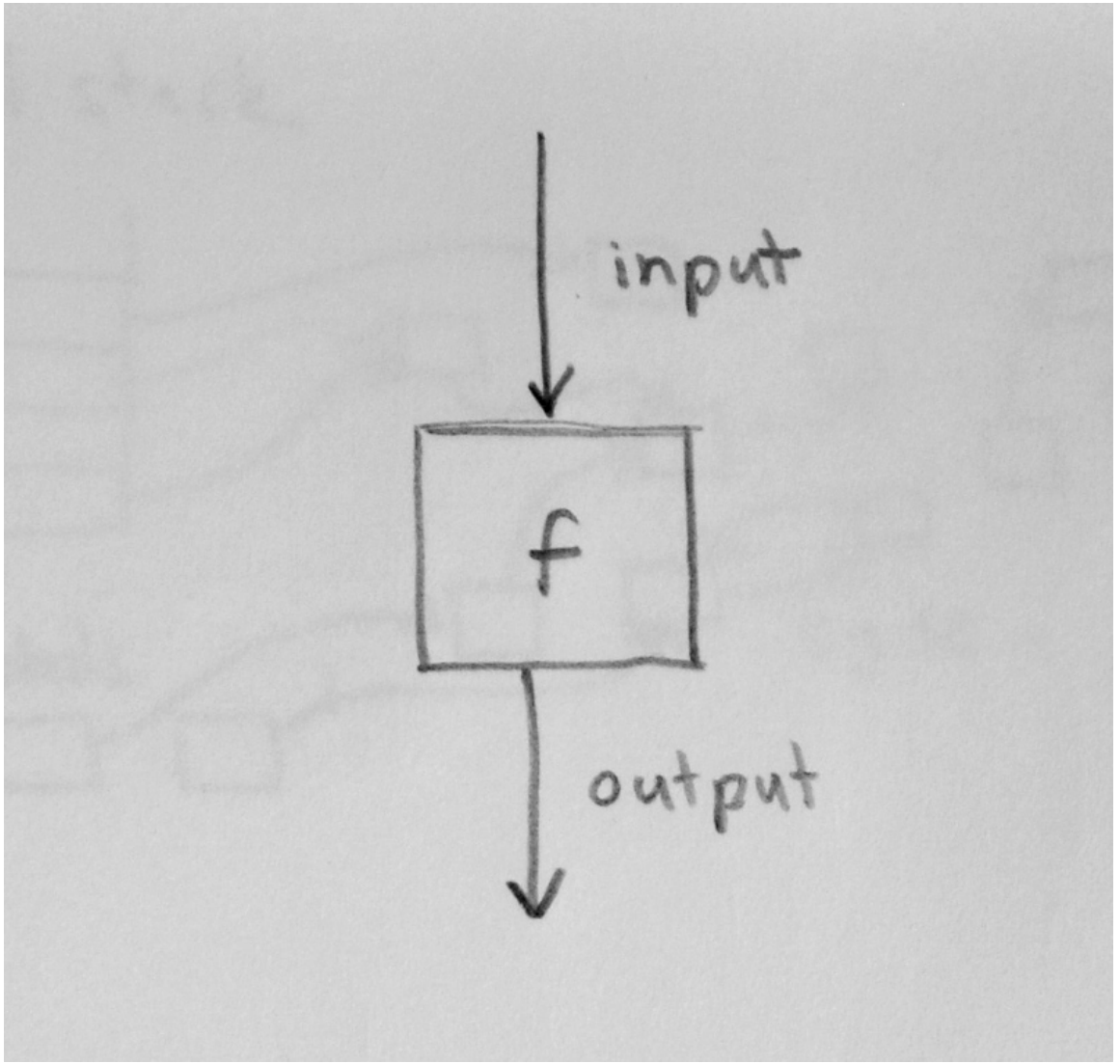
Improving performance approach 1: Parallelism

- If we can split our work up and do multiple tasks at once, we can get the work done quicker.
- We need language/platform support for this though.

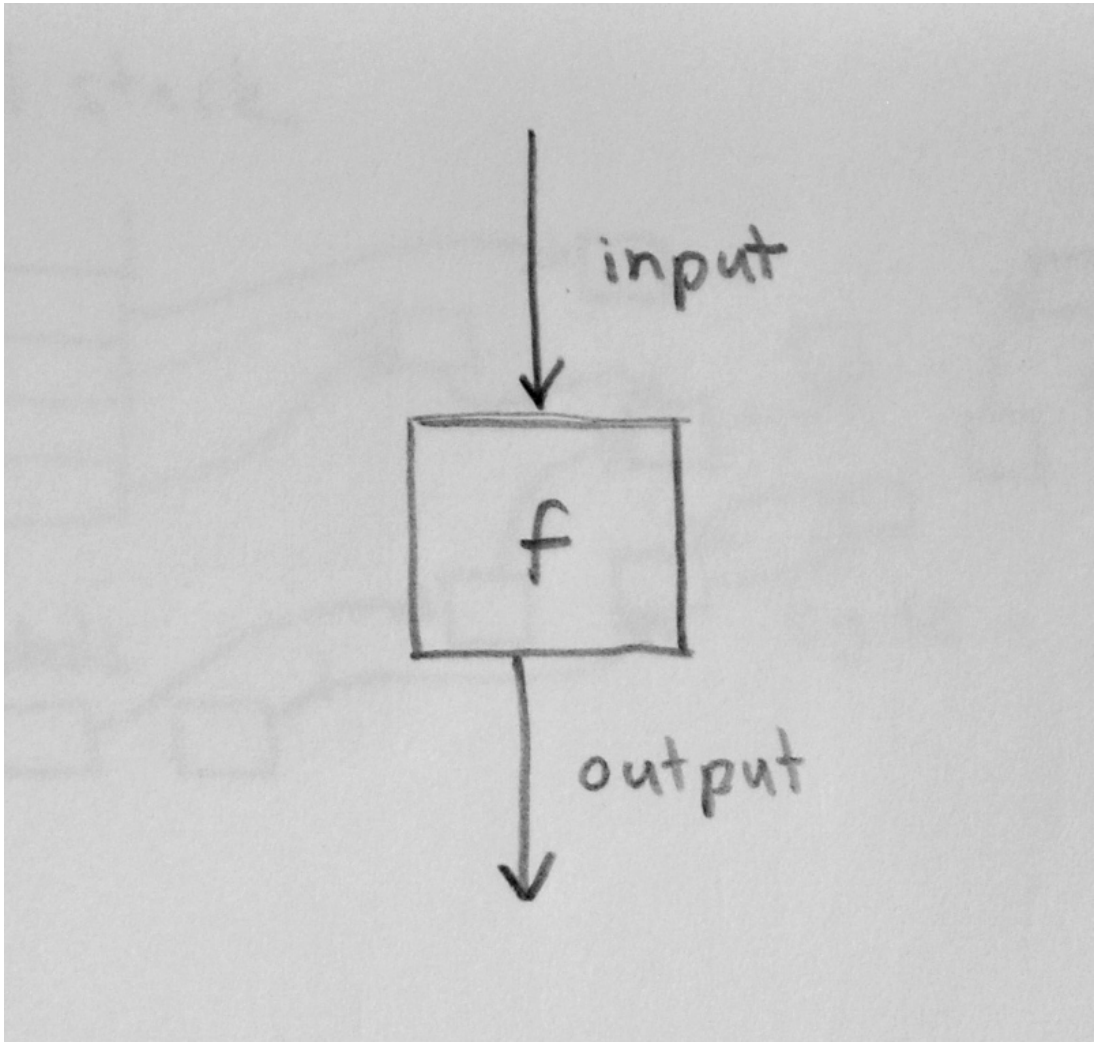
Parallelism and functional languages

- Lets compare to the handling of parallelism in imperative languages

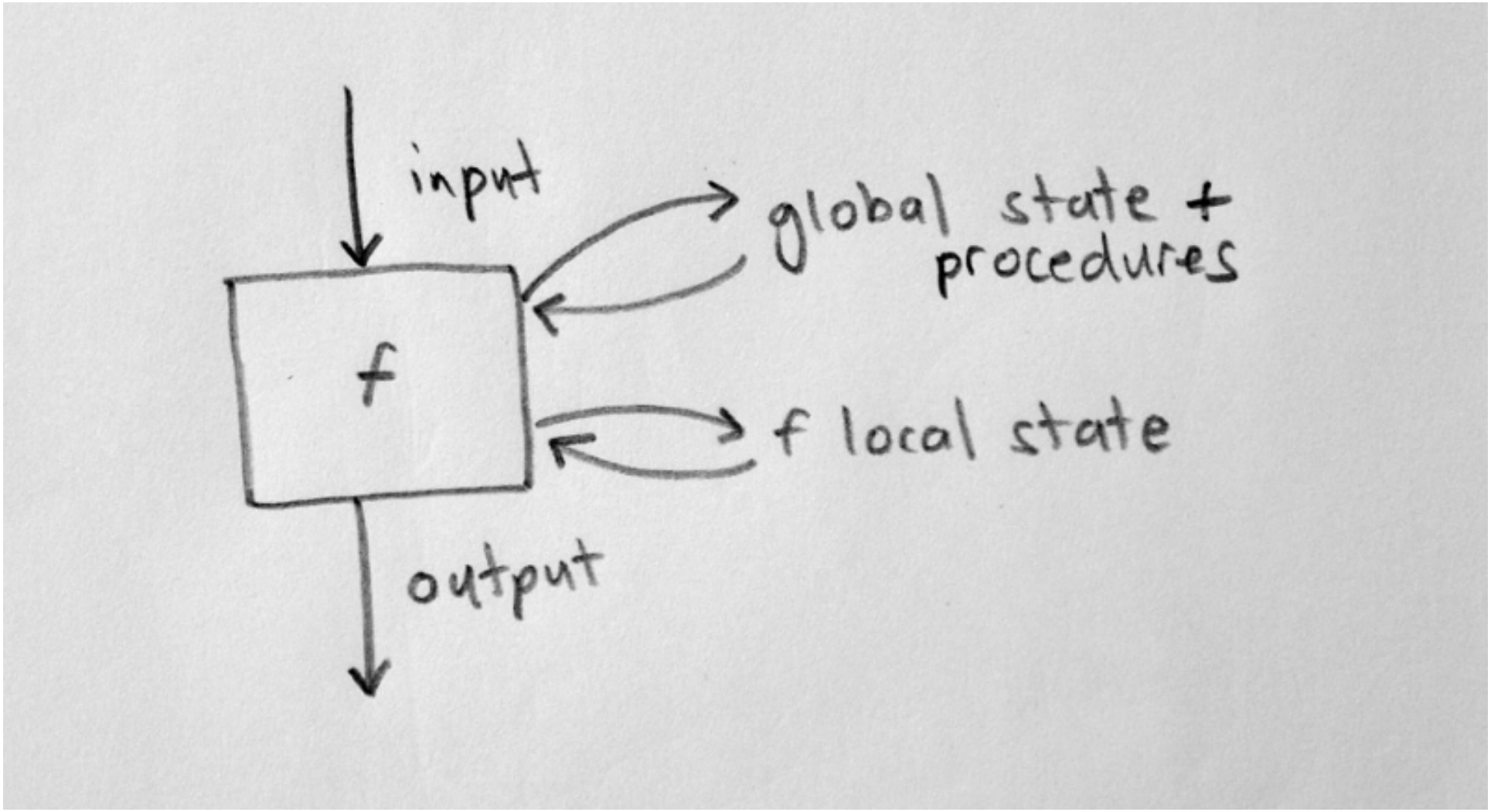
A function in maths



A function in a (pure) functional programming language



A function/procedure/method in an imperative language



Can we automatically parallelise this *for* loop?

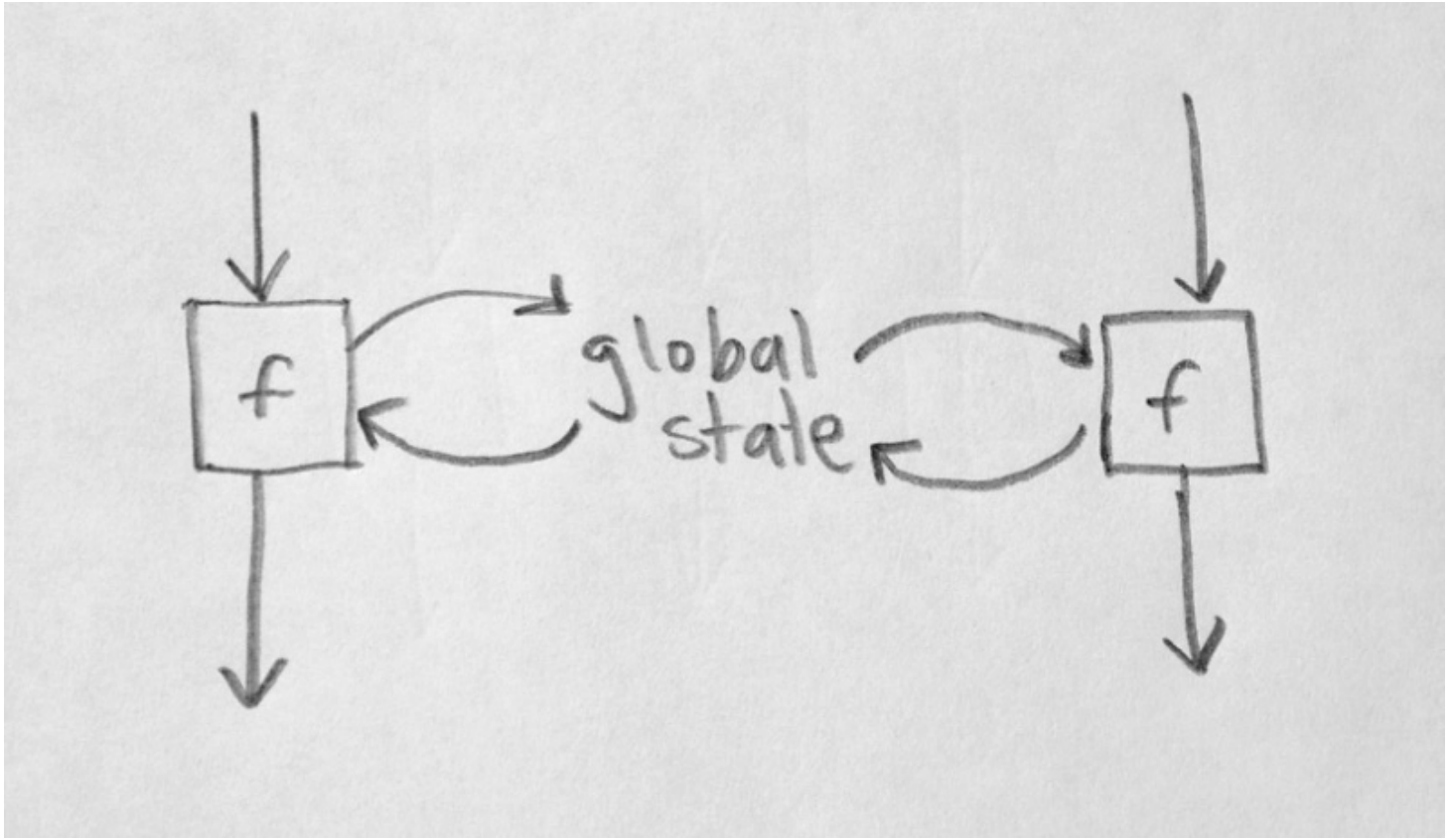
-

```
void doWork(const float* in,
float* out, size_t N)
{
  for(int i=0; i<N; ++i)
    out[i] = f(in[i]);
}
```

Can we automatically parallelise this *for* loop? (2)

- We can only do so if iterations of the loop are independent.

Information flow for the imperative for loop

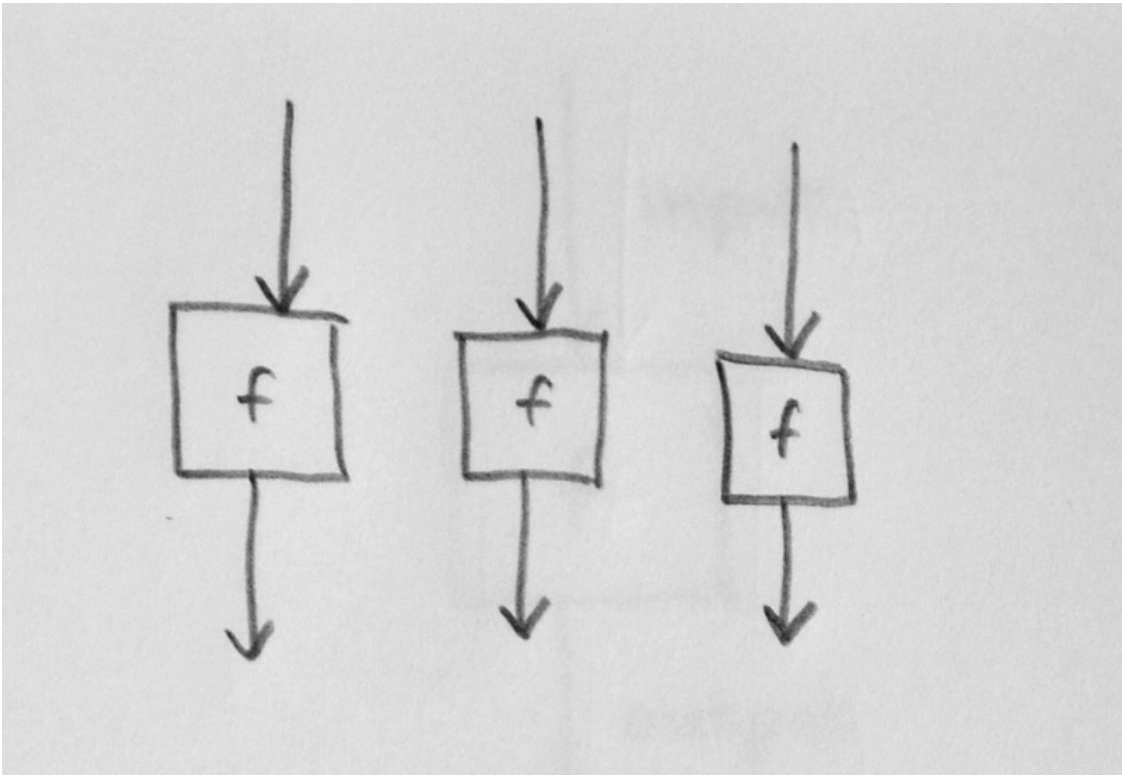


Applications of *f* can’t be separated due to entangling, due to possible access of global state!

Information flow for the equivalent functional program

```
def main( array<float, N> a) : map(f, a)
```

Apples the function f to each element in the array ‘a’.



Manifest Parallelism

- Parallelism is manifest – obvious and directly apparent – due to the language and notation used.

Example of Winter auto-parallelisation

```
def f(float x) float : pow(x, 2.2)
def main(array<float, 268435456> a, array<float, 268435456> b) array<float, 268435456> :
    map(f, a)
```

- With auto-parallelisation disabled:
- Winter throughput: 0.169819 GiB/s
- C++ throughput: 0.563838 GiB/s
- (C++ is faster due to Visual C++ vectorising the pow call, and LLVM doesn't)
- With auto-parallelisation enabled:
- Winter throughput: 0.793327 GiB/s
- C++ throughput: 0.552969 GiB/s
- Winter is **4.67x** faster with auto-parallelisation enabled compared to without.

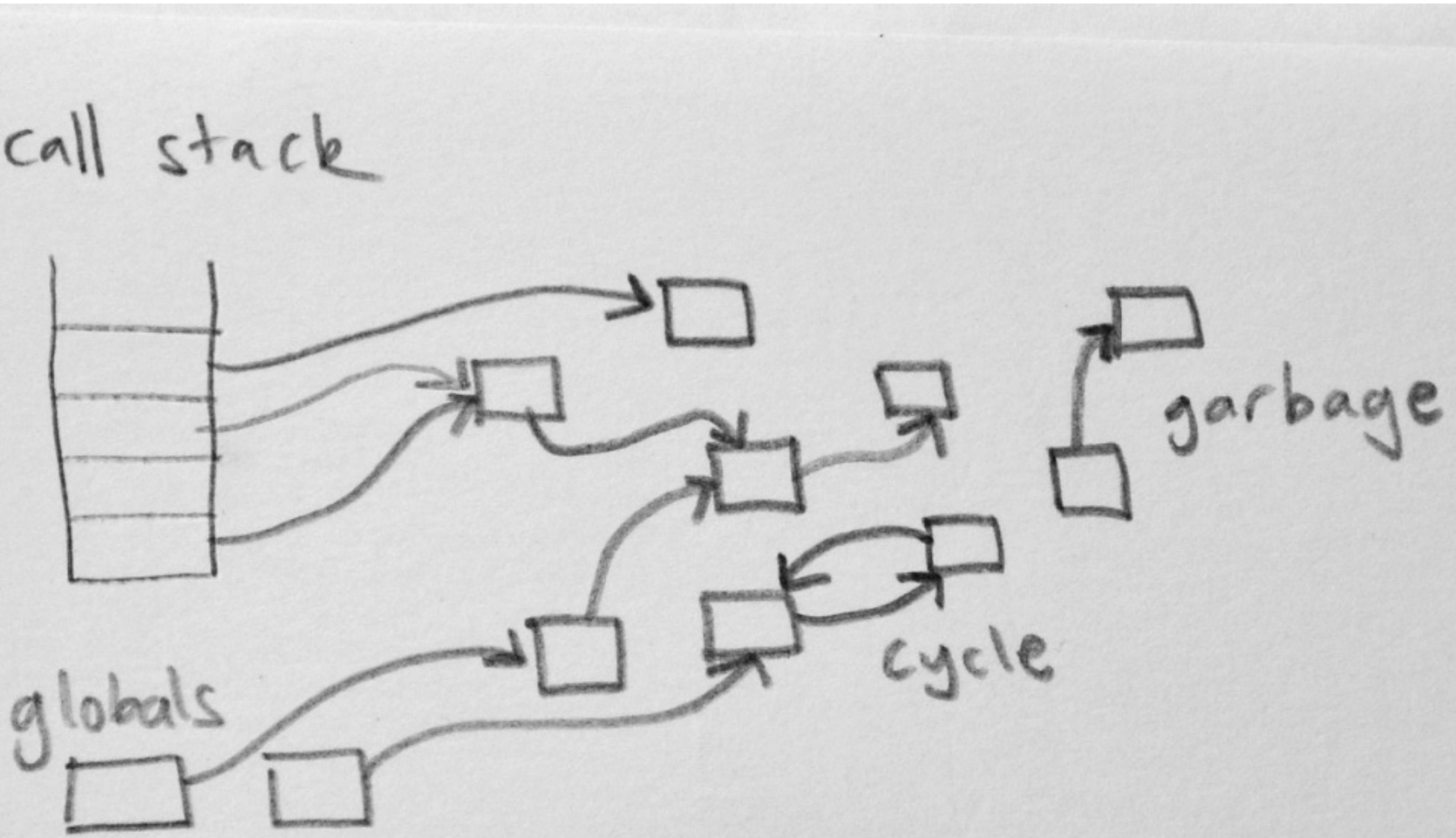
Part 1 Summary

- Functional programming languages exhibit manifest parallelism
- Allows automatic parallelisation.
- Imperative functions get ‘entangled’ and can’t be easily parallelised.

Part 2: Functional programming languages and Garbage Collection

Quick review of Garbage Collection(GC)

- Two main types of GC:
- Tracing (e.g. mark and sweep)
- Reference counting


call stack
garbage
globals
cycle

Reference counting

- Is efficient
- No need to traverse over entire heap
- No need to ‘stop the world’
- But can’t collect cycles

How a cycle gets made in an imperative language

```
struct Node
{
    Node* child;
};


Node* a = new Node();

Node* b = new Node();
b->child = a;

a->child = b;
```

Note that node *a* has to be modified after its creation.

Functional programming to the rescue!

- By default, in a language with immutable values, cycles cannot be formed.
- Have to explicitly add support for recursive data, e.g. *letrec* in lisp.

Functional programming to the rescue! (2)

- This means we can use a high performance GC technique – reference counting – in our functional language.

Even crazier forms of GC

- Compute a bound on the amount of memory used by a function or program
- Allocate just that amount initially as an ‘arena’
- Free in one chunk when done.
- Use with reference counting

Even crazier forms of GC (2)

- Compute a bound on the sum of individual allocations
- Allocate in one chunk
- Bump pointer in chunk for each alloc.
- No ref counting needed.
- Probably fastest possible memory management (as long as total size bound is not too large)

Not all sunshine and lollipops

- The major performance issue for F.P. on von Neumann architecture – slow to update single element in large collection.

Example 1:

- Count frequency of elements (e.g. frequency of byte values) in a large array.
- Described in ‘Let's Take a Trivial Problem and Make it Hard’: http://prog21.dadgum.com/41.html
- Actually a significant real world problem (counting sort, radix sort)

Example 2: quicksort

- Quicksort (partition sort) in imperative languages is extremely fast, in large part due to partitioning in place
- Not so much in functional languages.

```
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
    where
        lesser  = filter (< p) xs
        greater = filter (>= p) xs
```

Example Haskell quicksort, from https://wiki.haskell.org/Introduction

Example 2: quicksort (cont.)

- “[...] quicksort where Haskell’s elegant two-line sort is over 1,000x slower than Sedgewick’s Quicksort in C because the Haskell deep copies lists over and over again, completely blowing the asymptotic IO complexity of Hoare original algorithm.” - http://flyingfrogblog.blogspot.co.uk/2016/05/disadvantages-of-purely-functional.html

NOTE: Quite possibly an incorrect statement, but thought provoking at least!

Example 3: Conway’s game of life

- Potentially large world, with local updates.
- Lazy approaches for FP exist, but what is the perf compared to imperative updates?

Future research

- Solving these perf weaknesses of FP is doable, but tricky
- Possible to find brittle optimisations
- Difficult to make robust
- Hybrid solutions?

  (FP by default, with some imperative sprinkled in?)

Thanks!

- Questions?
- My blog: http://www.forwardscattering.org/

Activity

- Possible activity: write a fast byte frequency counting function in your language of choice?
- See http://prog21.dadgum.com/41.html

```
freq(B) when is_binary(B) ->
   freq(B, erlang:make_tuple(256, 0)).

freq(<<X, Rest/binary>>, Totals) ->
   I = X + 1,
   N = element(I, Totals),
   freq(Rest, setelement(I, Totals, N + 1));
freq(<<>>, Totals) ->
   Totals.
```