“Write Once, Parallelize Anywhere”

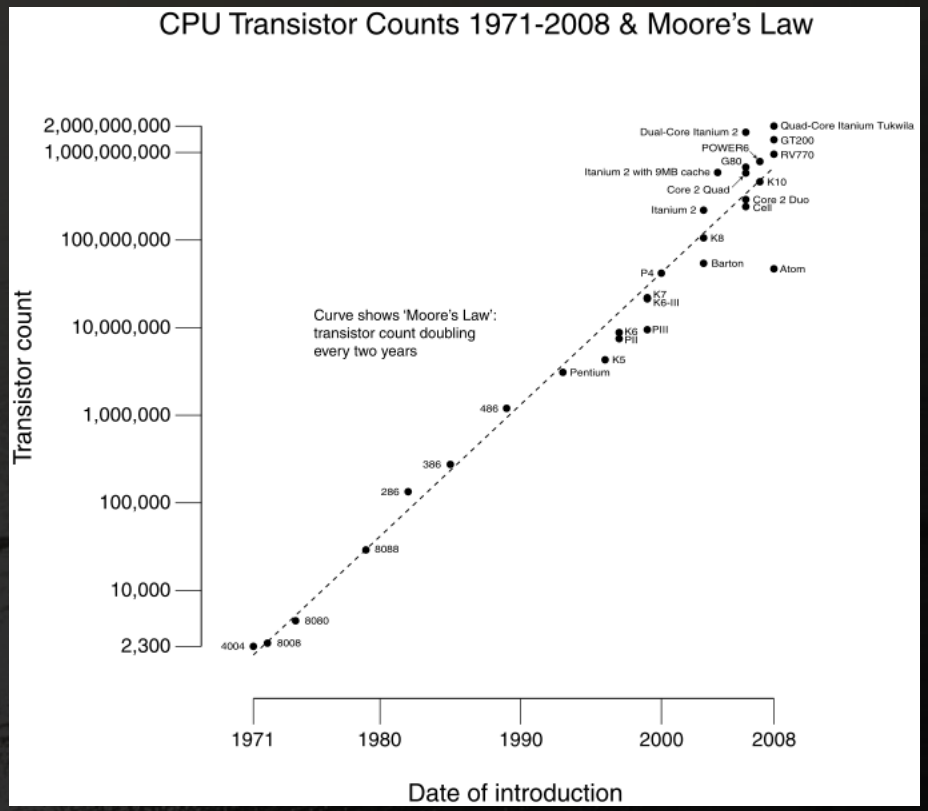
The Flow Programming Language:

An implicitly-parallelizing, programmer-safe language

Luke Hutchison

Moore's Law

- The number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every two years.



Moore's Law

- Computing is work; transistors do work; more transistors working in parallel should yield faster computers
  - => **Moore's Law has (so far) also applied to *CPU speed***

- BUT now we're hitting multiple walls:
  - **Transistor size**
    - tunneling; lithography feature size vs. wavelength
    - no more 2D density increases
  - **Thermal envelope**
    - function of frequency and feature size => we have hit a clockspeed wall
  - **Data width**
    - 128-bit CPUs? Need parallel control flow instead

But it's worse than that

It's not just the end of Moore's Law...

...it's the beginning of an era of **buggier software.**

Humans will **never** be good at
writing multi-threaded code –
our brains simply don't work like that.

(Abstractions like Futures only help a little bit.)

Parallel Programming Toolset

- Traditional locks: mutex, semaphore etc.
- Libraries: java.lang.concurrent
- Message passing: MPI; Actor model (Erlang)
- Futures, channels, STM
- **MapReduce**
- **Array programming**
  - Java7 ParallelArray
  - Intel Concurrent Collections (CnC)
  - Intel Array BuildingBlocks (ArBB)
  - Data Parallel Haskell (DPH)
  - ZPL

But—all require **programmer skill**, **extra effort** and **shoehorning of design.**

How to shoot yourself in the foot

Classic 1991 – http://bit.ly/hiwaG1

FORTRAN : You shoot yourself in each toe, iteratively, until you run out of toes, then you read in the next foot and repeat.

COBOL : USEing a COLT 45 HANDGUN, AIM gun at LEG.FOOT, THEN place ARM.HAND.FINGER on HANDGUN.TRIGGER and SQUEEZE. THEN return HANDGUN to HOLSTER. CHECK whether shoelace needs to be retied.

Lisp : You shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds...

BASIC : You shoot yourself in the foot with a water pistol.

Forth : Foot yourself in the shoot.

C++ : You accidently create a dozen instances of yourself and shoot them all in the foot. Providing emergency medical assistance is impossible since you can't tell which are bitwise copies and which are just pointing at others and saying "That's me, over there."

C : You shoot yourself in the foot.

My addition: **Explicit parallelization in any language** : You shoot in the yourself <*segfault*>

The state of Moore's Law, 2010

"*Finding:* There is no known alternative for sustaining growth in computing performance; however, no compelling programming paradigms for general parallel systems have yet emerged."

—The Future of Computing Performance: Game Over or Next Level?

Fuller & Millett (Eds.); Committee on Sustaining Growth in Computing Performance, National Research Council, The National Academies Press, **2010**, p.81

=> The Multicore Dilemma

The root of the problem

- *Purely* functional programming languages can be safely implicitly parallelized
  - No side-effects, no state
  - => threads cannot interact
  - e.g. several parallel versions of Haskell

The root of the problem

but

purely functional programming languages
are extremely hard for mere mortals
to use to solve real-world problems.

On the other hand,

getting explicit parallelization *right* –
and writing multithreaded code quickly –
may actually be *harder.*

Mere mortals

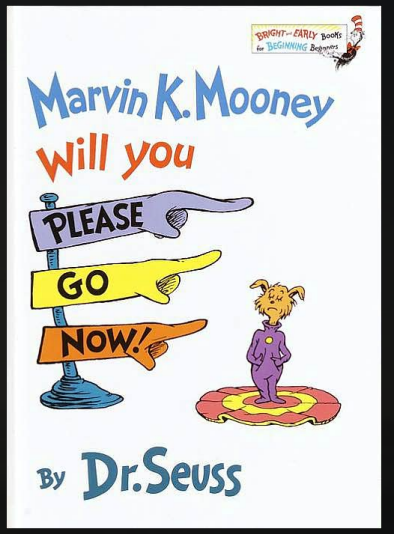
"*Recommendation:* Invest in research and development of programming methods that will enable efficient use of parallel systems not only by parallel systems experts *but also by typical programmers.*"

—The Future of Computing Performance: Game Over or Next Level?

Fuller & Millett (Eds.); Committee on Sustaining Growth in Computing Performance, National Research Council, The National Academies Press, **2010**, p.99

The root of the problem

- We like **imperative programming languages**: "do this, then this".



- But for imperative programming languages, it is impossible to tell the exact **data dependency graph** of a program by static analysis (by looking at the source).
  - (The data dependency graph tells you the order you need to compute values.)
  - Therefore you don't know what can be computed in parallel without:
    - (1) trusting the programmer (explicit parallelism) – VERY BAD
    - (2) guessing (static code analysis) – possibly/probably very bad
    - (3) trying and failing/bailing if you were wrong (STM)

Functional vs. imperative

- **Functional:**
  - *gather, pull, reduce*
- **Imperative:** same as functional, but adds
  - *scatter, push, produce*

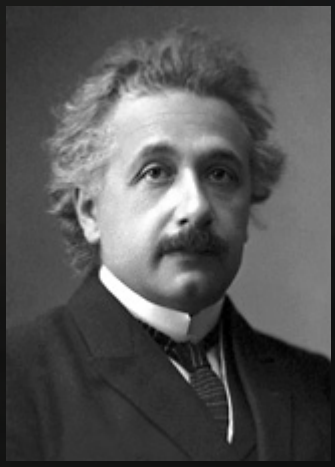
Training wheels

- How to *minimally* constrain imperative programming to allow for automatic implicit parallelization with guaranteed thread safety...

…*without* forcing the programmer to use training wheels?

The real root of the problem

- ...is **the ability to read the *current value of a variable*.** (Surprising. Foundational.)
  - => Einstein's theory of the relativity of simultaneity: there is no such thing as “now” given physical separation.
  - c.f. Values vs. variables.



- Remove the concept of “now”, and you *minimally restrict* a language such that it's impossible to create race conditions or deadlocks.
  - But you also don't have to program in purely functional style: Can support a *subset* of imperative, push-style programming.

Introducing Flow

- *Flow* is a new programming paradigm that enforces this restriction to solve the multicore dilemma while providing strong and specific guarantees about program correctness and runtime safety.

Introducing Flow

- Flow is a compiler framework that can be targeted by a wide range of different languages
  - [will probably plug into LLVM]
- ...and (eventually), a reference language

- It doesn't actually exist yet, come help make it exist
  - http://flowlang.net/

Goals of Flow

- **Solve the multicore dilemma**
  - Ubiquitous implicit parallelization, zero programmer effort
  - “Write once, parallelize anywhere”
    - Cluster : hadoop / MapReduce, MPI or similar
    - JVM via Java Threads
    - C via pthreads
    - CPU ↔ GPU via CUDA
      - Optimal load balancing via big-Oh analysis of computation and communication cost
- **Prevent programmers from shooting themselves in the foot**
  - Make race conditions, deadlocks, segfaults/NPEs, memory leaks **impossible**

Back to the drawing board

Flow enables implicit parallelization by

syntactically enforcing
every program to be
a lattice or partial ordering,

such that

the program source code itself is
the data dependency graph.

This makes parallelization trivial.

Back to the drawing board

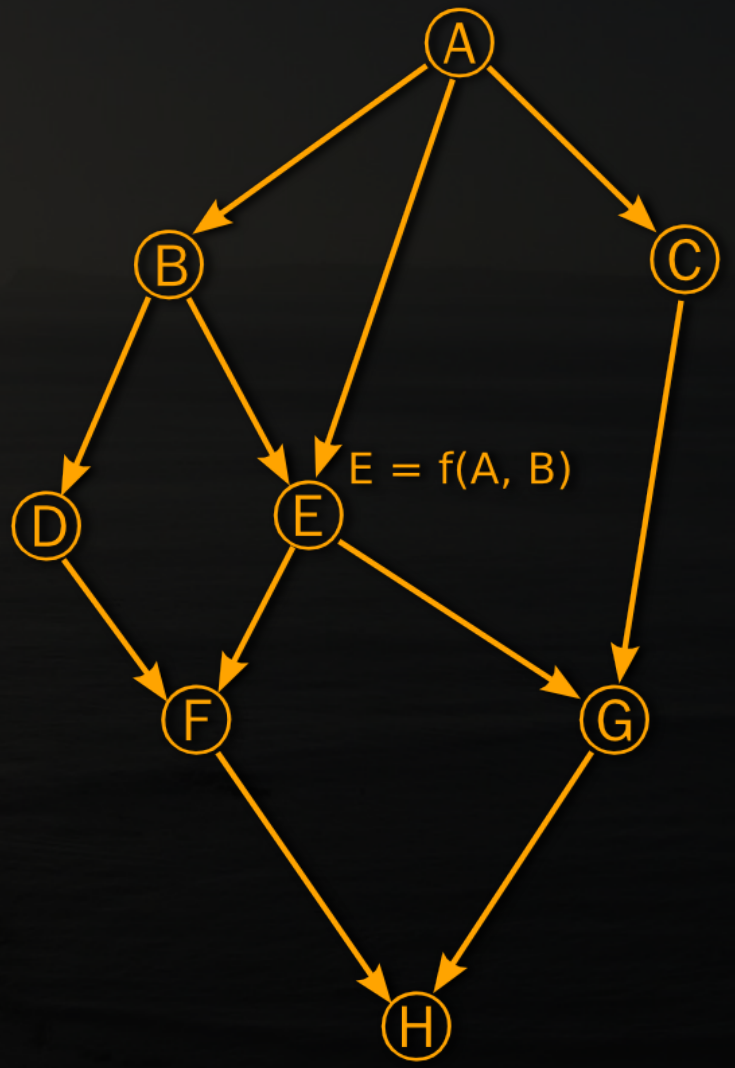
Also note that this does not eliminate “push” programming

- G can be an unordered collection

It also does not mean you can't “change the value of a variable”

- But you have to specify a specific timestamp, e.g.
  x@(t) = x@(t-1) + 1
  (then each value of x is a specific node in the DAG)

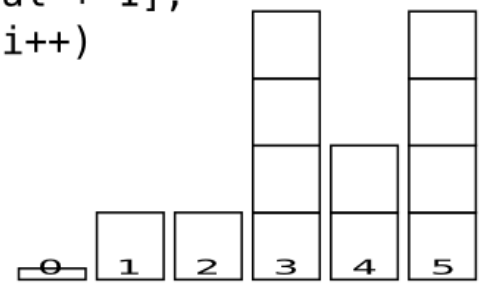
=> Minimally-restricted imperative programming with maximum implicit parallelization



An example

- Histogram generation

```
public int[] histogram(int[] input, int maxVal) {
    public int[] output = new int[maxVal + 1];
    for (int i = 0; i < input.length; i++)
        output[input[i]]++;
    return output;
}
```



{5,3,1,4,3,4,2,3,5,3,5,5} -> {0, 1, 1, 4, 2, 4}

- Multithreaded version? Two options:
  (1) locks (=> contention!)
  (2) keep separate copies in TLS; combine at end of operation
- Both options are manual
  => not easily composable, load-balanceable or scalable

An example

- More complex example

```
// Generate a histogram of avg number of spaces vs. string len

@SuppressWarnings("unchecked")  // Stupid Java generics
public int[][] avgNumSpacesGivenStrLen(String[] input, int maxLen) {

    // Alloc memory (even GC'd languages require this sort of stuff)
    ArrayList[] strsOfLen = new ArrayList[maxLen + 1];
    for (int i = 0, n = maxLen + 1; i < n; i++)
        stsOfLen[i] = new ArrayList();

    // Scatter inputs by string length
    for (int i = 0; i < input.length; i++)
        strsOfLen[input[i].length()].add(input[i]);

    // Calc avg number of spaces among all inputs with same string length
    public int[] output = new int[maxLen + 1];
    for (int len = 0; len <= maxLen; len++) {
        ArrayList<String> strs = (ArrayList<String>) strsOfLen[len];
        int totSpaces = 0;
        for (String str : strs)
            totSpaces += countSpaces(str);
        output[len] = strs.size() == 0 ? 0.0f : totSpaces / (float) strs.size();
    }
    return output;
}
```

- strsOfLen[*j*] can be an *unordered collection* (Set), does not affect result
- Allows writes to be interleaved or reordered => auto TLS split

An example

- More complex example

```
// Generate a histogram of avg number of spaces vs. string len

@SuppressWarnings("unchecked")  // Stupid Java generics
public int[][] avgNumSpacesGivenStrLen(String[] input, int maxLen) {

    // Alloc memory (even GC'd languages require this sort of stuff)
    ArrayList[] strsOfLen = new ArrayList[maxLen + 1];
    for (int i = 0, n = maxLen + 1; i < n; i++)
        stsOfLen[i] = new ArrayList();

    // Scatter inputs by string length
    for (int i = 0; i < input.length; i++)
        strsOfLen[input[i].length()].add(input[i]);

    // Calc avg number of spaces among all inputs with same string length
    public int[] output = new int[maxLen + 1];
    for (int len = 0; len <= maxLen; len++) {
        ArrayList<String> strs = (ArrayList<String>) strsOfLen[len];
        int totSpaces = 0;
        for (String str : strs)
            totSpaces += countSpaces(str);
        output[len] = strs.size() == 0 ? 0.0f : totSpaces / (float) strs.size();
    }
    return output;
}
```

- The less we constrain subproblems, the more parallelizable the code
- To relax constraints, must understand properties of functions / collections

An example

- More complex example

```
// Generate a histogram of avg number of spaces vs. string len

@SuppressWarnings("unchecked")  // Stupid Java generics
public int[][] avgNumSpacesGivenStrLen(String[] input, int maxLen) {

    // Alloc memory (even GC'd languages require this sort of stuff)
    ArrayList[] strsOfLen = new ArrayList[maxLen + 1];
    for (int i = 0, n = maxLen + 1; i < n; i++)
        stsOfLen[i] = new ArrayList();

    // Scatter inputs by string length
    for (int i = 0; i < input.length; i++)
        strsOfLen[input[i].length()].add(input[i]);

    // Calc avg number of spaces among all inputs with same string length
    public int[] output = new int[maxLen + 1];
    for (int len = 0; len <= maxLen; len++) {
        ArrayList<String> strs = (ArrayList<String>) strsOfLen[len];
        int totSpaces = 0;
        for (String str : strs)
            totSpaces += countSpaces(str);
        output[len] = strs.size() == 0 ? 0.0f : totSpaces / (float) strs.size();
    }
    return output;
}
```

Map

Reduce

- Turns out any function can be split into Map and Reduce stages
- But MapReduce doesn't incl flow control, + MR can't be auto-generated yet

Back to the drawing board

Flow syntactically enforces that a program's structure be a partial ordering or DAG (specifically a *lattice*).

Each node is the result of a function application of some sort

=> the process of computing each node can be thought of as a MapReduce operation

Back to the drawing board

Because Flow enforces program structure to be a partial ordering, some amazing properties emerge:

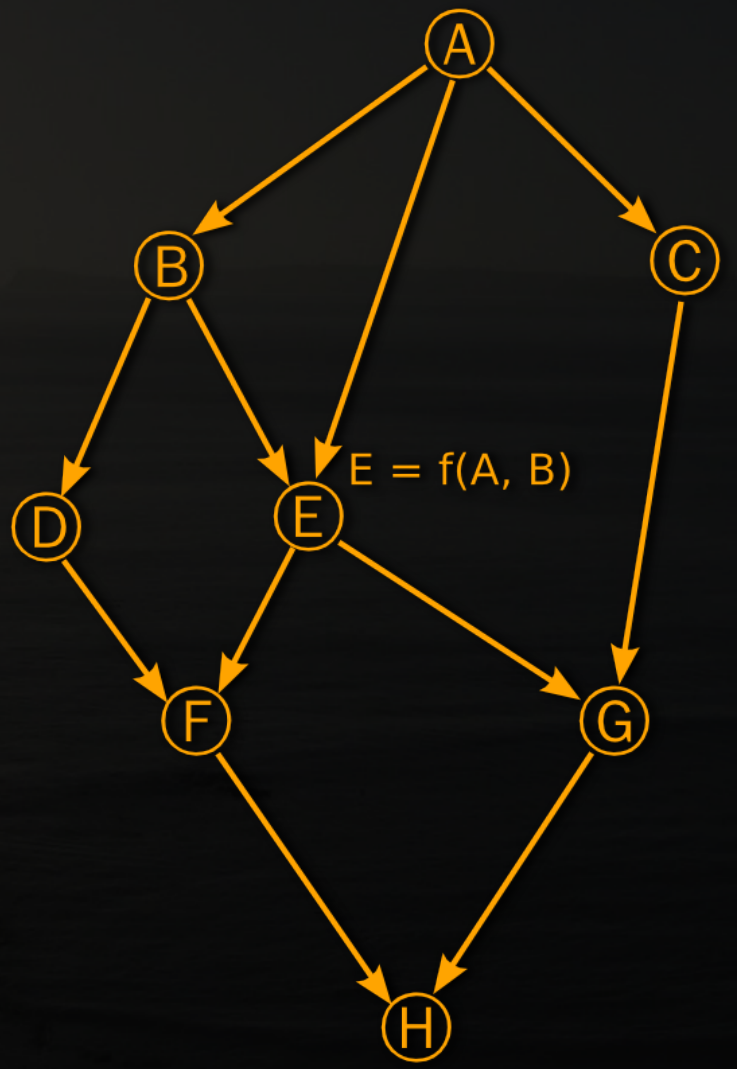
(1) Precise memory allocation:

- E only allocated (or computed) after both A and B have been computed
- E freed as soon as F and G are computed

No malloc/free, but *no GC either!*

- [Google will not use Java for core infrastructure because of GC]

No wasted memory



Back to the drawing board

(2) Precise execution order; direct knowledge of parallelizability.

- The program structure *is* the data dependency graph, which directly constrains the execution order, so there are no **race conditions**

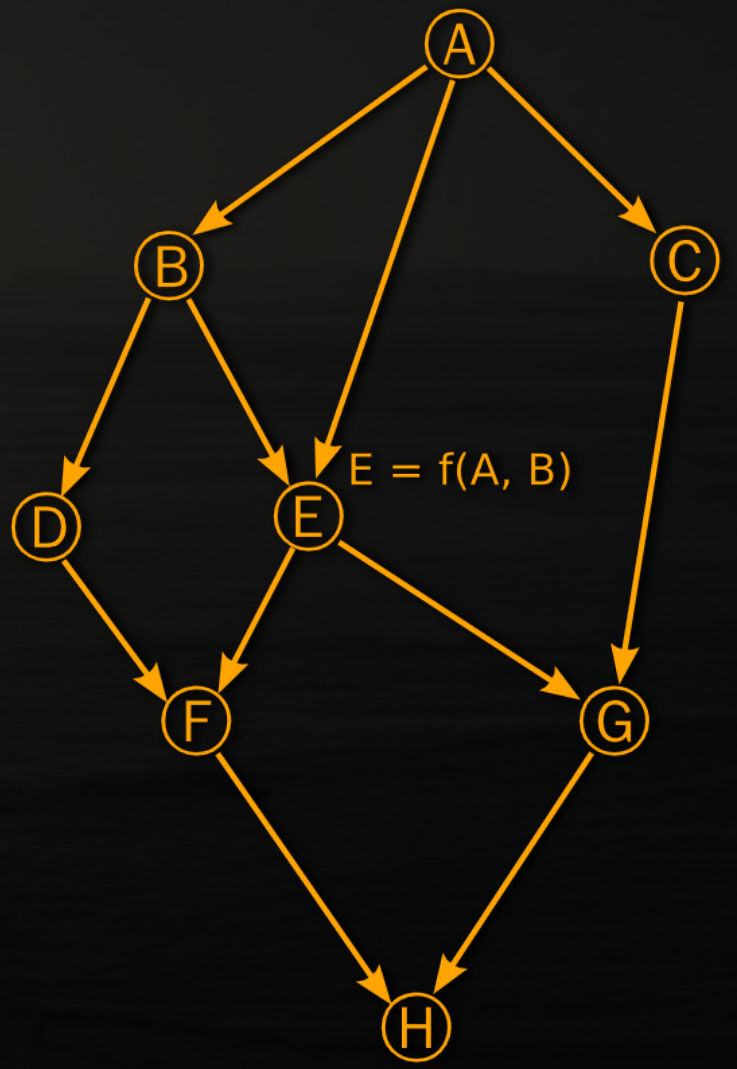
(3) There are no cycles in a DAG by definition, so **deadlocks** are impossible too.

(4) You can write the statements in your program *in any order* within each scope, and it will still run identically – you no longer need to do a *topological sort* in your head to artificially serialize your code (ABDEFCGH)

Back to the drawing board

(5) Next-gen source-code editor: forget text editors, edit the program directly as a DAG!

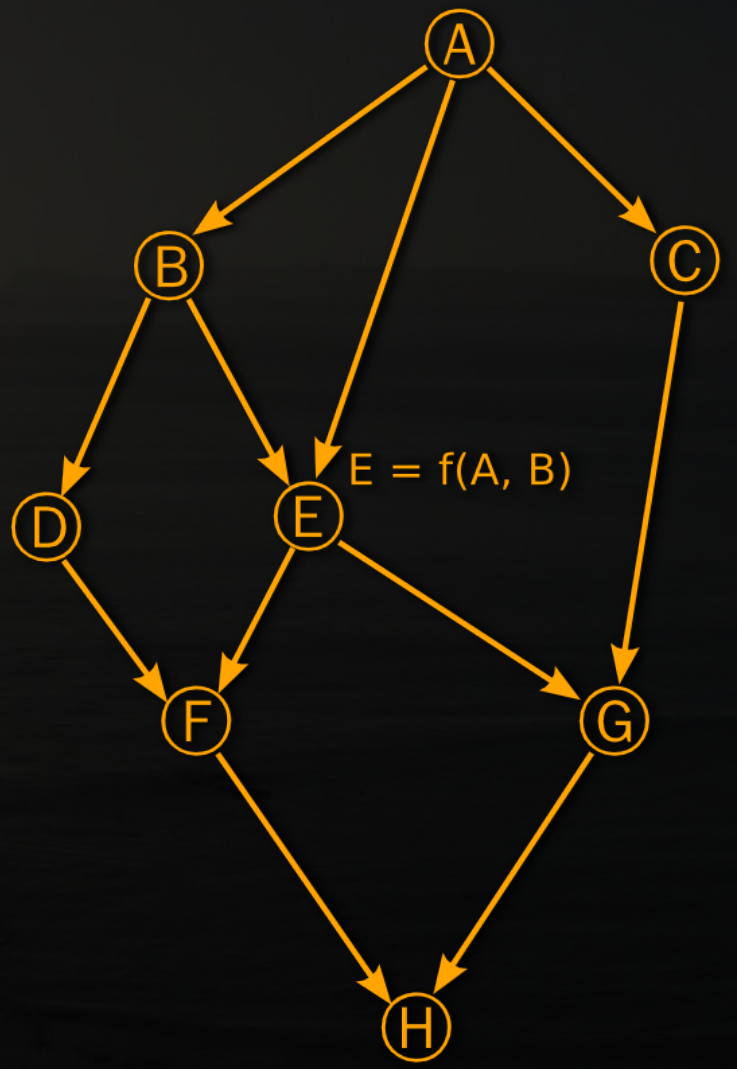
- Why do a topo sort on the source at all? It's completely artificial.
- Tap into the shape recognition power of the human visual system



Back to the drawing board

(6) Next-gen debugging

- Watch nodes light up on the DAG as they are computed
- Watch data literally flowing through the pipeline.
- Easy to add checkpointing for stop/restart because you know all data deps
  - Stopped at a breakpoint => run *backwards* one step and re-compute values at a previous node
  - Holy grail of debugging



Back to the drawing board

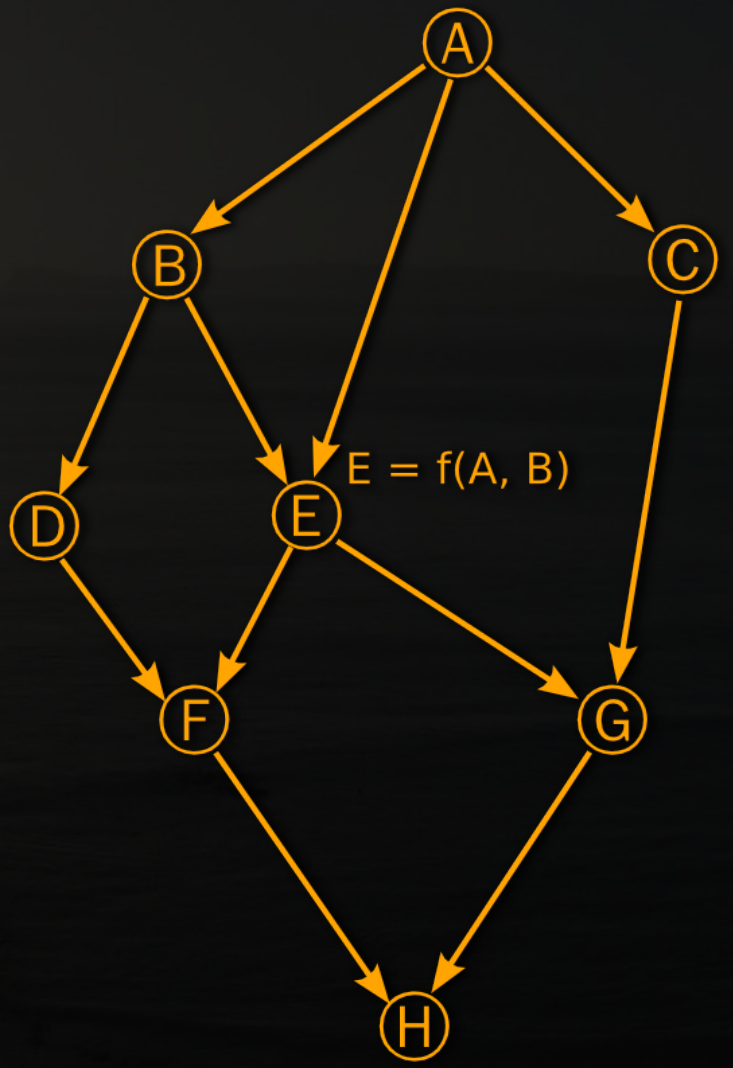
(7) There is no execution stack in Flow!

- Weird and rare among programming languages, Prolog is a notable exception
- Stack is conceptually replaced by conditional recursive expansion of sub-lattices within a lattice
- (Compiler will of course generate code that uses CPU stack)

Back to the drawing board

(8) Because each node is effectively a collection (Map/Set/List), we can calculate the big-Oh size of each collection, i.e. the big-Oh time required to compute it

- Every operation can now provide multiple equivalent algorithms or different parallelization strategies with different scaling characteristics, and Flow will switch between them at compiletime or runtime depending on input sizes
  - e.g. switch to single-threaded code for small inputs
- Find CPU bottlenecks and memory hogs before you even run your code (compile-time profiling!)



Category Theory

- **Further optimize implicit parallelization by data reordering and grouping**
  - Uses ***category theory*** for inference over function properties: associativity, commutativity, idempotence, …
  - Each property offers a degree of freedom to parallelize more efficiently
  - e.g. successive applications of functions that are associative and commutative can be reordered/grouped at will:
    (1+(3+(9+(2+(4+7))))) = (1+3+9)+(2+4+7)
    serial → parallel

Iteration and Turing-completeness

- The Structured Program Theorem: *every computable function can be implemented by combining subprograms in only three ways:*

  (1) Executing one subprogram, and then another subprogram (sequence)

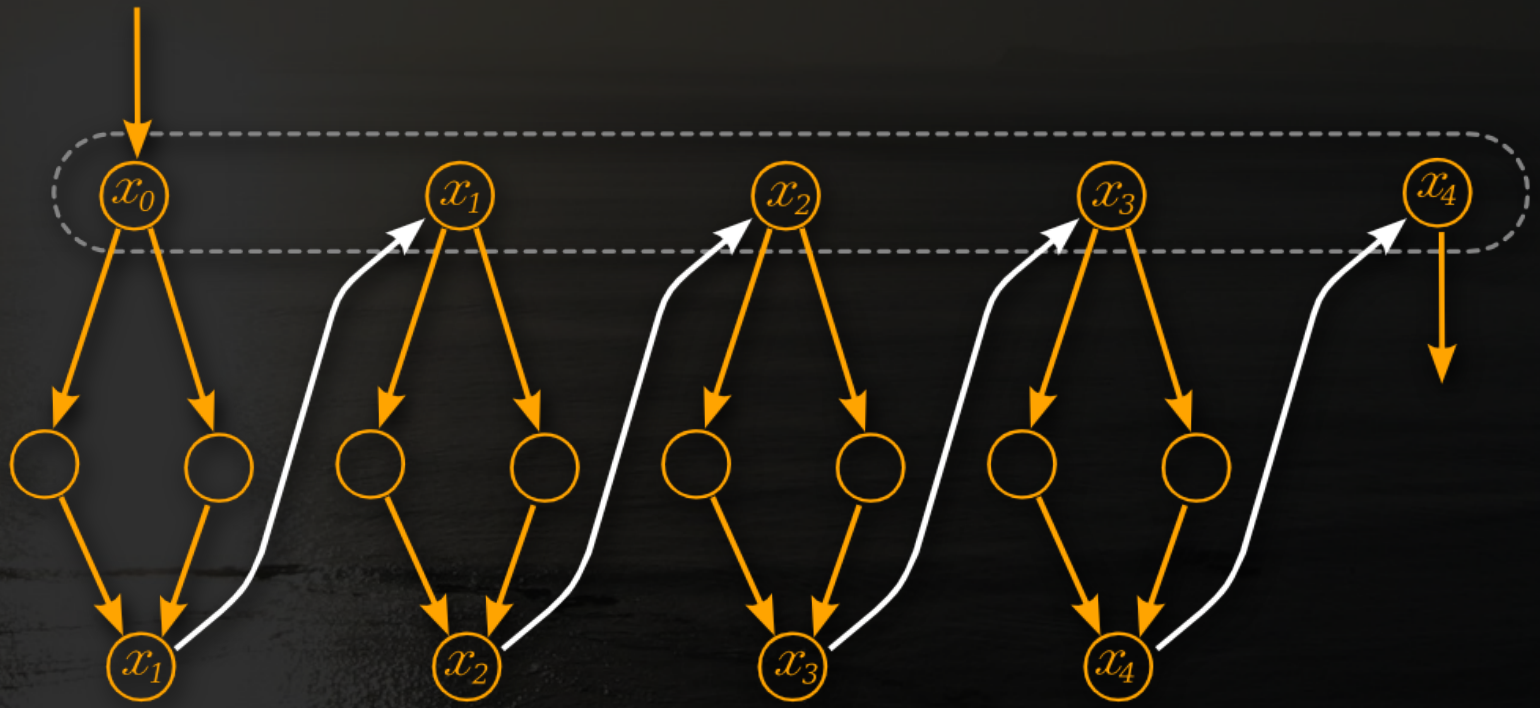
  - Composition of lattices in Flow

  (2) Executing one of two subprograms according to the value of a boolean variable (selection)

  - Trivial boolean function application in Flow

Iteration and Turing-completeness

(3) Executing a subprogram until a boolean variable is true (repetition)



Conclusion

- Flow imposes only the minimally invasive constraint on an imperative programming language to allow implicit parallelization with zero programmer effort
- *Many* different language syntaxes could be built on top of this paradigm (sharing the compiler backend)
- Mere mortal programmers can continue to work (mostly) as normal, the compiler will figure out the hard stuff
  - This (in theory) solves the multicore dilemma – it was a human issue to start with.

flowlang.net

twitter.com/LH

“Write Once,
Parallelize Anywhere”

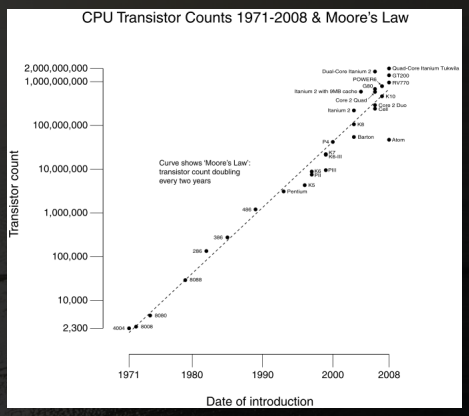
The Flow Programming Language:

An implicitly-parallelizing, programmer-safe language

Luke Hutchison

Moore's Law

- The number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every two years.



Moore's Law

- Computing is work; transistors do work; more transistors working in parallel should yield faster computers
  - => **Moore's Law has (so far) also applied to *CPU speed***
- BUT now we're hitting multiple walls:
  - **Transistor size**
    - tunneling; lithography feature size vs. wavelength
    - no more 2D density increases
  - **Thermal envelope**
    - function of frequency and feature size => we have hit a clockspeed wall
  - **Data width**
    - 128-bit CPUs? Need parallel control flow instead

But it's worse than that

It's not just the end of Moore's Law...

...it's the beginning of an era of **buggier software.**

Humans will **never** be good at
writing multi-threaded code –
our brains simply don't work like that.

(Abstractions like Futures only help a little bit.)

Parallel Programming Toolset

- Traditional locks: mutex, semaphore etc.
- Libraries: java.lang.concurrent
- Message passing: MPI; Actor model (Erlang)
- Futures, channels, STM
- **MapReduce**
- **Array programming**
  - Java7 ParallelArray
  - Intel Concurrent Collections (CnC)
  - Intel Array BuildingBlocks (ArBB)
  - Data Parallel Haskell (DPH)
  - ZPL

But—all require **programmer skill**, **extra effort** and **shoehorning of design.**

How to shoot yourself in the foot

Classic 1991 – http://bit.ly/hiwaG1

FORTRAN : You shoot yourself in each toe, iteratively, until you run out of toes, then you read in the next foot and repeat.

COBOL : USEing a COLT 45 HANDGUN, AIM gun at LEG.FOOT, THEN place ARM.HAND.FINGER on HANDGUN.TRIGGER and SQUEEZE. THEN return HANDGUN to HOLSTER. CHECK whether shoelace needs to be retied.

Lisp : You shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds...

BASIC : You shoot yourself in the foot with a water pistol.

Forth : Foot yourself in the shoot.

C++ : You accidently create a dozen instances of yourself and shoot them all in the foot. Providing emergency medical assistance is impossible since you can't tell which are bitwise copies and which are just pointing at others and saying "That's me, over there."

C : You shoot yourself in the foot.

My addition: **Explicit parallelization in any language** : You shoot in the yourself <*segfault*>

The state of Moore's Law, 2010

"*Finding:* There is no known alternative for sustaining growth in computing performance; however, no compelling programming paradigms for general parallel systems have yet emerged."

—The Future of Computing Performance: Game Over or Next Level?

Fuller & Millett (Eds.); Committee on Sustaining Growth in Computing Performance, National Research Council, The National Academies Press, **2010**, p.81

=> The Multicore Dilemma

The root of the problem

- *Purely* functional programming languages can be safely implicitly parallelized
  - No side-effects, no state
  - => threads cannot interact
  - e.g. several parallel versions of Haskell

The root of the problem

but

purely functional programming languages
are extremely hard for mere mortals
to use to solve real-world problems.

On the other hand,

getting explicit parallelization *right* –
and writing multithreaded code quickly –
may actually be *harder.*

Mere mortals

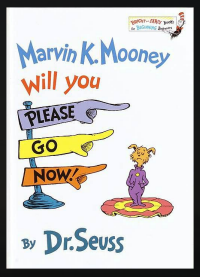
"*Recommendation:* Invest in research and development of programming methods that will enable efficient use of parallel systems not only by parallel systems experts *but also by typical programmers.*"

—The Future of Computing Performance: Game Over or Next Level?

Fuller & Millett (Eds.); Committee on Sustaining Growth in Computing Performance, National Research Council, The National Academies Press, **2010**, p.99

The root of the problem

- We like **imperative programming languages**: "do this, then this".



- But for imperative programming languages, it is impossible to tell the exact **data dependency graph** of a program by static analysis (by looking at the source).
  - (The data dependency graph tells you the order you need to compute values.)
  - Therefore you don't know what can be computed in parallel without:
    - (1) trusting the programmer (explicit parallelism) – VERY BAD
    - (2) guessing (static code analysis) – possibly/probably very bad
    - (3) trying and failing/bailing if you were wrong (STM)

Functional vs. imperative

- **Functional:**
  - *gather, pull, reduce*
- **Imperative:** same as functional, but adds
  - *scatter, push, produce*

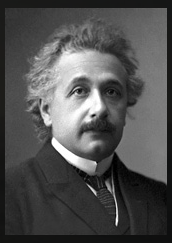
Training wheels

- How to *minimally* constrain imperative programming to allow for automatic implicit parallelization with guaranteed thread safety...

…*without* forcing the programmer to use training wheels?

The real root of the problem

- ...is **the ability to read the *current value of a variable*.** (Surprising. Foundational.)
  - => Einstein's theory of the relativity of simultaneity: there is no such thing as “now” given physical separation.
  - c.f. Values vs. variables.



- Remove the concept of “now”, and you *minimally restrict* a language such that it's impossible to create race conditions or deadlocks.
  - But you also don't have to program in purely functional style: Can support a *subset* of imperative, push-style programming.

Introducing Flow

- *Flow* is a new programming paradigm that enforces this restriction to solve the multicore dilemma while providing strong and specific guarantees about program correctness and runtime safety.

Introducing Flow

- Flow is a compiler framework that can be targeted by a wide range of different languages
  - [will probably plug into LLVM]
- ...and (eventually), a reference language

- It doesn't actually exist yet, come help make it exist
  - http://flowlang.net/

Goals of Flow

- **Solve the multicore dilemma**
  - Ubiquitous implicit parallelization, zero programmer effort
  - “Write once, parallelize anywhere”
    - Cluster : hadoop / MapReduce, MPI or similar
    - JVM via Java Threads
    - C via pthreads
    - CPU ↔ GPU via CUDA
      - Optimal load balancing via big-Oh analysis of computation and communication cost
- **Prevent programmers from shooting themselves in the foot**
  - Make race conditions, deadlocks, segfaults/NPEs, memory leaks **impossible**

Back to the drawing board

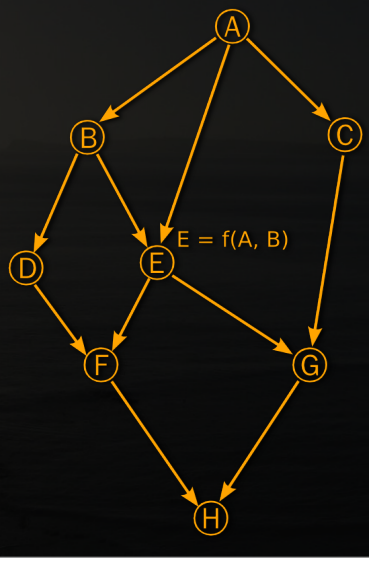
Flow enables implicit parallelization by

syntactically enforcing
every program to be
a lattice or partial ordering,

such that

the program source code itself is
the data dependency graph.

This makes parallelization trivial.



Back to the drawing board

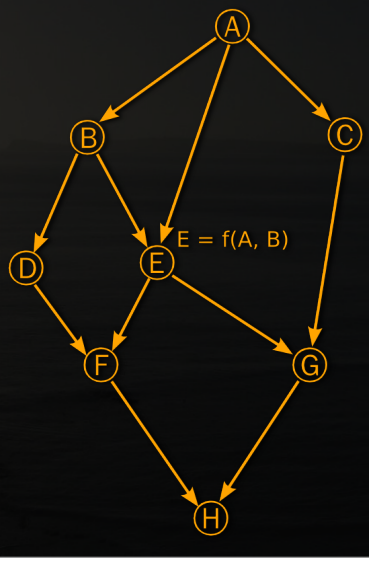
Also note that this does not eliminate “push” programming

- G can be an unordered collection

It also does not mean you can't “change the value of a variable”

- But you have to specify a specific timestamp, e.g.
  x@(t) = x@(t-1) + 1
  (then each value of x is a specific node in the DAG)

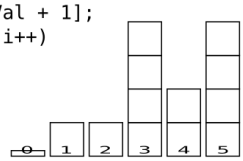
=> Minimally-restricted imperative programming with maximum implicit parallelization



An example

- Histogram generation

```
public int[] histogram(int[] input, int maxVal) {
    public int[] output = new int[maxVal + 1];
    for (int i = 0; i < input.length; i++)
        output[input[i]]++;
    return output;
}
```



{5,3,1,4,3,4,2,3,5,3,5,5} -> {0, 1, 1, 4, 2, 4}

- Multithreaded version? Two options:
  (1) locks (=> contention!)
  (2) keep separate copies in TLS; combine at end of operation
- Both options are manual
  => not easily composable, load-balanceable or scalable

An example

- More complex example

```
// Generate a histogram of avg number of spaces vs. string len

@SuppressWarnings("unchecked")  // Stupid Java generics
public int[][] avgNumSpacesGivenStrLen(String[] input, int maxLen) {

    // Alloc memory (even GC'd languages require this sort of stuff)
    ArrayList[] strsOfLen = new ArrayList[maxLen + 1];
    for (int i = 0, n = maxLen + 1; i < n; i++)
        stsOfLen[i] = new ArrayList();

    // Scatter inputs by string length
    for (int i = 0; i < input.length; i++)
        strsOfLen[input[i].length()].add(input[i]);

    // Calc avg number of spaces among all inputs with same string length
    public int[] output = new int[maxLen + 1];
    for (int len = 0; len <= maxLen; len++) {
        ArrayList<String> strs = (ArrayList<String>) strsOfLen[len];
        int totSpaces = 0;
        for (String str : strs)
            totSpaces += countSpaces(str);
        output[len] = strs.size() == 0 ? 0.0f : totSpaces / (float) strs.size();
    }
    return output;
}
```

- strsOfLen[*j*] can be an *unordered collection* (Set), does not affect result
- Allows writes to be interleaved or reordered => auto TLS split

An example

- More complex example

```
// Generate a histogram of avg number of spaces vs. string len

@SuppressWarnings("unchecked")  // Stupid Java generics
public int[][] avgNumSpacesGivenStrLen(String[] input, int maxLen) {

    // Alloc memory (even GC'd languages require this sort of stuff)
    ArrayList[] strsOfLen = new ArrayList[maxLen + 1];
    for (int i = 0, n = maxLen + 1; i < n; i++)
        stsOfLen[i] = new ArrayList();

    // Scatter inputs by string length
    for (int i = 0; i < input.length; i++)
        strsOfLen[input[i].length()].add(input[i]);

    // Calc avg number of spaces among all inputs with same string length
    public int[] output = new int[maxLen + 1];
    for (int len = 0; len <= maxLen; len++) {
        ArrayList<String> strs = (ArrayList<String>) strsOfLen[len];
        int totSpaces = 0;
        for (String str : strs)
            totSpaces += countSpaces(str);
        output[len] = strs.size() == 0 ? 0.0f : totSpaces / (float) strs.size();
    }
    return output;
}
```

- The less we constrain subproblems, the more parallelizable the code
- To relax constraints, must understand properties of functions / collections

An example

- More complex example

```
// Generate a histogram of avg number of spaces vs. string len

@SuppressWarnings("unchecked")  // Stupid Java generics
public int[][] avgNumSpacesGivenStrLen(String[] input, int maxLen) {

    // Alloc memory (even GC'd languages require this sort of stuff)
    ArrayList[] strsOfLen = new ArrayList[maxLen + 1];
    for (int i = 0, n = maxLen + 1; i < n; i++)
        stsOfLen[i] = new ArrayList();

    // Scatter inputs by string length
    for (int i = 0; i < input.length; i++)
        strsOfLen[input[i].length()].add(input[i]);

    // Calc avg number of spaces among all inputs with same string length
    public int[] output = new int[maxLen + 1];
    for (int len = 0; len <= maxLen; len++) {
        ArrayList<String> strs = (ArrayList<String>) strsOfLen[len];
        int totSpaces = 0;
        for (String str : strs)
            totSpaces += countSpaces(str);
        output[len] = strs.size() == 0 ? 0.0f : totSpaces / (float) strs.size();
    }
    return output;
}
```

Map

Reduce

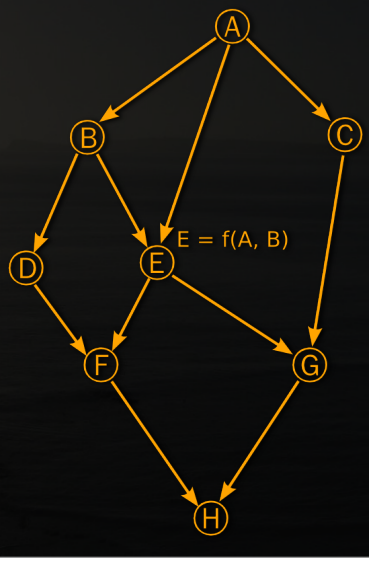
- Turns out any function can be split into Map and Reduce stages
- But MapReduce doesn't incl flow control, + MR can't be auto-generated yet

Back to the drawing board

Flow syntactically enforces that a program's structure be a partial ordering or DAG (specifically a *lattice*).

Each node is the result of a function application of some sort

=> the process of computing each node can be thought of as a MapReduce operation



Back to the drawing board

Because Flow enforces program structure to be a partial ordering, some amazing properties emerge:

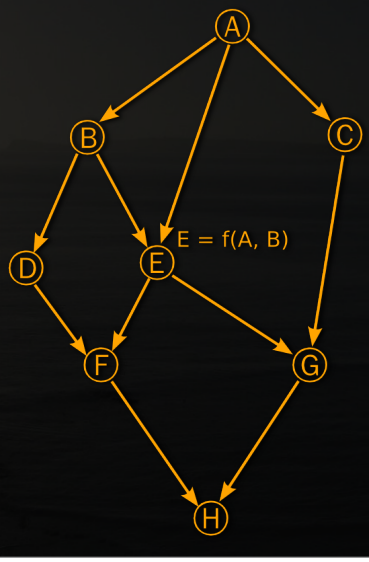
(1) Precise memory allocation:

- E only allocated (or computed) after both A and B have been computed
- E freed as soon as F and G are computed

No malloc/free, but *no GC either!*

- [Google will not use Java for core infrastructure because of GC]

No wasted memory



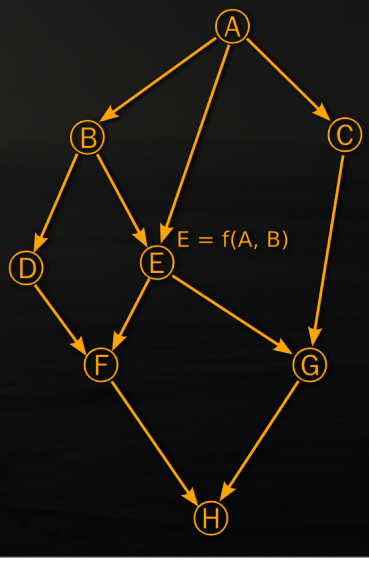
Back to the drawing board

(2) Precise execution order; direct knowledge of parallelizability.

- The program structure *is* the data dependency graph, which directly constrains the execution order, so there are no **race conditions**

(3) There are no cycles in a DAG by definition, so **deadlocks** are impossible too.

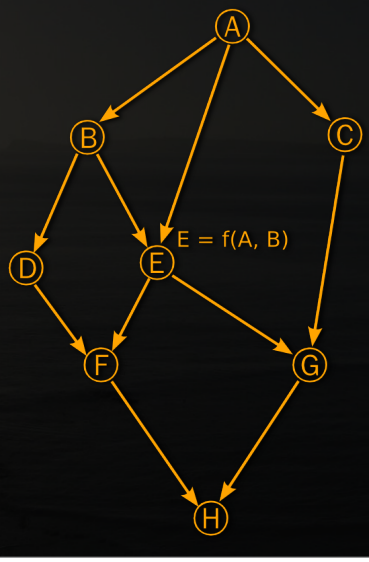
(4) You can write the statements in your program *in any order* within each scope, and it will still run identically – you no longer need to do a *topological sort* in your head to artificially serialize your code (ABDEFCGH)



Back to the drawing board

(5) Next-gen source-code editor: forget text editors, edit the program directly as a DAG!

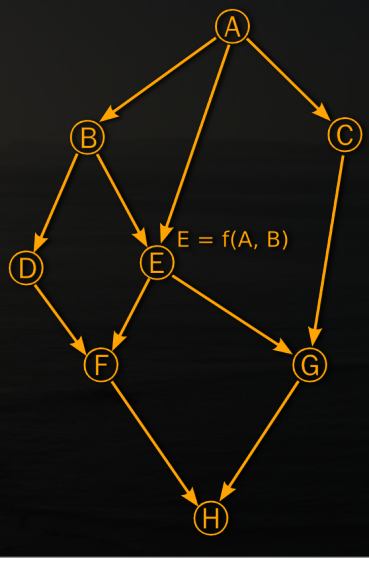
- Why do a topo sort on the source at all? It's completely artificial.
- Tap into the shape recognition power of the human visual system



Back to the drawing board

(6) Next-gen debugging

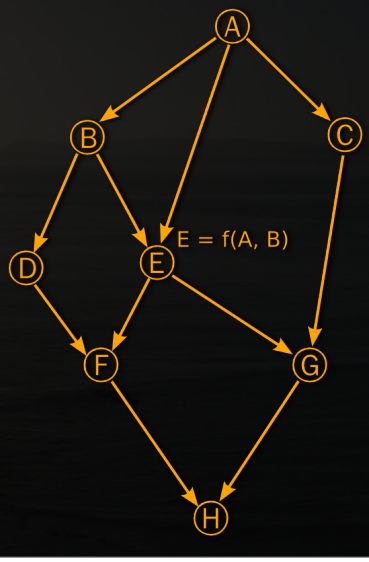
- Watch nodes light up on the DAG as they are computed
- Watch data literally flowing through the pipeline.
- Easy to add checkpointing for stop/restart because you know all data deps
  - Stopped at a breakpoint => run *backwards* one step and re-compute values at a previous node
  - Holy grail of debugging



Back to the drawing board

(7) There is no execution stack in Flow!

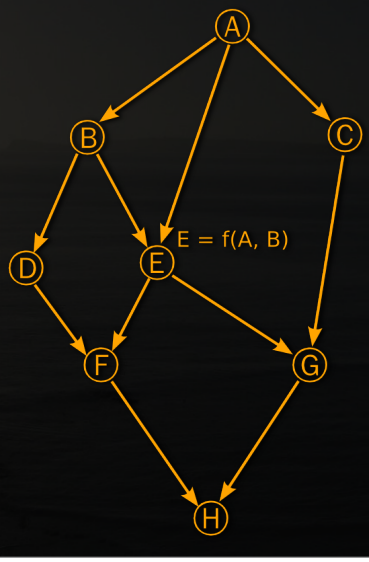
- Weird and rare among programming languages, Prolog is a notable exception
- Stack is conceptually replaced by conditional recursive expansion of sub-lattices within a lattice
- (Compiler will of course generate code that uses CPU stack)



Back to the drawing board

(8) Because each node is effectively a collection (Map/Set/List), we can calculate the big-Oh size of each collection, i.e. the big-Oh time required to compute it

- Every operation can now provide multiple equivalent algorithms or different parallelization strategies with different scaling characteristics, and Flow will switch between them at compiletime or runtime depending on input sizes
  - e.g. switch to single-threaded code for small inputs
- Find CPU bottlenecks and memory hogs before you even run your code (compile-time profiling!)



Category Theory

- **Further optimize implicit parallelization by data reordering and grouping**
  - Uses ***category theory*** for inference over function properties: associativity, commutativity, idempotence, …
  - Each property offers a degree of freedom to parallelize more efficiently
  - e.g. successive applications of functions that are associative and commutative can be reordered/grouped at will:
    (1+(3+(9+(2+(4+7))))) = (1+3+9)+(2+4+7)
    serial → parallel

Iteration and Turing-completeness

- The Structured Program Theorem: *every computable function can be implemented by combining subprograms in only three ways:*

  (1) Executing one subprogram, and then another subprogram (sequence)

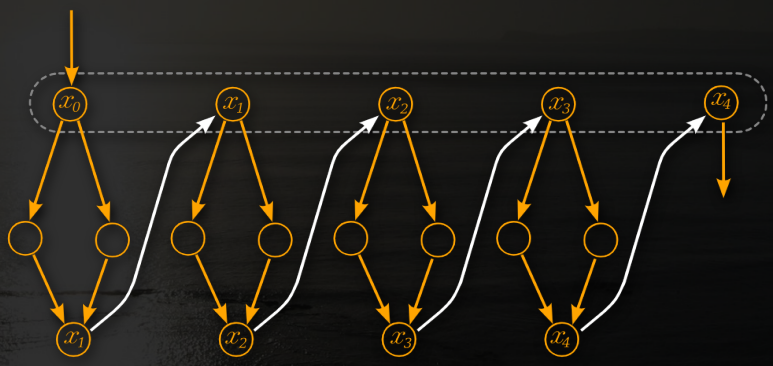
  - Composition of lattices in Flow

  (2) Executing one of two subprograms according to the value of a boolean variable (selection)

  - Trivial boolean function application in Flow

Iteration and Turing-completeness

(3) Executing a subprogram until a boolean variable is true (repetition)



Conclusion

- Flow imposes only the minimally invasive constraint on an imperative programming language to allow implicit parallelization with zero programmer effort
- *Many* different language syntaxes could be built on top of this paradigm (sharing the compiler backend)
- Mere mortal programmers can continue to work (mostly) as normal, the compiler will figure out the hard stuff
  - This (in theory) solves the multicore dilemma – it was a human issue to start with.


flowlang.net
twitter.com/LH