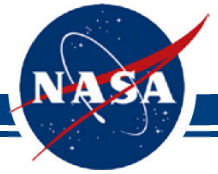
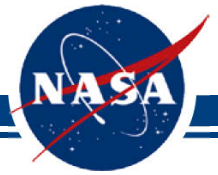


Functional Programming and Multi-core

Kim P. Gostelow
Jet Propulsion Laboratory, California Institute of Technology

August 2 , 2011



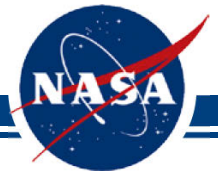
What Multi-core Allows Us To Do

- Concurrency/Parallel Programming
  - For speed and reliability
  - For general computing
    - Not just for special applications
- Policy-based Computing
  - Policies change in real-time
  - Examples
    - Reliability requirements
    - Timeliness requirements
      - Precision
    - Power and other resources
- A new knob to turn for system tradeoffs
  - More power > more speed and/or reliability
  - Without affecting the other, ongoing computations

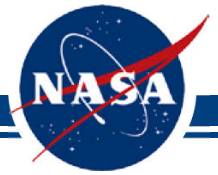
Examples

- Produce the best result in this estimation loop in 2 milliseconds.
- Use sequential-TMR for attitude control for the next 3 hours.
- You have 3 watts for entry.
- Do this in 2 milliseconds, or quit.
- If a processor or comm link fails, try again once.

Isn't Concurrent Programming Hard Enough Already?



- It need not be hard
- An old idea: Functional Programming
  - Every call to a given function produces the same result
- This ensures
  - Concurrency is the default, not sequentiality
  - Extra-functional properties
    - Automatic parallelization at run-time
      - Any two non-dependent sub-expressions can run at the same time
    - Can copy/move/stop and restart any function at any time



What is a Functional Language?

The relation f: A->B is a ***function*** if:

For-all a **in** A there is a *unique* b **in** B such that f(a) = b

- For programming, the consequences of the above are:
  - Immutable values
    - Can define a value only once (a variable has only one meaning)
    - Variables are mathematical variables
      - *Not* memory cells that can be modified/replaced/updated
  - No shared memory
  - Deterministic
  - No side-effects

Functions

The relation f: A->B is a ***function*** if:

For-all a **in** A there is a *unique* b **in** B such that f(a) = b

Not a function

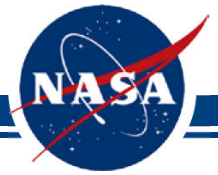
```
extern int sum;

int A(int a) {
   sum = get_clock();
   for (int i=0; i<a; i++)
      sum += f(i);
   return sum;
}
```

Function

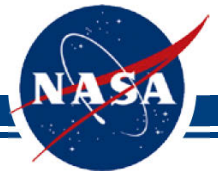
```
int B(int a, int time) {
   for i=0 .. a
      v[i] = f(i);
   return accum(v) + time;
}
```

Can run in parallel if f is a function



Requirements on Implementing Policies

- The policies
  - Reliability, Timeliness, and Power
- The requirements
  - It must be system code and operate automatically
  - It must not involve the programmer
  - Operates in Real-time

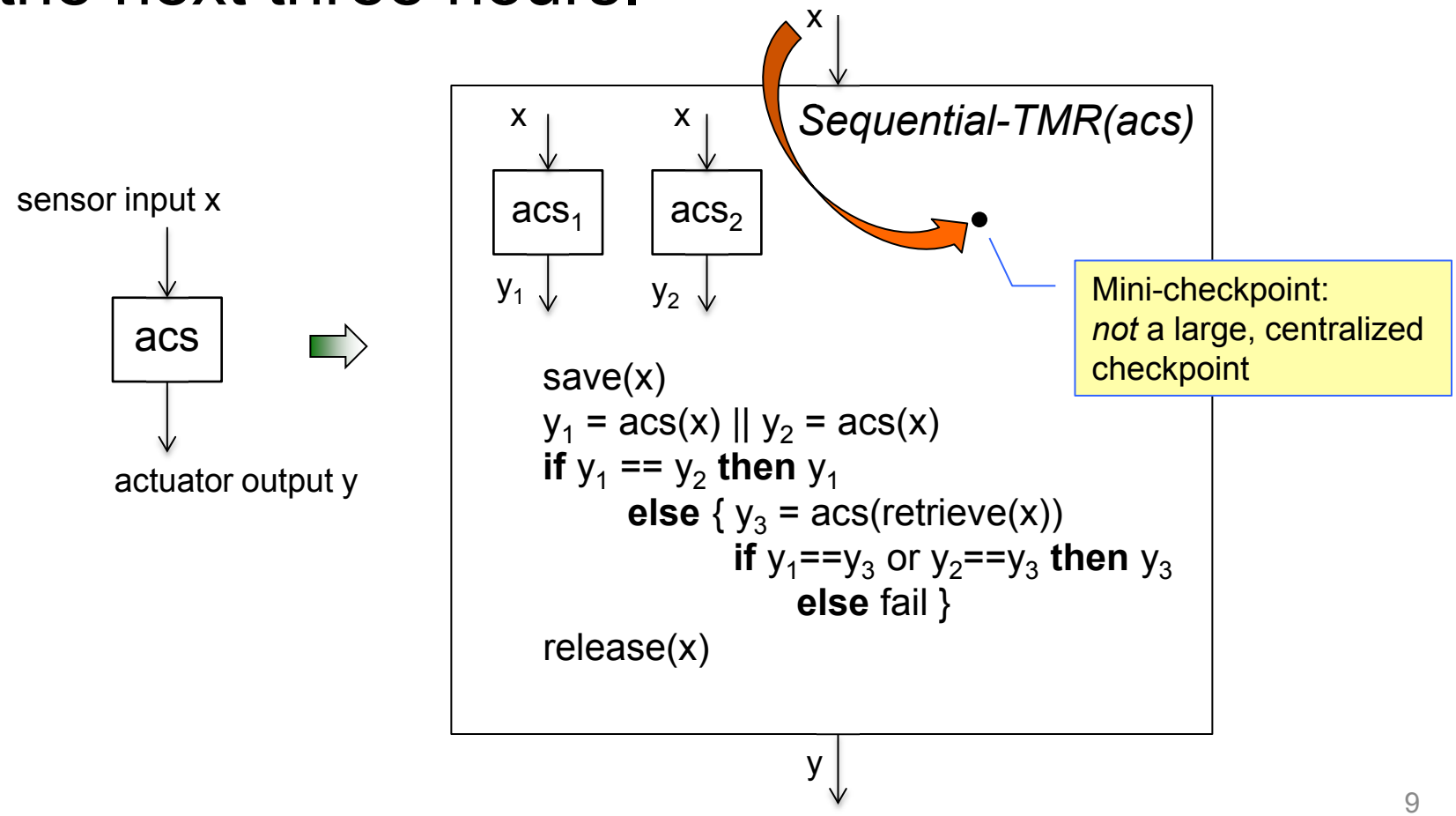


How Do We Implement Policies?

- At function calls
  - Extra-functional code is inserted and wraps the affected programs
  - Functions called are assigned to differing numbers of processors

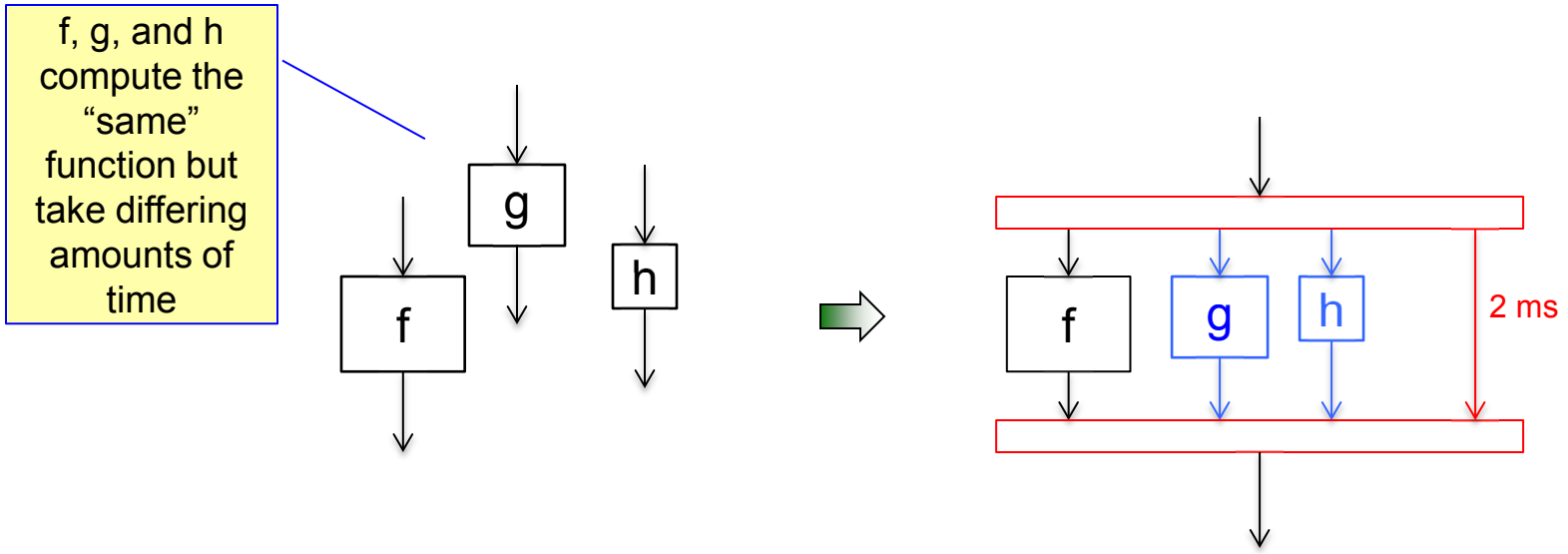
Example: Reliability

- "Use sequential-TMR on attitude-control for the next three hours."



Example: Timeliness

- "Get the best estimate you can in 2 ms."



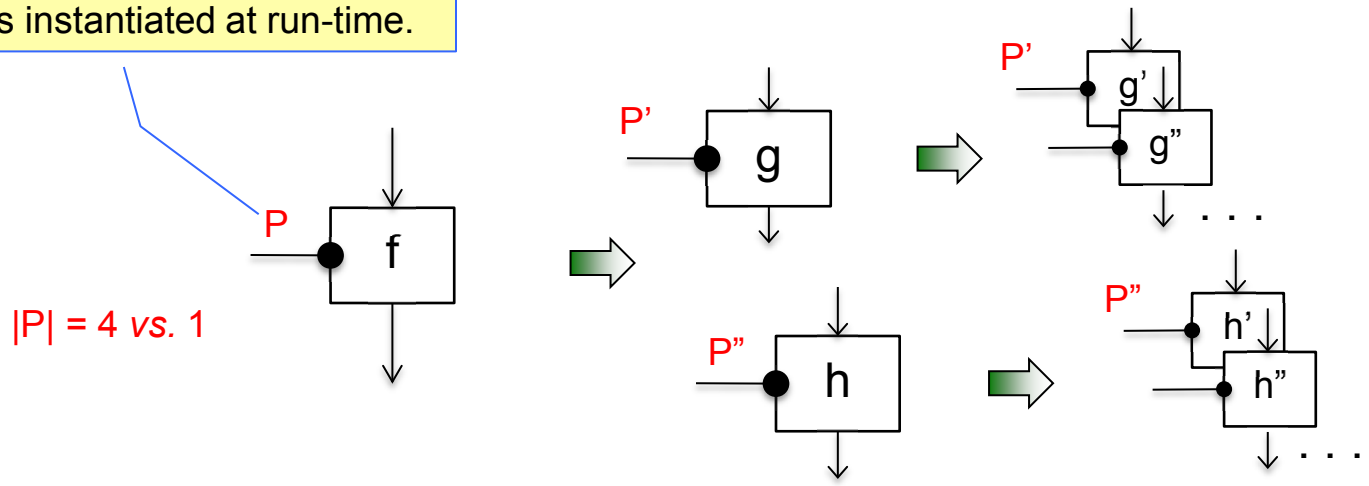
Example: Power

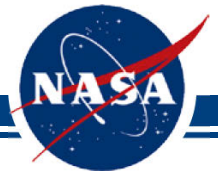
- "You have 3 watts to do entry."

let f(x) = { … g(u) … h(v) … }
where g(x) = { … g'() … g"() … }
h(x) = { *etc.* }

This might translate into an assignment of 4 cores to function f rather than the usual 1 core.

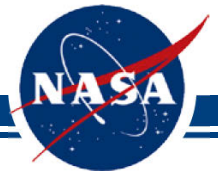
P is the set of processors allocated by the system to run f and is instantiated at run-time.





Not Everything

- Not everything should be done functionally
- ***Small*** parts of most programs have some kind of non-functional behavior that requires special treatment
  - Clock: get_time()
  - Database access
  - Nested Timeliness policies



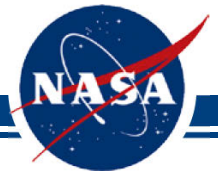
Can Loops and I/O Be Functional?

The short answer is ***yes***

- Loops
  - A loop is a recurrence relation
    - Previous “sum of f(i)” was empty recurrence with empty state
  - An old idea: We make the loop state explicit

- Functional I/O has been a long-standing problem
  - Unsatisfactory “solutions”: Monads in Haskel; **mutable** in Fortress; **unique** type in Clean
  - Different view here
    - A file is a logical data structure
      - A list, array, tree,or other sturcture
      - It is not a sequence of raw, physical records
    - Separate logical order from physical position

Example: A Simple Loop (1)



The factorial sequence (1 1 2 6 24 120 ...) is defined by the *recurrence relation*

$s_i = i \times s_{i-1}$ where $s_0 = 1$.

```
int factorial(int n) {
  int s= 1;

  for (i=1; i<n+1; i++) {
    s = i * s;
  }
  return s;
}
```

A typical C program to compute the n^{th} factorial for n = 0, 1, …

Example: A Simple Loop (2)

As a state-aware functional program

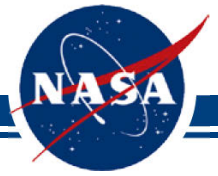
- Variable “s” represents a *stream* of state values s[0], s[1], …
- The computer need keep only the **current** and the **next** s[i] at any time.

```
int factorial(int n) {
  return
    initially
        int s = 1;
    recur j = 1..n+1 {
        next s = j * s;
    }
    finally last s;
}
```

The **initially** section defines s[0] and the number of slots needed.

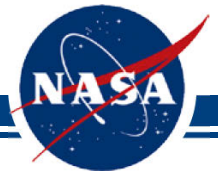
An implicit last step:
s[i+1] = **next** s

The **finally** section says what to return at loop end – that is, s[n].



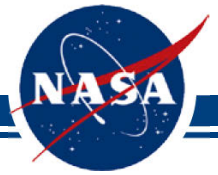
I/O

- Different view here
  - A file is a logical data structure
    - A list, array, tree,or other sturcture
    - It is not a sequence of raw, physical records
  - Separate logical order from physical position
- Examples
  - A list of text records ti: the list position i is written along with the contents as <i, ti>; the order of <i,ti> in the actual file is not necessarily “in order”.
  - HDR for files uses a tree
- Consequence for the programmer
  - Look at a file as another name for a variable
    - Just as if assigning values to an array



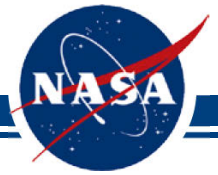
Simple Rules Work

- To write programs as functions
  - Often only small changes are needed
    - Move state changes outside
    - Move calls to non-functions outside
  - No shared memory
  - Don’t create state where it isn’t necessary
    - Don’t re-use variable names – each name has one meaning, one value



Summary

- *Policy-based computing* requires programs to have extra-functional properties
- The extra-functional properties of interest are
  - Start/stop/restart
  - Copy/reproduce
  - Move and distribute
- These properties are assurred when programs are functions.



References

- Dennis and Misunas “A Preliminary. Architecture for a Basic Data-Flow Processor”. 1975 Sagamore Computer Conference on. Parallel Processing
- Gostelow “The Design of a Fault-Tolerant, Real-Time, Multi-Core Computer System” IEEE Aerospace Conference, Big Sky, MT 2011
- Milojicic, Douglis, Paindaveine, Wheeler, and Zhou “Process Migration Survey” ACM Computing Surveys Sept 2000.
- Arvind, Gostelow, and Plouffe “Indeterminacy, Monitors, and Dataflow” Proc 6th ACM Symposium on Operating Systems Principles
- Fortress Programming Language http://projectfortress.java.net/