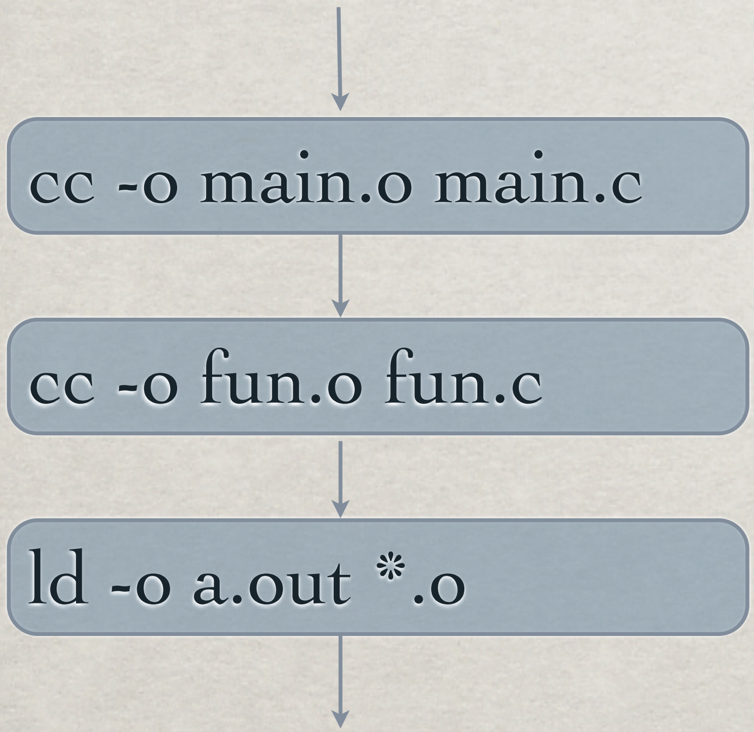
Parallel programming

Day 5
Dataflow programming

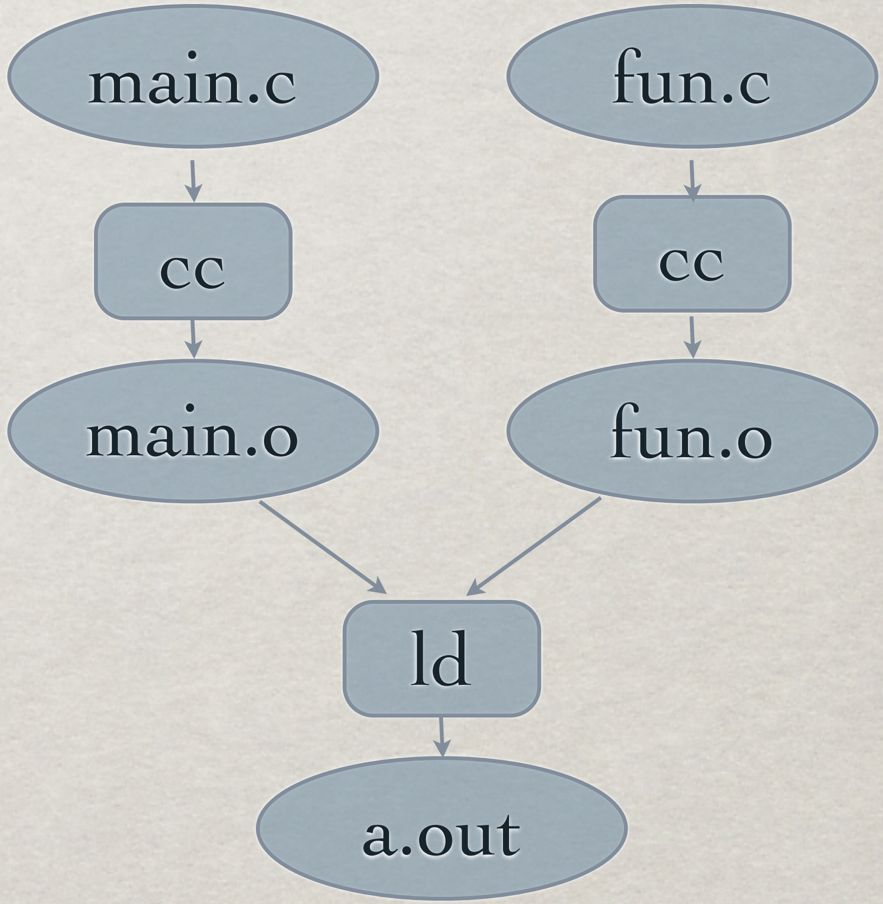
Overview

- Turing machines / imperative programming:
  - "do this, **then** do that" : **time** ordered
  - instructions "active," data is "passive"
- Dataflow programming:
  - "to compute this, first compute that" : **dependency** ordered
  - data is "active," instructions "passive"

Example



Imperative program



Dataflow program

Functions vs streams

- Functional and dataflow: both represent the **functional dependencies between values:** almost **no hidden state**

- What's the difference?

  - Functional "programs" compute 1 value
    (Any more requires read-eval-print or "infinite recursion")

  - Dataflow programs (usually) operate over **streams**: "flow" of inputs

Why it matters

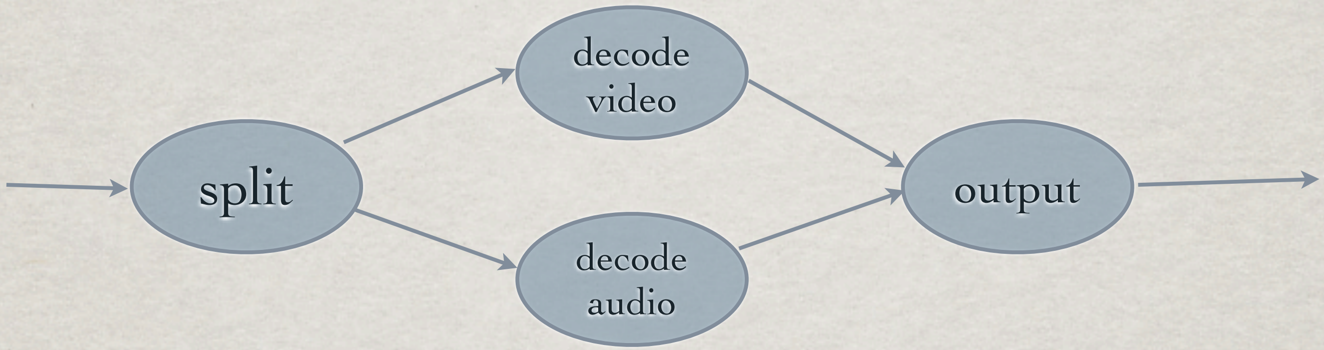
- **Programmer sanity**: no hidden state means sub-programs can be analyzed/tested individually
- Relevant to this summer school:
  - Two operations that have their input ready can run simultaneusly: **parallelization is easy**
  - synchronization needed at "merge" nodes
  - Inherently **heterogeneous** concurrency

Where and how

- Dataflow is a **style of programming** which can be used in most parallel programming languages:
    - **identify dependencies** between data and operations
    - let the **environment schedule execution**
- Few pure "dataflow programming languages," most common:
    - VHDL/Verilog
    - Spreadsheet formulas

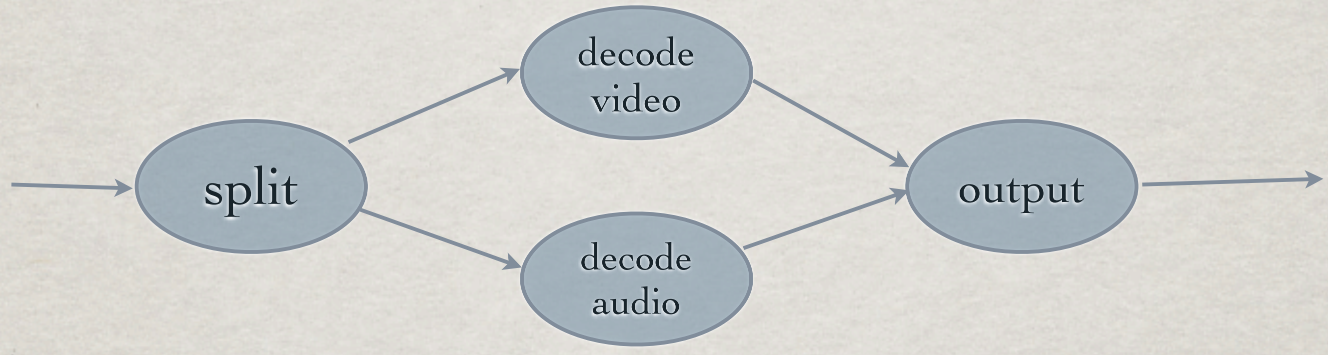
Techniques

Flavors of dataflow programs



- Synchronous: global time step, all nodes activated on their "current input"
- Asynchronous: each node computes at its own speed, may suspend on missing input or full output buffer

Flavors of dataflow programs



- Static: the number of nodes and edges stays the same over time
- Dynamic: nodes and edges appear and disappear over time

Example uses

Example uses

- Static, asynchronous:
  video compression/decompression

Example uses

- Static, asynchronous:
  video compression/decompression

- Static, synchronous:
  video filtering

Example uses

- Static, asynchronous:
  video compression/decompression
- Static, synchronous:
  video filtering
- Dynamic, asynchronous:
  any functional computations with recursion

Example uses

- Static, asynchronous:
  video compression/decompression
- Static, synchronous:
  video filtering
- Dynamic, asynchronous:
  any functional computations with recursion
- Dynamic, synchronous: N/A

How to execute static synchronous programs

- Program = graph
  - **Operations as nodes, data as edges**
- Implement each node as a **function + ready bit**
- Use a **buffer for each data edge**
- At each cycle:
  - execute ready functions
  - store result into output buffer
  - mark functions on the other side of buffers as ready

How to execute asynchronous programs

- Program = graph
  - **Operations as nodes, data as edges**
- At any point during execution, nodes are **activated** based on data availability
- The environment can **fire** operations in any order, **possibly in parallel**
- Implementations: **pipeline**, **job pooling**
- At low level: **FIFOs** and/or **dataflow variables**

Queues / FIFOs

- Each operation runs in a process connected to other processes using FIFOs representing data edges
- “First in, first out”
  - can be implemented in software as array + pointers to “head” and “tail”
  - Can be implemented in hardware too
- Processes **suspend** when:
  - read from empty: tail reaches head
    (resume when data becomes available)
  - write to full: head reaches tail
    (resume when space available in FIFO)

Dataflow variables

- Each operation is expressed as a function which saves its return value into a dataflow variable
- **Dataflow variable = data + state**
  - Data: a value, like usual
  - **State: full/empty**
- Initially empty
- "Read from empty" suspends the current processes and writes the process ID to the dataflow variable
- Write to suspended wakes up the process(es) whose ID is saved in the variable

Hybrid dataflow

- Often:
  pipes and dataflow variables are **narrow**
  - Can only pass **a few bytes at a time efficiently** through the device
- Combine with a **shared data store**
  - Shared memory for multiple threads
  - Shared filesystem for Unix processes
- **Pass the names (references)** to data items through the dataflow device,
- **Use the shared store for the body** of data items

High-level reasoning about dataflow behavior

Latency

- Latency for sub-graphs of a dataflow program:
  time between **arrival of input** item and **production of corresponding output** item

- Latency constrained by:
  - **latency of data links** (secs)
  - **performance of computing resource** (secs/op)
  - **latency to shared data store** (secs) for hybrid dataflow

- General constraint:
  L > max-flow(p ⊙ g)
  p = program graph (op nodes + dependency edges)
  g = resource graph (PUs + data links)
  p ⊙ g : mapping of program onto resources
  max-flow : **sum of latencies along the longest path** through the resource graph that must be followed to compute 1 output
  also called **critical path**

Latency optimization

Latency optimization

- Some problems are **latency-constrained**:

Latency optimization

- Some problems are **latency-constrained**:
  - Medical equipment: eg $L < 40$ms

Latency optimization

- Some problems are **latency-constrained**:
  - Medical equipment: eg $L < 40$ms
  - Weather forecast: eg $L < 1$ day

Latency optimization

- Some problems are **latency-constrained**:
  - Medical equipment: eg $L < 40$ms
  - Weather forecast: eg $L < 1$ day
- "Levers" to reduce latency:

Latency optimization

- Some problems are **latency-constrained**:
  - Medical equipment: eg L < 40ms
  - Weather forecast: eg L < 1 day
- "Levers" to reduce latency:
  - faster PUs, faster data links

Latency optimization

- Some problems are **latency-constrained**:
    - Medical equipment: eg L < 40ms
    - Weather forecast: eg L < 1 day
- "Levers" to reduce latency:
    - faster PUs, faster data links
    - shorter critical path for the computation

Throughput

- Throughput for sub-graphs of a dataflow program:
  how many nodes are fired by second
- Throughput constrained by:
  - **bandwidth of data links** (bytes/sec)
  - **max throughput of computing resource** (ops/sec)
  - **bandwidth of shared data store** (bytes/sec) for hybrid dataflow
- General constraint:
  T < min(Bin/a, MaxInternalT, Bout/b)
  a = number of input bytes consumed per op
  b = number of output bytes produced per op
  Bin/Bout = bandwidth of data links where computation is mapped
  MaxInternalT = maximum internal throughput for the sub-parts

Throughput optimization

Throughput optimization

- Most computing goals are about **throughput maximization**

Throughput optimization

- Most computing goals are about **throughput maximization**
  - Video / games: frames / sec

Throughput optimization

- Most computing goals are about **throughput maximization**
  - Video / games: frames / sec
  - Build system: compilations / sec

Throughput optimization

- Most computing goals are about **throughput maximization**
  - Video / games: frames / sec
  - Build system: compilations / sec
- "Levers" to increase throughput:

Throughput optimization

- Most computing goals are about **throughput maximization**
  - Video / games: frames / sec
  - Build system: compilations / sec
- “Levers” to increase throughput:
  - bandwidth of data links

Throughput optimization

- Most computing goals are about **throughput maximization**
  - Video / games: frames / sec
  - Build system: compilations / sec
- “Levers” to increase throughput:
  - bandwidth of data links
  - Number of parallel PUs

Dataflow with Unix pipes

Unix pipes

- Basic operations of processes: **read/write**
- Basic interface to the environment: **file names**
- A kind of special files: FIFOs
  - effect: direct data link between 2 processes
  - create: **mkfifo** (named), **pipe** (unnamed)
  - use: **read/write** - read empty blocks, write resumes suspended process

Image filters++

- Basic idea: hybrid, static asynchronous
  - Use **pipes** to pass the **image names**
  - Use the **filesystem** to pass the **image data**
  - A name goes through the pipe only after the image data has been committed to file
- Merge / synchronize:
  successive reads to multiple input pipes