Introduction

- First articulated by Jack Dennis at MIT in 1969

- In dataflow machines, hardware is optimized for fine-grain data-driven parallel computation

- Data-driven computation is intuitively appealing because:
  - only data dependences constrain parallelism
  - Programs usually represented as graphs

- Although around for 20 years, little impact on mainstream computing
  - More recently, convergence is happening (Ianucci, P-RISC effort)
Terminology

- Instructions are nodes (e.g., +, -, ...).
- Data items are tokens (e.g., 8.2, true, 4, ...).
- Producers of tokens are connectes to consuemers by arcs.
- Entry points to nodes are input ports.
- A node is enabled by an enabling rule. This typically is when ALL arcs have tokens on their inputs (strict enabling rule). non-strict allows firing before all arcs have tokens.
- A node fires after it is enabled, and in this process consumes tokens on its input arcs and produces tokens on its output arcs.
Node Types

- Functional: Output depends only on inputs and node type (e.g., +, -, ...).

- Conditional: Depending on control, token picked up from value input is passed onto true-output arc or false-output arc.

- Merge: Whenever there is a token on any of the input arcs, it is immediately copied to the output arc.

  - non-stick firing rule
  - acts as serializer/merger
Iteration, Recursion, Reuse and Reentrancy

- using the same graph to perform computation on different sets of data.
- In the following, we assume that things mess up if a token arrives at an arc while another token is still pending. This happens if each node has a token buffer that is only one deep... when the second token arrives, the old one gets overwritten.
Dynamic

Static

\[ x := y := 0 \]
\[ \text{while } x < 10 \]
\[ \text{do } x := x + 1 \]
\[ y := y + 1 + h(x) \]

Two general ways of handling the problem give rise to the classification of datalow machines into

A second token "x=2" can arrive at the input of function "h"

"h" is some arbitrary subgraph

h (y) = \text{some function of } y
Static Dataflow Machines

1. Use locks: ⇒ compound BRANCH & MERGE nodes

- The strict firing rule prevents x from going ahead until y arrives
- However: reduces concurrency
Static Dataflow Machines (II)

or use Acknowledge method: add extra ack arcs from consumer to producer node

/* like handshake protocols inasynckts */

- Allows greater concurrency than lock method, but at cost of at least doubling # of arcs and tokens

⇒ detailed analysis can help reduce # of extra arcs needed
Dynamic Dataflow Machines

- Allow each iteration to be executed in a separate instance of the graph. 2 Ways:

  1. **Code copying**: A new instance of subgraph is created per iteration; need mechanism to direct tokens to right ins.

  2. **Tagged tokens**: Attach a tag to each token saying what iteration it belongs to

    ⇒ Enabling rule: fire when tokens with same tag are present at each of the inputs of the node
Tag Management

- How to generate distinct tags in a distributed environment \( \Rightarrow \) too many bits
- Extra HW complexity, since tags have to be matched... Tags also have storage overhead
- The flexibility may overexpose parallelism, possibly causing deadlocks or storage problems

```plaintext
x := y := 0
while x < 10
do x := x + 1
    y := y + h(x)
end
```
Handling Procedures

- Extra facility required to direct the output token of the procedure to the proper calling site. Done by sending special token containing return node address.

Summary:

Static: simpler, good for pipelining
Dynamic: more concurrency but memory and complexity overhead
Architecture of Dataflow Machines

- composed of several PEs that communicate with each other
- example of PE:

Enabling unit: accepts tokens from ① and stores them at addressed node. If node is enabled: executable packet sent to FU to be processed.

Output tokens, with destination addresses are sent back to EU.
Tagged Machines

When a token arrives in ①, check in mem for tokens to see if all other inputs with the same tag are here. If so, send all tokens to the fdkfU, which combines them with a copy of the node description into an executable packet to be passed on to the FU.
Machine Architecture

- One level: Deliver tokens produced by a functional unit to the enabling unit of the correct PE
  - destination address
  - allocation policy

  /* just like msg passing except dF processors */

  e.g. Arvind

- Two-level: each FU consists of several functional elements which concurrently process executable packet

  e.g. Gurd (Manchester)
Two-Stage: Each Enabling unit can send executable packets to each FU

+ Heterogeneous PEs
+ Scheduling flexib
- More cost
The Manchester Dataflow Machine

- Gurd and Watson; Univ. of Manchester, UK start 76, working 81
- Two-level machine, although only 1 macroprocessor implemented

- Tokens carried in packets around pipelined ring
- Matching unit: pairs together tokens destined for same instruction
- Programs w/ large data set overflow in overflow unit
- Paired tokens fetch the appropriate instruction from the instruction store contains the machine code for the dataflow program
- Instruction + input forwarded to PU

- Link queue buffer to smooth traffic
- I/O switch connection to host

- Token queue: 32k tokens
- 1H tokens
- Overflow unit
Performance

- Poor: 5-10 times slower than VAX-11/780 on RSIM (switch level sim)
- Better technology claim 5X better

Average parallelism = \( AP = \frac{S_i}{S_{\infty}} \rightarrow \text{time steps with 1 FU} \)
\( \rightarrow \text{time steps with } \infty \text{ FU} \)

<table>
<thead>
<tr>
<th>Program</th>
<th>AP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laplace</td>
<td>134-210</td>
</tr>
<tr>
<td>Matmult</td>
<td>236</td>
</tr>
<tr>
<td>Rsim</td>
<td>15-20 { SISAC }</td>
</tr>
<tr>
<td>Simple</td>
<td>63</td>
</tr>
</tbody>
</table>

Found that need \( AP \geq 40 \) to get good speedup
Resource Requirements for Dataflow (Culler, ISCA'88)

- Dataflow exposes all of parallelism present
- Sometimes too much parallelism uses up valuable resources (buffer space, match unit, storage)

Inner product:

\[ \text{sum} = \emptyset \]

for \( (i=1; i \leq n; i++) \)

\[ \text{sum} += A[i] \times B[i] \]

\( \rightarrow \) concurrent ops for \( \infty \) PEs

\( \rightarrow \) total tokens in systems

waiting token profile

(tokens waiting for partners)
Bigger Example: Matrix Multiply

- Outer iterations all unfold immediately resulting in $n^2$ parallelism
- Innermost loop tokens unfold in a staggered fashion producing spikes
Simple

- Massive # of waiting tokens correspond to the deferred reads created for all 8 iterations for all grid points.
- Concurrency has 0 points as there is a global dependency.

Figure 4: ($p = \infty, l = 0$) Parallelism Profile and Token Profile for Simple, 8 iterations on 16x16 mesh.
Gaussian Elimination

- This program is similar to SIMPLE
- Shows how subtle change in pm structure can make big difference

Figure 5: \( p = \infty, t = 0 \) Parallelism Profile and Token Profile for Gaussian Relaxation, 4 iterations on a 10x10 grid.
Solution

- Constrain exposed parallelism by bounding the unfolding of loops in a flexible manner.

Figure 7: \( p = 50, 1 = 0 \) Profiles for 16x16 Matrix Multiply with Middle Loop Bound to two

Matrix multiply with

- 50 functional units
- allow only 2 outstanding iterations of middle loop

\( \exists \)

need HW mechanism
Evaluating Traditional Dataflow

- Benefits of dataflow model:
  - explodes parallelism
  - has mechanisms to tolerate latency (lazy evaluation)
  - provides mechanisms for fine-grain synch (tag-matching, I-structures)

- Drawbacks:
  - Not using implicit sequence within blocks results in large latencies through critical paths (Amdahl's law) e.g. destination address of tokens
  - Loss of locality due to massive concurrency and interleaving of instructions (high BW and low performance)
  - Too much exposed parallelism can result in waste of resources (memory for tokens, buffers, tags, ...)
  - Use of functional languages to expose concurrency makes garbage collection and copying serious problems
  - Overhead for providing capability for fine-grain synchronization are large (matching unit and F/E bits)
Synthesis of von-Neuman and Dataflow

- Basic requirements for architecture: *(embodied in P-RISC)*
  - Support for split-phase memory transactions and multi-threaded PEs to tolerate latency (multiple outstanding requests, out-of-order responses)

- Support for large synchronization name space and fine-grain synchronization mechanisms (e.g., l-structures, join instruction)

- Support for cheap forking of tasks
P-RISC Multiprocessor

- Large number of PEs connected by a scalable network

- Single address space but physically distributed memory. Plus,
  - PEs have local memory for code and frames
  - global heap or l-structure memory

- At each instant PE runs a thread that is completely defined by its continuation <FP.IP>.

Figure 3: P-RISC multiprocessor organization

Figure 4: A continuation
Development of P-RISC

- Start with basic RISC core
  - Add multithreading
    - helps hide latency
  
  - Loads are split phase. Request and response messages are identified by full continuation, so that:
    
    - responses may return in any order
  
  - Fork and Join instructions provide fine-grain creation and synchronization of parallel threads on same node

  - Start instruction provides mechanism to communicate and start computation on remote nodes

  - Heap memory with I-structures for fine-grain global synchronization
Some Issues and Summary

- What is the common case: finding data close by or far away?
  - dataflow optimized for data being far away
  - our experience is that data likely to be close by (in cache)

- However, the models are converging (e.g., active messages as basis)

  O00 SUPERSCALAR