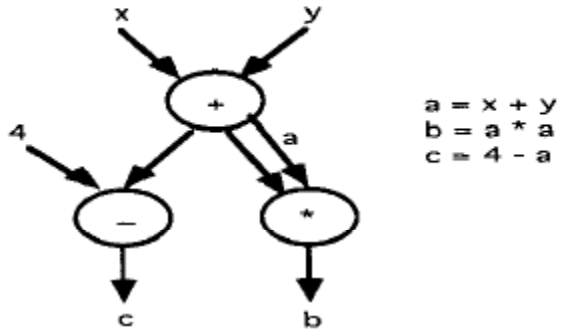
Dataflow

Instructor: Josep Torrellas

CS533

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 40 years, little impact on mainstream computing
  - However, ideas used in ooo superscalars

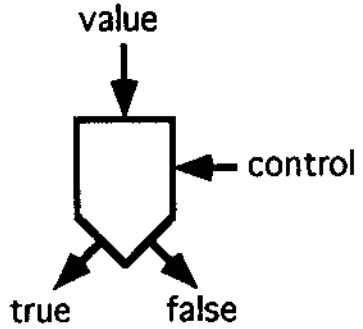
Terminology

- 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.

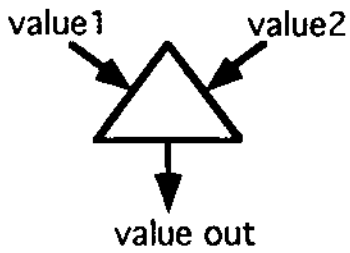
Node Types

- **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.**



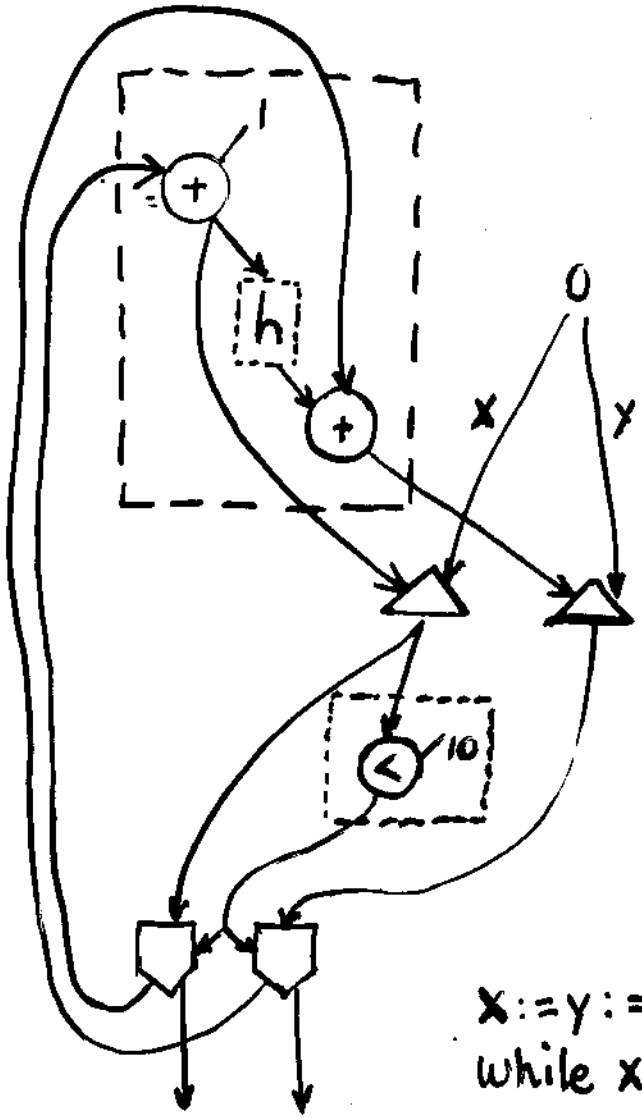
- **Merge: Whenever there is a token on any of the input arcs, it is immediately copied to the output arc.**



- non-strict firing rule
- acts as serializer/merger

Iteration, Recursion, Reuse and Reentrancy

- using the same graph to perform comput. on diff. 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

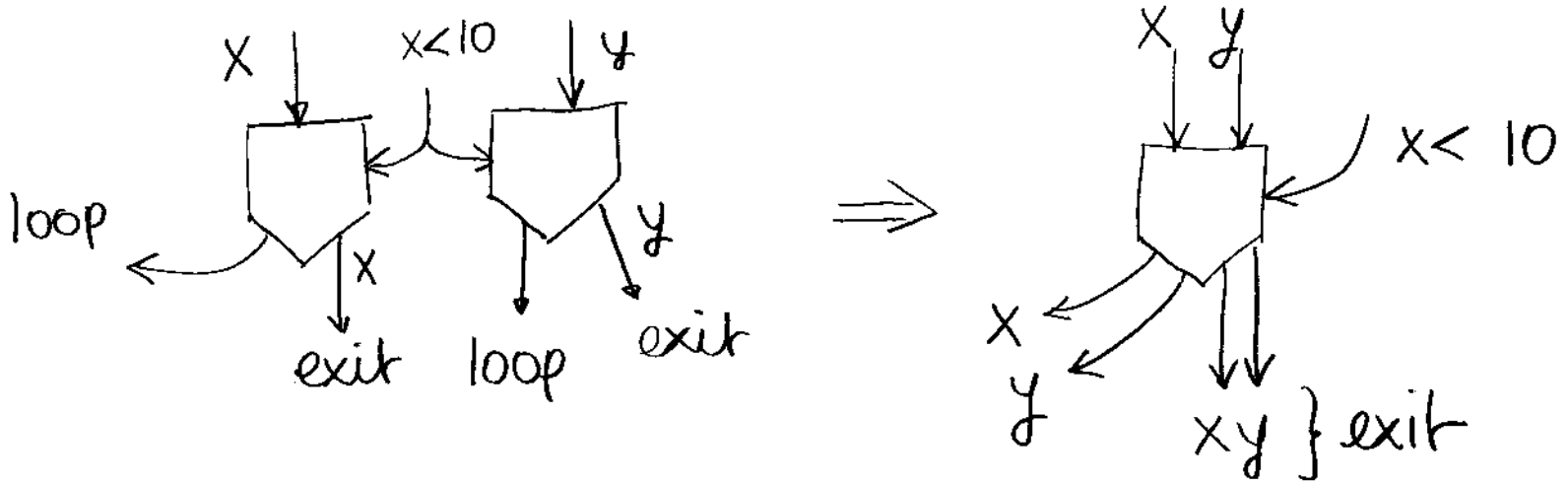



```
x:=y:=0
while x< 10
do x:=x+1
   y:=y+ h(x)
od
```

- "h" is some arbitrary subgraph
- A second token "x=2" can arrive at the input of function "h" if it takes a long time to compute it, compared to the ⊕ node.
- Two general ways of handling the problem give rise to the classification of dataflow machines into
  - Static
  - Dynamic

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 in asynckts */

- 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

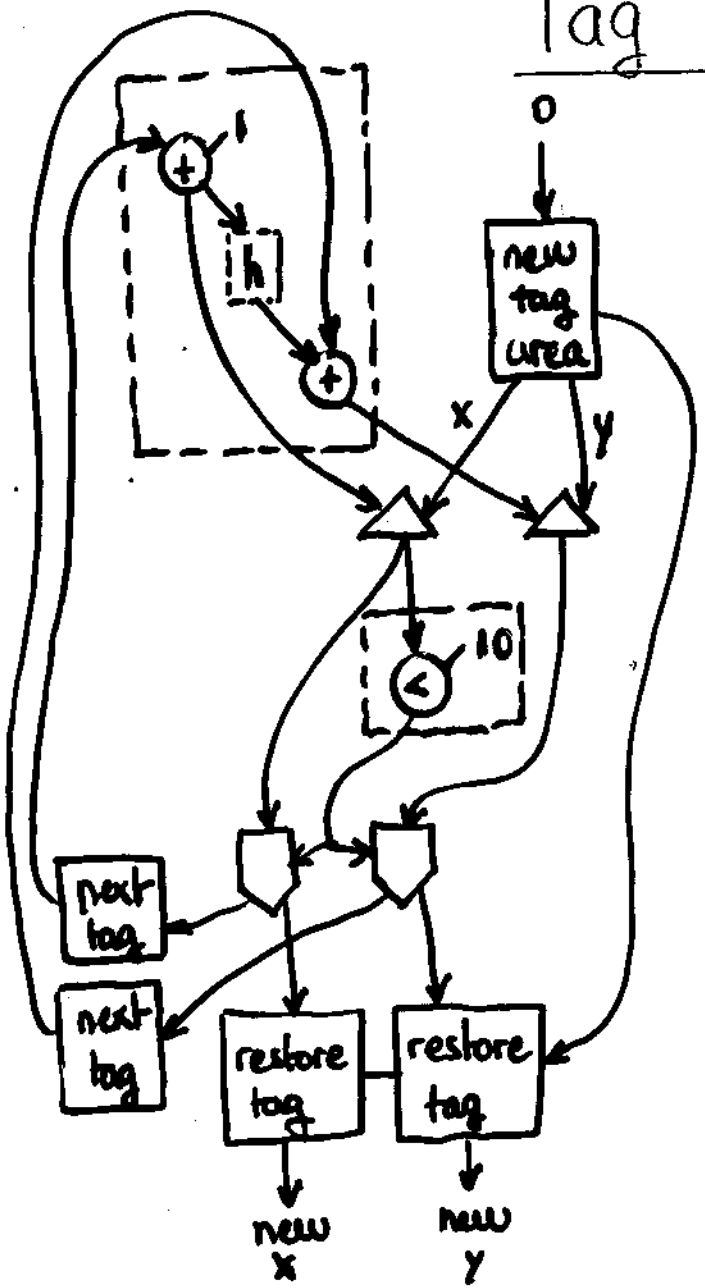
- Allow each iteration to be executed in a separate instance of the graph. 2 ways:

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

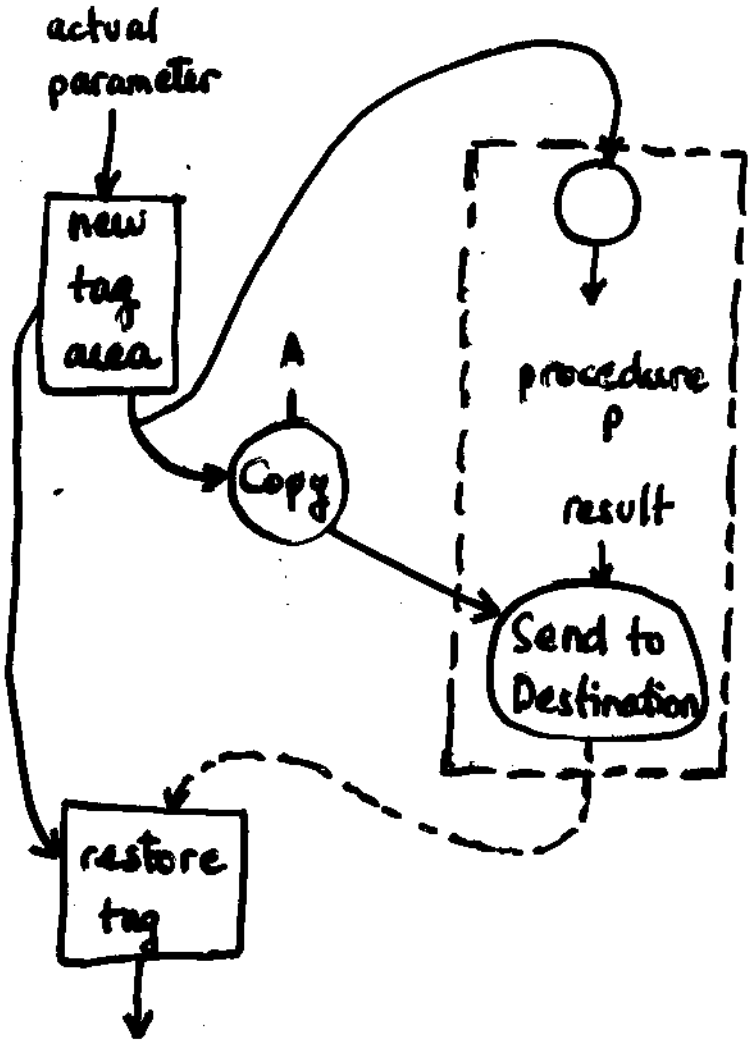
Tag Management



- How to generate distinct tags in a distributed environment (⇒ 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 pbms

```
x:=y:=0
while x<10
do x:=x+1
   y:=y+h(x)
od
```

Handling Procedures



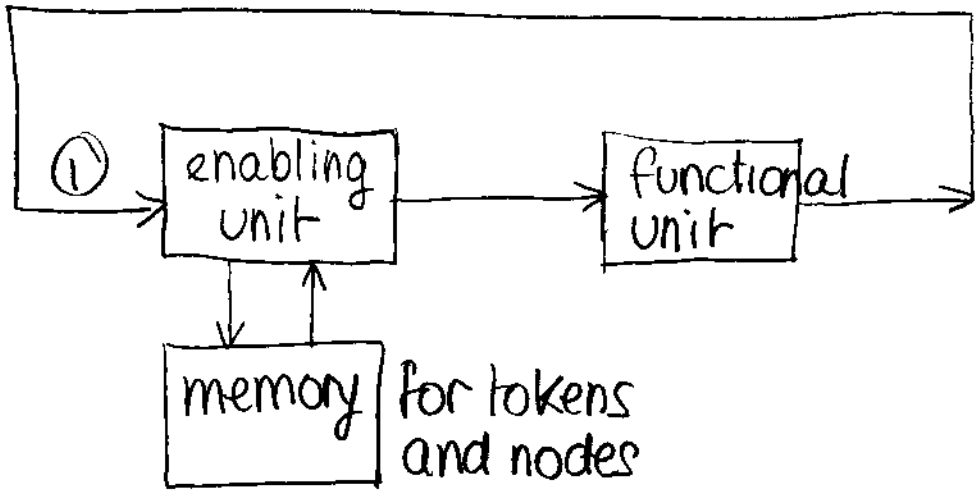
- Extra facility required to direct the output token of the procedure to the proper calling site ⇒ done by sending special token containing return node address

Summary:

Static: simpler, good for pipelnig
Dynamic: more concurrency but memory and complxity ovhd

Architecture of Dataflow Machines

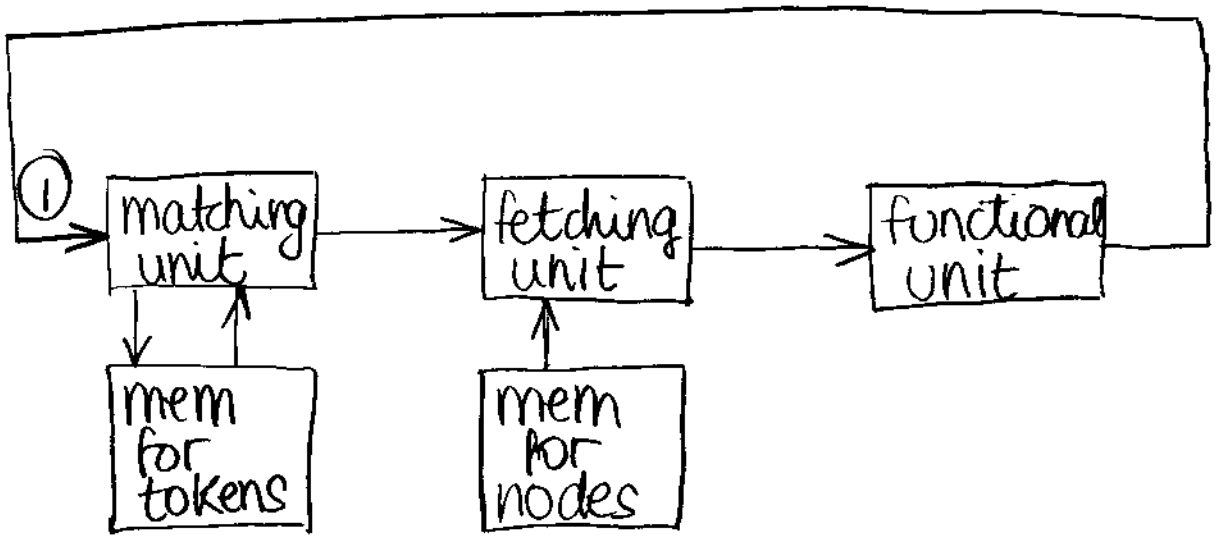
- composed of several PEs that communicate with each other
- example of PE:



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

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 fetchU. which combines them with a copy of the node desciption 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 procesors */

  e.g. Arvind

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

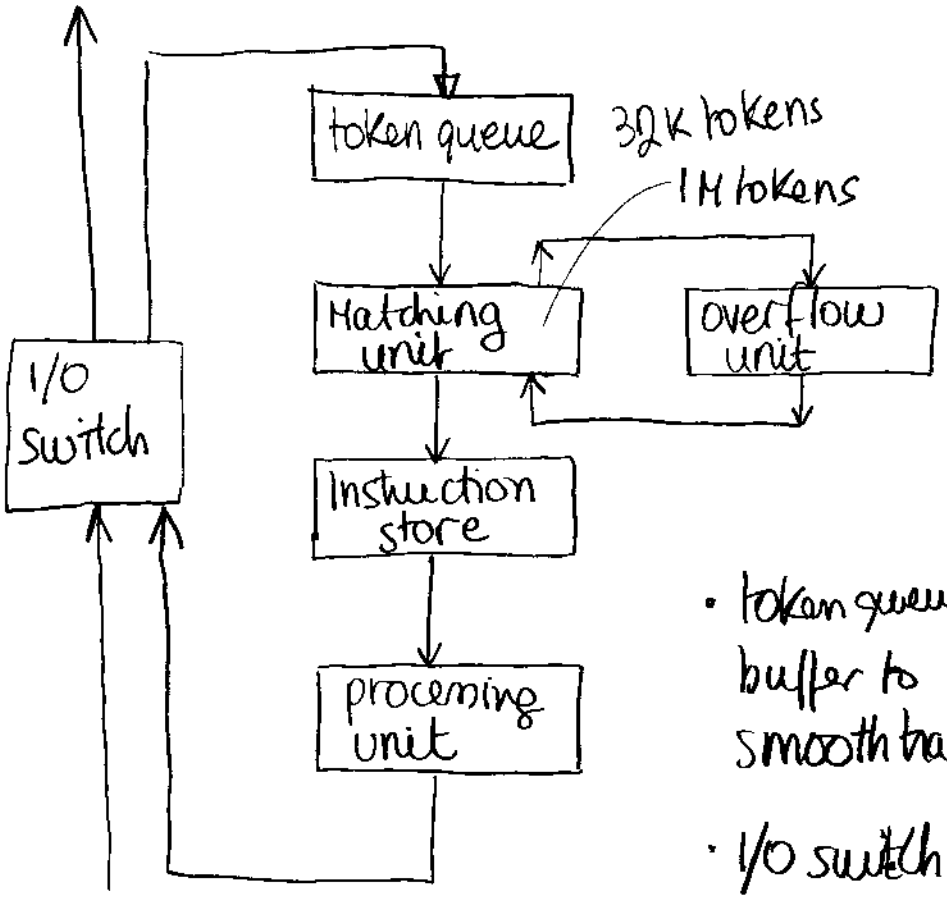
  e.g. Gurd (Manchester)

- 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



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

- 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 appropiate instruction from the instruction store contains the machine code for the data/flow program
- instruction + inputs forwarded to PU

Performance

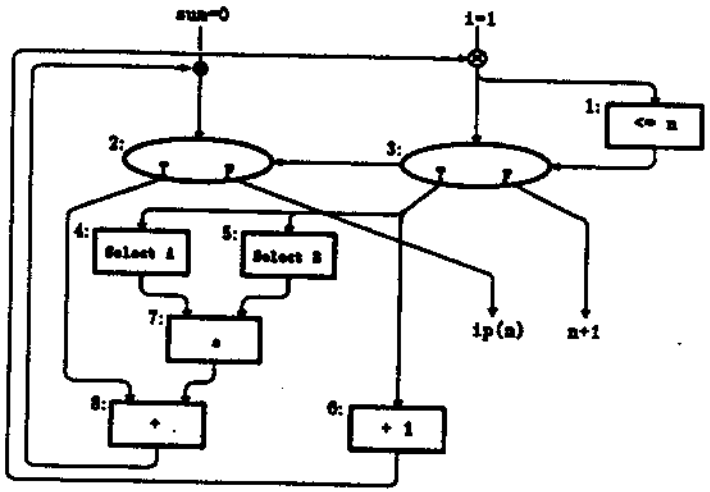
- Poor : 5-10 times slower than VAX-11/780 on RSIM (switch level SIM
- better technology : claim 5X better
- average parallelism = AP = $\frac{S_1}{S_\infty}$ → time steps with 1 FU / → time steps with ∞ FU

Program	AP	
Laplace	134-210	
Matmult	236	
Rsim	15-20	} SISAL
Simple	63	} SISAL

- found that need AP $\geq$ 40 to get good speedups

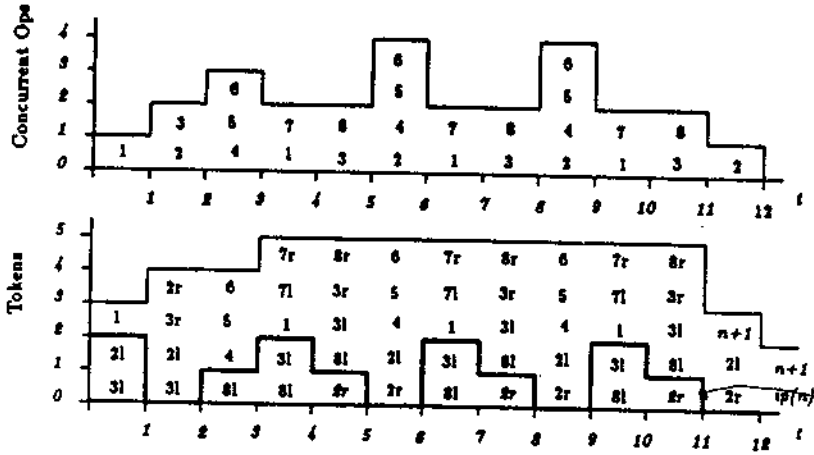
Resource Requirements for Dataflow (Culler, ISCA'88)

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



Inner product:

```
sum = ∅
for (i=1; i≤n; i++)
    sum += A[i]*B[i]
```



→ concurrent ops for ∞ PEs

→ total tokens in systems

→ waiting token profiles (tokens waiting for partners)

Bigger Example: Matrix Multiply

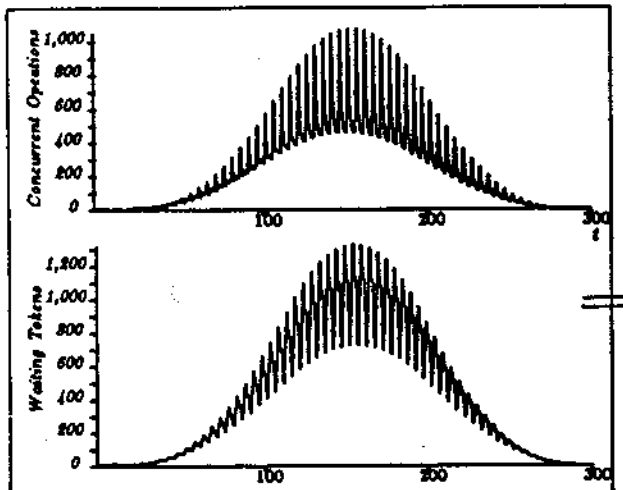


Figure 3: ($p = \infty, l = 0$) Parallelism Profile and Token Profile for 16x16 Matrix Multiply

- Outer iterations all unfold immediately resulting in n^2 parallelism

- Innermost loop tokens unfold in a staggered fashion producing spikes

SIMPLE

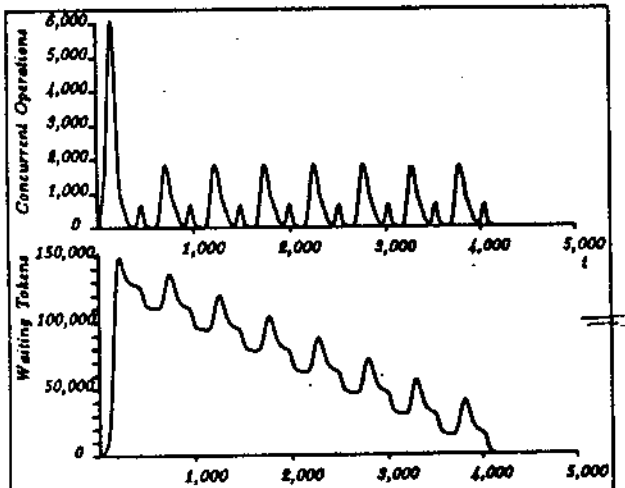


Figure 4: ($p = \infty, l = 0$) Parallelism Profile and Token Profile for Simple, 8 iterations on 16x16 mesh.

- Massive # of waiting tokens correspond to the deferred reads created for all 8 iterations for all grid points

- Concurrency has ∅ points as there is a global dependency

Gaussian Elimination

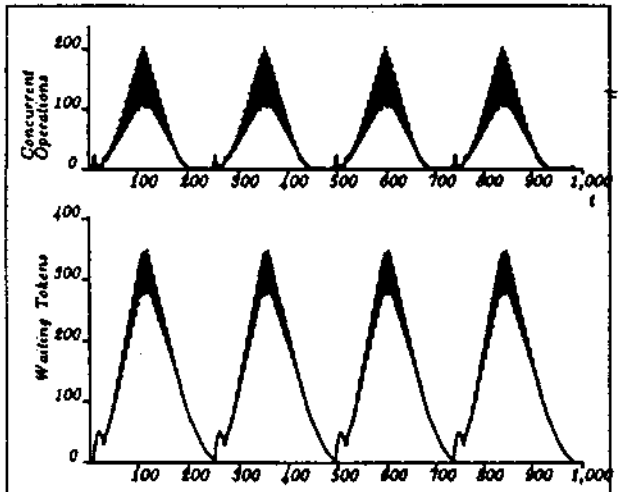


Figure 5: ($p = \infty, l = 0$) Parallelism Profile and Token Profile for Gaussian Relaxation, 4 iterations on a 16x16 grid

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

Solution

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

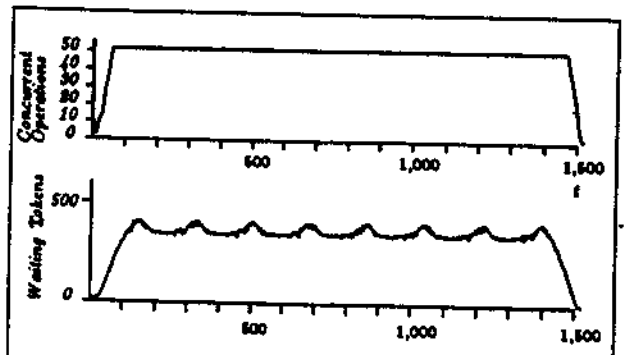


Figure 7: ($p = 50, l = 0$) Profiles for 16x16 Matrix Multiply with Middle Loop Bound to two

Matrix multiply with

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

need HW mechanism

Evaluating Traditional Dataflow

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

- Drawbacks:
  - Not using implicit sequence within basic 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)