

Abstractions for algorithms and parallel machines

Keshav Pingali

University of Texas, Austin

High-level idea

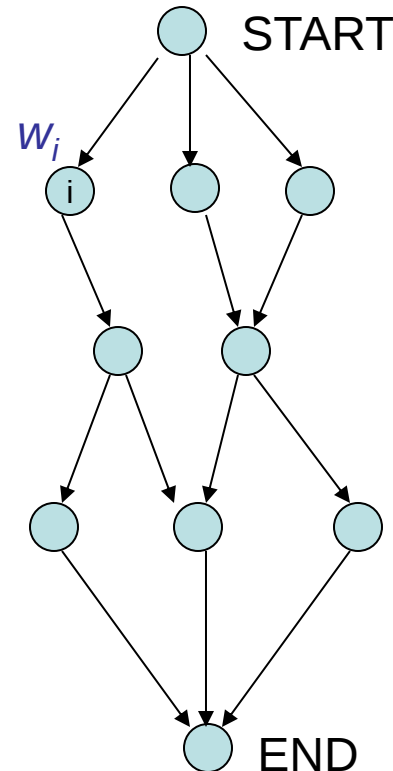
- Difficult to work directly with textual programs
 - Where is the parallelism in the program?
 - Solution: use an abstraction of the program that highlights opportunities for exploiting parallelism
 - What program abstractions are useful?
- Difficult to work directly with a parallel machine
 - Solution: use an abstraction of the machine that exposes features that you want to exploit and hides features you cannot or do not want to exploit
 - What machine abstractions are useful?

Abstractions introduced in lecture

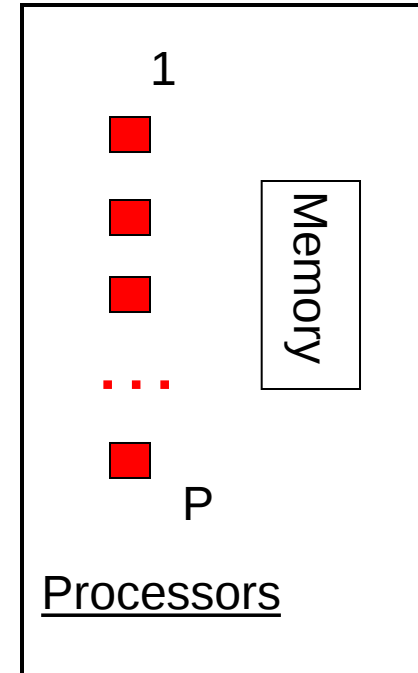
- Program abstraction: computation graph
 - nodes are computations
 - granularity of nodes can range from single operators (+,*,etc.) to arbitrarily large computations
 - edges are precedence constraints of some kind
 - edge $a \rightarrow b$ may mean computation a must be performed before computation b
 - many variations in the literature
 - imperative languages community:
 - data-dependence graphs, program dependence graphs
 - functional languages community
 - dataflow graphs
- Machine abstraction: PRAM
 - parallel RAM model
 - exposes parallelism
 - hides synchronization and communication

Computation DAG's

- DAG with START and END nodes
 - all nodes reachable from START
 - END reachable from all nodes
 - START and END are not essential
- Nodes are computations
 - each computation can be executed by a processor in some number of time-steps
 - computation may require reading/writing shared-memory
 - node weight: time taken by a processor to perform that computation
 - w_i is weight of node i
- Edges are precedence constraints
 - nodes other than START can be executed only after immediate predecessors in graph have been executed
 - known as **dependences**
- Very old model:
 - PERT charts (late 50's):
 - Program Evaluation and Review Technique
 - developed by US Navy to manage Polaris submarine contracts

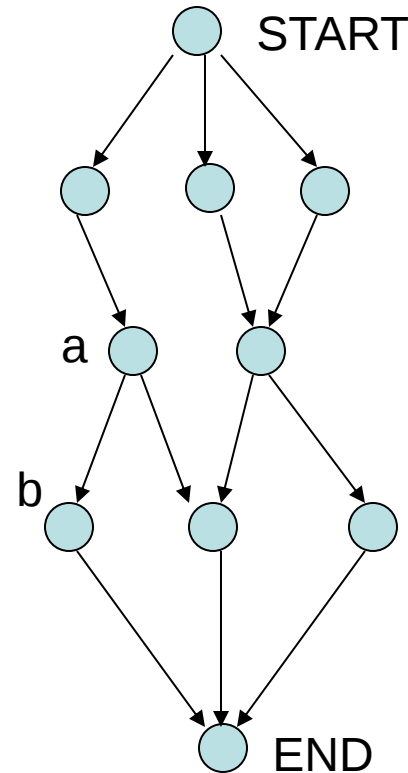


Computation DAG

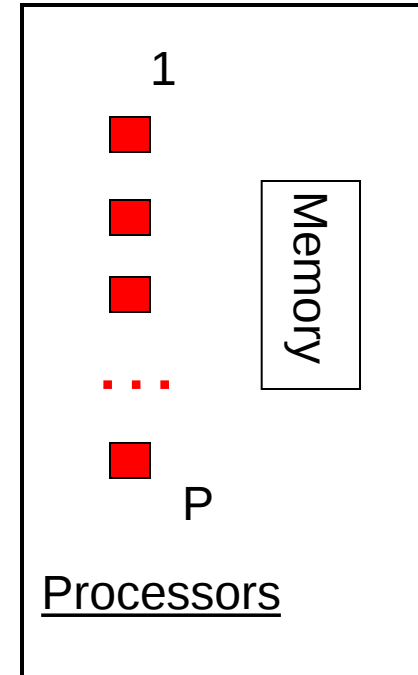


Computer model

- P identical processors
- Memory
 - processors have local memory
 - all shared-data is stored in global memory
- How does a processor know which nodes it must execute?
 - work assignment
- How does a processor know when it is safe to execute a node?
 - (eg) if P1 executes node a and P2 executes node b, how does P2 know when P1 is done?
 - synchronization
- For now, let us defer these questions
- In general, time to execute program depends on work assignment
 - for now, assume only that if there is an idle processor and a ready node, that node is assigned immediately to an idle processor
- T_P = best possible time to execute program on P processors

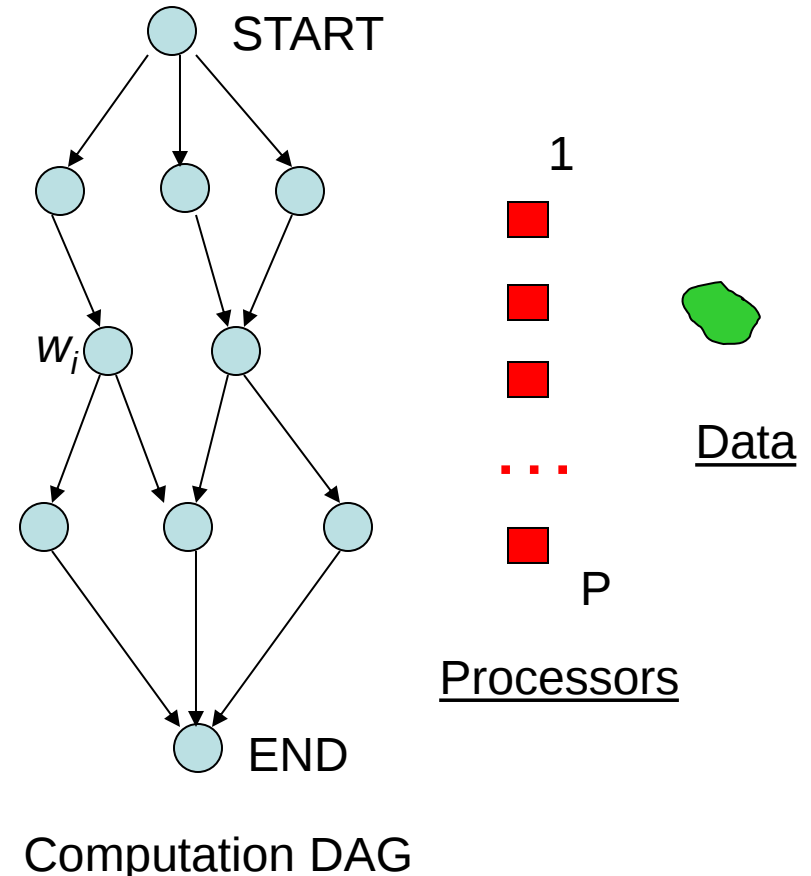


Computation DAG



Work and critical path

- **Work** = $\sum_i w_i$
 - time required to execute program on one processor
= T_1
- **Path weight**
 - sum of weights of nodes on path
- **Critical path**
 - path from START to END that has maximal weight
 - this work must be done sequentially, so you need this much time regardless of how many processors you have
 - call this T_∞



Unbounded number of processors

- Instantaneous parallelism

$IP(t)$ = maximum number of processors that can be kept busy at each point in execution of algorithm

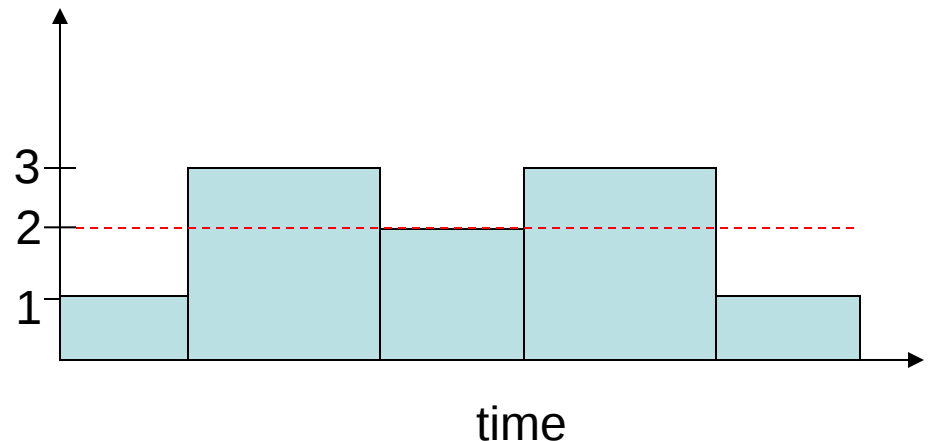
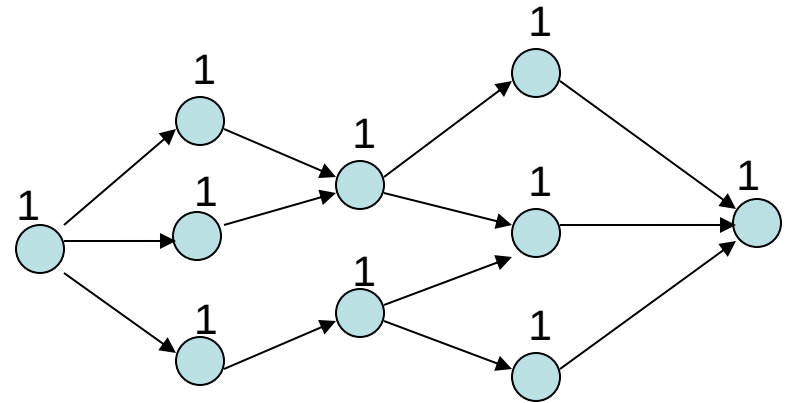
- Maximal parallelism

MP = highest instantaneous parallelism

- Average parallelism

$$AP = T_1 / T_\infty$$

- These are properties of the computation DAG, not of the machine or the work assignment



Instantaneous and average parallelism

Computing critical path etc.

- Algorithm for computing earliest start times of nodes
 - Keep a value called minimum-start-time (mst) with each node, initialized to 0
 - Do a topological sort of the DAG
 - ignoring node weights
 - For each node n ($\neq$ START) in topological order
 - for each node p in predecessors(n)
 - $mst_n = \max(mst_n, mst_p + w_p)$
- Complexity = $O(|V|+|E|)$
- Critical path and instantaneous, maximal and average parallelism can easily be computed from this

Speed-up

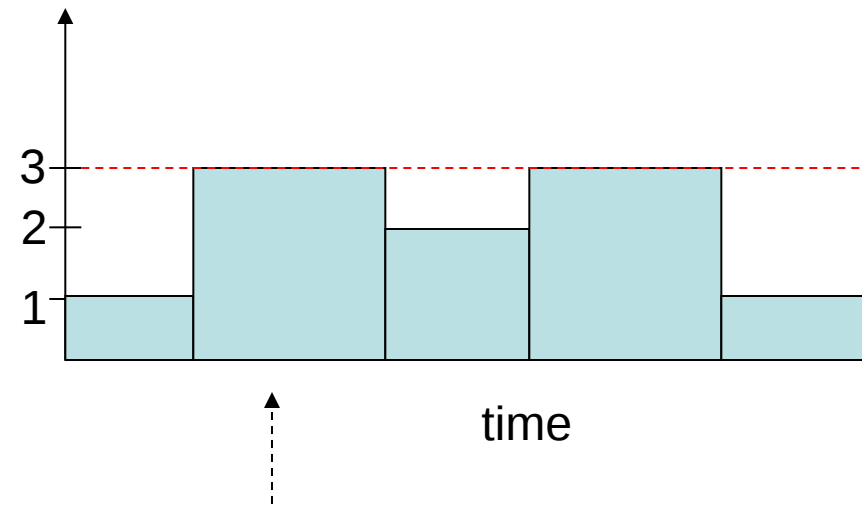
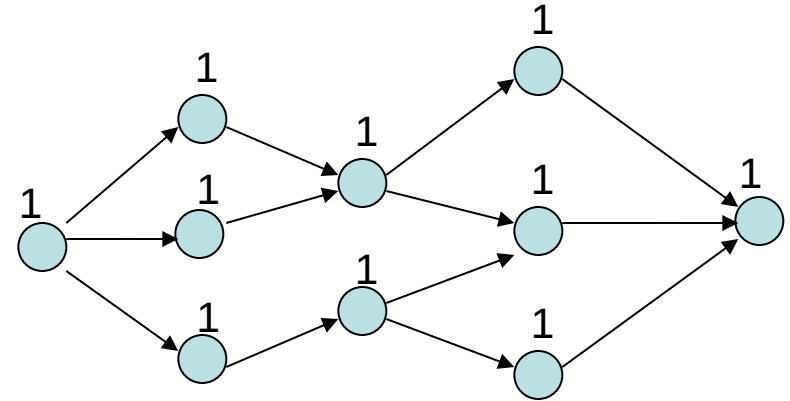
- $\text{Speed-up}(P) = T_1/T_P$
 - intuitively, how much faster is it to execute program on P processors than on 1 processor?
- **Bound on speed-up**
 - regardless of how many processors you have, you need at least T_∞ units of time
 - $\text{speed-up}(P) \leq T_1/T_\infty = \sum_i w_i / CP = AP$

Amdahl's law

- Amdahl:
 - suppose a fraction p of a program can be done in parallel
 - suppose you have an unbounded number of parallel processors and they operate infinitely fast
 - speed-up will be at most $1/(1-p)$.
- Follows trivially from previous result.
- Plug in some numbers:
 - $p = 90\% \rightarrow \text{speed-up} \leq 10$
 - $p = 99\% \rightarrow \text{speed-up} \leq 100$
- To obtain significant speed-up, most of the program must be performed in parallel
 - serial bottlenecks can really hurt you

Scheduling on finite number of processors

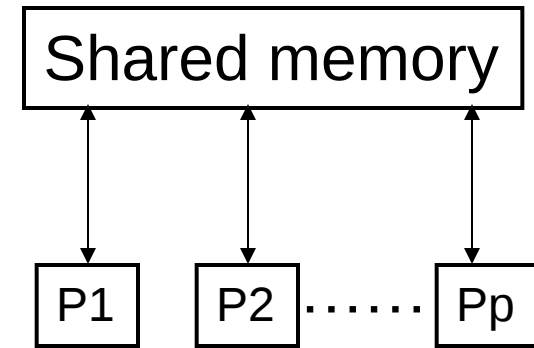
- Suppose $P \leq MP$ (more work than cores)
- There will be times during the execution when only a subset of “ready” nodes can be executed.
- Time to execute DAG can depend on which subset of P nodes is chosen for execution.
- To understand this better, it is useful to have a more formal model of the machine



What if we only had 2 processors?

PRAM Model

- Parallel Random Access Machine (PRAM)
- Natural extension of RAM model
- Processors operate synchronously (in lock-step)
 - synchronization in architecture
- Each processor has private memory



Details

- A PRAM step has three phases
 - read: each processor can read a value from shared-memory
 - compute: each processor can perform a computation on local values
 - write: each processor can write a value to shared-memory
- Variations:
 - Exclusive read, exclusive write (EREW)
 - a location can be read or written by only one processor in each step
 - Concurrent read, exclusive write (CREW)
 - Concurrent read, concurrent write (CRCW)
 - some protocol for deciding result of concurrent writes
- We will use the CREW variation
 - assume that computation graph ensures exclusive writes

Schedules

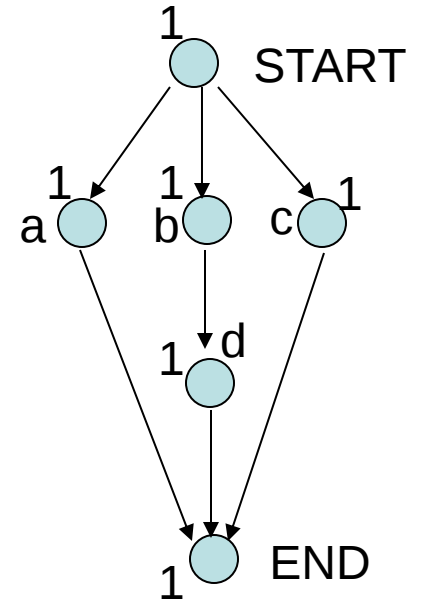
Schedule: function from node to (processor, start time)
Also known as “space-time mapping”

Schedule 1

		time →				
space ↓		0	1	2	3	4
	P0	START	a	c	END	
	P1		b	d		

Schedule 2

		time →				
space ↓		0	1	2	3	4
	P0	START	a	b	d	END
	P1		c			



■ P0

■ P1

Intuition: nodes along the critical path should be given preference in scheduling

Optimal schedules

- Optimal schedule
 - shortest possible schedule for a given DAG and the given number of processors
- Complexity of finding optimal schedules
 - one of the most studied problems in CS
- DAG is a tree:
 - level-by-level schedule is optimal (Aho, Hopcroft)
- General DAGs
 - variable number of processors (number of processors is input to problem): NP-complete
 - fixed number of processors
 - 2 processors: polynomial time algorithm
 - 3,4,5...: complexity is unknown!
- Many heuristics available in the literature

Heuristic: list scheduling

- Maintain a list of nodes that are ready to execute
 - all predecessor nodes have completed execution
- Fill in the schedule cycle-by-cycle
 - in each cycle, choose nodes from ready list
 - use heuristics to choose “best” nodes in case you cannot schedule all the ready nodes
- One popular heuristic:
 - assign node priorities before scheduling
 - priority of node n :
 - weight of maximal weight path from n to END
 - intuitively, the “further” a node is from END, the higher its priority

List scheduling algorithm

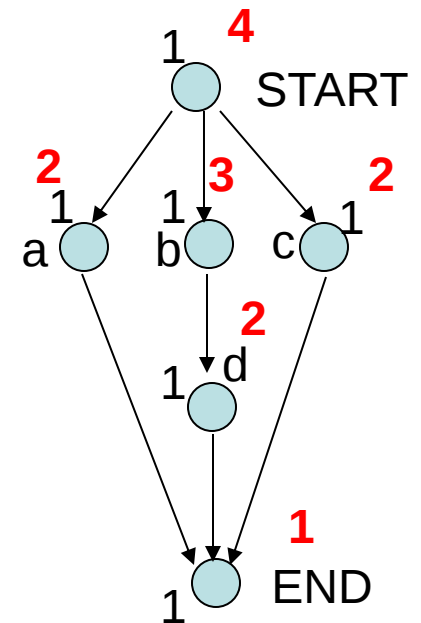
```
cycle c = 0;
ready-list = {START};
inflight-list = { };
while (|ready-list|+|inflight-list| > 0) {
    for each node n in ready-list in priority order {
        if (a processor is free at this cycle) {
            remove n from ready-list and add to inflight-list;
            add node to schedule at time cycle;
        }
        else break;
    }
    c = c + 1; //increment time
    for each node n in inflight-list {
        if (n finishes at time cycle) {
            remove n from inflight-list;
            add every ready successor of n in DAG to ready-list
        }
    }
}
```

Example

		time →				
space ↓		0	1	2	3	4
	P0	START	a	c	END	
	P1		b	d		

Heuristic picks the good schedule

Not always guaranteed to produce optimal schedule
(otherwise we would have a polynomial time algorithm!)



■ P0

■ P1

Applying scheduling theory in practice

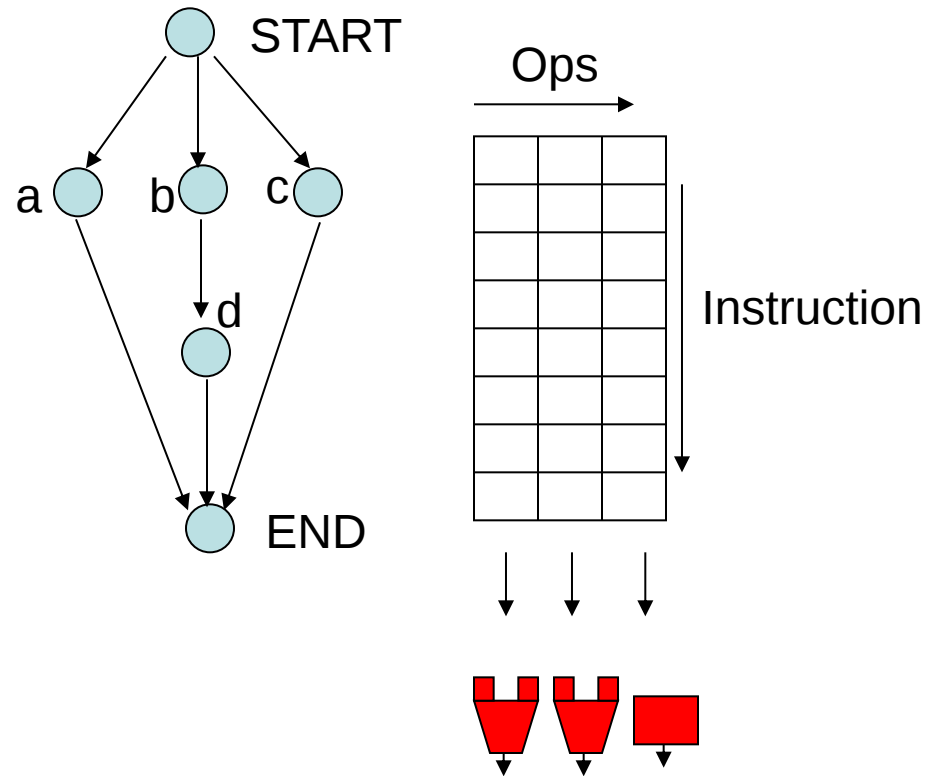
- What should a node be?
 - fine-grain: operation like +, *, ...
 - coarse-grain: single loop iteration
 - very coarse-grain: outer loop iteration
 - ...
- How do we determine the edges between nodes in DAG?
 - make user specify them
 - let compiler deduce them from sequential program
 -
- How do we determine how long each node takes to execute?
 - ask user to tell us
 - use a model
 - profiling
 -
- **Binding time:**
 - when do we know this information?
 - consider two applications
 - VLIW scheduling: information is known at compile-time
 - Multicore scheduling: node + edges known statically, node execution time known only at runtime

Compile-time scheduling: VLIW machines

- Processors → functional units
- Local memories → registers
- Global memory → memory
- Time → instruction

DAG scheduling:

- Nodes in DAG are basic block operations (load/store/add/mul/..)
 - **instruction-level** parallelism
- Edges: determined by compiler
- Execution time of operation
 - known except for loads



Increasing basic block size

- Basic blocks are fairly small
 - about 5 RISC operations on the average
- Many solutions for increasing scheduling scope
 - loop unrolling
 - trace scheduling: move operations past branches
 - predicated execution
 -
- DAG scheduling is used extensively in compilers for pipelines, superscalar and VLIW machines

Historical note on VLIW processors

- Ideas originated in late 70's-early 80's
- Two key people:
 - Bob Rau (Stanford, UIUC, TRW, Cydrome, HP)
 - Josh Fisher (NYU, Yale, Multiflow, HP)
- Bob Rau's contributions:
 - transformations for making basic blocks larger:
 - predication
 - software pipelining
 - hardware support for these techniques
 - predicated execution
 - rotating register files
 - most of these ideas were later incorporated into the Intel Itanium processor
- Josh Fisher:
 - transformations for making basic blocks larger:
 - trace scheduling: uses key idea of **branch probabilities**
 - Multiflow compiler used loop unrolling



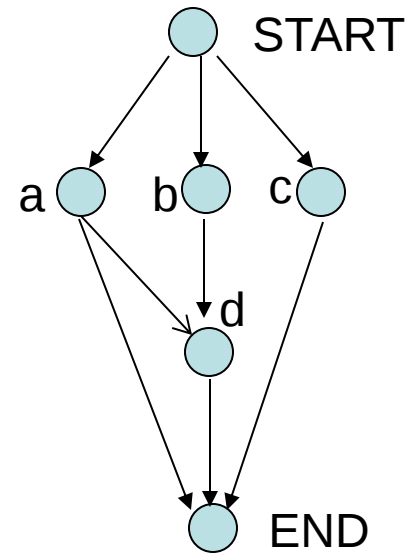
Bob Rau



Josh Fisher

DAG scheduling for multicores

- Reality:
 - hard to build single cycle memory that can be accessed by large numbers of cores
- Architectural change
 - decouple cores so there is no notion of a global step
 - each core/processor has its own PC and cache
 - memory is accessed independently by each core
- New problem:
 - since cores do not operate in lock-step, how does a core know when it is safe to execute a node?
- Solution: software synchronization
 - one solution: flag associated with each edge
 - written by processor that executes source of edge
 - read by processor that executes destination of edge
- Software synchronization increases overhead of parallel execution
 - ➔ cannot afford to synchronize at the instruction level
 - ➔ nodes of DAG must be coarse-grain: loop iterations



P0: a
P1: b
P2: c d

How does P2 know when
P0 and P1 are done?

Increasing granularity: Block Matrix Algorithms

Original matrix multiplication

```
for I = 1,N
  for J = 1,N
    for K = 1,N
      C(I,J) = C(I,J) + A(I,K)*B(K,J)
```

Block (tiled) matrix multiplication

```
for IB = 1,N step B
  for JB = 1,N step B
    for KB = 1,N step B
      for I = IB, IB+B-1
        for J = JB, JB+B-1
          for K = KB, KB+B-1
            C(I,J) = C(I,J) + A(I,K)*B(K,J)
```

parallel loops

B_{00}	B_{01}
B_{10}	B_{11}

A_{00}	A_{01}	C_{00}	C_{01}
A_{10}	A_{11}	C_{10}	C_{11}

$$\begin{aligned} C_{00} &= A_{00} * B_{00} + A_{01} * B_{10} \\ C_{01} &= A_{01} * B_{11} + A_{00} * B_{01} \\ C_{11} &= A_{11} * B_{01} + A_{10} * B_{01} \\ C_{10} &= A_{10} * B_{00} + A_{11} * B_{10} \end{aligned}$$

New problem

- Difficult to get accurate execution times of coarse-grain nodes
 - conditional inside loop iteration
 - cache misses
 - exceptions
 - O/S processes
 -
- Solution: runtime scheduling

Example: DAGuE

- Dongarra et al (UTK)
- Programming model for specifying DAGs for parallel tiled dense linear algebra codes
 - nodes: tiled computations
 - DAG edges specified by programmer (see next slides)
- Runtime system
 - keeps track of ready nodes
 - assigns ready nodes to cores
 - determines if new nodes become ready when a node completes

DAGuE: Tiled QR (1)

```
FOR k = 0 .. SIZE-1
  A[k][k], T[k][k] <- DGEQRT(A[k][k])
  FOR m = k+1 .. SIZE-1
    A[k][k], A[m][k], T[m][k] <-
      DTSQRT(A[k][k], A[m][k], T[m][k])
  FOR n = k+1 .. SIZE-1
    A[k][n] <- DORMQR(A[k][k], T[k][k], A[k][n])
    FOR m = k+1 .. SIZE-1
      A[k][n], A[m][n] <-
        DSSMQR(A[m][k], T[m][k], A[k][n], A[m][n])
```

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$	$A_{0,5}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	$A_{1,5}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	$A_{2,5}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$	$A_{3,5}$
$A_{4,0}$	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$	$A_{4,5}$
$A_{5,0}$	$A_{5,1}$	$A_{5,2}$	$A_{5,3}$	$A_{5,4}$	$A_{5,5}$

Tiled QR (using tiles and in/out notations)

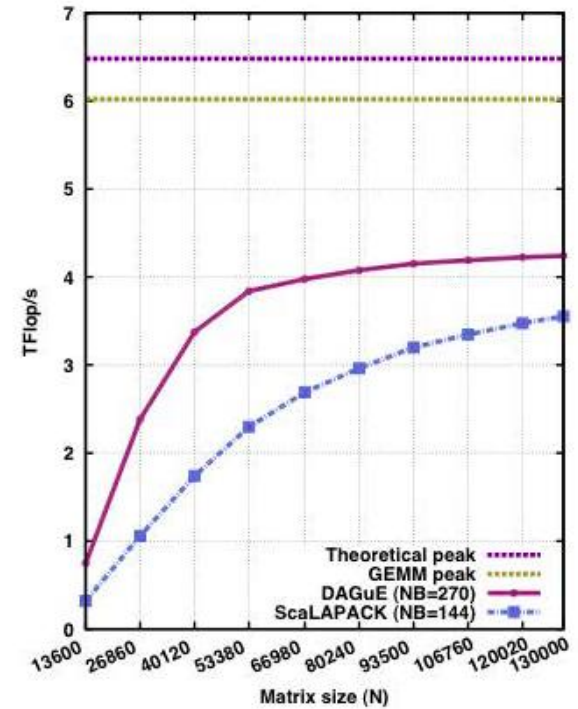
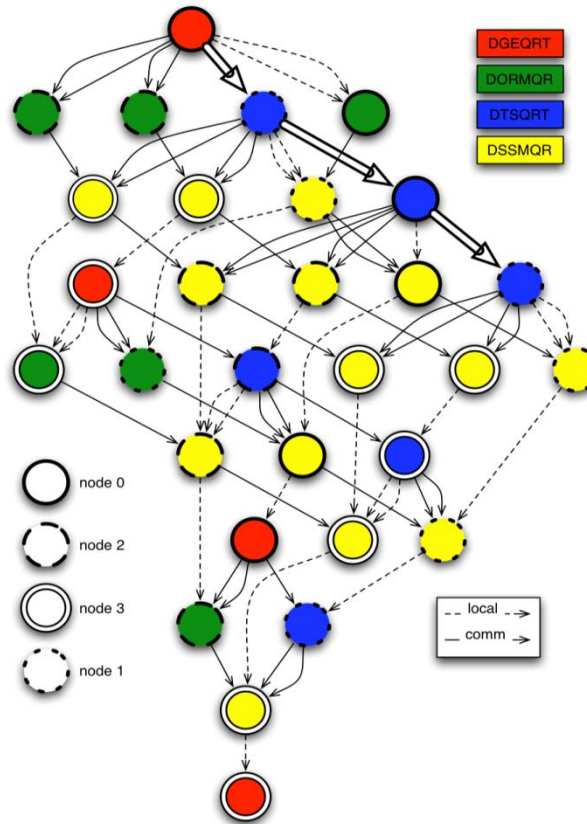
DAGuE: Tiled QR (2)

```

FOR k = 0 .. SIZE-1
  A[k][k], T[k][k] <- DGEQRT(A[k][k])
  FOR m = k+1 .. SIZE-1
    A[k][k], A[m][k], T[m][k] <-
      DTSQRT(A[k][k], A[m][k], T[m][k])
  FOR n = k+1 .. SIZE-1
    A[k][n] <- DORMQR(A[k][k], T[k][k], A[k][n])
    FOR m = k+1 .. SIZE-1
      A[k][n], A[m][n] <-
        DSSMQR(A[m][k], T[m][k], A[k][n], A[m][n])

```

Tiled QR



(b) QR factorization.

Dataflow Graph for 2x2 processor grid Machine: 81 nodes, 648 cores

Summary of DAG scheduling

- DAG:
 - Nodes are computations
 - Edges are dependences
 - Nodes and edges may have associated time
 - node: how long to execute
 - edge: communication time
- Basic algorithm: list scheduling based on priority
- Binding time: when do you know the DAG?
 - VLIW: fine-grain, so known at compile-time
 - Multicore: coarse-grain, so accurate execution time of node is known only at runtime

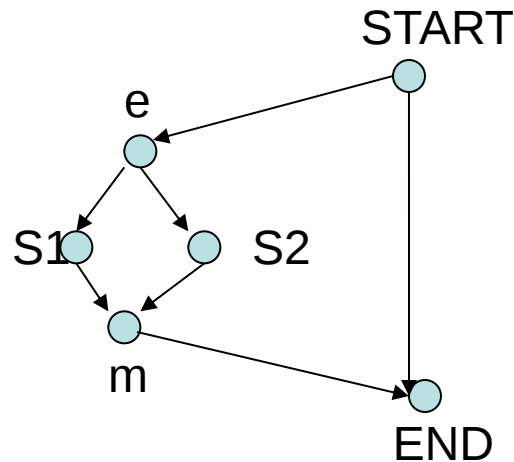
Variations of dependence graphs

Program dependence graph

- Program dependence graphs (PDGs) (Ferrante, Ottenstein, Warren)
 - data dependences + control dependences
- Intuition for control dependence
 - statement s is control-dependent on statement p if the execution of p determines whether s is executed
 - (eg) statements in the two branches of a conditional are control-dependent on the predicate
- Control dependence is a subtle concept
 - formalizing the notion requires the concept of postdominance in control-flow graphs

Control dependence

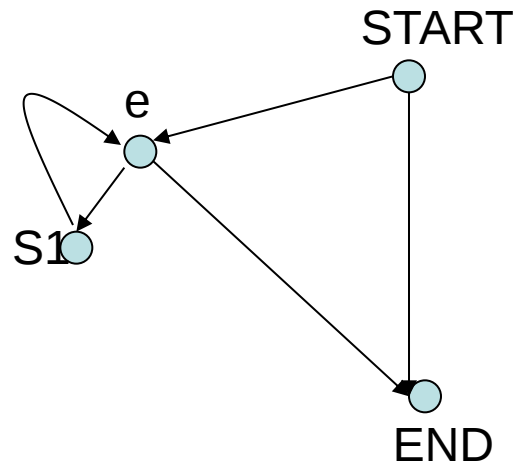
- Intuitive idea:
 - node w is control-dependent on a node u if node u determines whether w is executed
- Example:



```
START
.....
if e then S1 else S2
....
END
```

We would say S1 and S2 are control-dependent on e

Examples (contd.)



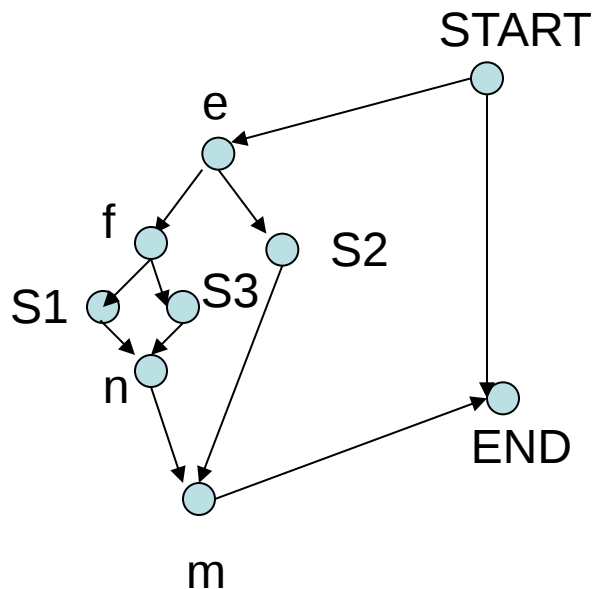
```
START
.....
while e do S1;
....
END
```

We would say node S1 is control-dependent on e.

It is also intuitive to say node e is control-dependent on itself:

- execution of node e determines whether or not e is executed again.

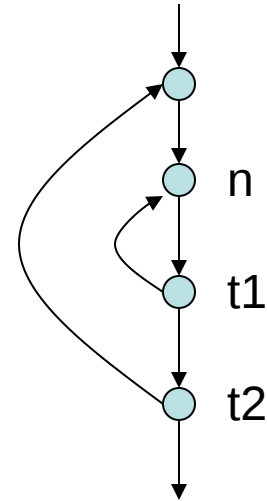
Example (contd.)



- S1 and S3 are control-dependent on f
- Are they control-dependent on e?
- Decision at e does not fully determine if S1 (or S3 is executed) since there is a later test that determines this
- So we will NOT say that S1 and S3 are control-dependent on e
 - Intuition: control-dependence is about “last” decision point
- However, f is control-dependent on e, and S1 and S3 are transitively (iteratively) control-dependent on e

Example (contd.)

- Can a node be control-dependent on more than one node?
 - yes, see example
 - nested repeat-until loops
 - n is control-dependent on t1 and t2 (why?)
- In general, control-dependence relation can be quadratic in size of program

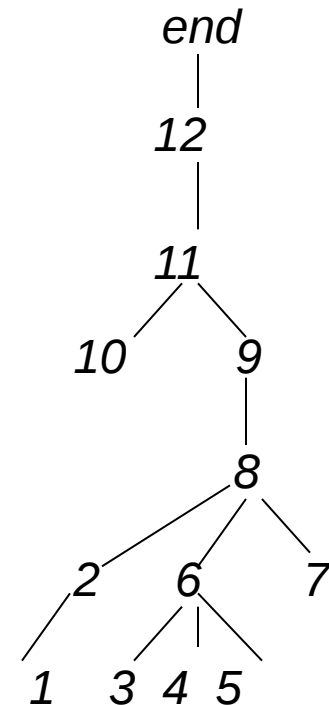
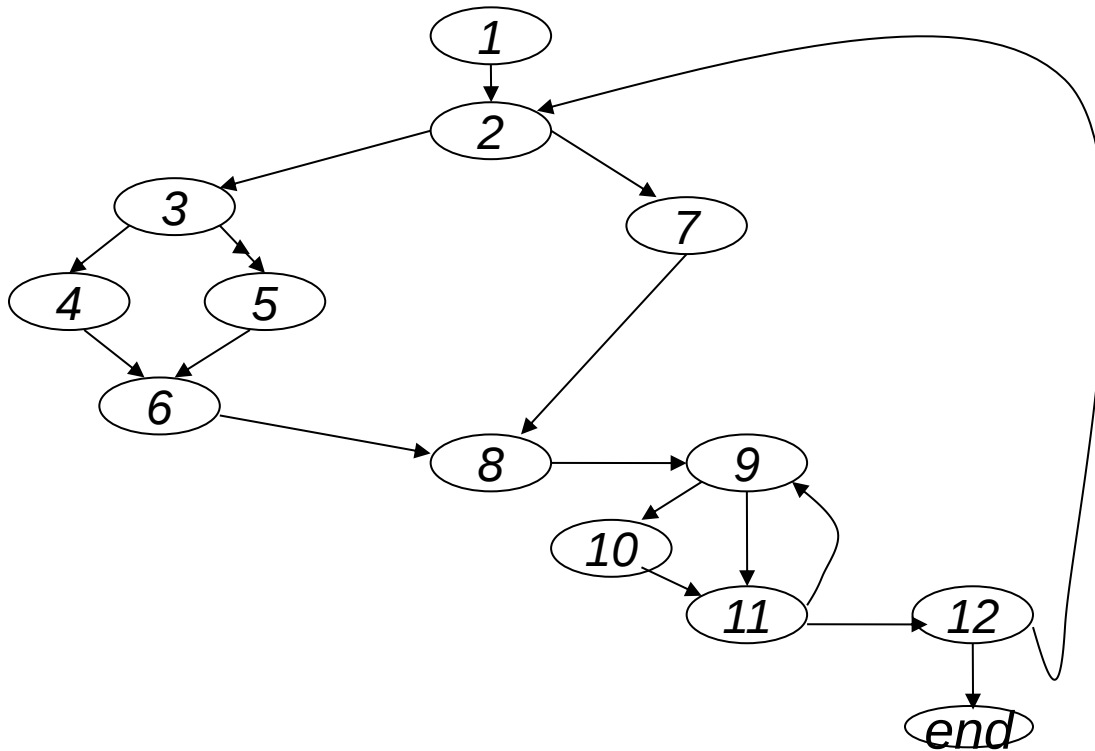


Formal definition of control dependence

- Formalizing these intuitions is quite tricky
- Starting around 1980, lots of proposed definitions
- Commonly accepted definition due to Ferrane, Ottenstein, Warren (1987)
- Uses idea of postdominance
- We will use a slightly modified definition due to Bilardi and Pingali which is easier to think about and work with

Postdominance relation

- Postdominance: relation on nodes ($\subseteq V \times V$)
 - u postdominates v if u occurs on all paths $v \rightarrow^* \text{END}$
 - postdominance is reflexive, transitive and anti-symmetric
 - transitive reduction is tree-structured
 - postdominator tree can be built in $O(|E|+|V|)$ time (Buchsbaum et al)
 - immediate postdominator of u: parent of u in tree

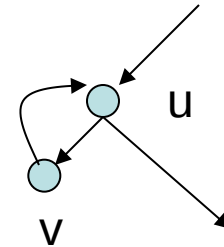


Control dependence definition

- First cut: given a CFG G , a node w is control-dependent on an edge $(u \rightarrow v)$ if
 - w postdominates v
 - w does not postdominate u
- Intuitively,
 - first condition: if control flows from u to v it is guaranteed that w will be executed
 - second condition: but from u we can reach END without encountering w
 - so there is a decision being made at u that determines whether w is executed

Control dependence definition

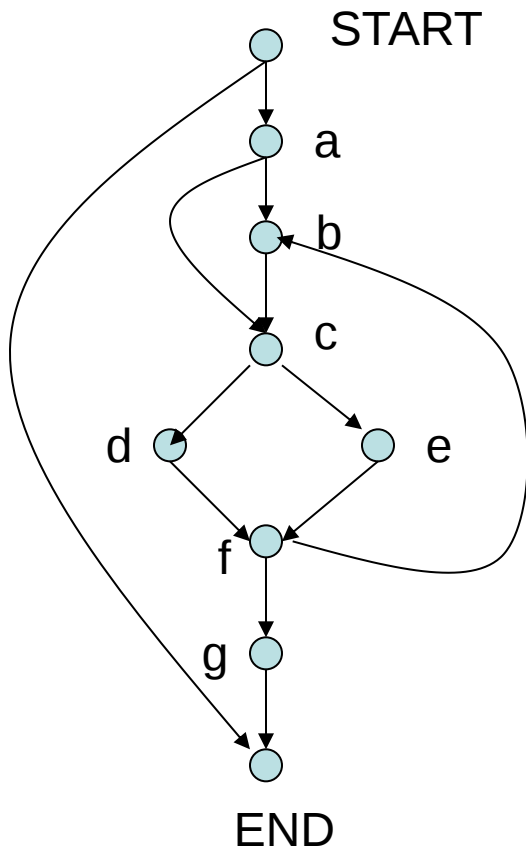
- Small caveat: what if $w = u$ in previous definition?
 - See picture: is u control-dependent on edge $u \rightarrow v$?
 - Intuition says yes, but definition on previous slides says “ u should not postdominate u ” and our definition of postdominance is reflexive
- Fix: given a CFG G , a node w is control-dependent on an edge $(u \rightarrow v)$ if
 - w postdominates v
 - if w is not u , w does not postdominate u



Strict postdominance

- A node w is said to strictly postdominate a node u if
 - $w \neq u$
 - w postdominates u
- That is, strict postdominance is the irreflexive version of the dominance relation
- Control dependence: given a CFG G , a node w is control-dependent on an edge $(u \rightarrow v)$ if
 - w postdominates v
 - w does not strictly postdominate u

Example

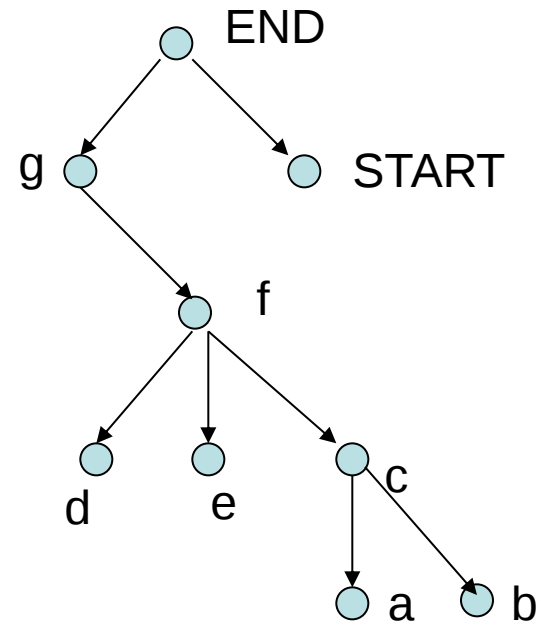


START $\rightarrow$ a
f $\rightarrow$ b
c $\rightarrow$ d
c $\rightarrow$ e
a $\rightarrow$ b

	a	b	c	d	e	f	g
START $\rightarrow$ a	x		x			x	x
f $\rightarrow$ b		x	x			x	
c $\rightarrow$ d				x			
c $\rightarrow$ e					x		
a $\rightarrow$ b		x					

Computing control-dependence relation

- Nodes control dependent on edge $(u \rightarrow v)$ are nodes on path up the postdominator tree from v to $\text{ipdom}(u)$, excluding $\text{ipdom}(u)$
 - We will write this as $[v, \text{ipdom}(u))$
 - half-open interval in tree



	a	b	c	d	e	f	g
START $\rightarrow$ a	X		X			X	X
f $\rightarrow$ b		X	X			X	
c $\rightarrow$ d				X			
c $\rightarrow$ e					X		
a $\rightarrow$ b		X					

Computing control-dependence relation

- Compute the postdominator tree
- Overlay each edge $u \rightarrow v$ on pdom tree and determine nodes in interval $[v, \text{ipdom}(u))$
- Time and space complexity is $O(EV)$.
- Faster solution: in practice, we do not want the full relation, we only make queries
 - $\text{cd}(e)$: what are the nodes control-dependent on an edge e ?
 - $\text{conds}(w)$: what are the edges that w is control-dependent on?
 - $\text{cdequiv}(w)$: what nodes have the same control-dependences as node w ?
- It is possible to implement a simple data structure that takes $O(E)$ time and space to build, and that answers these queries in time proportional to output of query (optimal) (Pingali and Bilardi 1997).

Effective abstractions

- Program abstraction is *effective* if you can write an interpreter for it
- Why is this interesting?
 - reasoning about programs becomes easier if you have an effective abstraction
 - (eg) give a formal Plotkin-style structured operational semantics for the abstraction, and use that to prove properties of execution sequences
- One problem with PDG
 - not clear how to write an interpreter for PDG

Dataflow graphs: an effective abstraction

- From functional languages community
- Functional languages:
 - values and functions from values to values
 - no notion of storage that can be overwritten successively with different values
- Dependence viewpoints:
 - only flow-dependences
 - no anti-dependences or output-dependences
- Dataflow graph:
 - shows how values are used to compute other values
 - no notion of control-flow
 - control-dependence is encoded as data-dependence
 - effective abstraction: interpreter can execute abstraction in parallel
- Major contributors:
 - Jack Dennis (MIT): static dataflow graphs
 - Arvind (MIT): dynamic dataflow graphs

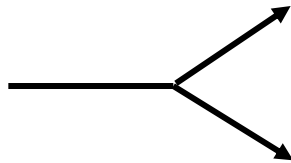
Static Dataflow Graphs

Slides from Arvind

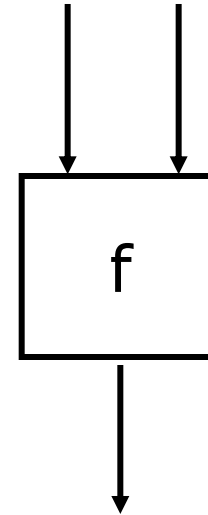
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Dennis' Program Graphs

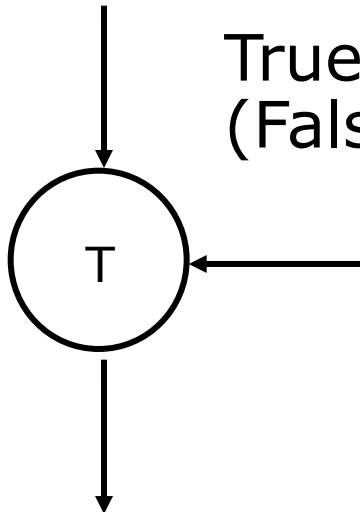
Operators connected by arcs



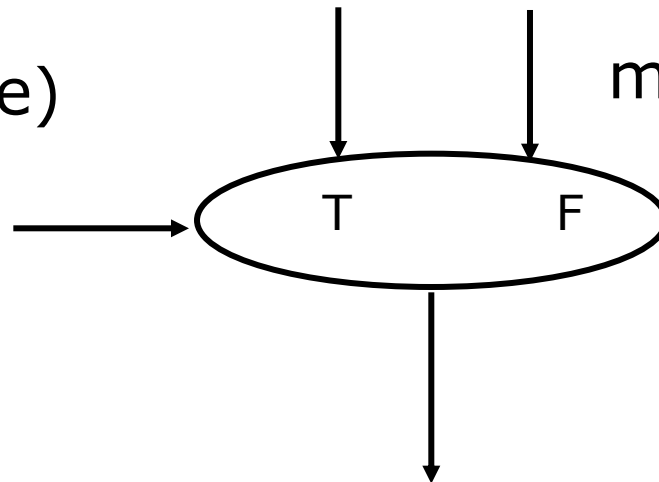
fork



arithmetic
operators and
predicates



True gate
(False gate)



merge

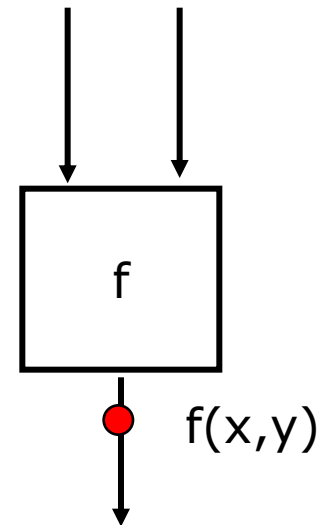
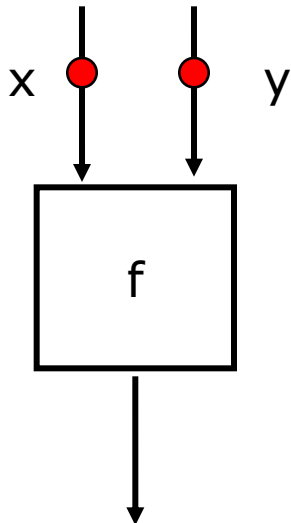
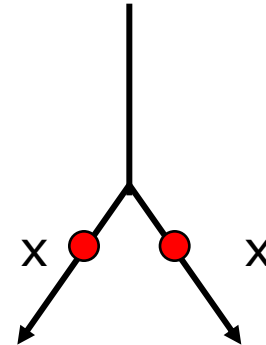
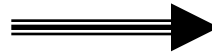
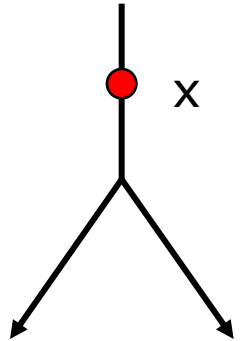
Dataflow

- Execution of an operation is *enabled* by *availability* of *the required operand* values. The completion of one operation makes the resulting values available to the elements of the program whose execution depends on them.

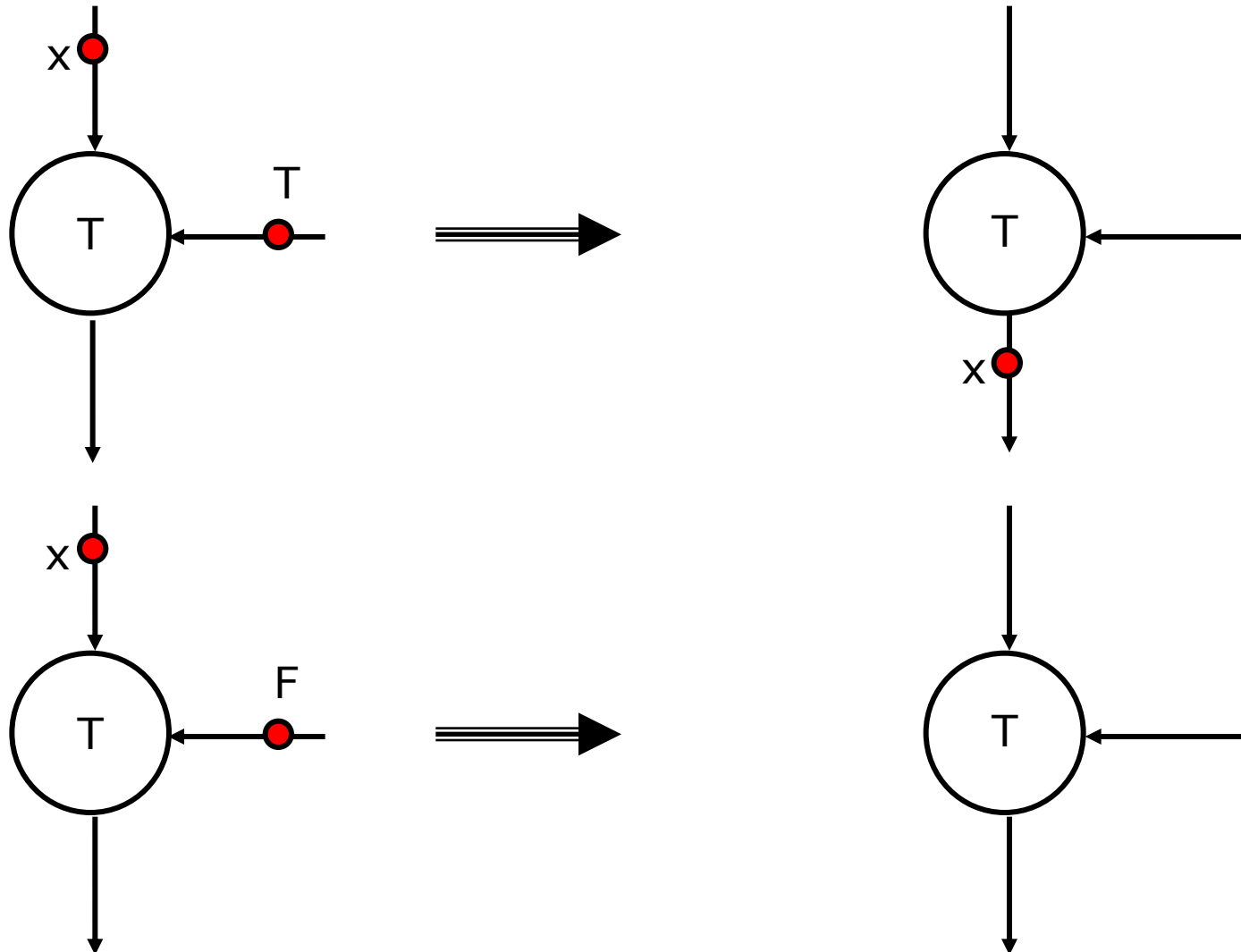
Dennis

- Execution of an operation must not cause *side-effect* to preserve *determinacy*. The effect of an operation must be local.

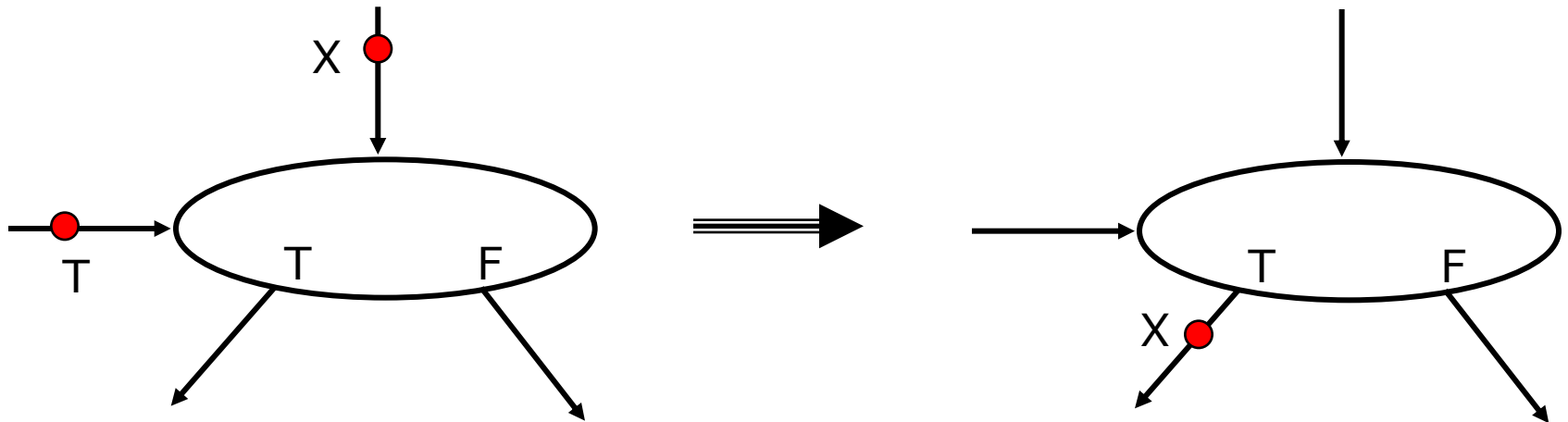
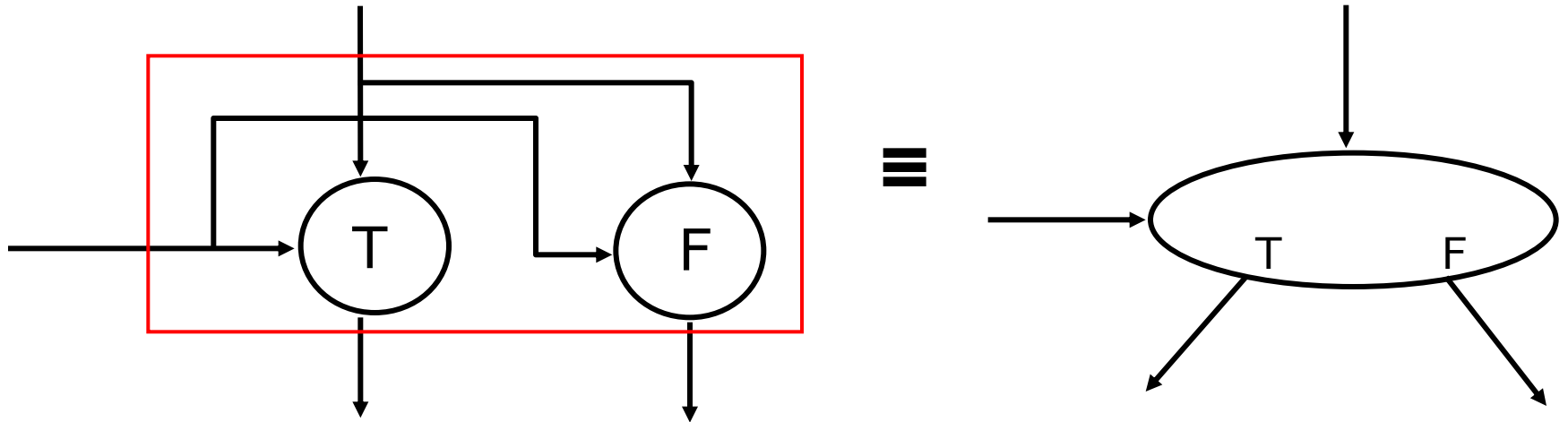
Firing Rules: Functional Operators



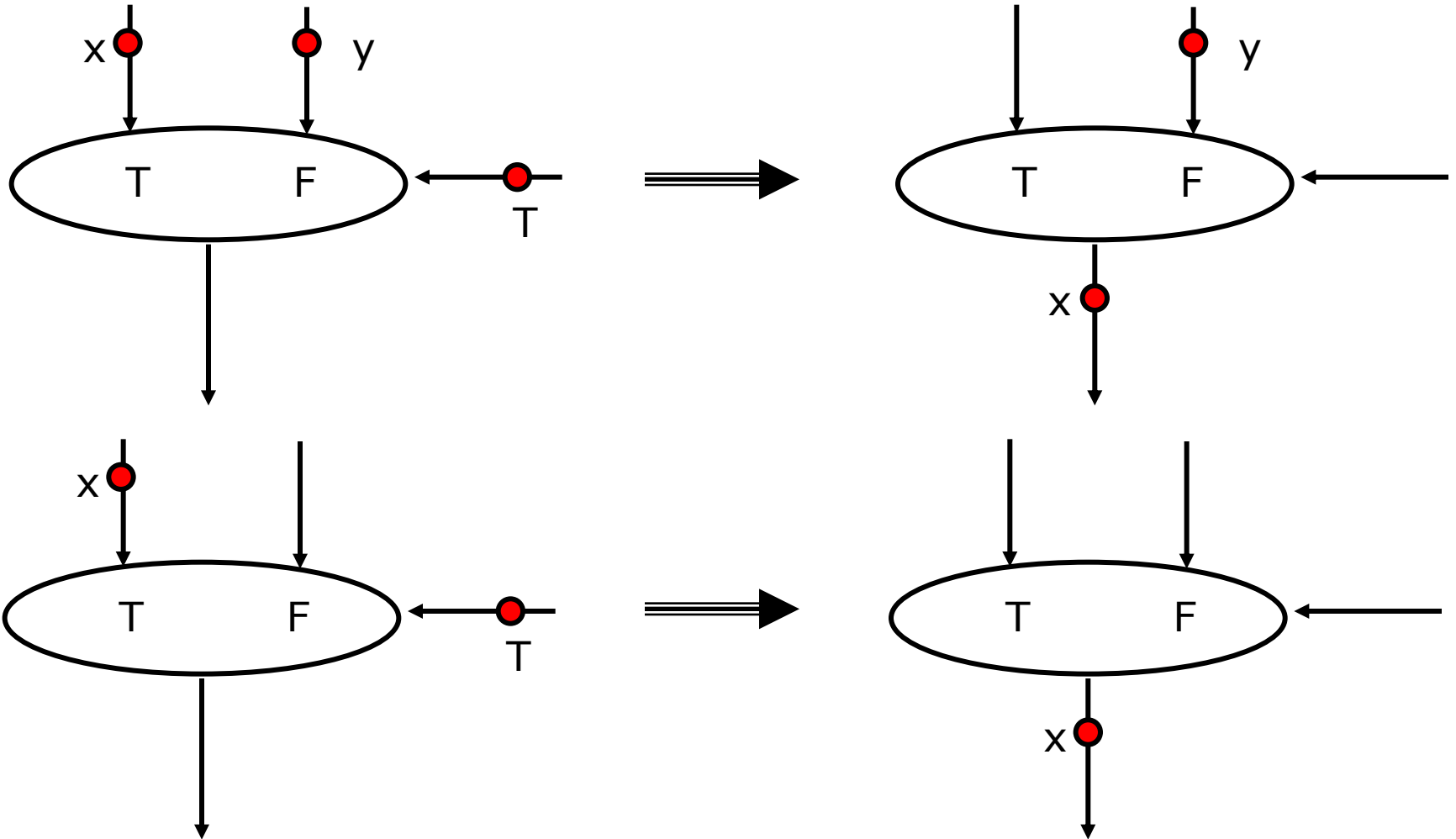
Firing Rules: T-Gate



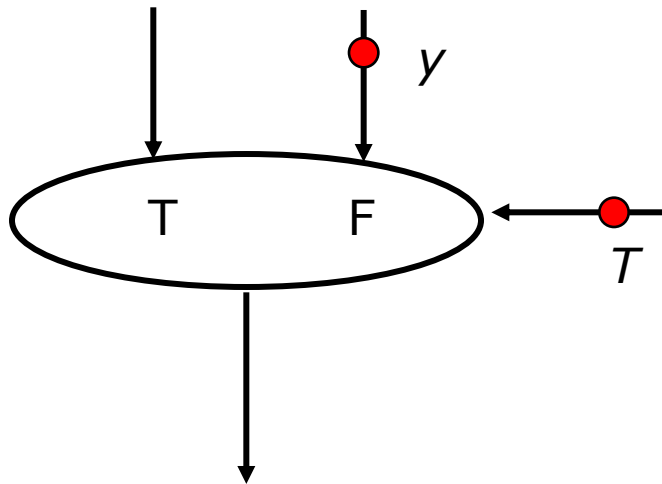
The Switch Operator



Firing Rules: Merge

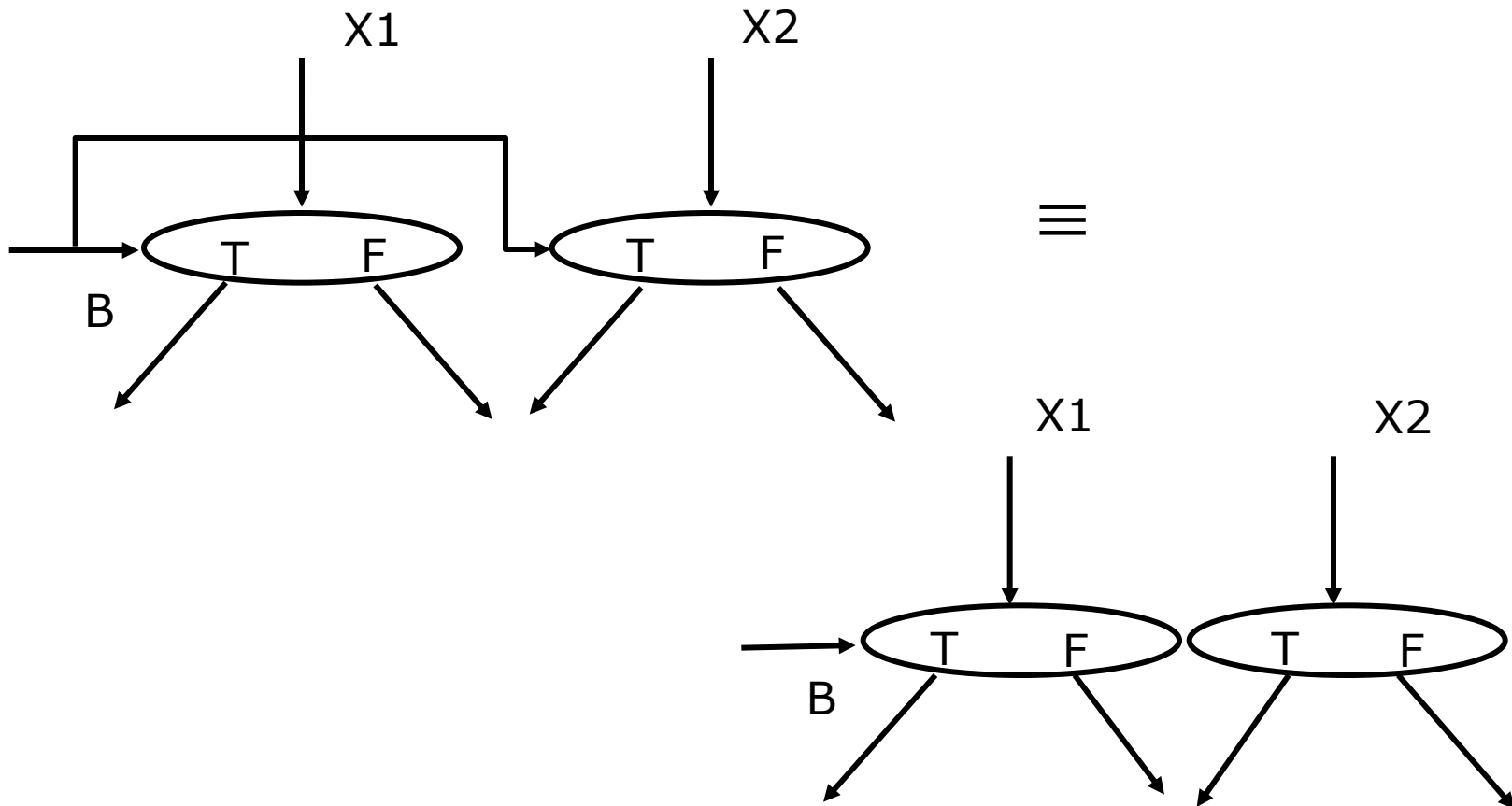


Firing Rules: Merge *cont*

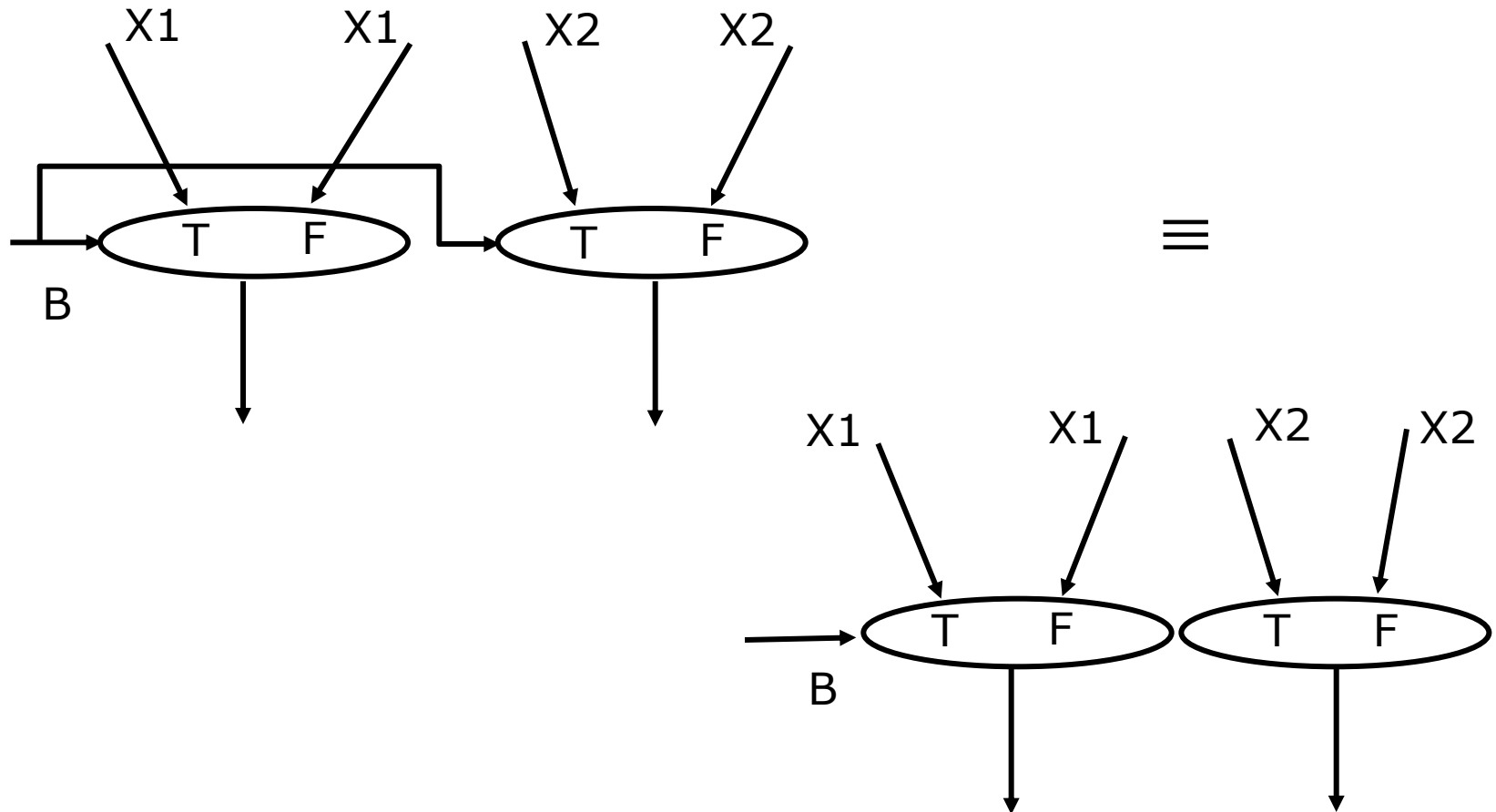


not ready
to fire

Some Conventions

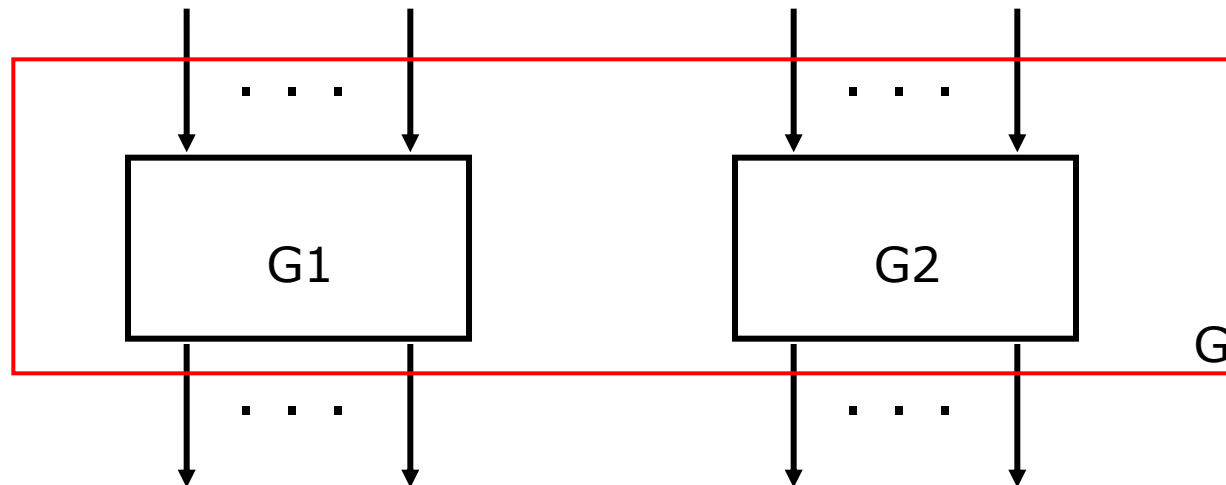
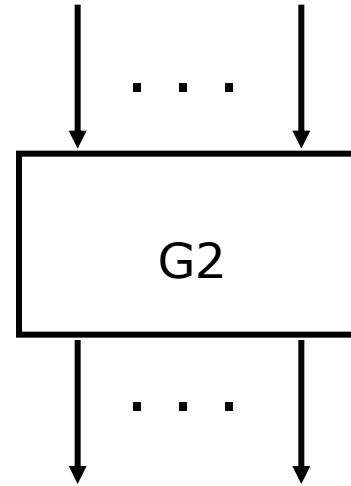
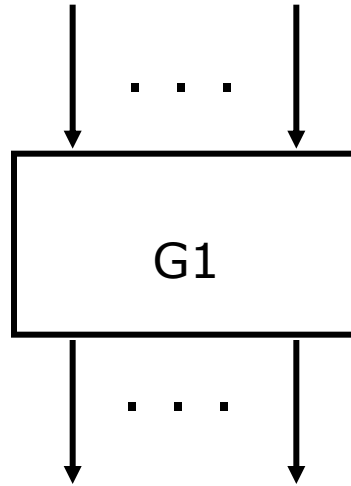


Some Conventions *Cont.*



Rules To Form Dataflow Graphs: *Juxtaposition*

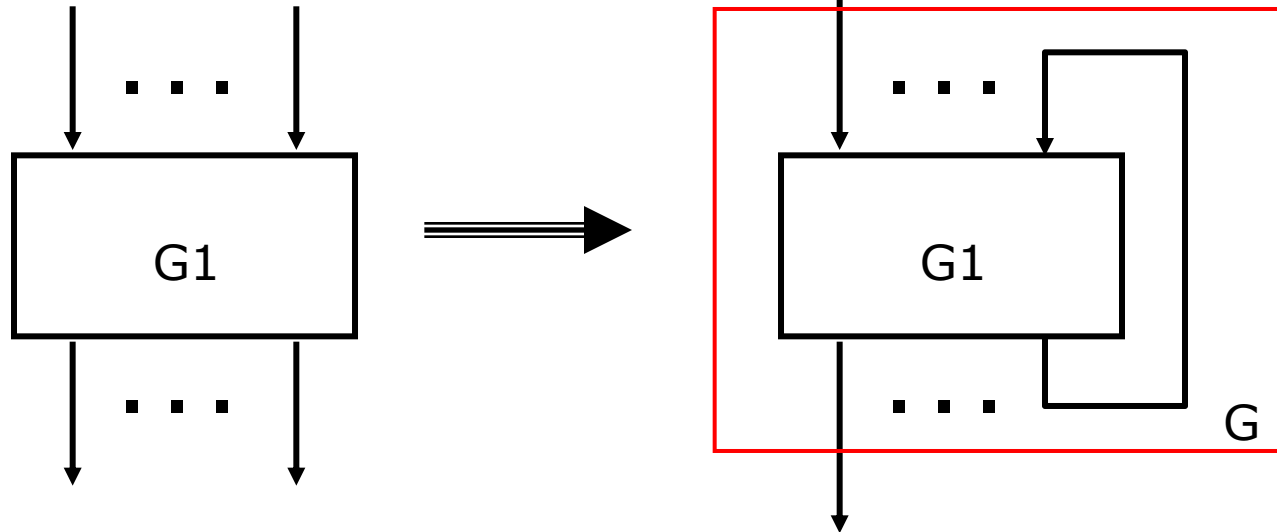
Given



Rules To Form Dataflow

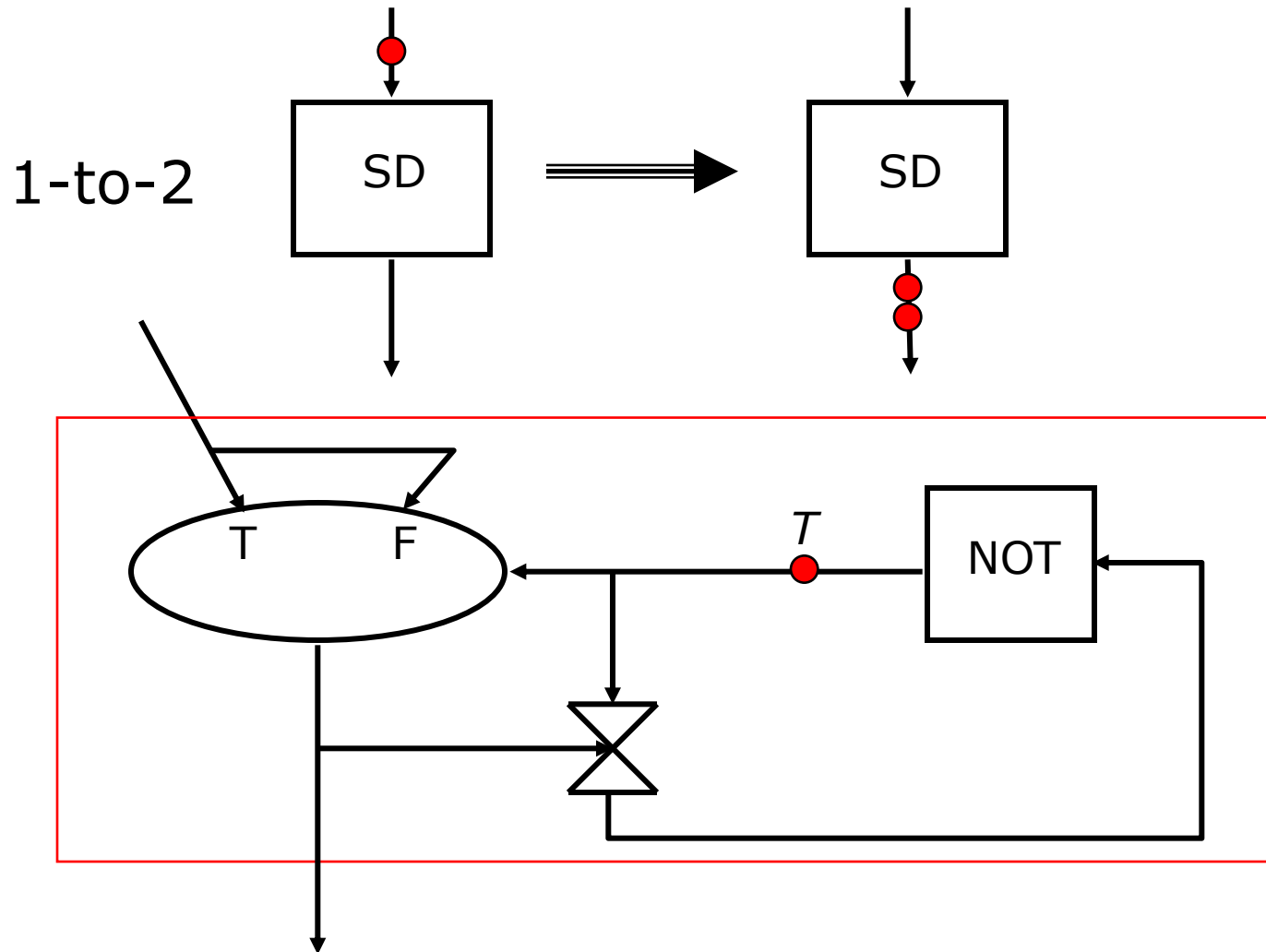
Graphs: *Iteration*

Given

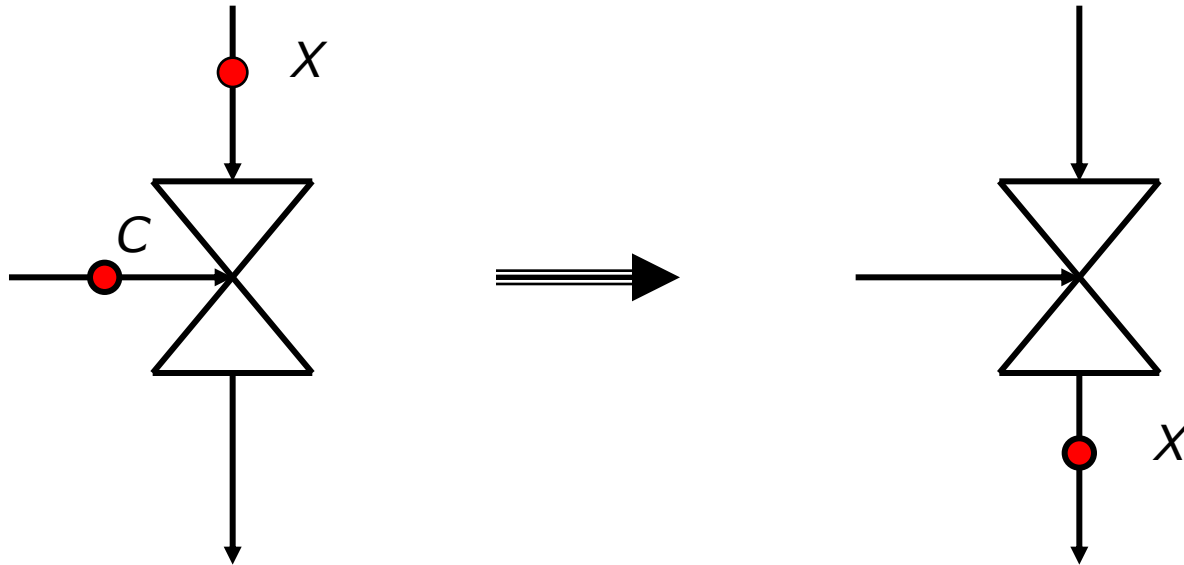


Example:

The Stream Duplicator



The Gate Operator

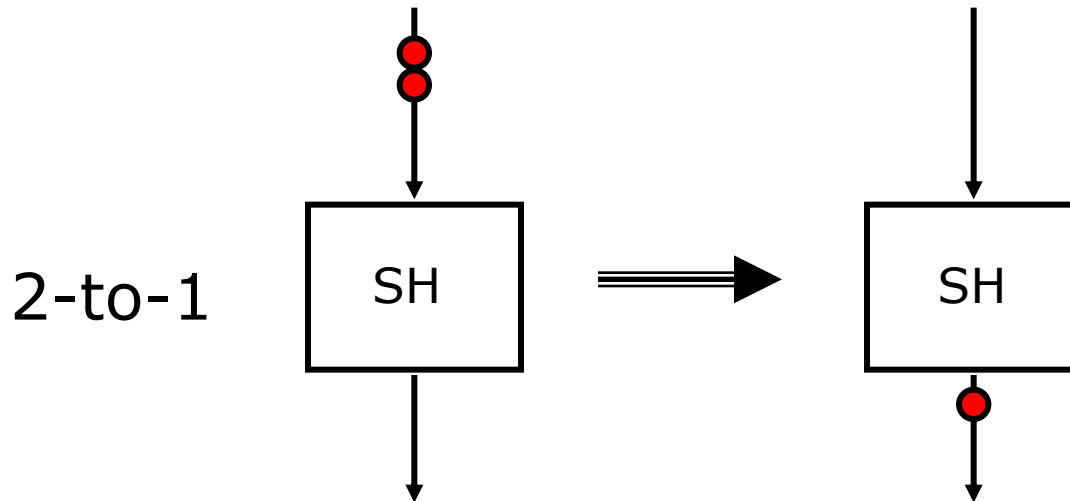


Lets X pass through only after C arrives.

What happens if we don't use the gate in the Stream Duplicator?

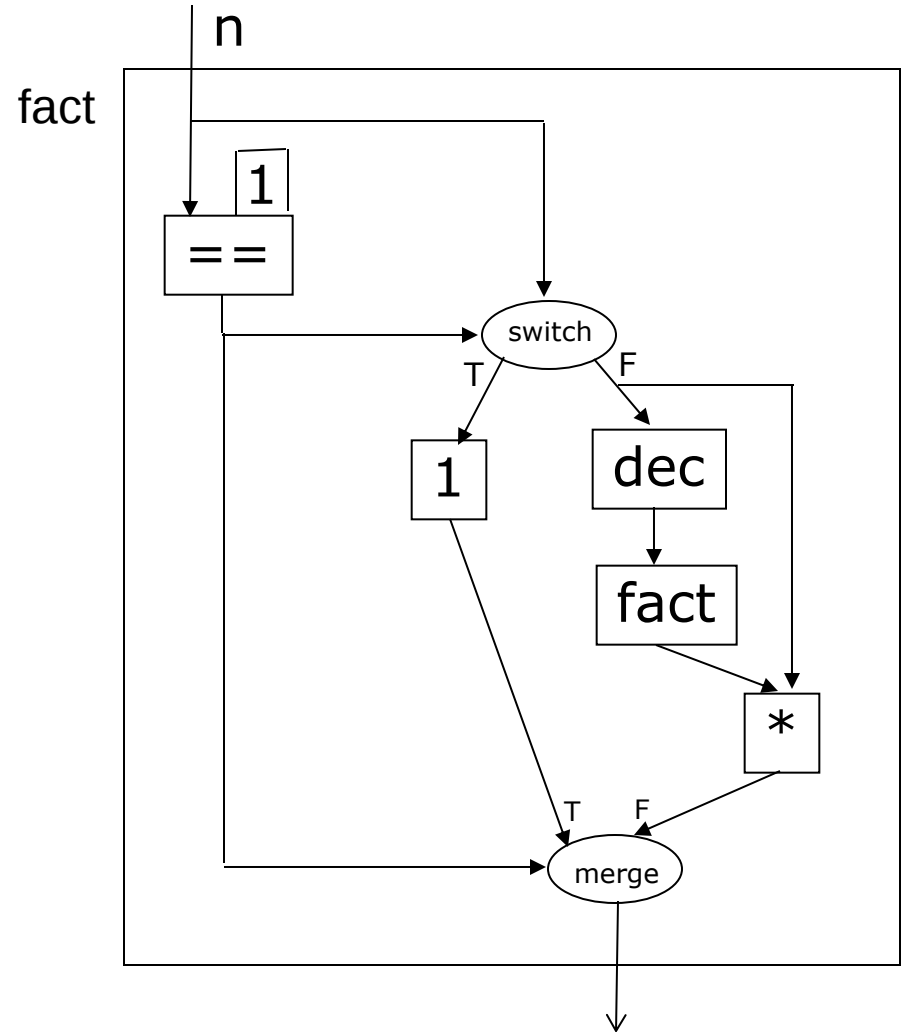
The Stream Halver

Throws away every other token.



Translation to dataflow graphs

- fact(n) =
if (n==1) then 1
else n*fact(n-1)



Determinate Graphs

Graphs whose *behavior is time independent*, i.e., the values of output tokens are uniquely determined by the values of input tokens.

A dataflow graph formed by repeated *juxtaposition and iteration of deterministic dataflow operators* results in a deterministic graph.

Problem with functional model

- Data structures are values
- No notion of updating elements of data structures
- Think about our examples:
 - How would you do DMR?
 - Can you do event-driven simulation without speculation?

Effective parallel abstractions for imperative languages

- Beck et al: From Control Flow to Dataflow
- Approach:
 - extend dataflow model to include side-effects to memory
 - control dependences are encoded as data-dependences as in standard dataflow model
- Uses:
 - execute imperative languages on dataflow machines (which were being built back in 1990)
 - intermediate language for reasoning operationally about parallelism in imperative languages

Limitations of computation graphs

- For most irregular algorithms, we cannot generate a static computation graph
 - dependences are a function of runtime data values
- Therefore, much of the scheduling technology developed for computation graphs is not useful for irregular algorithms
- Even if we can generate a computation graph, latencies of operations are often unpredictable
- Bottom-line
 - useful to understand what is possible if perfect information about program is available
 - but need heuristics like list-scheduling even in this case!

Summary

- Computation graphs
 - nodes are computations
 - edges are dependences
 - node weights are execution times
- Static computation graphs obtained by
 - studying the algorithm
 - analyzing the program
- Limits on speed-ups
 - critical path
 - Amdahl's law
- DAG scheduling
 - heuristic: list scheduling (many variations)
 - static scheduling: VLIW code generation problem
 - dynamic scheduling: DAGuE
- Static computation graphs are useful for regular algorithms, but not very useful for irregular algorithms

Generating computation graphs

- How do we produce computation graphs in the first place?
- Two approaches
 - specify DAG explicitly
 - like parallel programming
 - easy to make mistakes
 - race conditions: two nodes that write to same location but are not ordered by dependence
 - by compiler analysis of sequential programs
- Let us study the second approach
 - called **dependence analysis**

Putting it all together

- Write sequential program.
- Compiler produces parallel code
 - generates control-flow graph
 - produces computation DAG for each basic block by performing dependence analysis
 - generates schedule for each basic block
 - use list scheduling or some other heuristic
 - branch at end of basic block is scheduled on all processors
- Problem:
 - average basic block is fairly small (~ 5 RISC instructions)
- One solution:
 - transform the program to produce bigger basic blocks

Limitations

- PRAM model abstracts away too many important details of real parallel machines
 - synchronous model of computing does not scale to large numbers of processors
 - global memory that can be read/written in every cycle by all processors is hard to implement
- DAG model of programs
 - for irregular algorithms, we may not be able to generate *static* computation DAG
 - even if we could generate a static computation DAG, latencies of some nodes may be variable on a real machine
 - what is the latency of a load?
- Given all these limitation, why study list scheduling on PRAM's in so much detail?

Generating computation graphs

- How do we produce computation graphs in the first place?
- Two approaches
 - specify DAG explicitly
 - like parallel programming
 - easy to make mistakes
 - race conditions: two nodes that write to same location but are not ordered by dependence
 - by compiler analysis of sequential programs
- Let us study the second approach
 - called **dependence analysis**

Putting it all together

- Write sequential program.
- Compiler produces parallel code
 - generates control-flow graph
 - produces computation DAG for each basic block by performing dependence analysis
 - generates schedule for each basic block
 - use list scheduling or some other heuristic
 - branch at end of basic block is scheduled on all processors
- Problem:
 - average basic block is fairly small (~ 5 RISC instructions)
- One solution:
 - transform the program to produce bigger basic blocks

Limitations

- PRAM model abstracts away too many important details of real parallel machines
 - synchronous model of computing does not scale to large numbers of processors
 - global memory that can be read/written in every cycle by all processors is hard to implement
- DAG model of programs
 - for irregular algorithms, we may not be able to generate *static* computation DAG
 - even if we could generate a static computation DAG, latencies of some nodes may be variable on a real machine
 - what is the latency of a load?
- Given all these limitation, why study list scheduling on PRAM's in so much detail?

