

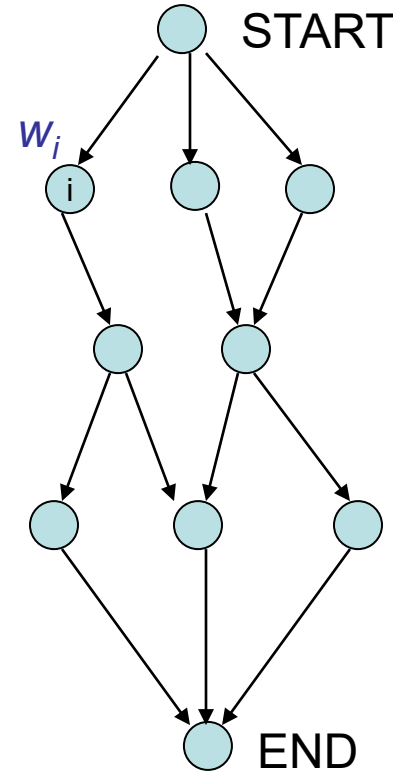
Parallel Computing:
Work, Span, Speedup bounds

Goal of lecture

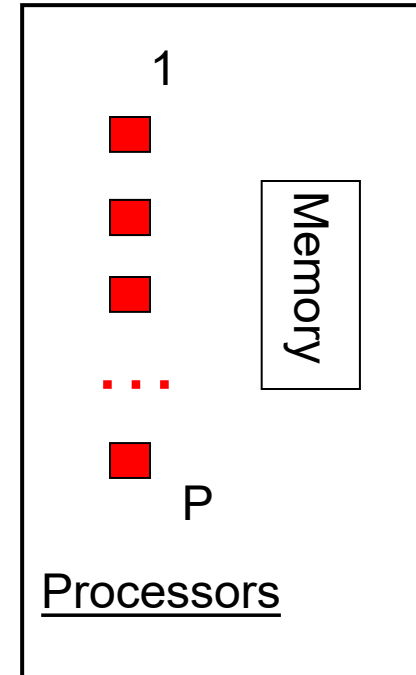
- So far, we have studied
 - how parallelism and locality arise in programs
 - ordering constraints between tasks for correctness or efficiency
- This lecture: introduce some key notions in parallel computing
 - Work
 - Span (critical path)
 - Bounds on speed-up from parallel computing

Program Model

- **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

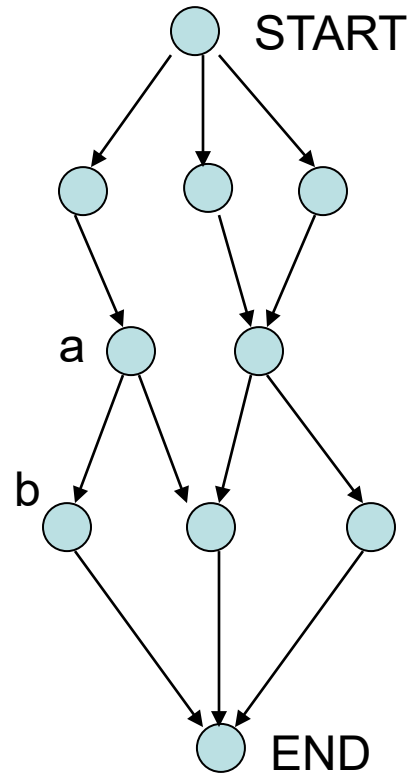


Dependence DAG

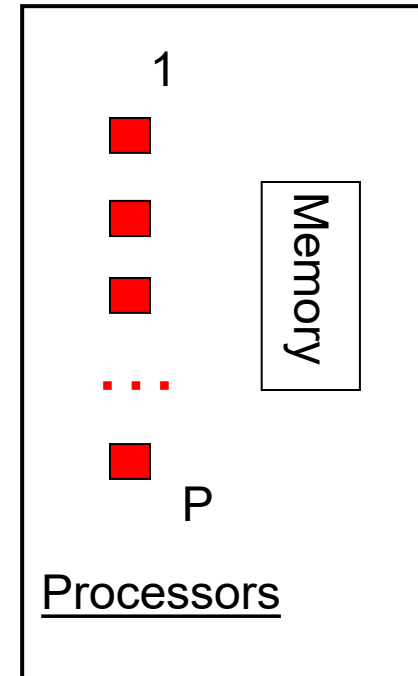


Computer model

- 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** (scheduling)
- 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

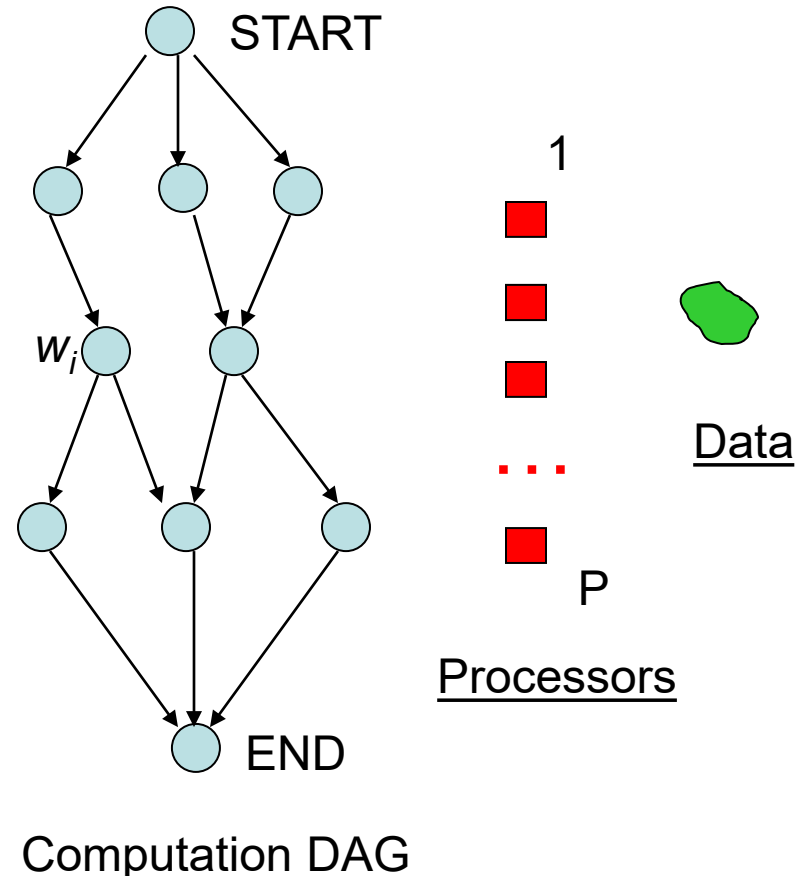


Dependence DAG



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
- **Span (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_1



Terminology

- Instantaneous parallelism

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

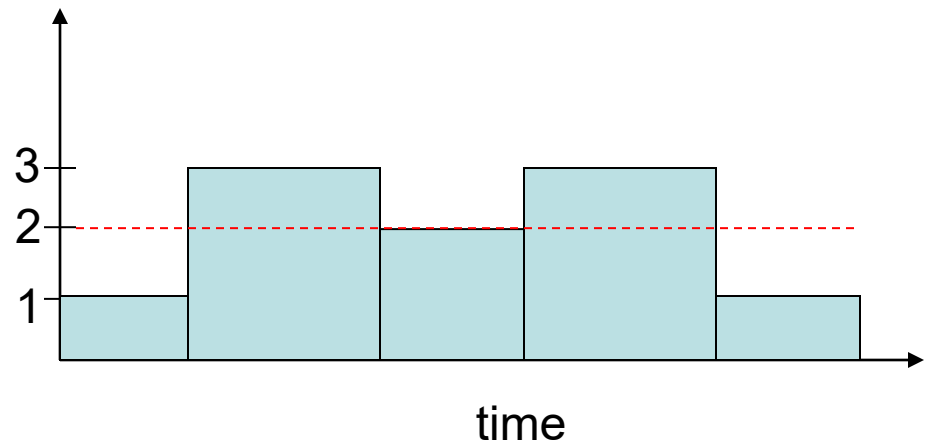
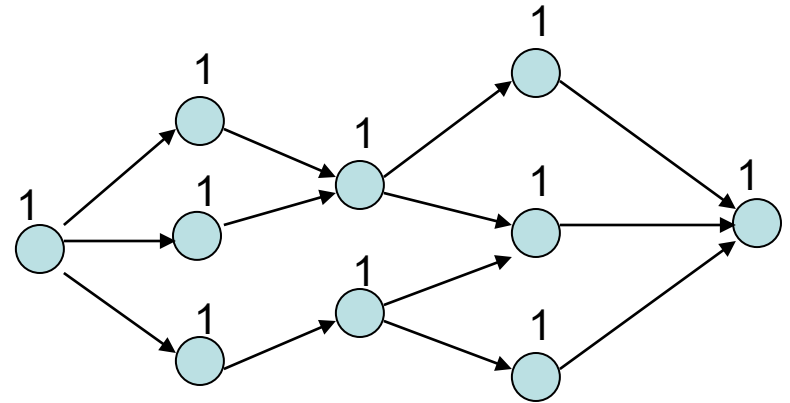
- Maximal parallelism

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

Speed-up bound (I)

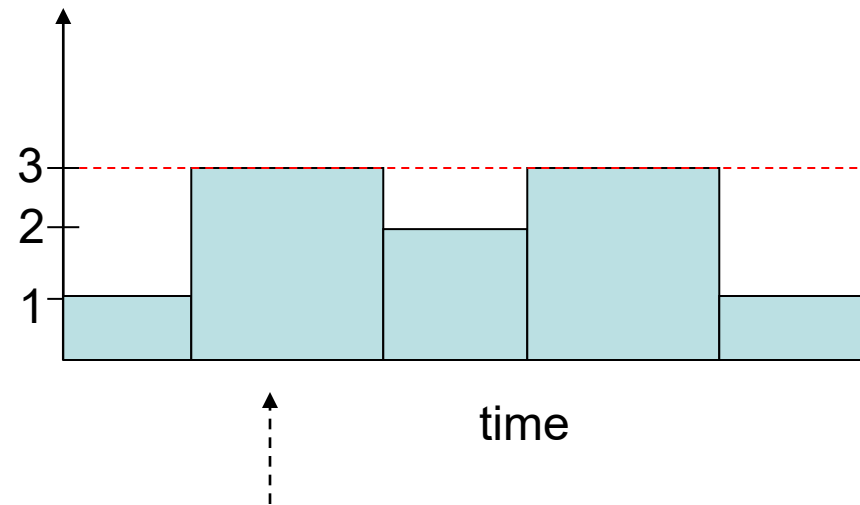
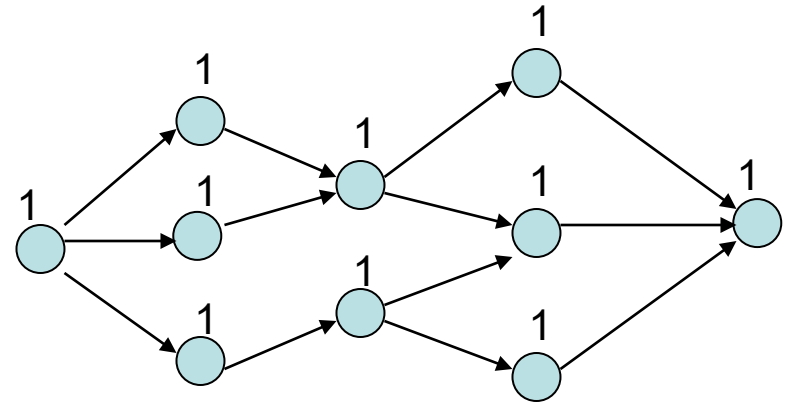
- Speed-up(P) = T_1/T_P
 - intuitively, how much faster is it to execute program on P processors than on 1 processor?
- Bound on speed-up
 - regardless of how many processors you have, you need at least T_1 units of time
 - speed-up(P) · $T_1/T_1 = \sum_i w_i / CP = AP$

Amdahl's Law

- Special case of previous bound
 - 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:
 - $p = 90\% \rightarrow \text{speed-up} \cdot 10$
 - $p = 99\% \rightarrow \text{speed-up} \cdot 100$
- To obtain significant speed-up, most of the program must be performed in parallel
 - serial bottlenecks can really hurt you

Scheduling

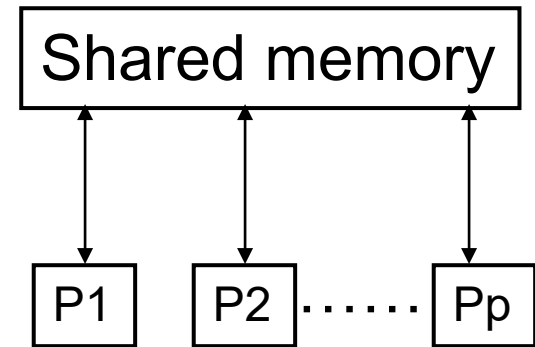
- Suppose $P \cdot MP$
- There will be times during the execution when only a subset of “ready” nodes can be executed.
- Time to execute DAG can depend on which subset of P nodes is chosen for execution.
- To understand this better, it is useful to have a more detailed machine model



What if we only had 2 processors?

Machine Model

- Processors operate synchronously (in lock-step)
 - barrier synchronization in hardware
 - if a processor has reached step i , it can assume all other processors have completed tasks in all previous steps
- Each processor has private memory



Schedules

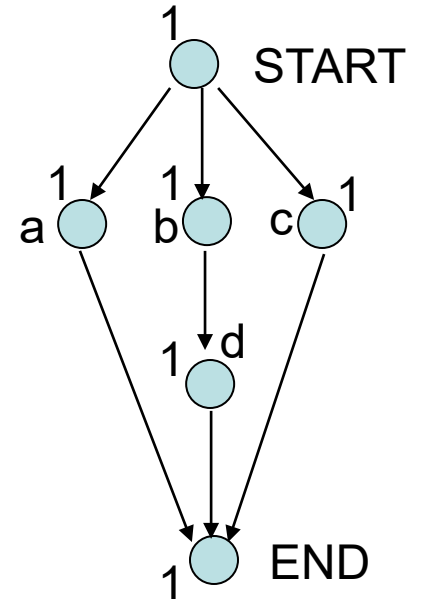
Schedule: function from node to (processor, start time)
Also known as “space-time mapping”

Schedule 1

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

Schedule 2

		time →				
		0	1	2	3	4
space ↓	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

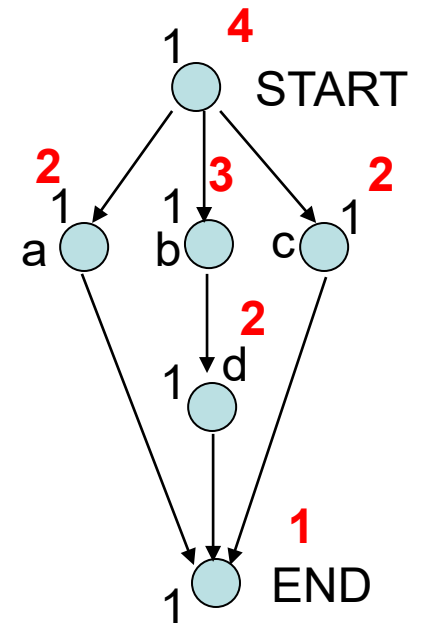
```
cycle c = 0;
ready-list = {START};
inflight-list = { };
while (|ready-list|+|inflight-list| > 0) {
    for each node n in ready-list in priority order { //schedule new tasks
        if (a processor is free at this cycle) {
            remove n from ready-list and add to inflight-list;
          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

Example

	time →				
	0	1	2	3	4
space ↓ 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

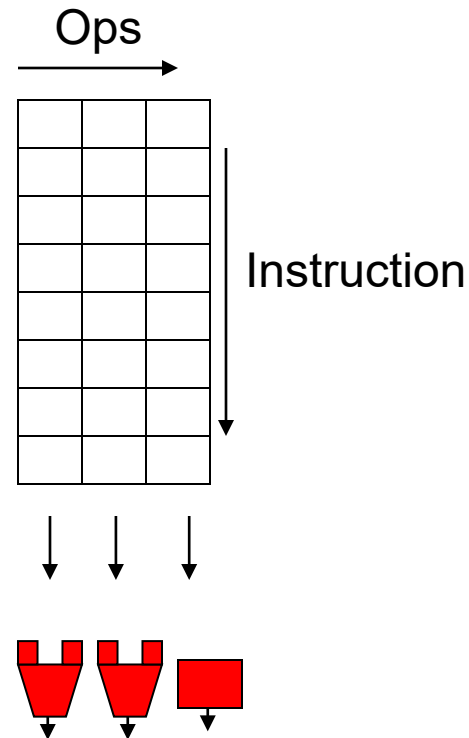
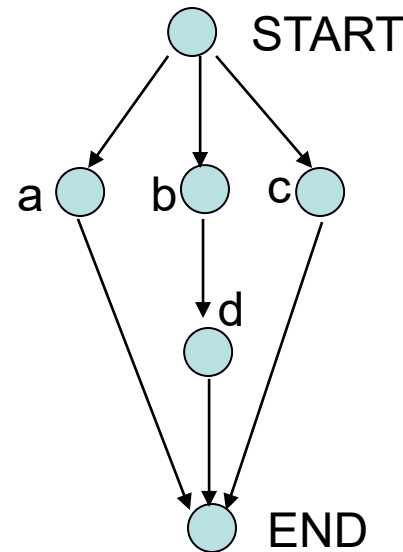
■ P1

Two applications

- **Static scheduling**
 - create space-time diagram at compile-time
 - VLIW code generation
- **Dynamic scheduling**
 - create space-time diagram at runtime
 - multicore scheduling for dense linear algebra

Scheduling instructions for VLIW machines

- Processors → functional units
- Local memories → registers
- Global memory → memory
- Time → instruction
- Nodes in DAG are operations (load/store/add/mul/branch/..)
 - **instruction-level** parallelism
- List scheduling
 - useful for scheduling innermost loop code for pipelined, superscalar and VLIW machines
 - used widely in commercial compilers



Loop unrolling

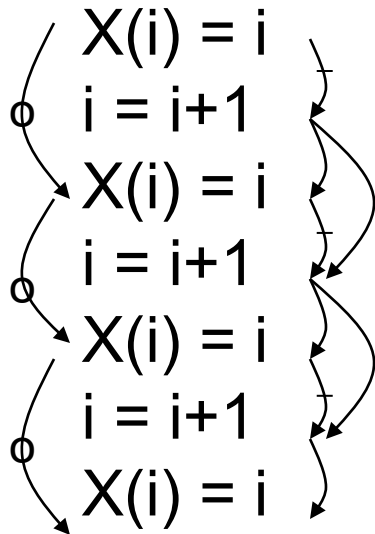
- Original program

```
for i = 1,100
```

```
  X(i) = i
```

- Unroll loop 4 times: not very useful!

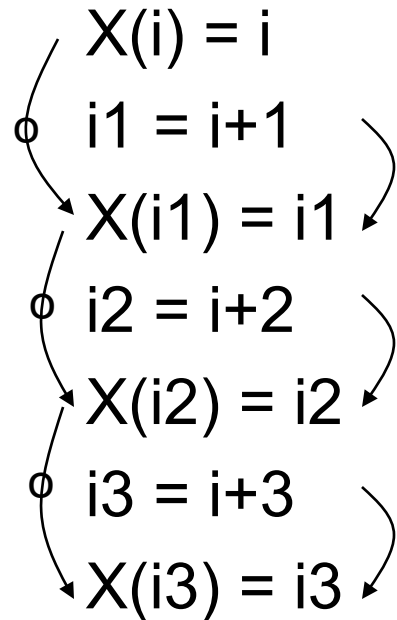
```
for i = 1,100,4
```



Smarter loop unrolling

- Use new name for loop iteration variable in each unrolled instance

for $i = 1, 100, 4$





Array dependence analysis


- If compiler can also figure out that $X(i)$, $X(i+1)$, $X(i+2)$, and $X(i+3)$ are different locations, we get the following dependence graph for the loop body


for $i = 1, 100, 4$


$X(i) = i$


$i1 = i+1$ 

$X(i1) = i1$ 

$i2 = i+2$ 

$X(i2) = i2$ 

$i3 = i+3$ 

$X(i3) = i3$ 

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



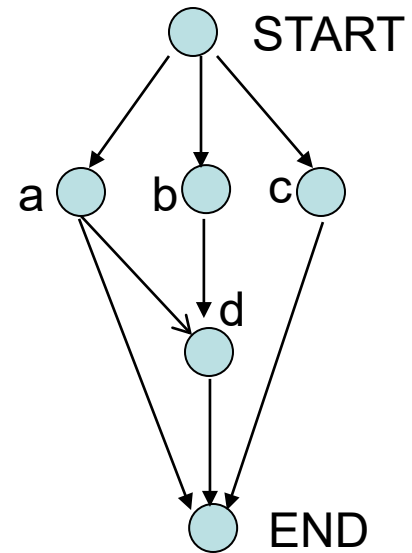
Bob Rau



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
 - counter associated with each DAG node
 - decremented when predecessor task is done
- 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)
```

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

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

$$C_{00} = A_{00} * B_{00} + A_{01} * B_{10}$$

$$C_{01} = A_{01} * B_{11} + A_{00} * B_{01}$$

$$C_{11} = A_{11} * B_{01} + A_{10} * B_{01}$$

$$C_{10} = A_{10} * B_{00} + A_{11} * B_{10}$$

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 blocked dense linear algebra codes
 - nodes: block 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])
```

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

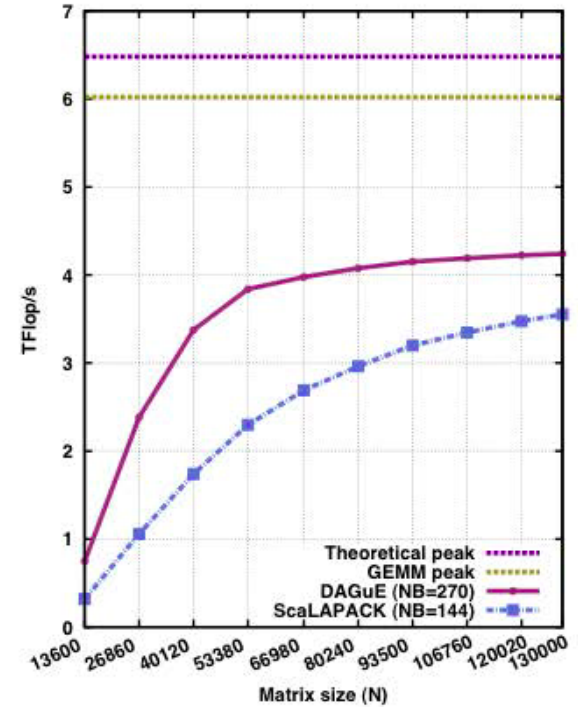
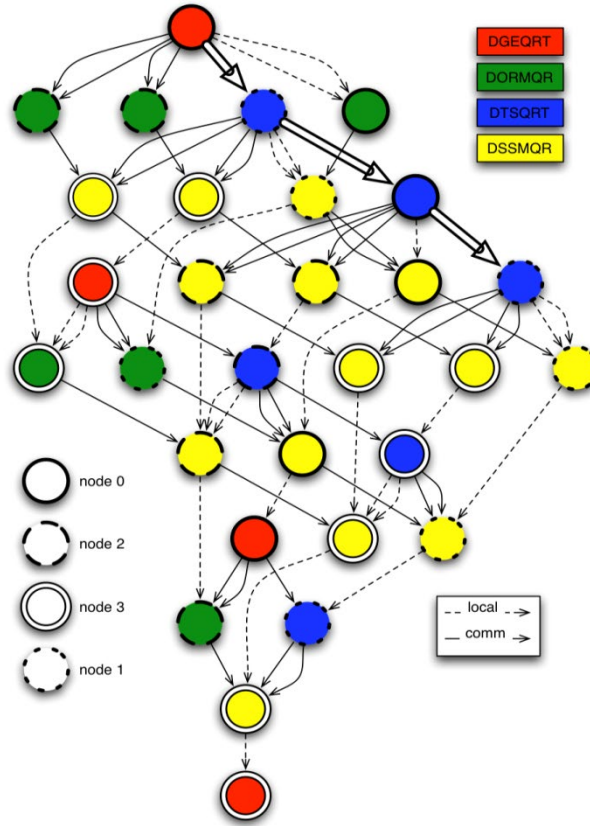
Tiled QR (using tiles and in/out notations)

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 multicore scheduling

- Assumptions

- DAG of tasks is known
- each task is “heavy-weight” and executing task on one worker exploits adequate locality
- no assumptions about runtime of tasks
- no lock-step execution of processors or synchronous global memory

- Scheduling

- keep a work-list of tasks that are ready to execute
- use heuristic priorities to choose from ready tasks

Summary

- Dependence graphs
 - nodes are computations
 - edges are dependences
- Limits on speed-ups
 - critical path
 - Amdahl's law
- DAG scheduling
 - heuristic: list scheduling (many variations)
 - static and dynamic scheduling
 - applications: VLIW code generation, multicore scheduling for dense linear algebra
- Major limitations:
 - works for topology-driven algorithms with fixed neighborhoods since we know tasks and dependences before executing program
 - not very useful for data-driven algorithms since tasks are created dynamically
 - one solution: work-stealing, work-sharing. Study later.