Scheduling

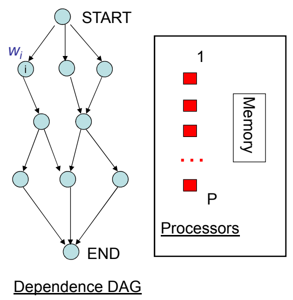
Keshav Pingali
University of Texas, Austin

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: How do we assign tasks to workers?
  - multicore: workers might be cores
  - distributed-memory machines: workers might be hosts/machines
- Scheduling
  - rich literature exists for dependence graph scheduling
  - most of it is not very useful in practice since they use unrealistic program and machine models
    - (e.g.) assume task execution times are known
  - nevertheless, it is useful to study it since it gives us intuition for what the issues are for scheduling in practice

Dependence DAG's

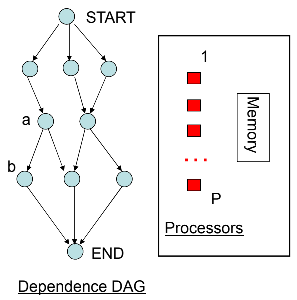
- DAG with START and END nodes
  - all nodes reachable from START
  - END reachable from all nodes
  - START and END are not essential
- Nodes are computations
  - each computation can be executed by a processor in some number of time-steps
  - computation may require reading/writing shared-memory
  - 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



Dependence DAG

Computer model

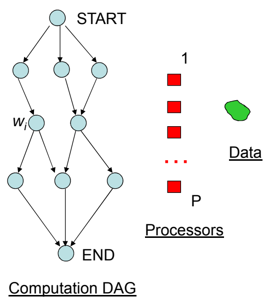
- P identical processors
- Memory
  - processors have local memory
  - all shared-data is stored in global memory
- How does a processor know which nodes it must execute?
  - work assignment
- How does a processor know when it is safe to execute a node?
  - (eg) if P1 executes node a and P2 executes 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

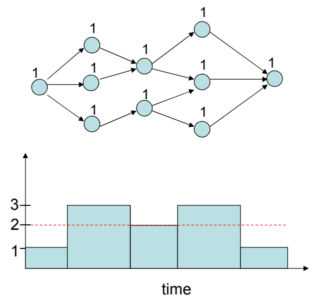
Work and critical path

- Work = « $_1 w_i$
  - 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_4



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_4
- 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 (≠ 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

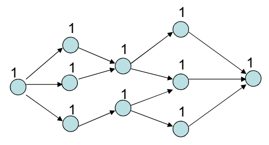
- Speed-up(P) = T_1/T_P
  - intuitively, how much faster is it to execute program on P processors than on 1 processor?
- Bound on speed-up
  - regardless of how many processors you have, you need at least T_4 units of time
  - speed-up(P) ‰ T_1/T_4 = « $_1 w_i$ /CP = AP

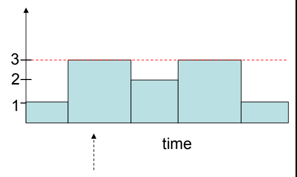
Amdahl's law

- 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% ➔ speed-up ‰ 10
  - p = 99% ➔ speed-up ‰ 100
- To obtain significant speed-up, most of the program must be performed in parallel
  - serial bottlenecks can really hurt you

Scheduling

- Suppose P ‰ 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

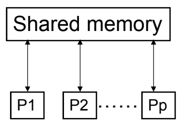




What if we only had 2 processors?

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

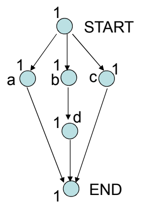
Schedule: function from node to (processor, start time)
Also known as "space-time mapping"

Schedule 1

	0	1	2	3	4
P0	START	a	c	END	
P1		b	d		

Schedule 2

	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

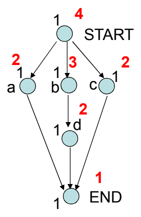
```
cycle c = 0;
ready-list = {START};
inflight-list = { };
while (|ready-list|+|inflight-list| > 0)  {
    for each node n in ready-list in priority order { //schedule new tasks
            if (a processor is free at this cycle) {
                        remove n from ready-list and add to inflight-list;
                        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 →

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!)

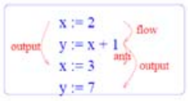
🟥 P0

🟥 P1

Generating dependence graphs

- How do we produce dependence graphs in the first place?
- Two approaches
  - specify DAG explicitly
    - parallel programming
    - easy to make mistakes
      - data races: two tasks 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

Data dependence

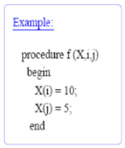


```
x := 2
y := x + 1
x := 3
y := 7
```

- Basic blocks
  - straight-line code
- Nodes represent statements
- Edge $S_1 \rightarrow S_2$
  - flow dependence (read-after-write (RAW))
    - S_1 is executed before S_2 in basic block
    - S_1 writes to a variable that is read by S_2
  - anti-dependence (write-after-read (WAR))
    - S_1 is executed before S_2 in basic block
    - S_1 reads from a variable that is written by S_2
  - output-dependence (write-after-write (WAW))
    - S_1 is executed before S_2 in basic block
    - S_1 and S_2 write to the same variable
  - input-dependence (read-after-read (RAR)) (usually not important)
    - S_1 is executed before S_2 in basic block
    - S_1 and S_2 read from the same variable

Conservative approximation

- In real programs, we often cannot determine precisely whether a dependence exists
  - in example,
    - i = j: dependence exists
    - i ≠ j: dependence does not exist
  - dependence may exist for some invocations and not for others
- Conservative approximation
  - when in doubt, assume dependence exists
  - at the worst, this will prevent us from executing some statements in parallel even if this would be legal
- Aliasing: two program names for the same storage location
  - (e.g.) X(i) and X(j) are *may*-aliases
  - may-aliasing is the major source of imprecision in dependence analysis



Example:

```
procedure f (X,i,j)
  begin
    X(i) = 10;
    X(j) = 5;
  end
```

Putting it all together

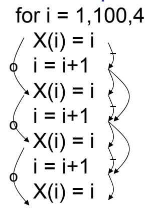
- Write sequential program.
- Compiler produces parallel code
  - generates control-flow graph
  - produces computation DAG for each basic block by performing dependence analysis
  - generates schedule for each basic block
    - use list scheduling or some other heuristic
    - branch at end of basic 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

One transformation: loop unrolling

- Original program

```
for i = 1,100
  X(i) = i
```

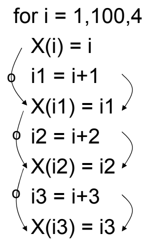
- Unroll loop 4 times: not very useful!



```
for i = 1,100,4
  X(i) = i
  i = i+1
  X(i) = i
  i = i+1
  X(i) = i
  i = i+1
  X(i) = i
```

Smarter loop unrolling

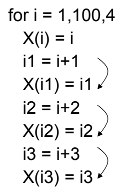
- Use new name for loop iteration variable in each unrolled instance



```
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
```

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
```

Array dependence analysis (contd.)

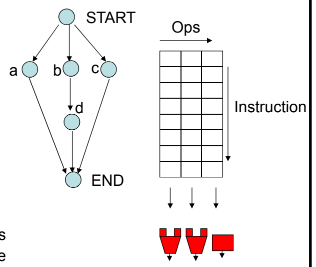
- We will study techniques for array dependence analysis later in the course
- Problem can be formulated as an integer linear programming problem:
  - Is there an integer point within a certain polyhedron derived from the loop bounds and the array subscripts?

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 code for pipelined, superscalar and VLIW machines
  - used widely in commercial compilers
  - loop unrolling and array dependence analysis are also used widely



Historical note on VLIW processors

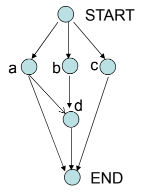
- Ideas originated in late 70's-early 80's
- Two key people:
  - Bob Rau (Stanford,UIUC, TRW, Cydrome, HP)
  - Josh Fisher (NYU,Yale, Multiflow, HP)
- Bob Rau's contributions:
  - transformations for making basic blocks larger:
    - predication
    - software pipelining
  - hardware support for these 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?

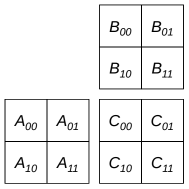
Increasing granularity: Block Matrix Algorithms

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          parallel loops
    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)
```



$C_{00} = A_{00}*B_{00} + A_{01}*B_{10}$
$C_{01} = A_{01}*B_{11} + A_{00}*B_{01}$
$C_{11} = A_{11}*B_{01} + A_{10}*B_{01}$
$C_{10} = A_{10}*B_{00} + A_{11}*B_{10}$

New problem

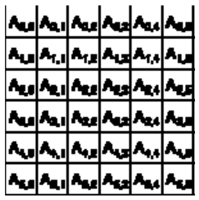
- Difficult to get accurate execution times of coarse-grain nodes
  - conditional inside loop iteration
  - cache misses
  - exceptions
  - O/S processes
  - ….
- Solution: runtime scheduling

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)

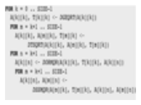
```
FOR k = 0 .. SIZE-1
  A[k][k], T[k][k] <- DGEQRT(A[k][k])
  FOR m = k+1 .. SIZE-1
    A[k][k], A[m][k], T[m][k] <-
         DTSQRT(A[k][k], A[m][k], T[m][k])
  FOR n = k+1 .. SIZE-1
    A[k][n] <- DORMQR(A[k][k], T[k][k], A[k][n])
    FOR m = k+1 .. SIZE-1
      A[k][n], A[m][n] <-
           DSSMQR(A[m][k],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



Tiled QR (using tiles and in/out notations)

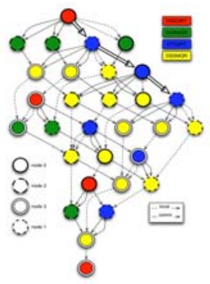
32

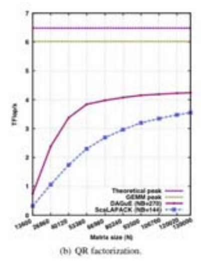
DAGuE: Tiled QR (2)



```
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





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

33

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
- Static dependence 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 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.