

15-853: Algorithms in the Real World

Parallelism: Lecture 4
Dynamic Programming
Scheduling

15-853

Page 1

Edit Distance: Dynamic Programming

| | | B | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----------------------------------|
| | | a | t | c | a | c | a | c | | |
| A | t | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | → insert ↓ delete ↘ common |
| | c | 1 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | |
| | a | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | |
| | a | 3 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | |
| | t | 4 | 3 | 2 | 3 | 2 | 3 | 4 | 5 | |

Note: can be filled in any order as long as the cells to the left and above are filled.

Can follow path back through matrix to construct edits.

15-853

Page 2

Edit Distance: Dynamic Programming

Sequential code:

```

for i = 1 to n
  M[i,1] = i;
for j = 1 to m
  M[1,j] = j;

for i = 2 to n
  for j = 2 to m
    if (A[i] == B[j])
      M[i,j] = M[i-1,j-1];
    else
      M[i,j] = 1 + min(M[i-1,j],M[i,j-1]);
  
```

15-853

Page 3

Edit Distance: Dynamic Programming

| | | B | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----------------------------------|
| | | a | t | c | a | c | a | c | | |
| A | t | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | → insert ↓ delete ↘ common |
| | c | 1 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | |
| | a | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | |
| | a | 3 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | |
| | t | 4 | 3 | 2 | 3 | 2 | 3 | 4 | 5 | |

Note: can be filled in any order as long as the cells to the left and above are filled.

Can follow path back through matrix to construct edits.

15-853

Page 4

Edit Distance: Dynamic Programming

| | | B | | | | | | | |
|---|---|---|---|---|---|---|---|---|----------|
| | | a | t | c | a | c | a | c | |
| A | t | 0 | 1 | 2 | 3 | 4 | 5 | 6 | → insert |
| | c | 1 | 2 | 1 | 2 | 3 | 4 | 5 | ↓ delete |
| | a | 2 | 3 | 2 | 1 | 2 | 3 | 4 | ↘ common |
| | a | 3 | 2 | 3 | 2 | 1 | 2 | 3 | |
| | t | 4 | 3 | 2 | 3 | 2 | 3 | 4 | |

Note: can be filled in any order as long as the cells to the left and above are filled.
Can follow path back through matrix to construct edits.

15-853

Page 5

Edit Distance: Dynamic Programming

Parallel code:

```
for i = 1 to n : M[i,1] = i;
for j = 1 to m : M[1,j] = j;
```

```
for k = 2 to n+m
  istart = max(2,k-m)
  iend = min(k,n)
  parallel for i = istart to iend
    j = k-i+2
    if (A[i] == B[j])
      M[i,j] = M[i-1,j-1];
    else
      M[i,j] = 1 + min(M[i-1,j],M[i,j-1]);
```

15-853

Page 6

Scheduling

So far we have assumed "magic" schedulers that maps dynamic tasks onto processors.

15-853

Page 7

Sidebar: Beyond Nested Parallelism

Assume a way to fork

- Pairwise or multiway

What types of synchronization are allowed

- Fork-join (nested parallelism)
- Futures
- Fully general

The first two can be made deterministic

Can have a large effect on the scheduler and what can be proved about the schedules.

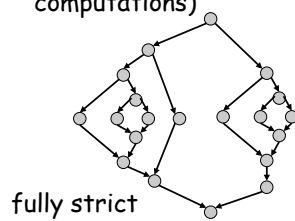
15-853

8

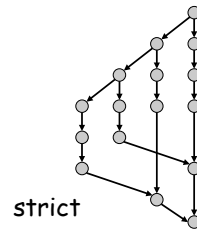
Strict and Fully Strict

Fully strict (fork-join, nested parallel): a task can only synchronize with its parent

Strict: a task can only synchronize with an ancestor.
(X10 recently extended to support strict computations)



15-853

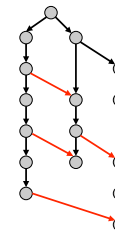


9

Futures

Futures or read-write synchronization variables can be used for pipelining of various forms, e.g. **producer consumer pipelines**. This cannot be supported in strict or fully strict computations.

If read always occurs "after" the write in sequential order then there is no deadlock



15-853

10

General

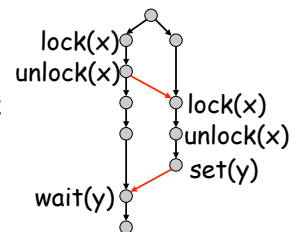
Locks

Transactions

Synch variables

Easy to create deadlock

Hard to schedule



15-853

11

Scheduling Outline

Theoretical results on scheduling

Graham, 1966

Greedy Schedules

Specific greedy schedules

- Breadth First
- Work Stealing
- P-DFS
- Hybrid

15-853

12

Graham Scheduling

"Bounds on Certain Multiprocessor Anomilies", 1966

Model:

Processing Units : $P_i, 1 \leq i \leq n$

Tasks : $T = \{T_1, \dots, T_m\}$

Partial order : $<_T$ on T

Time function : $\mu : T \rightarrow [0, \infty]$

$(T, <_T, \mu)$: define a weighted DAG

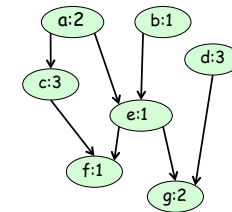
15-853

13

Graham Scheduling



The processors



The DAG

15-853

Page14

Graham: List Scheduling

For a task set T , a Task List L_T is some ordering $[T_{k1}, \dots, T_{km}]$ of T .

Task is **ready** when not yet started but all predecessors are finished

List scheduling : when a processor finishes a task it immediately takes the first ready task from L .
Ties broken by processor ID.

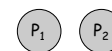
Showed that for any L_T and L'_T :

$$\frac{T(L_T)}{T(L'_T)} = 1 + \frac{n-1}{n}$$

15-853

15

List Scheduling

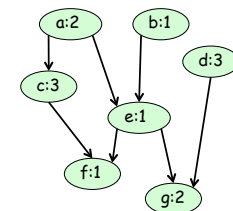


$L = [a, b, c, d, e, f, g]$

Time
↓



$L' = [g, e, a, b, c, d, f]$



The DAG

15-853

Page16

Some definitions

T_p : time on P processors

W : work (total weight of the DAG)

D : span (longest path in the DAG)

Lower bound on time : $T_p \geq \max(W/P, D)$

15-853

17

Greedy Schedules

As schedule is *greedy* if a processor cannot sit idle when a task is ready.

List schedules are greedy.

For any greedy schedule:

$$\text{Efficiency} = \frac{W}{T_p} \geq \frac{PW}{W + D(P - 1)}$$

$$\text{Parallel Time} = T_p < \frac{W}{P} + D$$

15-853

18

Greedy Schedules

Theorem: The time taken by a greedy scheduler is

$$T_p < \frac{W}{P} + D$$

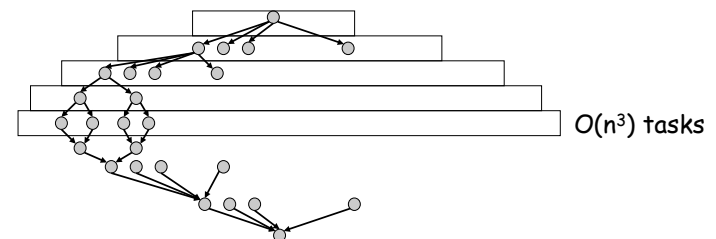
Proof: on board.

15-853

19

Breadth First Schedules

Most naïve schedule. Used by most implementations of P-threads.

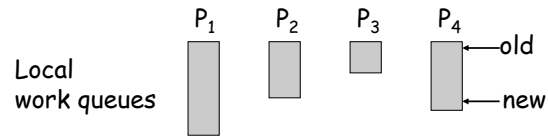


Bad space usage, bad locality

15-853

20

Work Stealing



push new jobs on "new" end

pop jobs from "new" end

If processor runs out of work, then "steal" from another "old" end

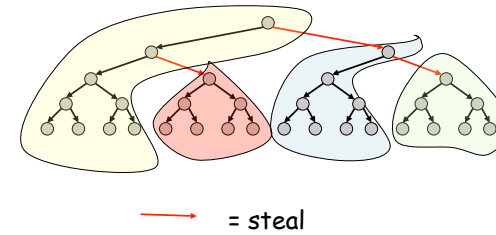
Each processor tends to execute a sequential part of the computation.

15-853

21

Work Stealing

Tends to schedule "sequential blocks" of tasks



15-853

22

Work Stealing Theory

For fork-join computations

of steals = $O(PD)$

Space = $O(PS_1)$

S_1 is the sequential space

cache misses on private caches = $Q_1 + O(MPD)$

Q_1 = sequential misses, M = cache size

15-853

23

Work Stealing Practice

Used in Cilk Scheduler

- Small overheads because common case of pushing/popping from local queue can be made fast (with good data structures and compiler help).
- No contention on a global queue
- Has good distributed cache behavior
- Can indeed require $O(S_1P)$ memory

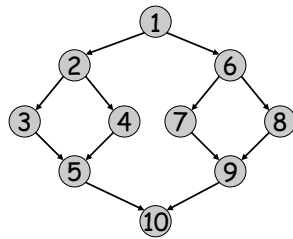
Used in many schedulers

15-853

24

Parallel Depth First Schedules (P-DFS)

List scheduling based on Depth-First ordering



2 processor
schedule

1
2, 6
3, 4
5, 7
8
9
10

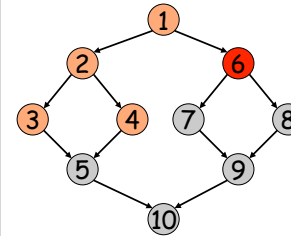
For strict computations a shared stack
implements a P-DFS

15-853

25

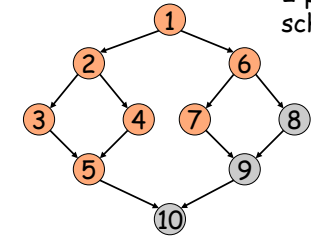
"Premature task" in P-DFS

A running task is premature if there is an earlier
sequential task that is not complete



● = premature

15-853



2 processor
schedule

1
2, 6
3, 4
5, 7
8
9
10

26

P-DFS Theory

For any computation:

Premature nodes at any time = $O(PD)$

Space = $S_1 + O(PD)$

With a shared cache of size $M_1 + O(PD)$

we have $Q_p = Q_1$

15-853

27

P-DFS Practice

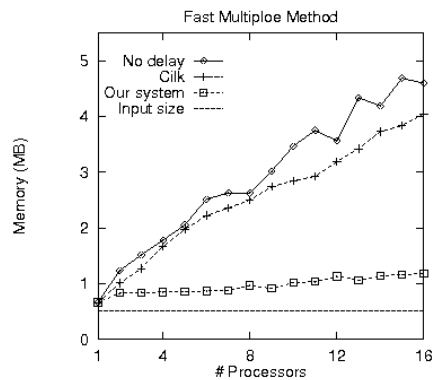
Experimentally uses less memory than work stealing
and performs better on a shared cache.

Requires some "coarsening" to reduce overheads

15-853

28

P-DFS Practice

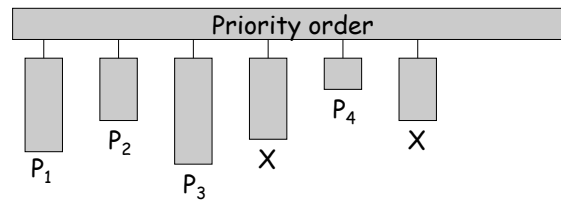


15-853

29

Hybrid Scheduling

Can mix Work Stealing and P-DFS



Gives a way to do automatic coarsening while still getting space benefits of PDF
Also allows suspending a whole Q

15-853

30

Other Scheduling

Various other techniques, but not much theory

e.g.

- Locality guided work stealing
- Affinity guided self-scheduling

Many techniques are for particular form of parallelism

15-853

31