**Edit Distance: Dynamic Programming**

**Sequential code:**

```plaintext
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]);
```

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.
Edit Distance: Dynamic Programming

![Matrix Diagram]

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.

Scheduling

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

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]);
```

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.
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)

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

General

Locks
Transactions
Synch variables
Easy to create deadlock
Hard to schedule

Scheduling Outline

Theoretical results on scheduling
Graham, 1966

Greedy Schedules
Specific greedy schedules
  - Breadth First
  - Work Stealing
  - P-DFS
  - Hybrid
Graham Scheduling

"Bounds on Certain Multiprocessor Anomalies", 1966

Model:
Processing Units: $P_i$, $1 \leq i \leq n$
Tasks: $T = (T_1, \ldots, T_m)$
Partial order: $<_T$ on $T$
Time function: $\mu: T \to [0, \infty]$
$(T, <_T, \mu)$: define a weighted DAG

---

Graham: List Scheduling

For a task set $T$, a Task List $L_T$ is some ordering $[T_{k1}, \ldots, T_{km}]$ 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 $L_{T'}$: $\frac{T(L_T)}{T(L_{T'})} = 1 + \frac{n - 1}{n}$
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: $T_p \geq \max(W/P, D)$

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:

- Efficiency: $\frac{W}{T_p} \geq \frac{PW}{W + D(P - 1)}$
- Parallel Time: $T_p < \frac{W}{P} + D$

Greedy Schedules

Theorem: The time taken by a greedy scheduler is

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

Proof: on board.

Breadth First Schedules

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

- $O(n^3)$ tasks
- Bad space usage, bad locality
Work Stealing

- Local work queues
- 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.

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

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
**Parallel Depth First Schedules (P-DFS)**

List scheduling based on Depth-First ordering

2 processor schedule

For strict computations a shared stack implements a P-DFS

---

**“Premature task” in P-DFS**

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

2 processor schedule

---

**P-DFS Theory**

For any computation:

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

Space = \(S_t + O(PD)\)

With a shared cache of size \(M_t + O(PD)\)

we have \(Q_p = Q_t\)

---

**P-DFS Practice**

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

Requires some “coarsening” to reduce overheads
**P-DFS Practice**

![Graph showing Memory (MB) vs. # Processors for different methods: No delay, GTK, Our System, and Input size.]

**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

**Other Scheduling**

Various other techniques, but not much theory

- Locality guided work stealing
- Affinity guided self-scheduling

Many techniques are for particular form of parallelism