DPHPC: Scheduling / Balance

Recitation session

Reference:
Algorithm Cost

Work and depth can be viewed as the running time of an algorithm at two limits: one processor (work) and an unlimited number of processors (depth).

Brent’s theorem provides bounds to the running time:

$$\frac{W}{P} \leq T \leq \frac{W}{P} + D$$

Defining a DAG

**Strand**: chain of serially executed instructions.

Strands are partially ordered with **dependencies**

**Spawn** nodes have two successors

**Sync** nodes are where the control flow merges
Defining a DAG

Given an input size $n$:
- The **work** $W(n)$ is the total number of strands.
  - $W(n)=13$
- The **depth** $D(n)$ is the length of the critical path (measured in number of strands).
  - Defines the minimum execution time of the computation
  - $D(n)=8$

The ratio $\frac{W(n)}{D(n)}$ measures the average available parallelism
int fib (int n) {
    if (n<2) return (n);
    else {
        int x,y;
        x = spawn fib(n-1);
        y = fib(n-2);
        sync;
        return (x+y);
    }
}

The DAG unfolds dynamically:

Node: Sequence of instructions without call, spawn, sync, return
Edge: Dependency

5 threads
Scheduling a DAG

The DAG unfolds dynamically:

5 threads

Remember oblivious algorithms?
Greedy Scheduler

- **Idea:** Do as much as possible in every step
- **Definition:** A node is ready if all predecessors have been executed

![Diagram of a directed acyclic graph with nodes marked as ready and executed, p = 3]
Greedy Scheduler

- **Idea**: Do as much as possible in every step
- **Definition**: A node is ready if all predecessors have been executed
- **Complete step**:
  - ≥ p nodes are ready
  - run any p

[p = 3]
Greedy Scheduler

- **Idea:** Do as much as possible in every step
- **Definition:** A node is ready if all predecessors have been executed
- **Complete step:**
  - $\geq p$ nodes are ready
  - run any $p$
- **Incomplete step:**
  - $< p$ nodes ready
  - run all

![Diagram of greedy scheduler](image)
Greedy Scheduler

Maintain thread pool of live threads, each is ready or not
- Initial: Root thread in thread pool, all processors idle
- At the beginning of each step each processor is idle or has a thread T to work on
- If idle
  - Get ready thread from pool
- If has thread T
  - Case 0: T has another instruction to execute
    - execute it
  - Case 1: thread T spawns thread S
    - return T to pool, continue with S
  - Case 2: T stalls
    - return T to pool, then idle
  - Case 3: T dies
    - if parent of T has no living children, continue with the parent, otherwise idle
Work Stealing Scheduler

- Each processor maintains a “ready deque:” deque of threads ready for execution; bottom is manipulated as a stack.
Work Stealing Scheduler
Work Stealing Scheduler
WorkStealingScheduler
Work Stealing Scheduler
When a processor runs out of work, it steals a task from the top of a random victim’s deque.
Work Stealing Scheduler
Work Stealing Scheduler
Work Stealing Scheduler

Each processor maintains a ready deque, bottom treated as stack

- **Initial:** Root thread in deque of a random processor
- **Deque not empty:**
  - Processor takes thread T from bottom and starts working
  - T spawns S: Put T on stack, continue with S
  - T stalls: Take next thread from stack
  - T dies: Take next thread from stack
  - If T enables a stalled thread S, S is put on the stack of T's processor
- **Deque empty:**
  - Steal thread from the top of a random (uniformly) processor’s deque
Recap: Balance Principle

Goal when optimizing/building HPC machine:
Minimize time to solution,
\[ \text{time(IO)} = \text{time(comp)} \quad \text{(otherwise we could have built a cheaper machine)} \]

Observation: Flops/second increase faster than Bytes/second read from memory

Solution: Use caches! Their size increases at a similar rate! – Good, but does this help? (Blackboard)
Recap: Assignment

Assume you have a balanced machine to compute the following code on a single processing element:

```
for (i=0..n)
    for (j=0..n)
        a[i,j] = (a[i+1,j]+a[i-1,j]+a[i,j+1]+a[i,j-1]+a[i,j])/5
```

If we increase the floating-point performance by a factor of 2, how much does the cache size M have to be increased to re-balance?