Chapter 4: Parallel Program Structures II

Elements of Parallel Computing

Eric Aubanel

Parallel Loops and Synchronization

parallel for $i \leftarrow 0$ *to* $n-1$ **do**

$c[i] = a[i] + b[i]$

end

- e.g. OpenMP
- implicit barrier at end of loop

Matrix-Vector Multiplication

parallel for each *row i of matrix A* **do**
    $b[i] \leftarrow 0$
    foreach *column j of A* **do**
        $b[i] \leftarrow b[i] + A[i, j] * x[j]$
    end
end

Loop Schedules

Static and **Dynamic**

Static: contiguous chunks

Each thread, $id \in [0..nt)$, executes iterations $\lfloor id * n/nt \rfloor$ to $\lfloor (id + 1) * n/nt \rfloor - 1$

E.g. for: $n = 2048$ and $nt = 5$:

thread 0: $i \leftarrow$ 0 to 408
thread 1: $i \leftarrow$ 409 to 818
thread 2: $i \leftarrow$ 819 to 1227
thread 3: $i \leftarrow$ 1228 to 1637
thread 4: $i \leftarrow$ 1638 to 2047

Loop Schedules

Static: round-robin

Dynamic:

- Master-worker: chunks assigned to each thread. After completing a chunk, thread gets new chunk (OpenMP)
- Recursive division: recursively divide work of loop in half (Cilk Plus)

Fractal

```
// Image coordinates: lower left (xmin,
   ymin) to upper right (xmin + len,
   ymin + len)
// xmin = ymin = −1.5 and len = 3 for full
   image
```

Input: α, n, $xmin$, $ymin$, len
Output: $n \times n$ pixel fractal
Data: niter `// max iterations`
threshold `// threshold for divergence`

$ax \leftarrow len/n$
$ymax \leftarrow ymin + len$

```
for i ← 0 to n − 1 do
    cx ← ax * i + xmin
    for j ← 0 to n − 1 do
        cy ← ymax − ax * j, c ← (cx, cy)
        if α > 0 then  z ← (0,0)  else  z ← (1,1)
        for k ← 1 to niter do
            if |z| < threshold then
                z ← z^α + c
                kount[i, j] ← k
            else
                break // exit inner loop
            end
        end
    end
end
```

Subset Sum

for $i \leftarrow 2$ *to* n **do**
    parallel for $j \leftarrow 1$ *to* S **do**
        $F[i,j] \leftarrow F[i-1,j]$
        if $j \geq s[i]$ **then**
            $F[i,j] \leftarrow F[i,j] \vee F[i-1, j-s[i]]$
        end
    end
end

Shared and Private Variables

Language model may assume variables private by default, or shared by default

parallel for each *row i of matrix A* **do**
    $b[i] \leftarrow 0$
    foreach *column j of A* **do**
        $b[i] \leftarrow b[i] + A[i,j] * x[j]$
    end
end

Either declare A, b, x to be shared, or j to be private

What about the fractal?

Synchronization

- Barrier
- Critical section

Variable Increment Isn't Atomic

```
// Danger, produces indeterminate result!
```

Procedure `iterPi`(n)

$sum \leftarrow 0$

parallel for $i \leftarrow 0$ *to* $n - 1$ **do**

$x \leftarrow$ pseudo-random number $\in [-1, 1]$

$y \leftarrow$ pseudo-random number $\in [-1, 1]$

if $x^2 + y^2 \leq 1$ **then**

$sum \leftarrow sum + 1$

end

end

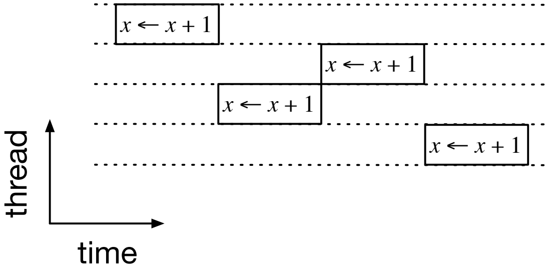
return $sum * 4/n$

end

Critical Section

Provides mutual exclusion

begin critical
$sum \leftarrow sum + 1$
end critical



Locks or Lock-Free

Critical sections can be built using locks, without locks, or avoided all together.

`lock()`
$sum \leftarrow sum + 1$
`unlock()`

- Locks can be tricky to use and can have large overhead
- Lock-free: Use atomic operations supported in hardware

Compare and Swap

atomic Procedure `cas`($\&x$, *old*, *new*)
  if $x = old$ **then**
    $x \leftarrow new$
    return true
  else
    return false
  end
end

repeat
  $old \leftarrow sum$
  $new \leftarrow sum + 1$
until `cas`($\&sum$, *old*, *new*) $=$ true

Input: array a of n nonnegative integers in the range with maximum value $high$.

Output: array a with duplicates removed, with k values.

Data: array t with $m = high + 1$ elements, initialized to 0.

```
parallel for i ← 0 to n − 1 do
    cas(&t[a[i]], 0, 1)
end
k ← 0
for i ← 0 to m − 1 do
    if t[i] = 1 then
        a[k] ← i
        k ← k + 1
    end
end
```

ABA Problem

Procedure `pop()`
  repeat
    $old \leftarrow top$
    $new \leftarrow (top \rightarrow next)$
  until `cas`$(\&top, old, new) = \text{true}$
  return old
end

Stack: $top \rightarrow A \rightarrow B \rightarrow C$:

thread 0: $old \leftarrow top$

thread 0: $new \leftarrow (top \rightarrow next)$

thread 1: $a \leftarrow$ `pop()` $//top \rightarrow B \rightarrow C$

thread 1: $b \leftarrow$ `pop()` $//top \rightarrow C$

thread 1: `push`(a) $//top \rightarrow A \rightarrow C$

thread 0: `cas`$(\&top, old, new)//top \rightarrow B$

Alternative to Critical Section

```
Procedure iterPi(n)
    sum ← 0
    // i, id, x, y private
    parallel for i ← 0 to n − 1 do
        id ← getThreadID()
        x ← pseudo-random number ∈ [−1, 1]
        y ← pseudo-random number ∈ [−1, 1]
        if x² + y² ≤ 1 then
            psum[id] ← psum[id] + 1
        end
    end
    for i ← 0 to nt − 1 do
        sum ← sum + psum[i]
    end
    return sum ∗ 4/n
end
```

Thread Safety

Thread-safe function: can be called by multiple threads without any data races occurring. AKA *re-entrant* function.

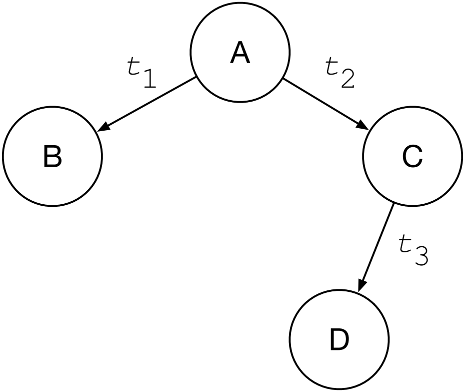
"pseudo-random number $\in [-1, 1]$" must be thread safe!

Guidelines for Parallel Loops

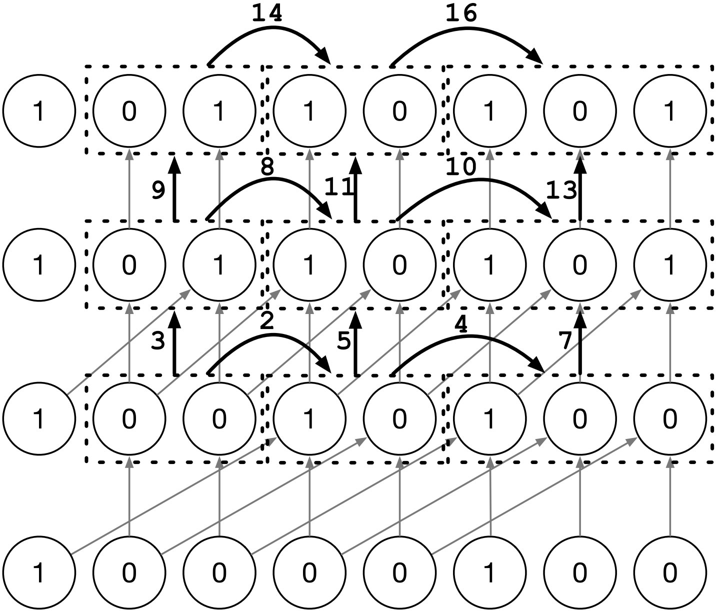
- Eliminate data races
- Load balance with loop schedules

Tasks with Dependencies

spawn `out(`$t1$, $t2$`) A()`
spawn `in(`$t1$`) B()`
spawn `in(`$t2$`) out(`$t3$`) C()`
spawn `in(`$t3$`) D()`



Blocked Subset Sum



Input: Array $s[1..n]$ of n positive integers, target sum S, number nB of blocks per row

Output: returns 1 if a subset that sums to S exists, 0 otherwise

Data: Array $F[1..n, 0..S]$ initialized to 0

for $i \leftarrow 1$ *to* n **do**
    $F[i, 0] \leftarrow 1$
end

$F[1, s[1]] \leftarrow 1$

spawn `out`(2, 3) `calcRowChunk`(2, 1, $\lfloor S/nB \rfloor$)
for $j \leftarrow 2$ *to* nB **do**
    spawn `in`($2(j-1)$) `out`($2j$, $2j+1$)
        `calcRowChunk`(2, $\lfloor (j-1) * S/nB \rfloor + 1$,
        $\lfloor j * S/nB \rfloor$)
end

```
for i ← 3 to n do
    iB ← 2 * (i − 2) * nB + 2
    spawn in(iB − 2 * nB + 1) out(iB, iB + 1)
          calcRowChunk(i, 1, ⌊S/nB⌋)
    for j ← 2 to nB do
        iB ← iB + 2
        spawn in(iB − 2, iB − 2 * nB + 1) out(iB, iB + 1)
              calcRowChunk(i, ⌊(j − 1) * S/nB⌋ + 1,
              ⌊j * S/nB⌋)
    end
end
return F[n, S]

Procedure calcRowChunk(i, j1, j2)
    for j ← j1 to j2 do
        F[i, j] ← F[i − 1, j]
        if j ≥ s[i] then
            F[i, j] ← F[i, j] ∨ F[i − 1, j − s[i]]
        end
end
```