Chapter 4: Parallel Program Structures I

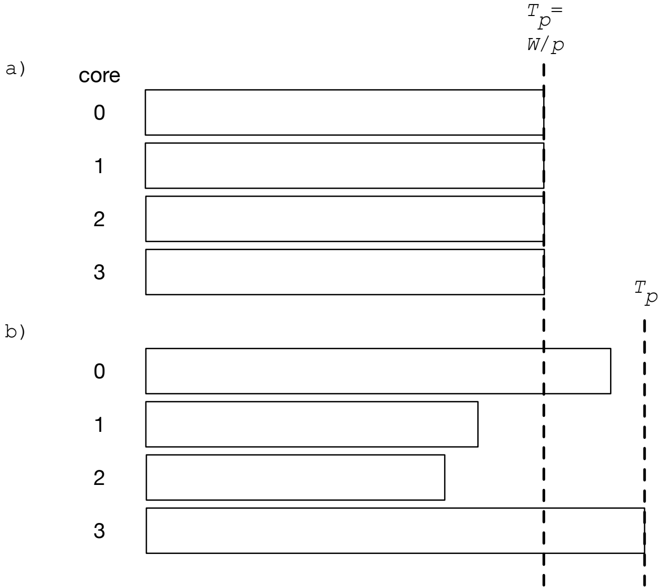
Elements of Parallel Computing

Eric Aubanel

Wall clock time

- start timer as soon as program launched
- stop timer once all cores finish execution

Load Balance



SIMD: Strictly Data Parallel

SIMD execution can be enabled via:

- automatic compiler vectorization
  - better if loops annotated by programmer (pragmas)
- array notation:
  - $c[0:n-1] \leftarrow a[0:n-1] + b[0:n-1]$
  - $c \leftarrow a + b$
- `forall` loops

Strip-Mining

Compiler break down iterations into chunks with a number of elements that depends on the number of bits in the data type

E.g. assume 512 bit vector registers, double precision arithmetic: 2^{20} operations broken into 2^{17} groups of 8

SIMD Notation

We will use set notation:
$\{c[i] \leftarrow a[i] + b[i] : i \in [0..n)\}$

Row-wise SIMD Matrix-Vector Multiplication

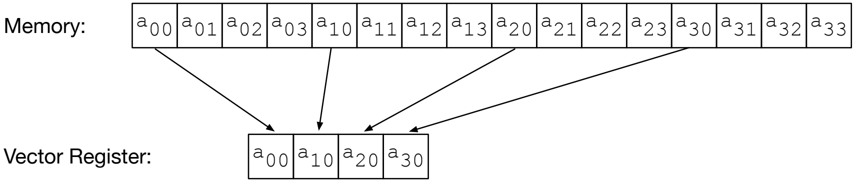
$\{b[i] \leftarrow 0 : i \in [0..n)\}$
for $j \leftarrow 0$ *to* $m - 1$ **do**
    $\{b[i] \leftarrow b[i] + A[i, j] * x[j] : i \in [0..n)\}$
end

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} A_{00}x_0 \\ A_{10}x_0 \\ A_{20}x_0 \\ A_{30}x_0 \end{bmatrix}$$

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} A_{01}x_1 \\ A_{11}x_1 \\ A_{21}x_1 \\ A_{31}x_1 \end{bmatrix}$$

Row-wise SIMD Matrix-Vector Multiplication

- Requires loading of column of matrix
  - not efficient if stored in row-major order



Column-wise Matrix-Vector Multiplication

Inner product of each row of A with b in parallel:
reduction

Reduction

Use SIMD *conditional execution*:

for $k \leftarrow 0$ *to* $\log n - 1$ **do**

$j \leftarrow 2^k$

$\{a[i] \leftarrow a[i] + a[i+j] : i \in [0..n) \mid i \bmod 2j = 0\}$

end

`// result in` $a[0]$

Control Divergence

Conditional execution can reduce performance

E.g. say SIMD width stores 4 elements of *a*

- first stage of reduction ($k = 0$): only two additions could be done in parallel
- other stages: additions would all be serialized.

Change Indexing

for $k \leftarrow 0$ *to* $\log n - 1$ **do**
$$j \leftarrow 2^{k+1}$$
$$\{a[i*j] \leftarrow a[i*j] + a[i*j+j/2] : i \in [0..n/j)\}$$
end

...but has scattered memory access pattern

Divergence-Free Reduction

Change order of additions:

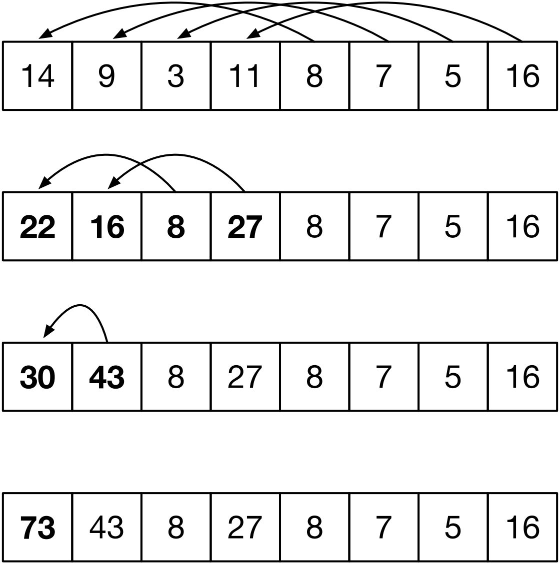
for $k \leftarrow \log n - 1$ *to* 0 **do**

$j \leftarrow 2^k$

$\{a[i] \leftarrow a[i] + a[i+j] : i \in [0..j)\}$

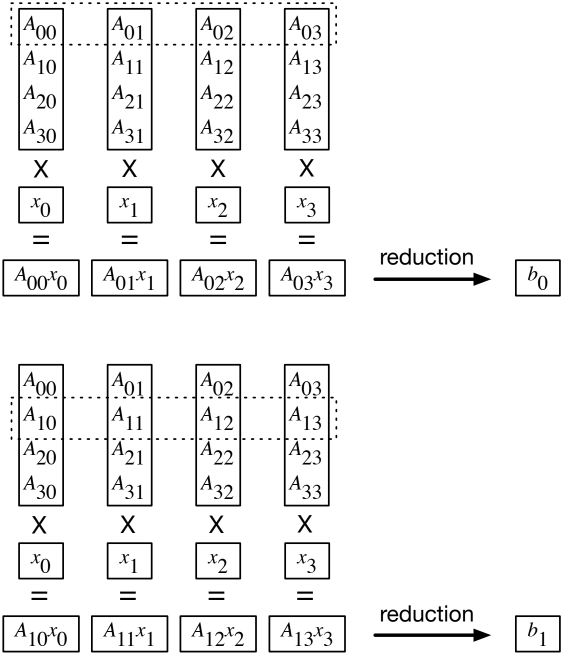
end

`// result in` $a[0]$


14 9 3 11 8 7 5 16
22 16 8 27 8 7 5 16
30 43 8 27 8 7 5 16
73 43 8 27 8 7 5 16

Column-wise Matrix-Vector Multiplication

$\{b[i] \leftarrow 0 : i \in [0..n)\}$
for $i \leftarrow 0$ *to* $n-1$ **do**
  $\{temp[j] \leftarrow A[i,j] * x[j] : j \in [0..m)\}$
  for $k \leftarrow \log n - 1$ *to* 0 **do**
    $\{temp[j] \leftarrow temp[j] + temp[j+2^k] : j \in [0..2^k)\}$
  end
  $b[i] \leftarrow temp[0]$
end


A_{00} A_{01} A_{02} A_{03}
A_{10} A_{11} A_{12} A_{13}
A_{20} A_{21} A_{22} A_{23}
A_{30} A_{31} A_{32} A_{33}
X X X X
x_0 x_1 x_2 x_3
= = = =
$A_{00}x_0$ $A_{01}x_1$ $A_{02}x_2$ $A_{03}x_3$
reduction
b_0
A_{00} A_{01} A_{02} A_{03}
A_{10} A_{11} A_{12} A_{13}
A_{20} A_{21} A_{22} A_{23}
A_{30} A_{31} A_{32} A_{33}
X X X X
x_0 x_1 x_2 x_3
= = = =
$A_{10}x_0$ $A_{11}x_1$ $A_{12}x_2$ $A_{13}x_3$
reduction
b_1

SIMD Subset Sum

for $i \leftarrow 2$ *to* n **do**
  $\{F[i,j] \leftarrow F[i-1,j] : j \in [1..S]\}$
  $\{F[i,j] \leftarrow F[i,j] \vee F[i-1, j-s[i]] : j \in [1..S] \mid j \geq s[i]\}$
end

Control divergence only for chunks where $j < s[i]$ and $j \geq s[i]$

SIMD Guidelines

1. Watch out for dependencies
2. Avoid control divergence
3. Optimize memory access patterns

Shared Memory Programming

- Fork-Join
- Parallel loops
- Tasks with dependencies
- Single Program Multiple Data (SPMD)

Threads

Thread:

- has its own program counter and private memory region in its stack frame
- shares instructions, heap and global memory with other threads.
- has a thread id

Fork-Join

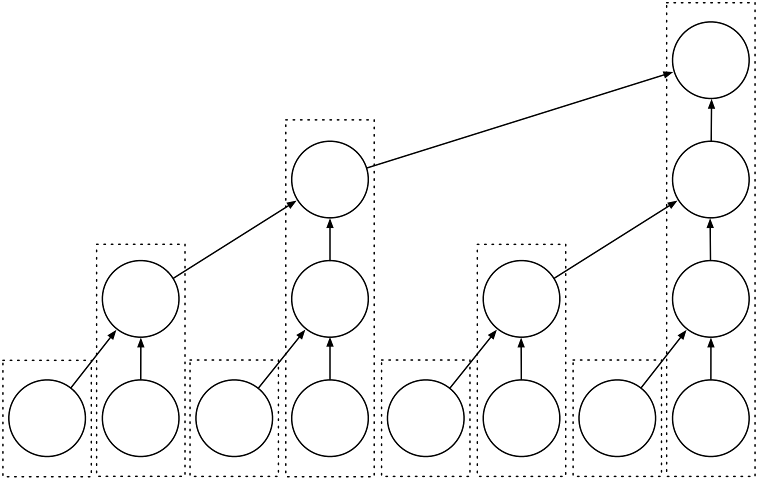
- Child tasks are *forked* by parent
- Parents may need to wait for children to complete
  - called *joining*
- Mainly used by recursively creating tasks

Recursive Estimation of Pi

```
// n = 2^k experiments
Procedure estimatePi(n)
    return recPi(n)*4/n
end

// returns sum of points in circle
Procedure recPi(n)
    if n = 1 then
        sum ← 0
        x ← pseudo-random number ∈ [−1, 1]
        y ← pseudo-random number ∈ [−1, 1]
        if x^2 + y^2 ≤ 1 then sum ← 1
    else
        sum ← recPi(n/2) + recPi(n/2)
    end
    return sum
end
```

Reduction Task Graph Mapped to Threads



Fork Tasks not Threads

Fork and join tasks and let runtime system assign them to threads.

- **spawn** = fork
- **sync** = join

$sum1 \leftarrow$ **spawn** `recPi(`$n/2$`)`
$sum2 \leftarrow$ `recPi(`$n/2$`)`
sync
$sum \leftarrow sum1 + sum2$

Sequential Cutoff

- Too few tasks: difficult to balance work across threads
- Too many tasks: significant runtime overhead

Choose base case to limit number of tasks

```
Procedure recPi(n)
    if n < cutoff then
        sum ← 0
        for i ← 1 to n do
            x ← pseudo-random number ∈ [−1, 1]
            y ← pseudo-random number ∈ [−1, 1]
            if x^2 + y^2 ≤ 1 then  sum ← sum + 1
        end
    else
        sum1 ← spawn  recPi(n/2)
        sum2 ← recPi(n/2)
        sync
        sum ← sum1 + sum2
    end
    return sum
end
```

Merge Sort

- Use fork-join for both splitting and merging
- Also use sequential cutoffs for both
- split the merge:
    - find the median *mid* of longest of the sorted subarrays
    - binary search of the other subarray to find the index that splits it into elements less than and greater than *mid*

Input: array a of length n.
Output: array b, containing array a sorted. Array a overwritten.

```
arrayCopy(a, b)  // copy array a to b
parMergeSort(a, 0, n, b)

// sort elements with index i ∈ [lower..upper)
Procedure parMergeSort(a, lower, upper, b)
    if (upper − lower) < cutoff then
        sequentialSort(a, lower, upper, b)
    else
        mid ← ⌊(upper + lower)/2⌋
        spawn parMergeSort(b, lower, mid, a)
        parMergeSort(b, mid, upper, a)
        sync
        parMerge(a, lower, mid, mid, upper, b, lower)
        return
    end
end
```

Parallel Merge

- parallel merge of sorted sub-arrays $i \in [low1..up1)$ and $i \in [low2..up2)$ of a into b starting at index *start*
- two indices for each of lower and upper subarrays, since they won't always be contiguous

Procedure `parMerge`(a, $low1$, $up1$, $low2$, $up2$, b, $start$)

$k1 \leftarrow up1 - low1$, $k2 \leftarrow up2 - low2$

if $k1 + k2 < \text{cutoff}$ **then**

`sequentialMerge`(a, $low1$, $up1$, $low2$, $up2$, b, $start$)

else

...

end

end

```
if k1 + k2 < cutoff then
    sequentialMerge(a, low1, up1, low2, up2, b, start)
else
    if k1 ≥ k2 then
        mid1 ← ⌊(low1 + up1 − 1)/2⌋
        // mid2: first index in [low2, up2) such
           that a[index] > a[mid1]
        mid2 ←binarySearch(a, low2, up2, mid1)
    else
        mid2 ← ⌊(low2 + up2 − 1)/2⌋
        mid1 ←binarySearch(a, low1, up1, mid2)−1
        mid2 ← mid2 + 1
    end
    spawn parMerge(a, low1, mid1 + 1, low2, mid2, b,
    start)
    parMerge(a, mid1 + 1, up1, mid2, up2, b,
    start + mid1 − low1 + 1 + mid2 − low2)
    sync
end
```

Fork-Join Guidelines

- Don't create more independent tasks than necessary
- Use a sequential cutoff to limit the depth of the recursion
- Avoid unnecessary allocation of memory
- Be careful if shared read/write access to data required
  - **Warning**: may be hidden inside functions