LECTURE 2
PARALLEL ALGORITHMS

Julian Shun

February 9, 2022

Lecture material taken from “Parallel Algorithms” by Guy Blelloch and Bruce Maggs and 6.172, developed by Charles Leiserson and Saman Amarasinghe

© 2018–2023 MIT Algorithm Engineering Instructors
Announcement

- Problem set will be released on Canvas this week, due on Monday 3/6
Multicore Processors

Q Why do semiconductor vendors provide chips with multiple processor cores?

A Because of Moore’s Law and the end of the scaling of clock frequency.

Intel Haswell–E

Slide adapted from 6.172 (Charles Leiserson and Saman Amarasinghe)

© 2018–2023 MIT Algorithm Engineering Instructors
Technology Scaling

Clock speed is bounded at ~4GHz.

Slide adapted from 6.172 (Charles Leiserson and Saman Amarasinghe)
Projected power density, if clock frequency had continued its trend of scaling 25%–30% per year.

Each generation of Moore’s Law potentially doubles the number of cores.
Parallel Languages

- Pthreads
- Cilk, OpenMP
- Message Passing Interface (MPI)
- CUDA, OpenCL

Today: Shared-memory parallelism

- Cilk and OpenMP are extensions of C/C++ that support parallel for-loops, parallel recursive calls, etc.
- Do not need to worry about assigning tasks to processors as these languages have a runtime scheduler
- Cilk has a provably efficient runtime scheduler
PARALLELISM MODELS
Basic multiprocessor models

- **Local memory machine**
- **Modular memory machine**
- **Parallel random-access Machine (PRAM)**

Source: “Parallel Algorithms” by Guy E. Blelloch and Bruce M. Maggs
Network topology

Bus

Mesh

2-level multistage network

Hypercube

Fat tree

Source: “Parallel Algorithms” by Guy E. Blelloch and Bruce M. Maggs

© 2018–2023 MIT Algorithm Engineering Instructors
Network topology

• Algorithms for specific topologies can be complicated
  • May not perform well on other networks
• Alternative: use a model that summarizes latency and bandwidth of network
  • Postal model
  • Bulk–Synchronous Parallel (BSP) model
  • LogP model
PRAM Model

• All processors can perform same local instructions as in the RAM model
• All processors operate in lock–step
• Implicit synchronization between steps
• Models for concurrent access
  • Exclusive–read exclusive–write (EREW)
  • Concurrent–read concurrent–write (CRCW)
    ■ How to resolve concurrent writes: arbitrary value, value from lowest–ID processor, logical OR of values, sum of values
  • Concurrent–read exclusive–write (CREW)
  • Queue–read queue–write (QRQW)
    ■ Allows concurrent access in time proportional to the maximal number of concurrent accesses
• Work = number of vertices in graph (number of operations)
• Span (Depth) = longest directed path in graph (dependence length)
• Parallelism = Work / Span
• A work-efficient parallel algorithm has work that asymptotically matches the best sequential algorithm for the problem

Goal: work-efficient and low (polylogarithmic) span parallel algorithms
Work–Span model

• Spawning/forking tasks
  • Model can support either binary forking or arbitrary forking
  • Cilk uses binary forking, as seen in 6.172
  • Converting between the two changes work by at most a constant factor and span by at most a logarithmic factor
    ■ Keep this in mind when reading textbooks/papers on parallel algorithms
  • We will assume arbitrary forking unless specified
Work-Span model

- State what operations are supported
  - Concurrent reads/writes?
  - Resolving concurrent writes
Scheduling

- For a computation with work \( W \) and span \( S \), on \( P \) processors a greedy scheduler achieves

\[
\text{Running time} \leq \frac{W}{P} + S
\]

- For a computation with work \( W \) and span \( S \), on \( P \) processors Cilk’s work–stealing scheduler achieves

\[
\text{Expected running time} \leq \frac{W}{P} + O(S)
\]

- Work–efficiency is important since \( P \) and \( S \) are usually small
PARALLEL SUM
Parallel Sum

- Definition: Given a sequence $A = [x_0, x_1, \ldots, x_{n-1}]$, return $x_0 + x_1 + \ldots + x_{n-2} + x_{n-1}$

What is the span?
$S(n) = S(n/2) + O(1)$
$S(1) = O(1)$
$\Rightarrow S(n) = O(\log n)$

What is the work?
$W(n) = W(n/2) + O(n)$
$W(1) = O(1)$
$\Rightarrow W(n) = O(n)$
PREFIX SUM
Prefix Sum

- Definition: Given a sequence $A = [x_0, x_1, \ldots, x_{n-1}]$, return a sequence where each location stores the sum of everything before it in $A$, $[0, x_0, x_0 + x_1, \ldots, x_0 + x_1 + \ldots + x_{n-2}]$, as well as the total sum $x_0 + x_1 + \ldots + x_{n-2} + x_{n-1}$

- Example:

  $\begin{array}{cccccc}
  2 & 4 & 3 & 1 & 3 \\
  \end{array}$

  $\begin{array}{cccccc}
  0 & 2 & 6 & 9 & 10 \\
  \end{array}$

  Total sum = 13

- Can be generalized to any associative binary operator (e.g., $\times$, $\min$, $\max$)
Sequential Prefix Sum

Input: array A of length n
Output: array A’ and total sum

cumulativeSum = 0;
for i=0 to n−1:
    A’[i] = cumulativeSum;
    cumulativeSum += A[i];
return A’ and cumulativeSum

- What is the work of this algorithm?
  - O(n)

- Can we execute iterations in parallel?
  - Loop carried dependence: value of cumulativeSum depends on previous iterations
Parallel Prefix Sum

\[ A = \begin{array}{cccccccc}
  x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\
\end{array} \]

\[ B = \begin{array}{cccccccc}
  & x_0 + x_1 & & x_2 + x_3 & & x_4 + x_5 & & x_6 + x_7 \\
\end{array} \]

Recursively compute prefix sum on \( B \)

\[ B' = \begin{array}{cccccccc}
  0 & x_0 + x_1 & x_0 + \ldots + x_3 & x_0 + \ldots + x_5 & & & \\
\end{array} \]

\[ A' = \begin{array}{cccccccc}
  0 & x_0 & x_0 + x_1 & x_0 + \ldots + x_2 & x_0 + \ldots + x_3 & x_0 + \ldots + x_4 & x_0 + \ldots + x_5 & x_0 + \ldots + x_6 \\
\end{array} \]

i=0 i=1 i=2 i=3 i=4 i=5 i=6 i=7

for even values of \( i \): \( A'[i] = B'[i/2] \)
for odd values of \( i \): \( A'[i] = B'[\lceil(i-1)/2\rceil] + A[i-1] \)

Total sum = \( x_0 + \ldots + x_7 \)
Parallel Prefix Sum

Input: array A of length n (assume n is a power of 2)
Output: array A’ and total sum

PrefixSum(A, n):
    if n == 1: return ([0], A[0])
    for i=0 to n/2-1 in parallel:
    (B’, sum) = PrefixSum(B, n/2)
    for i=0 to n-1 in parallel:
        if (i mod 2) == 0: A’[i] = B'[i/2]
        else: A’[i] = B’[(i-1)/2] + A[i-1]
    return (A’, sum)

What is the span?
S(n) = S(n/2)+O(1)
S(1) = O(1)
→ S(n) = O(log n)

What is the work?
W(n) = W(n/2)+O(n)
W(1) = O(1)
→ W(n) = O(n)
FILTER
Filter

• Definition: Given a sequence $A = [x_0, x_1, \ldots, x_{n-1}]$ and a Boolean array of flags $B[b_0, b_1, \ldots, b_{n-1}]$, output an array $A'$ containing just the elements $A[i]$ where $B[i] = \text{true}$ (maintaining relative order)

• Example:

$$
\begin{array}{cccccc}
A &=& 2 & 4 & 3 & 1 & 3 \\
B &=& T & F & T & T & T & F \\
\end{array}
$$

$$
A' = 2 \ 3 \ 1
$$

• Can you implement filter using prefix sum?
Filter Implementation

A = \begin{bmatrix} 2 & 4 & 3 & 1 & 3 \end{bmatrix}

B = \begin{bmatrix} T & F & T & T & F \end{bmatrix}

//Assume B'[n] = total sum
parallel-for i=0 to n-1:
  if(B'[i] != B'[i+1]):
    A'[B'[i]] = A[i];

Allocate array of size 3

B' = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \end{bmatrix}

Prefix sum

Total sum = 3

A' = \begin{bmatrix} 2 & 3 & 1 \end{bmatrix}
PARALLEL BREADTH-FIRST SEARCH
Parallel BFS Algorithm

- Can process each frontier in parallel
  - Parallelize over both the vertices and their outgoing edges
Parallel BFS Code

BFS(Offsets, Edges, source) {
    parent, frontier, frontierNext, and degrees are array
    parallel_for(int i=0; i<n; i++) parent[i] = -1;
    frontier[0] = source, frontierSize = 1, parent[source] = source;

    while(frontierSize > 0) {
        parallel_for(int i=0; i<frontierSize; i++) {
            degrees[i] = Offsets[frontier[i]+1] − Offsets[frontier[i]];
            perform prefix sum on degrees array
        }
        parallel_for(int i=0; i<frontierSize; i++) {
            v = frontier[i], index = degrees[i], d = Offsets[v+1]−Offsets[v];
            for(int j=0; j<d; j++) {
                //can be parallel
                ngh = Edges[Offsets[v]+j];
                if(parent[ngh] == -1 && compare−and−swap(&parent[ngh], -1, v)) {
                    frontierNext[index+j] = ngh;
                } else { frontierNext[index+j] = -1; }
            }
        }
        filter out “−1” from frontierNext, store in frontier, and update frontierSize to be
        the size of frontier (all done using prefix sum)
    }
}
BFS Work–Span Analysis

- Number of iterations $\leq$ diameter $\Delta$ of graph
- Each iteration takes $O(\log m)$ span for prefix sum and filter (assuming inner loop is parallelized)

$$\text{Span} = O(\Delta \log m)$$

- Sum of frontier sizes $= n$
- Each edge traversed once $\rightarrow m$ total visits
- Work of prefix sum on each iteration is proportional to frontier size $\rightarrow \Theta(n)$ total
- Work of filter on each iteration is proportional to number of edges traversed $\rightarrow \Theta(m)$ total

$$\text{Work} = \Theta(n+m)$$
Performance of Parallel BFS

- Random graph with $n=10^7$ and $m=10^8$
  - 10 edges per vertex
- 40–core machine with 2–way hyperthreading
- 31.8x speedup on 40 cores with hyperthreading
- Sequential BFS is 54% faster than parallel BFS on 1 thread
POINTER JUMPING AND LIST RANKING
**Pointer Jumping**

- Have every node in linked list or rooted tree point to the end (root)

```plaintext
for j = 0 to ceil(log n) - 1:
  parallel-for i = 0 to n - 1:
    temp[i] = P[P[i]];
  parallel-for i = 0 to n - 1:
    P[i] = temp[i];
```

What is the work and span?

- Work: \( W = O(n \log n) \)
- Span: \( S = O(\log n) \)
List Ranking

• Have every node in linked list determine its distance to the end

```plaintext
parallel-for i=0 to n-1:
    if P[i] == i then rank[i] = 0
    else rank[i] = 1
for j=0 to ceil(log n)-1:
    temp, temp2;
    parallel-for i=0 to n-1:
        temp[i] = rank[P[i]];
        temp2[i] = P[P[i]];
    parallel-for i=0 to n-1:
        rank[i] = rank[i] + temp[i];
        P[i] = temp2[i];
```

© 2018–2023 MIT Algorithm Engineering Instructors
Work–Span Analysis

parallel-for i=0 to n-1:
  if P[i] == i then rank[i] = 0
  else rank[i] = 1

for j=0 to ceil(log n)-1:
temp, temp2;
parallel-for i=0 to n-1:
temp = rank[P[i]];
temp2 = P[P[i]];
parallel-for i=0 to n-1:
  rank[i] = rank[i] + temp;
P[i] = temp2;

What is the work and span?

$W = O(n \log n)$
$S = O(\log n)$

Sequential algorithm only requires $O(n)$ work
ListRanking(list P)
1. If list has two or fewer nodes, then return //base case
2. Every node flips a fair coin
3. For each vertex u (except the last vertex), if u flipped Tails and P[u] flipped Heads then u will be paired with P[u]
   A. rank[u] = rank[u] + rank[P[u]]
   B. P[u] = P[P[u]]
4. Recursively call ListRanking on smaller list
5. Insert contracted nodes v back into list with rank[v] = rank[v] + rank[P[v]]
Work–Efficient List Ranking

Apply recursively

Contract

Expand
Work–Span Analysis

- Number of pairs per round is \((n-1)/4\) in expectation
  - For all nodes \(u\) except for the last node, probability of \(u\) flipping Tails and \(P[u]\) flipping Heads is 1/4
  - Linearity of expectations gives \((n-1)/4\) pairs overall
- Each round takes linear work and \(O(1)\) span
- Expected work: \(W(n) \leq W(7n/8) + O(n)\)
- Expected span: \(S(n) \leq S(7n/8) + O(1)\)

\[
W = O(n) \\
S = O(\log n)
\]

- Can show span with high probability with Chernoff bound