Lecture 2
Parallel Algorithms

Julian Shun

February 4, 2022

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

© 2018–2022 Julian Shun
• Presentation sign-up sheet posted on Piazza
• Problem set has been released on Canvas, due on 2/28
Why do semiconductor vendors provide chips with multiple processor cores?

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

Transistor count is still rising, ...

but clock speed is bounded at ~4GHz.
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 supports 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**
  - Interconnection Network
  - Processors: $P_1, P_2, P_3, \ldots, P_n$
  - Memory: $M_1, M_2, M_3, M_n$

- **Modular memory machine**
  - Interconnection Network
  - Processors: $P_1, P_2, P_3, \ldots, P_n$
  - Memory: $M_1, M_2, M_3, M_4, M_m$

- **Parallel random-access Machine (PRAM)**
  - Interconnection Network
  - Processors: $P_1, P_2, P_3, \ldots, P_n$
  - Shared Memory
Network topology

Bus

\[ P_1 \quad P_2 \quad P_3 \quad \ldots \quad P_n \]

Mesh

Hypercube

2-level multistage network

Memory modules

Level 2 (output switches)

Level 1 (input switches)

Processors

Fat tree

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

• Similar to PRAM but does not require lock–step or processor allocation

Computation graph

• 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
• 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

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

- Work–efficiency is important since $P$ and $S$ are usually small
**Idea:** Do as much as possible on every step.

**Definition.** A task is *ready* if all its predecessors have executed.

Slide adapted from 6.172 (Charles Leiserson and Saman Amarasinghe)
Greedy Scheduling

**IDEA:** Do as much as possible on every step.

**Definition.** A task is **ready** if all its predecessors have executed.

**Complete step**
- $\geq P$ tasks ready.
- Run any $P$. 

Slide adapted from 6.172 (Charles Leiserson and Saman Amarasinghe)
**Greedy Scheduling**

**IDEA:** Do as much as possible on every step.

**Definition.** A task is *ready* if all its predecessors have executed.

**Complete step**
- $\geq P$ tasks ready.
- Run any $P$.

**Incomplete step**
- $< P$ tasks ready.
- Run all of them.

---

*IDEA:* Do as much as possible on every step.

**Definition.** A task is *ready* if all its predecessors have executed.

**Complete step**
- $\geq P$ tasks ready.
- Run any $P$.

**Incomplete step**
- $< P$ tasks ready.
- Run all of them.

---

*IDEA:* Do as much as possible on every step.

**Definition.** A task is *ready* if all its predecessors have executed.

**Complete step**
- $\geq P$ tasks ready.
- Run any $P$.

**Incomplete step**
- $< P$ tasks ready.
- Run all of them.

---

*IDEA:* Do as much as possible on every step.

**Definition.** A task is *ready* if all its predecessors have executed.

**Complete step**
- $\geq P$ tasks ready.
- Run any $P$.

**Incomplete step**
- $< P$ tasks ready.
- Run all of them.

---

*IDEA:* Do as much as possible on every step.

**Definition.** A task is *ready* if all its predecessors have executed.

**Complete step**
- $\geq P$ tasks ready.
- Run any $P$.

**Incomplete step**
- $< P$ tasks ready.
- Run all of them.

---

*IDEA:* Do as much as possible on every step.

**Definition.** A task is *ready* if all its predecessors have executed.

**Complete step**
- $\geq P$ tasks ready.
- Run any $P$.

**Incomplete step**
- $< P$ tasks ready.
- Run all of them.
Theorem [G68, B75, EZL89]. Any greedy scheduler achieves

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

Proof.

- \# complete steps \( \leq \frac{W}{P} \), since each complete step performs \( P \) work.
- \# incomplete steps \( \leq S \), since each incomplete step reduces the span of the unexecuted dag by 1. ■
Cilk Scheduling

• For a computation with work $W$ and span $S$, on $P$ processors Cilk’s work–stealing scheduler achieves

$$\text{Expected running time } \leq \frac{W}{P} + O(S)$$
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:**
  
  \[
  \begin{array}{lllll}
  2 & 4 & 3 & 1 & 3 \\
  0 & 2 & 6 & 9 & 10
  \end{array}
  \]

  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

\[ A = x_0 \quad x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad x_7 \]

\[ B = x_0 + x_1 \quad x_2 + x_3 \quad x_4 + x_5 \quad x_6 + x_7 \]

Recursively compute prefix sum on \( B \)

\[ B' = 0 \quad x_0 + x_1 \quad x_0 + \ldots + x_3 \quad x_0 + \ldots + x_5 \]

\[ A' = 0 \quad x_0 \quad x_0 + x_1 \quad x_0 + \ldots + x_2 \quad x_0 + \ldots + x_3 \quad x_0 + \ldots + x_4 \quad x_0 + \ldots + x_5 \quad x_0 + \ldots + x_6 \]

for even values of \( i \): \( A'[i] = B'[i/2] \)

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

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

• 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:

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
& 2 & 4 & 3 & 1 & 3 \\
A = & \text{T} & \text{F} & \text{T} & \text{T} & \text{F} \\
B = & 2 & 3 & 1 \\
A' = & 2 & 3 & 1 \\
\end{array}
\]

• Can you implement filter using prefix sum?
Filter Implementation

\[
A = \begin{bmatrix} 2 & 4 & 3 & 1 & 3 \end{bmatrix} \quad B = \begin{bmatrix} T & F & T & T & F \end{bmatrix}
\]

// 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 arrays
  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++) {
        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)
  }
}

© 2018–2022 Julian Shun
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)

\[
\text{for } j=0 \text{ to } \lceil \log n \rceil - 1: \\
\text{parallel-for } i=0 \text{ to } n-1: \\
\text{temp} = P[P[i]]; \\
\text{parallel-for } i=0 \text{ to } n-1: \\
P[i] = \text{temp};
\]

What is the work and span?

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

Source: “Parallel Algorithms” by Guy E. Blelloch and Bruce M. Maggs
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 = rank[P[i]];
        temp2 = P[P[i]];
    parallel-for i=0 to n-1:
        rank[i] = rank[i] + temp;
        P[i] = temp2;
```

© 2018–2022 Julian Shun
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: \(S(n) \leq S(7n/8) + O(1)\)

\[
\begin{align*}
W &= O(n) \\
S &= O(\log n)
\end{align*}
\]

• Can show span with high probability with Chernoff bound
CONNECTED COMPONENTS
Connected Components

• Given an undirected graph, label all vertices such that \( L(u) = L(v) \) if and only if there is a path between \( u \) and \( v \)

• BFS span is proportional to diameter
  • Works well for graphs with small diameter

• Today we will see a randomized algorithm that takes \( O((n+m)\log n) \) work and \( O(\log n) \) span
  • Deterministic version in paper
  • We will study a work-efficient parallel algorithm next week
Random Mate

- Idea: Form a set of non-overlapping star subgraphs and contract them
- Each vertex flips a coin. For each Heads vertex, pick an arbitrary Tails neighbor (if there is one) and point to it

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

Form stars

Contract

Repeat until each component has a single vertex

Expand vertices back in reverse order with label of neighbor

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

CC_Random_Mate(L, E)
if(|E| = 0) Return L  // base case
else
  1. Flip coins for all vertices
  2. For v where coin(v)=Heads, hook to arbitrary Tails neighbor w and set L(v) = w
  3. E’ = { (L(u),L(v)) | (u,v) ∈ E and L(u) ≠ L(v) }
  4. L’ = CC_Random_Mate(L, E’)
  5. For v where coin(v)=Heads, set L’(v) = L’(w) where w is the Tails neighbor that v hooked to in Step 2
  6. Return L’

- Each iteration requires O(m+n) work and O(1) span
  - Assumes we do not pack vertices and edges
- Each iteration eliminates 1/4 of the vertices in expectation
  \[ W = O((m+n)\log n) \text{ w.h.p.} \]
  \[ S = O(\log n) \text{ w.h.p.} \]
(Minimum) Spanning Forest

- Spanning Forest: Keep track of edges used for hooking
  - Edges will only hook two components that are not yet connected

- Minimum Spanning Forest:
  - For each “Heads” vertex \( v \), instead of picking an arbitrary neighbor to hook to, pick neighbor \( w \) where \((v, w)\) is the minimum weight edge incident to \( v \)
  - Can find this edge using priority concurrent write
Minimum Spanning Forest

Form stars with min-weight edge

Contract

Repeat
Parallel Bellman–Ford
Bellman–Ford Algorithm

Bellman–Ford(G, source):

ShortestPaths = {∞, ∞, ..., ∞} //size n; stores shortest path distances
ShortestPaths[source] = 0
for i=1 to n-1:
    parallel for each vertex v in G:
        parallel for each w in neighbors(v):
            writeMin(&ShortestPaths[w], ShortestPaths[v] + weight(v,w))

if no shortest paths changed:
    return ShortestPaths
report “negative cycle”

• What is the work and span assuming writeMin has unit cost?
• Work = O(mn)
• Span = O(n)
QUICKSORT
Parallel Quicksort

static void quicksort(int64_t *left, int64_t *right)
{
    int64_t *p;
    if (left == right) return;
    p = partition(left, right);
    cilk_spawn quicksort(left, p);
    quicksort(p + 1, right);
    cilk_sync;
}

• Partition picks random pivot \( p \) and splits elements into left and right subarrays
• Partition can be implemented using prefix sum in linear work and logarithmic span
• Overall work is \( O(n \log n) \)
• What is the span?

Slide adapted from 6.172 (Charles Leiserson and Saman Amarasinghe)
Parallel Quicksort Span

- Pivot is chosen uniformly at random
- 1/2 chance that pivot falls in middle range, in which case sub-problem size is at most 3n/4
- Expected span:
  - $S(n) \leq \frac{1}{2} S(\frac{3n}{4}) + O(\log n)$
  - $= O(\log^2 n)$
- Can get high probability bound with Chernoff bound
RADIX SORT
Radix Sort

- Consider 1-bit digits

Radix_sort(A, b) // b is the number of bits of A
For i from 0 to b-1: // sort by i’th most significant bit
  Flags = { (a >> i) mod 2 | a ∈ A }
  NotFlags = { !(a >> i) mod 2 | a ∈ A }
  (sum₀, R₀) = prefixSum(NotFlags)
  (sum₁, R₁) = prefixSum(Flags)
Parallel-for j = 0 to |A|-1:
  if(Flags[j] = 0): A'[R₀[j]] = A[j]
A = A'

A =  
| 1 | 2 | 6 | 5 | 4 | 3 |

Flags =  
| 1 | 0 | 0 | 1 | 0 | 1 |

NotFlags =  
| 0 | 1 | 1 | 0 | 1 | 0 |

A' =  
| 2 | 6 | 4 | 1 | 5 | 3 |

R₀ =  
| 0 | 0 | 1 | 2 | 2 | 3 |

R₁ =  
| 0 | 1 | 1 | 1 | 1 | 2 | 2 |

sum₀ = 3

- Each iteration is stable
Work–Span Analysis

Radix_sort(A, b) //b is the number of bits of A
For i from 0 to b−1:
    Flags = { (a >> i) mod 2 | a ∈ A }
    NotFlags = { !(a >> i) mod 2 | a ∈ A }
    (sum₀, R₀) = prefixSum(NotFlags)
    (sum₁, R₁) = prefixSum(Flags)
    Parallel–for j = 0 to |A|−1:
        if(Flags[j] = 0):  A'[R₀[j]] = A[j]
    A = A'

• Each iteration requires O(n) work and O(log n) span
• Overall work = O(bn)
• Overall span = O(b log n)
Removing Duplicates
Removing Duplicates with Hashing

- Given an array $A$ of $n$ elements, output the elements in $A$ excluding duplicates

Construct a table $T$ of size $m$, where $m$ is the next prime after $2n$

1. $i = 0$
2. While ($|A| > 0$)
   1. Parallel–for each element $j$ in $A$ try to insert $j$ into $T$ at location $(\text{hash}(A[j], i) \mod m)$ //if the location was empty at the beginning of round $i$, and there are concurrent writes then an arbitrary one succeeds
   2. Filter out elements $j$ in $A$ such that $T[(\text{hash}(A[j], i) \mod m)] = A[j]$
   3. $i = i + 1$

- Use a new hash function on each round
- Claim: Every round, the number of elements decreases by a factor of 2 in expectation

$W = O(n)$ expected $\quad S = O(\log^2 n)$ w.h.p.