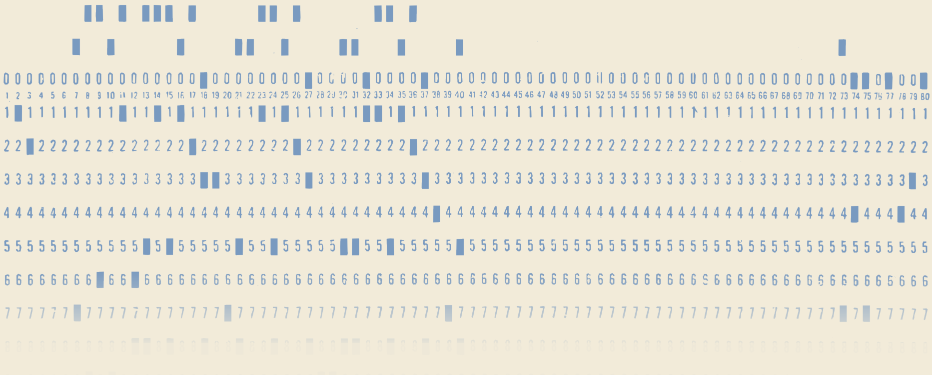


Lecture 18: Parallel Algorithms

COMP526: Efficient Algorithms

Will Rosenbaum
University of Liverpool

Updated: December 3, 2024

Announcements

1. Quiz 07 on Error Correcting Codes
   - Complete by 11:59pm, Friday 06 November
2. Grading is slow (sorry)
3. Last lectures:
   - Parallel Algorithms (today)
   - Text indexing (Thursday, next Tuesday)
   - Final review (next Thursday)
4. Attendance Code:

Meeting Goals

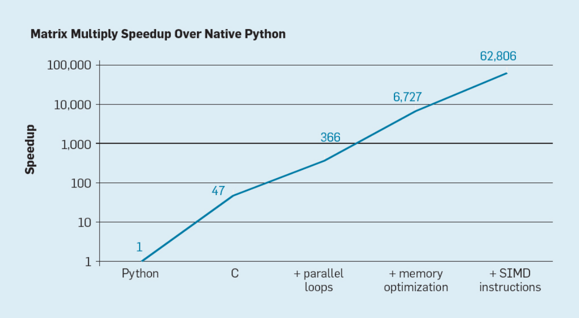
1. Discuss parallel algorithms!
2. Formalize cost measures for parallel algorithms
3. Argue Brent's theorem
4. Describe parallel searching algorithms
5. Describe parallel sorting algorithms
    - Sorting networks
    - Parallel MergeSort

Parallel Algorithms

From Last Time

Parallel Algorithms

- Modern computers can perform many operations simultaneously
- **SIMD**: single instruction, multiple data (e.g., GPU)
- **MIMD**: multiple instructions, multiple data (e.g., multicore CPU)
- To achieve maximal performance, parallelism of hardware must be exploited



PRAM Model

Parallel RAM

- Unbounded number of **processing elements** (PEs) think **cores**
- Access *shared memory*

PRAM Model

Parallel RAM

- Unbounded number of **processing elements** (PEs) think **cores**
- Access *shared memory*
- PEs run in lock-step synchronization

PRAM Model

Parallel RAM

- Unbounded number of **processing elements** (PEs) think **cores**
- Access *shared memory*
- PEs run in lock-step synchronization

Contention Resolution. How do we deal with conflicting operations?

PRAM Model

Parallel RAM

- Unbounded number of **processing elements** (PEs) think **cores**
- Access *shared memory*
- PEs run in lock-step synchronization

Contention Resolution. How do we deal with conflicting operations?

- EREW (exclusive read, exclusive write)
    - parallel access to same memory cell is forbidden

PRAM Model

Parallel RAM

- Unbounded number of **processing elements** (PEs) think **cores**
- Access *shared memory*
- PEs run in lock-step synchronization

Contention Resolution. How do we deal with conflicting operations?

- EREW (exclusive read, exclusive write)
  - parallel access to same memory cell is forbidden
- CREW (concurrent read, exclusive write)
  - parallel **write** access is forbiden

PRAM Model

Parallel RAM

- Unbounded number of **processing elements** (PEs) think **cores**
- Access *shared memory*
- PEs run in lock-step synchronization

Contention Resolution. How do we deal with conflicting operations?

- EREW (exclusive read, exclusive write)
  - parallel access to same memory cell is forbidden
- CREW (concurrent read, exclusive write)
  - parallel **write** access is forbiden
- CRCW (concurrent read, concurrent write)
  - need further contention resolution rules

PRAM Model

Parallel RAM

- Unbounded number of **processing elements** (PEs) think **cores**
- Access *shared memory*
- PEs run in lock-step synchronization

Contention Resolution. How do we deal with conflicting operations?

- EREW (exclusive read, exclusive write)
  - parallel access to same memory cell is forbidden
- CREW (concurrent read, exclusive write)
  - parallel **write** access is forbiden
- CRCW (concurrent read, concurrent write)
  - need further contention resolution rules

Bottom Line. No single model is well-suited for all applications

- we'll assume CREW
- reasoning about parallel programs can be **incredibly subtle!**

Measuring PRAM Efficiency

Main cost metrics

- **space:** the total amount of accessed memory
- **time:** the number of steps until all processes terminate
  - also known as **depth** or **span**
- **work:** total number of instructions executed by all processes

Measuring PRAM Efficiency

Main cost metrics

- **space:** the total amount of accessed memory
- **time:** the number of steps until all processes terminate
  - also known as **depth** or **span**
- **work:** total number of instructions executed by all processes

Goal:

- minimal span (= time)
- work is (asymptotically) no worse than the best *sequential* algorithm
  - called **work-efficient** algorithms

Models vs Reality

Idealization. The PRAM model does not limit the number of possible PEs (processing elements)

- “multithreaded” computing allows generation of unlimited threads

Models vs Reality

Idealization. The PRAM model does not limit the number of possible PEs (processing elements)

- “multithreaded” computing allows generation of unlimited threads

Reality. More threads does not magically speed up computation

- hardware limits the amount of parallel computation
    - e.g. limited to number of cores

Models vs Reality

Idealization. The PRAM model does not limit the number of possible PEs (processing elements)

- "multithreaded" computing allows generation of unlimited threads

Reality. More threads does not magically speed up computation

- hardware limits the amount of parallel computation
    - e.g. limited to number of cores

Questions.

- How relevant/applicable is the PRAM model if it assumes access to an *unlimited* number of PEs?

Models vs Reality

Idealization. The PRAM model does not limit the number of possible PEs (processing elements)

- "multithreaded" computing allows generation of unlimited threads

Reality. More threads does not magically speed up computation

- hardware limits the amount of parallel computation
  - e.g. limited to number of cores

Questions.

- How relevant/applicable is the PRAM model if it assumes access to an *unlimited* number of PEs?
- Can **every** task be performed efficiently in PRAM?
  - are there problems that are *inherently sequential?*

Brent's Theorem

Theorem (Brent). If an algorithm has span T and work W for an arbitrary number of processors, then the algorithm can be run on a PRAM with p PEs in time $O(T + W/p)$ using work W.

Brent's Theorem

Theorem (Brent). If an algorithm has span T and work W for an arbitrary number of processors, then the algorithm can be run on a PRAM with p PEs in time $O(T + W/p)$ using work W.

- Proof Idea: schedule parallel steps in a "round-robin" fashion on the p PEs.

Enough Generalities!

Parallel Algorithms

- Searching
- Sorting
  - Sorting Networks (SIMD)
    - sorting short lists
  - Parallel MergeSort
    - sorting long lists

Parallel Searching

Embarassingly Parallel Computation

A computational problem is **embarassingly parallel** if it can be split into *many* small subtasks that can be solved *independently* of each other.

Embarassingly Parallel Computation

A computational problem is **embarassingly parallel** if it can be split into *many* small subtasks that can be solved *independently* of each other.

- Example: vector sums $C[i] = A[i] + B[i]$

Embarassingly Parallel Computation

A computational problem is **embarassingly parallel** if it can be split into *many* small subtasks that can be solved *independently* of each other.

- Example: vector sums $C[i] = A[i] + B[i]$
- Non-examples?
  - Sorting
    - the final value of $A[i]$ depends on other values stored in A
    - not obvious how to employ parallelism

Embarassingly Parallel Computation

A computational problem is **embarassingly parallel** if it can be split into *many* small subtasks that can be solved *independently* of each other.

- Example: vector sums $C[i] = A[i] + B[i]$
- Non-examples?
  - Sorting
    - the final value of $A[i]$ depends on other values stored in A
    - not obvious how to employ parallelism
  - LZW compression ("P-complete")
    - Input: string S phrase p
    - Output: does LZW add p to the dictionary?

Parallel String Matching

Recall the **string matching** problem:

- Text T, length n
- Pattern P, length m
- **Goal:** find all occurrences of P in T
  - return array M of length n where $M[i] = 1$ if P matches T at index i, and $M[i] = 0$ otherwise

Parallel String Matching

Recall the **string matching** problem:

- Text T, length n
- Pattern P, length m
- **Goal:** find all occurrences of P in T
  - return array M of length n where $M[i] = 1$ if P matches T at index i, and $M[i] = 0$ otherwise

Question. Is this problem embarrassingly parallel?

Parallel String Matching

Recall the **string matching** problem:

- Text T, length n
- Pattern P, length m
- **Goal:** find all occurrences of P in T
  - return array M of length n where $M[i] = 1$ if P matches T at index i, and $M[i] = 0$ otherwise

Question. Is this problem embarrassingly parallel?

- can check each index i independently!

Parallel String Matching: Brute Force

Idea. Use the *brute force* procedure to check each i in parallel.

Parallel String Matching: Brute Force

Idea. Use the *brute force* procedure to check each i in parallel.

```
procedure ParallelBFMatch(T[0..n), P[0..m))
    for i = 0, 1, ..., n − 1 in parallel do
        for j = 0, 1, ..., m − 1 do
            if T[i + j] ≠ P[j] then break
        end for
        if j = m then M[i] = 1
        else M[i] = 0
    end for
end procedure
```

Parallel String Matching: Brute Force

Idea. Use the *brute force* procedure to check each i in parallel.

PollEverywhere

What is the **span** of this computation?

pollev.com/comp526

```
procedure ParallelBFMatch(T[0..n), P[0..m))
    for i = 0, 1, ..., n − 1 in parallel do
        for j = 0, 1, ..., m − 1 do
            if T[i + j] ≠ P[j] then break
        end for
        if j = m then M[i] = 1
        else M[i] = 0
    end for
end procedure
```

Parallel String Matching: Brute Force

Idea. Use the *brute force* procedure to check each i in parallel.

Efficiency

- Span:

procedure ParallelBFMatch($T[0..n), P[0..m)$)
    for $i = 0, 1, \ldots, n-1$ **in parallel do**
        for $j = 0, 1, \ldots, m-1$ **do**
            if $T[i+j] \neq P[j]$ **then break**
        end for
        if $j = m$ **then** $M[i] = 1$
        else $M[i] = 0$
    end for
end procedure

Parallel String Matching: Brute Force

Idea. Use the *brute force* procedure to check each i in parallel.

Efficiency

- Span: $T = O(m)$
- Work:

```
procedure ParallelBFMatch(T[0..n), P[0..m))
    for i = 0, 1, ..., n − 1 in parallel do
        for j = 0, 1, ..., m − 1 do
            if T[i + j] ≠ P[j] then break
        end for
        if j = m then M[i] = 1
        else M[i] = 0
    end for
end procedure
```

Parallel String Matching: Brute Force

Idea. Use the *brute force* procedure to check each i in parallel.

Efficiency

- Span: $T = O(m)$
- Work: $W = O(mn)$
  - **not** work efficient
- Brent: running time with p PEs is $O(m + mn/p)$

```
procedure ParallelBFMatch(T[0..n), P[0..m))
    for i = 0, 1, ..., n − 1 in parallel do
        for j = 0, 1, ..., m − 1 do
            if T[i + j] ≠ P[j] then break
        end for
        if j = m then M[i] = 1
        else M[i] = 0
    end for
end procedure
```

Parallel String Matching: Brute Force

Idea. Use the *brute force* procedure to check each i in parallel.

Efficiency

- Span: $T = O(m)$
- Work: $W = O(mn)$
  - **not** work efficient
- Brent: running time with p PEs is $O(m + mn/p)$

```
procedure ParallelBFMatch(T[0..n), P[0..m))
    for i = 0, 1, ..., n − 1 in parallel do
        for j = 0, 1, ..., m − 1 do
            if T[i + j] ≠ P[j] then break
        end for
        if j = m then M[i] = 1
        else M[i] = 0
    end for
end procedure
```

Question. Can we do better?

Parallelizing KMP

Recall the KMP (Knuth-Morris-Pratt) string matching algorithm

- compute failure link array
- apply FLA to search for matches

Parallelizing KMP

Recall the KMP (Knuth-Morris-Pratt) string matching algorithm

- compute failure link array
- apply FLA to search for matches

PollEverywhere

What is the running time of KMP to search for a pattern of length m in a text of length n?

pollev.com/comp526

Parallelizing KMP

Recall the KMP (Knuth-Morris-Pratt) string matching algorithm

- compute failure link array
- apply FLA to search for matches

Question. How to parallelize KMP?

Parallelizing KMP

Recall the KMP (Knuth-Morris-Pratt) string matching algorithm

- compute failure link array
- apply FLA to search for matches

Question. How to parallelize KMP?

- partition T into segments
- apply KMP to each segment

Parallelizing KMP

Recall the KMP (Knuth-Morris-Pratt) string matching algorithm

- compute failure link array
- apply FLA to search for matches

Question. How to parallelize KMP?

- partition T into segments
- apply KMP to each segment
- why doesn't this work?

Parallelizing KMP

Recall the KMP (Knuth-Morris-Pratt) string matching algorithm

- compute failure link array
- apply FLA to search for matches

Question. How to parallelize KMP?

- partition T into segments
- apply KMP to each segment
- why doesn't this work?
- use **overlapping** segments!

Parallelizing KMP

Recall the KMP (Knuth-Morris-Pratt) string matching algorithm

- compute failure link array
- apply FLA to search for matches

Question. How to parallelize KMP?

- partition T into segments
- apply KMP to each segment
- why doesn't this work?
- use **overlapping** segments!

procedure $\textsc{ParallelKMP}(T[0..n), P[0..m))$
  for $b = 0, 1, \ldots, \lceil n/m \rceil$ **in parallel do**
    $T_b = T[mb, mb + 2m - 1)$
    $M[i, i + m) \leftarrow \text{KMP}(T_b, P)$
  end for
end procedure

Parallelizing KMP

Recall the KMP (Knuth-Morris-Pratt) string matching algorithm

- compute failure link array
- apply FLA to search for matches

Question. How to parallelize KMP?

- partition T into segments
- apply KMP to each segment
- why doesn't this work?
- use **overlapping** segments!
- Span:

procedure ParallelKMP($T[0..n), P[0..m)$)
    for $b = 0, 1, \ldots, \lceil n/m \rceil$ **in parallel do**
        $T_b = T[mb, mb + 2m - 1)$
        $M[i, i+m) \leftarrow \text{KMP}(T_b, P)$
    end for
end procedure

Parallelizing KMP

Recall the KMP (Knuth-Morris-Pratt) string matching algorithm

- compute failure link array
- apply FLA to search for matches

Question. How to parallelize KMP?

- partition T into segments
- apply KMP to each segment
- why doesn't this work?
- use **overlapping** segments!

- Span: $O(m)$
- Work:

procedure ParallelKMP($T[0..n), P[0..m)$)
  for $b = 0, 1, \ldots, \lceil n/m \rceil$ **in parallel do**
    $T_b = T[mb, mb + 2m - 1)$
    $M[i, i + m) \leftarrow \mathrm{KMP}(T_b, P)$
  end for
end procedure

Parallelizing KMP

Recall the KMP (Knuth-Morris-Pratt) string matching algorithm

- compute failure link array
- apply FLA to search for matches

Question. How to parallelize KMP?

- partition T into segments
- apply KMP to each segment
- why doesn't this work?
- use **overlapping** segments!

- Span: $O(m)$
- Work: $O(n)$
  - this is work efficient!

```
procedure ParallelKMP(T[0..n), P[0..m))
    for b = 0, 1, ..., ⌈n/m⌉ in parallel do
        T_b = T[mb, mb + 2m − 1)
        M[i, i + m) ← KMP(T_b, P)
    end for
end procedure
```

Parallel String Matching Discussion

Assessment

- very simple methods
- can be run in a *distributed* setting
- parallel speedup only for $m \ll n$

Parallel String Matching Discussion

Assessment

- very simple methods
- can be run in a *distributed* setting
- parallel speedup only for $m \ll n$

Questions

- What if we only want to find if there is a single occurrence of P in T?

Parallel String Matching Discussion

Assessment

- very simple methods
- can be run in a *distributed* setting
- parallel speedup only for $m \ll n$

Questions

- What if we only want to find if there is a single occurrence of P in T?
- What if m large? State of the art:
    - $O(\log m)$ & work efficient for CREW-PRAM
    - CRCW-PRAM $O(1)$ matching part in $O(1)$ time, with $\Theta(\log\log m)$ preprocessing

Sorting Networks

Comparitors

Recall. In-place sorting algorithms modified the array according to the following pattern:

- check if $A[i]$ and $A[j]$ are out of order
- if so, swap their values

Comparitors

Recall. In-place sorting algorithms modified the array according to the following pattern:

- check if $A[i]$ and $A[j]$ are out of order
- if so, swap their values

Example. INSERTIONSORT

procedure INSERTIONSORT(a, n)
    for $i = 1, 2, \ldots, n-1$ **do**
        $j \leftarrow i$
        while $j > 0$ and $a[j] < a[j-1]$ **do**
            SWAP$(a, j, j-1)$
            $j \leftarrow j-1$
        end while
    end for
end procedure

Comparitors

Recall. In-place sorting algorithms modified the array according to the following pattern:

- check if $A[i]$ and $A[j]$ are out of order
- if so, swap their values

Example. INSERTIONSORT

procedure INSERTIONSORT(a, n)
  for $i = 1, 2, \ldots, n-1$ **do**
    $j \leftarrow i$
    while $j > 0$ and $a[j] < a[j-1]$ **do**
      SWAP$(a, j, j-1)$
      $j \leftarrow j - 1$
    end while
  end for
end procedure

Abstract View. A **comparator** is is a PE that takes two values as inputs and returns the values in sorted order.

- $\text{comp}(x, y) = (\min\{x, y\}, \max\{x, y\})$
- all array modifications of INSERTIONSORT can be performed by comparators

Comparitors

Recall. In-place sorting algorithms modified the array according to the following pattern:

- check if $A[i]$ and $A[j]$ are out of order
- if so, swap their values

Example. InsertionSort

```
procedure InsertionSort(a, n)
    for i = 1, 2, ..., n − 1 do
        j ← i
        while j > 0 and a[j] < a[j − 1] do
            Swap(a, j, j − 1)
            j ← j − 1
        end while
    end for
end procedure
```

Abstract View. A **comparator** is is a PE that takes two values as inputs and returns the values in sorted order.

- $\text{comp}(x, y) = (\min\{x, y\}, \max\{x, y\})$
- all array modifications of InsertionSort can be performed by comparators

Question. Which comparator operations of InsertionSort can be performed in parallel (while still ensuring correct output)?

Comparator Networks

Visual Representation.

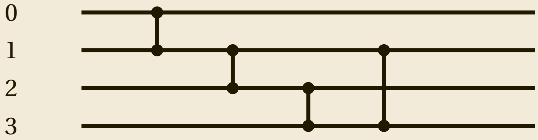
- Inputs/indices are represented by **wires** (horizontal lines)
- Comparators are vertical line segments between wires
  - interpretation: wire between wire i and j performs comp to indices i and j input
- Execution: Scan diagram from left to right and apply comparators in order they are encountered

Comparator Networks

Visual Representation.

- Inputs/indices are represented by **wires** (horizontal lines)
- Comparators are vertical line segments between wires
  - interpretation: wire between wire i and j performs comp to indices i and j input
- Execution: Scan diagram from left to right and apply comparators in order they are encountered

Example. Consider the following comparator network on 4 wires. What is the output on input $[4, 3, 2, 1]$?



Sorting Algorithms to Networks

Consider InsertionSort on inputs of size 5. What are the (possible) comparator operations performed by the algorithm?

- Which comparator operations could be performed *in parallel*?

```
procedure InsertionSort(a, n)
    for i = 1, 2, ..., n − 1 do
        j ← i
        while j > 0 and a[j] < a[j − 1] do
            Swap(a, j, j − 1)
            j ← j − 1
        end while
    end for
end procedure
```

Sorting Network Terminology

Definitions.

- A **comparator network** is defined by a set of wires and a sequence of comparators (left to right).

Sorting Network Terminology

Definitions.

- A **comparator network** is defined by a set of wires and a sequence of comparators (left to right).
- A comparator network is a **sorting network** if for all wire inputs, the resulting outputs are sorted.

Sorting Network Terminology

Definitions.

- A **comparator network** is defined by a set of wires and a sequence of comparators (left to right).
- A comparator network is a **sorting network** if for all wire inputs, the resulting outputs are sorted.
- The **depth** of a comparator network is the maximum number of comparators touched on any path from input to output (including crossed comparators).

Sorting Network Terminology

Definitions.

- A **comparator network** is defined by a set of wires and a sequence of comparators (left to right).
- A comparator network is a **sorting network** if for all wire inputs, the resulting outputs are sorted.
- The **depth** of a comparator network is the maximum number of comparators touched on any path from input to output (including crossed comparators).

Sorting networks and parallel algorithms.

- Each comparator is a process element
- The depth is the span (running time) of the network
- The work is the number of comparators

Sorting Network Terminology

Definitions.

- A **comparator network** is defined by a set of wires and a sequence of comparators (left to right).
- A comparator network is a **sorting network** if for all wire inputs, the resulting outputs are sorted.
- The **depth** of a comparator network is the maximum number of comparators touched on any path from input to output (including crossed comparators).

Sorting networks and parallel algorithms.

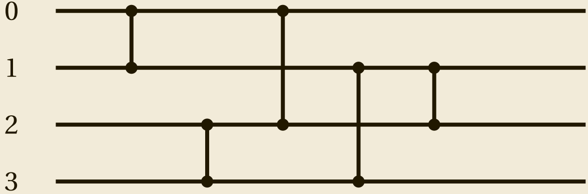
- Each comparator is a process element
- The depth is the span (running time) of the network
- The work is the number of comparators

Question. What is the smallest/shallowest sorting network for a given input size?

- Optimal *size* sorting networks are only known for up to 12 inputs
- Optimal *depth* is only known up to 18 inputs

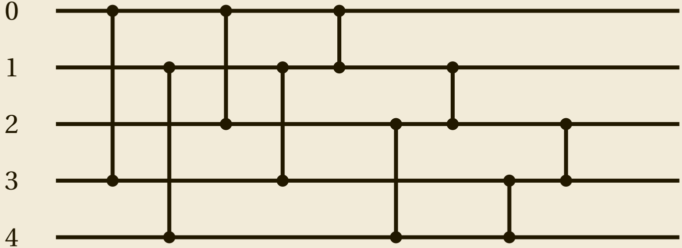
Some Optimal Sorting Networks

Example. $n = 4$ wires. What is the depth?



Some Optimal Sorting Networks

Example. $n = 5$ wires.



PollEverywhere

What is the **depth** of this sorting network?

pollev.com/comp526

Sorting Network Discussion

Applications

- Hardware-level implementations
    - comparators implemented with simple circuits
    - operations in one/few clock cycles

Sorting Network Discussion

Applications

- Hardware-level implementations
  - comparators implemented with simple circuits
  - operations in one/few clock cycles
- Sorting with GPUs
  - apply many (software) comparators in parallel (SIMD)

Sorting Network Discussion

Applications

- Hardware-level implementations
  - comparators implemented with simple circuits
  - operations in one/few clock cycles
- Sorting with GPUs
  - apply many (software) comparators in parallel (SIMD)

General Construction. Bitonic Merge Sort

- Mimic recursive structure of MERGESORT
- Size $O(n \log^2 n)$
- Depth $O(\log^2 n)$
- Not work-efficient, but still practical

Parallel MergeSort

Parallel Divide & Conquer?

Observation. The Divide & Conquer strategy can lend itself well to parallelism:

1. Divide problem into sub-tasks
2. Solve the subtasks

3. Merge solutions of the subtasks

Parallel Divide & Conquer?

Observation. The Divide & Conquer strategy can lend itself well to parallelism:

1. Divide problem into sub-tasks
2. Solve the subtasks (**independently**)
    - Parallelize these!
3. Merge solutions of the subtasks

Parallel Divide & Conquer?

Observation. The Divide & Conquer strategy can lend itself well to parallelism:

1. Divide problem into sub-tasks
2. Solve the subtasks (**independently**)
   - Parallelize these!
3. Merge solutions of the subtasks (**...?**)
   - How to parallelize this?

Parallel MergeSort?

Revisited: MergeSort

```
procedure MergeSort(A, i, k)
    if i < k then
        j ← ⌊(i + k)/2⌋
        MergeSort(A, i, j)
        MergeSort(A, j + 1, k)
        B ← Copy(A, i, j)
        C ← Copy(A, j + 1, k)
        Merge(B, C, A, i)
    end if
end procedure
```

Parallel MergeSort?

PollEverywhere

What is the span of MergeSort with parallel recursive calls and sequential merges?

pollev.com/comp526

procedure MergeSort(A, i, k)
  if $i < k$ **then**
    $j \leftarrow \lfloor (i+k)/2 \rfloor$
    MergeSort(A, i, j)
    MergeSort($A, j+1, k$)
    $B \leftarrow$ Copy(A, i, j)
    $C \leftarrow$ Copy($A, j+1, k$)
    Merge(B, C, A, i)
  end if
end procedure

Parallelizing Merges

Question. How can we *parallelize merges?*

Parallelizing Merges

Question. How can we *parallelize merges?*

- For each x, find the final index of x
- How do we find this?

Parallelizing Merges

Question. How can we *parallelize merges?*

- For each x, find the final index of x
- How do we find this?
- index = # elements $\leq x$
  - # in x's sub-array
  - # in other sub-array
- How to compute these?

Parallelizing Merges

Question. How can we *parallelize merges?*

- For each x, find the final index of x
- How do we find this?
- index = # elements $\leq x$
    - # in x's sub-array
    - # in other sub-array
- How to compute these?

Idea.

- In x's own sub-array, just use x's index!

Parallelizing Merges

Question. How can we *parallelize merges?*

- For each x, find the final index of x
- How do we find this?
- index = # elements $\leq x$
  - # in x's sub-array
  - # in other sub-array
- How to compute these?

Idea.

- In x's own sub-array, just use x's index!
- For the other sub-array, use binary search!

Parallelizing Merges

Question. How can we *parallelize merges?*

- For each x, find the final index of x
- How do we find this?
- index = # elements $\leq x$
  - # in x's sub-array
  - # in other sub-array
- How to compute these?

Idea.

- In x's own sub-array, just use x's index!
- For the other sub-array, use binary search!
- **Parallelize:** do each x in parallel!

Parallel MergeSort in Code

```
procedure ParallelMerge(A[l..m), A[m..r), B)
    for i = l, ..., m − 1 in parallel do
        k ← (i − l) + BinarySearch(A[m..r), A[i])
        B[k] ← A[i]
    end for
    for j = m, m + 1, ..., r − 1 in parallel do
        k ← BinarySearch(A[l..m), A[j])
        B[k] ← A[j]
    end for
end procedure
```

Parallel MergeSort in Code

```
procedure ParallelMerge(A[l..m), A[m..r), B)
    for i = l, ..., m − 1 in parallel do
        k ← (i − l) + BinarySearch(A[m..r), A[i])
        B[k] ← A[i]
    end for
    for j = m, m + 1, ..., r − 1 in parallel do
        k ← BinarySearch(A[l..m), A[j])
        B[k] ← A[j]
    end for
end procedure
```

Questions.

- What is the **span** of ParallelMerge?

Parallel MergeSort in Code

```
procedure ParallelMerge(A[l..m), A[m..r), B)
    for i = l, ..., m − 1 in parallel do
        k ← (i − l) + BinarySearch(A[m..r), A[i])
        B[k] ← A[i]
    end for
    for j = m, m + 1, ..., r − 1 in parallel do
        k ← BinarySearch(A[l..m), A[j])
        B[k] ← A[j]
    end for
end procedure
```

Questions.

- What is the **span** of ParallelMerge?
    - $\Theta(\log n)$
- What is the **work** of ParallelMerge?

Parallel MergeSort in Code

```
procedure ParallelMerge(A[l..m), A[m..r), B)
    for i = l, ..., m − 1 in parallel do
        k ← (i − l) + BinarySearch(A[m..r), A[i])
        B[k] ← A[i]
    end for
    for j = m, m + 1, ..., r − 1 in parallel do
        k ← BinarySearch(A[l..m), A[j])
        B[k] ← A[j]
    end for
end procedure
```

Questions.

- What is the **span** of ParallelMerge?
    - $\Theta(\log n)$
- What is the **work** of ParallelMerge?
    - $\Theta(n \log n)$

Parallel MergeSort Analysis

Overall Procedure

1. Split (sub)array in half
2. Parallel recursive MergeSorts
3. ParallelMerge sorted halves

Parallel MergeSort Analysis

Overall Procedure

1. Split (sub)array in half
2. Parallel recursive MergeSorts
3. PARALLELMERGE sorted halves

Span Analysis

- Merge has span $\Theta(\log n)$
- Depth of recursion tree is $\Theta(\log n)$
- Total time: $\Theta(\log^2 n)$

Parallel MergeSort Analysis

Overall Procedure

1. Split (sub)array in half
2. Parallel recursive MergeSorts
3. PARALLELMERGE sorted halves

Span Analysis

- Merge has span $\Theta(\log n)$
- Depth of recursion tree is $\Theta(\log n)$
- Total time: $\Theta(\log^2 n)$

Work Analysis

- Merge has work $\Theta(n \log n)$
- Summing over recursive calls gives $\Theta(n \log^2 n)$

Parallel MergeSort Analysis

Overall Procedure

1. Split (sub)array in half
2. Parallel recursive MergeSorts
3. PARALLELMERGE sorted halves

Span Analysis

- Merge has span $\Theta(\log n)$
- Depth of recursion tree is $\Theta(\log n)$
- Total time: $\Theta(\log^2 n)$

Work Analysis

- Merge has work $\Theta(n \log n)$
- Summing over recursive calls gives $\Theta(n \log^2 n)$

Improvements. Merge can be improved to $\Theta(n)$ work! (but it's complicated)

Concluding Thoughts

Parallelism is Necessary

- Computer hardware is naturally parallel
    - sequential computing is an illusion!

Concluding Thoughts

Parallelism is Necessary

- Computer hardware is naturally parallel
  - sequential computing is an illusion!

Parallelism is Powerful

- Recent explosion in computing power is due to parallelism!

Concluding Thoughts

Parallelism is Necessary

- Computer hardware is naturally parallel
    - sequential computing is an illusion!

Parallelism is Powerful

- Recent explosion in computing power is due to parallelism!

Parallelism is Subtle

- Reasoning about parallel programs is hard
- Writing correct parallel programs is hard
- Idealized models abstract away many challenges
    - no perfect synchrony?
    - tolerate faults?

Next Time

- Text Indexing

Scratch Notes