Algorithms at Scale
(Week 10)
Summary

Today: Parallelism

Models of Parallelism
- How to predict the performance of algorithms?

Some simple examples...

Sorting
- Parallel MergeSort

Trees and Graphs

Last Week: Caching

Breadth-First-Search
- Sorting your graph

MIS
- Luby’s Algorithm
- Cache-efficient implementation

MST
- Connectivity
- Minimum Spanning Tree
Today:

MiniProject update due today.

Next week:

MiniProject explanatory section due
Moore’s Law

Number of transistors doubles every 2 years!

“The complexity for minimum component costs has increased at a rate of roughly a factor of two per year... Certainly over the short term this rate can be expected to continue, if not to increase.” Gordon Moore, 1965

Limits will be reached in 10-20 years...maybe.

More transistors == faster computers?

- More transistors per chip $\Rightarrow$ smaller transistors.
- Smaller transistors $\Rightarrow$ faster
- Conclusion:
  
  Clock speed doubles every two years, also.
Parallel Algorithms


Clock Speed

1000 1000 100 100 10 10 1 1 1 1 1

Intel 8080
Intel 286
Intel 386
Intel 486
Pentium Pro
AMD Athlon
Pentium 4
Core i7

Parallel Algorithms

What to do with more transistors?

– More functionality
  • GPUs, FPUs, specialized crypto hardware, etc.
– Deeper pipelines
– More clever instruction issue (out-of-order issue, scoreboard, etc.)
– More on chip memory (cache)

Limits for making faster processors?
Parallel Algorithms

Problems with faster clock speeds:

- Heat
  - Faster switching creates more heat.

- Wires
  - Adding more components takes more wires to connect.
  - Wires don’t scale well!

- Clock synchronization
  - How do you keep the entire chip synchronized?
  - If the clock is too fast, then the time it takes to propagate a clock signal from one edge to the other matters!
Parallel Algorithms

Conclusion:

– We have lots of new transistors to use.
– We can’t use them to make the CPU faster.

What do we do?
Parallel Algorithms

Instructions per Clock Cycle

Multi-cycle instructions

Pipelined execution

Out-of-order issue

In-order issue

Multi-core Era

To make an algorithm run faster:

- Must take advantage of multiple cores.
- Many steps executed at the same time!
Parallel Algorithms

To make an algorithm run faster:

– Must take advantage of multiple cores.
– Many steps executed at the same time!

CS5234 algorithms:

– Sampling ➔ lots of parallelism
– Sketches ➔ lots of parallelism
– Streaming ➔ lots of parallelism
– Cache-efficient algorithms??
Parallel Algorithms

Challenges:

– How do we write parallel programs?
  • Partition problem over multiple cores.
  • Specify what can happen at the same time.
  • Avoid unnecessary sequential dependencies.
  • Synchronize different threads (e.g., locks).
  • Avoid race conditions!
  • Avoid deadlocks!
Parallel Algorithms

Challenges:

– How do we analyze parallel algorithms?
  
  • Total running time depends on # of cores.
  
  • Cost is harder to calculate.
  
  • Measure of scalability?
Challenges:

- How do we debug parallel algorithms?
  - More non-determinacy
  - Scheduling leads to un-reproduceable bugs
    - Heisenbugs!
  - Stepping through parallel programs is hard.
  - Race conditions are hard.
  - Deadlocks are hard.
Parallel Algorithms

Different types of parallelism:

- **multicore**
  - on-chip parallelism: synchronized, shared caches, etc.

- **multisocket**
  - closely coupled, highly synchronized, shared caches

- **cluster / data center**
  - connected by a high-performance interconnect

- **distributed networks**
  - slower interconnect, less tightly synchronized
Different types of parallelism:

- **multicore**
  - on-chip parallelism:

- **multisocket**
  - closely coupled, highly synchronized, shared caches

- **cluster / data center**
  - connected by a high-performance interconnect

- **distributed networks**
  - slower interconnect, less tightly synchronized

Parallel Algorithms

Different settings

1) Different costs
2) Different solutions
Parallel Algorithms

Different types of parallelism:

- **multicore**
  - on-chip parallelism: synchronized, shared caches, etc.

- **multisocket**
  - closely coupled, highly synchronized, shared caches

- **cluster / data center**
  - connected by a high-performance interconnect

- **distributed networks**
  - slower interconnect, less tightly synchronized
Parallel Algorithms

Different types of parallelism:

- **multicore**
  - on-chip parallelism: synchronized, shared caches, etc.

- **multisocket**
  - closely coupled, highly synchronized, shared caches

- **cluster / data center**
  - connected by a high-performance interconnect

- **distributed networks**
  - slower interconnect, less tightly synchronized
How to model parallel programs?

PRAM

Assumptions

- \( p \) processors, \( p \) large.
- shared memory
- program each proc separately
How to model parallel programs?

PRAM

Assumptions
• \( p \) processors, \( p \) large.
• shared memory
• program each proc separately

Example problem: AllZeros
• Given array \( A[1..n] \).
• Return \textbf{true} if \( A[j] = 0 \) for all \( j \).
• Return \textbf{false} otherwise.
How to model parallel programs?

\textbf{AllZero}(A, 1, n, p)_j

\texttt{for } i = (n/p)(j-1)+1 \texttt{ to } (n/p)(j) \texttt{ do}

\texttt{if } A[i] \neq 0 \texttt{ then } \texttt{answer} = \texttt{false}

\texttt{done} = \texttt{done} + 1

\texttt{wait until } (\texttt{done} == p)

\texttt{return } \texttt{answer}.
How to model parallel programs?

AllZero$(A, 1, n, p)_j$

\[
\text{for } i = (n/p)(j-1)+1 \text{ to } (n/p)(j) \text{ do }
\]

\[
\text{if } A[i] \neq 0 \text{ then } answer = false
\]

\[
done = done + 1
\]

\[
\text{wait until } (done == p)
\]

\[
\text{return } answer.
\]
How to model parallel programs?

\[ \text{AllZero}(A, 1, n, p)_j \]

\[ \text{for } i = \frac{(n/p)(j-1)+1}{\text{to }} \frac{(n/p)(j)}{\text{do}} \]

\[ \text{if } A[i] \neq 0 \text{ then } \text{answer} = \text{false} \]

\[ \text{done} = \text{done} + 1 \]

\[ \text{wait until } (\text{done} == p) \]

\[ \text{return } \text{answer}. \]
How to model parallel programs?

AllZero(A, 1, n, p)_j

\[
\text{for } i = (n/p)(j-1)+1 \text{ to } (n/p)(j) \text{ do }
\]

\[
\text{if } A[i] \neq 0 \text{ then } answer = \text{false}
\]

\[
done = done + 1
\]

\[
\text{wait until } (done == p)
\]

\[
\text{return } answer.
\]
How to model parallel programs?

\[ \text{AllZero}(A, 1, n, p)_j \]

\[
\text{for } i = (n/p)(j-1)+1 \text{ to } (n/p)(j) \text{ do}
\]

\[
\text{if } A[i] \neq 0 \text{ then } \text{answer} = \text{false}
\]

\[
done = done + 1
\]

\[
\text{wait until } (done == p)
\]

\[
\text{return } \text{answer}.
\]
How to model parallel programs?

\[ \text{AllZero}(A, 1, n, p)_j \]

\[
\text{for } i = (n/p)(j-1)+1 \text{ to } (n/p)(j) \text{ do} \\
\quad \text{if } A[i] \neq 0 \text{ then } \text{answer} = \text{false} \\
\quad \text{done} = \text{done} + 1 \\
\text{wait until } (\text{done} == p) \\
\text{return } \text{answer}.
\]

specifies behavior on processor j

Time: \( O\left(\frac{n}{p}\right) \)
How to model parallel programs?

PRAM

Assumptions
• $p$ processors, $p$ large.
• shared memory
• program each proc separately

Limitations
• Must carefully manage all processor interactions.
• Manually divide problem among processors.
• Number of processors may be hard-coded into the solution.
• Low-level way to design parallel algorithms.
How to model parallel programs?

Another example: summing an array

Idea: use a tree
Another example: summing an array

Algorithm:

RandomSum:

repeat until root is not empty:

Choose a random node u in the tree.

If both children are not empty, then:

set u = u.left + u.right
How to model parallel programs?

RandomSum:

repeat until root is not empty:

Choose a random node \( u \) in the tree.

If both children are not empty, then:

set \( u = u.\text{left} + u.\text{right} \)

Fun exercise: Prove the theorem.

Theorem:
RandomSum finishes in time: \( \Theta \left( \frac{n \log n}{p} + \log n \right) \)
How to sum an array?

PRAM-Sum:
How to sum an array?

**PRAM-Sum:**

Assign processors to nodes in tree.

Each processor does assigned work in tree?

Not as easy to specify precise behavior.
How to sum an array?

```
Sum(A[1..n], b, e):
  if (b = e) return A[b]
  mid = (b+e)/2
  in parallel:
  1. L = Sum(A, b, mid)
  2. R = Sum(A, mid+2, e)
  sync
  return L+R
```
How to sum an array?

Sum(A[1..n], b, e):

if (b = e) return A[b]

mid = (b+e)/2

in parallel:
1. L = Sum(A, b, mid)
2. R = Sum(A, mid+2, e)

sync
return L+R

Observations:

Same tree calculation!
Each L+R computes 1 node
How to sum an array?

Sum(A[1..n], b, e):

\[
\begin{align*}
&\text{if } (b = e) \text{ return } A[b] \\
&mid = (b+e)/2 \\
&\text{in parallel:} \\
&1. \text{ L } = \text{ Sum(}A, b, \text{ mid}) \\
&2. \text{ R } = \text{ Sum(}A, \text{ mid+2, e}) \\
&\text{sync} \\
&\text{return } L+R
\end{align*}
\]

Observations:

Number of processors is not specified anywhere.
How to sum an array?

Sum(A[1..n], b, e):

if (b = e) return A[b]

\[ mid = \frac{(b+e)}{2} \]

in parallel:
1. \( L = \text{Sum}(A, b, mid) \)
2. \( R = \text{Sum}(A, mid+2, e) \)

sync

return \( L+R \)

Observations:

Number of processors is not specified anywhere.

A scheduler assigns parallel computations to processors.
How to sum an array?

Sum(A[1..n], b, e):

if (b = e) return A[b]

mid = (b+e)/2

in parallel:
1. L = Sum(A, b, mid)
2. R = Sum(A, mid+2, e)

sync

return L+R

Time:

On one processor??
How to sum an array?

Sum(A[1..n], b, e):

\[
\text{if } (b = e) \text{ return } A[b]
\]

mid = (b+e)/2

in parallel:
1. L = Sum(A, b, mid)
2. R = Sum(A, mid+2, e)

sync

return L+R

Time:

On one processor:

\[
T_1(n) = 2T_1(n/2) + O(1)
\]

\[
= O(n)
\]

Just ignore parallel parts and run all the code!

Work

Total steps done by all processors.
How to sum an array?

Sum(A[1..n], b, e):

if (b = e) return A[b]

mid = (b+e)/2

in parallel:
1. L = Sum(A, b, mid)
2. R = Sum(A, mid+2, e)

sync

return L+R

Time:

On infinite processors??
How to sum an array?

\[ \text{Sum}(A[1..n], b, e): \]
\[
\begin{align*}
\text{if } (b = e) & \text{ return } A[b] \\
\text{mid} & = (b+e)/2 \\
\text{in parallel:} & \\
1. \quad L = \text{Sum}(A, b, \text{mid}) \\
2. \quad R = \text{Sum}(A, \text{mid}+2, e) \\
\text{sync} & \\
\text{return } L+R
\end{align*}
\]

Each parallel part is delegated to two different processors.

\[ T_\infty(n) = T_\infty(n/2) + O(1) \]
\[ = O(\log n) \]

Time:

Critical Path or Span:
longest path in the program
How to sum an array?

Sum(A[1..n], b, e):

- if (b = e) return A[b]
- mid = (b+e)/2

in parallel:
1. L = Sum(A, b, mid)
2. R = Sum(A, mid+2, e)

sync

return L+R

Time:

On p processors??

\[ T_p(n) = ?? \]
How to sum an array?

Sum($A[1..n]$, b, e):

if $b = e$ return $A[b]$

$mid = (b+e)/2$

in parallel:
1. $L = Sum(A, b, mid)$
2. $R = Sum(A, mid+2, e)$

sync

return $L+R$

Time:

On $p$ processors??

DEPENDS!
The scheduler matters.
Dynamic Multithreading

- Two special commands:
  - fork (or “in parallel”): start a new (parallel) procedure
  - sync: wait for all concurrent tasks to complete

- Machine independent
  - No fixed number of processors.

- Scheduler assigns tasks to processors.
How to sum an array?

\[
\text{Sum}(A[1..n], b, e):
\]

\[
\text{if } (b = e) \text{ return } A[b]
\]

\[
\text{mid} = (b+e)/2
\]

fork:

1. \( L = \text{Sum}(A, b, \text{mid}) \)
2. \( R = \text{Sum}(A, \text{mid}+2, e) \)

sync

return \( L+R \)
Model as a DAG

```
fib(4)    fib(3)    fib(2)
fib(2)    fib(1)    fib(0)
fib(1)    fib(0)
```
Model as a DAG

Work = T_1 = ??
Model as a DAG

\[
\text{Work} = T_1 = 17
\]
Model as a DAG

Span = $T_\infty = ??$
Model as a DAG

\[
\text{Span} = T_\infty = 8
\]
Analyzing Parallel Algorithms

Key metrics:

- Work: $T_1$
- Span: $T_\infty$

Work = 18
Span = 9
Analyzing Parallel Algorithms

Key metrics:
- Work: $T_1$
- Span: $T_\infty$

Parallelism:

$$\frac{T_1}{T_\infty}$$

Determines number of processors that we can use productively.
Analyzing a Parallel Computation

Running Time: $T_p$

– Total running time if executed on $p$ processors.

– Claim: $T_p > T_\infty$

  • Cannot run slower on more processors!
  • Mostly, but not always, true in practice.
Analyzing a Parallel Computation

Running Time: $T_p$

- Total running time if executed on $p$ processors.

- Claim: $T_p > T_1 / p$
  
  - Total work, divided perfectly evenly over $p$ processors.
  
  - Only for a perfectly parallel program.
Analyzing a Parallel Computation

Running Time: $T_p$

- Total running time if executed on $p$ processors.
- $T_p > T_1 / p$
- $T_p > T_\infty$

Goal: $T_p = (T_1 / p) + T_\infty$

- Almost optimal (within a factor of 2).
- We have to spend time $T_\infty$ on the critical path. We call this the “sequential” part of the computation.
- We have to spend time $(T_1 / p)$ doing all the work. We call this the “parallel” part of the computation.
Analyzing Parallel Algorithms

Key metrics:
- Work: $T_1$
- Span: $T_\infty$

Parallelism:

\[
\frac{T_1}{T_\infty}
\]

Assume $p = \frac{T_1}{T_\infty}$:

\[
T_p = \frac{T_1}{p} + T_\infty
\]

\[
= \frac{T_1}{T_1/T_\infty} + T_\infty
\]

\[
= 2T_\infty
\]
Analyzing a Parallel Computation

Greedy Scheduler

- If $\leq p$ tasks are ready, execute all of them.
- If $> p$ tasks are ready, execute $p$ of them.
Analyzing a Parallel Computation

Greedy Scheduler

- If $\leq p$ tasks are ready, execute all of them.
- If $> p$ tasks are ready, execute $p$ of them.

Assume $p = 3$
Analyzing a Parallel Computation

Greedy Scheduler

- If $\leq p$ tasks are ready, execute all of them.
- If $> p$ tasks are ready, execute $p$ of them.

Assume $p = 3$
Analyzing a Parallel Computation

Greedy Scheduler

– If \( \leq p \) tasks are ready, execute all of them.
– If \( > p \) tasks are ready, execute \( p \) of them.

Assume \( p = 3 \)
Analyzing a Parallel Computation

Greedy Scheduler

- If \( \leq p \) tasks are ready, execute all of them.
- If \( > p \) tasks are ready, execute \( p \) of them.

Assume \( p = 3 \)
Analyzing a Parallel Computation

Greedy Scheduler

- If \( \leq p \) tasks are ready, execute all of them.
- If \( > p \) tasks are ready, execute \( p \) of them.

Assume \( p = 3 \)
Greedy Scheduler

1. If $\leq p$ tasks are ready, execute all of them.
2. If $> p$ tasks are ready, execute $p$ of them.

Theorem (Brent-Graham): $T_p \leq (T_1 / p) + T_\infty$

Proof:

– At most steps $(T_1 / p)$ of type 2.
– Every step of type 1 works on the critical path, so at most $+ T_\infty$ steps of type 1.
Analyzing a Parallel Computation

Greedy Scheduler

1. If \( \leq p \) tasks are ready, execute all of them.
2. If \( > p \) tasks are ready, execute \( p \) of them.

Problem:

- Greedy scheduler is *centralized*.
- How to determine which tasks are ready?
- How to assign processors to ready tasks?
Work-Stealing Scheduler

- Each process keeps a queue of tasks to work on.
- Each `spawn` adds one task to queue, keeps working.
- Whenever a process is free, it takes a task from a randomly chosen queue (i.e., work-stealing).

Theorem (work-stealing): \( T_p \leq (T_1 / p) + O(T_\infty) \)

- See, e.g., Intel Parallel Studio, Cilk, Cilk++, Java, etc.
- Many frameworks exist to schedule parallel computations.
How to design parallel algorithms

PRAM

– Schedule each processor manually.
– Design algorithm for a specific number of processors.

Fork-Join model

– Focus on parallelism (and think about algorithms).
– Rely on a good scheduler to assign work to processors.
Parallel Sorting
Parallel Sorting

\textbf{MergeSort}(A, n)

\hspace{1cm} \textbf{if} (n=1) \textbf{then} return;

\hspace{1cm} \textbf{else}

\hspace{2cm} X = \text{MergeSort}(A[1..n/2], n/2)

\hspace{2cm} Y = \text{MergeSort}(A[n/2+1, n], n/2)

\hspace{2cm} A = \text{Merge}(X, Y);
pMergeSort(A, n)

if (n==1) then return;
else
    X = fork pMergeSort(A[1..n/2], n/2)
    Y = fork pMergeSort(A[n/2+1, n], n/2)
    sync;
    A = Merge(X, Y);
Parallel Sorting

\texttt{pMergeSort(A, n)}

\begin{itemize}
\item \textbf{if} (n==1) \textbf{then} return;
\item else
\begin{itemize}
\item \texttt{X = fork pMergeSort(A[1..n/2], n/2)}
\item \texttt{Y = fork pMergeSort(A[n/2+1, n], n/2)}
\item \texttt{sync;}
\item \texttt{A = Merge(X, Y);}
\end{itemize}
\end{itemize}

Work Analysis

\begin{itemize}
\item \(T_1(n) = 2T_1(n/2) + O(n) = O(n \log n)\)
\end{itemize}
Parallel Sorting

\texttt{pMergeSort}(A, n)

\textbf{if} (n==1) \textbf{then} return;
\textbf{else}

\hspace{1em}X = \texttt{fork} \ pMergeSort(A[1..n/2], n/2)
\hspace{1em}Y = \texttt{fork} \ pMergeSort(A[n/2+1, n], n/2)
\hspace{1em}sync;
\hspace{1em}A = \text{Merge}(X, Y);

Critical Path Analysis

\hspace{1em}- \ \ T_\infty(n) = T_\infty(n/2) + O(n) = O(n)

Oops!
Parallel Merge

How do we merge two arrays A and B in parallel?
Parallel Merge

How do we merge two arrays A and B in parallel?

- Let’s try divide and conquer:
  
  \[
  X = \text{fork } \text{Merge}(A[1..n/2], B[1..n/2])
  \]

  \[
  Y = \text{fork } \text{Merge}(A[n/2+1..n], B[n/2+1..n])
  \]

<table>
<thead>
<tr>
<th>A</th>
<th>5</th>
<th>8</th>
<th>9</th>
<th>11</th>
<th>13</th>
<th>20</th>
<th>22</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>23</td>
<td>27</td>
<td>29</td>
<td>32</td>
<td>35</td>
</tr>
</tbody>
</table>

- How do we merge X and Y?

<table>
<thead>
<tr>
<th>X</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>13</td>
<td>20</td>
<td>22</td>
<td>24</td>
<td>27</td>
<td>29</td>
<td>32</td>
<td>35</td>
</tr>
</tbody>
</table>
Parallel Merge

A = [1, n/2, x, n]

B = [1, j, y, n]

Binary Search: B[j] \leq x \leq B[j+1]

Recurse:  \texttt{pMerge}(A[1..n/2], B[1..j])
            \texttt{pMerge}(A[n/2+1..n], B[j+1..n])
Parallel Merge

\[ \text{pMerge}(A[1..k], B[1..m], C[1..n]) \]

\begin{align*}
\text{if} \ (m > k) \ & \text{then} \ \text{pMerge}(B, A, C); \\
\text{else if} \ (n==1) \ & \text{then} \ C[1] = A[1]; \\
\text{else if} \ (k==1) \ & \text{and} \ (m==1) \ \text{then} \\
& \text{if} \ (A[1] \leq B[1]) \ \text{then} \\
& \text{else} \\
\text{else} \\
& \text{binary search for} \ j \ \text{where} \ B[j] \leq A[k/2] \leq B[j+1] \\
& \text{fork} \ \text{pMerge}(A[1..k/2],B[1..j],C[1..k/2+j]) \\
& \text{fork} \ \text{pMerge}(A[k/2+1..1],B[j+1..m],C[k/2+j+1..n]) \\
& \text{sync};
\end{align*}
Parallel Merge

A =

Binary Search: B[j] ≤ x ≤ B[j+1]

B =

Recurse:

\textit{pMerge}(A[1..n/2], B[1..j])
\textit{pMerge}(A[n/2+1..n], B[j+1..n])
Parallel Merge

Critical Path Analysis:

- Define $T_\infty(n)$ to be the critical path of parallel merge when the two input arrays A and B together have $n$ elements.
- There are $k > n/2$ elements in A, and $(n-k)$ elements in B, so in total:
  
  \[ k/2 + (n - k) = n - (k/2) < n - (n/4) < 3n/4 \]

- $T_\infty(n) \leq T_\infty(3n/4) + O(\log n)$
  
  $\approx O(\log^2 n)$
Parallel Merge

Work Analysis:

- Define $T_1(n)$ to be the work done by parallel merge when the two input arrays A and B together have $n$ elements.
- Fix: $\frac{1}{4} \leq \alpha \leq \frac{3}{4}$

\[ T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + O(\log n) \]
\[ \approx 2T_1(n/2) + O(\log n) \]
\[ = O(n) \]
Parallel Sorting

\[ \text{pMergeSort}(A, n) \]

\[
\text{if } (n=1) \text{ then return;}
\]

\[
\text{else}
\]

\[
X = \text{fork} \text{ pMergeSort}(A[1..n/2], n/2)
\]

\[
Y = \text{fork} \text{ pMergeSort}(A[n/2+1, n], n/2)
\]

\[
\text{sync;}
\]

\[
A = \text{fork} \text{ pMerge}(X, Y);
\]

\[
\text{sync;}
\]

Critical Path Analysis

\[
- \quad T_\infty(n) = T_\infty(n/2) + O(\log^2 n) = O(\log^3 n)
\]
How do we store a set of items?
How do we store a set of items?

- **insert**: add an item to the set
- **delete**: remove an item from the set

Data Structures

![Diagram showing set operations](image)
How do we store a set of items?

- **insert**: add an item to the set
- **delete**: remove an item from the set
- **divide**: divide the set into two (approximately) equal sized pieces
Data Structures

How do we store a set of items?

- **insert**: add an item to the set
- **delete**: remove an item from the set
- **divide**: divide the set into two (approximately) equal sized pieces
- **union**: combine two sets
- **subtraction**: remove one set from another
How do we store a set of items?

- **insert**: add an item to the set
- **delete**: remove an item from the set
- **divide**: divide the set into two (approximately) equal sized pieces
- **union**: combine two sets
- **subtraction**: remove one set from another
How do we store a set of items?

- **insert**: add an item to the set
- **delete**: remove an item from the set
- **divide**: divide the set into two (approximately) equal sized pieces
- **union**: combine two sets
- **subtraction**: remove one set from another
- **intersection**: find the intersection of two sets
- **set difference**: find the items only in one set

**Data Structures**

\[
\text{set difference} \quad \begin{array}{c}
\text{A, C, E, G} \\
\text{B, C, G, H}
\end{array} \quad \rightarrow \quad \text{A, B, H}
\]
Data Structures

How do we store a set of items?

- **insert**: add an item to the set
- **delete**: remove an item from the set
- **divide**: divide the set into two (approximately) equal sized pieces
- **union**: combine two sets
- **subtraction**: remove one set from another
- **intersection**: find the intersection of two sets
- **set difference**: find the items only in one set
How do we store a set of items?

- **insert**: add an item to the set
- **delete**: remove an item from the set
- **divide**: divide the set into two (approximately) equal sized pieces

Cost:

\[ n \text{ items} \rightarrow T_1 = O(\log n) \]
\[ T_\infty = O(\log n) \]
How do we store a set of items?

- **insert**: add an item to the set
- **delete**: remove an item from the set
- **divide**: divide the set into two (approximately) equal sized pieces
- **union**: combine two sets
- **subtraction**: remove one set from another
- **intersection**: find the intersection of two sets
- **set difference**: find the items only in one set

**Cost:**

$n$ items $\Rightarrow T_1 = O(\log n)$  
$T_\infty = O(\log n)$

**Sequential solution:**  
Any balanced binary search tree.
Data Structures

Cost: set 1 (n items), set 2 (m items), n > m

\[ T_1 = O(n + m) \]
\[ T_\infty = O(\log n + \log m) \]

- union: combine two sets
- subtraction: remove one set from another
- intersection: find the intersection of two sets
- set difference: find the items only in one set
Data Structures

Cost: set 1 (n items), set 2 (m items), n > m

- union: combine two sets
- subtraction: remove one set from another
- intersection: find the intersection of two sets
- set difference: find the items only in one set

$T_1 = O(n + m)$
$T_\infty = O(\log n + \log m)$

need linear time to examine all items in both sets!
**Parallel Sets**

Basic building block:

Balanced binary tree that supports four operations:

1. $\text{split}(T, k) \Rightarrow (T_1, T_2, x)$

   - $T_1$ contains all items $< k$
   - $T_2$ contains all items $> k$
   - $x = k$ if $k$ was in $T$

A, B, C, D, E, F, G

split(T, D)

A, B, C

D

E, F, G
Parallel Sets

Basic building block:

Balanced binary tree that supports four operations:

2. $\text{join}(T_1, T_2) \rightarrow T$

$\text{every item in } T_1 < \text{ every item in } T_2$

$A,C,E$ $\rightarrow$ $A,C,E,F,G,H$

$F,G,H$
Basic building block:

Balanced binary tree that supports four operations:

2. $\text{join}(T_1, T_2) \rightarrow T$

Note: easier than Union operation because trees are ordered and disjoint!

- $A, C, E$
- $F, G, H$

$\text{every item in } T_1 < \text{every item in } T_2$
Basic building block:

Balanced binary tree that supports four operations:

3. root(T) ➔ item at root

Tree T is unchanged.
Root is approximate median.

A,C,E,G,H,J,K ➔ G
Parallel Sets

Basic building block:

Balanced binary tree that supports four operations:

4. insert(T, x) \(\Rightarrow\) T'  
   Tree T' = T with x inserted.

\[ A,C,E,G,H,J,K \]

\[ \text{insert}(F) \]

\[ A,C,E,F,G,H,J,K \]
Parallel Sets

Basic building block:

**Balanced binary tree that supports four operations:**

1. split(T, k) $\rightarrow$ (T1, T2, x)
2. join(T1, T2) $\rightarrow$ T
3. root(T) $\rightarrow$ x
4. insert(T, x) $\rightarrow$ T’
Basic building block:

Balanced binary tree that supports four operations:

1. split(T, k) \rightarrow (T_1, T_2, x)
2. join(T_1, T_2) \rightarrow T
3. root(T) \rightarrow x
4. insert(T, x) \rightarrow T'

Can implement all four operations with a (2,4)-tree with:
- Work: $O(\log n + \log m)$
- Span: $O(\log n + \log m)$
Parallel Sets

Basic building block:

Balanced binary tree that supports four operations:

1. split(T, k) \(\Rightarrow\) (T1, T2, x)
2. join(T1, T2) \(\Rightarrow\) T
3. root(T) \(\Rightarrow\) x
4. insert(T, x) \(\Rightarrow\) T’

Can implement all four operations with a \((2,4)\)-tree with:
- Work: \(O(\log n + \log m)\)
- Span: \(O(\log n + \log m)\)

Exercise!
Data Structures

How do we store a set of items?

- **insert**: add an item to the set
- **delete**: remove an item from the set
- **divide**: divide the set into two (approximately) equal sized pieces
- **union**: combine two sets
- **subtraction**: remove one set from another
- **intersection**: find the intersection of two sets
- **set difference**: find the items only in one set

**Example:**

```plaintext
delete(T, k):
    (T1, T2, x) = split(T, k)
    T = join(T1, T2)
```
How do we store a set of items?

- **insert**: add an item to the set
- **delete**: remove an item from the set
- **divide**: divide the set into two (approximately) equal sized pieces
- **union**: combine two sets
- **subtraction**: remove one set from another
- **intersection**: find the intersection of two sets
- **set difference**: find the items only in one set

**Example:**

divide(T, k):

- \( k = \text{root}(T) \)
- \((T1, T2, x) = \text{split}(T, k)\)
- \(T2 = \text{insert}(T2, k)\)
Parallel Sets

Union(T1, T2)

if T1 = null: return T2
if T2 = null: return T1

...
Parallel Sets

Union(T1, T2)

\[
\text{if } T1 = \text{null: return } T2 \\
\text{if } T2 = \text{null: return } T1 \\
\text{key} = \text{root}(T1) \\
(L, R, x) = \text{split}(T2, \text{key}) \\
\text{fork:}
\]

\[
\ldots
\]
Parallel Sets

Union(T1, T2)

   if T1 = null: return T2
   if T2 = null: return T1
   key = root(T1)
   (L, R, x) = split(T2, key)
   fork:
   ...

...
Parallel Sets

Union(T1, T2)

if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, R, x) = split(T2, key)

fork:
...

...
Parallel Sets

Union(T1, T2)

\[
\text{if } T1 = \text{null}: \text{return } T2 \\
\text{if } T2 = \text{null}: \text{return } T1 \\
\text{key} = \text{root}(T1) \\
(L, G, x) = \text{split}(T2, \text{key}) \\
\text{fork:} \\
1. \quad TL = \text{Union}(\text{key.left}, L) \\
2. \quad TR = \text{Union}(\text{key.right}, R) \\
\text{sync} \\
\ldots
\]
Union(T1, T2)

if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)
fork:
1. TL = Union(key.left, L)
2. TR = Union(key.right, R)
sync
...

Parallel Sets
Parallel Sets

Union(T1, T2)

if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)
fork:
1. TL = Union(key.left, L)
2. TR = Union(key.right, R)
sync

TR = recursive right union
TL = recursive left union
Union(T1, T2)

if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)
fork:
1. TL = Union(key.left, L)
2. TR = Union(key.right, R)
sync
T = join(TL, TR)
insert(T, key)
return T
Parallel Sets

Union(T1, T2)

    if T1 = null: return T2
    if T2 = null: return T1
    key = root(T1)
    (L, G, x) = split(T2, key)
    fork:
    1. TL = Union(key.left, L)
    2. TR = Union(key.right, R)
    sync
    T = join(TL, TR)
    insert(T, key)
    return T
Union(T1, T2)

if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)

fork:
1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync
T = join(TL, TR)
insert(T, key)
return T
Union(T1, T2)

if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)

fork:
1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync
T = join(TL, TR)
insert(T, key)

return T
Union(T1, T2)

if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)

fork:
1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync
T = join(TL, TR)
insert(T, key)
return T

Work Analysis

O(log n + log m)
Work Analysis

Union(T1, T2)

if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)

fork:
1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync
T = join(TL, TR)
insert(T, key)
return T

Recursive calls where T1 is half the size.
Work Analysis

Union(T1, T2)

if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)

fork:
1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync
T = join(TL, TR)
insert(T, key)
return T

Recursive calls where T1 is half the size.

\[
T(n, m) = 2T(n/2, m) + O(\log n + \log m) = O(n \log m)
\]
Work Analysis

Union(T1, T2)

if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)

fork:
1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync
T = join(TL, TR)
insert(T, key)
return T

Lying (a little):
Left and right subtrees are not exactly sized n/2.

Still true...

$$T(n, m) = 2T(n/2, m) + O(\log n + \log m)$$
$$= O(n \log m)$$
Work Analysis

Union(T1, T2)

if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)

fork:
1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync
T = join(TL, TR)
insert(T, key)
return T

Be more careful
if m < n then:

Work = O(m log(n/m))

\[
T(n, m) = 2T(n/2, m) + O(\log n + \log m) \\
= O(n \log m)
\]
**Span Analysis**

**Union(T1, T2)**

if \( T1 = \text{null} \): \textbf{return} \( T2 \)

if \( T2 = \text{null} \): \textbf{return} \( T1 \)

\( \text{key} = \text{root}(T1) \)

\((L, G, x) = \text{split}(T2, \text{key})\)

**fork:**

1. \( TL = \text{Union}(\text{key.left}, L) \)
2. \( TR = \text{Union}(\text{key.right}, R) \)

**sync**

\( T = \text{join}(TL, TR) \)

\text{insert}(T, \text{key})

\text{return } T

---

Recursive calls where \( T1 \) is half the size.

\[
S(n, m) = T(n/2, m) + O(\log n + \log m)
\]

\[
= O(\log^2 n)
\]
Span Analysis

Union(T1, T2)

if T1 = null: return T2
if T2 = null: return T1

key = root(T1)
(L, G, x) = split(T2, key)

fork:
1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync
T = join(TL, TR)
insert(T, key)
return T

Use a different type of model / scheduler:
if m < n then:
Span = O(log n)

\[ S(n, m) = T(n/2, m) + O(\log n + \log m) \]
\[ = O(\log^2 n) \]
Union(T1, T2)

if T1 = null: return T2
if T2 = null: return T1

key = root(T1)

(L, G, x) = split(T2, key)

fork:
1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync

T = join(TL, TR)
insert(T, key)

return T

Span Analysis

\[ S(n, m) = T(n/2, m) + O(\log n + \log m) \]
\[ = O(\log^2 n) \]

Use a different type of model / scheduler:
if m < n then:
Span = O(log n)

Not in CS5234
Other operations?
Intersection(T1, T2)

    if T1 = null: return null
    if T2 = null: return null

key = root(T1)
(L, G, x) = split(T2, key)

fork:
1. TL = Intersection(key.left, L)
2. TR = Intersection(key.right, R)

sync
T = join(TL, TR)
if (x = key) then insert(T, key)
return T
SetDifference(T1, T2)

* if T1 = null: return T2
* if T2 = null: return T1

key = root(T1)
(L, G, x) = split(T2, key)

fork:
1. TL = Intersection(key.left, L)
2. TR = Intersection(key.right, R)

sync
T = join(TL, TR)
if (x = null) then insert(T, key)
return T
Problem: Breadth First Search

Searching a graph:

- undirected graph $G = (V,E)$
- source node $s$
Problem: Breadth First Search

Searching a graph:

• undirected graph $G = (V,E)$
• source node $s$

• assume each node stores its adjacency list as a (parallel) set, using the data structure from before.
Problem: Breadth First Search

Searching a graph:

- undirected graph $G = (V,E)$
- source node $s$

Layer-by-layer...
Sequential Algorithm

BFS(G, s)

F = \{s\}

repeat until F = \{

F' = \{

for each u in F:

visited[u] = true

for each neighbor v of u:

if (visited[v] = false) then F'.insert(v)

F = F'

}
Sequential Algorithm

BFS(G, s)

\[
\begin{align*}
F & = \{s\} \\
\text{repeat until } & F = \{} \\
F' & = \{} \\
\text{for each } & u \text{ in } F: \\
\quad & \text{visited}[u] = \text{true} \\
\text{for each neighbor } & v \text{ of } u: \\
\quad & \text{if } (\text{visited}[v] = \text{false}) \text{ then} \\
\quad & \text{F'.insert}(v) \\
F & = F'
\end{align*}
\]

Problems to solve:
- need to do parallel exploration of the frontier
- visited is hard to maintain in parallel
Parallel Algorithm

parBFS(G, s)

\[
\begin{align*}
F &= \{s\} \\
D &= \{
\end{align*}
\]

repeat until F = {}

\[
\begin{align*}
D &= \text{Union}(D, F) \\
F &= \text{ProcessFrontier}(F) \\
F &= \text{SetSubtraction}(F, D)
\end{align*}
\]

F and D are parallel sets, built using the parallel data structure we saw earlier!
Parallel Algorithm

parBFS(G, s)

\[ F = \{ s \} \]
\[ D = \{ \} \]

repeat until \( F = \{ \} \)

\[ D = \text{Union}(D, F) \]
\[ F = \text{ProcessFrontier}(F) \]
\[ F = \text{SetSubtraction}(F, D) \]

Mark everything already explored as done.
Parallel Algorithm

\[ \text{parBFS}(G, s) \]

\[
\begin{align*}
F &= \{s\} \\
D &= \{} \\
\text{repeat until } F &= \{} \\
& \quad D = \text{Union}(D, F) \\
& \quad F = \text{ProcessFrontier}(F) \\
& \quad F = \text{SetSubtraction}(F, D)
\end{align*}
\]

Mark everything already explored as done.

Explore all the neighbors of every node in \( F \).
**Parallel Algorithm**

parBFS(G, s)

1. \[ F = \{s\} \]
2. \[ D = \{} \]
3. **repeat until** \[ F = \{} \]
   1. \[ D = \text{Union}(D, F) \]
   2. \[ F = \text{ProcessFrontier}(F) \]
   3. \[ F = \text{SetSubtraction}(F, D) \]

---

Mark everything already explored as done.

Explore all the neighbors of every node in F.

Remove already visited nodes from the new frontier.
Parallel Algorithm

ProcessFrontier(F)

\[
\text{if } |F| = 1 \text{ then} \\
\quad u = \text{root}(F) \\
\quad \text{return } u.\text{neighbors} \\
\text{else} \\
\quad (F1, F2) = \text{divide}(F) \\
\text{fork:} \\
\quad 1. \quad F1 = \text{ProcessFrontier}(F1) \\
\quad 2. \quad F2 = \text{ProcessFrontier}(F2) \\
\text{sync} \\
\text{return } \text{Union}(F1, F2)
\]

Base case: return the set containing the neighbors of one node.
Parallel Algorithm

ProcessFrontier(F)

if $|F| = 1$ then
    $u = \text{root}(F)$
    return $u$.neighbors
else
    $(F_1, F_2) = \text{divide}(F)$
    fork:
    1. $F_1 = \text{ProcessFrontier}(F_1)$
    2. $F_2 = \text{ProcessFrontier}(F_2)$
    sync
    return $\text{Union}(F_1, F_2)$

Base case: return the set containing the neighbors of one node.

Divide the set (approximately) in half.
Parallel Algorithm

ProcessFrontier(F)

\[
\text{if } |F| = 1 \text{ then} \\
\quad u = \text{root}(F) \\
\quad \text{return } u.\text{neighbors} \\
\text{else} \\
(\text{F1, F2}) = \text{divide}(F) \\
\text{fork:} \\
1. \text{ F1} = \text{ProcessFrontier(F1)} \\
2. \text{ F2} = \text{ProcessFrontier(F2)} \\
\text{sync} \\
\text{return } \text{Union}(F1, F2)
\]

- **Base case**: return the set containing the neighbors of one node.
- **Divide**: the set (approximately) in half.
- **Fork**: recursively process the two frontiers.
Parallel Algorithm

ProcessFrontier(F)

if |F| = 1 then
  u = root(F)
  return u.neighbors
else
  (F1, F2) = divide(F)
  fork:
      1. F1 = ProcessFrontier(F1)
      2. F2 = ProcessFrontier(F2)
  sync
  return Union(F1, F2)

Base case: return the set containing the neighbors of one node.
Divide the set (approximately) in half.
Recursively process the two frontiers.
Merge the two frontiers and return.
ProcessFrontier(F)

if |F| = 1 then
  u = root(F)
  return u.neighbors
else
  (F1, F2) = divide(F)
  fork:
    1. F1 = ProcessFrontier(F1)
    2. F2 = ProcessFrontier(F2)
  sync
  return Union(F1, F2)
ProcessFrontier(F)

if |F| = 1 then
  u = root(F)
  return u.neighbors
else
  (F1, F2) = divide(F)
  fork:
    1. F1 = ProcessFrontier(F1)
    2. F2 = ProcessFrontier(F2)
  sync
  return Union(F1, F2)

Work Analysis

n = nodes in F
m = # adjacent edges to F

$O(1)$
**Work Analysis**

ProcessFrontier(F)

\[
\text{if } |F| = 1 \text{ then } \\
\quad u = \text{root}(F) \\
\quad \text{return } u.\text{neighbors} \\
\text{else } \\
\quad (F1, F2) = \text{divide}(F) \\
\text{fork: } \\
\quad 1. \quad F1 = \text{ProcessFrontier}(F1) \\
\quad 2. \quad F2 = \text{ProcessFrontier}(F2) \\
\quad \text{sync} \\
\quad \text{return } \text{Union}(F1, F2)
\]

\[O(1)\]

\[O(\log n)\]

n = nodes in F
m = # adjacent edges to F
ProcessFrontier(F)

if |F| = 1 then
  u = root(F)
  return u.neighbors
else
  (F1, F2) = divide(F)

  fork:
  1. F1 = ProcessFrontier(F1)
  2. F2 = ProcessFrontier(F2)

  sync

return Union(F1, F2)

$n = \text{nodes in } F$

$m = \# \text{ adjacent edges to } F$

$O(1)$

$O(\log n)$

Two recursive calls of size approximately $n/2$. 
ProcessFrontier(F)

if |F| = 1 then
    u = root(F)
    return u.neighbors
else
    (F1, F2) = divide(F)
    fork:
    1. F1 = ProcessFrontier(F1)
    2. F2 = ProcessFrontier(F2)
    sync
    return Union(F1, F2)
\[ W(n, m) = 2W(n/2, m) + O(m \log m) + O(\log n) \]
\[ = O(m \log n \log m) \]
\[ = O(m \log^2 n) \]

\begin{itemize}
  \item \( u = \text{root}(F) \)
  \item \textbf{return} \( u.\text{neighbors} \)
\end{itemize}

\textbf{else}

\begin{itemize}
  \item \((F_1, F_2) = \text{divide}(F)\)
  \item \textbf{fork:}
    \begin{enumerate}
      \item \( F_1 = \text{ProcessFrontier}(F_1) \)
      \item \( F_2 = \text{ProcessFrontier}(F_2) \)
    \end{enumerate}
  \item \textbf{sync}
  \item \textbf{return} \( \text{Union}(F_1, F_2) \)
\end{itemize}

\( O(1) \)
\( O(\log n) \)
\( O(m \log m) \)

Two recursive calls of size approximately \( n/2 \).
Span Analysis

ProcessFrontier(F)

\[
\text{if } |F| = 1 \text{ then }
\]
\[u = \text{root}(F)\]
\[\text{return } u.\text{neighbors}\]

else
\[(F_1, F_2) = \text{divide}(F)\]

fork:
1. \(F_1 = \text{ProcessFrontier}(F_1)\)
2. \(F_2 = \text{ProcessFrontier}(F_2)\)

sync

\[\text{return } \text{Union}(F_1, F_2)\]

\[n = \text{nodes in } F\]
\[m = \# \text{ adjacent edges to } F\]

\[O(1)\]

\[O(\log n)\]

One recursive calls
of size approximately \(n/2\).

\[O(\log^2 m)\]
\[
S(n, m) = S(n/2, m) + O(\log^2 m) + O(\log n)
\]
\[
= O(\log n \log^2 m)
\]
\[
= O(\log^3 n)
\]

\begin{align*}
\text{u} &= \text{root}(F) \\
\text{return u.neighbors} \\
\text{else} \\
(F1, F2) &= \text{divide}(F) \\
\text{fork:} \\
1. & \quad F1 = \text{ProcessFrontier}(F1) \\
2. & \quad F2 = \text{ProcessFrontier}(F2) \\
\text{sync} \\
\text{return Union}(F1, F2)
\end{align*}

\begin{align*}
\text{O(1)} & \quad \text{One recursive calls of size approximately n/2.} \\
\text{O(\log n)} & \quad \text{O(\log^2 m)}
\end{align*}
parBFS(G, s)

F = {s}
D = {}

repeat until F = {}
  D = Union(D, F)
  F = ProcessFrontier(F)
  F = SetSubtraction(F, D)

Work: $O(m \log^2 n)$
Span: $O(\log^3 n)$
Work Analysis

parBFS(G, s)

\[
\begin{align*}
F &= \{s\} \\
D &= \{} \\
\text{repeat until } F &= \{} \\
D &= \text{Union}(D, F) \\
F &= \text{ProcessFrontier}(F) \\
F &= \text{SetSubtraction}(F, D)
\end{align*}
\]

- \(O(m \log n)\)
- \(O(m \log^2 n)\)

Note: every edge appears in at most two iterations!

Note: every node appears in at most one frontier.

\(F_j = \text{number of nodes in frontier in } j\text{th iteration.}\)
parBFS(G, s)

\[
\begin{align*}
F &= \{s\} \\
D &= \{\} \\
\text{repeat until } F &= \{\} \\
D &= \text{Union}(D, F) \\
F &= \text{ProcessFrontier}(F) \\
F &= \text{SetSubtraction}(F, D)
\end{align*}
\]

Work Analysis

\[
T_1(n, m) = O(m \log^2 n)
\]

Note: every edge appears in at most two iterations!

Note: every node appears in at most one frontier.

\[F_j = \text{number of nodes in frontier in } j\text{th iteration.}\]
Span Analysis

parBFS(G, s)

\[ F = \{s\} \]
\[ D = \{\} \]

repeat until \( F = \{\} \)

\[ D = \text{Union}(D, F) \]
\[ F = \text{ProcessFrontier}(F) \]
\[ F = \text{SetSubtraction}(F, D) \]

\[ O(\log^2 m) \]
\[ O(\log^3 m) \]
\[ O(\log^3 m) \]

Assume the graph has diameter \( D \).

\[ T_\infty = D \log^3 m \]

Hard to do better than \( D \).
Problem: Breadth First Search

Searching a graph:

• undirected graph $G = (V,E)$
• source node $s$

Layer-by-layer...
Problem: Breadth First Search

Searching a graph:

- undirected graph $G = (V,E)$
- source node $s$

$$T_p = O \left( \frac{m \log^2 n}{p} + D \log^3 m \right)$$
Problem: Breadth First Search

Searching a graph:

- undirected graph $G = (V,E)$
- source node $s$

$$T_p = O \left( \frac{m \log^2 n}{p} + D \log^3 m \right)$$

Interpretation:
With a good scheduler and enough processors, you can perform a BFS in time roughly proportional to the diameter.
Problem: Breadth First Search

Searching a graph:

- undirected graph \( G = (V,E) \)
- source node \( s \)

\[
T_p = O \left( \frac{m \log^2 n}{p} + D \log^3 m \right)
\]

Caveat:
Only useful if \( p > \log^2 n \).
Searching a graph:

- undirected graph $G = (V,E)$
- source node $s$
- search graph in depth-first order

$\Rightarrow$ Best we know is $\Omega(n)$

Why does DFS seem so much harder than BFS?
Summary

Today: Parallelism

Models of Parallelism
• How to predict the performance of algorithms?

Some simple examples...

Sorting
• Parallel MergeSort

Trees and Graphs

Last Week: Caching

Breadth-First-Search
• Sorting your graph
MIS
• Luby’s Algorithm
• Cache-efficient implementation
MST
• Connectivity
• Minimum Spanning Tree