Measurements can be expensive!
Yet necessary to reach certain confidence

How to determine the minimal number of measurements?

- Measure until the confidence interval has a certain acceptable width
  - For example, measure until the 95% CI is within 5% of the mean/median
  - Can be computed analytically assuming normal data
  - Compute iteratively for nonparametric statistics

- Often heard: “we cannot afford more than a single measurement”
  - E.g., Gordon Bell runs
  - Well, then one cannot say anything about the variance
  - Even 3-4 measurement can provide very tight CI (assuming normality)

Can be used to proof optimality of implementation

We recommend factorial design

Compute iteratively for nonparametric statistics

Model computer system as k-dimensional space
- Each dimension represents a capability
  - Floating point, integer, memory bandwidth, cache bandwidth, etc.
- Features are typical rates
- Determine maximum rate for each dimension
  - E.g., from documentation or benchmarks

Can be used to proof optimality of implementation
- If the requirements of the bottleneck dimension are minimal

Features are typical rates
Can be computed analytically assuming normal data

Synchronize processes
This is hard in practice and not easy to capture in rules

Measure events separately
- Use high-precision timers
- Synchronize processes
- Summarize across processes:
  - Min/max (unstable), average, median – depends on use-case
Final project presentation: next Monday 12/17 during lecture

- Report will be due in January!
  - Starting to write early is very helpful — write — rewrite — rewrite (no panic)

- Coordinate your talk! You have 10 minutes (8 talk + 2 Q&A)

Sums and prefix sums on linked lists

- Connected components
- Distributed algorithms
- Broadcast in alpha-beta and LogP

Obvious question: is there a depth- and work-optimal algorithm?

- This took years to settle! The answer is surprisingly: no

- Coordinate your talk! You have 10 minutes (8 talk + 2 Q&A)

Strict optimality

- Work/depth tradeoffs and bounds
- Applications of prefix sums
- Parallelization according to algorithms

Learning goals for today

- Strict optimality
- Work/depth tradeoffs and bounds
- Applications of prefix sums
- Parallelization according to algorithms

Review of last lecture(s)

- Impossibility of wait-free consensus with atomic registers
  - “perhaps one of the most striking impossibility results in Computer Science” (Herlihy, Shavit)
- Large-scale locks
  - Scaling MCS to thousands of nodes (MPI/RMA)
- Oblivious algorithms
  - Execution oblivious vs. structural oblivious
- Why do we care about obliviousness?
  - Strict optimality of work and depth — reduction ⊕ scan ⊕ Linear scan, tree scan, dissemination scan, surprising work-depth tradeoff \( W + D \geq 2n - 2 \)
- I/O complexity
  - The red-blue pebble game (four rules: input, output, compute, delete)
  - S partitioning proof
  - Geometric arguments for dense linear algebra – example matrix multiplication
  - Loomis Whitney inequality: \[ |F| \leq (|E| + |F| + |G|) \text{ (is set is smaller than sqrt of the sum of orthogonal projections)} \]
- Simple recomputation – trade off I/O for compute

Recap: Work-depth tradeoff in parallel prefix sums

- Obvious question: is there a depth- and work-optimal algorithm?
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: \( W + D \geq 2n - 2 \)

- Coordinate your talk! You have 10 minutes (8 talk + 2 Q&A)

Strict optimality

- Work/depth tradeoffs and bounds
- Applications of prefix sums
- Parallelization according to algorithms

Learning goals for today

- Strict optimality
- Work/depth tradeoffs and bounds
- Applications of prefix sums
- Parallelization according to algorithms

Review of last lecture(s)

- Impossibility of wait-free consensus with atomic registers
  - “perhaps one of the most striking impossibility results in Computer Science” (Herlihy, Shavit)
- Large-scale locks
  - Scaling MCS to thousands of nodes (MPI/RMA)
- Oblivious algorithms
  - Execution oblivious vs. structural oblivious
- Why do we care about obliviousness?
  - Strict optimality of work and depth — reduction ⊕ scan ⊕ Linear scan, tree scan, dissemination scan, surprising work-depth tradeoff \( W + D \geq 2n - 2 \)
- I/O complexity
  - The red-blue pebble game (four rules: input, output, compute, delete)
  - S partitioning proof
  - Geometric arguments for dense linear algebra – example matrix multiplication
  - Loomis Whitney inequality: \[ |F| \leq (|E| + |F| + |G|) \text{ (is set is smaller than sqrt of the sum of orthogonal projections)} \]
- Simple recomputation – trade off I/O for compute

Recap: Work-depth tradeoff in parallel prefix sums

- Obvious question: is there a depth- and work-optimal algorithm?
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: \( W + D \geq 2n - 2 \)

- Coordinate your talk! You have 10 minutes (8 talk + 2 Q&A)

Strict optimality

- Work/depth tradeoffs and bounds
- Applications of prefix sums
- Parallelization according to algorithms

Learning goals for today

- Strict optimality
- Work/depth tradeoffs and bounds
- Applications of prefix sums
- Parallelization according to algorithms

Review of last lecture(s)

- Impossibility of wait-free consensus with atomic registers
  - “perhaps one of the most striking impossibility results in Computer Science” (Herlihy, Shavit)
- Large-scale locks
  - Scaling MCS to thousands of nodes (MPI/RMA)
- Oblivious algorithms
  - Execution oblivious vs. structural oblivious
- Why do we care about obliviousness?
  - Strict optimality of work and depth — reduction ⊕ scan ⊕ Linear scan, tree scan, dissemination scan, surprising work-depth tradeoff \( W + D \geq 2n - 2 \)
- I/O complexity
  - The red-blue pebble game (four rules: input, output, compute, delete)
  - S partitioning proof
  - Geometric arguments for dense linear algebra – example matrix multiplication
  - Loomis Whitney inequality: \[ |F| \leq (|E| + |F| + |G|) \text{ (is set is smaller than sqrt of the sum of orthogonal projections)} \]
- Simple recomputation – trade off I/O for compute

Recap: Work-depth tradeoff in parallel prefix sums

- Obvious question: is there a depth- and work-optimal algorithm?
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: \( W + D \geq 2n - 2 \)

- Coordinate your talk! You have 10 minutes (8 talk + 2 Q&A)

Strict optimality

- Work/depth tradeoffs and bounds
- Applications of prefix sums
- Parallelization according to algorithms

Learning goals for today

- Strict optimality
- Work/depth tradeoffs and bounds
- Applications of prefix sums
- Parallelization according to algorithms

Review of last lecture(s)

- Impossibility of wait-free consensus with atomic registers
  - “perhaps one of the most striking impossibility results in Computer Science” (Herlihy, Shavit)
- Large-scale locks
  - Scaling MCS to thousands of nodes (MPI/RMA)
- Oblivious algorithms
  - Execution oblivious vs. structural oblivious
- Why do we care about obliviousness?
  - Strict optimality of work and depth — reduction ⊕ scan ⊕ Linear scan, tree scan, dissemination scan, surprising work-depth tradeoff \( W + D \geq 2n - 2 \)
- I/O complexity
  - The red-blue pebble game (four rules: input, output, compute, delete)
  - S partitioning proof
  - Geometric arguments for dense linear algebra – example matrix multiplication
  - Loomis Whitney inequality: \[ |F| \leq (|E| + |F| + |G|) \text{ (is set is smaller than sqrt of the sum of orthogonal projections)} \]
- Simple recomputation – trade off I/O for compute

Recap: Work-depth tradeoff in parallel prefix sums

- Obvious question: is there a depth- and work-optimal algorithm?
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: \( W + D \geq 2n - 2 \)

- Coordinate your talk! You have 10 minutes (8 talk + 2 Q&A)

Strict optimality

- Work/depth tradeoffs and bounds
- Applications of prefix sums
- Parallelization according to algorithms

Learning goals for today

- Strict optimality
- Work/depth tradeoffs and bounds
- Applications of prefix sums
- Parallelization according to algorithms

Review of last lecture(s)

- Impossibility of wait-free consensus with atomic registers
  - “perhaps one of the most striking impossibility results in Computer Science” (Herlihy, Shavit)
- Large-scale locks
  - Scaling MCS to thousands of nodes (MPI/RMA)
- Oblivious algorithms
  - Execution oblivious vs. structural oblivious
- Why do we care about obliviousness?
  - Strict optimality of work and depth — reduction ⊕ scan ⊕ Linear scan, tree scan, dissemination scan, surprising work-depth tradeoff \( W + D \geq 2n - 2 \)
- I/O complexity
  - The red-blue pebble game (four rules: input, output, compute, delete)
  - S partitioning proof
  - Geometric arguments for dense linear algebra – example matrix multiplication
  - Loomis Whitney inequality: \[ |F| \leq (|E| + |F| + |G|) \text{ (is set is smaller than sqrt of the sum of orthogonal projections)} \]
- Simple recomputation – trade off I/O for compute

Recap: Work-depth tradeoff in parallel prefix sums

- Obvious question: is there a depth- and work-optimal algorithm?
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: \( W + D \geq 2n - 2 \)

- Coordinate your talk! You have 10 minutes (8 talk + 2 Q&A)

Strict optimality

- Work/depth tradeoffs and bounds
- Applications of prefix sums
- Parallelization according to algorithms

Learning goals for today

- Strict optimality
- Work/depth tradeoffs and bounds
- Applications of prefix sums
- Parallelization according to algorithms

Review of last lecture(s)

- Impossibility of wait-free consensus with atomic registers
  - “perhaps one of the most striking impossibility results in Computer Science” (Herlihy, Shavit)
- Large-scale locks
  - Scaling MCS to thousands of nodes (MPI/RMA)
- Oblivious algorithms
  - Execution oblivious vs. structural oblivious
- Why do we care about obliviousness?
  - Strict optimality of work and depth — reduction ⊕ scan ⊕ Linear scan, tree scan, dissemination scan, surprising work-depth tradeoff \( W + D \geq 2n - 2 \)
- I/O complexity
  - The red-blue pebble game (four rules: input, output, compute, delete)
  - S partitioning proof
  - Geometric arguments for dense linear algebra – example matrix multiplication
  - Loomis Whitney inequality: \[ |F| \leq (|E| + |F| + |G|) \text{ (is set is smaller than sqrt of the sum of orthogonal projections)} \]
- Simple recomputation – trade off I/O for compute

Recap: Work-depth tradeoff in parallel prefix sums

- Obvious question: is there a depth- and work-optimal algorithm?
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: \( W + D \geq 2n - 2 \)
Work-Depth Tradeoffs and deficiency

“The deficiency of a prefix circuit $c$ is defined as $\delta(c) = W(c) + D(c) - (2n - 2)$.”

But why do we care about this prefix sum so much?

- It’s the simplest problem to demonstrate and prove W-D tradeoffs
- And it’s one of the most important parallel primitives

Prefix sum as magic bullet for other seemingly sequential algorithms

- Any time a sequential chain can be modeled as function composition!
  - Let $f_1, ..., f_n$ be an ordered set of functions and $(f_1(x)) = x$
  - Define ordered function composition: $f_2(f_1(x)) = f_3(f_2(f_1(x))) = ... = f_n(x)
  - If we can write function composition $g(x) = f_1(x)f_2(x) ... f_n(x)$ as $h(x)$, then we can compute $c$ with a prefix sum!

We saw an example with the adder (Max were functions)

Example: linear recurrence $f(x) = a_1 f_1(x) + b_1 f_3(x)$

- Write as matrix form $f(x) = \begin{bmatrix} a_1 & b_1 \end{bmatrix} \begin{bmatrix} f_1(x) \\ f_2(x) \end{bmatrix}$

- Function composition is now simple matrix multiplication!

For example: $f(x) = (\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix}) x = \begin{bmatrix} 2x + 3 \\ x + 2 \end{bmatrix}$

- Most powerful! Homework:
  - Parallelize tridiagonal solve (e.g., Thomas’ algorithm)
  - Parallelize string parsing

Prefix sums as magic bullet for other seemingly sequential algorithms

- Any time a sequential chain can be modeled as function composition!
- Let $f_1, ..., f_n$ be an ordered set of functions and $(f_1(x)) = x$
- Define ordered function composition: $f_2(f_1(x)) = f_3(f_2(f_1(x))) = ... = f_n(x)
- If we can write function composition $g(x) = f_1(x)f_2(x) ... f_n(x)$ as $h(x)$, then we can compute $c$ with a prefix sum!

We saw an example with the adder (Max were functions)

Example: linear recurrence $f(x) = a_1 f_1(x) + b_1 f_3(x)$

- Write as matrix form $f(x) = \begin{bmatrix} a_1 & b_1 \end{bmatrix} \begin{bmatrix} f_1(x) \\ f_2(x) \end{bmatrix}$

- Function composition is now simple matrix multiplication!

For example: $f(x) = (\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix}) x = \begin{bmatrix} 2x + 3 \\ x + 2 \end{bmatrix}$

- Most powerful! Homework:
  - Parallelize tridiagonal solve (e.g., Thomas’ algorithm)
  - Parallelize string parsing

Prefix as magic bullet for other seemingly sequential algorithms

- Any time a sequential chain can be modeled as function composition!
- Let $f_1, ..., f_n$ be an ordered set of functions and $(f_1(x)) = x$
- Define ordered function composition: $f_2(f_1(x)) = f_3(f_2(f_1(x))) = ... = f_n(x)
- If we can write function composition $g(x) = f_1(x)f_2(x) ... f_n(x)$ as $h(x)$, then we can compute $c$ with a prefix sum!

We saw an example with the adder (Max were functions)

Example: linear recurrence $f(x) = a_1 f_1(x) + b_1 f_3(x)$

- Write as matrix form $f(x) = \begin{bmatrix} a_1 & b_1 \end{bmatrix} \begin{bmatrix} f_1(x) \\ f_2(x) \end{bmatrix}$

- Function composition is now simple matrix multiplication!

For example: $f(x) = (\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix}) x = \begin{bmatrix} 2x + 3 \\ x + 2 \end{bmatrix}$

- Most powerful! Homework:
  - Parallelize tridiagonal solve (e.g., Thomas’ algorithm)
  - Parallelize string parsing

Another use for prefix sums: Parallel radix sort

- Radix sort works bit-by-bit
  - Sorts k-bit numbers in k iterations
  - In each iteration j: stably sort all values by the j-th bit

Example: $k=1$: $a=011$, $b=101 
M_{101}, M_{011}, M_{110}$

- Scan computes: $M_{101} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$, $M_{011} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$, $M_{110} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$ in parallel
- All carry states and $c_i$ can now be computed in parallel by multiplying scan result with $q_0$.

Exercise: simplify!
Seems paradoxical but isn’t (may just not be most efficient)

Use adjacency matrix representation of graph — “compute with all zeros”

Unweighted graph – binary matrix

Weighted graph – general matrix

Construct distance matrix from adjacency matrix by replacing all off-diagonal
zeros with ∞

Initialize distance vector \(d_{ij}\) of size \(n\) everywhere but zero at start vertex

E.g., \(d_{ij} = \langle 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle\)

Show evolution when multiplied!

SSSP can be performed with \(n+1\) matrix-vector multiplications!

Question: Total work and depth?

Multiply the matrix (\(A + I\)) \(n\) times with itself in the Boolean semiring!

Why?

Demonstrate that \((A + I)^2\) has \(1\)s for each path of at most length 2

By induction show that \((A + I)^k\) has \(1\)s for each path of at most length \(k\)

What is work and depth of transitive closure?

Repeated squaring: \(W = (A + I)^0 = 0 = (0\times\log n)\)

How to get to connected components from a transitive closure matrix?

Each component needs unique label

Create label matrix \(L_{ij} = \langle 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle\)

For each column (vertex) perform min-reduction to determine its component label!

Overall work and depth?

Many if not all graph problems have oblivious or tensor variants!

Not clear whether they are most efficient

Efforts such as GraphBLAS exploit existing BLAS implementations and techniques

Generalisations to other algorithms possible

Can everything be modeled as tensor computations on the right ring?

E. Solomonik, TH: “Sparse Tensor Algebra as a Parallel Programming Model”

Much of machine learning/deep learning is oblivious

Many algorithms get non-oblivious though

All sparse algorithms are data-dependent!

E.g., use sparse graphs for graph algorithms on semirings \(\langle a+b, *, 0, 1 \rangle \)

May recover some of the lost efficiency by computing zeros!

Now moving to non-oblivious ☝

A semiring is an algebraic structure that

Has two binary operations called “addition” and “multiplication”

Addition must be associative \((a+b)+c = a+(b+c)\) and commutative \((a+b) = (b+a)\) and have an identity element

Multiplication must be associative and have an identity element

Multiplication distributes over addition \(a*(b+c) = a*b + a*c\) and multiplication by additive identity annihilates

Semirings are denoted by tuples \((\mathbb{F}, *, +, 0, 1)\)

“Standard” ring of rational numbers: \((\mathbb{Q}, +, *, 0, 1)\)

Boolean semiring: \((\{0,1\}, \lor, \land, 0, 1)\)

Tropical semiring: \((\mathbb{R}, \oplus, \ominus, 0, \infty)\) (also called min-plus semiring)
Reduction on a linked list

- Given: n values in linked list, looking for sum of all values

Sequential algorithm:

```c
typedef struct elem {
    int next; /*next*/
    int val; /*val*/
} elem;
```

A set \( S \subseteq \mathbb{N} \) is called an independent set if no two elements in \( S \) are connected.

- Are the following sets independent or not?
  - (1)
  - (1,3)
  - (1,3,5)

Class question: What is the maximum size of an independent set of a linked list with \( n \) elements?

Parallel reduction on a linked list

- Given: n values in linked list, looking for sum of all values

Parallel algorithm:

```c
typedef struct elem {
    int next; /*next*/
    int val; /*val*/
} elem;
```

A subset \( I \subseteq S \) is called an independent set of \( S \) if no two elements in \( I \) are connected!

- \( S \): \{1,5,3\}
- \( I \): \{1,5\}

Class question: Assuming picking a maximum \( I \) is free, what are work and depth?

\( W = n-1 \), \( D = \lceil \log_2 n \rceil \)

Is this optimal?

How to pick the independent set \( I \)?

- That's now the whole trick!
  - It's simple: if all linked values are consecutive in an array – same as "standard" reduction!
  - Can compute independent set up-front!

  **Irregular linked list though:**
  - Idea 3: find the order of elements requires parallel prefix sum, Doh!
  - Observation: if we pick \( |S| > 2^{|I|} \) in each iteration, we finish in logarithmic time!

Symmetry breaking:

- Assume \( p \) processes work on \( p \) consecutive nodes
- How to find the independent set?

- They all look the same (well, only the first and last differ, they have no left/right neighbor)
- Local decisions cannot be made!

- Introduce randomness to create local differences!
  - Each node tosses a coin \( \Pr \{ \text{true} \} = p \)
  - Let \( I \) be the set of nodes such that \( v \) drew 1 and \( v.next \) drew 0
  - Show that \( I \) is indeed independent!

  What is the probability that \( v \in I \)?

\[ \Pr \{ v \in I \} = \frac{1}{2} \]

Optimizations

- As the set shrinks, the random selection will get less efficient
  - When \( p \) is close to \( n \) (\( \log n \)), then most processors will fail to make useful progress
  - Switch to a different algorithm
  - Recursive doubling!

  ```c
  for (i = 0; i < \lceil \log n \rceil; ++i) {
    for (each elem do in parallel) {
      elem.val += elem.next.val;
      elem.next = elem.next.next;
    }
  }
  ```

  Class question: What are work and depth?

\[ W' = n - 1, D' = \lceil \log_2 n \rceil \]

Prefix summation on a linked list

- Didn't we just see it? Yes, but work-inefficient (if \( p < n \))!

  We extend the randomized symmetry-breaking reduction algorithms
  - First step: run the reduction algorithm as before
  - Second step: reintert in reverse order of deletion

  When reinterting, add the value of their successor

  Class question: how to implement this in practice?

  - Either recursion or a stack!
  - Design the algorithm as homework (using a parallel for loop)

Prefix summation on a linked list

- Didn't we just see it? Yes, but work-inefficient (if \( p < n \))!

  We extend the randomized symmetry-breaking reduction algorithms
  - First step: run the reduction algorithm as before
  - Second step: reintert in reverse order of deletion

  When reinterting, add the value of their successor

  Class question: how to implement this in practice?

  - Either recursion or a stack!
  - Design the algorithm as homework (using a parallel for loop)
Finding connected components as example

A connected component of an undirected graph is a subgraph in which any two vertices are connected by a path and no vertex in the subgraph is connected to any vertices outside the subgraph. Each undirected graph \( G = (V, E) \) contains one or multiple (at most \(|V|\)) connected components.

- Straight forward and cheap to compute sequentially – question: how?
  - Any traversal algorithm in work \( O(|V| + |E|) \)
  - Seemsingly trivial - becomes very interesting in parallel!
  - Our oblivious semiring-based algorithm was \( W = O(n^2 \log n), B = O((\log n)^2) \)

- Far from work optimality - Question: can we do better by dropping obliviousness?

Let’s start simple – assuming concurrent read/write is free

- Arbitrary write wins

- Concept of supervertices
  - A supervertex represents a set of vertices in a graph
  1. Initially, each vertex is a (singleton) supervertex
  2. Successively merge neighboring supervertices
  3. When no further merging is possible → each supervertex is a component
  4. Question is now only about the merging strategy

A fixpoint algorithm proceeds iteratively and monotonically until it reaches a final state that is not left by iterating further.

Shiloach/Vishkin’s algorithm

- Pointer graph/forest:
  - Define pointer array \( P[i] \) is a pointer from \( i \) to some other vertex
  - We call the graph defined by \( P \) (excluding self loops) the pointer graph
  - During the algorithm, \( P[i] \) forms a forest such that \( \forall i : (i, P[i]) \) there exists a path from \( i \) to \( P[i] \) in the original graph
  - Initially, all \( P[i] = i \)
  - The algorithm will run until each forest is a directed star pointing at the (smallest id) root of the component

- Supervertices:
  - Initially, each vertex is its own supervertex
  - Supervertices induce a graph - \( S_i \) and \( S_j \) are connected if \( (u, v) \in E \) with \( u \in S_i \) and \( v \in S_j \)
  - A supervertex is represented by its tree in \( P \)

- Correctness proofs:
  - Lemma 1: The shortcut operation converts rooted trees to rooted stars. Proof: obvious
  - Theorem 1: The pointer graph always forms a forest (set of rooted trees). Proof: shortcut doesn’t violate, hook works

- Performance proofs:
  - Lemma 2: The number of iterations of the outer loop is at most \( \log n \). Proof: consider connected component, if it has two supervertices before hook, number of supervertices is halved, if no hooking happens, component is done
  - Lemma 2: The number of iterations of the inner loop in shortcut is at most \( \log n \). Proof: consider tree of height \( h > 2 \) at some iteration, the height of the tree halves during that iteration
  - Corollary: Class question: work and depth?
    \[ W = O(n^2 \log n), B = O((\log n)^2) \] (assuming conflicts are free)

Distributed networking basics

- Familiar (non-HPC) network: Internet TCP/IP
  - Common model:
    - Source
    - Network
    - Destination

- Class Question: What parameters are needed to model the performance (including pipelining)?
  - Latency, Bandwidth, Injection Rate, Host Overhead
  - What network models do you know and what do they model?

Remember: A Simple Model for Communication

- Transfer time \( T(s) = \alpha + \beta s \)
  - \( \alpha = \text{startup time (latency)} \)
  - \( \beta = \text{cost per byte (bandwidth)=1/\beta} \)

- As \( s \) increases, bandwidth approaches \( 1/\beta \) asymptotically

- Convergence rate depends on \( \alpha \)
  - \( T(s) = \alpha/\beta \)

- Assuming no pipelining (new messages can only be issued from a process after all arrived)
Bandwidth vs. Latency

- \( s_1 = \alpha / \beta \) is often used to distinguish bandwidth- and latency-bound messages.
- \( s_1 \) is in the order of kilobytes on real systems.

Quick Example

- Simplest linear broadcast
  - One process has a data item to be distributed to all processes.
  - \( T(s) = (P - 1) \cdot (\alpha + \beta) = O(P) \)
- Class question: Do you know a faster method to accomplish the same?

Faster Trees?

- Class question: Can we broadcast faster than in a ternary tree?
  - Yes because each respective root is idle after sending three messages!
  - Those roots could keep sending!
- Class question: What about the runtime?
  - \( T(s) = (P - 2 + s/z)(\alpha + \beta z) \)
  - Compare 2-nomial tree with simple pipeline for \( \alpha = 10, \beta = 1, P = 4, s = 10^6, \) and \( z = 10^5 \):
    - 2,000,020 vs. 1,200,120
- Class question: Can we do better for given \( \alpha, \beta, P, s \)?
  - Combine pipeline and tree \( \rightarrow \) pipelined tree
- Class question: What is the runtime of the pipelined binary tree algorithm?
  - \( T \approx \left( \frac{1}{2} + \log_2(P) \right) \cdot 2 \cdot (\alpha + \beta s) \)
- Class question: What is the optimal \( z \)?
  - \( z_{opt} = \sqrt{\frac{P}{2(\log_2(P) - 2)}} \)

Very Large Message Broadcast

- Extreme case (P small, s large): simple pipeline
  - Split message into segments of size \( z \)
  - Send segments from PE \( i \) to PE \( i+1 \)
  - Class question: What is the runtime?
    - \( T(z) = (P - 2 + s/z)(\alpha + \beta z) \)
  - Compare 2-nomial tree with simple pipeline for \( \alpha = 10, \beta = 1, P = 4, s = 10^6, z = 10^5 \): 1,008,964
- Class question: What is the time for simple pipeline for \( \alpha = 10, \beta = 1, P = 4, s = 10^6, z_{opt} \)?
  - 1,008,964

Lower Bounds

- Class question: What is a simple lower bound on the broadcast time?
  - \( T_{LB} \geq \min(\log_2(P) \cdot s, P) \)
  - How close are the binomial tree for small messages and the pipeline for large messages (approximately)?
  - Bin. tree is a factor of \( \log_2(P) \) slower in bandwidth
  - Pipeline is a factor of \( P \) slower in latency
- Class question: What can we do for intermediate message sizes?
  - Combine pipeline and tree \( \rightarrow \) pipelined tree
- Class question: What is the runtime of the pipelined binary tree algorithm?
  - \( T \approx \left( \frac{1}{2} + \log_2(P) \right) \cdot 2 \cdot (\alpha + \beta s) \)
- Class question: What is the optimal \( z \)?
  - \( z_{opt} = \sqrt{\frac{P}{2(\log_2(P) - 2)}} \)
Towards an Optimal Algorithm

- What is the complexity of the pipelined tree with zopt for small s, large P and for large s, constant P?
  - Small messages, large P: s=1, z=1 (i.e., will give O(log P))
  - Large messages, constant P: assume α, β, P-constant, will give asymptotically O(β)

Bandwidth-optimal algorithms exist, e.g., Sanders et al. “Full Bandwidth Broadcast, Reduction and Scan with Only Two Trees” 2007

- Intuition: in binomial tree, all leaves (P/2) only receive data and never send → wasted bandwidth
  - Send along two simultaneous binary trees where the leafs of one tree are inner nodes of the other

- Construction needs to avoid endpoint congestion (makes it complex)

Can be improved with linear programming and topology awareness (talk to me if you’re interested)

Open Problems

- Look for optimal parallel algorithms (even in simple models!)
  - And then check the more realistic models
  - Useful optimization targets are MPI collective operations
    - Broadcast/Reduce, Scatter/Barrier, Alltoall, Allreduce, Allgatherv, Scan/Exscan, ...
  - Implementations of those (check current MPI libraries)
  - Useful also in scientific computations
    - Barriers that, linear algebra, FFT, ...

- Lots of work to do!
  - Contact me for thesis ideas (or check SPCL) if you like this topic
    - Usually involve optimization (ILP/LP) and clever algorithms (algebra) combined with practical experiments on large-scale machines (10,000+ processors)

The LogP Model

- Defined by four parameters:
  - L: an upper bound on the latency, or delay, incurred in communicating a message containing a word (or small number of words) from its source module to its target module.
  - o: the overhead, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations.
  - g: the gap, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g corresponds to the available per-processor communication bandwidth.
  - P: the number of processor/memory modules. We assume unit time for local operations and call it a cycle.

Simple Examples

- Sending a single message
  - T = 2o+L
- Ping-Pong Round-Trip
  - T_{RTT} = 4o+2L
- Transmitting n messages
  - T(n) = L+(n-1)*max(g, o) + 2o

Simplifications

- o is bigger than g on some machines
- g can be ignored (eliminates max() terms)
- be careful with multicore!
- Offloading networks might have very low o
- Can be ignored (not yet but hopefully soon)
- L might be ignored for long message streams
- If they are pipelined
- Account g also for the first message
- Eliminate “-1”
Benefits over Latency/Bandwidth Model
- Models pipelining
  - L/g messages can be "in flight"
- Captures state of the art (cf. TCP windows)
- Models computation/communication overlap
- Asynchronous algorithms
- Models endpoint congestion/overload
- Benefits balanced algorithms

Example: Broadcasts
- Class Question: What is the LogP running time for a linear broadcast of a single packet?
  \[ T_{\text{lin}} = L + (P-2)^* \text{max}(o,g) + 2o \]
- Class Question: Approximate the LogP runtime for a binary-tree broadcast of a single packet?
  \[ T_{\text{bin}} \leq \log_2 P \times (L + \text{max}(o,g) + 2o) \]
- Class Question: Approximate the LogP runtime for a k-ary tree broadcast of a single packet?
  \[ T_{k-n} \leq \log_k P \times (L + (k-1)\text{max}(o,g) + 2o) \]

Class Question: What is the LogP running time for a binomial tree broadcast of a single packet (assume \( L > g! \))?
\[ T_{\text{bin}} \leq \log_2 P \times (L + 2o) \]

Class Question: Approximate the LogP runtime for a k-polynomial tree broadcast of a single packet?
\[ T_{k-n} \leq \log_k P \times (L + (k-2)\text{max}(o,g) + 2o) \]

Class Question: What is the optimal k (assume \( o > g \))?
\[ \text{Derive by: } k-0 = o \times \ln(k_{opt}) - L/k_{opt} + o \text{ (solve numerically)} \]

For larger L, k grows and for larger o, k shrinks
- Models pipelining capability better than simple model!

Example: Optimal Broadcast
- Broadcast to \( P-1 \) processes
  - Each process who received the value sends it on; each process receives exactly once

This determines the maximum number of PEs (P(t)) that can be reached in time \( t \)
- P(t) can be computed with a generalized Fibonacci recurrence (assuming \( o > g \)):
  \[ P(t) = \begin{cases} 1 & t < 2o + L \\ P(t-o) + P(t-L-2o) & \text{otherwise} \end{cases} \]

Which can be bounded by (see (1)):
\[ \left\lfloor \frac{2^{t/2}}{t} \right\rfloor \leq P(t) \leq 2^t \]

A closed solution is an interesting open problem!

(1) Herfner et al.: "Scalable Communication Protocols for Dynamic Sparse Data Exchange" (Lemma 1)
The Bigger Picture

- We learned how to program shared memory systems
- Coherency & memory models & linearizability
- Locks as examples for reasoning about correctness and performance
- List-based sets as examples for lock-free and wait-free algorithms
- Consensus number
- We learned about general performance properties and parallelism
- Amdahl's and Gustafson's laws
- Little's law, Work-span,
- Balance principles & scheduling
- We learned how to perform model-based optimizations
- Distributed memory broadcast example with two models
- What next? MPI? OpenMP? UPC?
- Next-generation machines "merge" shared and distributed memory concepts - Partitioned Global Address Space (PGAS)

If you're interested in any aspect of parallel algorithms, programming, systems, or large-scale computing and are looking for a thesis, let us know! (and check our webpage http://spcl.inf.ethz.ch/SeMa)