Lecture 9: Finishing consensus, scalable lock study, and oblivious algorithms

Teaching assistant: Salvatore Di Girolamo

Motivational video: https://www.youtube.com/watch?v=qx2dRIQXnbs
The simplest networking question: ping pong latency!

The latency of Piz Dora is 1.77us!

How did you get to this?

I averaged 10 tests, it must be right!

Why do you think so? Can I see the data?

Rule 5: Report if the measurement values are deterministic. For nondeterministic data, report confidence intervals of the measurement.

- CIs allow us to compute the number of required measurements!
- Can be very simple, e.g., single sentence in evaluation: “We collected measurements until the 99% confidence interval was within 5% of our reported means.”
Rule 6: Do not assume normality of collected data (e.g., based on the number of samples) without diagnostic checking.

- Most events will slow down performance
  - Heavy right-tailed distributions
- The Central Limit Theorem only applies asymptotically
  - Some papers/textbook mention “30-40 samples”, don’t trust them!
Thou shalt not trust your system!

Look what data I got!

Clearly, the mean/median are not sufficient!

Try quantile regression!

S

D

Image credit: nersc.gov

Min: 1.57
Max: 7.2

Min: 1.48
Max: 11.59

Piz Dora

Median

Arithmetic Mean

99% CI (Mean)

99% CI (Median)

Pilatus

Median

Arithmetic Mean

99% CI (Mean)

99% CI (Median)
Wow, so Pilatus is better for (worst-case) latency-critical workloads even though Dora is expected to be faster.

**Rule 8:** Carefully investigate if measures of central tendency such as mean or median are useful to report. Some problems, such as worst-case latency, may require other percentiles.

**Administrivia**

- **Final project presentation:** last Monday 12/17 during lecture
  - Report will be due in January!
    
    *Starting to write early is very helpful --- write – rewrite – rewrite (no joke!)*

- Coordinate your talk! You have 10 minutes (8 talk + 2 Q&A)
  - *What happened since the intermediate report?*
  - *Focus on the key aspects (time is tight)!*
  - *Try to wrap up – only minor things left for final report.*
  - *Engage the audience 😊*

- Send slides by Sunday night (11:59pm Zurich time) to Salvatore!
  - *We will use a single (windows) laptop to avoid delays when switching*
  - *Expect only Windows (powerpoint) or a PDF viewer*
  - *The order of talks will again be randomized for fairness*
Review of last lecture(s)

- Lock implementation(s)
  - Advanced locks (CLH + MCS)

- Started impossibility of wait-free consensus with atomic registers
  - “perhaps one of the most striking impossibility results in Computer Science” (Herlihy, Shavit)
    
    *Will continue/finish proof today as starter!*

- Theoretical background for performance
  - Amdahl’s law
  - Models: PRAM, Work/Depth, simple alpha-beta (Hockney) model
  - Simple algorithms: reduce, scan, mergesort,
  - Brent’s scheduling lemma + Little’s law
  - Greedy scheduling + random work stealing

- Practical performance
  - Roofline and balance modeling for practical performance optimization
  - Vectorization
Learning goals for today

- Quickly recap consensus and first part of valence proof
  - impossibility of atomic registers for wait-free consensus
  - Complete proof together

- Case study about scalable locking
  - Complete correctness section!

- Oblivious algorithms
  - How do work-depth graphs relate to practice?

- Strict optimality
  - Work/depth tradeoffs and bounds

- Applications of prefix sums
  - Parallelize seemingly sequential algorithms
DPHPC Overview

- locality
  - caches
  - memory hierarchy

- parallelism
  - vector ISA
  - shared memory
  - distributed memory

- cache coherency

- memory models
- distributed algorithms
- locks
- lock free
- wait free
- linearizability
- group communications

- Amdahl's and Gustafson's law
- memory
  - $\alpha - \beta$
- PRAM
- LogP

I/O complexity
balance principles I
Little's Law
balance principles II
scheduling
Remember: lock-free vs. wait-free

- A locked method
  - May deadlock (methods may never finish)

- A lock-free method
  - Guarantees that infinitely often some method call finishes in a finite number of steps

- A wait-free method
  - Guarantees that each method call finishes in a finite number of steps (implies lock-free)

- Synchronization instructions are not equally powerful!
  - Indeed, they form an infinite hierarchy; no instruction (primitive) in level x can be used for lock-/wait-free implementations of primitives in level z>x.
Concept: Consensus Number

- Each level of the hierarchy has a “consensus number” assigned.
  - Is the maximum number of threads for which primitives in level x can solve the consensus problem

- The consensus problem:
  - Has single function: decide(v)
  - Each thread calls it at most once, the function returns a value that meets two conditions:
    - **consistency**: all threads get the same value
    - **validity**: the value is some thread’s input
  - Simplification: binary consensus (inputs in {0,1})
Understanding Consensus

- Can a particular class solve n-thread consensus wait-free?
  - A class C solves n-thread consensus if there exists a consensus protocol using any number of objects of class C and any number of atomic registers.
  - The protocol has to be wait-free (bounded number of steps per thread).
  - The consensus number of a class C is the largest n for which that class solves n-thread consensus (may be infinite).
  - Assume we have a class D whose objects can be constructed from objects out of class C. If class C has consensus number n, what does class D have?
Starting simple ...

- **Binary consensus with two threads (A, B)!**
  - Each thread moves until it decides on a value
  - May update shared objects
  - Protocol state = state of threads + state of shared objects
  - Initial state = state before any thread moved
  - Final state = state after all threads finished
  - States form a tree, wait-free property guarantees a finite tree
    
    Example with two threads and two moves each!

- **Define various states**
  - Bivalent, univalent, critical

- **Two helper lemmata**
  - Lemma 1: the initial state is bivalent
  - Lemma 2: every wait-free consensus protocol has a critical state
Atomic Registers

- Theorem [Herlihy’91]: Atomic registers have consensus number one
  - I.e., they cannot be used to solve even two-thread consensus! Really?

- Proof outline:
  - Assume arbitrary consensus protocol, thread A, B
  - Run until it reaches critical state where next action determines outcome (show that it must have a critical state first)
  - Show all options using atomic registers and show that they cannot be used to determine one outcome for all possible executions!
    1) Any thread reads (other thread runs solo until end)
    2) Threads write to different registers (order doesn’t matter)
    3) Threads write to same register (solo thread can start after each write)
Atomic Registers

- Theorem [Herlihy’91]: Atomic registers have consensus number one
- Corollary: It is impossible to construct a wait-free implementation of any object with consensus number of >1 using atomic registers
  - “perhaps one of the most striking impossibility results in Computer Science” (Herlihy, Shavit)
  - We need hardware atomics or Transactional Memory!
- Proof technique borrowed from:
  - Very influential paper, always worth a read!
    - Nicely shows proof techniques that are central to parallel and distributed computing!
Other Atomic Operations

- Simple RMW operations (Test&Set, Fetch&Op, Swap, basically all functions where the op commutes or overwrites) have consensus number 2!
  - Similar proof technique (bivalence argument)

- CAS and TM have consensus number $\infty$
  - Constructive proof:

```c
const int first = -1
volatile int thread = -1;
int proposed[n];

int decide(v) {
    proposed[tid] = v;
    if(CAS(thread, first, tid))
        return v; // I won!
    else
        return proposed[thread]; // thread won
}
```

- Machines providing CAS are asynchronous computation equivalents of the Turing Machine i.e., any concurrent object can be implemented in a wait-free manner (not necessarily fast!)
Now you know everything about parallel program correctness 😊

- At least a lot ... ;-)  
  - We’ll argue more about performance now!

- You have all the tools for:
  - Efficient locks
  - Efficient lock-based algorithms
  - Reasoning about parallelism!

- What now?
  - Now you understand practice and will appreciate theory
    - Wasn’t that all too messy ☹?
  - Focus on (parallel) performance, techniques, and algorithms

- But let’s start with another case study about locks
  - Research (best) paper published at a top-tier conference some years ago
    - So you get a feeling of the field – and deepen understanding of MCS locks in practice
Case study: Fast Large-scale Locking in Practice

Inuitive semantics

Various performance penalties

Proc p

lock
accesses
unlock

Proc q

lock
accesses
Locks: Challenges
Locks: Challenges

- We need intra- and inter-node topology-awareness
- We need to cover arbitrary topologies
We need to distinguish between readers and writers

We need flexible performance for both types of processes

What will we use in the design?
Ingredient 1 - MCS Locks

Mellor-Crummey and Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM TOCS'91
Ingredient 2 - Reader-Writer Locks
How to manage the design complexity?

How to ensure tunable performance?

What mechanism to use for efficient implementation?
REMOTE MEMORY ACCESS (RMA) PROGRAMMING

Process p
Memory
A
B

Process q
Memory
A
B

A put
get B
flush

Cray
BlueWaters

REMOTE MEMORY ACCESS PROGRAMMING

- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).
RMA-RW - Required Operations

### Process p

- **Memory**
  - 3
  - 3
  - 3

### Process q

- **Memory**
  - 6
  - 3
  - 9
  - 6
  - 6
  - 8

**Required Operations**

- put
- get
- Fetch-and-Add (FAA)
- replace
- Compare-and-Swap (CAS)
MPI RMA primer ([much] more in the recitation sessions)

- Windows expose memory
  - Created explicitly
- Remote accesses
  - Put, get
  - Atomics
    - Accumulate (also atomic Put)
    - Get_accumulate (also atomic Get)
    - Fetch and op (faster single-word get_accumulate)
    - Compare and swap
- Synchronization
  - Two modes: passive and active target
    - We use passive target today, similar to shared memory!
    - Synchronization: flush, flush_local
- Memory model
  - Unified (coherent) and separate (not coherent) view - it’s complicated but versatile
How to manage the design complexity?

How to ensure tunable performance?

What mechanism to use for efficient implementation?
How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers.

Readers and writers synchronize with a distributed counter (DC).

MCS queues form a distributed tree (DT).

Modular design

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC’16, best paper
How to ensure tunable performance?

Each DQ: fairness vs throughput of writers

DC: a parameter for the latency of readers vs writers

DT: a parameter for the throughput of readers vs writers

A tradeoff parameter for every structure

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC’16, best paper
Distributed MCS Queues (DQs) - Throughput vs Fairness

Larger $T_{L,i}$: more throughput at level $i$. Smaller $T_{L,i}$: more fairness at level $i$.

Each DQ: The maximum number of lock passings within a DQ at level $i$, before it is passed to another $T_{L,i}$ DQ at $i$.

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC’16, best paper
Distributed Tree of Queues (DT) - Throughput of readers vs writers

DT: The maximum number of consecutive lock passings within readers ($T_R$).

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC’16, best paper
Distributed Counter (DC) - Latency of readers vs writers

DC: every \( k \)th compute node hosts a partial counter, all of which constitute the DC.

\[ k = T_{DC} \]

A writer holds the lock \( b|x|y \)

Readers that arrived at the CS

Readers that left the CS

\[ T_{DC} = 1 \]

\[ T_{DC} = 2 \]
Design space

Higher throughput of writers vs readers

Design A
Design B

Locality vs fairness (for writers)

Lower latency of writers vs readers

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC’16, best paper
Lock Acquire by Readers

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation.

A writer holds the lock.

Readers that arrived at the CS.

Readers that left the CS.

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC’16, best paper.
Lock Acquire by Writers

Acquire the main lock

Acquire MCS

Acquire the main MCS lock

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC’16, best paper
EVALUATION

- CSCS Piz Daint (Cray XC30)
- 5272 compute nodes
- 8 cores per node
- 169TB memory

- Microbenchmarks: acquire/release: latency, throughput
- Distributed hashtable
Evaluation - Distributed Counter Analysis

Throughput, 2% writers
Single-operation benchmark

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC’16, best paper
Evaluation - Reader Threshold Analysis

Throughput, 0.2% writers,
Empty-critical-section benchmark

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC’16, best paper
Evaluation - Comparison to the State-of-the-Art

Evaluation - Comparison to the State-of-the-Art

Throughput, single-operation benchmark

Evaluation - Distributed Hashtable

20% writers

10% writers

Evaluation - Distributed Hashtable

2% of writers

0% of writers

Another application area - Databases

- MPI-RMA for distributed databases?

Hash-Join

Sort-Join

C. Barthels, et al., TH: Distributed Join Algorithms on Thousands of Cores presented in Munich, Germany, VLDB Endowment, Aug. 2017
Another application area - Databases

- MPI-RMA for distributed databases on Piz Daint
Another application area - Databases

- MPI-RMA for distributed databases on Piz Daint

C. Barthels, et al., TH: Distributed Join Algorithms on Thousands of Cores presented in Munich, Germany, VLDB Endowment, Aug. 2017
Now on to parallel algorithms!

- **Oblivious parallel algorithms**
  - Fixed structure work-depth graphs

- **Nonoblivious parallel algorithms**
  - Data-dependent structure work-depth graphs

- **Data movement and I/O complexity**
  - Communication complexity
"An algorithm is execution-oblivious if, for each problem size, the sequence of instructions executed, the set of memory locations read and the set of memory locations written by each executed instruction are determined by the input size and are independent of the values of the other inputs."

Execution oblivious or not?

- Quicksort?
- Prefix sum on an array?
- Simple dense matrix multiplication?
- Dense matrix vector product?
- Sparse matrix vector product?
- Queue-based breadth-first search?

```c
int reduce(int n, arr[n]) {
    for(int i=0; i<n; ++i)
        sum += arr[i];
}
```

```c
int findmin(int n, a[n]) {
    for(int i=1; i<n; i++)
        if(a[i]<a[0]) a[0] = a[i];
}
```

```c
int finditem(list_t list) {
    item = list.head;
    while(item.value!=0 && item.next!=NULL) {
        item = item.next;
    }
}
```
Obliviousness as property of an execution

“An algorithm is execution-oblivious if, for each problem size, the sequence of instructions executed, the set of memory locations read and the set of memory locations written by each executed instruction are determined by the input size and are independent of the values of the other inputs”

```c
int reduce(int n, arr[n]) {
    for(int i=0; i<n; ++i)
        sum += arr[i];
}

int findmin(int n, a[n]) {
    for(int i=1; i<n; i++)
        if(a[i]<a[0]) a[0] = a[i];
}

int finditem(list_t list)
    item = list.head;
    while(item.value!=0 && item.next!=NULL)
        item = item.next;
```

- Class question: Can an algorithm decide whether a program is oblivious or not?
  - Answer: no, proof similar to decision problem whether a program always outputs zero or not
Structural obliviousness as stronger property

“A program is **structurally-oblivious** if any value used in a conditional branch, and any value used to compute indices or pointers is structurally-dependent only in the input variable(s) that contains the problem size but not on any other input”

**Structurally oblivious or not?**

```c
int reduce(int n, arr[n]) {
    for(int i=0; i<n; ++i)
        sum += arr[i];
}

int oblivious(int n, a[n], b[n]) {
    for(int i=0; i<n; ++i) {
        x = a[i] + 1;
        if (x > a[i]) b[i] = 1;
        else b[i] = 2;
    }
}
```

```c
int finditem(list_t list) {
    item = list.head;
    while(item.value!=0 && item.next!=NULL)
        item=item.next;
}
```

- Clear that structurally oblivious programs are also execution oblivious
  - Can be programmatically (statically decided)
  - Sufficient for practical use

- The middle example is not structurally oblivious but execution oblivious
  - First branch is always taken (assuming no overflow) but static dependency analysis is conservative
Why obliviousness?

- We can easily reason about oblivious algorithms
  - Execution DAG can be constructed “statically”
  - We have done this in the last weeks intuitively but you never asked how to do it for BFS for example 😊

- Simple example (that you know): parallel summation
  - Question: what is $W(n)$ and $D(n)$ of sequential summation?
    \[ W(n)=D(n)=n-1 \]
  - Question: is this optimal? How would you define optimality?
    Separate for $W$ and $D$! Typically try to achieve both!
  - Question: what is $W(n)$ and $D(n)$ of the optimal parallel summation?
    \[ W(n)=n-1 \quad D(n)=\lceil \log_2 n \rceil \]
    Are both $W$ and $D$ optimal?
    Yes!
Starting simple: optimality?

- Next example you know: scan!
  - For a vector \([x_1, x_2, \ldots, x_n]\) compute vector of \(n\) results: \([x_1; x_1 + x_2; x_1 + x_2 + x_3; \ldots; x_1 + x_2 + x_i \ldots + x_{n-1} + x_n]\)
  - Simple serial schedule

\[
\begin{array}{cccccccc}
  x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\
  \downarrow & \sum & \sum & \sum & \sum & \sum & \sum & \\
  x_1 & x_1 + x_2 & x_1 + \cdots + x_4 & x_1 + \cdots + x_6 & x_1 + \cdots + x_8 \\
  x_1 + \cdots + x_3 & x_1 + \cdots + x_5 & x_1 + \cdots + x_7 & \\
\end{array}
\]

Class question: work and depth?

\[W(n) = n-1, \quad D(n) = n-1\]

Class question: is this optimal?
What did we learn earlier?

- Recursive to get to $W = O(n)$ and $D = O(\log n)$! Assume $n = 2^k$ for simplicity!
  - Sounds “optimal”, doesn’t it? Well, let’s look at the constants!
- Algorithm

Class question: work?
(hint: after the way up, all powers of two are done, all others require another operation each)

$$W(n) = 2n - \log_2 n - 1$$

Class question: depth?
(needs to go up and down the tree)

$$D(n) = 2 \log_2 n - 1$$

Class question: what happened to optimality?
Oh no, not good, another algorithm to the rescue!

- Dissemination/recursive doubling – another well-known algorithmic technique – similar to trees

\[ W(n) = n \log_2 n - n + 1 \]

Class question: work?
(hint: number of count number of omitted ops)

Class question: depth?
\[ D(n) = \log_2 n \]

Class question: is this now optimal?
Oh no, three non-optimal algorithms so far!

- **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: \( W + D \geq 2n - 2 \)

Output tree:
- leaves are all inputs, rooted at \( x_n \)
- binary due to binary operation
- \( W = n - 1, D = D_o \)

Input tree:
- rooted at \( x_1 \), leaves are all outputs
- not binary (simultaneous read)
- \( W = n - 1 \)

Ridge can be at most \( D_o \) long!
Now add trees and subtract shared vertices:
\[
(n - 1) + (n - 1) - D_o = 2n - 2 - D_o \leq W
\]
q.e.d.
Work-Depth Tradeoffs and deficiency

“The deficiency of a prefix circuit $c$ is defined as $\text{def}(c) = W_c + D_c - (2n - 2)$”

Latest 2006 result for zero-deficiency construction for $n > F(D + 3) - 1$ ($f(n)$ is inverse)

From Zhu et al.: “Construction of Zero-Deficiency Parallel Prefix Circuits”
Work- and depth-optimal constructions

- **Work-optimal?**
  - Only sequential! Why?
  - $W = n - 1$, thus $D = 2n - 2 - W = n - 1$ q.e.d. ☺

- **Depth-optimal?**
  - Ladner and Fischer propose a construction for work-efficient circuits with minimal depth $D = \lceil \log_2 n \rceil$, $W \leq 4n$
  - *Simple set of recursive construction rules (boring for class, check 1980’s paper if needed)*
    - *Has an unbounded fan-out! May thus not be practical* ☹

- **Depth-optimal with bounded fan-out?**
  - Some constructions exist, interesting open problem
  - Nice research topic to define optimal circuits
But why do we care about this prefix sum so much?

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

- Prefix summation as function composition is extremely powerful!
  - Many seemingly sequential problems can be parallelized!

- Simple first example: binary adder – \( s = a + b \) (n-bit numbers)
  - Starting with single-bit (full) adder for bit \( i \)

Example 4-bit ripple carry adder

Question: what are the functions for \( s_i \) and \( c_{out,i} \)?

\[
\begin{align*}
s_i &= a_i \text{ xor } b_i \text{ xor } c_{in,i} \\
c_{out,i} &= a_i \text{ and } b_i \text{ or } c_{in,i} \text{ and } (a_i \text{ xor } b_i)
\end{align*}
\]

Show example 4-bit addition!
Question: what is work and depth?

\[\text{source: electronics-tutorials.ws}\]
Seems very sequential, can this be parallelized?

- We only want \( s \)!
  - Requires \( c_{in,1}, c_{in,2}, \ldots, c_{in,n} \) though \( \oplus \)
    \[ s_i = a_i \oplus b_i \oplus c_{in,i} \]

- Carry bits can be computed with a scan!
  - Model carry bit as state starting with 0
    
    *Encode state as 1-hot vector:* 
    \[ q_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad q_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \]
  
  - Each full adder updates the carry bit state according to \( a_i \) and \( b_i \)
    
    *State update is now represented by matrix operator, depending on \( a_i \) and \( b_i \) \((M_{a_i b_i})\):*
    \[
    M_{00} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \quad M_{10} = M_{01} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad M_{11} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}
    \]
  
  - Operator composition is defined on algebraic ring \( \{0, 1, \text{or}, \text{and}\} \) – i.e., replace “+” with “and” and “*” with “or”
    
    *Prefix sum on the states computes now all carry bits in parallel!*

- Example: \( a=011, b=101 \rightarrow M_{11}, M_{10}, M_{01} \)
  
  - Scan computes: \( M_{11} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} \); \( M_{11}M_{10} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} \); \( M_{11}M_{10}M_{01} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} \) in parallel
  
  - All carry states and \( s_i \) can now be computed in parallel by multiplying scan result with \( q_0 \).
Prefix sums as magic bullet for other seemingly sequential algorithms

- Any time a sequential chain can be modeled as function composition!
  - Let $f_1, \ldots, f_n$ be an ordered set of functions and $f_0(x) = x$
  - Define ordered function compositions: $f_1(x); f_2(f_1(x)); \ldots; f_n(...f_1(x))$
  - If we can write function composition $g(x) = f_i(f_{i-1}(x)$ as $g = f_i \circ f_{i-1}$ then we can compute $\circ$ with a prefix sum!
    
    *We saw an example with the adder ($M_{ab}$ were our functions)*

- Example: linear recurrence $f_i(x) = a_i f_{i-1}(x) + b_i$ with $f_0(x) = x$
  - Write as matrix form $f_i(x) = \begin{pmatrix} a_i & b_i \\ 0 & 1 \end{pmatrix} f_{i-1}(x)$
  - Function composition is now simple matrix multiplication!
    
    For example: $f_2(x) = \begin{pmatrix} a_2 & b_2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_1 & b_1 \\ 0 & 1 \end{pmatrix} f_0(x) = \begin{pmatrix} a_1 a_2 & a_2 b_1 + b_2 \\ 0 & 1 \end{pmatrix} x$

- Most powerful! Homework:
  - Parallelize tridiagonal solve
  - 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 \( i \) stably sort all values by the \( i \)-th bit
  - Example, \( k=1 \):
    - Iteration 0: 101 111 010 011 110 001
    - Iteration 1: 010 110 101 111 011 001
    - Iteration 2: 101 001 010 110 111 011
    - Iteration 3: 001 010 011 101 110 111

- Now on \( n \) processors
  - Each processor owns single k-bit number, each iteration
    - \( \text{low} = \text{prefix\_scan}(!\text{bit}, \text{sum}) \)
    - \( \text{high} = n-1\text{-backwards\_prefix\_scan}(\text{bit}, \text{sum}) \)
    - \( \text{new\_idx} = (\text{bit} == 0) : \text{low} \ ? \text{high} \)
    - \( b[\text{new\_idx}] = a[i] \)
    - \( \text{swap}(a,b) \)

Show one example iteration!

Question: work and depth?
Oblivious graph algorithms

- Seems paradoxical but isn’t (may just not be most efficient)
  - Use adjacency matrix representation of graph – “compute with all zeros”

```
0 1 1 0 0 0
0 0 0 0 1 1
0 0 0 0 0 1
0 0 1 0 0 0
0 0 0 1 0 0
0 0 0 0 0 1

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

```

Unweighted graph – binary matrix

```
0 2 3 0 0 0
0 0 0 0 3 1
0 0 0 0 0 2
0 0 4 0 0 0
0 0 0 7 0 0
0 0 0 0 8 0

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

```

Weighted graph – general matrix
Algebraic semirings

- 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 \((S, +, *, 0, 1)\)
    - “Standard” ring of rational numbers: \((\mathbb{R}, +, *, 0, 1)\)
    - Boolean semiring: \(\{0,1\}, \lor, \land, 0, 1\)
    - Tropical semiring: \((\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0)\) (also called min-plus semiring)
Oblivious shortest path search

- Construct distance matrix from adjacency matrix by replacing all off-diagonal zeros with $\infty$.
- Initialize distance vector $d_0$ of size $n$ to $\infty$ everywhere but zero at start vertex.
  - E.g., $d_0 = (\infty, 0, \infty, \infty, \infty, \infty)^T$
  - Show evolution when multiplied!

- SSSP can be performed with $n+1$ matrix-vector multiplications!
  - Question: total work and depth?
    \[ W = O(n^3), \ D = O(n \log n) \]
  - Question: Is this good? Optimal?
    \[ Dijkstra = O(|E| + |V| \log |V|) \]

- Homework:
  - Define a similar APSP algorithm with
    \[ W = O(n^3 \log n), \ D = O(\log^2 n) \]
Oblivious connected components

Question: How could we compute the transitive closure of a graph?
- Multiply the matrix \((A + I)\) \(n\) times with itself in the Boolean semiring!
- Why?
  
  Demonstrate that \((A + I)^2\) has 1s for each path of at most length 1
  
  By induction show that \((A + I)^k\) has 1s for each path of at most length \(k\)

What is work and depth of transitive closure?
- Repeated squaring! \(W = O(n^3 \log n)\) \(D = O(\log^2 n)\)

How to get to connected components from a transitive closure matrix?
- Each component needs unique label
- Create label matrix \(L_{ij} = j\) iff \((A_i)^n\) \(i\_j = 1\) and \(L_{ij} = \infty\) otherwise
- For each row (vertex) perform min-reduction to determine its component label!
- Overall work and depth?
  
  \(W = O(n^3 \log n), D = O(\log^2 n)\)
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

- Generalizations 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 (if $|E| < |V|^2 / \log |V|$)
    - May recover some of the lost efficiency by computing zeros!

- Now moving to non-oblivious 😊