Fast Computation by Population Protocols With A Leader

Dana Angluin, James Aspnes, David Eisenstat

presented by: Hugo Platzer

Jan 13 2020

Introduction

- There are a number of different formal models of computation: Turing machines, register machines, lambda calculus etc.
- Population Protocols are another such model of computation
- Lots of agents with limited local state and no information about the global state (e.g., molecules in solution, sensors on vehicles etc.)
- Agents interact randomly without a central authority
- By carefully tuning the way agents interact, they can be made to compute some useful global property

Contents

- Population Protocols (in general)
- Population Protocols (for computation)
- Building blocks of the Population Protocol computer
- Operations of the Population Protocol computer
- Possible optimizations, outlook, applications

Population Protocols

- Set of agents $\{A_1 ... A_n\}$, not ordered (numbering used to facilitate description of model)
- Finite set of states $\{Q_1 ... Q_k\}$: Each agent is in one of these states at a time
- Number of states is a property of the protocol, not the input size $\Rightarrow$ The number of states **must not depend on n**
- Total agent state: Multiset of elements of Q
- Transition function $(a, b) \mapsto (a', b')$, takes an **ordered** pair of states (can be thought of as initiator and responder) and gives new states for both agents

Population Protocols

- Interaction: Pick two **distinct** agents (A_i, A_j) of Q, apply the transition function to update their states
- Execution: infinite sequence of agent pairs (A_i, A_j), specifying which two agents transition in this interaction
- Fairness: originally an adversary that guarantees: if some agent configuration occurs infinitely often, then any configuration reachable also occurs infinitely often during the execution
- but in this paper: focus on random uniform pick of pairs (i, j)
- Convergence: After a certain number of execution steps, all agents will remain in one of the final states forever
- Initialization of states: can be uniform (if doing leader election), or based on input (when computing predicates)

Population Protocols by example: Leader election

- Two states: 1 (leader), 0 (follower)
- All agents start in state 1
- Transition: $(1, 1) \mapsto (1, 0)$
- Example (red: initiator, blue: responder):
  $[1, 1, 1, 1, 1]$
  $[1, 1, 0, 1, 1]$
  $[1, 1, 0, 1, 1]$
  $[1, 1, 0, 0, 1]$
  $[1, 1, 0, 0, 1]$
  $[1, 0, 0, 0, 1]$
  $[1, 0, 0, 0, 0]$
- This protocol takes an expected n^2 interactions to converge

Computation with Population Protocols

- Agents $A_1..A_n$
- Integer registers $R_1..R_m$, each agent stores one bit of each register **in unary**
- Value of register R_k: $\sum_{i=1}^{n} A_i[k]$ (remember agents are not ordered)
- Therefore, for a population size of n, each register can store a number from 0 to n.
- State of agent: One bit for each register, plus additional information about the current instruction being executed, remember the number of states is **not dependent on n**
- Designated **leader agent**: Tells other agents which instruction to execute, when to move from one instruction to the next
- Program: List of instructions that operate on registers (addition, comparison, zero test) plus control flow instructions (conditions, loops)

Building block: Epidemics

- Simplest building block of Population Protocol algorithms
- Used to spread a small piece of information (register bit, current instruction)
- Leader starts epidemics to tell all agents to execute next instruction
- States: 0 (susceptible), 1 (infected)
- Initialization: all agents start in 0 state, except for leader
- Transition: $(1, 0) \mapsto (1, 1)$
- Convergence (all agents infected) w.h.p. guaranteed in $\mathcal{O}(n \log n)$ interactions

Building block: Phase clock

- Any instruction needs a certain number of interactions to complete w.h.p. (typically $\Theta(n \log n)$)
- Leader needs to broadcast signal to start next instruction at the right time
- Problem: leader has no knowledge of other interactions, finite state
- Solution: use duration of an epidemic to get a sense of time
- reduce variance by giving the epidemic m different stages, tunable parameter, larger m means longer clock cycle (m too big does not hurt)
- States $0 \ldots m-1$, leader starts in state 0, all others in state $m-1$

Building block: Phase clock

- Transition:

$(a, b) \mapsto (a, b+1 \mod m)$	responder is leader, $a = b$
$(a, b) \mapsto (a, b)$	responder is leader, $a \neq b$
$(a, b) \mapsto (a, a)$	responder is not leader, $a \in [b+1..b+\frac{m}{2}] \mod m$
$(a, b) \mapsto (a, b)$	responder is not leader, $a \notin [b+1..b+\frac{m}{2}] \mod m$

- phase: leader receives its own stage, goes to next stage
- round: leader returns to stage 0 (m phases)
- For any d_1 and c, there is a parameter m and a constant d_2 so that the phase clock completes n^c rounds each taking between $d_1 \ln n$ and $d_2 \ln n$ interactions with probability at least $1 - n^{-c}$.

Building block: Duplication

- used to add two registers A, B
- States: $(0,0),(0,1),(1,1)$ (two register bits)
- Register state $(1,0)$ is converted to $(0,1)$ beforehand
- Transition: $$\begin{array}{l}((1,1),(0,0)) \mapsto ((0,1),(0,1)) \\ ((0,0),(1,1)) \mapsto ((0,1),(0,1))\end{array}$$
- 1s from first register are moved to second register
- invariant: preserves $A+B$ after every step
- if $A+B \leq n$, eventually all 1s from A will have been moved to B
- Convergence w.h.p. can take $\Theta(n^2)$ interactions
- Convergence w.h.p. in $\mathcal{O}(n \log n)$ interactions guaranteed if $2A+B \leq \frac{n}{2}$
- Test for success: $A=0$?

Building block: Cancellation

- used to compare two registers A, B
- States: $(0,0),(0,1),(1,0)$ (two register bits)
- Register state $(1,1)$ is converted to $(0,0)$ beforehand
- Transition: $$((1,0),(0,1)) \mapsto ((0,0),(0,0))$$ $$((0,1),(1,0)) \mapsto ((0,0),(0,0))$$
- invariant: preserves $A - B$ after every step
- if $A > B$, eventually A will have $A - B$ 1s, B will have all 0s
- if $B > A$, eventually B will have $B - A$ 1s, A will have all 0s
- if $A = B$, eventually $A = B = 0$
- Convergence w.h.p. can take $\Theta(n^2)$ interactions
- After $\mathcal{O}(n \log n)$ interactions, w.h.p. the number of $(0,1)$ states is at most $\frac{n}{8}$, same for the number of $(1,0)$ states
- Test for success: $A = 0 \vee B = 0$?

Building block: Probing

- Test whether there is any agent that satisfies some predicate (typically: is some register bit 1?)
- States: $0, 1, 2$ (in addition to other information at agent)
- Initialization: leader in state 1 (if not satisfied), 2 (if satisfied) , all other agents in state 0
- Transition:
  $(x, y) \mapsto (x, max(x, y))$ responder not satisfied
  $(0, y) \mapsto (0, y)$ responder satisfied
  $(1, y) \mapsto (1, 2)$ responder satisfied
  $(2, y) \mapsto (2, 2)$ responder satisfied
- if there is an agent satisfying the predicate, eventually all agents will be (and stay) in state 2
- otherwise, eventually all agents will be (and stay) in state 1
- Leader checks its state to get result
- Convergence w.h.p. in $\mathcal{O}(n \log n)$ interactions

Microcode instructions

-

Instruction	Effect on state of agent i
NOOP	No effect.
SET(A)	Set $A_i = 1$.
COPY(A,B)	Copy A_i to B_i
DUP(A,B)	Run duplication protocol on state (A_i, B_i).
CANCEL(A,B)	Run cancellation protocol on state (A_i, B_i).
PROBE(A)	Run probe protocol with predicate $A_i = 1$.

- run all operations for $\Theta(n \log n)$ interactions, the constant needs to be tuned (large enough) of course

High-level operations

-

Operation	Effect	Implementation	Notes
Constant 0	$A \leftarrow 0$	SET$(\neg A)$	
Constant 1	$A \leftarrow 1$	SET$(\neg A)$ $A_{\text{leader}} \leftarrow 1$	
Assignment	$A \leftarrow B$	COPY(B, A)	
Addition	$A \leftarrow A + B$	COPY(B, X) DUP(X, A) PROBE(X)	May fail with $X \neq 0$ if $A + B > n/2$.
Multiplication	$A \leftarrow kB$	Use repeated addition.	$k = O(1)$
Zero test	$A \neq 0?$	PROBE(A)	

- These basic operations take a constant number of microcode operations, therefore $\mathcal{O}(n \log n)$ interactions

Operation: Comparison

Algorithm 1 Comparison algorithm COMPARE.

$A' \leftarrow A$.
$B' \leftarrow B$.
$C \leftarrow 1$.
$r \leftarrow 0$.
while true **do**
  CANCEL(A', B').
  if $A' = 0$ and $B' = 0$ **then**
    return $A = B$.
  else if $A' = 0$ **then**
    return $A < B$.
  else if $B' = 0$ **then**
    return $A > B$.
  end if
  $r \leftarrow 1 - r$.
  if $r = 0$ **then**
    $C \leftarrow C + C$.
    if addition failed **then**
      return $A = B$.
    end if
  end if
  $A' \leftarrow A' + A'$.
  $B' \leftarrow B' + B'$.
end while

A' - B' preserved

CANCEL successful (eliminated one register)

CANCEL failed, however A < n / 8, B < n / 8

repeat loop for 2 log2 n times
if still no elimination, A = B

A' - B' doubled

-
- Requires $\mathcal{O}(log(n))$ instructions, returns correct result w.h.p.

Operation: Subtraction

- **Algorithm 2** Subtraction algorithm SUBTRACT.

```
A' ← A.
B' ← B.
CANCEL(A', B').
if B' = 0 then                      // If this fails, A < n / 8, B < n / 8
    C ← A.
    return.
end if
C ← 0.                              // build C (difference) using binary search
while A' ≠ B' + C do
    D ← 1.
    while A' ≥ B' + C + D + D do    // find most significant bit still missing in C
        D ← D + D.
    end while
    C ← C + D.                      // introduce this bit to C
end while
```

- Requires $\mathcal{O}(log^3(n))$ instructions, returns correct result w.h.p.

Other operations

- Division
  - Shift divisor to the left as long as its not larger than dividend ($\log^2 n$ instructions)
  - Subtract from dividend ($\log^3 n$ instructions)
  - Repeat for all $log(n)$ bits of dividend
  - Also keep track of quotient (shift from 1 to the left, add to total)
  - $\Theta(\log^4 n)$ instructions
- Extract individual bits
  - Extract bit: Divide by 2 until desired bit is least significant ($\log n$ divisions)
  - Test for even / odd by dividing by 2, multiplying by 2, comparing
  - Set bit: Test bit, if not already correct: Shift 1 to the left to match up with bit, add / subtract to change bit
  - $\Theta(\log^5 n)$ instructions

Outlook

- Optimize subtraction by balanced representation
  - Balanced representation: Each register consists of a positive and a negative part: $A = A^+ - A^-$
  - Addition: add positive to positive, negative to negative
  - Subtraction: add positive to negative, negative to positive
  - Use cancellation to keep parts from growing too big
  - Faster subtraction also means faster division
- Faster simulation of LOGSPACE turing machines
- Faster evaluation of semilinear predicates using random-walk broadcast
- Obtain a single leader in $\mathcal{O}(n \log^k n)$ interactions
- Fault tolerance, non-uniform distribution of interactions