Population Protocols

James Aspnes
Yale University

June 10th, 2009
Joint work with:

- Dana Angluin (Yale)
- Melody Chan (Princeton)
- Zoë Diamadi (McKinsey & Company)
- David Eisenstat (Brown)
- Michael J. Fischer (Yale)
- Hong Jiang (Google)
- René Peralta (NIST)
- Eric Ruppert (York)
The past and future of computing

Economics of mass production push computer systems toward large numbers of very limited standardized components:

- Centralized systems
- Distributed systems
- Wireless distributed systems
- Sensor networks/RFID chips
- Smart molecules?

Our goal: take the limit of this process.

An interaction between two neighbors updates the state of both agents according to a joint transition function.

Interactions are asymmetric: one agent is the initiator and one the responder.

Leader Election

\[ \bullet, \bullet \rightarrow \bullet, \bullet \]
\[ \bullet, \bullet \rightarrow \bullet, \bullet \]
\[ \bullet, \bullet \rightarrow \bullet, \bullet \]

\[ \bullet, \bullet \rightarrow \bullet, \bullet \]

An interaction between two neighbors updates the state of both agents according to a joint transition function.

Interactions are asymmetric: one agent is the initiator and one the responder.

An interaction between two neighbors updates the state of both agents according to a joint transition function.

Interactions are asymmetric: one agent is the initiator and one the responder.

An interaction between two neighbors updates the state of both agents according to a joint transition function.

Interactions are asymmetric: one agent is the initiator and one the responder.

An interaction between two neighbors updates the state of both agents according to a joint transition function.

Interactions are asymmetric: one agent is the initiator and one the responder.

An interaction between two neighbors updates the state of both agents according to a joint transition function.

Interactions are asymmetric: one agent is the initiator and one the responder.

An interaction between two neighbors updates the state of both agents according to a joint transition function.

Interactions are asymmetric: one agent is the initiator and one the responder.
A **population protocol** (Angluin, Aspnes, Diamadi, Fischer, and Peralta, PODC 2004) consists of a collection of **finite-state agents** organized in an **interaction graph**.

An **interaction** between two neighbors updates the state of both agents according to a joint **transition function**.

Interactions are **asymmetric**: one agent is the **initiator** and one the **responder**.

**Leader Election**

- $\bullet = leader$
- $\bullet = non-leader$

- $\bullet, \bullet \rightarrow \bullet, \bullet$
- $\bullet, \bullet \rightarrow \bullet, \bullet$
- $\bullet, \bullet \rightarrow \bullet, \bullet$

An interaction between two neighbors updates the state of both agents according to a joint transition function.

Interactions are asymmetric: one agent is the initiator and one the responder.

An interaction between two neighbors updates the state of both agents according to a joint transition function.

Interactions are asymmetric: one agent is the initiator and one the responder.
A **population protocol** (Angluin, Aspnes, Diamadi, Fischer, and Peralta, PODC 2004) consists of a collection of **finite-state agents** organized in an **interaction graph**.

An **interaction** between two neighbors updates the state of both agents according to a joint **transition function**.

Interactions are **asymmetric**: one agent is the **initiator** and one the **responder**.
Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

### Parity

**In:**
- \(0^*, 0^* \rightarrow 0, 0^*\)
- \(0^*, 1^* \rightarrow 1, 1^*\)
- \(1^*, 0^* \rightarrow 1, 1^*\)

**Out:**
- \(1^*, 1^* \rightarrow 0, 0^*\)
- \(x \rightarrow x\)
- \(x^* \rightarrow x\)
- \(x, y^* \rightarrow y^*, y\)
- \(x^*, y \rightarrow x, x^*\)
Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

### Parity

**In:**
- $0^*, 0^* \rightarrow 0, 0^*$
- $x \rightarrow x^*$
- $1^*, 0^* \rightarrow 1, 1^*$

**Out:**
- $1^*, 1^* \rightarrow 0, 0^*$
- $x \rightarrow x$
- $x, y^* \rightarrow y^*, y$
- $x^*, y \rightarrow x, x^*$

June 10th, 2009
DNA15: Population Protocols
Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

### Parity

- **Input**
  - $0^*, 0^* \rightarrow 0, 0^*$
  - $0^*, 1^* \rightarrow 1, 1^*$
  - $1^*, 0^* \rightarrow 1, 1^*$

- **Output**
  - $1^*, 1^* \rightarrow 0, 0^*$
  - $x \rightarrow x$
  - $x^* \rightarrow x$
  - $x^*, y \rightarrow x, x^*$
  - $x, y^* \rightarrow y^*, y$
Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

**Parity**

<table>
<thead>
<tr>
<th>In:</th>
<th>0*, 0* → 0, 0*</th>
</tr>
</thead>
<tbody>
<tr>
<td>x → x*</td>
<td>0*, 1* → 1, 1*</td>
</tr>
<tr>
<td></td>
<td>1*, 0* → 1, 1*</td>
</tr>
<tr>
<td>Out:</td>
<td>1*, 1* → 0, 0*</td>
</tr>
<tr>
<td>x → x</td>
<td>1*, y* → y*, y</td>
</tr>
<tr>
<td>x* → x</td>
<td>x*, y → x, x*</td>
</tr>
</tbody>
</table>

\[ \begin{aligned}
1 & \rightarrow 1* \\
1* & \rightarrow 0* \\
1* & \rightarrow 0* \\
0* & \rightarrow 0* \\
\end{aligned} \]
Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

### Parity

**In:**
- \(0^*, 0^* \rightarrow 0, 0^*\)
- \(0^*, 1^* \rightarrow 1, 1^*\)
- \(1^*, 0^* \rightarrow 1, 1^*\)

**Out:**
- \(1^*, 1^* \rightarrow 0, 0^*\)
- \(x, y^* \rightarrow y^*, y\)
- \(x^*, y \rightarrow x, x^*\)

![Parity Diagram](image-url)
Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

### Parity

**In:**
- $0^*, 0^* \rightarrow 0, 0^*$
- $x \rightarrow x^*$
  - $0^*, 1^* \rightarrow 1, 1^*$
  - $1^*, 0^* \rightarrow 1, 1^*$
**Out:**
- $1^*, 1^* \rightarrow 0, 0^*$
- $x \rightarrow x$
  - $x, y^* \rightarrow y^*, y$
- $x^* \rightarrow x$
  - $x^*, y \rightarrow x, x^*$
Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

### Parity

- **In:**
  - $0^*, 0^* \rightarrow 0, 0^*$
  - $x \rightarrow x^*$
    - $0^*, 1^* \rightarrow 1, 1^*$
    - $1^*, 0^* \rightarrow 1, 1^*$

- **Out:**
  - $1^*, 1^* \rightarrow 0, 0^*$
  - $x \rightarrow x$
    - $x, y^* \rightarrow y^*, y$
  - $x^* \rightarrow x$
    - $x^*, y \rightarrow x, x^*$

![Parity Diagram]
Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

### Parity

**Input map**

- \(0^*, 0^* \rightarrow 0, 0^*\)
- \(0^*, 1^* \rightarrow 1, 1^*\)
- \(1^*, 0^* \rightarrow 1, 1^*\)

**Output map**

- \(1^*, 1^* \rightarrow 0, 0^*\)
- \(x, y^* \rightarrow y^*, y\)
- \(x^*, y \rightarrow x, x^*\)
Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

**Parity**

<table>
<thead>
<tr>
<th>In:</th>
<th>0*, 0* → 0, 0*</th>
</tr>
</thead>
<tbody>
<tr>
<td>x → x*</td>
<td>0*, 1* → 1, 1*</td>
</tr>
<tr>
<td>1*, 0* → 1, 1*</td>
<td>Out:</td>
</tr>
<tr>
<td>x → x</td>
<td>1*, 1* → 0, 0*</td>
</tr>
<tr>
<td>x*, y* → y*, y</td>
<td></td>
</tr>
<tr>
<td>x*, y → x, x*</td>
<td></td>
</tr>
</tbody>
</table>

![Parity Diagram]

June 10th, 2009
Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

### Parity

**In:**
- $0^*, 0^* \rightarrow 0, 0^*$
- $x \rightarrow x^*$
  - $0^*, 1^* \rightarrow 1, 1^*$
  - $1^*, 0^* \rightarrow 1, 1^*$

**Out:**
- $x \rightarrow x$
  - $1^*, 1^* \rightarrow 0, 0^*$
- $x^* \rightarrow x$
  - $x, y^* \rightarrow y^*, y$
- $x^*, y \rightarrow x, x^*$
Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

**Parity**

**In:**
- $0^*, 0^* \rightarrow 0, 0^*$
- $x \rightarrow x^*$
- $0^*, 1^* \rightarrow 1, 1^*$
- $1^*, 0^* \rightarrow 1, 1^*$

**Out:**
- $1^*, 1^* \rightarrow 0, 0^*$
- $x \rightarrow x$
- $x, y^* \rightarrow y^*, y$
- $x^* \rightarrow x$
- $x^*, y \rightarrow x, x^*$
Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

**Parity**

<table>
<thead>
<tr>
<th>In:</th>
<th>0*, 0* → 0, 0*</th>
</tr>
</thead>
<tbody>
<tr>
<td>x → x*</td>
<td>0*, 1* → 1, 1*</td>
</tr>
<tr>
<td></td>
<td>1*, 0* → 1, 1*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Out:</th>
<th>1*, 1* → 0, 0*</th>
</tr>
</thead>
<tbody>
<tr>
<td>x → x</td>
<td>x, y* → y*, y</td>
</tr>
<tr>
<td>x* → x</td>
<td>x*, y → x, x*</td>
</tr>
</tbody>
</table>

June 10th, 2009

DNA15: Population Protocols
Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

- **In:**
  - \(0^*, 0^* \rightarrow 0, 0^*\)
  - \(x, 1^* \rightarrow 1, 1^*\)
  - \(1^*, 0^* \rightarrow 1, 1^*\)

- **Out:**
  - \(1^*, 1^* \rightarrow 0, 0^*\)
  - \(x \rightarrow x\)
  - \(x^* \rightarrow x\)
  - \(x^*, y \rightarrow x, x^*\)

\[\text{Parity diagram:} \quad 1 \rightarrow 0 \rightarrow 0^* \rightarrow 1 \rightarrow 0 \rightarrow 0^*\]
Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

### Parity

- **In:**
  - $0^*, 0^* \rightarrow 0, 0^*$
  - $x \rightarrow x^*$
    - $0^*, 1^* \rightarrow 1, 1^*$
    - $1^*, 0^* \rightarrow 1, 1^*$
- **Out:**
  - $1^*, 1^* \rightarrow 0, 0^*$
  - $x \rightarrow x$
    - $x, y^* \rightarrow y^*, y$
  - $x^* \rightarrow x$
    - $x^*, y \rightarrow x, x^*$

![Parity Diagram](attachment:parity-diagram.png)
Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

<table>
<thead>
<tr>
<th>Parity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In:</strong></td>
</tr>
<tr>
<td>0*, 0* → 0, 0*</td>
</tr>
<tr>
<td>x → x* 0*, 1* → 1, 1*</td>
</tr>
<tr>
<td>1*, 0* → 1, 1*</td>
</tr>
<tr>
<td><strong>Out:</strong></td>
</tr>
<tr>
<td>1*, 1* → 0, 0*</td>
</tr>
<tr>
<td>x, y* → y*, y</td>
</tr>
<tr>
<td>x*, y → x, x*</td>
</tr>
</tbody>
</table>

[Diagram of Parity]
What we can compute

- Trick: represent numbers by tokens scattered across the population.
- Population protocols on connected graphs can **stably compute** all of **first-order Presburger arithmetic** on counts of input tokens, including
  - Addition.
  - Subtraction.
  - Multiplication by a constant $k$.
  - Remainder mod $k$.
  - $>, <$, and $=.$
  - $\land$, $\lor$, $\neg$, $\forall x$, and $\exists x$, applied to above.
- Example: “Are there at least twice as many 0 bits as 1 bits?”
Presburger predicates in disguise

Other ways to define a Presburger predicate:

- Take a regular language $L$ and forget about the order of symbols in each word.
  - Resulting Parikh map of a regular set is Presburger-definable.
  - All Presburger-definable sets can be constructed this way.
  - Cute fact: going to context-free languages doesn’t change anything.

- Take a finite union of linear sets of the form
  \[
  \{ \vec{b} + k_1 \vec{x}_1 + k_2 \vec{x}_2 + \cdots + k_m \vec{x}_m \}.
  \]

  - Resulting semilinear set is Presburger-definable.
  - All Presburger-definable sets can be constructed this way.
Example

A semilinear set $S$, equal to the union of
$\{(1,0) + k \cdot (1,0) + k \cdot (2,1)\}$ (dark circles), and
$\{(0,2) + k \cdot (2,0)\}$ (shaded circles).

Formula:
$((x \mod 2 = 1) \land (2y + 1 \geq x)) \lor ((x = 2) \land (y \mod 2 = 0))$. 

June 10th, 2009  DNA15: Population Protocols
Computability of Presburger predicates

- Computable for fixed inputs (Angluin et al., PODC 2004)
- Computable if inputs converge after some finite time (Angluin, Aspnes, Chan, Fischer, Jiang, and Peralta, DCOSS 2005).
- Computable with one-way communication (Angluin, Aspnes, Eisenstat, Ruppert, OPODIS 2005).
- Computable if a small number of agents fail (Delporte-Gallet, Fauconnier, Guerraoui, Ruppert, DCOSS 2006).
- Nothing else is computable on a complete interaction graph, i.e. if any agent can interact with any other (Angluin, Aspnes, Eisenstat, PODC 2006).
  - Example: can’t compute “Is the number of 0 bits the square of the number of 1 bits?”
Hooray! We’re done!

- Question: If we have an exact characterization of what population protocols can do, aren’t we done?
- Answer: No.
Question: If we have an exact characterization of what population protocols can do, aren’t we done?

Answer: No.

- Bounded-degree interaction graph gives all of LINSPACE (Angluin et al., DCOSS 2005).
- Random scheduling in a complete graph gives all of LOGSPACE with exponential slowdown using simple techniques (Angluin et al., PODC 2004), or polylogarithmic slowdown using more sophisticated techniques (Angluin et al., DISC 2006).

Random scheduling + complete graph = test-tube full of molecules.
Randomized population protocols

- Assume next pair of agents to interact is chosen uniformly (i.e. with probability \( \frac{1}{N(N-1)} \)).
- This gives the randomized population protocol model from (Angluin et al., PODC 2004).
- It also is equivalent to the uniform-rate case of the standard model for well-mixed chemical systems (e.g. (Gillespie, 1977)), population processes from the stochastic processes literature ((Kurtz, 1981)), and corresponds closely to the stochastic chemical reaction networks of (Soloveichik et al., 2008).
- Expected time is obtained by dividing expected interactions by \( N \)—each agent interacts at a fixed rate regardless of size of the population.
What does this have to do with DNA computing?

<table>
<thead>
<tr>
<th>Agents</th>
<th>Molecules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent states</td>
<td>Species</td>
</tr>
<tr>
<td>Interactions</td>
<td>Reactions</td>
</tr>
<tr>
<td>Complete interaction graph</td>
<td>Well-mixed test tube</td>
</tr>
<tr>
<td>Uniform interaction rates  ≠ Varying reaction rates</td>
<td></td>
</tr>
<tr>
<td>Conservation of agents     ≠ Synthesis and decomposition</td>
<td></td>
</tr>
</tbody>
</table>

- Disclaimer: I just write transition tables, I don’t know if they can be realized in a lab.

- For more chemically realistic models see (Soloveichik, Cook, Winfree, and Bruck, Computing with finite stochastic chemical reaction networks, Natural Computing 7(4):615–633, December 2008).
A test-tube computer

- **Register values** (up to $O(N)$) are stored as tokens distributed across the population.
- A unique **leader agent** acts as the (finite-state) CPU.
- We want to support the usual operations of addition, subtraction, comparison, multiplication, division, etc.
- We want to do them all in polylogarithmic time ($O(N \log^{O(1)} N)$ interactions).
- We’ll accept a small ($O(N^{-\Theta(1)})$) probability of error.
Epidemics

- Key fact: An epidemic starting from one infected agent spreads to all agents in $\Theta(\log N)$ time with high probability.

- This gives us a broadcast primitive.
Epidemics

- Key fact: An epidemic starting from one infected agent spreads to all agents in $\Theta(\log N)$ time with high probability.

- This gives us a broadcast primitive.
Epidemics

- Key fact: An epidemic starting from one infected agent spreads to all agents in $\Theta(\log N)$ time with high probability.

- This gives us a broadcast primitive.
Key fact: An epidemic starting from one infected agent spreads to all agents in $\Theta(\log N)$ time with high probability.

This gives us a broadcast primitive.
Epidemics

- Key fact: An epidemic starting from one infected agent spreads to all agents in $\Theta(\log N)$ time with high probability.

- This gives us a broadcast primitive.
Epidemics

- Key fact: An epidemic starting from one infected agent spreads to all agents in $\Theta(\log N)$ time with high probability.

- This gives us a broadcast primitive.
Epidemics

Key fact: An epidemic starting from one infected agent spreads to all agents in $\Theta(\log N)$ time with high probability.

This gives us a broadcast primitive.
Instruction cycle

- Leader propagates a new opcode via epidemic.
- Followers carry out chosen operation:
  - $A \leftarrow 0$: Erase your $A$ token upon receipt of opcode.
  - $A \leftarrow A + B$: Make a new $A$ token for each $B$ token.
  - $A \not= 0$: Start a counter-epidemic if you have an $A$.
  - $A > B$, $A \leftarrow A - B$, etc.: more complicated.
- Leader collects response (if any) from counter-epidemic, updates its state, and starts a new cycle.
Instruction cycle

- Leader propagates a new opcode via epidemic.
- Followers carry out chosen operation:
  - \( A \leftarrow 0 \): Erase your \( A \) token upon receipt of opcode.
  - \( A \leftarrow A + B \): Make a new \( A \) token for each \( B \) token.
  - \( A \not?= 0 \): Start a counter-epidemic if you have an \( A \).
  - \( A > B, A \leftarrow A - B \), etc.: more complicated.
- Leader collects response (if any) from counter-epidemic, updates its state, and starts a new cycle.
Problem: How does the leader know when to start the next instruction cycle?
Bounding the time for epidemics

- Average interactions to infect next victim is $\frac{N(N-1)}{i(N-i)}$.
- For $i > N/2$, this is $\Theta(N/i)$, the waiting time for coupon collector.

⇒ Known coupon collector concentration results (Kamath et al., 1995) bound $i > N/2$ case: $\Theta(N \log N)$ w.h.p.

- Symmetry bounds $i > N/2$ case.
Each agent is in a **phase** in the range 0 to \( m - 1 \).

An initiator in a later phase \( \mod m \) recruits agents in earlier phases.

The leader advances if it sees an initiator in its own phase.

Result: Leader goes all the way around every \( \Theta(\log N) \) time units.
Phase clock

- Each agent is in a **phase** in the range 0 to $m - 1$.
- An initiator in a later phase $\mod m$ recruits agents in earlier phases.
- The leader advances if it sees an initiator in its own phase.
- Result: Leader goes all the way around every $\Theta(\log N)$ time units.
Phase clock

- Each agent is in a **phase** in the range 0 to \( m - 1 \).
- An initiator in a later phase mod \( m \) recruits agents in earlier phases.
- The leader advances if it sees an initiator in its own phase.
- Result: Leader goes all the way around every \( \Theta(\log N) \) time units.
Each agent is in a phase in the range 0 to $m - 1$.

An initiator in a later phase mod $m$ recruits agents in earlier phases.

The leader advances if it sees an initiator in its own phase.

Result: Leader goes all the way around every $\Theta(\log N)$ time units.
Each agent is in a **phase** in the range 0 to $m - 1$.

An initiator in a later phase $\mod m$ recruits agents in earlier phases.

The leader advances if it sees an initiator in its own phase.

Result: Leader goes all the way around every $\Theta(\log N)$ time units.
Phase clock: simulation results

Phase clock with $N = 1000$ and $m = 8$. 
Phase clock: simulation results

Zoomed view of phase 0 and phase 4.
Why it works

- Phases $i$ and higher act as an epidemic wiping out phases $i-1$ and lower.
- This epidemic finishes in $a \log N$ time (with high probability).
- When the leader advances, it takes at least $b \log N$ time (w.h.p.) to generate at least $N^\epsilon$ agents in the same phase $\Rightarrow$ leader advances before $b \log N$ time (a short phase) with probability $N^{O(\epsilon)-1}$.
- For a sufficiently large number of phases $m$, the chance of too many short phases in a row is $O(N^{-c})$.
- **Amazing fact:** $m$ depends on $c$ but not $N$. 
Other operations

- Operations like assignment and addition that don’t require tokens to interact can be done in one instruction cycle ($O(\log N)$ time).
- Operations that do require interaction may take longer.
  - Naive $A > B$ algorithm: Have $A$ and $B$ tokens cancel until only one kind is left.
  - This takes $\Omega(N^2)$ interactions if there are few $A$’s and $B$’s.
- How can we do cancellation faster?
Cancellation by amplification

- Cancellation is fast if there are many tokens to cancel.
- Solution: Alternate between canceling and doubling.
- Invariant $A_k - B_k = 2^k(A_0 - B_0)$ after $k$ rounds.
- If no winner in $2 \log N$ rounds, $A_0 = B_0$.
- This gives $A < B$ in $O(\log^2 N)$ time.
To compute $C \leftarrow A - B$, do binary search for $C$ such that $A = B + C$.

This takes $O(\log N)$ rounds of binary search at $O(\log^2 N)$ time each $\Rightarrow O(\log^3 N)$ time.

Similar approach for division gives $O(\log^5 N)$ time. (This is our most expensive operation.)
Results

For a randomized population protocol with a unique initial leader, we have:

- **Register machine simulation:**
  - $\Theta(\log N)$-bit registers.
  - $O(\log^5 N)$ expected time per operation. ($O(\log N)$ in later work.)
  - $O(N^{-c})$ probability of failure.

- **Presburger predicate computation:**
  - $O(\log^5 N)$ expected time. (Cf. $O(N)$ for previous protocols.)
  - **Zero** probability of failure.
  - **Trick:** Combine fast fallible protocol with slow robust one.
Why $O(\log^5 N)$?

- Main problem: Comparisons take too long.
- Solution: See next slide.
(Angluin, Aspnes, and Eisenstat, DISC 2007).

1. Start with mixed population of $x$ and $y$ agents:

2. Run for $O(\log N)$ time.

3. Obtain (with high probability) majority value everywhere:
Confusion creates blank agents

- Three states: $x$, $y$, and $b$ (blank).
- If I see disagreement, I go blank:
- Equally likely to remove an $x$ or a $y$.
- Never removes last non-blank token.
Fashion favors the majority

- Blank agents adopt whatever value they see:

  ![Diagram showing the process]

  - Favors more common value ⇒ pushes towards unanimity.
Simulation results

Approximate majority starting with $x = y = 500$. 

June 10th, 2009 DNA15: Population Protocols
Simulation results (log scale)

Approximate majority starting with $x = y = 500$. 

DNA15: Population Protocols

June 10th, 2009
Transitions push to \( y = N \) and \( x = N \) corners.

Unstable equilibrium at \( y = x = b \).

Want to show \( O(N \log N) \) bound on steps to convergence.
The proof I wish we had

One thing to try:

- Use standard results on limits of population processes \((Kurtz, Wormald)\) to get system of differential equations.
- Solve them to find convergence bounds in the limit as \(N \to \infty\).

But it doesn’t work:

- Known limit results only work up to \(O(N)\) interactions.
- Small \((o(1))\) concentrations of agents go to 0 in the limit.
- Resulting differential equations don’t have nice solutions anyway.
What we did instead

- Use a potential function to bound number of $xb$ and $yb$ interactions.
- Basic idea: track $u = x - y$.
  - When $|u|$ is small, acts like random walk: $u^2$ rises on average.
  - When $|u|$ is big, acts like exponential growth: $\log |u|$ rises on average.
  - Compromise: $f = \log \left( \frac{3}{2} N + u^2 \right)$ acts like $u^2$ for small $|u|$ and $\log |u|$ for large $|u|$.
- Bound $xy$ and $yx$ interactions by conservation of agents.
- This leaves $xx$, $yy$, and $bb$ interactions, but these are rare except in the corners.
- Separate potential functions cover corner cases.
- Final result: $O(N \log N)$ steps with high probability.
Correctness

Majority value is correct if the initial margin is $\omega(\sqrt{N \log N})$

- Couple $(u_i)$ with an unbiased random walk $(t_i)$ so that $|t_i| \leq |u_i|$
  - $\Pr[u \text{ increases}] \geq 1/2$ for $u \geq 0$
  - $\Pr[u \text{ decreases}] \geq 1/2$ for $u \leq 0$

- Suppose $t_0 = u_0 = x_0 - y_0 = \omega(\sqrt{N \log N})$

- With high probability, random walk $t_i$ is positive for $\Theta(N \log N)$ steps $\Rightarrow x$ wins.

- Argue symmetrically for $y$.

This even works if $o(\sqrt{N})$ agents are Byzantine, meaning they can pretend to have any value in any interaction.
Application

- Previous register machine simulation
- + fast comparison operation
- + some other tricks
- = $O(\log N)$-time register machine operations.
- This is optimal.
Can we build it?

- I can’t, but maybe somebody here can.
- Fast robust approximate majority is both simple and fault-resistant.
- Other protocols are more elaborate and more brittle.
Can we analyze it?

- Brute force analysis works, but isn’t pretty.
- Can’t even analyze majority with 3 non-blank token types.
- Better tools are needed.
- But ability to do computation limits what we can do.

More information: