Computing by Waiting and Guessing

Pekka Orponen

T-79.4001 Seminar on Theoretical Computer Science

21.3.2007
Outline

1. Problems and assumptions
2. Minimum-finding by waiting
   - Waiting in rings
   - Waiting in general networks
   - Computing Boolean functions
   - Randomised election
3. Minimum-finding by guessing
   - The general protocol
   - A natural guessing strategy
   - The optimal guessing strategy
   - Removing the constraints
1. Problems and Assumptions

- Basic algorithms presented for unidirectional rings; simple extensions to other topologies.
- Assumptions:
  - Minimum-Finding: \( R + \text{Synch} \)
    \( (R = \{\text{Bidirectional Links, Connectivity, Total Reliability}\}) \)
  - Election: \( R + \text{Synch} + \text{ID} \)


2. Minimum-Finding by Waiting

- Unidirectional ring of size $n$, each entity $x$ has positive integer $\text{id}(x)$ and knows $n$.

- **Min-Find-Wait:**
  - 1. Entity $x$ wakes up and waits for $f(\text{id}(x), n)$ time units.
  - 2. If nothing happens in this time, $x$ determines “I am the smallest” and sends a *Stop* message.
  - 3. If instead $x$ receives a *Stop* message, it determines “I am not smallest” and forwards the message.

- If all entities wake up simultaneously and the *waiting function* $f$ is monotone:

  \[
  \text{id}(x) < \text{id}(y) \Rightarrow f(\text{id}(x), n) < f(\text{id}(y), n),
  \]

  then minimal elements correctly determine their status.

- *However* the minimal elements must also eliminate the non-minimal ones . . .
For the elimination it suffices that

\[ id(x) < id(y) \Rightarrow f(id(x), n) + d(x, y) < f(id(y), n), \]

where \( d(x, y) \leq n - 1 \) is the distance from \( x \) to \( y \).

Thus in a ring one may choose \( f(i, n) = i \cdot n \).

Note: If elements have unique id’s, then protocol also solves leader election.
In case of non-simultaneous wake-up, when entity $x$ wants to start the protocol it first sends its neighbour a $Start$ message and then starts waiting.

To account for the wake-up differences it suffices that

$$id(x) < id(y) \Rightarrow f(id(x), n) + 2d(x, y) < f(id(y), n),$$

i.e. in a ring one may choose $f(i, n) = 2i \cdot n$.

In a bidirectional ring one needs in addition take care that each element forwards its messages in a consistent direction.
Comparison of minimum-finding protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Bits</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>$O(n \log i_{\text{max}})$</td>
<td>$O(2^{i_{\text{max}}} n)$</td>
<td></td>
</tr>
<tr>
<td>SynchStages</td>
<td>$O(n \log n)$</td>
<td>$O(i_{\text{max}} n \log n)$</td>
<td></td>
</tr>
<tr>
<td>Wait</td>
<td>$O(n)$</td>
<td>$O(i_{\text{max}} n)$</td>
<td>$n$ known</td>
</tr>
</tbody>
</table>
Waiting in general networks

The waiting protocol actually works in exactly the same way in all (connected) networks, assuming the entities know (a bound on) the network diameter \( d \).

**Min-Find-Wait:**

1. Entity \( x \) wakes up either spontaneously or by a \( Start \) message from one of its neighbours; it sends/forwards \( Start \) to its neighbours.
2. Entity \( x \) waits for \( f(\text{id}(x)) = 2\text{id}(x)(d + 1) \) time units.
3. If nothing happens in this time, \( x \) determines “I am the smallest” and sends its neighbours a \( Stop \) message.
4. If instead \( x \) receives a \( Stop \) message, it determines “I am not smallest” and forwards the \( Stop \) message.

**Correctness:** Definition of the waiting function \( f(i) \) guarantees that, if \( t(z) \) is the wake-up time of entity \( z \), then

\[
\text{id}(x) < \text{id}(y) \Rightarrow t(x) + f(\text{id}(x)) + d(x, y) < t(y) + f(\text{id}(y)).
\]
Application: computing Boolean functions

- Assume each entity $x$ has a Boolean value $b(x) \in \{0, 1\}$ and the goal is to have everyone know the AND of those values.
- Observe that in this case $\text{AND} = \text{Min}$, and apply the Min-Find-Wait protocol.
- Note that:

$$f(b(x)) = \begin{cases} 
2(d + 1), & \text{if } b(x) = 1, \\
0, & \text{if } b(x) = 0.
\end{cases}$$

- Thus the time complexity of the protocol is $2(d + 1)$ units, and the bit complexity is $\leq 2n$ bits. (Can probably be decreased to just $n$.)
- The OR function can be computed by an analogous protocol.
**Application: randomised election**

- Assume $n$ entities in a unidirectional ring. (Method can be generalised to also other topologies.)
- Entities know $n$ but do not have identities. Because of symmetry, deterministic leader election is impossible. Symmetry can be broken by randomisation.

**Randomised-Election:**

1. The protocol works in rounds.
2. In a round, each entity $x$ chooses a random identity $b(x) \in \{0, 1\}$ with $\Pr(b(x) = 0) = 1/n$, $\Pr(b(x) = 1) = 1 - 1/n$.
3. An entity $x$ with $b(x) = 0$ sends the signal Leader? to its neighbour and waits. Entities $x$ with $b(x) = 1$ just forward any possible Leader? signals.
4. If an entity $x$ with $b(x) = 0$ gets its Leader? signal back after exactly $n$ time units, it will become the leader and sends a Terminate signal to notify the others. Otherwise it sends a Restart signal to initiate a new round.
The bit and time complexity of each round is $O(n)$. How many rounds are needed?

The probability that exactly one entity $x$ chooses $b(x) = 0$ is

$$n \cdot \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n-1} = \left(1 - \frac{1}{n}\right)^{n-1} \approx \frac{1}{e} \approx 0.37.$$ 

Thus the number of rounds is geometrically distributed with parameter $p \approx 1/e$, and so

$$E[\#\text{rounds}] = \frac{1}{p} \approx e \approx 2.78$$

and

$$\Pr(\geq k \text{ rounds needed}) = (1 - p)^{k-1} \approx (0.63)^{k-1}.$$
3. Guessing

- More precisely: distributed interval search.
- Consider again minimum finding in a unidirectional ring with \(n\) entities; all entities know the size of the ring and start simultaneously.

**Decide(p):**

1. Each entity \(x\) compares \(p :: id(x)\).
2. If \(p \geq id(x)\), then \(x\) decides “High” and sends signal \textit{High} to neighbour.
3. If \(p < id(x)\) then \(x\) waits for any possible \textit{High}-signals for \(n\) time units. If one is received, also \(x\) decides “High” and forwards the signal. If no \textit{High}-signal is received, \(x\) decides “Low”.

Denote \(i_{\min} = \min\{id(x)\}\). After one round of protocol \textbf{Decide(p)}, all entities know whether \(p \geq i_{\min}\) (“High”) or \(p < i_{\min}\) (“Low”).

- The time complexity of one round is \(n\) units. The bit complexity of deciding “High” is \(n\), and the bit complexity of deciding “Low” is 0.
The common goal of the entities (“players”) is to determine the value $i_{\text{min}}$. They start at some guess $p = p_1$, and based on whether this was “High” or “Low” choose another guess $p = p_2$ etc. until $i_{\text{min}}$ can be determined.

What is the optimal sequence of guesses $p_1, p_2, \ldots$? Note that each guess costs $n$ time units, but only high guesses incur a bit cost.

Thus there is a tradeoff between time and bit cost. E.g. a simple linear search has expected time cost $O(n^2)$ and bit cost $n$; a binary search, assuming $i_{\text{min}} \leq n$, has expected time and bit cost both $O(n \log n)$. 
Assume that $i_{\text{min}} \in [1, M]$. Denote $q$ total number of guesses, $k \leq q$ number of high guesses.

Then a guessing strategy with given $q$, $k$ costs $qn$ time and $kn$ bits.

E.g. for linear search: $k = 1$, $q = M$ in the worst case.

What is the nature of the $k$ vs. $q$ tradeoff? E.g. how much does allowing $k = 2$ decrease $q$?
A natural guessing strategy

For $k = 2$:

1. Partition the interval $[1, M]$ into $\lceil \sqrt{M} \rceil$ subintervals of length $\lceil \sqrt{M} \rceil$. (The last subinterval may be shorter than the others.)
2. Query first the endpoints of the subintervals, $p_1 = \lceil \sqrt{M} \rceil - 1$, $p_2 = 2\lceil \sqrt{M} \rceil - 1$, ... until one of the guesses is high or the last subinterval is reached.
3. Then search the relevant subinterval linearly.

This strategy clearly has $k = 2$, $q = 2\lceil \sqrt{M} \rceil$. Thus, a linear increase in bit cost allows a superlinear decrease in time cost.

The strategy can easily be generalised in a hierarchical way to arbitrary $k$, yielding $q = kM^{1/k}$.

Can we do better? If we want to keep the bit cost linear, then we must have $k = \text{constant}$. What is the optimal way to allocate a given constant number of high guesses?
The optimal guessing strategy

To find the optimal strategy, consider the quantity

\[ h(q, k) = \text{largest } M \text{ such that interval } [1, M] \text{ can be covered by } q \text{ queries, out of which at most } k \leq q \text{ are high.} \]

Then for \( k = 1 \) we have:

\[ h(q, 1) = q, \]

because linear search is the only safe strategy in this case.

At the other extreme, binary search yields:

\[ h(q, q) = 2^q - 1. \]
Consider an optimal strategy with $q$ queries out of which $k$ may be high.

Let $p$ be the first guess of the strategy. Now $p$ may be either low or high as compared to the number being sought.

If $p$ is low, then we have $q - 1$ queries left, including all our $k$ high queries. Thus, for any initial low guess $p$, an interval of length $p + h(q - 1, k)$ can be covered, and it seems ideal to make the first guess as large as possible.

However, if the first guess $p$ is high, then we only have $k - 1$ high queries left, with which we must be able to cover all of the interval $[1, p]$. Thus the largest safe first guess is $p = h(q - 1, k - 1)$, and we get the recurrence equation:

$$h(q, k) = h(q - 1, k - 1) + h(q - 1, k).$$
The recurrence equation with boundary conditions:

\[
\begin{align*}
    h(q, k) &= h(q - 1, k - 1) + h(q - 1, k), \quad 1 < k < q, \\
    h(q, 1) &= 1, \quad h(q, q) = 2^q - 1,
\end{align*}
\]

has solution:

\[
h(q, k) = \sum_{j=1}^{k} \binom{q}{j}.
\]

The optimal guessing strategy for searching interval \([1, M]\) with at most \(k\) high guesses is thus:

1. Query \(p = h(q - 1, k - 1)\), where \(q \geq k\) is smallest integer such that \(M \leq h(q, k)\).
2. If \(p\) is low, then optimally search interval \([p + 1, M]\) with at most \(k\) high guesses.
3. If \(p\) is high, then optimally search interval \([1, p]\) with at most \(k - 1\) high guesses.

\[1\]There’s something wrong here: the recurrence should have an additional “+1” on the r.h.s. for this to hold.
Removing the constraints

- **Bounded interval**: Use an initial sequence of monotonically increasing guesses $g(1) < g(2) < \ldots$ until one of them, say $g(t)$, is high. Then search interval $[g(t - 1) + 1, g(t)]$ using the optimal strategy. If e.g. $g(j) = 2^j$, and one denotes

$$r(M, k) = \min\{ q \mid h(q, k) \geq M \},$$

then

$$r(\ast, k) \leq \lceil \log_2 i_{\min} \rceil + r(i_{\min}, k - 1).$$
> **Knowledge of n:** The entities may use a common upper bound \( \bar{n} \geq n \).

> **Network topology:** Assume the entities have a common upper bound \( \bar{d} \) on the network diameter \( d \). Transform the protocol into a reset with signal \textit{High}, initiated by entities with \( id(x) \leq p \). Use \( \bar{d} \) as the timeout value.

> **Simultaneous start:** Perform a wakeup before running the protocol and use a longer delay between successive guesses.

---

\(^2\)There’s also a method, discussed in Santoro’s book Section 6.3.3., for combining the Waiting and Guessing methods to remove the dependence on the network size/diameter altogether.
Comparison of minimum-finding protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Bits</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>$O(n \log i_{\text{max}})$</td>
<td>$O(2^{i_{\text{max}}} n)$</td>
<td></td>
</tr>
<tr>
<td>SynchStages</td>
<td>$O(n \log n)$</td>
<td>$O(i_{\text{max}} n \log n)$</td>
<td></td>
</tr>
<tr>
<td>Wait</td>
<td>$O(n)$</td>
<td>$O(i_{\text{max}} n)$</td>
<td>$n$ known</td>
</tr>
<tr>
<td>Guess</td>
<td>$O(kn)$</td>
<td>$O(i_{\text{max}}^{1/k} kn)$</td>
<td>$n$ known</td>
</tr>
</tbody>
</table>