Courtesy of Dean Christakos and Deva Seetharam. Used with permission.

Computing with Infinitely Many Processes under assumptions on concurrency and participation - M. Merritt & G. Taubenfeld

Dean Christakos & Deva Seetharam

November 25, 2003

Abstract
This paper explores four classic problems in concurrent computing (election, mutual exclusion, consensus, and naming) when the number of processes which may participate is infinite.

Motivation

Scalability.

While in practice the number of processes will always be finite, algorithms designed for an infinite number of processes may scale well: their time complexity may depend on the actual contention and not on the total number of processes.

M. Merritt says:

"It is really mostly of theoretical interest...to what extent is finite participation a determining factor for distributed computability? Or, put differently, *it was fun to do!* "

Terminology

- Concurrency: The maximum number of processes that may be active simultaneously.

  Levels of concurrency:

  1. Finite - A finite bound (denoted by c) on the max number of processes that are simultaneously active over all runs.
  2. Bounded - In each run, there is a finite bound on the max number of processes that are simultaneously active.
  3. Unbounded - In each run, the number of processes that are simultaneously active is finite but can grow without bound.
  4. Infinite - The max number of processes that are simultaneously active may be infinite.

Terminology (continued)

- Required participation - Every process must eventually execute its code.

- Participation not required - For example, in the mutual exclusion problem a process can stay in the reminder region forever and not try to enter its C.
  - $[\ell, u]$-participation - At least ℓ and at most u processes participate.
  - $[1, n]$-participation - Participation is not required.
  - $[n, n]$-participation - Participation is required.

(Concurrency level might be smaller than the participation.)

If an algorithm is correct assuming $[\ell, u]$-participation, then it is also correct assuming $[\ell', u']$-participation, where $\ell \leq \ell' \leq u' \leq u$.

Terminology (continued)

- Adaptive algorithms:- If the time complexity of processes' operations is bounded by a function of the actual concurrency.

- Symmetric algorithms:- If the only way for distinguishing processes is by comparing (unique) process identifiers. Process id's can be written, read, and compared, but there is no way of looking inside any identifier. In particular, identifiers cannot be used to index shared registers.

Atomic Registers

An atomic register is a data object that is read or written by one or more processes according to the following assumption: If several read or write operations are enabled simultaneously in different processes, then these operations are executed in some sequence, one after the other and not concurrently.

Theorem 2.1 *There is no solution to election with finite concurrency* $c > 1$ *and with* $[c, \infty] - participation$ *using finite shared memory (finitely many registers of finite size).*

Atomic Registers Participation Required

Theorem 2.2
For any finite concurrency c, with‘ $[c, \infty] - participation$*, there are non-adaptive asymmetric solutions to election using an infinite number of atomic bits.*

Atomic Registers Participation Required (continued)

For finite concurrency $c = 2$ with $[2, \infty]$-participation, there are adaptive symmetric solutions to election and consensus using one atomic register.

`Process` i`'s program`
Shared:
$(Leader, Marked)$: (Process id, boolean) initially $(0, 0)$
Local:
$local\_leader$: Process id

if $Marked = 0$ **then**
  $(Leader, Marked) := (i, 0)$
  waitfor $(Leader \neq i)$ *or* $(Marked = 1)$
  $local\_leader := Leader$
  $(Leader, Marked) := (local\_leader, 1)$
fi
return$(Leader)$

- Second process to write to the shared register is elected.
- The first process to write spins until the second process writes.
- The concurrency bound guarantees no other process will interfere.

Atomic Registers Participation Required (continued)

For any finite concurrency c with $[c, \infty]$-participation, there are adaptive symmetric solutions to election and consensus using two atomic registers.

`Process` i`'s program.`

Shared:

$(Leader, Marked)$: (Process id, boolean), initially $(0, 0)$
Union: set of at most c process id's, initially $\emptyset$

Local:

$local\_leader$: Process id
local_union1: set of at most c process id's
local_union2: set of at most c process id's, initially $\{i\}$

```
if Marked = 0 then (Leader, Marked) := (i, 0) fi
local_union1 := Union
while (|local_union1| < c) ∧ (Marked = 0) do
      if ¬(local_union2 ⊆ local_union1) then
            Union := local_union1 ∪ local_union2
      fi
      local_union2 := local_union1 ∪ local_union2
      local_union1 := Union
od
local_leader := Leader
(Leader, Marked) := (local_leader, 1)
return(Leader)
```

- The leader will be the c'th process to write to $Leader$.
- No process terminates until it sets the $Marked$ field to 1.
- The concurrency bound guarantees that the $c+1$'st process cannot take a step until the first c processes terminate.

Atomic Registers Participation Required (continued)

All processes that terminate elect exactly the same participant as a leader. Proof

- Only participating processes id's can appear in the shared $Union$ register.
- A process terminates only when after it sees that c processes are already participating. Upon exit, it sets Marked=1.
- Thus, the first c processes to participate will first see that $Marked$ is not set, and no later process can participate since $Marked = 1$.Hence, only the first c processes will write their id's into $Leader$.
- No process sets $Marked$ until the last of these c processes have written to $Leader$. Hence, any processes that terminate will elect the last process to write its id to $Leader$.

Termination

- First c processes cannot read-spin on a set in $Union$ smaller than ever: one will read a set in which (at least) itself is absent.
- Every write to $Union$ is preceded by a read of set containing fewer values. Suppose there are infinitely

many writes. Then let $c' < c$ be the smallest value such that sets of size c' are written infinitely often. It follows that there are infinitely many reads (and hence writes) of sets with fewer than c' elements, a contradiction.

Mutual Exclusion

- Like the leader election problem, we assume a participation of at least c processes with the number of potential concurrent processes approaching infinity.

- Mututal exclusion algorithms with bounded concurrency are just the same as our old mutex problems limited to n processes

- Current mutual exclusion algorithms can be adapted to assume unbounded concurrency

- However, notice that we can get away with using a finite amount of memory for leader election because with infinite participation, the other processes "piggyback" on the results of the first c processes

- For mutual exclusion, we will see that we cannot come up with such a convenient set of algorithms to bound the amount of shared memory required– mutual exclusion is an ongoing process by all participating processes, not something a subset of processes can finish, allowing other participants to take advantage of, later.

Concurrency in Mutual Exclusion Problems

- Concurrency is precisely the issue dealt with in mutual exclusion– how do we take a group of concurrent processes and ensure that only one process at a time is in the critical region?

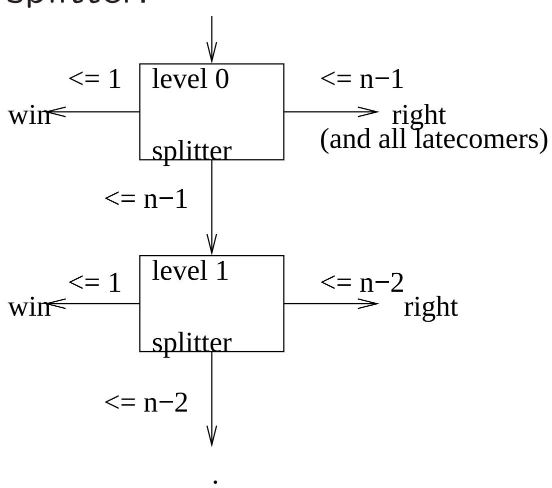
- If there is bounded concurrency, the possibility of a finite number of concurrent processes contending for the critical region reduces to the previously solved mutual exclusion problem with n processes.

- Problem solved for deadlock-free mutual exclusion!

- The trick is creating mutual exclusion algorithms that can handle unbounded concurrency

Algorithms that use an infinite number of registers for mutual exclusion

Here's an example of such a mutual exclusion algorithm– the infinite splitter:



The Splitter for Unbounded Concurrency

```
Process i's program
Shared:
      next: integer, initially 0
      x[0..∞]: array of integers (the initial values are immaterial)
      b[0..∞], y[0..∞], z[0..∞]: array of boolean, initially all 0
Local:
      level: integer, initially 0
      win: boolean, initially 0
start: level := next
repeat
   x[level] := i
   if y[level] then b[level] := 1                          /* move right */
            await level < next
            goto start fi
   y[level] := 1
   if x[level] ≠ i then await (b[level] = 1) or (z[level] = 1)
            if z[level] = 1 then await level < next /* move right */
               goto start
            else level := level + 1 fi                      /* move down */
   else z[level] := 1
      if b[level] = 0 then win := 1                              /* win */
      else level := level + 1 fi fi                         /* move down */
until win = 1
critical section
next := level + 1
```

Explaining the splitter

- This is deadlock-free

- In a nutshell– Processes enter the trying region and contend to enter the critical region. Processes that show up late go "right" and do not enter the trying region again until after there's a final winner. Meanwhile, other contending processes go "down" and contend amongst themselves until all remaining competing processes go "right" and we remain with one winner

- x indicates the last process to enter the splitter

- y is a boolean variable announcing the presence of a process in the trying region

- b is a boolean variable announcing the existence of a latecomer

- z is a boolean variable indicating that a process has finally eliminated other competitors

Mututal Exclusion Algorithms with Unbounded Concurrency

Let's look at the case with Atomic Registers

- Remember for read/write atomic registers, we need n read/write shared variables

- Why? Because in the shared variable case, if we have k shared variables where $1 \leq k \leq n-1$ variables, then it is possible that the state of the k shared variables where a processes i, $1 \leq i \leq n-1$ is about to enter the critical region will look identical to process n as when there is all processes i, $1 \leq i \leq n-1$ are idle and process n will enter the critical region. (Theorem 10.33 in the text)

- This is true for n, and if $n \to \infty$, the number of shared variables required $\to \infty$.

- Let's prove this directly...

Proving the requirement for an infinite number of variables directly

Take an infinite number of processes.

Remember, we assume $[c, \infty]$ participation, with unbounded concurrency, so we have to support up to an infinite number of processes.

Take an infinite number of competing processes. To preserve well-formedness, all processes have the ability to enter the critical region. Assume that the state of the shared variables allows 1 of the processes, i to go into the critical region. If the algorithm is well-formed, and progress is guaranteed, then each process is allowed to enter the critical region based on the state of the shared memory for any reachable system state.

However, because there are only a finite number of states of the shared memory, then there is at least one other process j that believes it is also allowed to enter the critical region given this state of the shared memory. In fact, with our assumption that the k shared variables have x bits, there are x^k possible states of the shared memory, and thus we are unable to allow an infinite number of competing processes to stay out of the Critical region while process i is in the critical region.

If each process writes a distinct value to the k shared variables, the size of these values will require an infinite number of bits to represent the distinct values, so this is a contradiction in our attempt to use only a finite amount of space.

Using Read-Modify-Write Bits

- Providing deadlock-free mutual exclusion with read-modify-write, even in the case of unbounded concurrency, is easy. The *TrivialME* algorithm from Chapter 10 is powerful enough

- Starvation freedom (Lockout Freedom) is another matter

- A quick proof of the requirements for starvation-freedom:

  Assume that there is a finite amount of memory and starvation freedom. Further, assume there are an infinite number of concurrent processes.

  Note: If there is a finite number of values, then we should be able to create a situation in which we repeat that set of values for an infinite amount of time, shutting out a process j indefinitely.