New algorithms for efficient Mutual Exclusion in multi-core and cloud settings

Mohammed Shahid Abdulla
IT and Systems Area
Indian Institute of Management,
Kozhikode, Kerala, INDIA
shahid@iimk.ac.in

Mutual Exclusion problem

- Considered key to Distributed Computing
- Problem of ensuring that only one process or program is present inside a region of code known as a *Critical Section* at any given time.
- Parallel software, including many running on public or private clouds, have critical sections and synchronization issues (Edmondson et. Al.).
- Assumption made in our work: memory is shared by all processors, known as Distributed Shared Memory (DSM) paradigm

State-of-art in Mutual Exclusion

- For n competing processes, Algorithm 1 has a time-complexity (counted as the number of accesses made to remote memory) of min(k,log(n)) for each process.
- Here, k<n is the *point contention*
- The MCS algorithm has a time complexity of O(1), but requires atomic instructions such as `fetch--and--store' (FS) and `compare--and--swap' (CAS).

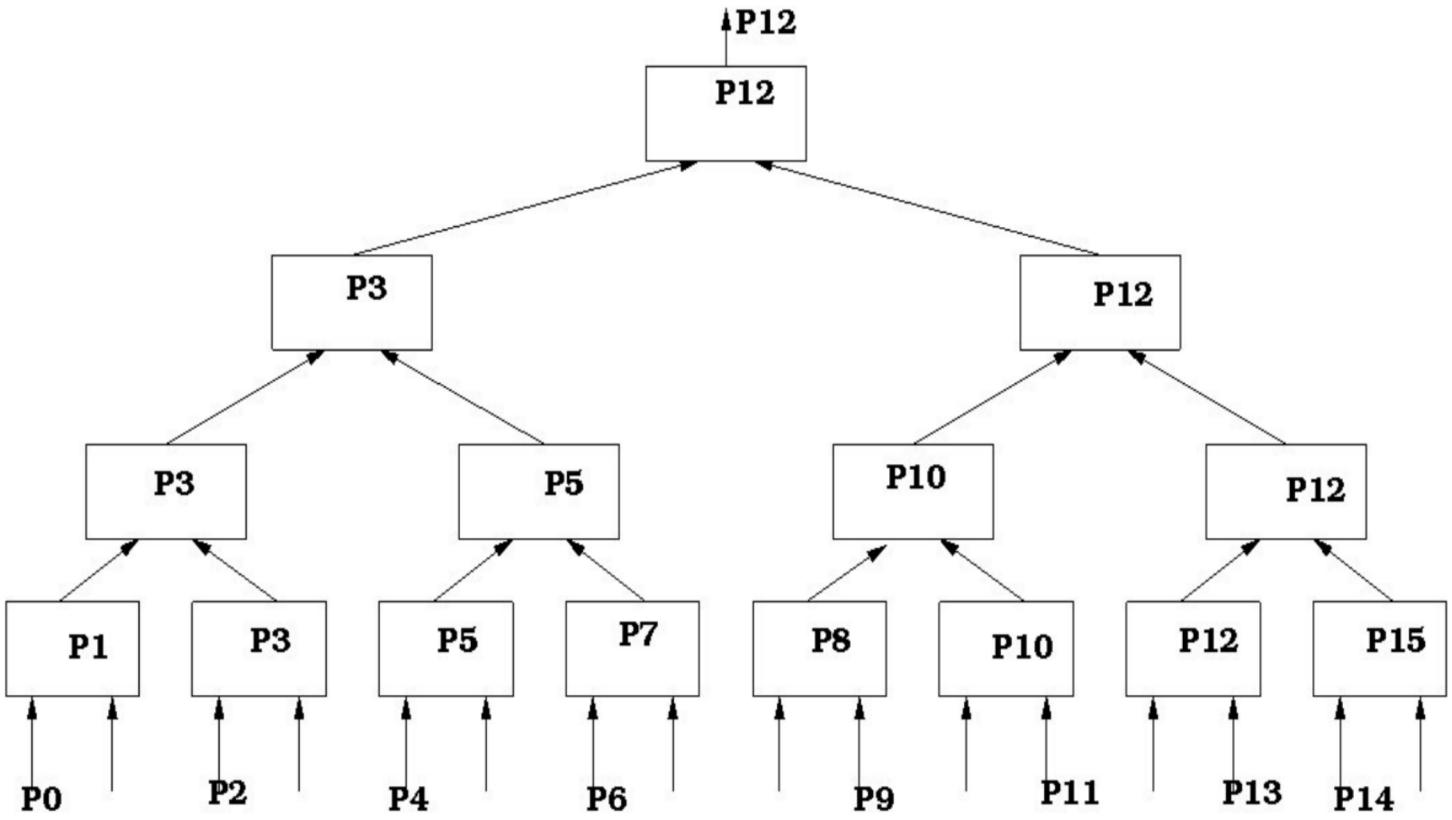
Production-grade locking schemes

- Mutual-Exclusion alone not a replacement for lock managers (eg. ZooKeeper, Chubby) in distributed file systems hosted on cloud.
- Lock managers are designed with a focus on fault-tolerance, consensus etc.
- Yet, recent lock manager FLEASE has O(n) RMR complexity, uses atomic instructions beyond read/write.
- Lustre and PVFS2: locking within file system not supported, so application-implemented locks are key.
- Multi-core architectures like SGI ccNUMA and IBM P5+, have latency not bandwidth as bottleneck: maximum of atomic read/write instructions should be used.

YA tree Vs. MCS

- Yang-Anderson tree is precursor to the efficient O(min(k, logn)) algorithm.
- In YA tree:
  - n global locations are reserved for flags
  - process moves to the apex of the binary tree
  - here it is free to enter its critical section.
  - It exits the lock nodes in order of acquisition
  - Competing processes, at various levels, occupy the lock node just vacated.

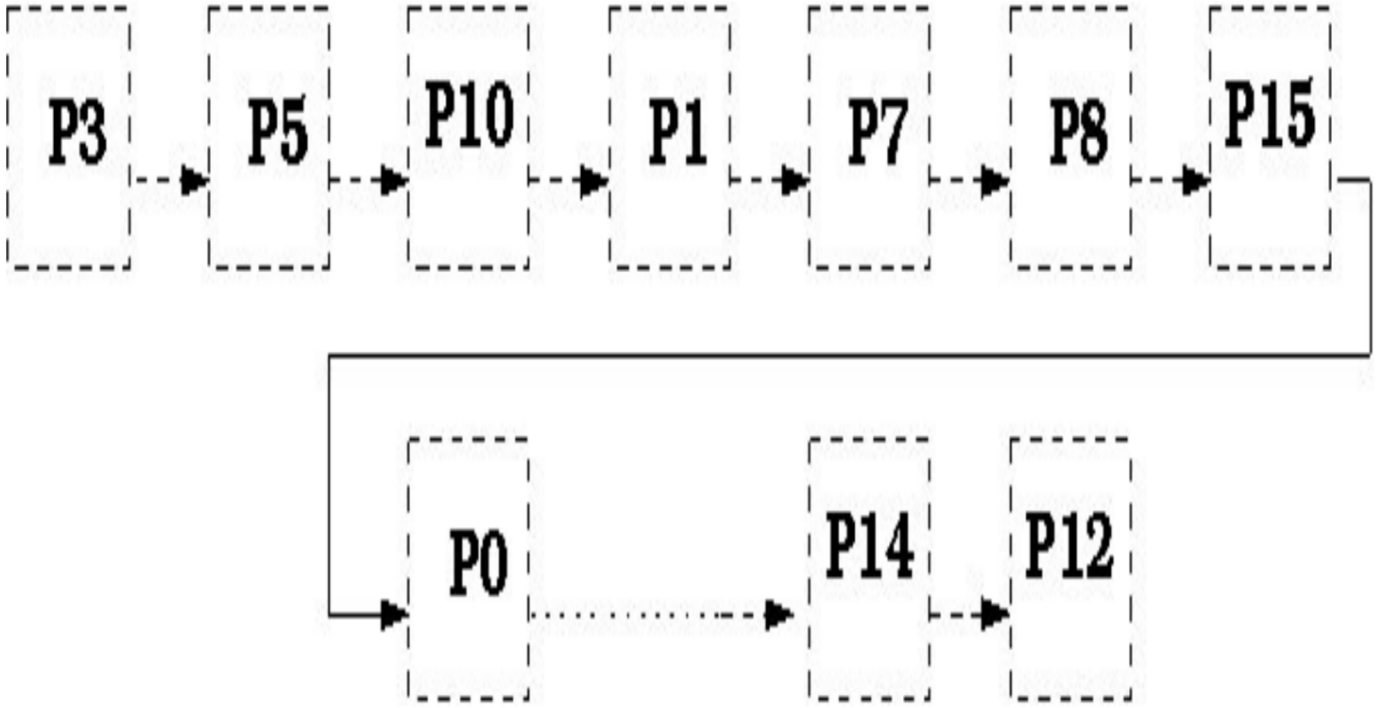
Yang-Anderson tournament tree



MCS algorithm (Dijkstra Prize, ‘06)

- In MCS, processes arrange themselves into a queue using special instructions.
- Single global variable, called L, is used.
- Processes perform a finite sequence of compare-and-swap on L while
  - Entering the MCS queue
  - Exiting the MCS queue
- For each process: the address of the process to wake up next is stored in local `next’ variable.

MCS-like queue obtained from YA tournament tree



Analysis: O(1) RMRs on average

$$T = \log n + n + 1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + \ldots + (\log n - 1) \cdot 2$$

Using arithmetico-geometric series:

T=3n-log(n)-2

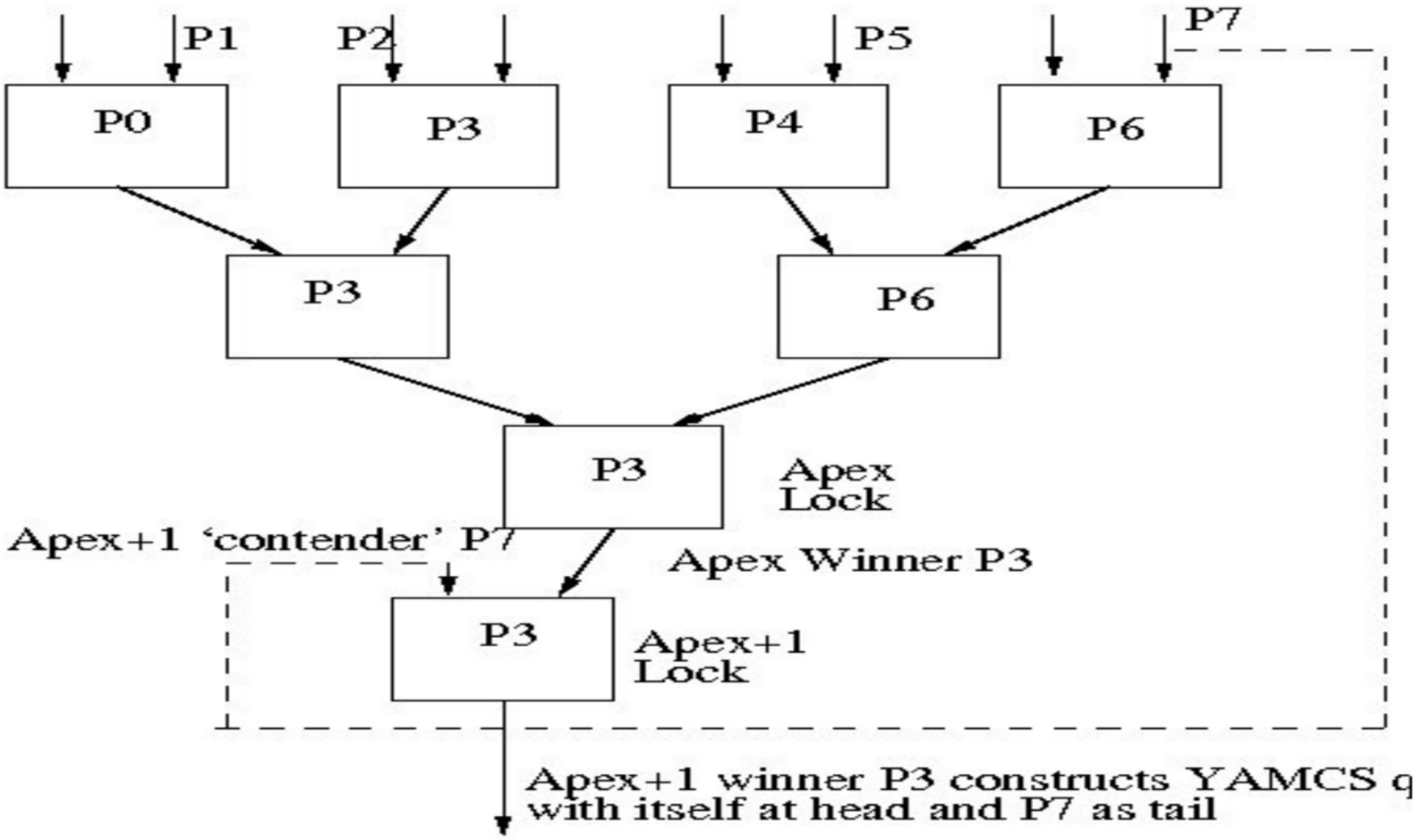
Hence, per process RMR is:

t=T/n=O(1)

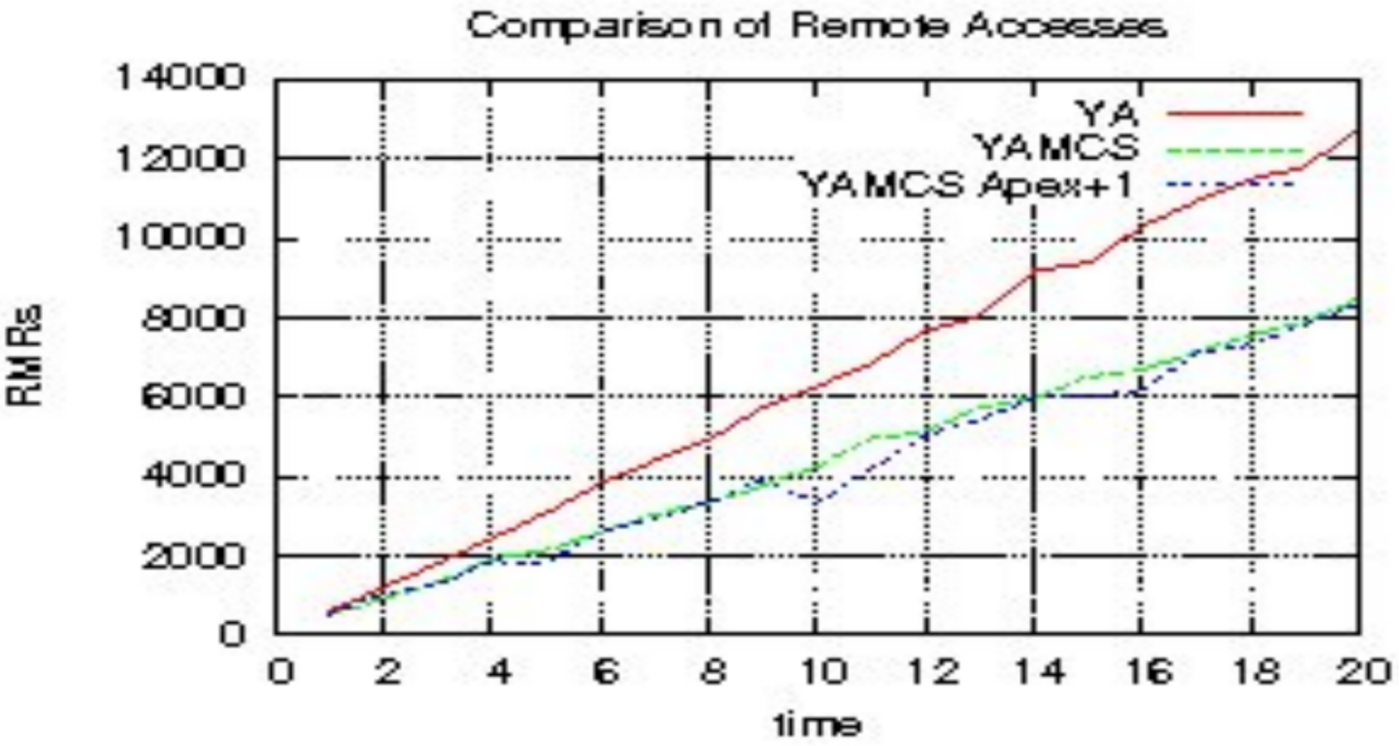
Problem with 1st-cut YA-MCS

- Note how P12, despite occupying apex, has to link-in last in the arranged queue.
- Because: being first in MCS queue instead would have made others apex-winners.
- Winners would build own MCS queues.
- Apex has to be guarded from occupation.
- Effect on P12: much larger *waiting time*.

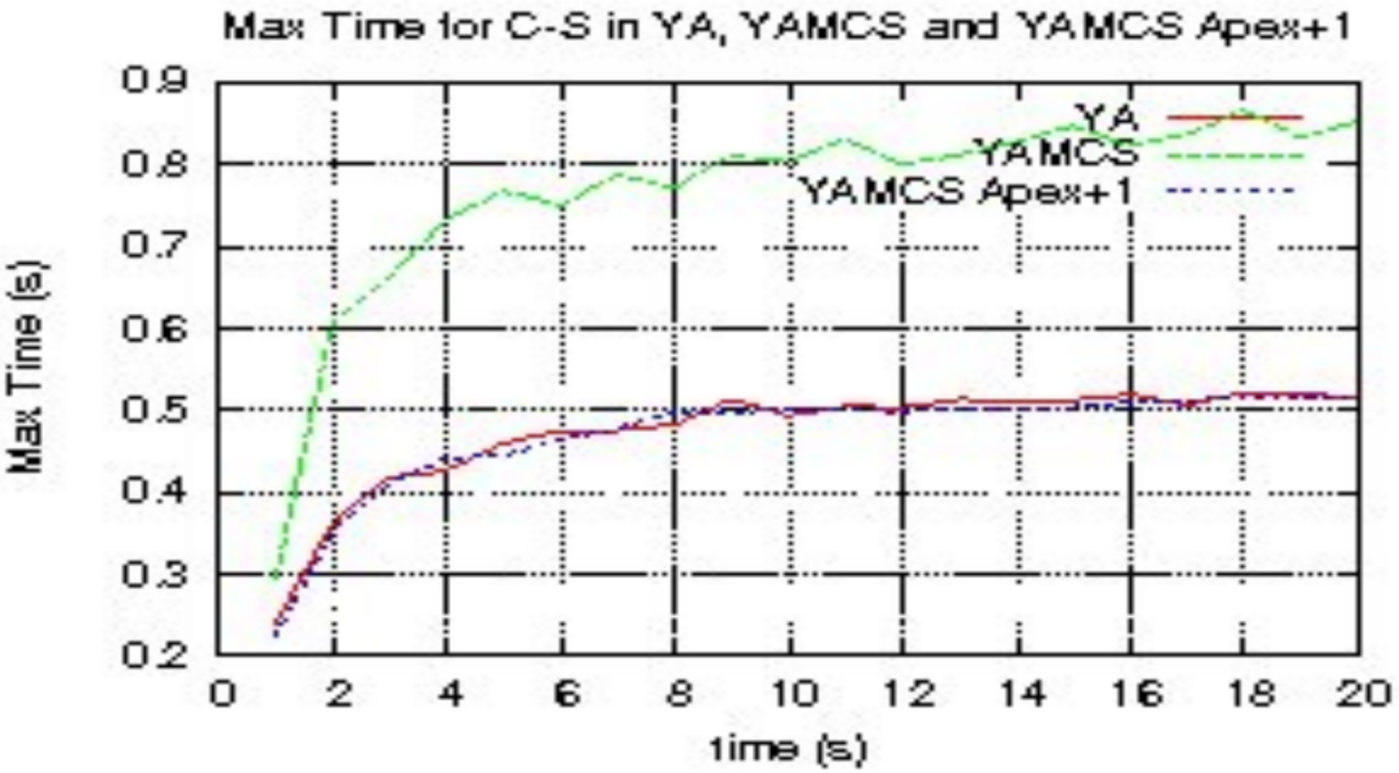
For n=8, Apex+1 YAMCS



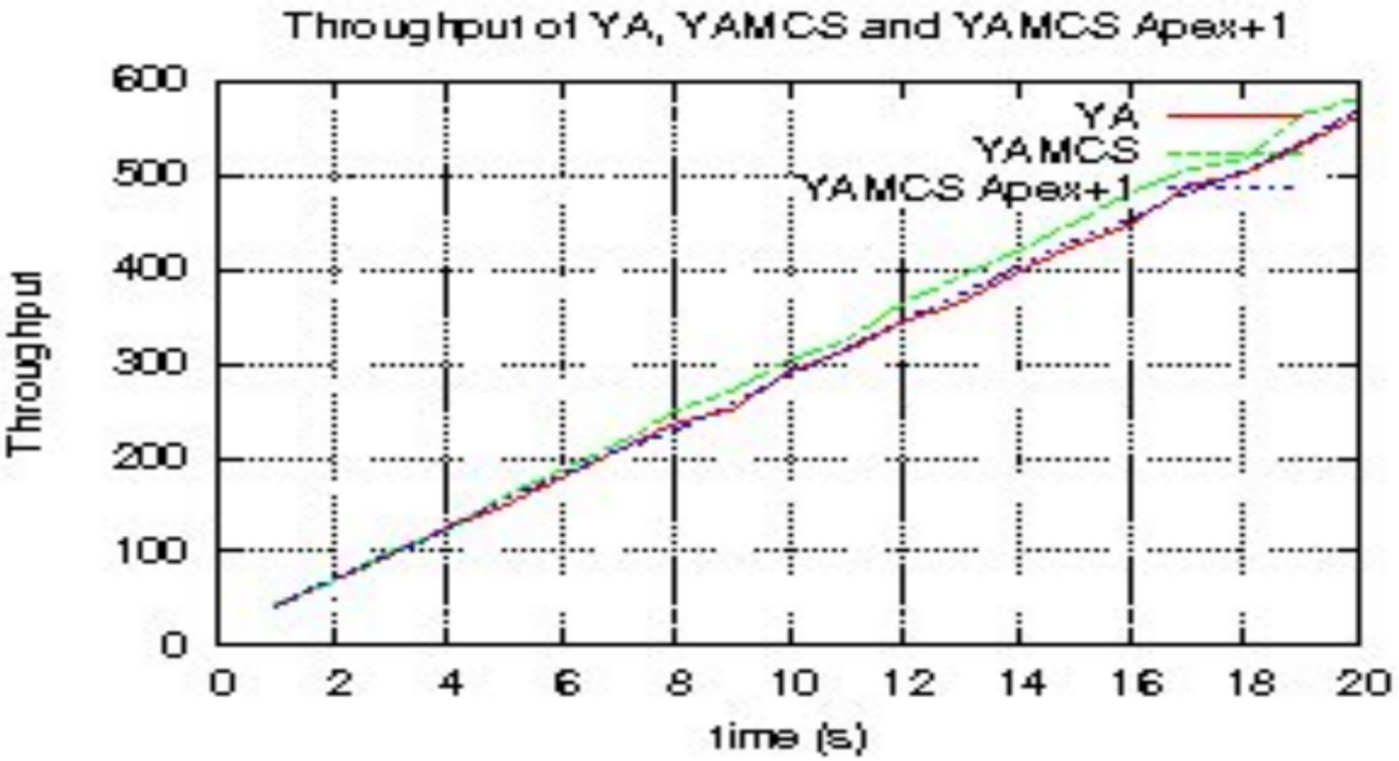
Comparing RMRs YA, YAMCS and apex+1 YAMCS



Comparison of waiting time



Comparison of throughput



Other contributions

- FIFO-respecting MCS – using only fetch-and-store – eluded authors of MCS.
- The FIFO-respecting MCS' in literature uses a `local' record per process – whose ownership, however, passes onto another.
- Our work proposes a FIFO-respecting MCS where the record ownership doesn't change, resulting in truly `local' record.