EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 17: October 25

Lecturer: Vijay Garg *Scribe: Cassidy Burden*

17.1 Sorting

Previously we have talked about three different sorting methods:

1. Brick Sort
2. Bitonic Sort
3. Merge Sort

We will finish parallel sorting by talking about the parallel merge sort algorithm.

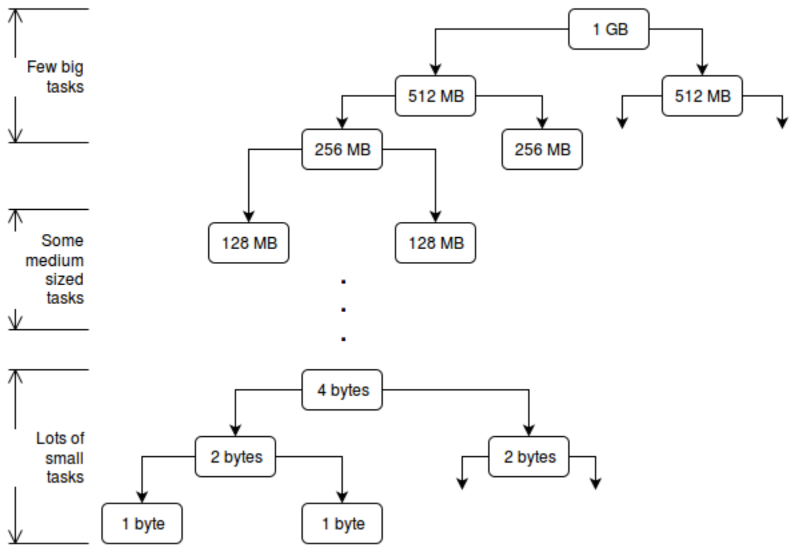
17.1.1 Parallel Merge Sort

Merge sort works by splitting the array into 2 halves, sorting both halves recursively, and then merging the two halves together.

The majority of the work done will be in the merge step. We have previously talked about a work optimal method of merging arrays in parallel by splitting the arrays into chunks and calculating the rank of each element.

We will break the problem into three chunks to allow us to most efficiently distribute our blocks and threads:

1. Large merges: do parallel scatter merge, assign one block per chunk of scatter
2. Medium sized merges: one merge per block, do parallel scatter merge
3. Small merges: one merge per thread, done sequentially



We eventually end up with a time complexity of $O(\log n)$ and work complexity of $O(n \log n)$.

17.2 Wait-free Registers

We now will continue our dicussion on how to build differing levels atomic registers. We will present the following hierarchy in which every register can be built with an unbounded amount of the preceding register.

To recap there are 3 different types of registers that act differently during a concurrent access:

- Safe - Returns any possible value
- Regular - Returns value currently being written or previously written value
- Atomic - Returns values such that history is linearizable

SRSW Safe Bit
(store previous value)
$\Downarrow$
SRSW Regular Bit
(unary construction)
$\Downarrow$
SRSW Regular Multivalue
(timestamp, keep seq number)
$\Downarrow$
SRSW Atomic Multivalue
(replication, timestamp, communication matrix)
$\Downarrow$
MRSW Atomic Multivalue
(bakery-style choosing of timestamp)
$\Downarrow$
MRMW Atomic Multivalue

17.2.1 SRSW Regular Bit

Bits can only have the four histories when looking at the two most recent writes:
$(0, 0)$ $(0, 1)$ $(1, 0)$ and $(1, 1)$

If we ensure that the only histories are $(0, 1)$ and $(1, 0)$ then our histories will contain all possible values for a bit. Thus when a concurrent access on the safe register occurs, we will either randomly return the value currently being written or the previous value, satisfying the regular register property. To ensure only those two histories occur, we will ignore all duplicate writes to the register.

17.2.2 SRSW Regular Multivalue

To build a regular register than can hold more than just two values, we will have a max size for our register X. To represent X values we will need X of our SRSW regular bit registers. We will keep these bits in an array and say the first non-zero bit in the array represents the value of our multivalue register.

```
SRSW_Reg_Bit A[X]

read():
  for i = [0 -> X]:
    if A[i] == 1: return i

set(x):
  A[x] = 1
  A[x-1 -> 0] = 0
```

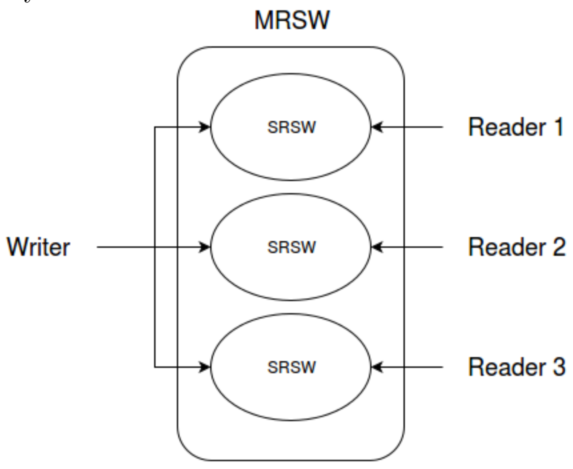
17.2.3 SRSW Atomic Multivalue

Next we will build an atomic register by attaching a sequence number or timestamp to our value. This is with the assumption that we cannot overflow our sequence number.

17.2.4 MRSW Atomic Multivalue

This will be the first register to account for multiple readers. Because we want this register to be atomic we want to make sure that each reader gets the same value once a writer commits a new value.

Because we only have SRSW registers, we will need a register for every reader. The single writer will write to every reader's register, and each reader will read from their own register. However, this is flawed, and can lead to a non-atomic history.



The final solution is to have n^2 SRSW registers, constructing a matrix. We will call this a communication matrix, and it essentially allows each register to share information. When a reader reads from the MRSW register it will want to alert other readers what it just read in order to satisfy atomicity. For example, writing 5 to $comm[i, j]$ is Reader i telling Reader j it just read value 5. Given n readers our MRSW register will work as such:

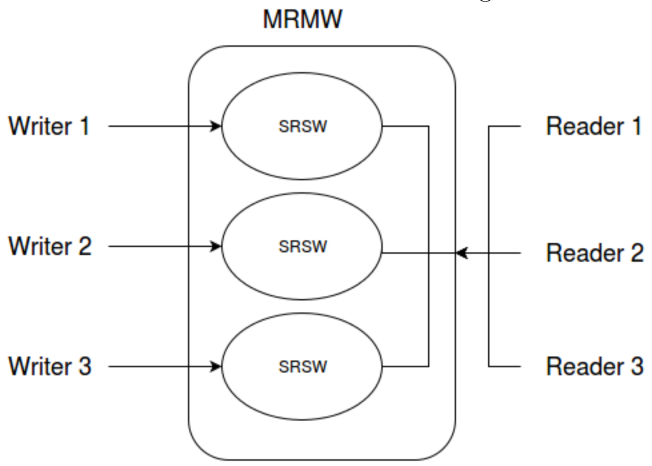
```
SRSW_Atomic_Multi V[n]
SRSW_Atomic_Multi comm[n, n]

set(x):
  V[0 -> n] = x

read(pid):
  val = max_TS(V[pid], comm[0, pid], comm[1, pid], ..., comm[n-1, pid])
  comm[pid, 0 -> pid] = val
  return val
```

17.2.5 MRMW Atomic Multivalue

For n writers we will need n MRSW registers. Because MRSW only allows one writer, each writer will have to write to their own register. Readers will read from whichever register has the highest timestamp.



However, because we can, in the worst case, have n writers deciding on a new TS at once, we need an algorithm to atomically decide on a timestamp. Lamport's Bakery has already solved this problem, and we will choose our writer's timestamp in the same way processes choose their tickets in Lamport's Bakery:

```
MRSW_Atomic_Multi Regs[n]

set(pid, x):
  ts = max_TS(Regs) + 1
  Regs[pid].set(x, ts, pid) // ts and pid together allow for total ordering

read():
  min_reg = Regs[0]
  for all r in Regs:
    if (min_reg.ts < r.ts) || (min_reg.ts == r.ts && min_reg.pid < r.pid):
      min_reg = r
  return min_reg.read()
```

References

[1] V.K. Garg, Introduction to Multicore Computing

[2] V.K. Garg, https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/tree/master/chapter5-wait-free