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

Lecture 10: September 27

Lecturer: Vijay Garg *Scribe: Prateek Srivastava*

10.1 Introduction

This lecture covers following items.

- Project outlines
- Merging Puzzle
- Consistency conditions

10.2 Projects

Projects need to be done in groups of 2 or 3. Undergrads should do project with other undergrads while grads with grads. For undergrads, aim would be to implement and turn in just one plot of the observations. For grads, they need to pick a topic, implement it and possibly improve on it. Topics can be chosen from recent conference proceedings as well.
If undergrads choose same topics, they can compete among themselves for the performance. Topics are posted on canvas. A brief overview is present here.

- Mutex algorithms not covered in class can be implemented e.g. colored bakery.
- Monitors with additional feature like abort.
- Parallel work scheduling and work distribution algorithms using some language e.g. language CILK.
- Concurrent trees, queues and skiplist. Avoids serialization by locking.
- Concurrent Hash-tables like Cuckoo Hashing, and Hopscotch Hashing.
- Poset, lattice theory. some part will be covered in class.
- Graph shortest path, spanning trees, max flows can be implemented on stampede on GPU.
- Implement sorting and compare performance.
- Image processing on GPU, scientific computation like fft and polynomial. Data mining algorithms like K-means.
- Text analysis used heavily by Google, pattern matching.
- Iphone/Android for this topic need to give exact details on what the app is going to do.

- Run model checker on algorithm and verify correctness.

- Lamport's Temporal Logic of Actions for specifying and verifying the correctness of concurrent algorithms.

- Verification of consistency conditions.

The project should contain some references.

10.3 Merge Puzzle contd...

10.3.1 Problem

Merge two sorted arrays of length n to form a sorted array of length 2n.

10.3.2 Solutions

(a) Sequential algorithm was like the merge sort.
(b) Parallel algorithm computes rank for each element which is the index into the final merged array. Each element computes its rank, using its own position and its position in the other array using binary search. Following table summarizes the time and work complexity of already discussed solution.

	Time	Work
Sequential	O(n)	O(n)
Parallel	O(log n)	O(n log n)

10.3.3 Work optimal parallel solution

We would like to reduce the work from $O(n * logn)$. This can be achieved using cascaded algorithm.
Divide the input array in log n groups. The starting element in each group is called the splitter.
No. of splitters $= O\left(\frac{n}{logn}\right)$

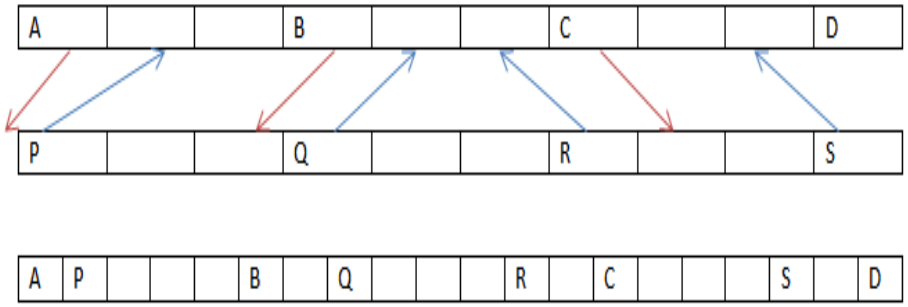


Figure 10.1: Merging Sorted Arrays

Fill the splitter in the target array by finding ranks as done in the parallel algorithm as shown in the figure
Find the sublist in first list say, α and second list say, β such that they are in between the splitters.
Number of such lists = $O\left(\frac{n}{logn}\right)$
Size of each list = $Ologn$)
We know that the $logn$ size lists can be merged in $Ologn$) using the sequential algorithm.

Following table summarizes the steps. Hence, the solution is work as well as time optimal. Although, there

	Time	Work
Step 1	O(log n)	O(n)
Step 2	O(log n)	O(n)
Total	O(log n)	O(n)

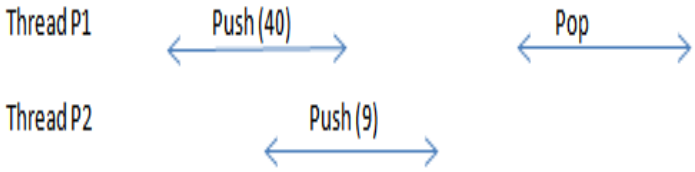
are solutions optimal then this one as well.

10.4 Parallel prefix sum puzzle

Given an array find an output array such that each element in output array is sum of input array elements till that index.
e.g. Input 3, 4, 19, 11, 13
Output 3, 7, 26, 37, 50
The algorithm is called scan.

10.5 Consistency condition

Consider a stack with following operations
push(40) push(9) pop (Pop should return 9 not 40)
This is sequential correctness
Now consider following situation.



The correctness of output depends on the definition of correct output. Lamport wrote a 2 page paper explaining what it means to be sequentially consistent in a multiprocessor environment.

10.5.1 Notations

A method call is split into two events

* *Invocation* method name + args e.g. f.foo(arg1, arg2)

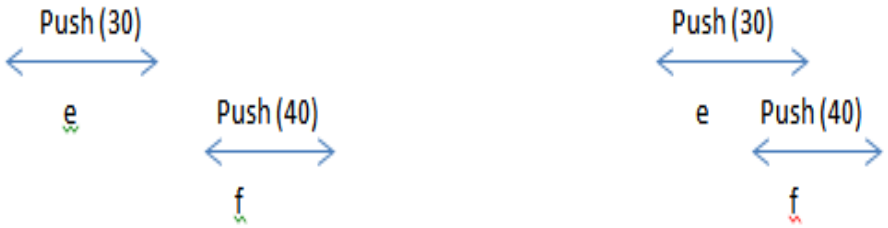
* *Response* result or exception

Consider, P f.foo(arg1, arg2)
This is an invocation of object f on thread P
foo is method name
arg1, arg2 are arguments
Consider, P f.response
This is the return value, for object f on thread P

We define following
inv(e) is invocation of e
resp(e) is response of e
proc(e) is process on which e runs

10.5.2 History

A sequence of invocations and responses constitutes history.
History $(H, <_H)$ is set of operations in real time order.
The relation $e <_H f$ holds, if resp(e) occurred before inv(f)



In first case, response of e occurs before invocation of f. In second case, e overlaps with f or is concurrent with f. There is no particular order

The relation $<_H$ is
(a) irreflexive (each element does not satisfy the relation with itself)
(b) transitive (if relation applies between successive members of a sequence, it must also apply between any two members taken in order)
This two conditions imply that the relation is asymmetric.
Since the relation $(H, <_H)$ is irreflexive and transitive, $(H, <_H)$ is a partial ordered set (poset).

10.5.3 Process Order

$e < f$ is in process order if

proc(e) = proc(f)

resp(e) occurred before inv(f)

10.5.4 Total order

A poset $(H, <_H)$ is a total order if for all distinct x,y $(<_X)$, satisfies $(x < y)$ or $(y < x)$

10.5.5 Sequential History

A history $(H, <_H)$ is sequential if $(<_H)$ is a total order.
A sequential history is legal if it satisfies sequential specs of the objects.

References

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