CA3404: Winter 2018
Distributed and Parallel Algorithms

Week - 3 : Distributed and parallel algorithms and their complexity

---

Course email-id: dispalgo2018@gmail.com

Dt: 24-01-2018, 27-01-2018, 31-01-2018, 07-02-2018

async

async says is, when followed by a statement, that that statement should run asynchronously.

You've come across the idea of asynchrony before, maybe if you used SwingUtilities or Java NIO,

where you specify some work that needs to be done asynchronously with the program.

That's the same idea here for parallel computing.

finish

So the idea behind finish is it specifies a scope of work where you can have a number of asynchronous tasks running.

And at the end of the finish scope, you're guaranteed

that all those tasks will have completed before you can proceed.

async and finish

For task creation:

“async ⟨stmt1⟩”, causes the parent task (i.e., the task executing the async statement) to create a new child task to execute the body of the async, ⟨stmt1⟩, asynchronously (i.e., before, after, or in parallel) with the remainder of the parent task. We also learned the finish notation

For task termination:

“finish ⟨stmt2⟩” causes the parent task to execute ⟨stmt2⟩, and then wait until ⟨stmt2⟩ and all async tasks created within ⟨stmt2⟩ have completed. Async and finish constructs may be arbitrarily nested.

async and finish

```
finish {
  async S1; // asynchronously compute sum of the lower half of the array
  S2;       // compute sum of the upper half of the array in parallel with S1
}
S3; // combine the two partial sums after both S1 and S2 have finished
```

async and finish

HW:

Go through the open-source Java-8 library called PCDP (for Parallel, Concurrent, and Distributed Programming), for which the source code is available at https://github.com/habanero-rice/pcdp.

PCDP contains APIs (application programming interfaces) that directly support async and finish constructs so that you can use them in real code as well.

Java fork-join framework

It is one of the most popular ways of exploiting multi-core parallelism in Java today.
And first see if we can extend it to a divide and conquer algorithm.
So now, let's think more object oriented-edly and say we have a class, ASUM.

And we have some fields - an array A, a lower bound, an upper bound, and a sum that we will compute.

And, more interestingly, we have a compute method, which is actually going to compute the sum, given the lower and the upper bounds and an array

Fork-join

```
CLASS ASUM { A, Lo, Hi, Sum;
        COMPUTE() { IF(Lo==Hi) Sum = A[Lo];
                        ELSE IF(Lo > Hi) Sum=0;
                        ELSE {
                                MID = (LO+HI) / 2;
                                L = NEW ASUM(A, LO, MID);
                                R = NEW ASUM(A, MID+1, HI);
ASYNC➜ L.FORK()         L.COMPUTE( );
                                R.COMPUTE( );
                                L.JOIN( );
                                SUM = L.SUM + R.SUM;
                }       }       }
```

A sketch of the Java code for the ASum class is as follows:

```
1  private static class ASum extends RecursiveAction {
2    int[] A; // input array
3    int LO, HI; // subrange
4    int SUM; // return value
5    . . .
6    @Override
7    protected void compute() {
8      SUM = 0;
9      for (int i = LO; i <= HI; i++) SUM += A[i];
10   } // compute()
11 }
```

FJ tasks are executed in a ForkJoinPool, which is a pool of Java threads. This pool supports the invokeAll() method that combines both the fork and join operations by executing a set of tasks in parallel, and waiting for their completion. For example, invokeAll(left,right) implicitly performs fork() operations on left and right, followed by join() operations on both objects.

HW: Go through the following links:

1. Tutorial on Java's Fork/Join framework

2. Documentation on Java's RecursiveAction class

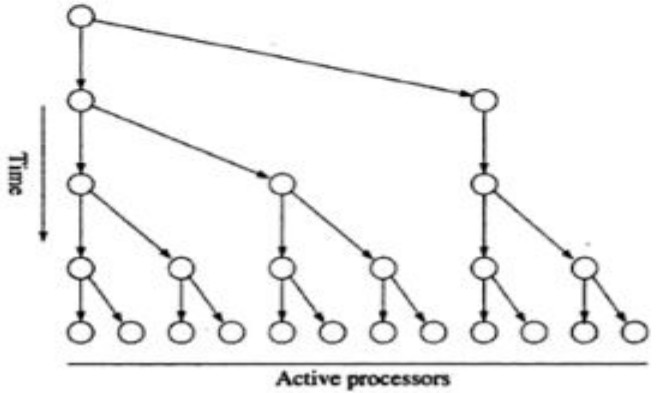
PRAM ALGORITHMS

PRAM algorithms work in two phases:

First phase: a sufficient number of processors are activated.

Second phase: These activated processors perform the computation in parallel.

Given a single active processor to begin with it is easy to see that $\lceil \log p \rceil$ activation steps are needed to activate p processors.



Meta-Instruction in the PRAM algorithms:
spawn (<processor names>)
To denote the logarithmic time activation of processors from a single active processor.

SECOND PHASE OF PRAM ALGORITHMS

To make the programs of the second phase of the PRAM algorithms easier to read, we allow references to global registers to be array references.

We assume there is a mapping from these array references to appropriate global registers.

The construct

for all <processor list> do <statement list> endfor

denotes a code segment to be executed **in parallel** by all the specified processors.

Besides the special constructs already described, we express PRAM algorithms using familiar control constructs: if…then….else…endif, for…endfor, while…endwhile, and repeat…until. The symbol ← denotes assignment.

PARALLEL REDUCTION

The binary tree is one of the most important paradigms of parallel computing.

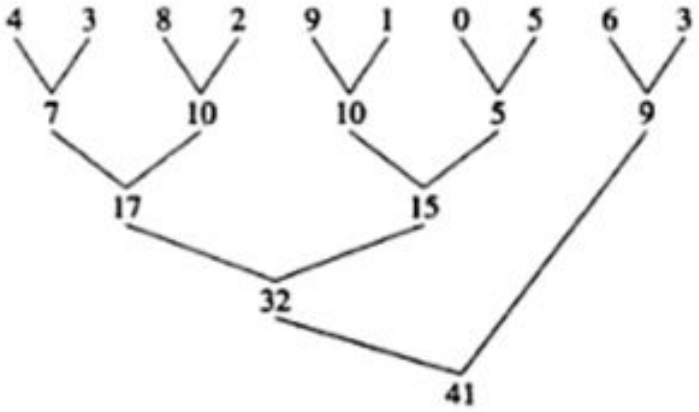
In the algorithms that we refer here, we consider an inverted binary tree.

- Data flows from the leaves to the root. These are called fan-in or reduction operations.

More formally, given a set of n values a1, a2, …, an and an associative binary operator $\oplus$, reduction is the process of computing $a1 \oplus a2 \oplus \cdots \oplus an$.

- Parallel Sum is an example of a reduction operation.

PARALLEL SUMMATION IS AN EXAMPLE OF REDUCTION



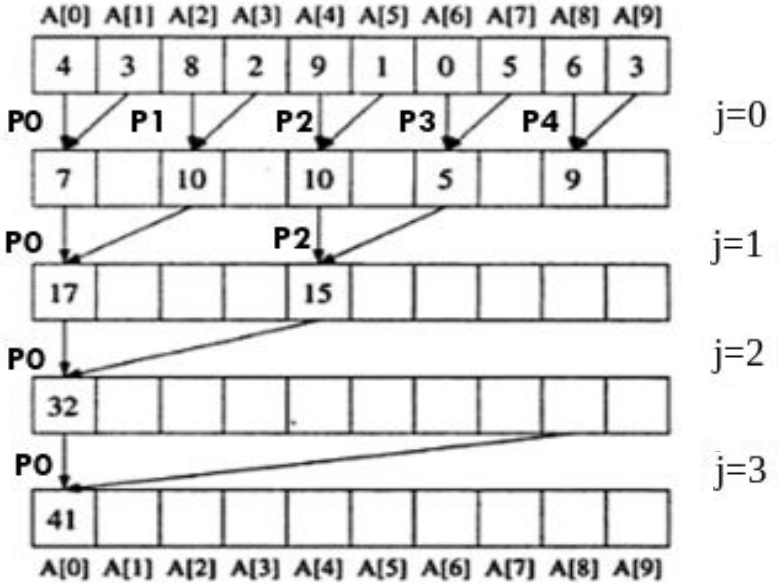
How do we write the PRAM algorithm for doing this summation?

GLOBAL ARRAY BASED EXECUTION

The processors in a PRAM algorithm manipulate data stored in global registers.

For adding n numbers we spawn $\lfloor\left(\frac{n}{2}\right)\rfloor$ processors.

Consider the example to generalize the algorithm.



GLOBAL ARRAY BASED EXECUTION

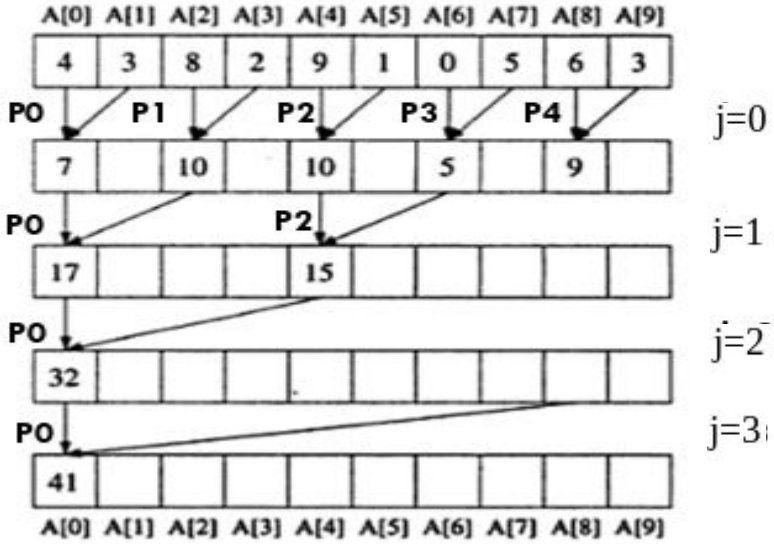
Each addition corresponds to:

$$A[2i] + A[2i + 2^j]$$

Note, the processor which is active has an i such that: $i \bmod 2^j = 0$ (ie. keep only those processors active).

Also check that the array does not go out of bound.

- ie, $2i + 2^j < n$



EREW PRAM PROGRAM

EREW PRAM algorithm to sum n elements using $\lfloor n/2 \rfloor$ processors.

```
SUM (EREW PRAM)
Initial condition: List of n ≥ 1 elements stored in A[0 ··· (n − 1)]
Final condition: Sum of elements stored in A[0]
Global variables: n, A[0 ··· (n − 1)], j
begin
  spawn (P0, P1, P2, ···, P⌊n/2⌋−1)
  for all Pi where 0 ≤ i ≤ ⌊n/2⌋ − 1 do
    for j ← 0 to ⌈lcg n⌉ − 1 do
      if i modulo 2^j = 0 and 2i + 2^j < n then
          A[2i] ← A[2i] + A[2i + 2^j]
      endif
    endfor
  endfor
end
```

COMPLEXITY

The SPAWN routine requires $\lceil log \left\lfloor \frac{n}{2} \right\rfloor \rceil$ doubling steps.

The sequential for loop executes $\lceil \log n \rceil$ times.

- Each iteration takes constant time.

Hence overall time complexity is $\Theta(\log n)$ given n/2 processors.

PREFIX SUM

Given a set of n values a1, a2, …, an, and an associative operation $\oplus$, the prefix sum problem is to calculate the n quantities:

a1,

a1 $\oplus$ a2,

…

a1 $\oplus$ a2 $\oplus$ … $\oplus$ an

AN APPLICATION OF PREFIX SUM

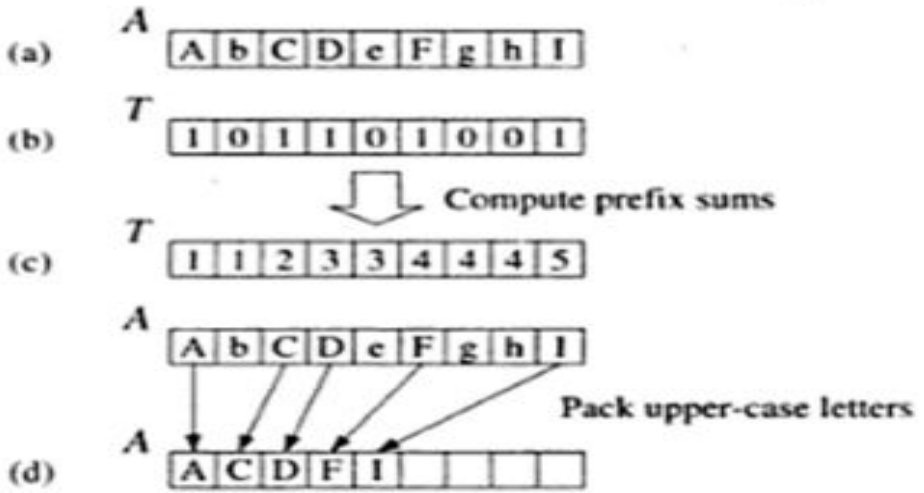
We are given an array A of n letters. We want to pack the uppercase letters in the initial portion of A while maintaining their order. The lower case letters are deleted.

a) Array A contains both uppercase and lowercase letters. We want to pack uppercase letters into beginning of A.

b) Array T contains a 1 for every uppercase letter, and 0 for lowercase.

c) Array T after prefix sum. For every element of A containing an uppercase letter, the corresponding element of T is the element's index in the packed array.

d) Array A after packing.



THE PRAM PSEUDOCODE

```
PREFIX.SUMS (CREW PRAM):
Initial condition: List of n ≥ 1 elements stored in A[0 ··· (n − 1)]
Final condition: Each element A[i] contains A[0] ⊕ A[1] ⊕ ··· ⊕ A[i]
Global variables: n, A[0 ... (n − 1)], j
begin
  spawn (P1, P2, ..... Pn−1)
  for all Pi where 1 ≤ i ≤ n − 1 do
    for j ← 0 to ⌈log n⌉ − 1 do
      if i − 2^j ≥ 0 then
          A[i] ← A[i] + A[i − 2^j]
      endif
    endfor
  endfor
end
```

COMPLEXITY

Running time is $t(n) = O(\lg n)$

Cost is $c(n) = p(n) \times t(n) = O(n \lg n)$

Note not cost optimal, as RAM takes $O(n)$

MAKING THE ALGORITHM COST OPTIMAL

Example Sequence – 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

Use $n / \lceil \lg n \rceil$ PEs with lg(n) items each

0,1,2,3 4,5,6,7 8,9,10,11 12,13,14,15

STEP 1: Each PE performs sequential prefix sum

0,1,3,6 4,9,15,22 8,17,27,38 12,25,39,54

STEP 2: Perform parallel prefix sum on last nr. in PEs

0,1,3,6 4,9,15,28 8,17,27,66 12,25,39,120

Now prefix value is correct for last number in each PE

STEP 3: Add last number of each sequence to incorrect sums in next sequence (in parallel)

0,1,3,6 10,15,21,28 36,45,55,66 78,91,105,120

A COST-OPTIMAL EREW ALGORITHM

In order to make the prefix algorithm optimal, we must reduce the cost by a factor of lg n.

We reduce the nr of processors by a factor of lg n (and check later to confirm the running time doesn't change).

Let $k = \lceil \lg n \rceil$ and $m = \lceil n/k \rceil$

The input sequence $X = (x_0, x_1, ..., x_{n-1})$ is partitioned into m subsequences $Y_0, Y_1, ..., Y_{m-1}$ with k items in each subsequence.

- While Y_{m-1} may have fewer than k items, without loss of generality (WLOG) we may assume that it has k items here.

Then all sequences have the form,

$$Y_i = (x_{ik}, x_{ik+1}, ..., x_{ik+k-1})$$

PRAM ALGORITHM OUTLINE

Step 1: For $0 \leq i < m$, each processor P_i computes the prefix computation of the sequence $Y_i = (x_{i*k}, x_{i*k+1}, \ldots, x_{i*k+k-1})$ using the RAM prefix algorithm (using $\oplus$) and stores prefix results as sequence $s_{i*k}, s_{i*k+1}, \ldots, s_{i*k+k-1}$.

Step 2: All m PEs execute the preceding PRAM prefix algorithm on the sequence $(s_{k-1}, s_{2k-1}, \ldots, s_{n-1})$

- Initially P_i holds s_{i*k-1}
- Afterwards P_i places the prefix sum $s_{k-1} \oplus \ldots \oplus s_{ik-1}$ in s_{ik-1}

Step 3: Finally, all P_i for $1 \leq i \leq m-1$ adjust their partial value sums for all but the final term in their partial sum subsequence by performing the computation

$$s_{ik+j} \leftarrow s_{ik+j} \oplus s_{ik-1}$$

for $0 \leq j \leq k-2$.

COMPLEXITY ANALYSIS

Analysis:

- Step 1 takes $O(k) = O(\lg n)$ time.
- Step 2 takes $O(\lg m) = O(\lg n/k)$

$$= O(\lg n - \lg k) = O(\lg n - \lg \lg n)$$

$$= O(\lg n)$$

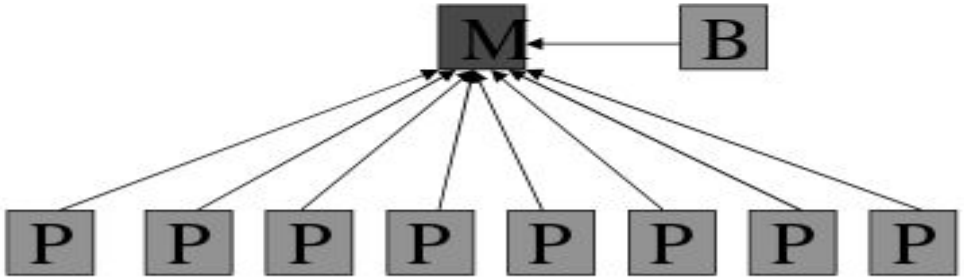
- Step 3 takes $O(k) = O(\lg n)$ time
- The running time for this algorithm is $O(\lg n)$.
- The cost is $O((\lg n) \times n/(\lg n)) = O(n)$
- Cost optimal, as the sequential time is $O(n)$

Can you write the complete pseudocode in the PRAM model?

BROADCASTING ON A PRAM

"Broadcast" can be done on CREW PRAM in O(1) steps:

- Broadcaster sends value to shared memory
- Processors read from shared memory



Requires lg(P) steps on EREW PRAM.

CONCURRENT WRITE — FINDING MAX

Finding max problem
- Given an array of n elements, find the maximum(s)
- sequential algorithm is O(n)

Data structure for parallel algorithm
- Array A[1..n]
- Array m[1..n]. m[i] is true if A[i] is the maximum
- Use n^2 processors

```
Fast_max(A, n)
1.for i = 1 to n do, in parallel
2.    m[i] = true        // A[i] is potentially maximum
3.for i = 1 to n, j = 1 to n do, in parallel
4.    if A[i] < A[j] then
5.        m[i] = false
6.for i = 1 to n do, in parallel
7.    if m[i] = true then max = A[i]
8.return max
```

Time complexity: O(1)

		A[j]					
		5	6	9	2	9	*m*
A[i]	5	F	T	T	F	T	F
	6	F	F	T	F	T	F
	9	F	F	F	F	F	T
	2	T	T	T	F	T	F
	9	F	F	F	F	F	T
							max 9

CRCW

- Hardware implementations are expensive
- Used infrequently
- Easier to program, runs faster, more powerful.
- Implemented hardware is slower than that of EREW
  - In reality one cannot find maximum in O(1) time

EREW

- Programming model is too restrictive
  - Cannot implement powerful algorithms

So, CREW is the most popular parallel model.