EE 382V: Parallel Algorithms **Summer 2017**

Lecture 2: June 9

Lecturer: Vijay Garg *Scribe: Eric Addison*

2.1 Introduction

During this lecture, we were exposed to the first parallel algorithm of the class (a simple array sum). We continued with this example to introduce and explore some of the fundamental concepts in parallel algorithm construction and analysis, including:

- the reduce operation
- parallel algorithm metrics (time, work, cost)
- Brent's Scheduling Principle
- Work-Optimal Algorithms

This set of lecture notes will briefly re-examine the topics covered in this lecture, in the order in which they appeared during class.

2.2 The `reduce` Operation

A `reduce` operation, for the purposes of this lecture, refers to the process of "reducing" a collection of values to a single result, presumably through the application of an *associative* binary operation. For example, consider an array of values `A` and a binary operation $*$ (standard multiplication, perhaps). Performing the `reduce` operation on `A`, in this case, is equivalent to:

$$r = \texttt{reduce}(\texttt{A}, *) = \texttt{A}[1] * \texttt{A}[2] * \texttt{A}[3] * ... * \texttt{A}[N]$$

Common `reduce` operations include minimum, maximum, sum, and product.

Associativity is desired so that the operation can be distributed among separate threads for parallel execution. Recall that a binary operation $*$ is associative if $a * (b * c) = (a * b) * b$.

2.3 Parallel Algorithm Metrics

Next, we took a few minutes so each of us could write an implementation of a simple array sum. Naturally, most of us wrote down about three to five lines of code to implement the trivial $O(n)$ sequential solution:

```
1  s:=0;
2  for i:=0 to n-1 do
3     s:= s + A[i];
```

As most of us know, this algorithm has time complexity $O(n)$ for an array of size n. We denote the time complexity by $T(n)$, i.e.:

$$T(n) \equiv \text{the time complexity of an algorithm with input size } n$$

For parallel algorithms, we are also interested in a new quantity called *work*, denoted $W(n)$:

$$W(n) \equiv \text{the work required for an algorithm with input size } n$$

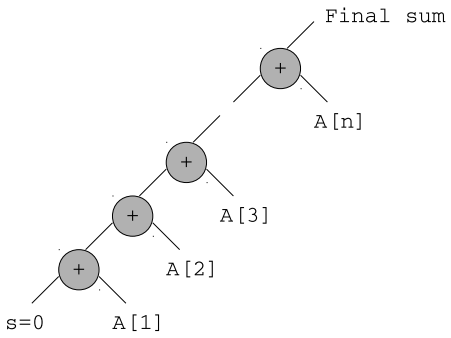
The work required by an algorithm was explained by the *Work-Depth Model of Computation*.

2.3.1 Work-Depth Model of Computation

The work-depth model consists of a directed acyclic graph (DAG) that represents the computation. For the sequential `reduce`-sum algorithm, the DAG looks like the figure on the right. In the model, each level of the graph represents one time step, so the running time is equal to the depth of the graph. We then have the following relationships:

- depth = time required on a parallel machine
- work = number of nodes in the graph

So here we have $W(n) = O(n)$.

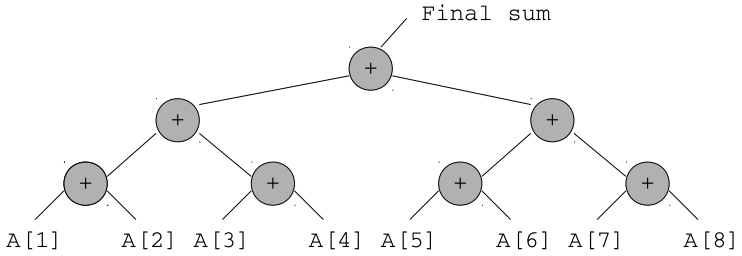


2.3.2 Parallel Sum

How can we improve on the sequential algorithm? We must adjust our thinking to move away from the old fashioned coding style of the 1960s and 70s, and start thinking in a 21st century way. Assume we have p processors available. The big question becomes:

Can we arrange the work in a binary tree?

Consider an array of length eight. The work-depth model can be arranged as a binary tree as follows:



We assign one processor to each of the graph nodes along the bottom of the tree (assume $p \geq 8$ for the moment). Since each level of the tree takes one time step to complete, we have parallel metrics $T(n) = O(\log n)$ (depth of tree) and $W(n) = O(n)$ (number of nodes). And of course, we know that reducing the time complexity from $O(n)$ to $(\log n)$ is a *big* deal!

2.4 Brent's Scheduling Principle

What if we have fewer than n processors available, i.e. $p < n$? Then we cannot assign each node along the bottom row of the work-depth model tree to a single processor. The required time $T(n, p)$ for an algorithm with a problem of size n on p processors is given by **BRENT'S SCHEDULING PRINCIPLE**, i.e.:

$$T(n,p) = \frac{W(n)}{p} + T(n),$$

where $T(n)$ is the depth of the work-depth model. We can prove that this is true.

Proof: Time to simulate level i of the work-depth model with p processors is

$$\left\lceil \frac{W^i(n)}{p} \right\rceil \leq \frac{W^i(n)}{p} + 1,$$

where $W^i(n)$ is the work required at level i of the model. The total running time is the sum of all the level times:

$$\begin{aligned} T(n,p) &\leq \sum_{i=1}^{T(n)} \left(\frac{W^i(n)}{p} + 1 \right) \\ &= \frac{1}{p} \sum_{i=1}^{T(n)} W^i(n) + \sum_{i=1}^{T(n)} 1 \\ &= \frac{W(n)}{p} + T(n) \end{aligned}$$

The first line of the previous block of equations is an inequality, however when we work in asymptotic quantities, this can be disregarded and the strict equality of Brent's Scheduling Principle is valid. More formally, we might want to write $T(n,p) = O\left(\frac{W(n)}{p} + T(n)\right)$.

□

Consider the previous sum example with $p = O(n/\log n)$. We then have parallel run time:

$$\begin{aligned} T(n,p) &= \frac{O(n)}{O(n/\log n)} + O(\log n) \\ &= O\left(\frac{n}{n/\log n} + \log n \right) \\ &= O(\log n) \end{aligned}$$

This implies that in order to compute a `reduce` operation in $O(\log n)$ time, we only need $O(n/\log n)$ processors! (not $O(n)$!)

2.4.1 Cost

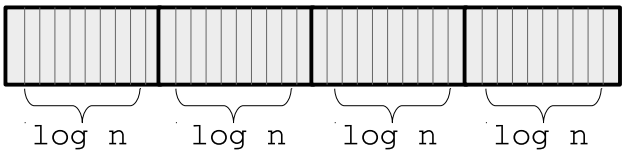
We also defined a fourth parallel metric, the *cost*, with $C(n,p) = T(n,p) \cdot p$, though we did not discuss it much in this lecture.

2.5 Work-Optimal Algorithms

Let $T_{\text{seq}}(n)$ be the best case sequential time complexity for some problem. A parallel algorithm $\mathcal{A}$ is said to be **WORK OPTIMAL** if its required work $W(n) = O(T_{\text{seq}}(n))$.

2.5.1 Cascading

Next we will use the notion of *cascading* two algorithms together to achieve work optimality. Consider breaking the array `A` up into $n/\log n$ segments of size $\log n$:



The cascade technique works by applying an algorithm $\mathcal{A}$ with the desired work complexity to each of the segments (where $\mathcal{A}$ could be sequential or parallel), and then applying a parallel algorithm $\mathcal{B}$ on the results. For example, let's say that $\mathcal{A}$ is a sequential algorithm with $T_{\mathcal{A}}(n) = O(n)$ and $W_{\mathcal{A}}(n) = O(n)$, and $\mathcal{B}$ is a non-optimal parallel algorithm with $T_{\mathcal{B}}(n) = O(\log n)$ and $W_{\mathcal{B}}(n) = O(n \log n)$. The cascade technique proceeds as:

step 1: run $\mathcal{A}$ on each segment
Total time complexity $T(n) = O(\log n)$
Total work complexity $W(n) = O(n)$

step 2: run $\mathcal{B}$ on the segment results
Total time complexity $T(n) = O(\log(n/\log n)) < O(\log n)$
Total work complexity $W(n) = O(n/\log n \cdot \log(n/\log n)) < O(n)$

Total Complexity: sum of step 1 and step 2 complexities
Total time complexity $T(n) = O(\log n) + O(\log n) = O(\log n)$
Total work complexity $W(n) = O(n) + O(n) = O(n)$

Use of the cascade technique in this case has produced a work complexity of $O(n)$, which makes it a work optimal algorithm. Cool!

2.6 More Parallel `reduce` Analysis

See the official course lecture notes for a convenient table comparing complexities of different algorithms.

2.6.1 Parallel `reduce`-sum: Binary Tree

The initial parallel algorithm we examined for the `reduce`-sum problem was to rearrange the work-depth model into a binary tree (instead of a "stupid" tree): see section 2.3.2. A pseudo-code implementation of this algorithm is:

```
1  i:= get_my_id();
2  B[i]:= A[i]
3  for h:= 1 to log n do
4     if (i <= n/(2^h)) then
5        B[i]:= B[2i-1] + B[2i];
6  print B[1];
```

Assume 1-based array indexing. By relying on the lock-step feature of our PRAM model, this implementation builds the binary tree upward by overwriting array entries on the left side of the array at each time step, finally producing the sum at position `B[1]`. The complexity of this algorithm is:

$$T(n) = O(\log n), \qquad W(n) = O(n)$$

Since the best case sequential time for `reduce`-sum is $T_{\text{seq}}(n) = O(n)$, this algorithm is **work optimal**.

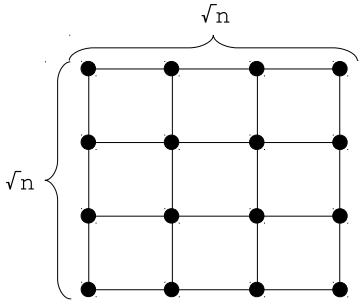
2.6.2 Parallel `reduce`-sum: Network Models

What if we change our parallel computation model? We can visualize a 2D mesh network of n nodes by constructing a $\sqrt{n} \times \sqrt{n}$ grid as shown on the right. If we perform the sum by summing across all rows (in parallel), and then down along one column, the resulting complexities are

$$T(n) = O(\sqrt{n}), \qquad W(n) = O(n)$$

If the network is arranged as a hypercube of dimension d, and the sum is performed by summing along one dimension at a time, then the complexities become:

$$T(n) = O(\log n), \qquad W(n) = O(n)$$



2.6.3 Parallel `reduce`-max: All Pairs

Can we achieve a better time complexity than $O(\log n)$ for a `reduce` operation? Note that our previous attempts only made use of an EREW PRAM model ... no concurrent reads or writes were needed. Consider evaluating the `reduce`-max operation on a common-CRCW machine. In this case, it is possible to achieve a time complexity of $O(1)$! This is done with a parallel technique known as **all pairs**.

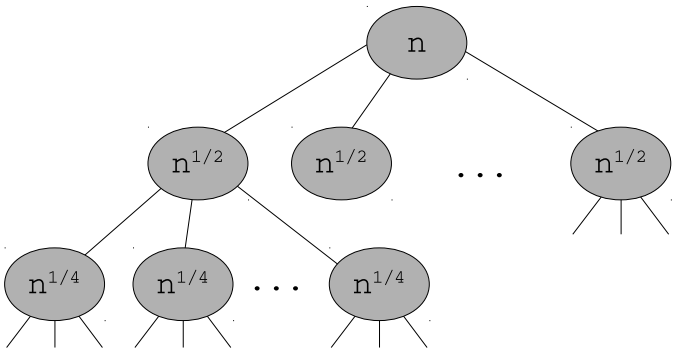
Instead of utilizing only n processors for this problem of size n, let's use n^2. Essentially we will use one processor to perform each of the n^2 possible comparisons. An array `isBiggest` will be initialized to all `true` values, and all but the actual max value will attempt to change the corresponding array value to `false`. Pseudo-code for this algorithm follows:

```
 1  for all i parallel do
 2    isBiggest[i]:= true
 3
 4  for all i,j: i!=j parallel do
 5    if(A[j]>A[i])
 6      isBiggest[i]:= false
 7
 8  for all i parallel do
 9    if isBiggest[i]
10      max:= A[i]
```

Since each of the `for` loops are performed completely in parallel, each loop has time complexity $T_{\text{loop}}(n) = O(1)$, and so the total time is also $O(1)$. Unfortunately, the loop over i, j results in $O(n^2)$ work, so this algorithm, while very time efficient, is not work-optimal.

2.6.4 Parallel `reduce`-max: Doubly Logarithmic

Clearly we can't do better than $T(n) = O(1)$ for time complexity, but we can still do better by finding a work-optimal algorithm. Consider again the `reduce`-max problem of size n, and imagine dividing the n elements into $\sqrt{n}$ segments of size $\sqrt{n}$. Take each of those new segments and split each of them in the same way, i.e. each segment of size $n^{1/2}$ will be split into $n^{1/4}$ new segments of size $n^{1/4}$, and so on. This splitting can be visualized in a tree:



At the i^{th} level of the tree (for $i > 0$), there are $\prod_{j=1}^{i} n^{2^{-j}}$ segments containing $n^{2^{-i}}$ elements each. We can confirm that there are still n elements at every level:

$$n^{2^{-i}} \prod_{j=1}^{i} n^{2^{-j}} = n^{2^{-i}+\sum_{j=1}^{i} 2^{-j}} = n^{2^{-i}-1+\frac{1-2^{-i-1}}{1-1/2}} = n^{2^{-i}-1+2-2^{-i}} = n$$

How many levels of the tree are there? Assume that the original size $n = 2^{2^h}$. After i splits, each segment contains $n_i = (2^{2^h})^{2^{-i}} = 2^{2^h 2^{-i}} = 2^{2^{h-i}}$. After h levels, there will be $2^{2^0} = 2$ elements per segment, the tree cannot be split any further, so we can say the tree has height h. With a work-depth model that has depth h, if we can evaluate the max at each level of the tree in $O(1)$ time (we can do this with the all-pairs algorithm), then the time complexity is given by the relation of h to n, namely:

$$T(n) = O(h) = O(\log \log n)$$

This is a *doubly logarithmic* time complexity, and is really really good! For some sense of the smallness of the double log, assume a tree height of $h = 10$ (a pretty small run time). This corresponds to an input of size $2^{2^{10}} \sim O(10^{300})$, a positively *huge* number!

For work complexity, notice that each level of the tree requires $O(n)$ work. Consider tree level i, where we will use the all-pairs algorithm locally at each tree node to compute the max of its children. At this level, each node must aggregate the results from its $n^i_{\text{ch}} = n^{2^{-i-1}}$ children. The all-pairs algorithm then results in time $T^i_{\text{node}}(n) = O(1)$ and requires $(n^i_{ch})^2$ work. This must be done for all $N^i_{\text{nodes}} = \prod_{j=1}^{i} n^{2^{-j}}$ nodes at a given level, so the total work required for level i is:

$$W^i(n) = O((n^i_{ch})^2 N^i_{\text{nodes}}) = O\left(n^{2^{-i}} \prod_{j=1}^{i} n^{2^{-j}}\right) = O(n)$$

And, if the work at every level is $W^i(n) = O(n)$, then the cumulative work at all depths is

$$W(n) = O(n \log n \log n),$$

which means this algorithm is *not* work-optimal.

2.6.5 Parallel `reduce`-max: Cascaded Doubly Logarithmic (optimal)

By applying the cascade technique to the doubly logarithmic tree, we can describe a very efficient work-optimal algorithm for `reduce`-max. Referring back to the section on cascading, let's assign algorithm $\mathcal{A}$ as the standard $O(n)$ sequential algorithm, and algorithm $\mathcal{B}$ as the doubly logarithmic algorithm. This time, split the initial array into $N_{\text{seg}} = O(n/(\log \log n))$ segments of size $n_{\text{seg}} = O(\log \log n)$. Now the cascade technique proceeds as:

step 1: run $\mathcal{A}$ on each segment (in parallel)
Total time complexity $T(n) = O(\log \log n)$
Total work complexity $W(n) = O(\log \log n)$

step 2: run $\mathcal{B}$ on the segment results
Total time complexity $T(n) = O(\log \log(n/(\log \log n))) < O(\log \log n)$
Total work complexity $W(n) = O(n/(\log \log n) * \log \log(n/(\log \log n))) < O(n)$

Total Complexity: sum of step 1 and step 2 complexities
Total time complexity $T(n) = O(\log \log n) + O(\log \log n) = O(\log \log n)$
Total work complexity $W(n) = O(n) + O(n) = O(n)$

This is a **work-optimal**, efficient algorithm for the `reduce`-max problem. Since we use the all-pairs algorithm as part of the doubly logarithmic algorithm, it is written for a common-CRCW PRAM model.