1 Overview

In the last lecture we covered the Circuit Model, defined runtime and work, and proved Brent’s Theorem. We previously defined these as:

- Runtime: \( t_p(n) = t(n,p) \) for \( p \) number of processor
- Cost: \( \text{cost}(n) = t_p(n) \cdot p \)
- Work: \( w(n) = \) total number of operations

In this lecture we will elaborate on runtime, work and cost and their definitions without a dependence on the number of processors, \( p \). We will define parallelism and work efficiency.

2 Runtime, Work & Cost

We want to define runtime, work, and cost of an algorithm independently of the number of processors, \( p \), that will be used.

2.1 Cost vs. Work when \( p = n \)

We’ll use the following summation algorithm to illustrate the next few points:

```plaintext
sum(A[1...n]){
    if n == 1
        return 1
    else
        in parallel do {
            l = sum(A[1...\lfloor n/2 \rfloor])
            r = sum(A[\lceil n/2 \rceil + 1...n])
        }
        return l + r
}
```

Let’s say that the number of processors, \( p \), is equal to the number of elements, \( n \), being summed. Now, we’ll calculate the runtime, work and cost for this algorithm. Figure 1 is a depiction of the work done on each level of the recursive Sum algorithm.
Similarly to solving a recurrence, we can define the runtime as follows:

\[
\begin{align*}
  t_{p=n}(n) &= \begin{cases} 
    t_p \left( \frac{n}{2} \right) + \Theta(1) & \text{if } n > 1 \\
    \Theta(1) & \text{otherwise} 
  \end{cases} \\
  &= \Theta(\log n)
\end{align*}
\]

By looking at Figure 1, we can see that the equation given above for runtime is true. We know that constant work is done at each node. The array is split in half on each level, which gives us the \( t_p \left( \frac{n}{2} \right) \) when \( n > 1 \). We only need to follow one path down the tree since we assume that the nodes on each level are done in parallel. When \( n = 1 \), we hit the base case and constant work is done at each node.

It may seem odd that the number of processors has been halved in this equation, \( t_{p=n}(n) \rightarrow t_p \left( \frac{n}{2} \right) + \Theta(1) \)

We can see that this works out because of our assumption that \( p = n \):

\[
\begin{align*}
  t_p(n) &= \\
  t(n, p) &= t \left( \frac{n}{2}, \frac{p}{2} \right) \\
  t(n, n) &= t \left( \frac{n}{2}, \frac{n}{2} \right) + \Theta(1)
\end{align*}
\]

Let \( g(n) = t(n, n) \), then \( g(n) = g \left( \frac{n}{2} \right) + \Theta(1) \), which can then be solved.

\[
\begin{align*}
  g(n) &= \Theta(\log n) \\
  t(n, n) &= \Theta(\log n)
\end{align*}
\]

We can now determine the work done in the algorithm.

\[
\begin{align*}
  w(n) &= \begin{cases} 
    2w \left( \frac{n}{2} \right) + \Theta(1) & \text{if } n > 1 \\
    \Theta(1) & \text{otherwise} 
  \end{cases} \\
  &= \Theta(n)
\end{align*}
\]
Looking at the tree in Figure 1, the right column shows the amount of work done on each level. We can see that there are $\log n + 1$ levels and each level has $\frac{n^2}{2^i}$ work, where $i$ is equal to the level. We see that the work could then be described as

$$\sum_{i=0}^{n^2} \frac{n^2}{2^i} = 2n - 1 \leq 2n$$

Thus, we get the idea that work $= \Theta(n)$, which can be proved using substitution or the Master method. We can see that work is the same as it would be for the sequential sum algorithm.

We previously defined cost to be $p \cdot t_p(n)$. Thus,

$$cost = n \cdot \Theta(\log n) = \Theta(n \log n)$$

### 2.1.1 Work vs. Cost

We can see that cost is not equal to work. In most cases, cost is the same as work, but it is not always the case, as we have seen here. Essentially, work is a measure of the total number of operations used in the algorithm, or the number of operations that one processor would need to perform when $p = 1$. Cost, on the other hand, is dependent on the number of processors being used. If $p > n$, some of the processors will not be utilized on most of the levels, meaning that cost will be greater than the actual number of operations needed for the algorithm. If $p < n$, then $w(n)$ work will be done. We can see that the following must then be true: $cost(n) \geq w(n)$.

### 2.2 Summation with $\log(n)$ base case

Let’s rewrite the earlier sum algorithm as follows:

```plaintext
sum(A[1...n], N){
  if $n \leq \log N$ {
    for $i = 1$ to $n$ {sum = sum + A[i]}
    return sum
  }
  else
    in parallel do {
      $l = sum(A[1\ldots \frac{n}{2}], N)$
      $r = sum(A[\frac{n}{2} + 1 \ldots n], N)$
    }
    return $l + r
}
```

We can see that the tree for this algorithm is slightly different from earlier in Figure 2, below.
We will now determine the runtime of this algorithm.

\[
t_p(n) = \begin{cases} 
  t_p \left( \frac{n}{2} \right) + \Theta(1) & \text{if } n > \log n \\
  \Theta(n) & \text{otherwise}
\end{cases}
\]

\[= \Theta(\log n)\]

We see that the depth of the tree is now \(\log \left( \frac{N}{\log N} \right)\). We obtain this by finding the difference in depth of the original \(\log n\) tree and the depth below the level where \(N = \log n\).

\[
\log \left( \frac{N}{\log N} \right) = \log N - \log \log N = \log p
\]

Each level above the leaves requires constant time because there are \(p\) processors that work in parallel. The last level of leaves has \(\log n\) elements at each node. These elements need to be summed up by one processor, which will take linear time, meaning that the leaves will take \(\log n\) time to process. By this reasoning, we get:

\[
t_p(n) = \Theta(\log N - \log \log N + \log N)
\]

\[= O(\log N)\]

In order to achieve this runtime, we need a processor for each leaf, so

\[
p = 2^{\log \frac{N}{\log N}} = \frac{N}{\log N}
\]

The work done in the algorithm is

\[
w(n) = \begin{cases} 
  2w \left( \frac{n}{2} \right) + \Theta(1) & \text{if } n > \log n \\
  \Theta(\log n) & \text{otherwise}
\end{cases}
\]
= Θ(n)

We can use the Master Method to verify that this is true:

- Compare Θ(1) to $n^{\log_2 2}$
- Since Θ(1) = O(n), by Case 1,
  \( w(n) = Θ(n) \)

Using the equation stated earlier, we can determine the cost.

\[
\begin{align*}
  \text{Cost}(n) &= t_p(n) \cdot p \\
  &= Θ(\log n) \cdot \frac{n}{\log n} \\
  &= Θ(n)
\end{align*}
\]

### 2.3 Without Brent’s Theorem

If we didn’t know about Brent’s Theorem, then we would instead, design this algorithm so that the base case is $\frac{N}{p}$. We can see the tree for this algorithm in Figure 3.

![Figure 3 Tree for the (N/p) Sum algorithm](image)

The runtime will now be

\[
  t_p(n) = \begin{cases} 
  t_{\frac{N}{p}}(\frac{n}{2}) + Θ(1) & \text{if } n > \frac{N}{p} \\
  Θ(n) & \text{otherwise}
  \end{cases}
\]

= Θ(\log p + \frac{N}{p})

Work would still be Θ(N), and cost will be Θ(p log p + N).

We would want cost to equal work. We know that cost is equal to Θ(p log p + N), and work
is equal to $\Theta(N)$. In order to make cost equal to $\Theta(N)$, we need the term $p \log p$ to be dominated by $N$. If we choose $p$ where $p < \frac{N}{\log N}$, then we can see that $p \log p < N$:

$$\frac{N}{\log N} \log \frac{N}{\log N} < N$$
$$\frac{N}{\log N} (\log N - \log \log N) < N$$
$$N(1 - \frac{\log \log N}{\log N}) < N$$
$$N - \frac{N \log \log N}{\log N} < N$$

$N$ should be bigger than one for the algorithm, which makes the term $\frac{N \log \log N}{\log N}$ positive, meaning the inequality holds.

So, we say that $P < \frac{N}{\log N}$.

### 3 Efficiency & Parallelism

**Definition 1.** *Efficiency* is defined as $t_1(n)/w(n)$, where $t_1(n)$ is the runtime of the best sequential algorithm, and $w(n)$ is the work of a parallel algorithm. Efficiency is always less than or equal to one since $w(n) \geq t_1(n)$.

**Definition 2.** A parallel algorithm is *work-efficient* if efficiency is equal to 1.

**Definition 3.** We define *parallelism* to be $w(n)/t_\infty(n)$, work over the time needed for max processors to complete the algorithm.

In our summation example earlier, $w(n) = N$, and $t_\infty(n) = O(\log n)$, since it could not be completed in less time.

**Theorem 4.** If $p \leq \frac{w(n)}{t_\infty(n)}$, then cost $= \Theta(w(n))$.

**Proof.**

$$t_p(n) = \frac{w(n)}{p} + t_\infty(n)$$

(By Brent’s theorem. Note: $t_\infty(n) = T(n)$ for circuits.)

$$\text{cost} = p \cdot t_p(n)$$
$$= p\left(\frac{w(n)}{p} + t_\infty(n)\right)$$
$$= w(n) + p \cdot t_\infty(n)$$
$$\leq w(n) + w(n)$$
$$= 2w(n)$$
$$= \Theta(w(n))$$

Thus, if we create an algorithm so that it runs with max parallelism and uses fewer $p$, then cost will be equal to work.

Looking back at the equation for parallelism, $\frac{w(n)}{t_\infty(n)}$, we can think of parallelism as the maximum number of processors to use so that we’re not wasting parallel resources. If we stop the algorithm so that the number of leaves is equal to the number of processors, $p$, that we have, then the work done at the leaves will dominate the work on internal nodes. We know that $w(n)$ is fixed since it is equal to the best sequential algorithm’s runtime. We want to try to minimize $t_\infty(n)$. We do this
by keeping work efficient while making the critical path, \( t_\infty(n) \), length as short as possible. Then, 
\[
t_p(n) = \frac{w(n)}{p} + t_\infty(n)
\]
and minimizing \( t_\infty(n) \) maximizes \( p \), which minimizes \( \frac{w(n)}{p} \) and the runtime. Since \( t_\infty(n) \) cannot be greater than \( \frac{w(n)}{p} \) because of \( p \), we need to minimize \( t_\infty(n) \).

3.1 Simulation of CRCW PRAM algorithms on EREW PRAM

**Theorem 5.** An algorithm \( A \) that runs in time \( T_A(n) \) on a \( p \)-processor CRCW PRAM can be implemented on a \( p \)-processor EREW PRAM in time \( t'_p = \Theta(T_A(n) \cdot \log p) \).

The proof will be shown in the next lecture.