# How Fast are Parallel Programs?

Parallel Programming in Scala

Viktor Kuncak

# How long does our computation take?

Performance: a key motivation for parallelism

How to estimate it?

- empirical measurement
- asymptotic analysis

Asymptotic analysis is important to understand how algorithms scale when:

- inputs get larger
- we have more hardware parallelism available

We examine worst-case (as opposed to average) bounds

# Asymptotic analysis of sequential running time

You have previously learned how to concisely characterize behavior of *sequential* programs using the number of operations they perform as a function of arguments.

- inserting into an integer into a sorted linear list takes time $O(n)$, for list storing $n$ integers
- inserting into an integer into a balanced binary tree of $n$ integers takes time $O(\log n)$, for tree storing $n$ integers

Let us review these techniques by applying them to our sum segment example

# Asymptotic analysis of sequential running time

Find time bound on sequential `sumSegment` as a function of `s` and `t`

```
def sumSegment(a: Array[Int], p: Double, s: Int, t: Int): Int = {
  var i= s; var sum: Int = 0
  while (i < t) {
    sum= sum + power(a(i), p)
    i= i + 1
  }
  sum }
```

# Asymptotic analysis of sequential running time

Find time bound on sequential `sumSegment` as a function of `s` and `t`

```
def sumSegment(a: Array[Int], p: Double, s: Int, t: Int): Int = {
  var i= s; var sum: Int = 0
  while (i < t) {
    sum= sum + power(a(i), p)
    i= i + 1
  }
  sum }
```

The answer is: $W(s,t) = O(t-s)$, a function of the form: $c_1(t-s)+c_2$

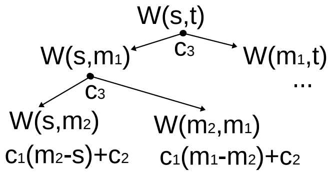
- $t-s$ loop iterations
- a constant amount of work in each iteration

# Analysis of recursive functions

```
def segmentRec(a: Array[Int], p: Double, s: Int, t: Int) = {
  if (t - s < threshold)
    sumSegment(a, p, s, t)
  else {
    val m= s + (t - s)/2
    val (sum1, sum2)= (segmentRec(a, p, s, m),
                       segmentRec(a, p, m, t))
    sum1 + sum2 } }
```

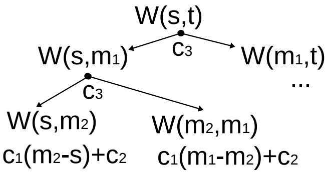
# Analysis of recursive functions

```
def segmentRec(a: Array[Int], p: Double, s: Int, t: Int) = {
  if (t - s < threshold)
    sumSegment(a, p, s, t)
  else {
    val m= s + (t - s)/2
    val (sum1, sum2)= (segmentRec(a, p, s, m),
                       segmentRec(a, p, m, t))
    sum1 + sum2 } }
```



# Analysis of recursive functions

```scala
def segmentRec(a: Array[Int], p: Double, s: Int, t: Int) = {
  if (t - s < threshold)
    sumSegment(a, p, s, t)
  else {
    val m= s + (t - s)/2
    val (sum1, sum2)= (segmentRec(a, p, s, m),
                       segmentRec(a, p, m, t))
    sum1 + sum2 } }
```




$$W(s,t) = \begin{cases} c_1(t-s)+c_2, & \text{if } t-s < \text{threshold} \\ W(s,m)+W(m,t)+c_3 & \text{otherwise, for } m = \lfloor (s+t)/2 \rfloor \end{cases}$$

# Bounding solution of recurrence equation

$$W(s,t) = \begin{cases} c_1(t-s) + c_2, & \text{if } t - s < \text{threshold} \\ W(s,m) + W(m,t) + c_3 & \text{otherwise, for } m = \lfloor (s+t)/2 \rfloor \end{cases}$$

Assume $t - s = 2^N(\mathit{threshold} - 1)$, where $N$ is the depth of the tree

Computation tree has $2^N$ leaves and $2^N - 1$ internal nodes

$W(s,t) = 2^N(c_1(\mathit{threshold} - 1) + c_2) + (2^N - 1)c_3 = 2^N c_4 + c_5$

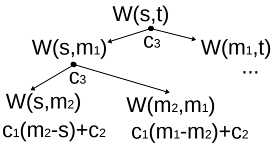
If $\underline{2^{N-1} < (t-s)/(\mathit{threshold} - 1)} \leq 2^N$, we have

$$W(s,t) \leq 2^N c_4 + c_5 < (t-s) \cdot 2/(\mathit{threshold} - 1) + c_5$$

$W(s,t)$ is in $O(t-s)$. Sequential `segmentRec` is linear in $t - s$

# Recursive functions with unbounded parallelism

```
def segmentRec(a: Array[Int], p: Double, s: Int, t: Int) = {
 if (t - s < threshold)
   sumSegment(a, p, s, t)
 else {
   val m = s + (t - s)/2
   val (sum1, sum2)= parallel(segmentRec(a, p, s, m),
                              segmentRec(a, p, m, t))
   sum1 + sum2 } }
```




$$D(s,t) = \begin{cases} c_1(t-s) + c_2, & \text{if } t - s < \text{threshold} \\ \max(D(s,m), D(m,t)) + c_3 & \text{otherwise, for } m = \lfloor (s+t)/2 \rfloor \end{cases}$$

# Solving recurrence with unbounded parallelism

$$D(s,t) = \begin{cases} c_1(t-s)+c_2, & \text{if } t-s < \text{threshold} \\ \max(D(s,m), D(m,t)) + c_3 & \text{otherwise, for } m = \lfloor (s+t)/2 \rfloor \end{cases}$$

Assume $t - s = 2^N(\mathit{threshold} - 1)$, where $N$ is the depth of the tree

Computation tree has $2^N$ leaves and $2^N - 1$ internal nodes

The value of $D(s,t)$ in leaves of computation tree: $c_1(\mathit{threshold} - 1) + c_2$

One level above: $c_1(\mathit{threshold} - 1) + c_2 + c_3$

Root: $c_1(\mathit{threshold} - 1) + c_2 + (N-1)c_3$

Solution bounded by $O(N)$. Also, running time is monotonic in $t - s$

If $2^{N-1} < (t-s)/(\mathit{threshold} - 1) \leq 2^N$, we have $N < \log(t-s) + c_6$

$D(s,t)$ is in $O(\log(t-s))$

# Work and depth

We would like to speak about the asymptotic complexity of parallel code

- but this depends on available parallel resources
- we introduce *two measures* for a program

Work $W(e)$: number of steps $e$ would take if there was no parallelism

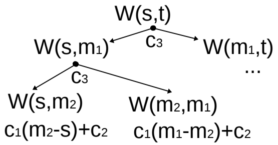
- this is simply the sequential execution time
- treat all `parallel(e1,e2)` as `(e1,e2)`

Depth $D(e)$: number of steps if we had unbounded parallelism

- we take maximum of running times for arguments of `parallel`

# Rules for depth (span) and work

Key rules are:




- $W(parallel(e_1, e_2)) = W(e_1) + W(e_2) + c_2$
- $D(parallel(e_1, e_2)) = \max(D(e_1), D(e_2)) + c_1$

If we divide work in equal parts, for depth it counts only once!

For parts of code where we do not use `parallel` explicitly, we must add up costs. For function call or operation $f(e_1, ..., e_n)$:

- $W(f(e_1, ..., e_n)) = W(e_1) + ... + W(e_n) + W(f)(v_1, ..., v_n)$
- $D(f(e_1, ..., e_n)) = D(e_1) + ... + D(e_n) + D(f)(v_1, ..., v_n)$

Here $v_i$ denotes values of $e_i$. If $f$ is primitive operation on integers, then $W(f)$ and $D(f)$ are constant functions, regardless of $v_i$.

Note: we assume (reasonably) that constants are such that $D \leq W$

# Computing time bound for given parallelism

Suppose we know $W(e)$ and $D(e)$ and our platform has $P$ parallel threads

Regardless of $P$, cannot finish sooner than $D(e)$ because of dependencies

Regardless of $D(e)$, cannot finish sooner than $W(e)/P$: every piece of work needs to be done

So it is reasonable to use this estimate for running time:

$$D(e) + \frac{W(e)}{P}$$

Given $W$ and $D$, we can estimate how programs behave for different $P$

- If $P$ is constant but inputs grow, parallel programs have same asymptotic time complexity as sequential ones
- Even if we have infinite resources, $(P \to \infty)$, we have non-zero complexity given by $D(e)$

# Consequences for segmentRec

The call to parallel function `segmentRec` had:

- work $W$: $O(t - s)$
- depth $D$: $O(\log(t - s))$

On a platform with $P$ parallel threads the running time is, for some constants $b_1, b_2, b_3, b_4$:

$$b_1 \log(t - s) + b_2 + \frac{b_3(t - s) + b_4}{P}$$

- if $P$ is bounded, we have linear behavior in $t - s$
  - possibly faster than sequential, depending on constants
- if $P$ grows, the depth starts to dominate the cost and the running time becomes logarithmic in $t - s$

# Parallelism and Amdahl's Law

Suppose that we have two parts of a sequential computation:

- part1 takes fraction $f$ of the computation time (e.g. 40%)
- part2 take the remaining $1 - f$ fraction of time (e.g. 60%) and we can speed it up

If we make part2 $P$ times faster the speedup is

$$1/\left(f + \frac{1-f}{P}\right)$$

For $P = 100$ and $f = 0.4$ we obtain $2.46$

Even if we speed the second part infinitely, we can obtain at most $1/0.4 = 2.5$ speed up.