In [3]:
```
# setup
from IPython.core.display import display,HTML
display(HTML('<style>.prompt{width: 0px; min-width: 0px; visibility: collapse}</style>'))
display(HTML(open('rise.css').read()))

# imports
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
sns.set(style="whitegrid", font_scale=1.5, rc={'figure.figsize':(12, 6)})
```

CMPS 2200

Introduction to Algorithms

Cost models

Today's agenda:

- Three different ways to model computational cost
  - Random Access Machine
  - Parallel Random Access Machine
  - Language Based Models

Recall first lecture:

In [ ]:
```
def linear_search(mylist, key):          #   cost          number of times run
    for i,v in enumerate(mylist):        #   c1                 n
        if v == key:                     #   c2                 n
            return i                     #   c3                 0
    return -1                            #   c4                 1
```

$\text{Cost}(\text{linear-search}, n) = c_1 n + c_2 n + c_4 = O(n)$

machine-based cost model

- define the cost of each instruction
- runtime is sum of costs of each instruction

Random Access Machine model

- "Simple" operations (+, *, =, if) take exactly one time step
  - no matter the size of operands
- loops consist of many single-step operations
- **memory access takes one step**
  - unbounded memory
  - each cell holds integers of unbounded size
- assumes sequential execution

- one input tape, one output tape

Hypothetically, we could compute the run time of an algorithm:

1. compute the number of steps
2. determine the number of steps per second our machine can perform
3. time = steps / steps per second

All models make incorrect assumptions. What are some that the RAM model makes?

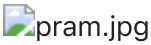
- multiplication does not take the same time as addition
- cache is faster than RAM

Not terrible assumptions since we are interested in asymptotic analysis.

E.g., if cache lookup is half the time of RAM lookup, then we're "only" off by a factor of 2.
E.g. $2n \in O(n)$

PRAM: Parallel Random Access Machine

- The RAM model extended to have
  - multiple processors $P_0, P_1, P_2 \ldots$
  - unbounded, **shared** memory cells $M[0], M[1], M[2] \ldots$
  - any processor P_i can access any memory cell $M[j]$ in one time step


pram.jpg
source

Assumes some control mechanisms to deal with race conditions/synchronization.

To compute the runtime:

- compute time for the slowest processor

Two drawbacks of this model:

1. Mapping data to each processor can be tricky
2. Nested parallelism is hard to specify in this model (e.g., recursive fork-join)

Language Based Cost Models

- Define a language to specify algorithms
- Assign a cost to each expression
- Cost of algorithm is sum of costs for each expression

Work-Span model

- We can define this model for SPARC

Recall our definitions of **work** and **span**

- **work**: total number of primitive operations performed by an algorithm
- **span**: longest sequence of dependencies in computation
  - time to run with an infinite number of processors
  - measure of how "parallelized" an algorithm is
  - also called: *critical path length* or *computational depth*

intuition:
work: total energy consumed by a computation
span: minimum possible time that the computation requires

work: T_1 = time using one processor
span: T_∞ = time using ∞ processors

For a given SPARC expression e, we will analyze the work $W(e)$ and span $S(e)$

Composition

- (e_1, e_2): Sequential Composition
  - Add work and span
- $(e_1||e_2)$: Parallel Composition
  - Add work but **take the maximum span**

Rules of composition

$\mathbf{e}$	$\mathbf{W(e)}$	$\mathbf{S(e)}$
v	1	1
`lambda` p `.` e	1	1
(e_1, e_2)	$1 + W(e_1) + W(e_2)$	$1 + S(e_1) + S(e_2)$
$(e_1 \vert e_2)$	$1 + W(e_1) + W(e_2)$	$1 + \max(S(e_1), S(e_2))$
$(e_1 e_2)$	$W(e_1) + W(e_2) + W([\text{Eval}(e_2)/x]e_3) + 1$	$S(e_1) + S(e_2) + S([\text{Eval}(e_2)/x]e_3) + 1$
`let val` $x = e_1$ `in` e_2 `end`	$1 + W(e_1) + W([\text{Eval}(e_1)/x]e_2)$	$1 + S(e_1) + S([\text{Eval}(e_1)/x]e_2)$
$\{f(x) \mid x \in A\}$	$1 + \sum_{x \in A} W(f(x))$	$1 + \max_{x \in A} S(f(x))$

parallel composition: $e_1 || e_2$

$W(e_1||e_2) = 1 + W(e_1) + W(e_2)$
$S(e_1||e_2) = 1 + \max(S(e_1), S(e_2))$

What are we assuming here?

In a *pure* functional language, we can run two functions in parallel if there is no explicit sequencing.

- no side effects
- data persistence

function application: $e_1 e_2$

$W(e_1 e_2) = W(e_1) + W(e_2) + W([\text{Eval}(e_2)/x]e_3) + 1$

- $\text{Eval}(e)$ evaluates e and returns resulting value
- $[v/x]e$: replace all free (unbound) occurrences of x in e with value v
  - e.g., $[10/x](x^2 + 10) \rightarrow 110$

$e_1 e_2$

E.g., if e_1 is $f(x)$, we

1. evaluate e_2 to a value v
2. substitute v for x in $f(x)$
3. run $f(v)$

We assume $\text{Eval}(e_2) = \text{lambda}\ x\ .\ e_3$

$[\text{Eval}(e_2)/x]e_3$: all free occurrences of x in e_3 are replaced with $\text{Eval}(e_2)$

Example

$f(x) = \texttt{lambda}\ x\ .\ x + 1$

$W(f(x)(1)) =$

$W(f(x)) + W(1) + W([1/x](x + 1)) + 1 =$ (by definition of $W(e_1 e_2)$)

$W(f(x)) + W(1) + (W(1) + W(x + 1) + 1) + 1 =$ (by definition of $W(e_1 e_2)$)

$1 + 1 + 1 + 1 + 1 =$

5

$S(f(x)(1)) = 5$

serial composition:

$W(f(x)(1), f(x)(2)) =$

$W(f(x)(1)) + W(f(x)(2)) + 1 =$

$5 + 5 + 1 = 11$

$S(f(x)(1), f(x)(2)) =$

$S(f(x)(1)) + S(f(x)(2)) + 1 =$

$5 + 5 + 1 = 11$

parallel composition:

$W(f(x)(1)||f(x)(2)) =$

$W(f(x)(1)) + W(f(x)(2)) + 1 =$

$5 + 5 + 1 = 11$

$S(f(x)(1)||f(x)(2)) =$

$\max(S(f(x)(1)), S(f(x)(2))) + 1 =$

$\max(5, 5) + 1 = 6$

In [9]:
```
e1 = lambda x: x**2 + 10
e2 = lambda y: 5*y

e1(
    e2(2)  # evaluate e2 to a value v
  )        # substitute v for x in e1
           # return result of e1
```

Out[9]:
```
110
```

let ... in

$W(\texttt{ let val } x = e_1 \texttt{ in } e_2 \texttt{ end }) = 1 + W(e_1) + W([\text{Eval}(e_1)/x]e_2)$

Expression e_2 is applied using the bindings defined inside **let**.

- $[\text{Eval}(e_1)/x]e_2$: all free occurrences of x in e_2 are replaced with $\text{Eval}(e_1)$
- $W([\text{Eval}(e_1)/x]e_2)$ accounts for this cost.

let
  $x = 10 * 5 \quad (e_1)$
in
  $x + 100 \quad (e_2)$
end

returns 150

$W(\texttt{ let val } x = e_1 \texttt{ in } e_2 \texttt{ end }) =$

$1 + W(e_1) + W([\text{Eval}(e_1)/x]e_2) =$

$1 + W(e_1) + (W(\text{sub in 50 for x}) + W(x + 100) + 1)$ (by function application rule)

$1 + 1 + 1 + 1 + 1$

$= 5$

Parallelism

how many processors can we use efficiently?

average parallelism:

$$\overline{P} = \frac{W}{S}$$

To increase parallelism, we can either:

- decrease span
- increasing work (but that's not really desireable, since we want the overall cost to be low)

work efficiency: a parallel algorithm is *work efficient* if it performs asymptotically the same work as the best known sequential algorithm for the problem.

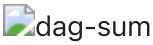
So, we want a *work efficient* parallel algorithm with low span.

Scheduling

Key issue of parallel algorithms is **scheduling**: which processor will run which task when?

- typically have more tasks than processors.

Recall our parallel sum method:



source

We must decide when to run each part of the sum. There are dependencies that constrain the order.

Scheduler

- For each task generated by a parallel algorithm, assign it to an available processor
- Goal: minimize execution time.

Greedy Scheduler

Whenever there is a processor available and a task ready to execute, assign the task to the processor and start it immediately.

Why might this not be optimal?

Greedy schedulers have an important property that is summarized by the greedy scheduling principle.

Assuming P processors, then the time T_P to perform computation with work W and span S is bounded by:

$T_P < \frac{W}{P} + S$

Because we know:

- $T_P \geq \frac{W}{P}$, since that would be the optimal division of work to processors
- $T_P \geq S$, by the definition of span

we can conclude that the best we can hope for is:

$T_P \geq \max(\frac{W}{P}, S)$

Therefore the time using a greedy scheduler is bounded by:

$\max(\frac{W}{P}, S) \leq T_P < \frac{W}{P} + S$

How good is greedy? How close is $(\frac{W}{P} + S)$ to $\max(\frac{W}{P}, S)$?

actually pretty close.

$\frac{W}{P} + S \leq 2 * \max(\frac{W}{P}, S)$

(why? consider what the worst possible span is...)

Greedy scheduler gets better the more parallelism is possible in the algorithm.

Recall average parallelism: $\overline{P} = \frac{W}{S}$

We can rewrite:

$$
\begin{aligned}
T_P &< \frac{W}{P} + S \\
&= \frac{W}{P} + \frac{W}{\overline{P}} \\
&= \frac{W}{P}(1 + \frac{P}{\overline{P}})
\end{aligned}
$$

So, the greater $\overline{P}$ is than P, the closer to optimal we get.

E.g., recall our parallel sum method, which has

$W = O(n)$
$S = O(\lg n)$

$\overline{P} = \frac{W}{S} = \frac{O(n)}{O(\lg n)}$

$\max(\frac{W}{P}, S) \leq T_P < \frac{W}{P} + S$

so, if we have 2 processors:

$$\max(\frac{O(n)}{2}, O(\lg n)) \leq T_2 < \frac{O(n)}{2} + O(\lg n)$$

$$\frac{O(n)}{2} \leq T_2 < \frac{O(n)}{2} + O(\lg n)$$

if we have $\lg n$ processors:

$$\max(\frac{O(n)}{\lg n}, O(\lg n)) \leq T_{\lg n} < \frac{O(n)}{\lg n} + O(\lg n)$$

$$\frac{O(n)}{\lg n} \leq T_{\lg n} < \frac{O(n)}{\lg n} + O(\lg n)$$

The advantage of the Work-Span model:

- We can design parallel algorithms without worrying about scheduling details.
- We are ignoring some overhead in the creation of the schedule itself.
    - This is acceptable since we are focused on asymptotics (just as in RAM model)