Brent's Theorem

- Assumption when formulating parallel algorithms: we have arbitrarily many processors
  - E.g., $O(n)$ many processors for input of size n
  - Kernel launch even reflects that!
    - Often, we run as many threads as there are input elements
    - I.e., CUDA/GPU provide us with this (nice) abstraction
- Real hardware: only has fixed number p of processors
  - E.g., on current GPUs: $p \approx 200–2000$ (depending on viewpoint)
- Question: how fast can an implementation of a massively parallel algorithm really be?

- Assumptions for Brent's theorem: PRAM model
  - No explicit synchronization needed
  - Memory access = free

- Brent's Theorem:
  Given a massively parallel algorithm *A*; let *D*(*n*) = its depth (i.e., parallel time complexity), and *W*(*n*) = its work complexity.
  Then, *A* can be run on a *p*-processor PRAM in time

$$T(n, p) \leq \left\lfloor \frac{W(n)}{p} \right\rfloor + D(n)$$

(Note the "≤")

- Proof:
  - For each iteration step i, $1 \leq i \leq D(n)$, let $W_i(n)$ = number of operations in that step
  - Distribute those operations on p processors:
    - Groups of $\left\lceil \frac{W_i(n)}{p} \right\rceil$ operations in parallel on the p processors
    - Takes $\left\lceil \frac{W_i(n)}{p} \right\rceil$ time steps on the PRAM
  - Overall :

$$T(n,p) = \sum_{i=1}^{D(n)} \left\lceil \frac{W_i(n)}{p} \right\rceil \leq \sum_{i=1}^{D(n)} \left(\left\lfloor \frac{W_i(n)}{p} \right\rfloor + 1 \right) \leq \left\lfloor \frac{W(n)}{p} \right\rfloor + D(n)$$

Application of Brent's Theorem to our Optimization of Prefix-Sum

- Assume that the optimized version loads f floats into local registers
- Work complexity:
  - Without optimization: $W_1(n) = 2n$
  - With optimization: $W_2(n) = 2\frac{n}{f} + \frac{n}{f} \cdot f = n\left(1 + \frac{2}{f}\right)$
- Depth complexity:
  - Without optimization: $D_1(n) = 2\log(n)$
  - With optimization: $D_2(n) = 2\log(\frac{n}{f}) + f = 2\log n - 2\log f + f$
- If $f = 2$, then $W_2 = W_1$ and $D_2 = D_1$, i.e., we gain nothing
- If $f > 2$, speedup of version 2 (opt.) over version 1 (original):

$$\text{Speedup}(n) = \frac{T_2(n)}{T_1(n)} = \frac{\frac{W_1(n)}{p} + D_1(n)}{\frac{W_2(n)}{p} + D_2(n)} \approx \frac{2\frac{n}{p}}{\frac{n}{p}\left(1 + \frac{2}{f}\right)} = \frac{2f}{f+2}$$

Other Consequences of Brent's Theorem

- Obviously, $\text{Speedup}(n) \leq p$
- In the sequential world, time = work: $T_S(n) = W_S(n)$
- In the parallel world: $T_P(n) = \frac{W_P(n)}{p} + D(n)$
- Our speedup is $\text{Speedup}(n) = \frac{T_S(n)}{T_P(n)} = \frac{W_S(n)}{\frac{W_P(n)}{p} + D(n)}$
- Assume, $W_P(n) \in \Omega(\, W_S(n)\,)$

  i.e., our parallel algorithm would do asymptotically more work
- Then, $$\text{Speedup}(n) = \frac{W_S(n)}{\Omega(\, W_S(n)\,) + D(n)} \rightarrow 0 \quad \text{as } n \rightarrow \infty$$

  because, on real hardware, p is bounded
- This is the reason why we want work-efficient parallel algorithms!

- Now, look at work-efficient parallel algorithms, i.e.

$$W_P(n) \in \Theta(\, W_S(n)\,)$$

- Then,

$$\text{Speedup}(n) = \frac{W(n)}{\frac{W(n)}{p} + D(n)} = \frac{pW(n)}{W(n) + pD(n)}$$

- In this situation, we will achieve the optimal speedup of p, so long as

$$p \in O(\frac{W(n)}{D(n)})$$

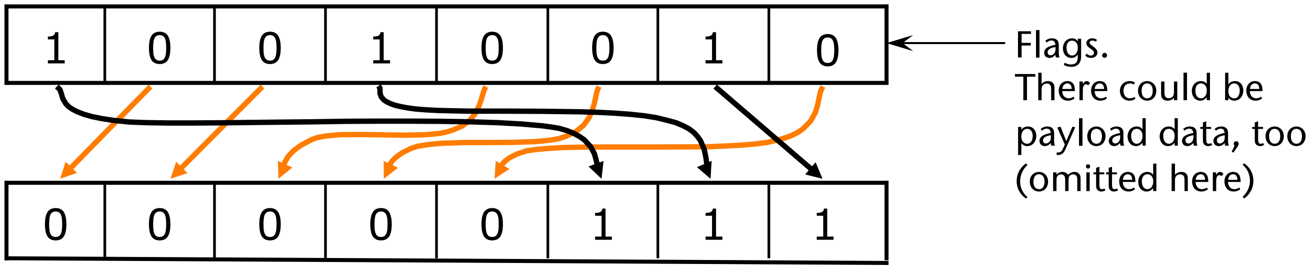
- Consequence: given two work-efficient parallel algorithms, the one with the smaller depth complexity is better, because we can run it on hardware with more processors (cores) and still obtain a speedup of p over the sequential algorithm (in theory). We say this algorithm scales better.

Limitations of Brent's Theorem

- Brent's theorem is based on the PRAM model
- That model makes a number of unrealistic assumption:
  - Memory access has zero latency
  - Memory bandwidth is infinite
  - No synchronization among processors (threads) is necessary
  - Arithmetic operations cost unit time
- With current hardware, rather the opposite is realistic

Radix Sort, Based on the Split Operation

- The split operation: rearrange elements according to a flag



  - Note: split maintains order within each group! (i.e., it is stable)

- Radix sort (massively parallel):

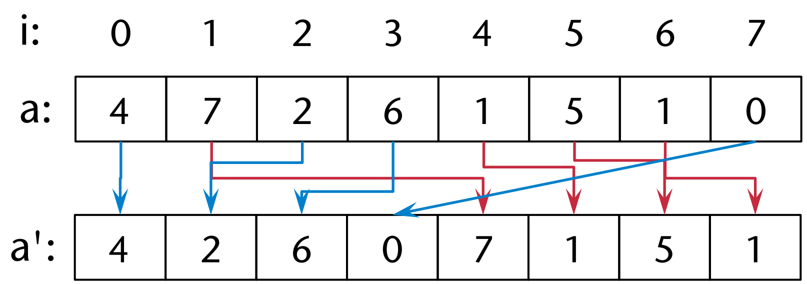
```
radix_sort( array a, int len ):
  for i = 0...n-1:  // important: go from low to high bit!
    split(i, a)     // split a, based on bit i of keys
```

where **`split(i,a)`** rearranges **`a`** by moving all keys that have bit **`i`** = 0 to the bottom, all keys that have bit **`i`** = 1 to the top (lowest bit = bit no. 0)

  - Reminder: stability of *split* is essential!

Algorithm for the Split Operation

- Split's job:
  - Determine new index for each element
  - Then perform the permutation



- Algorithm (by way of an example):
  - Consider lowest bit of the keys
  1. Compute "0"-scan (exclusive):
     f_i = # "0"s in (a_0, ..., a_{i-1})
  2. Set F = total number of "0"s
     $$= \begin{cases} f_{n-1} + 1 & a_{n-1} = 0 \\ f_{n-1} & a_{n-1} = 1 \end{cases}$$
  3. If $a_i = 0 \rightarrow$ new pos. $d = f_i$
  4. If $a_i = 1 \rightarrow$ new pos. $d = F + (i - f_i)$
     - Because $i - f_i$ = # "1"s to the left of i

i:	0	1	2	3	4	5	6	7
a:	100	111	010	110	001	101	001	000
f:	0	1	1	2	3	3	3	3
F=4								
$d_{\text{for "0"s}}$:	0		1	2				3
$d_{\text{for "1"s}}$:		4+(1-1)			4+(4-3)	4+(5-3)	4+(6-3)	
d:	0	4	1	2	5	6	7	3

- A conceptual algorithm for the "0"-scan:
  - Extract the relevant bit (conceptually only)
  - Invert the bit
  - Compute regular scan with +-operation

a:	100	111	010	110	001	101	001	000
a':	1	0	1	1	0	0	0	1
f:	0	1	1	2	3	3	3	3

- In a real implementation, you would, of course, implement this as a native "0"-scan routine!