Parallel Programming Patterns

Dmitri Nesteruk
@dnesteruk

pluralsight
hardcore developer training

Rules of the Game

- **Different types of memory**
  - Shared vs. private
  - Access speeds
- **Data is in arrays**
  - No parallel data structures
  - No other data structures
  - No auto-parallelization/vectorization compiler support
  - No CPU-type SIMD equivalent
- **Compiler constraint**
  - No C++11 support

Overview

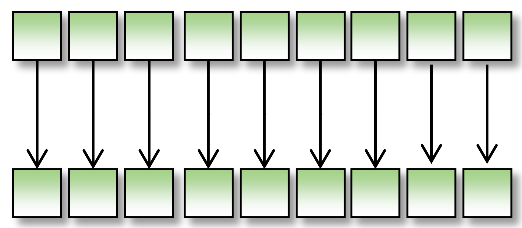
- Data Access
- Map
- Gather
- Reduce
- Scan

Data Access

- **A real problem!**
- **Thread space can be up to 6D**
  - 3D grid of 3D thread blocks
- **Input space typically 1D**
  - 2D arrays are possible
- **Need to map threads to inputs**
- **Some examples**
  - 1 block, N threads → threadIdx.x
  - 1 block, MxN threads → threadIdx.y * blockDim.x + threadIdx.x
  - N blocks, M threads → blockIdx.x * gridDim.x + threadIdx.x
  - … and so on

Map

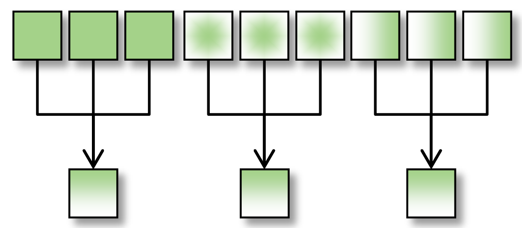
- Applying a function to an array and replicating that function over every element in the array



- $x_i = f(x_i)$

Gather

- Applying a function to an arbitrary selection of input values to get an output value



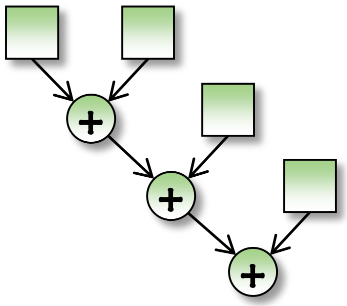
- $y_i = f(a_m, b_n, c_p, \dots)$

Black Scholes Formula

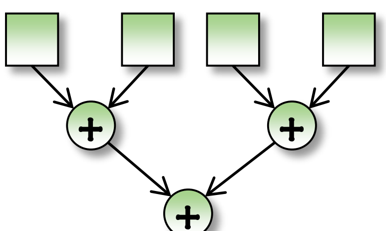
- **An *option* is a right to buy (call) or sell (put) an asset at a specific price and date**
- **The *theoretical price* of an option depends on**
  - K – the price at which an asset can be bought or sold (a.k.a. *strike*)
  - S – the price of the underlying asset
  - t – the time, in years, until the option expires
  - r – the risk-free rate; rate at which money can be borrowed
  - σ – the *volatility* of the option (a measure of how much the price jumps)
- **Black-Scholes formula:**
  - $C = N(d_1)S - N(d_2)Ke^{-rt}$ and $P = Ke^{-rt} - S + C$
  - $d_1 = \frac{\ln(S/K) + \left(r + \frac{\sigma^2}{2}\right)t}{\sigma\sqrt{t}}$
  - $d_2 = d_1 - \sigma\sqrt{t}$
  - $N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2}\, dt = \frac{1}{2}\left[1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$

Reduce

- Consider a calculation of $\sum_{i=1}^{n} x_i$
- Can be expressed as $\left((x_1 + x_2) + x_3\right) + \cdots$

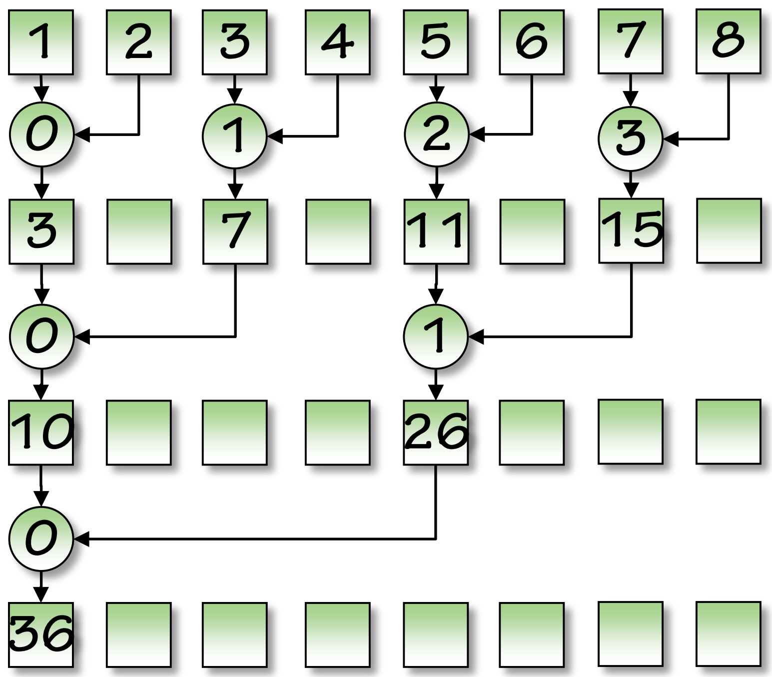


- Since + is associative, we can subdivide problem space:
  - $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n/2} x_i + \sum_{i=n/2}^{n} x_i = \cdots$



Reduce in Practice

- Adding up N data elements



- Adding up N data elements
- Use 1 block of N/2 threads
- Each thread does
  `x[i] += x[j];`
- At each step
  - # of threads halved
  - Distance (j-i) doubled
- `x[0]` is the result

Scan

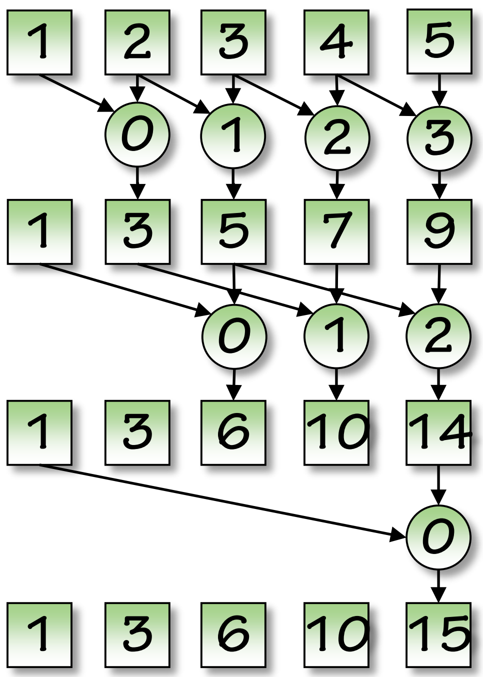
- Each output value y_n is calculated as a function involving inputs from 1 to n, i.e.
  - $y_n = f(x_1, x_2, \dots, x_n)$
- E.g. a running sum of elements

4	2	5	3	6
4	6	11	14	20

- Looks sequential (just like Reduce)

Scan in Practice

- Similar to reduce



- Require N-1 threads
- Step size keeps doubling
- Number of threads reduced by step size
- Each thread n does
  x[n+step] += x[n];