Australian National University

Data Parallelism

P01

Vectorization - Haskell

```haskell
type Real_Precision = Float
type Scalar = Real_Precision
type Vector = [Real_Precision]
scale :: Scalar -> Vector -> Vector
scale scalar vector = map (scalar *) vector
```


Potentially concurrent, yet executed sequentially

Vectorization - Haskell

```haskell
import Control.Parallel.Strategies
type Real_Precision = Float
type Scalar = Real_Precision
type Vector = [Real_Precision]
scale :: Scalar -> Vector -> Vector
scale scalar vector = parMap rpar (scalar *) vector
```


Executed in parallel (may be faster or slower than sequential execution)

Vectorization - Ada

```
type Real    is digits 15;
type Vectors is array (Positive range <>) of Real;
function Scale (Scalar : Real; Vector : Vectors) return
Vectors is
  Scaled_Vector : Vectors (Vector'Range);
begin
  for i in Vector'Range loop
    Scaled_Vector (i) := Scalar * Vector (i);
  end loop;
return Scaled_Vector;
end Scale;
```

Vectorization - Ada

```
…
   for i in Vector'Range loop
      Scaled_Vector (i) := Scalar * Vector (i);
   end loop;
…
```


- Ada compiler translates into CPU-level vector operations (AltiVec, SPE, MMX, SSE, NEON, SPU, AVX, …)
- Combined with inlining, loop unrolling, and caching, this is as fast as a single CPU will get

Vectorization – Ada 202x

```
…
  parallel for i in Vector'Range loop
    Scaled_Vector (i) := Scalar * Vector (i);
  end loop;
…
```


- generates 'tasklets' for independent execution of each iteration

Ada 202x https://www.adaic.org/advantages/ada-202x/

Vectorization - Chapel

```
const dom = {1 .. 100000000},
      vector: [dom] real = 2.718,
      scale: real = 3.14;
const scaled: [dom] real = scale * vector;
```


Function is "promoted" into data-parallel operation over all indices in `Dom` results in:

- CPU-level vector operations
- multi-core parallelism
- distributed parallelism

Reduction - Haskell

```haskell
type Real_Precision = Float
type Vector = [Real_Precision]
equal :: Vector -> Vector -> Bool
equal v_1 v_2 = foldr (&&) True $ zipWith (==) v_1 v_2
```


Potentially concurrent, yet executed (lazy) sequentially

Reduction - Haskell

```
type Real_Precision = Float
type Vector = [Real_Precision]
equal :: Vector -> Vector -> Bool
equal = (==)
```


Potentially concurrent, yet executed (lazy) sequentially

Reduction - Ada

```
type Real     is digits 15;
type Vectors  is array (Positive range <>) of Real;
function "=" (Vector_1, Vector_2 : Vectors) return Boolean is
  (for all i in Vector_1'Range =>
     Vector_1 (i) = Vector_2 (i));
```


translates into CPU-level vector operations

∧-chain is evaluated lazy sequentially

Reduction - Chapel

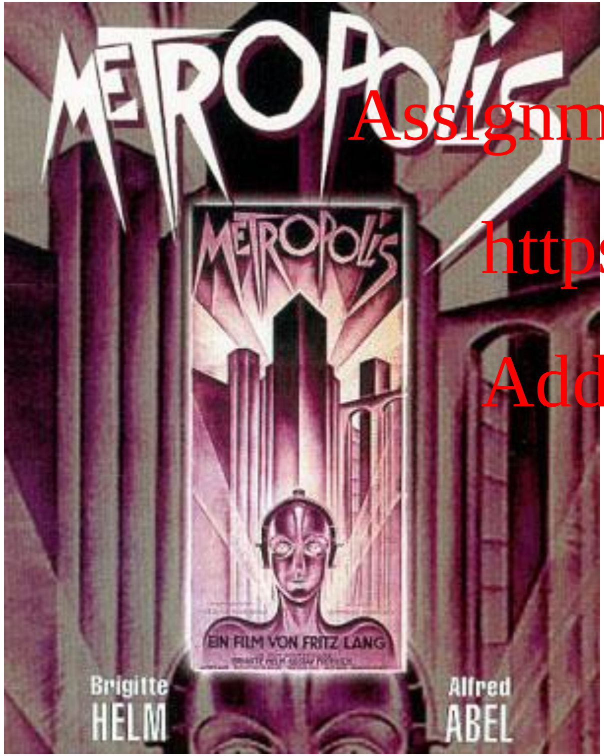
```
const dom = {1 .. 100000000},
      vector1, vector2: [dom] real = 1.0;

proc equal(v1, v2) : bool {
  return && reduce (v1 == v2);
}
```


`==` function is "promoted" (multi-level parallelism)

∧-chain is evaluated by concurrent divide-and-conquer (binary tree)

General Data Parallelism - Chapel



Sharpen →

General Data Parallelism - Chapel

```
const mask: [1..3, 1..3] real
  = ((0, -1, 0), (-1, 5, -1), (0, -1, 0));
proc unsharpMask(P, (i, j): index(image)) : real {
  return + reduce (mask * P [i - 1..i + 1, j - 1..j + 1]);
}
const sharpenedPicture = forall px in imageDom do
  unsharpMask(picture, px);
```


Translated to multi-level parallelism (vector, multi-core, distributed)

General Data Parallelism – Game of Life

- Cellular automaton transitions from a state S into the next state S':

  $$S \rightarrow S' \leftrightarrow \forall c \in S: c \rightarrow c' = r(S,c)$$

  i.e. all cells of a state transition concurrently into new cells by following a rule r.