Systems, Networks & Concurrency 2020

5

Data Parallelism

Uwe R. Zimmer - The Australian National University

Data Parallelism

References

[Bacon98]
J. Bacon
Concurrent Systems
1998 (2nd Edition) Addison Wesley Longman Ltd, ISBN 0-201-17767-6

[Ada 2012 Language Reference Manual]
see course pages or http://www.ada-auth.org/standards/ada12.html

[Chapel 1.13 Language Specification Version 0.981]
see course pages or
http://chapel.cray.com/docs/latest/_downloads/chapelLanguageSpec.pdf
released on 7. April 2016

Vector Machines

Vectorization

```
type Real_Precision = Float

type Scalar = Real_Precision
type Vector = [Real_Precision]

scale :: Scalar -> Vector -> Vector
scale scalar vector = map (scalar *) vector
```

Vector Machines

Vectorization

Potentially concurrent, yet:

Executed sequentially.

```
type Real_Precision = Float

type Scalar = Real_Precision
type Vector = [Real_Precision]

scale :: Scalar -> Vector -> Vector
scale scalar vector = map (scalar *) vector
```

Vector Machines

Vectorization

```
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.

This **may be faster or slower** than a sequential execution.

Vector Machines

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;
```

Data Parallelism

Vector Machines

Vectorization

Buzzword collection:
AltiVec, SPE, MMX, SSE,
NEON, SPU, AVX, …

Translates into
CPU-level vector operations

```
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;
```

Combined with
in-lining, **loop unrolling** and **caching**
this is as fast as a single CPU will get.

Vector Machines

Vectorization

Function is “promoted”

```
const Index = {1 .. 100000000},
      Vector  : [Index] real = 1.0,
      Scale   : real = 5.1,
      Scaled  : [Vector] real = Scale * Vector;
```

Vector Machines

Vectorization

Function is “promoted”

```
const Index = {1 .. 100000000},
      Vector   : [Index] real = 1.0,
      Scale    : real = 5.1,
      Scaled   : [Vector] real = Scale * Vector;
```

Translates into **CPU-level vector operations** *as well as* **multi-core** or **fully distributed operations**

Vector Machines

Reduction

```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
```

Vector Machines

Reduction

```
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.

Vector Machines

Reduction

```
type Real_Precision = Float

type Vector = [Real_Precision]

equal :: Vector -> Vector -> Bool
equal = (==)
```

Potentially concurrent, yet:

Executed lazy sequentially.

Data Parallelism

Vector Machines

Reduction

```
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));
```

Data Parallelism

Vector Machines

Reduction

```
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

$\wedge$-chain is evaluated lazy sequentially.

Vector Machines

Reduction

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

Infinite recursion

Translates into **CPU-level vector operations**

∧-chain is evaluated lazy sequentially.

Vector Machines

Reduction

```
type Real is digits 15;
type Vectors is array (Positive range <>) of Real;
function Equal (Vector_1, Vector_2 : Vectors) return Boolean is (Vector_1 = Vector_2);
```

Translates into
CPU-level vector operations

∧-chain is evaluated lazy sequentially.

Vector Machines

Reduction

```
type Real is digits 15;
type Vectors is array (Positive range <>) of Real;
function Equal (Vector_1, Vector_2 : Vectors) return Boolean renames ”=”;
```

Translates into
CPU-level vector operations

∧-chain is evaluated lazy sequentially.

Vector Machines

Reduction

$$v_1 = v_2 \Leftrightarrow \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \Leftrightarrow (x_1 = x_2) \wedge (y_1 = y_2) \wedge (z_1 = z_2)$$

```
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.

Vector Machines

Reduction

```
const Index = {1 .. 100000000},
      Vector_1, Vector_2 : [Index] real = 1.0;
proc Equal (v1, v2) : bool
   {return && reduce (v1 == v2);}
```

Function is **“promoted”**

Vector Machines

Reduction

```
const Index = {1 .. 100000000},
      Vector_1, Vector_2 : [Index] real = 1.0;
proc Equal (v1, v2) : bool
   {return && reduce (v1 == v2);}
```

Function is **“promoted”**

$\wedge$-operations are evaluated in a **concurrent divide-and-conquer** (binary tree) structure.

Translates into **CPU-level vector operations** *as well as* **multi-core** or **fully distributed operations**

Vector Machines

Reduction

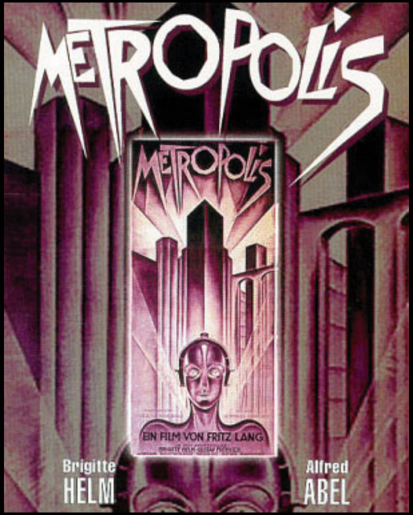
```
const Index = {1 .. 100000000},
      Vector_1, Vector_2 : [Index] real = 1.0;
proc Equal (v1, v2) : bool
   {return v1 == v2;}
writeln (Equal (Vector_1, Vector_2));
```

Type mismatch

Data Parallelism

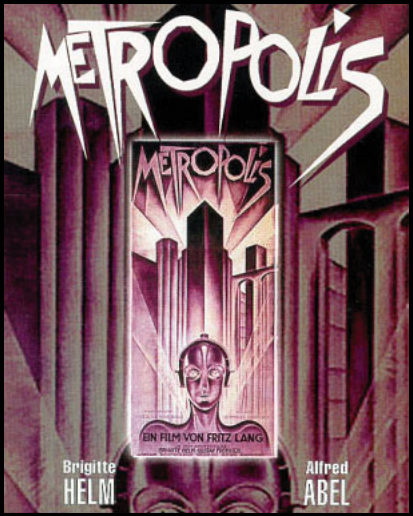
Vector Machines

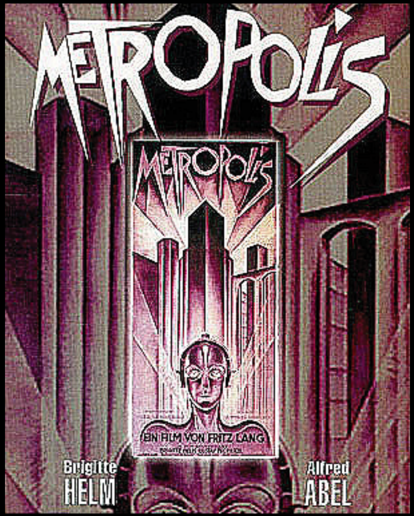
General Data-parallelism



Vector Machines

General Data-parallelism





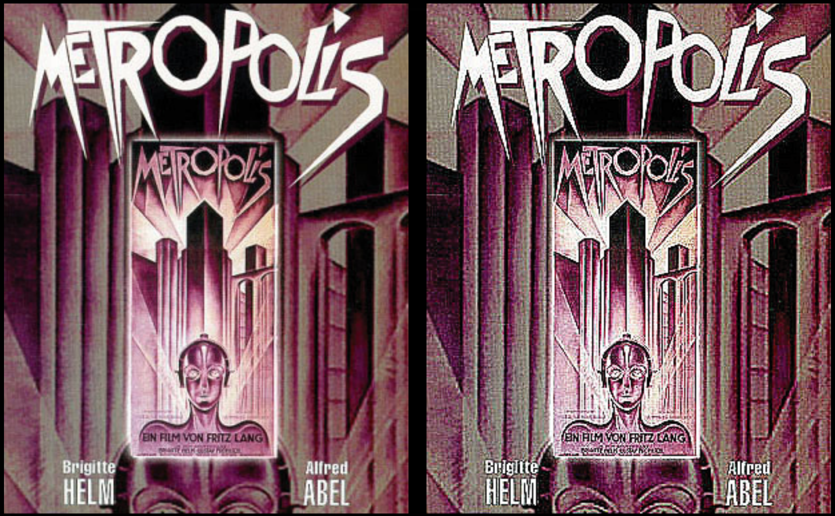
METROPOLIS

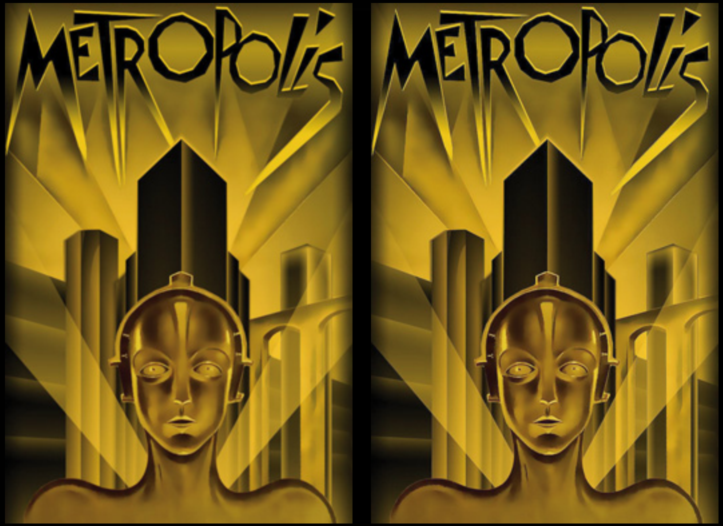
METROPOLIS

Data Parallelism

Vector Machines

General Data-parallelism



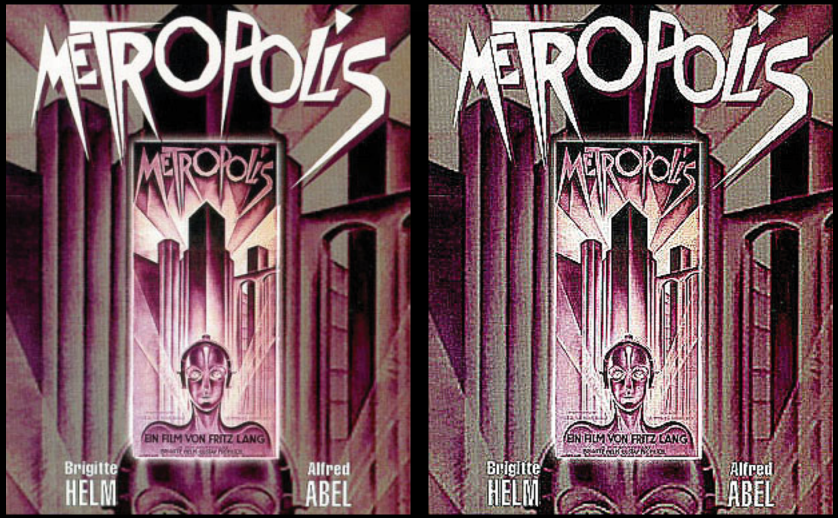


```
const Mask : [1 .. 3, 1 .. 3] real = ((0, -1, 0), (-1, 5, -1), (0, -1, 0));
```

Data Parallelism

Vector Machines

General Data-parallelism



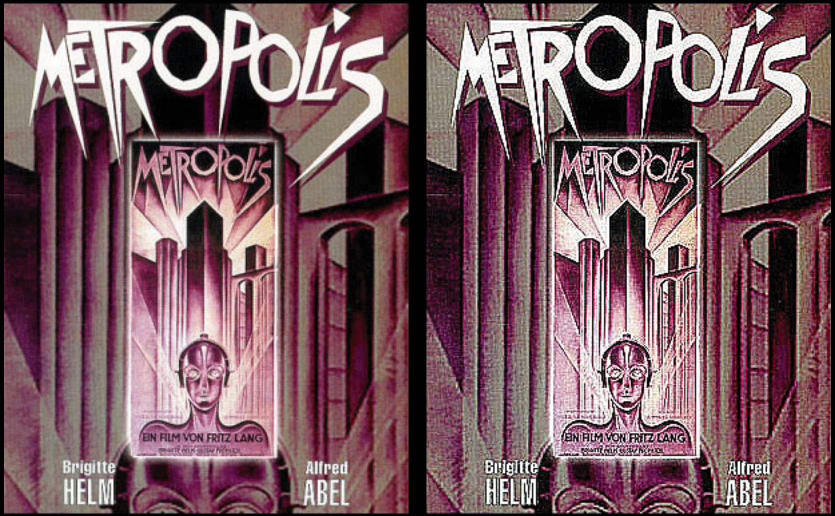
METROPOLIS
METROPOLIS

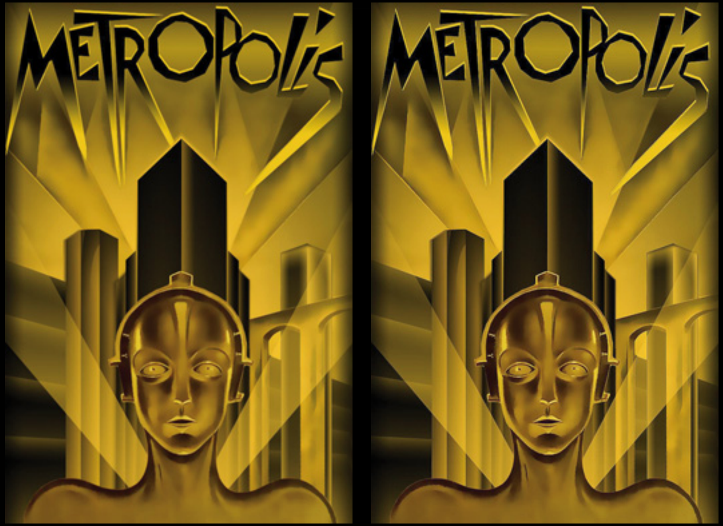
```
const Mask : [1 .. 3, 1 .. 3] real = ((0, -1, 0), (-1, 5, -1), (0, -1, 0));
proc Unsharp_Mask (P, (i, j) : index (Image)) : real
   {return + reduce (Mask * P [i - 1 .. i + 1, j - 1 .. j + 1]);}
```

Data Parallelism

Vector Machines

General Data-parallelism



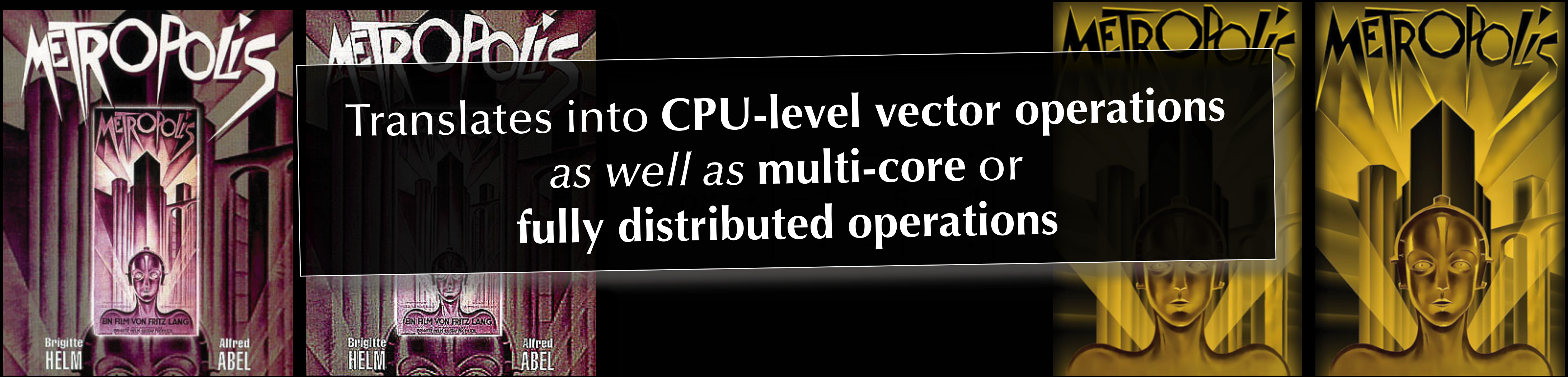


```
const Mask : [1 .. 3, 1 .. 3] real = ((0, -1, 0), (-1, 5, -1), (0, -1, 0));
proc Unsharp_Mask (P, (i, j) : index (Image)) : real
   {return + reduce (Mask * P [i - 1 .. i + 1, j - 1 .. j + 1]);}
const Sharpened_Picture = forall px in Image do Unsharp_Mask (Picture, px);
```

Data Parallelism

Vector Machines

General Data-parallelism



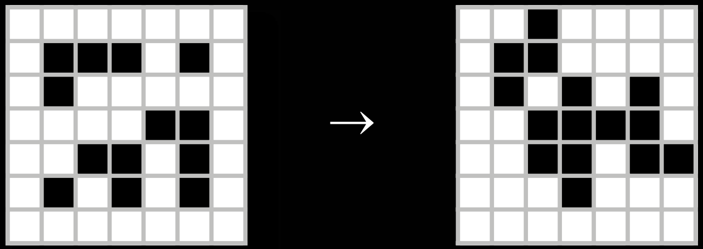
Translates into **CPU-level vector operations** *as well as* **multi-core** or **fully distributed operations**

```
const Mask : [1 .. 3, 1 .. 3] real = ((0, -1, 0), (-1, 5, -1), (0, -1, 0));
proc Unsharp_Mask (P, (i, j) : index (Image)) : real
   {return + reduce (Mask * P [i - 1 .. i + 1, j - 1 .. j + 1]);}
const Sharpened_Picture = forall px in Image do Unsharp_Mask (Picture, px);
```

Data Parallelism

Vector Machines

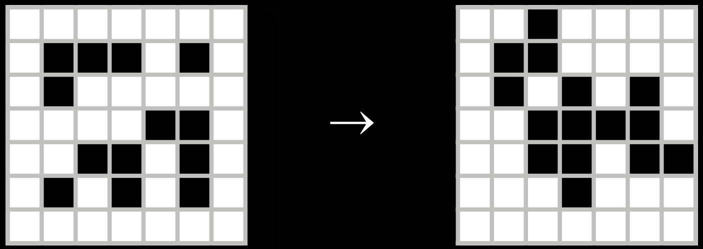
General Data-parallelism



Data Parallelism

Vector Machines

General Data-parallelism

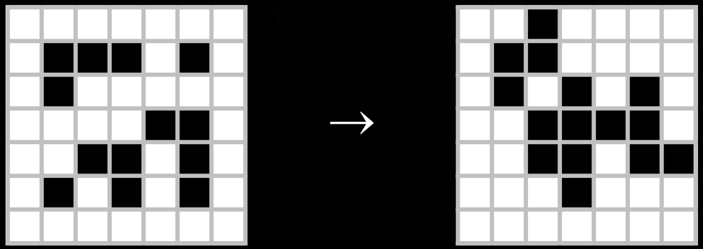


Cellular automaton transitions from a state S into the next state S': $S \rightarrow S' \Leftrightarrow \forall c \in S: c \rightarrow c' = r(S, c)$, i.e. all cells of a state transition *concurrently* into new cells by following a rule r.

Data Parallelism

Vector Machines

General Data-parallelism

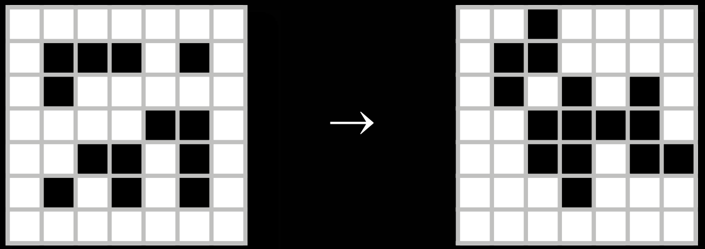


Cellular automaton transitions from a state $\mathtt{S}$ into the next state $\mathtt{S}'$: $\mathtt{S} \rightarrow \mathtt{S}' \Leftrightarrow \forall \mathtt{c} \in \mathtt{S}: \mathtt{c} \rightarrow \mathtt{c}' = \mathtt{r}(\mathtt{S}, \mathtt{c})$, i.e. all cells of a state transition *concurrently* into new cells by following a rule $\mathtt{r}$.

```
Next_State = forall World_Indices in World do Rule (State, World_Indices);
```

Vector Machines

General Data-parallelism



Cellular automaton transitions from a state S into the next state S': $S \to S' \Leftrightarrow \forall c \in S: c \to c' = r(S, c)$, i.e. all cells of a state transition *concurrently* into new cells by following a rule r.

```
Next_State = forall World_Indices in World do Rule (State, World_Indices);
```

John Conway's **Game of Life** rule:

```
proc Rule (S, (i, j) : index (World)) : Cell {
  const Population : index ({0 .. 9}) =
                  + reduce Count (Cell.Alive, S [i - 1 .. i + 1, j - 1 .. j + 1]);
  return (if Population == 3
         || (Population == 4 && S [i, j] == Cell.Alive) then Cell.Alive
                                                        else Cell.Dead);
}
```

Summary

Data Parallelism

- **Data-Parallelism**
  - Vectorization
  - Reduction
  - General data-parallelism
- **Examples**
  - Image processing
  - Cellular automata