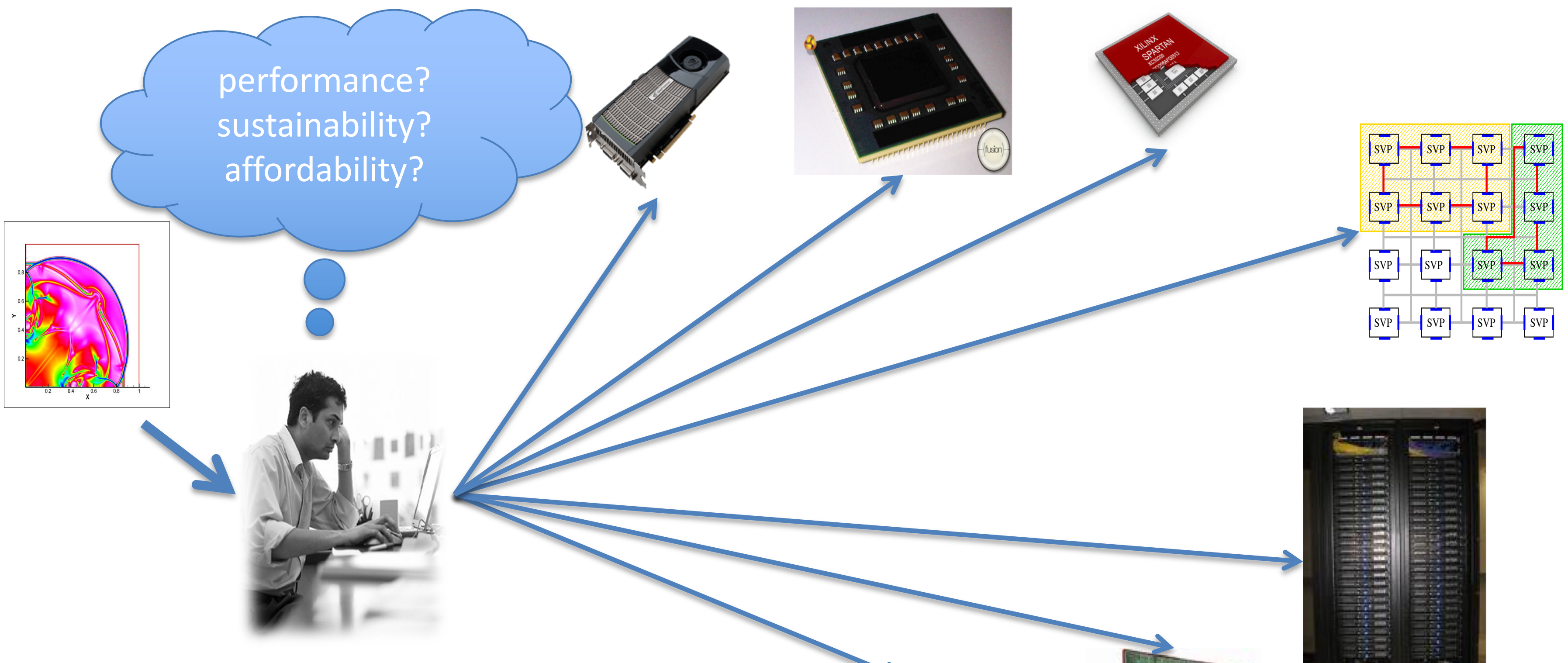
SaC – Functional Programming for HP^3

Chalmers Tekniska Högskola

3.5.2018

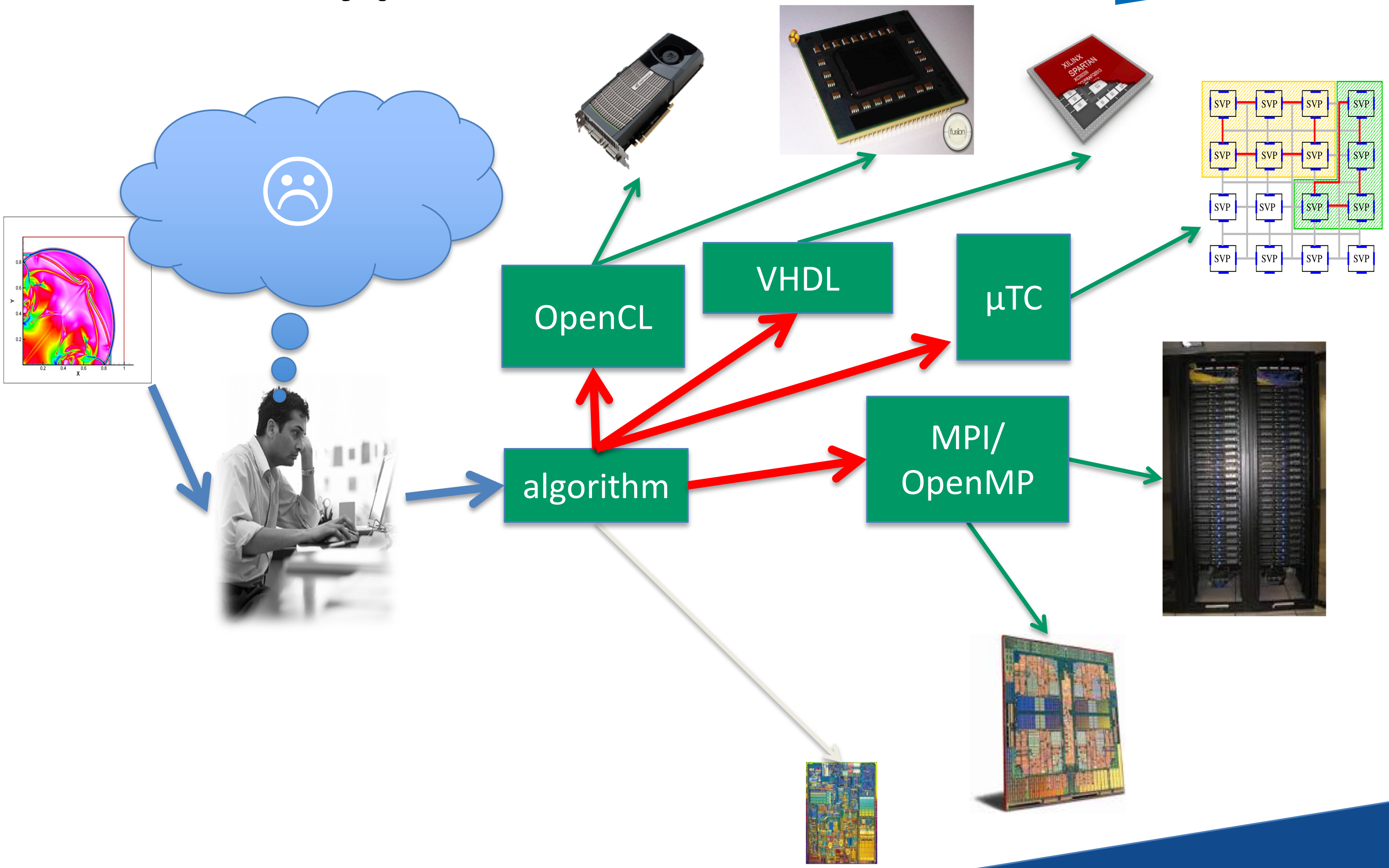
Sven-Bodo Scholz

The Multicore Challenge

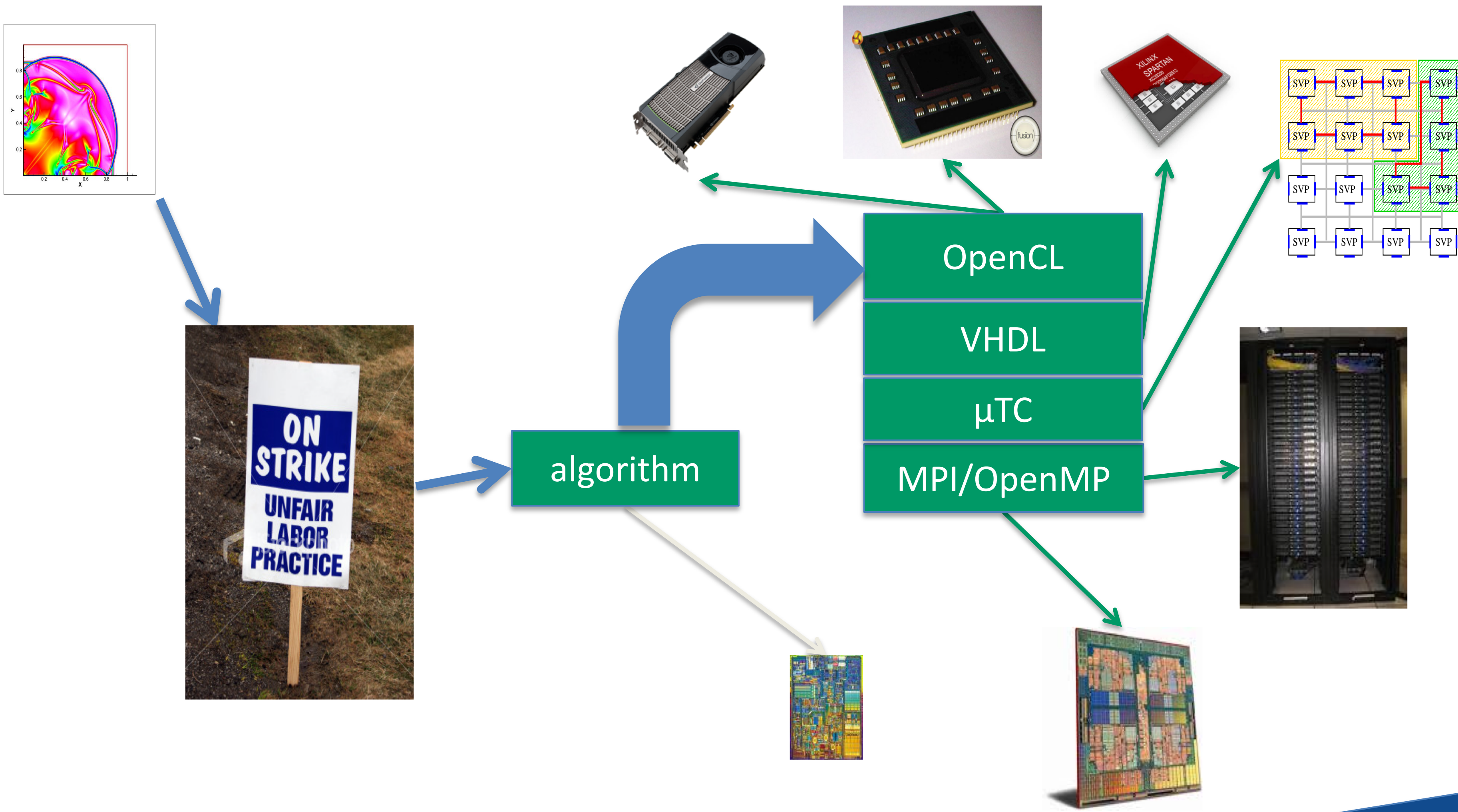


H*igh* ***P****erformance*
H*igh* ***P****ortability*
H*igh* ***P****roductivity*

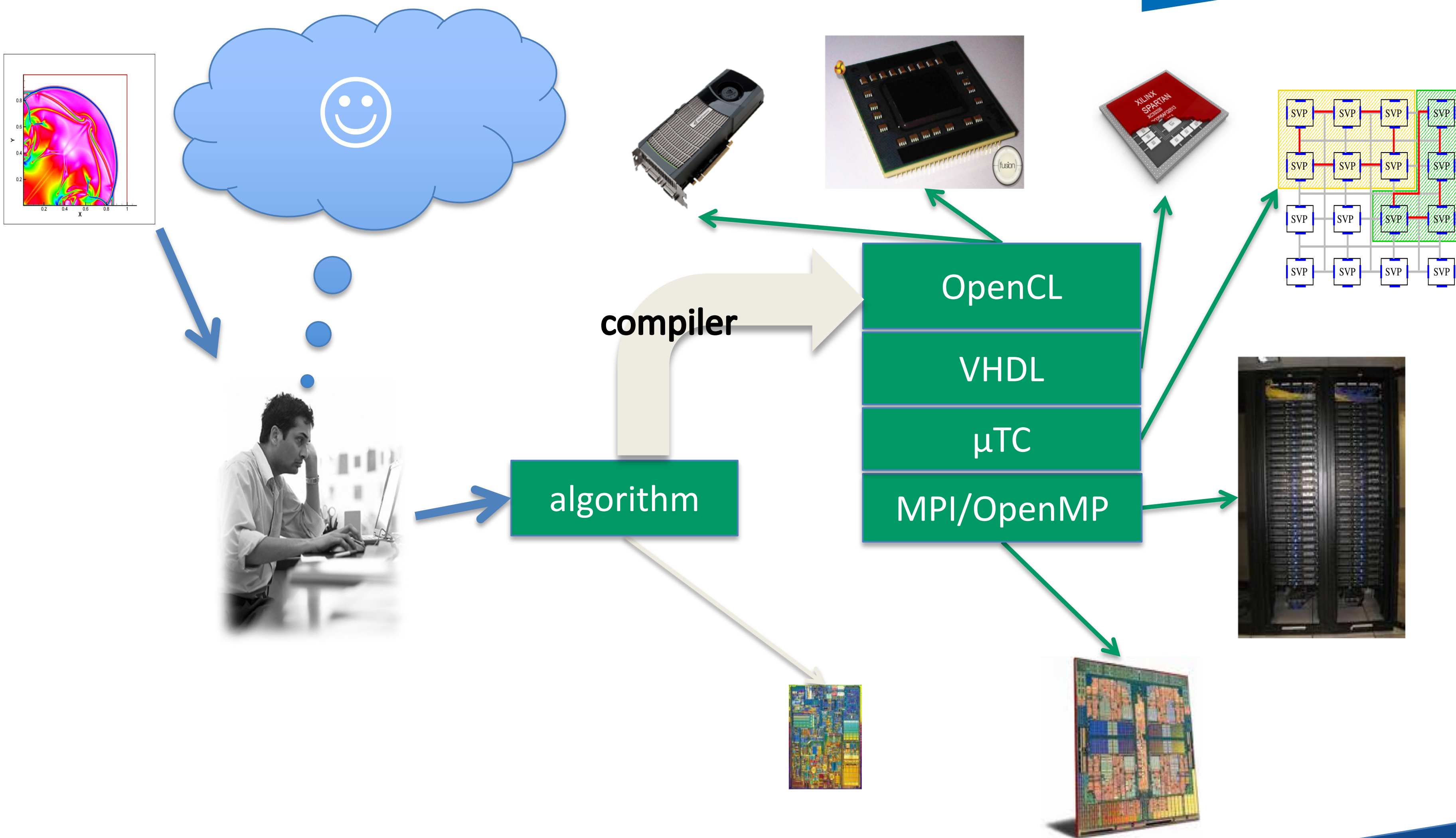
Typical Scenario



Tomorrow’s Scenario



The HP^3 Vision



SaC: HP^3 Driven Language Design

High-Productivity

- easy to learn
  - C-like look and feel
- easy to program
  - Matlab-like style
  - OO-like power
  - FP-like abstractions
- easy to integrate
  - light-weight C interface

&

High-Performance

- no frills
  - lean language core
- performance focus
  - strictly controlled side-effects
  - implicit memory management
- concurrency apt
  - data-parallelism at core

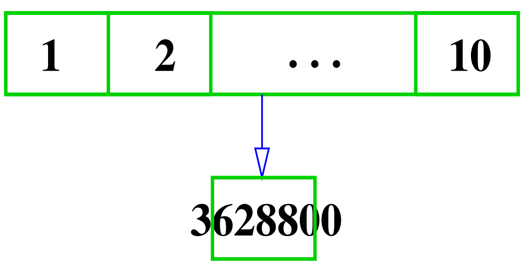
High-Portability

- no low-level facilities
  - no notion of memory
  - no explicit concurrency/ parallelism
  - no notion of communication

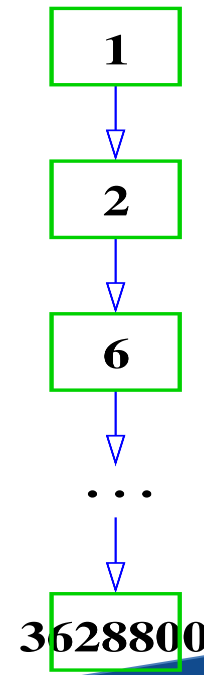
What is Data-Parallelism?

Formulate algorithms in space rather than time!

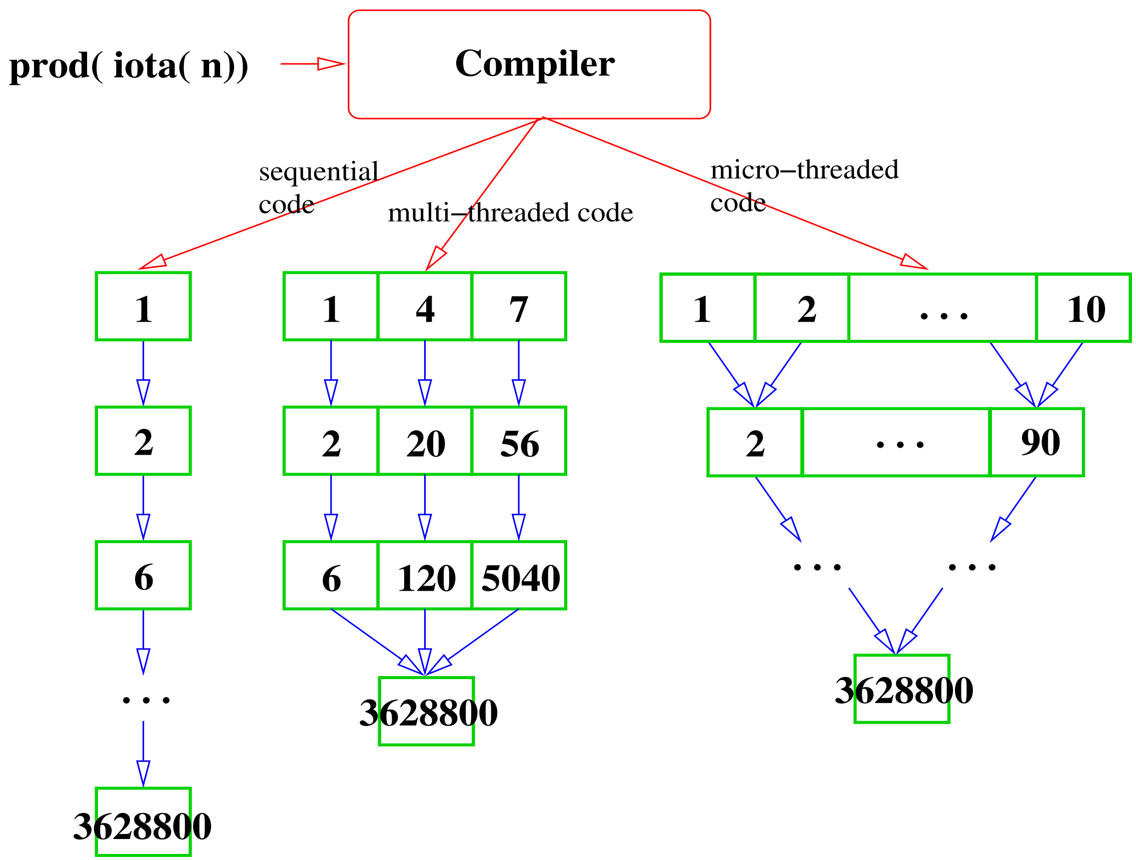
```
prod = prod( iota( 10)+1)
```




```
prod = 1;
for( i=1; i<=10; i++) {
  prod = prod*i;
}
```

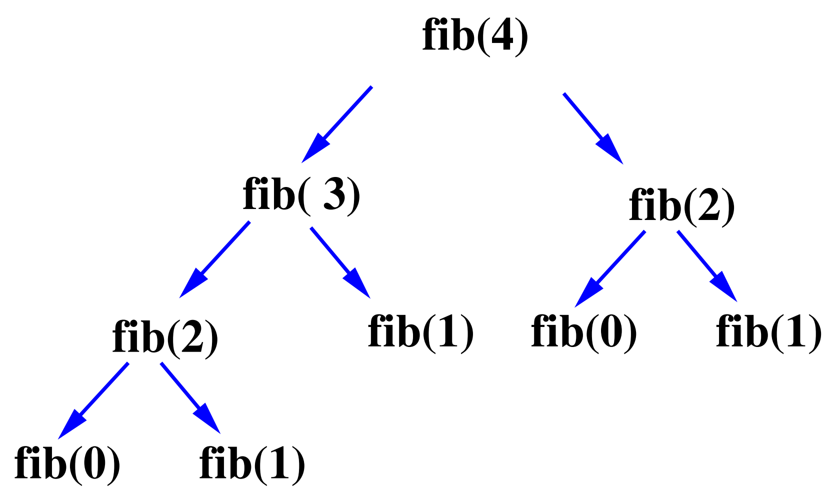


Why is Space Better than Time?



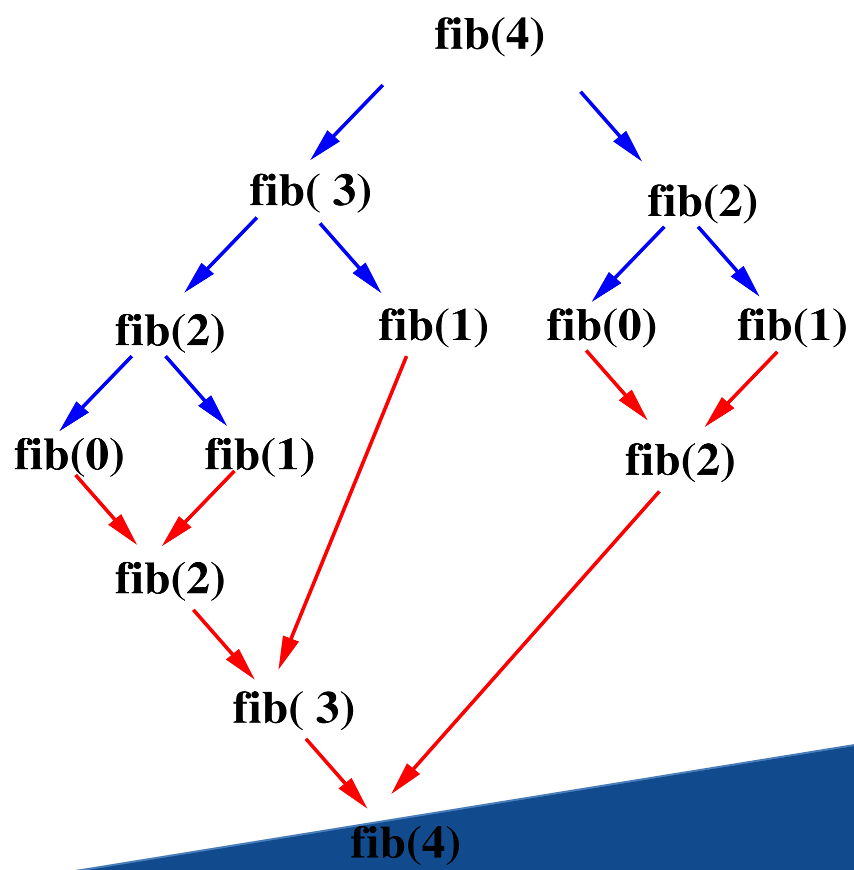
Another Example: Fibonacci Numbers

```
if( n<=1)
  return n;
} else {
  return  fib( n-1) + fib( n-2);
}
```



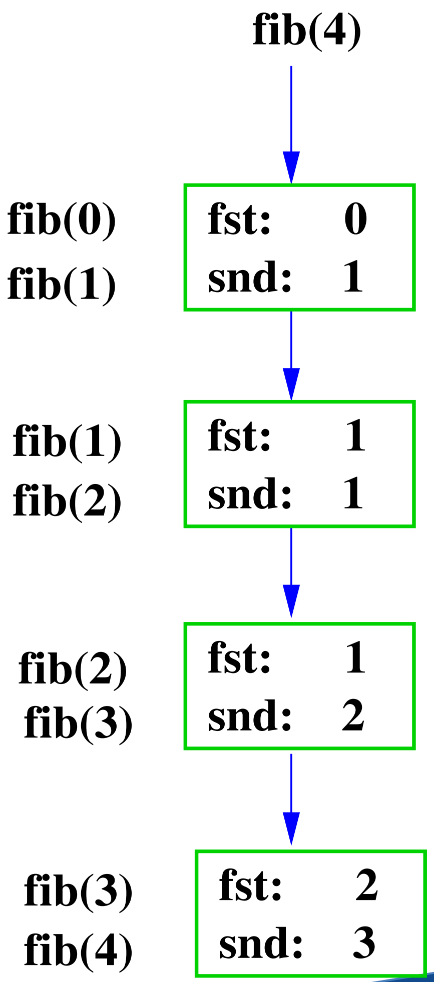
Another Example: Fibonacci Numbers

```
int fib( int n)

if( n<=1)
  return n;
} else {
  return  fib( n-1) + fib( n-2);
}
```



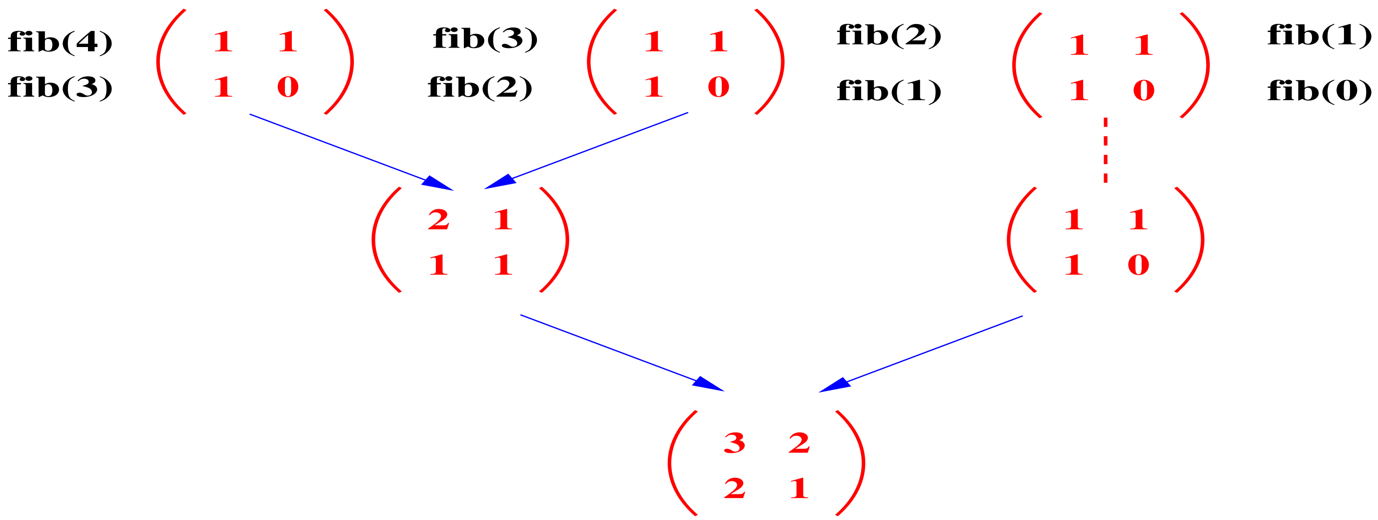
Fibonacci Numbers – now linearised!

```
Int fib'( int fst, int snd, int n)

if( n== 0)
  return fst;
else
  return fib'( snd, fst+snd, n-1)
```



Fibonacci Numbers – now data-parallel!

matprod(genarray([n], [[1, 1], [1, 0]])) [0,0]



Everything is an Array

Think Arrays!

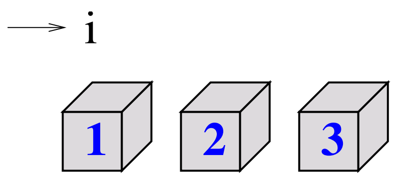
- Vectors are arrays.
- Matrices are arrays.
- Tensors are arrays.
- .... are arrays.

Everything is an Array

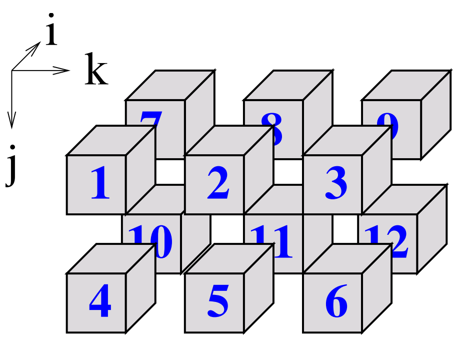
Think Arrays!

- Vectors are arrays.
- Matrices are arrays.
- Tensors are arrays.
- .... are arrays.
- Even scalars are arrays.
- Any operation maps arrays to arrays.
- Even iteration spaces are arrays

Multi-Dimensional Arrays



shape vector: [3]
data vector: [1, 2, 3]



shape vector: [2, 2, 3]
data vector: [1, 2, 3, ..., 11, 12]

42

shape vector: [ ]
data vector: [42]

Index-Free Combinator-Style Computations

L2 norm:

```
sqrt( sum( square( A)))
```

Convolution step:

```
W1 * shift(-1, A) + W2 * A + W1 * shift( 1, A)
```

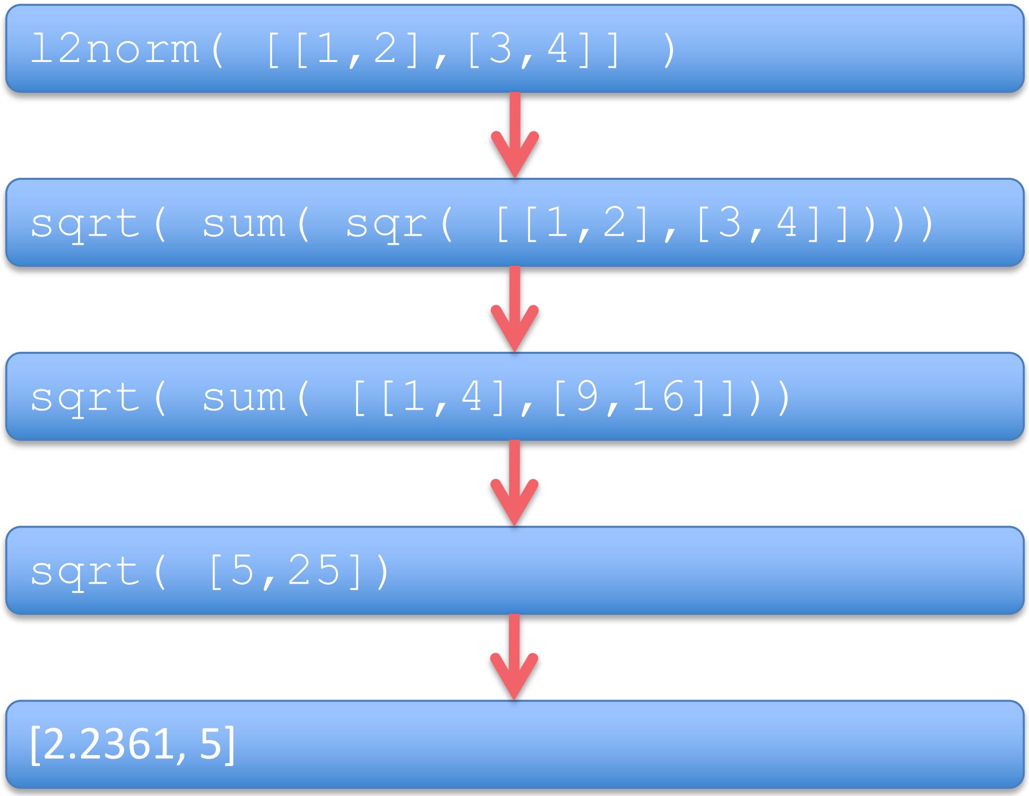
Convergence test:

```
all( abs( A-B) < eps)
```

Shape-Invariant Programming

```
l2norm( [1,2,3,4] )
        ↓
sqrt( sum( sqr( [1,2,3,4])))
        ↓
sqrt( sum( [1,4,9,16]))
        ↓
sqrt( 30)
        ↓
5.4772
```

Shape-Invariant Programming



Where do these Operations Come from?

```
double l2norm( double[*] A)
{
  return( sqrt( sum( square( A))));
}

double square( double A)
{
  return( A*A);
}
```

Where do these Operations Come from?

```
double square( double A)
{
  return( A*A);
}

double[+] square( double[+] A)
{
  res = with {
          (. <= iv <= .) : square( A[iv]);
        } : modarray( A);
  return( res);
}
```

With-Loops

```
with {
   ([0,0] <= iv < [3,4]) : square( iv[0]);
} : genarray( [3,4], 42);
```

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]

→

0	0	0	0
1	1	1	1
4	4	4	4

indices values

With-Loops

```
with {
   ([0,0] <= iv <= [1,1]) : square( iv[0]);
   ([0,2] <= iv <= [1,3]) : 42;
   ([2,0] <= iv <= [2,2]) : 0;
} : genarray( [3,4], 21);
```

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]

indices

0	0	42	42
1	1	42	42
0	0	0	21

values

With-Loops

```
with {
   ([0,0] <= iv <= [1,1]) : square( iv[0]);
   ([0,2] <= iv <= [1,3]) : 42;
   ([2,0] <= iv <= [2,3]) : 0;
} : fold( +, 0);
```

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]

map →

0	0	42	42
1	1	42	42
0	0	0	0

reduce → 170

indices

values

Set-Notation and With-Loops

```
{ iv -> a[iv] + 1}
```

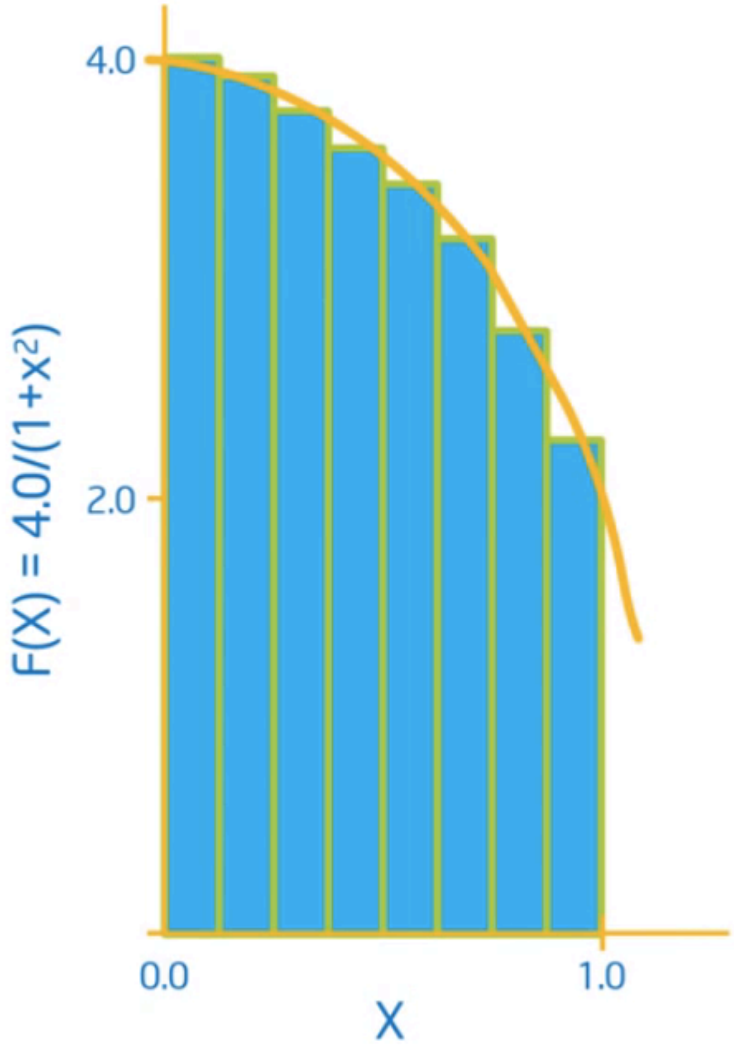
↓

```
with {
  ( 0*shape(a) <= iv < shape(a)) : a[iv] + 1;
} : genarray( shape( a), zero(a))
```

Observation

- most operations boil down to With-loops
- With-Loops are **the** source of concurrency

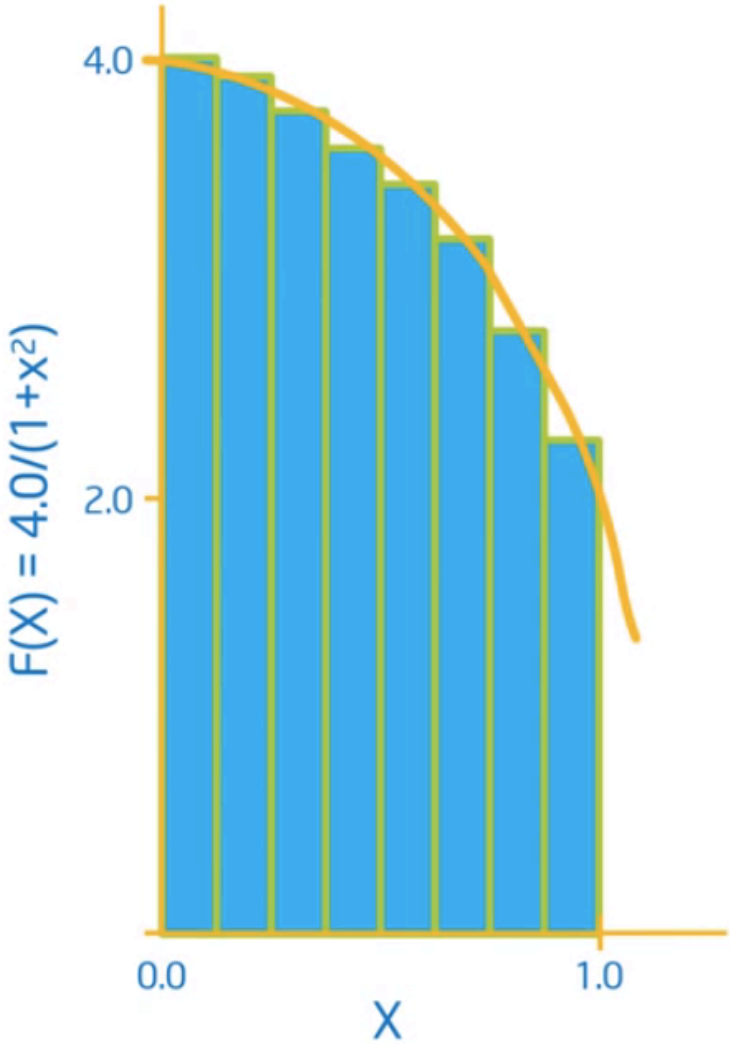
Computation of π



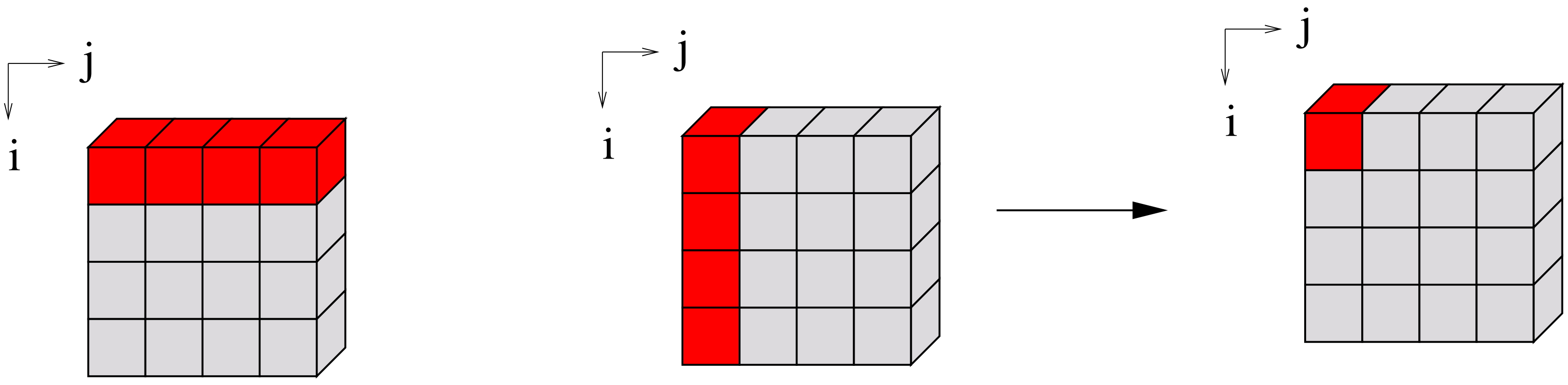
$$\int_0^1 \frac{4.0}{(1+X^2)}$$

Computation of π

```
double f( double x)
{
   return 4.0 / (1.0+x*x);
}

int main()
{
  num_steps = 10000;
  step_size = 1.0 / tod( num_steps);
  x = (0.5 + tod( iota( num_steps))) * step_size;
  y = { iv-> f( x[iv])};
  pi = sum( step_size * y);

  printf( " ...and pi is: %f\n", pi);
  return(0);
}
```



Example: Matrix Multiply

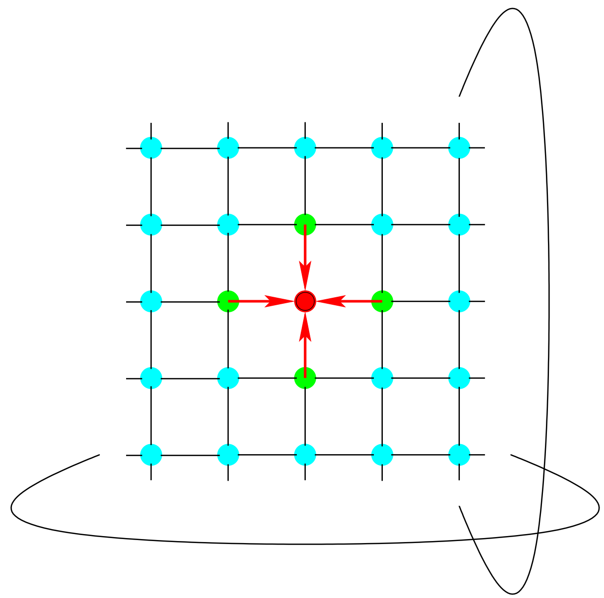


$$(AB)_{i,j} = \sum_k A_{i,k} * B_{k,j}$$

```
{ [i,j] -> sum( A[[i,.]]  * B[[.,j]]) }
```

Example: Relaxation

$$\begin{pmatrix} 0 & 1/8 & 0 \\ 1/8 & 4/8 & 1/8 \\ 0 & 1/8 & 0 \end{pmatrix}$$



```
weights = [[0d,1d,0d], [1d,4d,1d], [0d,1d,0d]] / 8d;
in = ….
out = { iv -> sum(
          { ov -> weights[ov] * rotate( 1-ov, in)[iv]} ) };
```

Programming in a Data-Parallel Style - Consequences

- much less error-prone indexing!
- combinator style
- increased reuse
- better maintenance
- easier to optimise
- huge exposure of concurrency!

What not How (1)

re-computation **not** considered harmful!

```
a = potential( firstDerivative(x));
a = kinetic( firstDerivative(x));
```

What not How (1)

re-computation **not** considered harmful!

```
a = potential( firstDerivative(x));
a = kinetic( firstDerivative(x));
```

compiler

```
tmp = firstDerivative(x);
a = potential( tmp);
a = kinetic( tmp);
```

What not How (2)

variable declaration **not** required!

```
int main()
{
    istep = 0;
    nstop = istep;
    x, y = init_grid();
    u = init_solv (x, y);
...
```

What not How (2)

variable declaration **not** required, ...

but sometimes useful!

```
int main()
{
    double[ 256] x,y;

    istep = 0;
    nstop = istep;
    x, y = init_grid();
    u = init_solv (x, y);
...
```

acts like an assertion here!

What not How (3)

data structures do **not** imply memory layout

```
a = [1,2,3,4];

b = genarray( [1024], 0.0);

c = stencilOperation( a);

d = stencilOperation( b);
```

What not How (3)

data structures do **not** imply memory layout

```
a = [1,2,3,4];

b = genarray( [1024], 0.0);

c = stencilOperation( a);

d = stencilOperation( b);
```

could be implemented by:

```
int a0 = 1;
int a1 = 2;
int a2 = 3;
int a3 = 4;
```

What not How (3)

data structures do **not** imply memory layout

```
a = [1,2,3,4];

b = genarray( [1024], 0.0);

c = stencilOperation( a);

d = stencilOperation( b);
```

or by:

```
int a[4] = {1,2,3,4};
```

What not How (3)

data structures do **not** imply memory layout

```
a = [1,2,3,4];

b = genarray( [1024], 0.0);

c = stencilOperation( a);

d = stencilOperation( b);
```

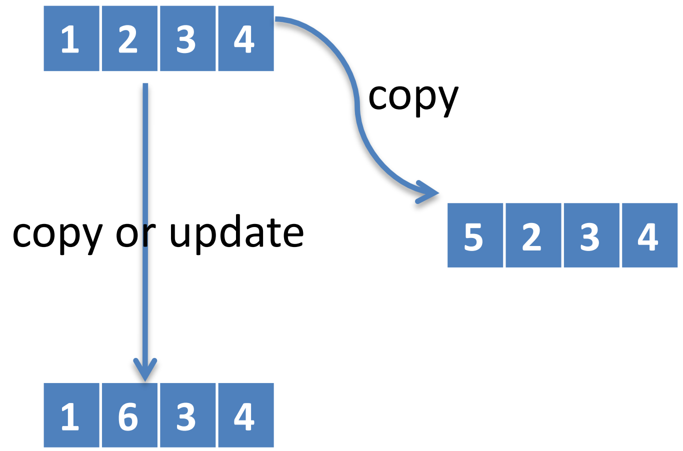
or by:

```
adesc_t a = malloc(...)
a->data = malloc(...)
a->data[0] = 1;
a->desc[1] = 2;
a->desc[2] = 3;
a->desc[3] = 4;
```

What not How (4)

data modification does **not** imply in-place operation!

```
a = [1,2,3,4];

b = modarray( a, [0], 5);

c = modarray( a, [1], 6);
```



What not How (5)

truely implicit memory management

```
qpt = transpose( qp);
deriv = dfDxBoundary( qpt);
qp = transpose( deriv);
```

≡

```
qp = transpose( dfDxNoBoundary( transpose( qp), DX));
```

Challenge: Memory Management: What does the λ-calculus teach us?

conceptual copies

f(a, b , c)

```
f( a, b ,c )
{
   ... a..... a....b.....b....c...
}
```

How do we implement this? – the scalar case


```
f( a, b , c)
```

```
f( a, b ,c )
{
    ... a..... a....b.....b....c...
}
```

operation	implementation
read	read from stack
funcall	push copy on stack

How do we implement this? – the non-scalar case naive approach

conceptual copies

f(a, b , c)

```
f( a, b ,c )
{
    ... a..... a....b.....b....c...
}
```

operation	non-delayed copy
read	O(1) + free
update	O(1)
reuse	O(1)
funcall	O(1) / O(n) + malloc

How do we implement this? – the non-scalar case widely adopted approach

conceptual copies

f(a, b , c)

```
f( a, b ,c )
{
   ... a..... a....b.....b....c...
}
```

GC

operation	delayed copy + delayed GC
read	O(1)
update	O(n) + malloc
reuse	malloc
funcall	O(1)

How do we implement this? – the non-scalar case reference counting approach

conceptual copies

f(a, b , c)

```
f( a, b ,c )
{
   ... a..... a....b.....b....c...
}
```

operation	delayed copy + non-delayed GC
read	O(1) + DEC_RC_FREE
update	O(1) / O(n) + malloc
reuse	O(1) / malloc
funcall	O(1) + INC_RC

How do we implement this? – the non-scalar case a comparison of approaches

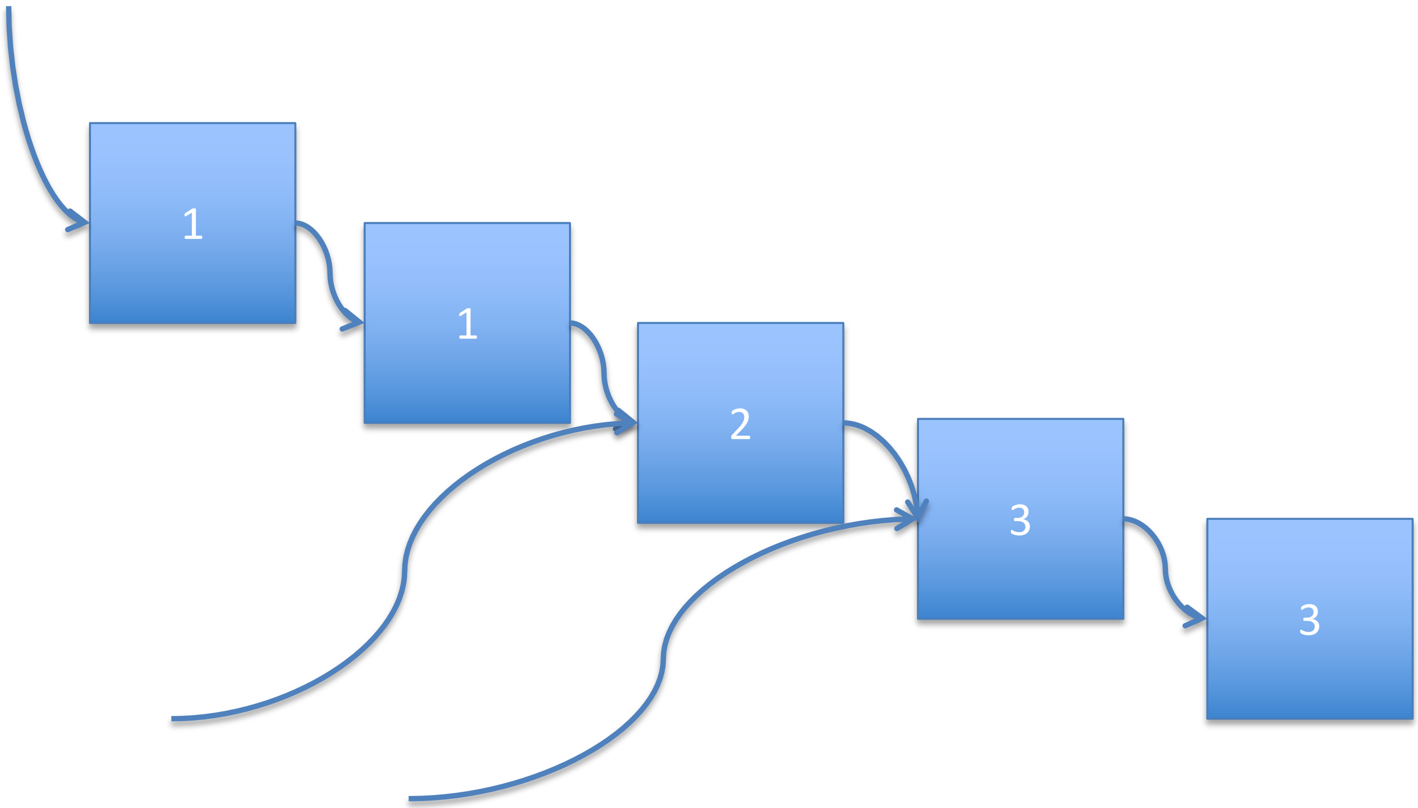
operation	non-delayed copy	delayed copy + delayed GC	delayed copy + non-delayed GC
read	O(1) + free	O(1)	O(1) + DEC_RC_FREE
update	O(1)	O(n) + malloc	O(1) / O(n) + malloc
reuse	O(1)	malloc	O(1) / malloc
funcall	O(1) / O(n) + malloc	O(1)	O(1) + INC_RC

Avoiding Reference Counting Operations

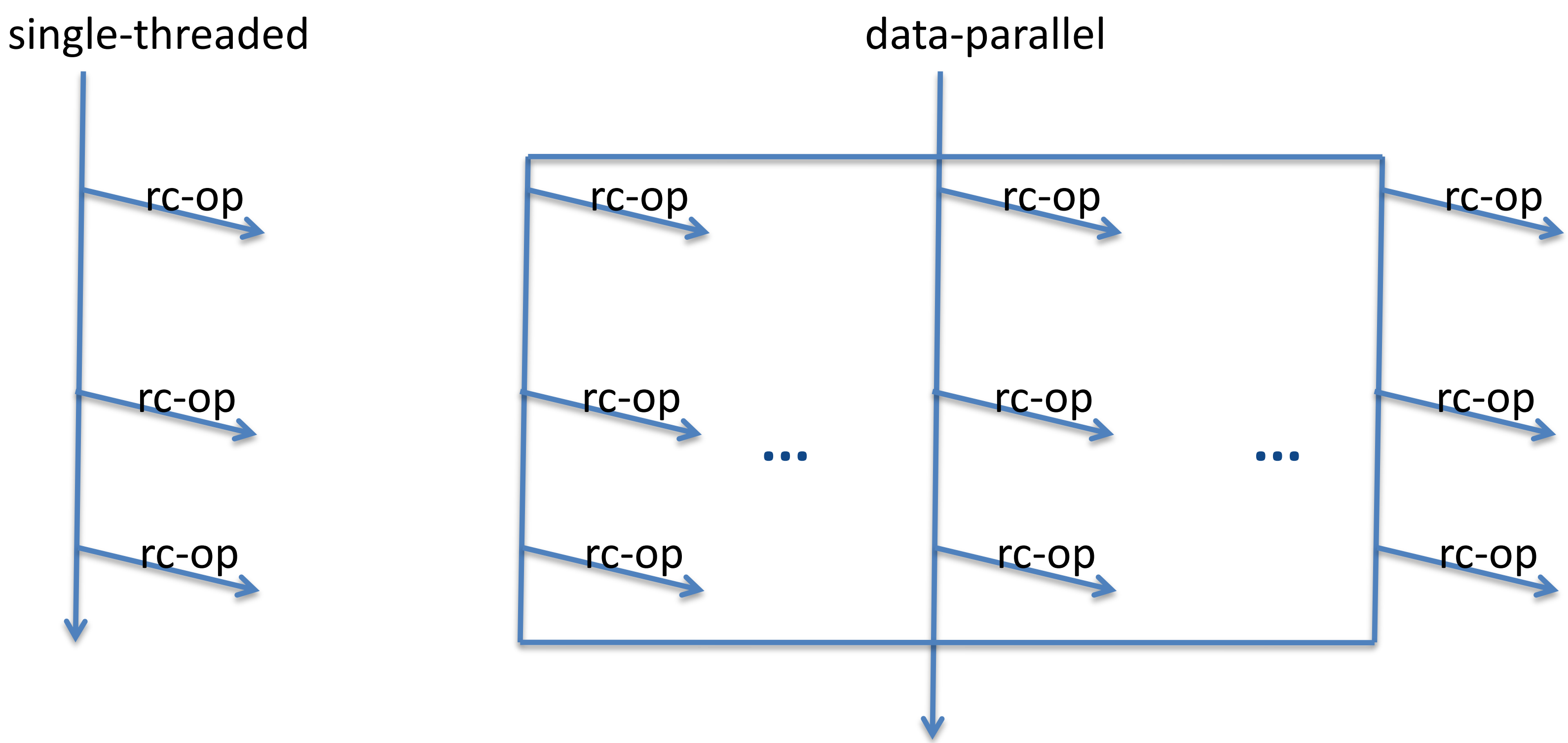
```
a = [1,2,3,4];

b = a[1];

c = f( a, 1);

d= a[2];

e = f( a, 2);
```

- **b = a[1];** ← clearly, we can avoid RC here!
- **c = f(a, 1);** ← we would like to avoid RC here!
- **d= a[2];** ← and here!
- **e = f(a, 2);** ← BUT, we cannot avoid RC here!

NB: Why don't we have RC-world-domination?



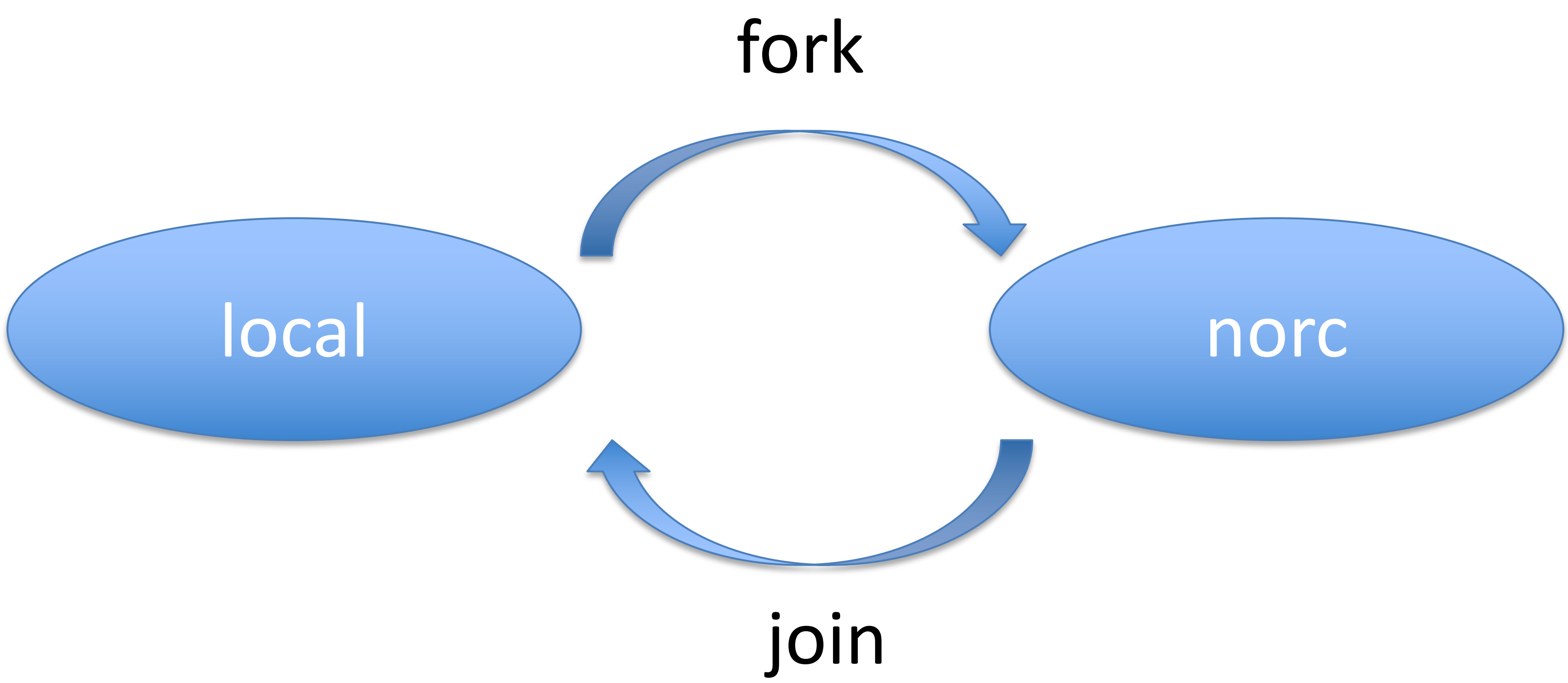
Going Multi-Core



local variables do not escape!
relatively free variables can only benefit from reuse in 1/n cases!

=> use thread-local heaps
=> inhibit rc-ops on rel-free vars

Bi-Modal RC:



SaC Tool Chain

- sac2c – main compiler for generating executables; try
  - sac2c –h
  - sac2c –o hello_world hello_world.sac
  - sac2c –t mt_pth
  - sac2c –t cuda
- sac4c – creates C and Fortran libraries from SaC libraries
- sac2tex – creates TeX docu from SaC files

More Material

- www.sac-home.org
  - Compiler
  - Tutorial
- [GS06b] Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383--427, 2006.
- [WGH+12] V. Wieser, C. Grelck, P. Haslinger, J. Guo, F. Korzeniowski, R. Bernecky, B. Moser, and S.B. Scholz. Combining high productivity and high performance in image processing using Single Assignment C on multi-core CPUs and many-core GPUs. *Journal of Electronic Imaging*, 21(2), 2012.
- [vSB+13] A. Šinkarovs, S.B. Scholz, R. Bernecky, R. Douma, and C. Grelck. SAC/C formulations of the all-pairs N-body problem and their performance on SMPs and GPGPUs *Concurrency and Computation: Practice and Experience*, 2013.

Outlook

- There are still many challenges ahead, e.g.
  - Non-array data structures
  - Arrays on clusters
  - Joining data and task parallelism
  - Better memory management
  - Application studies
  - Novel Architectures
  - … and many more ...
- If you are interested in joining the team:
  - talk to me ☺