HERIOT WATT UNIVERSITY

Data-Parallel Programming using SaC lecture 3

F21DP Distributed and Parallel Technology

Sven-Bodo Scholz

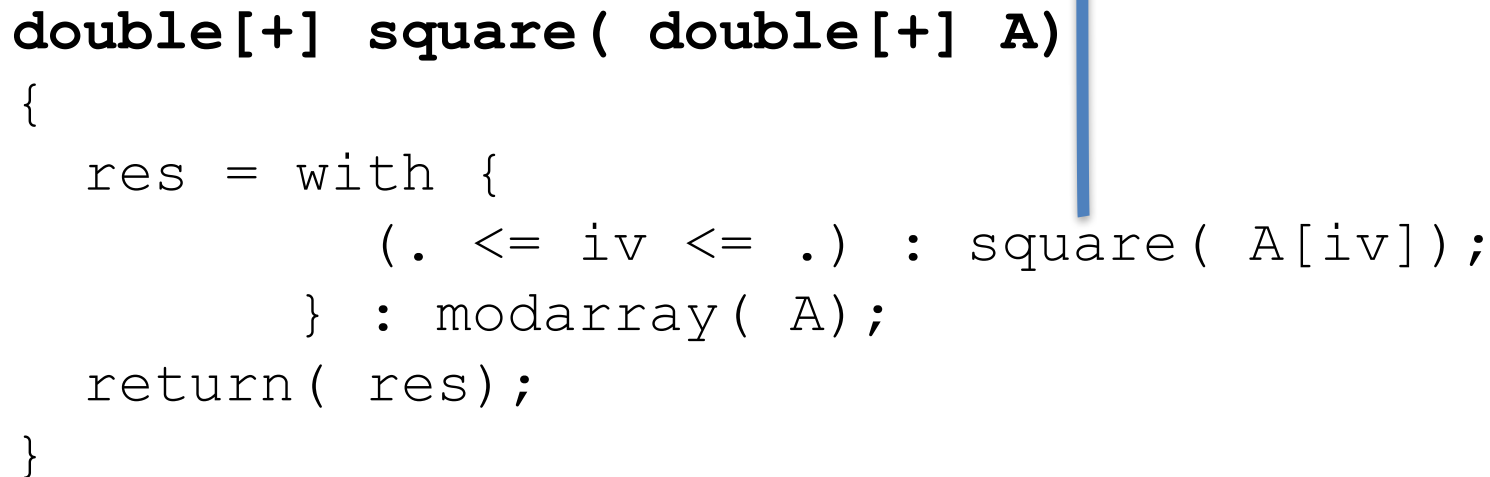
Where do these Operations Come from?

HERIOT WATT UNIVERSITY

```
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

HERIOT WATT UNIVERSITY

```
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]

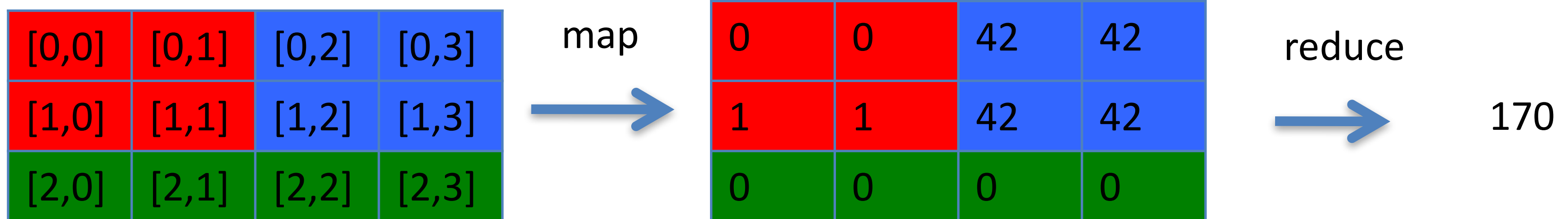
0	0	42	42
1	1	42	42
0	0	0	21

indices

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);
```

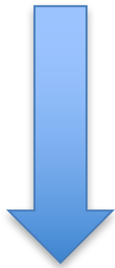


indices values

Set-Notation and With-Loops

HERIOT WATT UNIVERSITY

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



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

Observation

HERIOT WATT UNIVERSITY

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

Modules and Namespaces

HERIOT WATT UNIVERSITY

```
Module A;
export all;

int foo()
{...}

int bar()
{...}
```

Defines a *namespace* “A”

```
int main()
{
  x = A::foo();
  y = A::bar();
     ...
}
```

refers to “A”

```
int foo()
{...}

int main()
{
  x = A::foo();
  y = A::bar();
  z = foo();
     ...
}
```

lives in “MAIN”

Using Modules...

HERIOT WATT UNIVERSITY

```
Module A;
export all;

int foo()
{...}

int bar()
{...}
```

makes all symbols of "A" directly *usable* in "MAIN"

```
use A: all;

int main()
{
  x = foo();
  y = bar();
  ...
}
```

```
use A: all;

int foo()
{...}

int main()
{
  x = A::foo();
  y = bar();
  z = foo();
  ...
}
```

```
use A: all
  except {foo};

int foo()
{...}

int main()
{
  x = A::foo();
  y = bar();
  z = foo();
  ...
}
```

Modules and Overloading

```
Module A;
export all;

int foo( int[*] a)
{...}
```

literally *imports* all definitions from "A"

inhibits imports but allows uses

```
import A: all;

int foo( int[.] v)
{...}

int main()
{
  ...
  x = foo( y);
  ...
}
```

?

```
Module B;
provide all;

int foo( int[*] a)
{...}
```

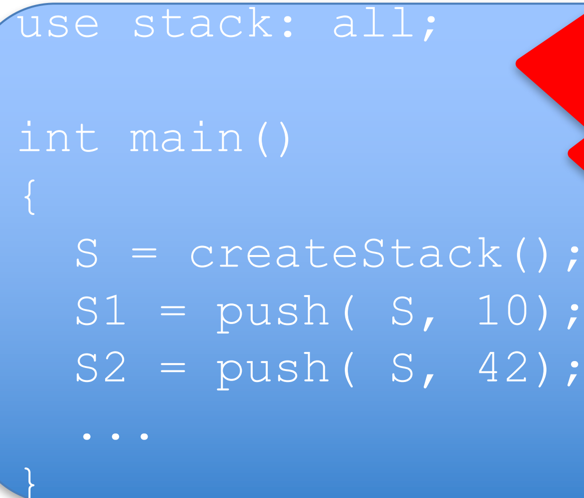
States in SaC

```
Class stack;
classtype int[100];
export all;

stack createStack()
{...}
stack push( stack s, int val)
{ ...}
```

introduces new type "stack"

defines the representation of the type "stack"

```
use stack: all;

int main()
{
  S = createStack();
  S = push( S, 10);
  S = push( S, 42);
  ...
}
```



```
use stack: all;

int main()
{
  S = createStack();
  S1 = push( S, 10);
  S2 = push( S, 42);
  ...
}
```

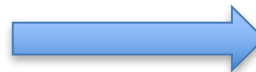
can be done destructively!

Reference Parameters

```
Class stack;
classtype int[100];
export all;

stack createStack()
{...}
void push( stack &s, int val)
{ ...}
```

declares that a modified version of s is returned

```
use stack: all;

int main()
{
  S = createStack();
  push( S, 10);
  push( S, 42);
  ...
}
```



```
use stack: all;

int main()
{
  S = createStack();
  S = push( S, 10);
  S = push( S, 42);
  ...
}
```

internally transformed

Global Objects!!

```
Class stack;
classtype int[100];
export all;

objdef stack myS= createStack();
stack createStack()
{...}
void push(int val)
{ ...myS...}
```

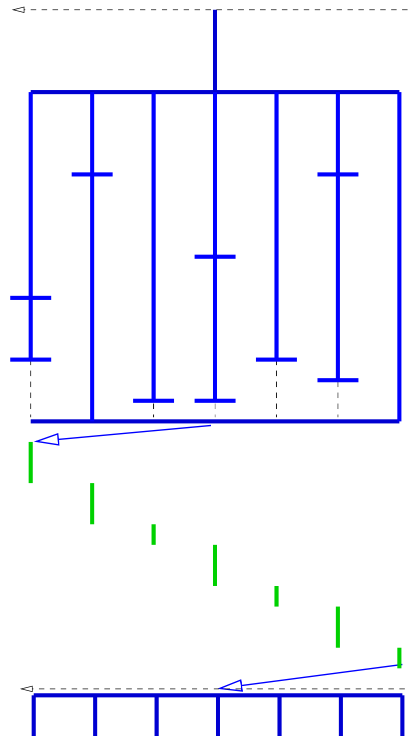
```
use stack: all;

int main()
{
  push( 10);
  push( 42);
  ...
}
```

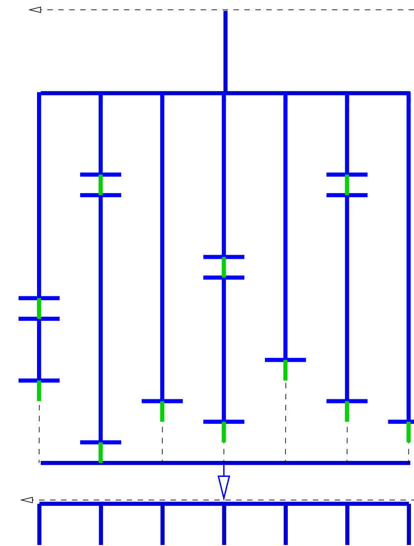
```
use stack: all;

int main()
{
  myS = createStack();
  myS = push( myS, 10);
  myS = push( myS, 42);
  ...
}
```

internally transformed

Asynchronous I/O?

HERIOT WATT UNIVERSITY

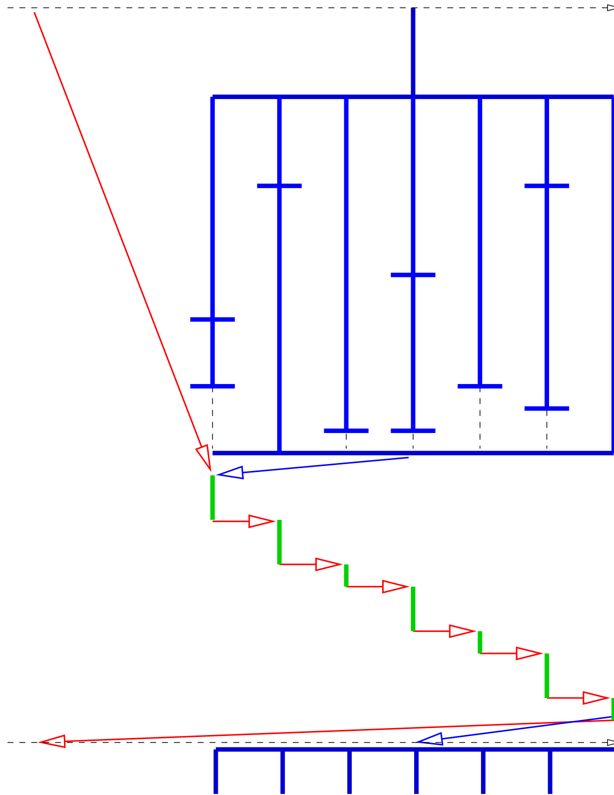


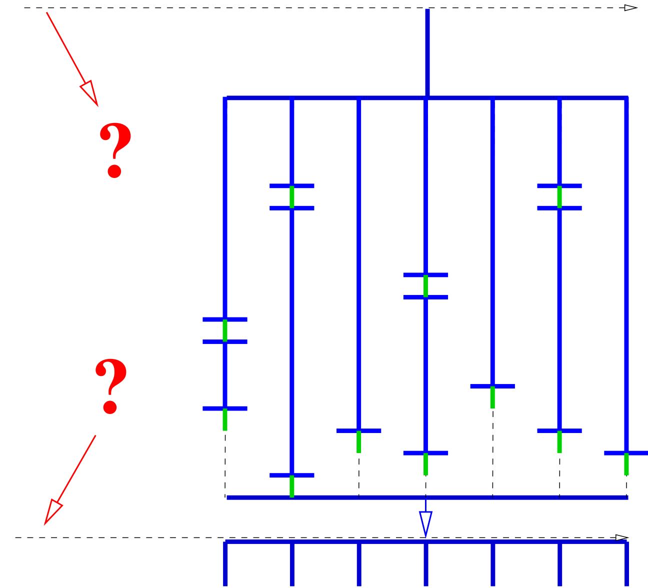


- scales with #cores
- improves debugging
- enables visualisation

But how to model the data dependencies?

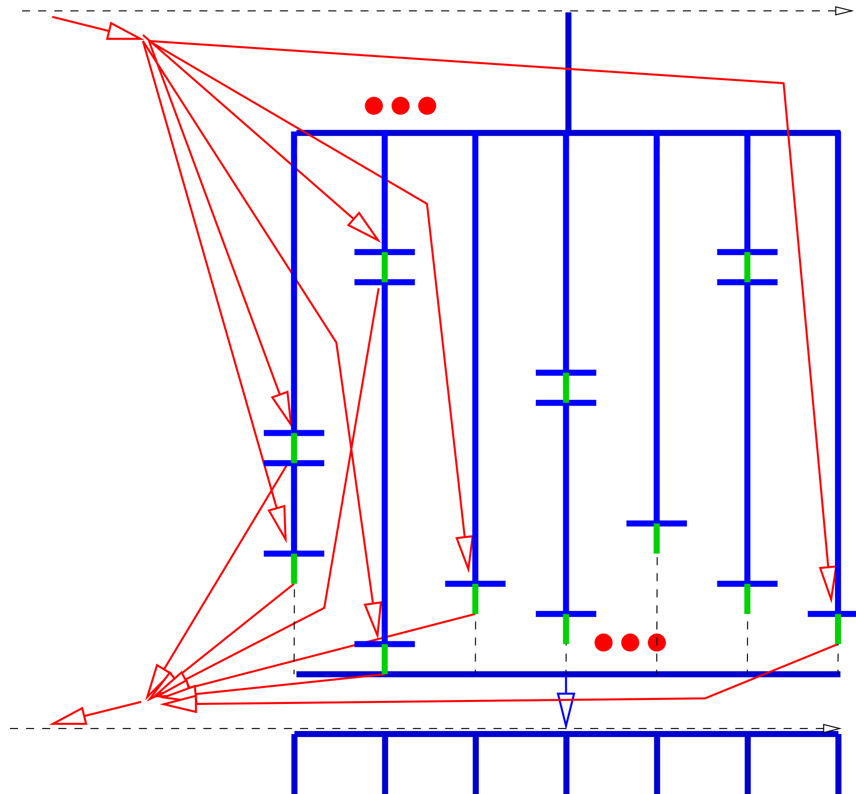
HERIOT WATT UNIVERSITY



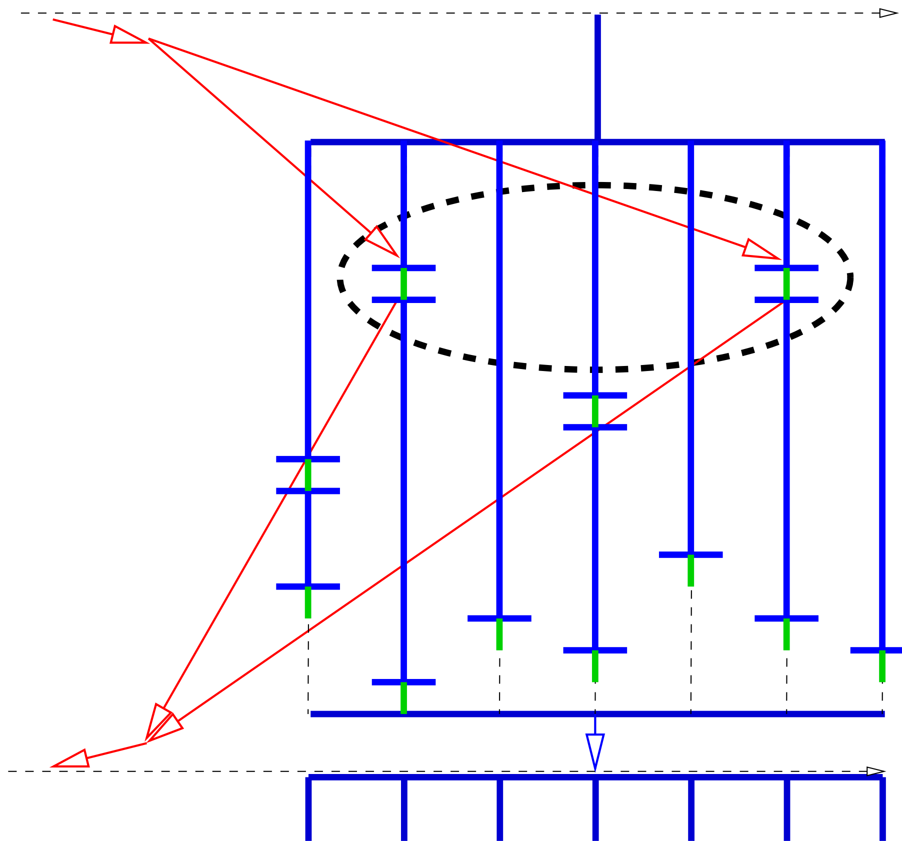


Attempt #1: Split State

HERIOT WATT UNIVERSITY



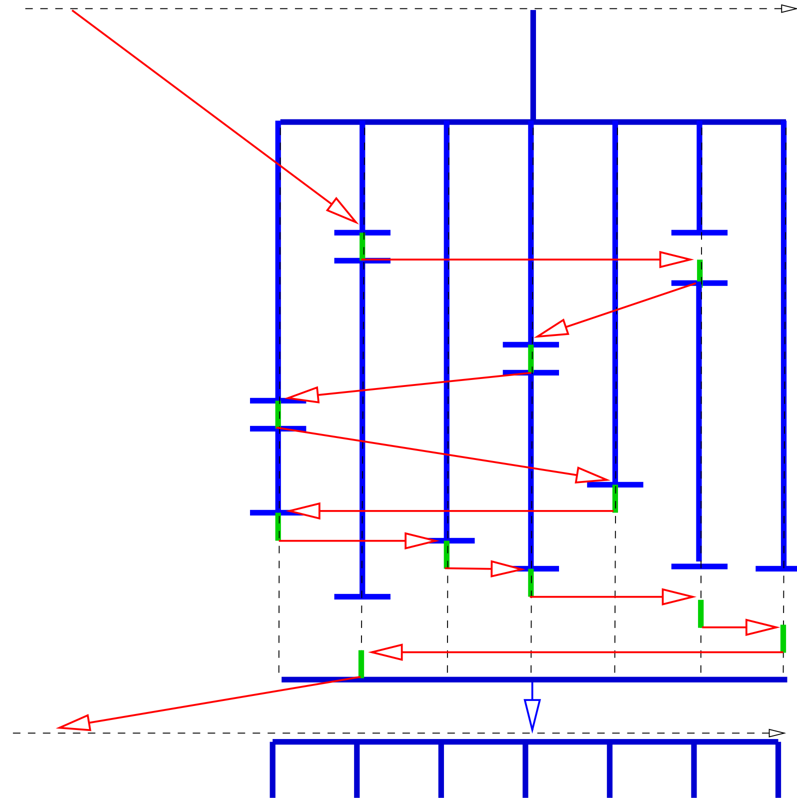
Attempt #1: Split State



may not be possible!

Solution: Non-Deterministic Order!

HERIOT WATT UNIVERSITY



Practical Consequence

HERIOT WATT UNIVERSITY

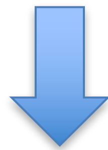
```
with {
  ([0] <= [i] <[10]) {
     printf( "Hi, I am # %d\n", i);
  }
} : void
```

is legal SaC!!

Practical Consequence

HERIOT WATT UNIVERSITY

```
with {
  ([0] <= [i] <[10]) {
     printf( "Hi, I am # %d\n", i);
  } : void;
} : void ;
```



translates into

```
stdout = with {
            ([0] <= [i] <[10]) {
              stdout = printf( stdout,
                               "Hi, I am # %d\n", i);
            } : stdout;
          } : propagate( stdout);
```