Data-Parallel Programming using SaC

lecture 2

F21DP Distributed and Parallel Technology

Sven-Bodo Scholz
The Big Picture

Single Assignment C (SAC)

> 250,000 loc
> 15 years development
> 30 contributors

sac2c auto-parallelising compiler

highly optimised sequential C code

MPI code for distributed systems

POSIX threaded code for SMP

CUDA code for GPGPUs

μTC code for the Microgrid

μTC code for the Microgrid
SAC: HP$^2$ 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
SAC: Basic Principles

- Purely Functional Core
  - enables radical transformations

- Subtyping and Overloading
  - enables high reuse

- Pervasive Array Programming
  - enables massive concurrency
The Functional Programming Perspective

- What not How
  - close to the algorithm
  - no resource / cost awareness

- Features Abstraction
  - generic specifications
  - elaborate type systems

- Consequences:
  + high programming productivity
  + high code reuse
  + high potential for code-modication
  - huge semantic gap
What not How (1)

re-computation **not** considered harmful!

\[
a = \text{potential}(\text{firstDerivative}(x));
\]
\[
a = \text{kinetic}(\text{firstDerivative}(x));
\]
What not How (1)

re-computation **not** considered harmful!

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

tmp = firstDerivative(x);
a = potential( tmp);
a = kinetic( tmp);
```
What not How (2)

variable declaration **not** required!

```c
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!

```c
int main()
{
    double[256] x,y;
    istep = 0;
    nstop = istep;
    x, y = init_grid();
    u = init_solv(x, y);
    ...
    acts like an assertion here!
```
data structures do not imply memory layout

```c
a = [1,2,3,4];
b = genarray([1024], 0.0);
c = stencilOperation(a);
d = stencilOperation(b);
```
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;
data structures do not imply memory layout

```c
a = [1,2,3,4];
b = genarray([1024], 0.0);
c = stencilOperation(a);
d = stencilOperation(b);
```
or by:

```c
int a[4] = {1,2,3,4};
```
What not How (3)

data structures do not imply memory layout

```c
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)

\textbf{truly} implicit memory management

\begin{verbatim}
qpt = transpose(qp);
deriv = dfDxBoundary(qpt);
qp = transpose(deriv);
\end{verbatim}

≡

\begin{verbatim}
qp = transpose(dfDxNoBoundary(transpose(qp), DX));
\end{verbatim}
Subtyping and Overloading

- **Aims**
  - extreme code reuse
  - multi-inheritance

- **Consequences:**
  + high programming productivity
  + high code reuse
  - huge semantic gap
Subtyping in SaC

```c
int[*] a;
int[.,.] m;
int[1,0] m2;

a = 2;
a = [[1,2],[3,4]];
m = [[1,2],[3,4]];
m = [[]];
m2 = [[]];
```
Function Overloading

double methodX( double[.,.] a, double[*] b)
{ ... }
double methodX( double[2,2] a, double[.,.] b)
{ ... }
double methodX( double[.,.] a, double b)
{ ... }
The Array Programming Perspective

- Everything is an Array!
- Index-Free, Combinator Style Computations
- Shape-Invariant Programming
- Consequences:
  - high programming productivity
  - excellent code maintainability
  - high potential for data-parallelism (!)
  - huge semantic gap
Basic Operations

dim(42) == 0

dim([1,2,3]) == 1

dim([[1, 2, 3],
       [4, 5, 6]]) == 2

shape(42) == []

shape([1, 2, 3]) == [3]

shape([[1, 2, 3],
       [4, 5, 6]]) == [2, 3]

a = [[1, 2, 3],
     [4, 5, 6]];

a[[1,0]] == 4

a[[1]] == [1,5,6]
a[[[]]] == a
The Usual Suspects

\[a = [1, 2, 3];\]
\[b = [4, 4, 2];\]

\[a + b = [5, 6, 5];\]
\[a <= b = [true, true, false];\]
\[sum(a) = 6\]

...
Matlab/ APL stuff

```
a = [[1,2,3],
     [4,5,6]];

take( [2,1], a ) == [[1],
                    [4]]
take( [1], a ) == [[1,2,3]] != [1,2,3]
take( [], a ) == a

take( [-1, 2], a ) == [[4,5]]
```

...
Set Notation

\[
\{ \text{iv} \rightarrow a[\text{iv}] + 1 \} = a + 1
\]

\[
\{ [i,j] \rightarrow \text{mat}[[j,i]] \} = \text{transpose( mat)}
\]

\[
\{ [i,j] \rightarrow (i==j? \text{mat}[[i,j]]: 0)\}
\]
Example: Matrix Multiply

\[(AB)_{i,j} = \sum_{k} A_{i,k} \times B_{k,j}\]

\{ [i,j] \rightarrow \text{sum}( A[[i,.]] \times 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
mat = ...
res = { [i,j] -> sum(
    { iv -> weights[iv] * rotate( iv-1, mat)
        [[...,i,j]]})
};
Getting Started: Hello World

use Structures : all;
use Numerical : all;
use StdIO : all;

int main() {
    printf("hello world\n");
    return( 0);
}
SaC Tool Chain

- **sac2c** – main compiler for generating executables; try
  - `sac2c -h`
  - `sac2c -o hello_world hello_world.sac`
  - `sac2c -mt`
  - `sac2c -t cuda`

- **sac4c** – creates C libraries from SaC libraries

- **sac2tex** – creates TeX docu from SaC files