Iteration-wise adjoining

Stefka Fidanova,
Laurent Hascoët,
Christophe Held

Projets TROPICS et SINUS, INRIA Sophia-Antipolis

May 3rd, 2000

Iteration-wise adjoining

Plan:

- Automatic Differentiation reverse mode
- Adjoining a Data-Dependence Graph
- Parallel and Reduction loops
- Iteration-wise adjoint for a terminal parallel loop
- Checkpointing and iteration-wise adjoining
- Comparison with related approaches
- Experimental results

A.D. Reverse mode

Computes the gradient of the program *wrt* its inputs, by mutiplying backwards the *adjoint state* with the local Jacobian of each instruction.

...
save d
```
d = a*b + c
```
$(\mathbf{u})$
save a
```
a = a*c
```
$(\mathbf{v})$
...

Computing $\bar{\mathbf{S}}^t = \bar{\mathbf{S}}^t.J(\mathbf{v}).J(\mathbf{u})$

$$J(\mathbf{v}) = \begin{pmatrix} c & & a & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{pmatrix} \quad J(\mathbf{u}) = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ b & a & 1 & \end{pmatrix}$$

leads to the differentiated program:

...

restore a

$\bar{c} = \bar{c} + a*\bar{a}$

$\bar{a} = c*\bar{a}$

restore d

$\bar{c} = \bar{c} + \bar{d}$

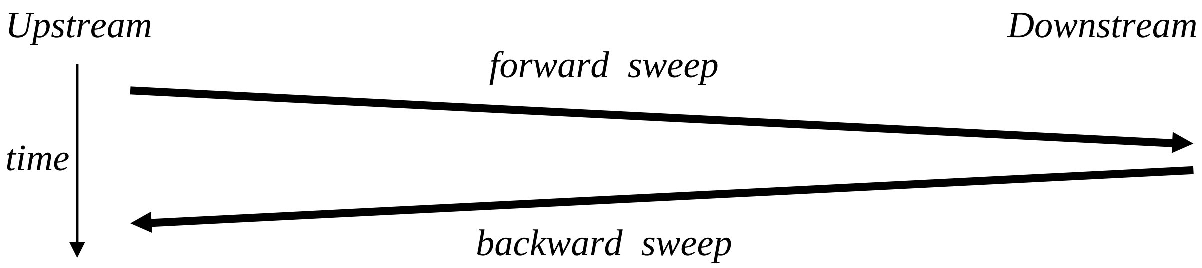
$\bar{b} = \bar{b} + a*\bar{d}$

$\bar{a} = \bar{a} + a*\bar{d}$

$\bar{d} = 0$

...

and to the following execution scheme:

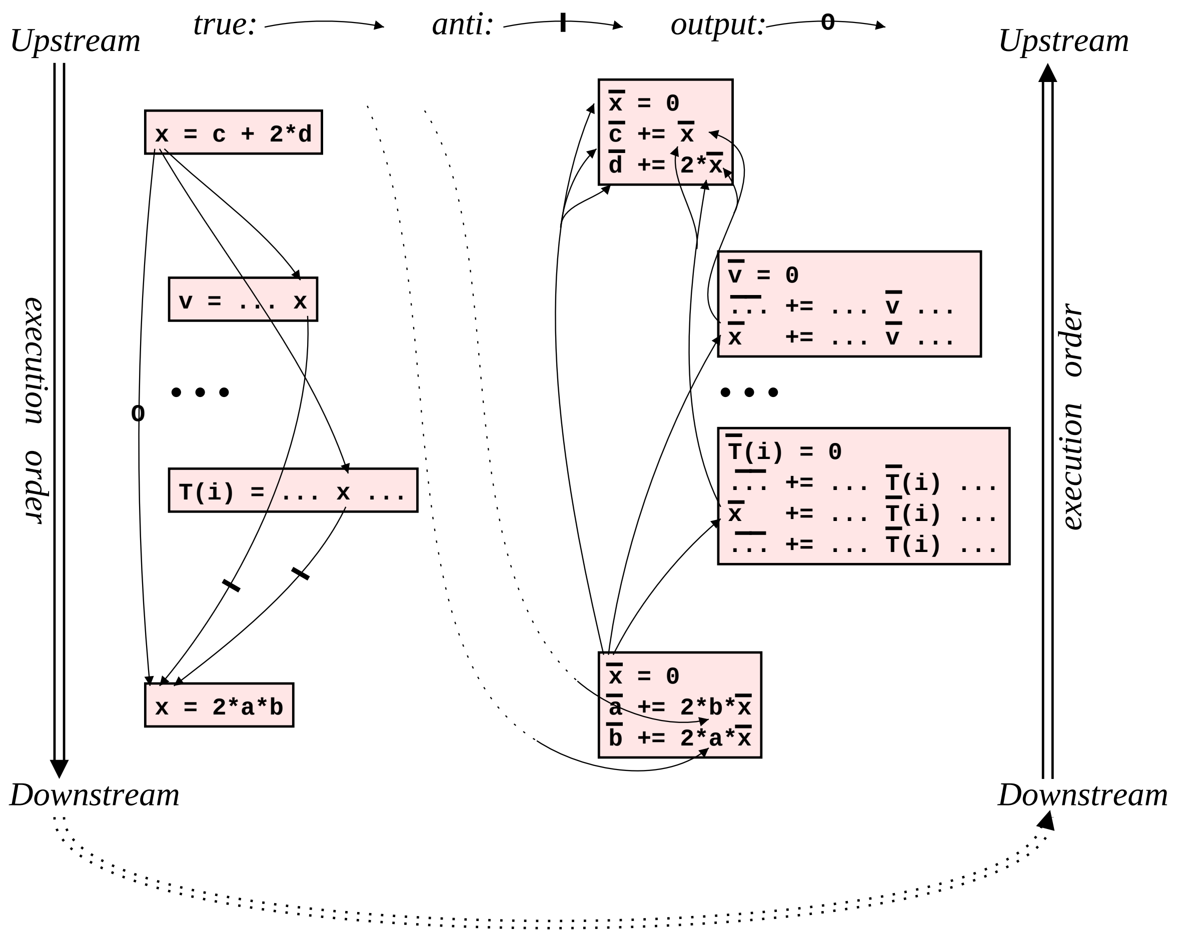


- drawback: the trajectory storage.

Adjoining a Data-Dependence Graph

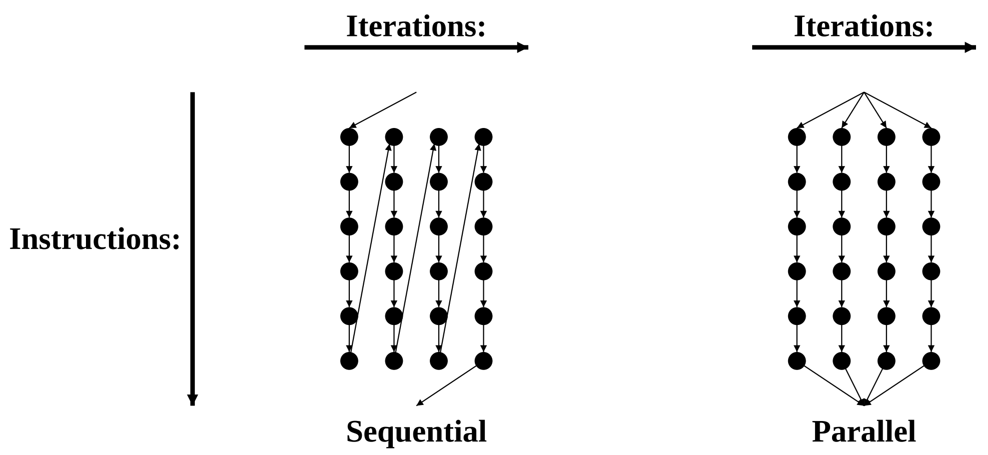
What is the shape of the data-dependence graph of the transposed instructions?

⇒ Same shape as the original dependence graph, with



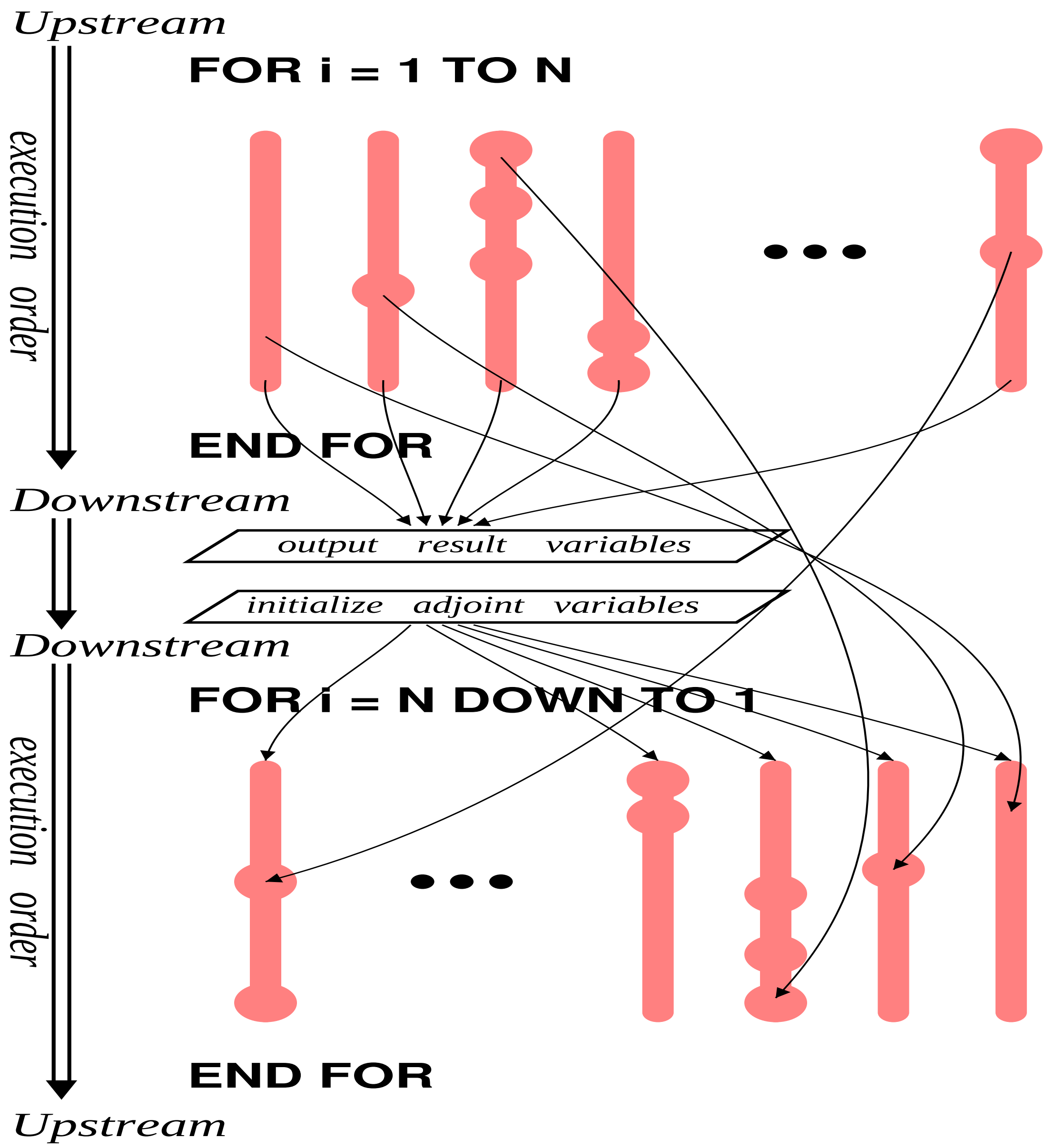
all dependence arrows reversed.

Parallel and Reduction loops



- A loop is parallelizable when there are no *loop-carried* dependences.
- We also consider additive *reduction* operations.

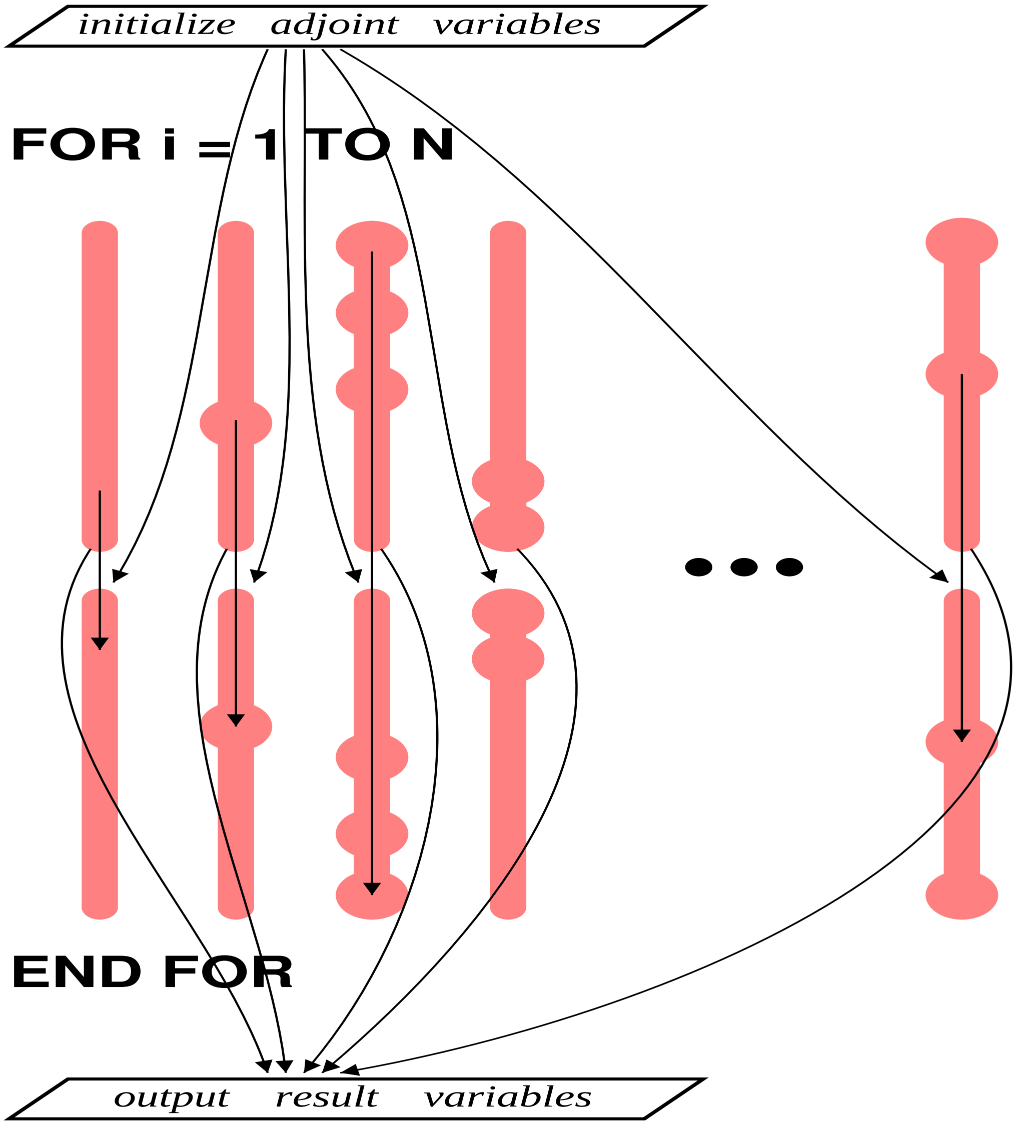
Terminal parallel loop: standard adjoint



Some observations

- The transposed loop is parallel if the original loop is.
- Initial adjoint variables do not depend on the loop
- No cross-iteration dependence between the loops

Terminal parallel loop: iteration-wise adjoint



- This reduces the trajectory storage by a factor N.

Extension to additive reductions

Suppose the loop computes an additive reduction

$$s = s + \sum_i V(i)$$

Then the adjoint mechanism yields:

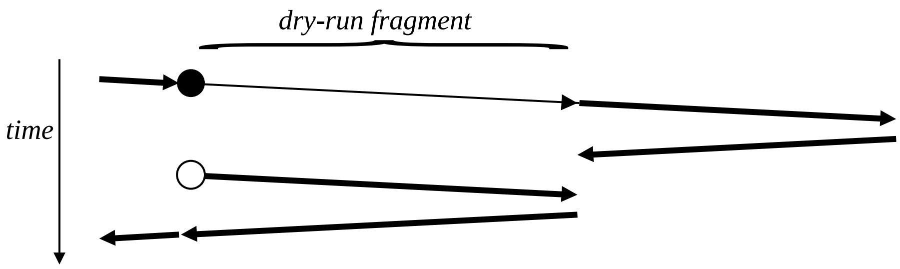
$$\begin{pmatrix} \dot{s} \\ \dot{V}(1) \\ \dot{V}(2) \\ \vdots \\ \dot{V}(N) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{pmatrix} \times \begin{pmatrix} \dot{s} \\ \dot{V}(1) \\ \dot{V}(2) \\ \vdots \\ \dot{V}(N) \end{pmatrix}$$

$$\Longrightarrow \quad \begin{pmatrix} \bar{s} \\ \bar{V}(1) \\ \bar{V}(2) \\ \vdots \\ \bar{V}(N) \end{pmatrix} = \begin{pmatrix} 1 & & & & \\ 1 & 1 & & & \\ 1 & & 1 & & \\ \vdots & & & \ddots & \\ 1 & & & & 1 \end{pmatrix} \times \begin{pmatrix} \bar{s} \\ \bar{V}(1) \\ \bar{V}(2) \\ \vdots \\ \bar{V}(N) \end{pmatrix}$$

Therefore Iteration-wise adjoining can be applied to additive reductions.

What if the loop is not terminal?

Apply “dry-running” of the loop:

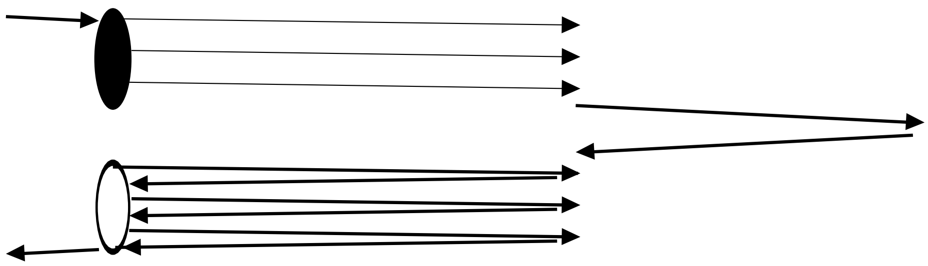


Cost is

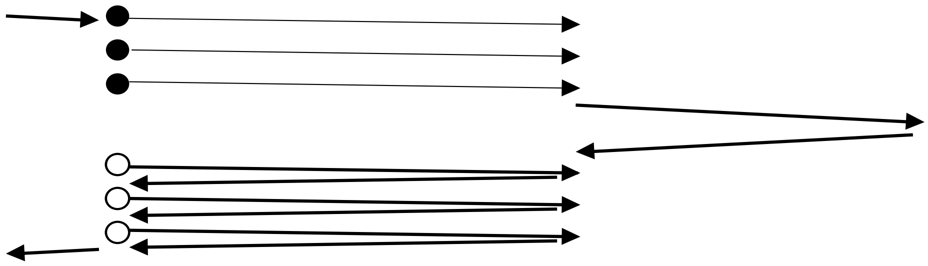
- one extra execution
- storage of the input state

Comparison with other approaches

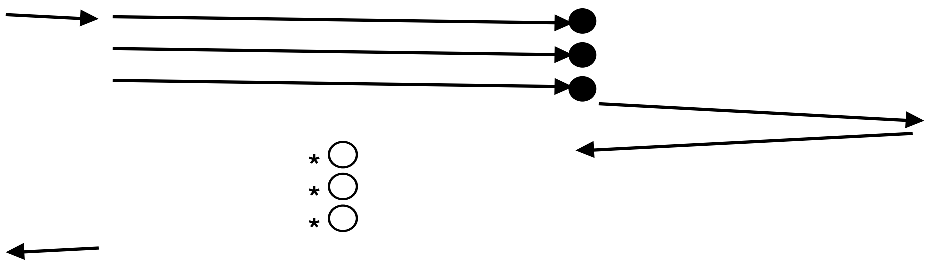
Dry run (//) *memory cost: O(N)* *exec cost: O(N)*



Local dry run *memory cost: O(3N)* *exec cost: O(N)*



Local Jacobian *memory cost: O(9N)* *exec cost: O(N?)*



Experimental results

On the central assembly loops of a 2D Navier-Stokes flow solver. Finite element/finite volume approach, unstructured triangular mesh. 10 successive loops on O(1000) elements.

	reference	**iteration-wise**
orig	171,000	171,000
adj	297,000	297,000
ckp	0	47,000
traj	2,400,000	800
total	2,868,000	515,800

Execution time: 3% slower