Static Analyses and Transformations of Programs: from Parallelization to Differentiation

Mémoire d'Habilitation

Laurent Hascoët
Laurent.Hascoet@inria.fr
http://www-sop.inria.fr/tropics

INRIA Sophia Antipolis

January 28, 2005

Goals of this presentation

- To show what I learnt and what I contributed in program analyses and transformations.
- To claim that experience from Parallelization can be profitable for Differentiation.

3: **Parallelization**	*technology transposed to*	4: **Differentiation**
		4.1: **Presentation of AD** **Basic principles** **Focus on the Reverse Mode**
		4.2: **Motivating Applications of Reverse AD** **Data Assimilation; Optimization**
3.1: **Internal Representation of Programs**	→	*4.3:* **Using Flow Graphs for Reverse AD**
3.2: **Static Data-Flow Analyses for better //**	→	*4.4:* **Static Data-Flow Analyses for better AD**
3.3: **Parallelization using the Dependence Graph**	→	*4.5:* **Using the Dependence Graph in AD**
3.4: **Application Tool 1: PARTITA**		*4.6:* **Application Tool : TAPENADE**
3.5: **Application Tool 2: SPMD Parallelization**		

Outline

1. Introduction
2. (Semi-)Automatic Parallelization
   - Parallelization and Data Dependencies
   - Parallelization strategy in PARTITA
   - SPMD parallellization
3. (Semi-)Automatic Differentiation
   - Control Flow and Discontinuities
   - Reverse AD: the quest for cheap gradients
   - Reverse AD: trajectory inversion problem
   - Static Analyses for Reverse AD
   - Applications of Data Dependencies
   - TAPENADE
4. Conclusion

Program Analysis and Transformation Tools

- One of the most important class of programs: *Compilers, Translators, Debuggers, Parallelizers, Predicate Provers, Partial Evaluators, . . .*

- Can be static or dynamic

- Can preserve or augment the results

- Yield reliable, optimized results, in a short time, and it can be reproduced!

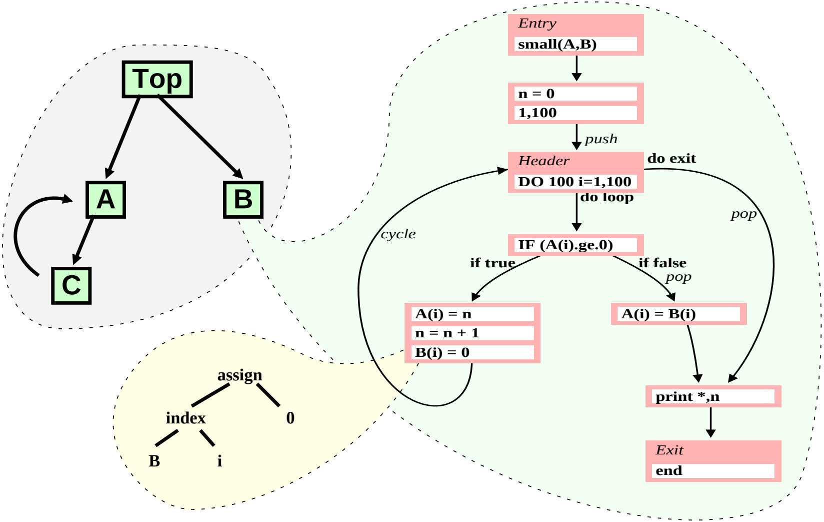
Tools' internal representation of programs

Internal representation should be

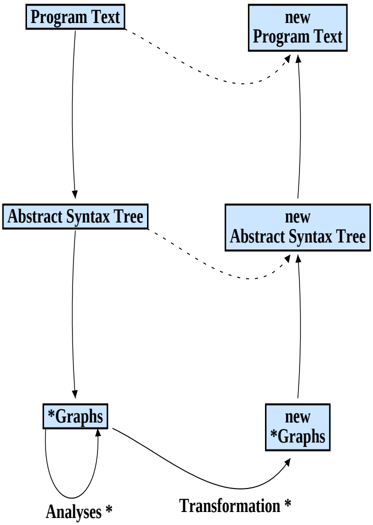
- independent from the language
  (e.g. abstract syntax, CFG),
- convenient for tools analyses
  rather than for human reader,
- independent from the transformation targetted.

Classically done with 3 levels:
Call Graphs, Flow Graphs, Syntax Trees

Call Graphs, Flow Graphs, Syntax Trees



Analyses and Transformations



- Analyses run on graphs and AST's,
- with fixpoint iterations for cyclic structures.
- Analyses and transformations must remain at the internal representation level.

Things (not) to do

- Forget non-decidability :-)
- Limit specialization to a strict minimum
- Avoid undoing transformations
- Don't restrict to mini-languages
- Source code is only a hint, but use it!

Outline

(Semi-)Automatic Parallelization

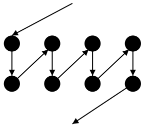
- Parallelizing is: rescheduling a program's statements to take advantage of a special target (language+compiler+architecture)
- Rescheduling must preserve the results or semantics
- Quality depends on executions time and communications time
- Naturally extends to Grid computing as well as cache optimizations
- Some compilers can do it partly

Outline

All languages guarantee some execution order:

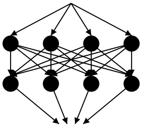
Sequential:

```
DO i=4,7
  A(i) = C(i)
  C(i+2)=B(i)
ENDDO
```



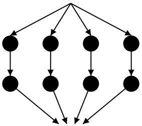
Vectorial:

```
A(4:7) = C(4:7)
C(6:9) = B(4:7)
```



Parallel:

```
PARALLEL DO i=4,7
  A(i) = C(i)
  C(i+2)=B(i)
END PARALLEL DO
```

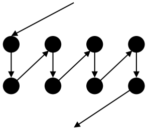


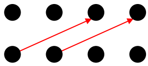
which strongly impacts the semantics!

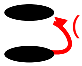
What can't be changed in the execution order?

- Everything is permitted, except switching a *write-then-read*, *read-then-overwrite*, or a *write-then-overwrite* at a given memory location.
- Summarized as the *dependence graph*, a sub-order of the execution order.

```
DO i=4,7
  A(i) = C(i)
  C(i+2)=B(i)
ENDDO
```







Actual detection of dependencies

```
...
if (g.ge.7) then
  k = k0
  do i=0,n
     do j=1,m,2
        T(j,k) = ...
        ...  T(j+g,k+6)
     enddo
     k = k - 3
  enddo
endif
end
...
```

Dependence from (i,j) to (i′,j′) iff

$$\mathtt{j} = \mathtt{j}' + \mathtt{g}'$$
$$\mathtt{k} = \mathtt{k}' + 6$$

constraint propagation (g), induction variables (j,k), loop and array bounds:

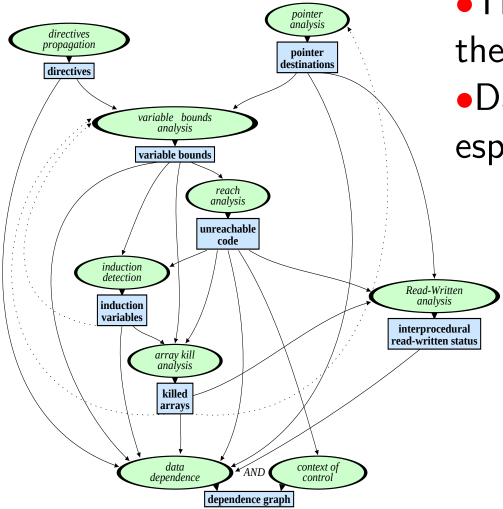
$$-3*\mathtt{lc}_1 + 3*\mathtt{lc}'_1 = 6$$
$$2*\mathtt{lc}_2 - 2*\mathtt{lc}'_2 - \mathtt{g} = 0$$
$$\mathtt{g} >= 7$$
$$2 <= 2*\mathtt{lc2} + 1 <= \mathtt{m}$$
$$0 <= \mathtt{lc1} <= \mathtt{lc1}'$$

solved with an integer programming system:

$$2 \leq \mathtt{lc}'_1 - \mathtt{lc}_1 \leq 2$$
$$-\infty < \mathtt{lc}'_2 - \mathtt{lc}_2 \leq -4$$

$\Rightarrow$ dependency, with distance $(2,]-\infty, -4])$

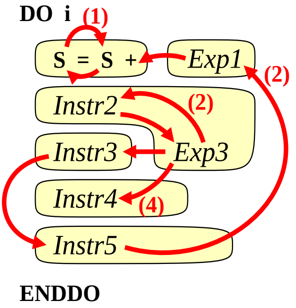
Static Data-Flow analyses reduce dependencies

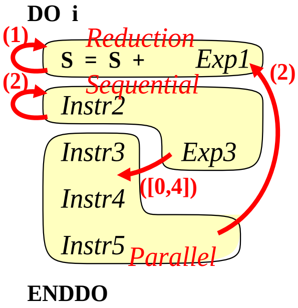


- The fewer dependencies, the better
- Data-Flow analyses help especially:
  - constraint propagation
  - induction detection
  - used/killed variables

Loop parallelization by Acyclic Condensation

For (nested) loops, cycles in the dependence graph correspond to non-parallel parts.






```
DO i
    Instr2
    tmp(i) = Exp3
ENDDO

PARALLEL DO i
    Instr3...tmp(i)...
    Instr4
    Instr5
END PARALLEL DO

S = S + SUM(Exp1)
```

Loop parts not in cycles are parallel (or vectorial).

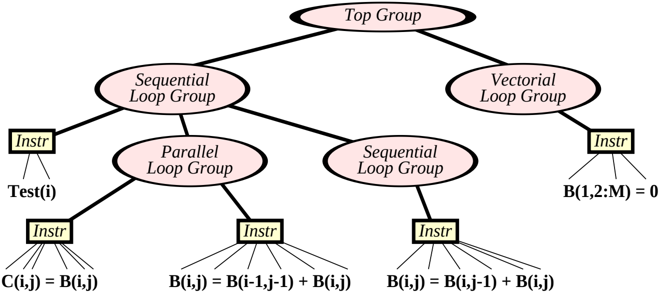
Outline

1. Introduction
2. (Semi-)Automatic Parallelization
   - Parallelization and Data Dependencies
   - Parallelization strategy in PARTITA
   - SPMD parallellization
3. (Semi-)Automatic Differentiation
   - Control Flow and Discontinuities
   - Reverse AD: the quest for cheap gradients
   - Reverse AD: trajectory inversion problem
   - Static Analyses for Reverse AD
   - Applications of Data Dependencies
   - TAPENADE
4. Conclusion

Nested Loops as Nested Groups

A tree of nested loop levels, indicating parallelism, with leaves for instructions:

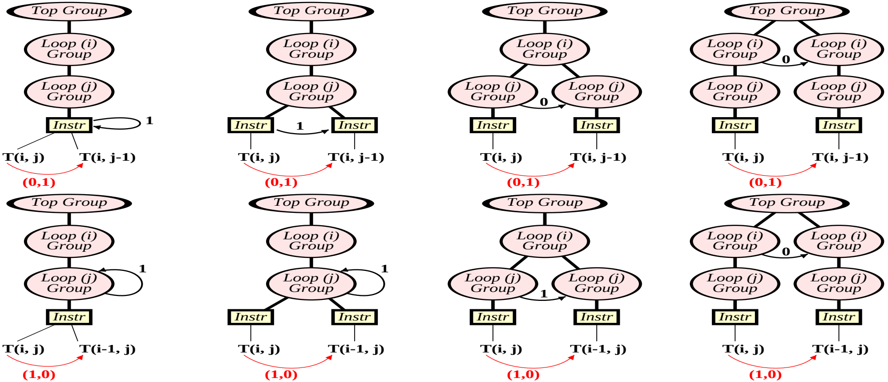
```
DO i=1,N
  IF (Test(i)) THEN
    PARALLEL DO j=2,M
      C(i,j) = B(i,j)
      B(i,j) = B(i-1,j-1) + B(i,j)
    END DO
  ELSE
    DO j=2,M
      B(i,j) = B(i,j-1) + B(i,j)
    END DO
  END IF
END DO
B(1,2:M) = 0
```



Loop transformations are simple transformations (split, fuse, . . .) on these groups

Projection of Data Dependencies

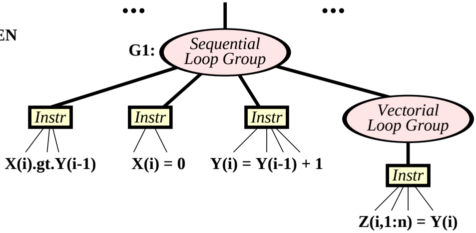
Data Dependencies between program operations are projected between groups, following simple rules:



Projected Data Dependencies drive group transfos.

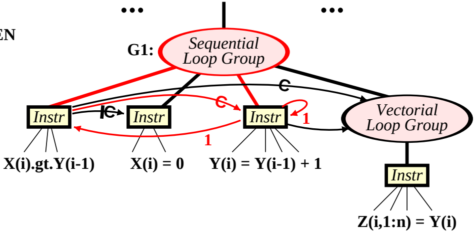
Example: Loop fission and fusion

```
DO i=1,1000
   IF (X(i).gt.Y(i-1)) THEN
      X(i) = 0
   ELSE
      Y(i) = Y(i-1) + 1
      Z(i,1:n) = Y(i)
   END IF
END DO
```



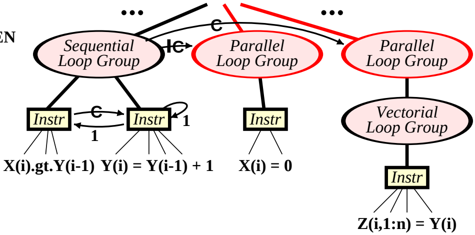
Example: Loop fission and fusion

```
DO i=1,1000
  IF (X(i).gt.Y(i-1)) THEN
    X(i) = 0
  ELSE
    Y(i) = Y(i-1) + 1
    Z(i,1:n) = Y(i)
  END IF
END DO
```



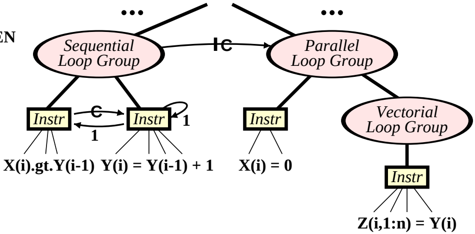
Example: Loop fission and fusion

```
DO i=1,1000
   IF (X(i).gt.Y(i-1)) THEN
      X(i) = 0
   ELSE
      Y(i) = Y(i-1) + 1
      Z(i,1:n) = Y(i)
   END IF
END DO
```



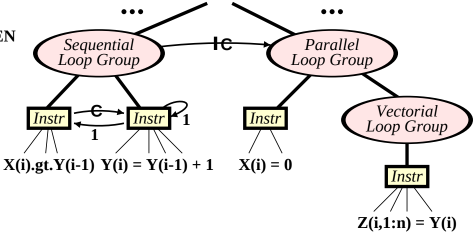
Example: Loop fission and fusion

```
DO i=1,1000
   IF (X(i).gt.Y(i-1)) THEN
      X(i) = 0
   ELSE
      Y(i) = Y(i-1) + 1
      Z(i,1:n) = Y(i)
   END IF
END DO
```



Example: Loop fission and fusion

```
DO i=1,1000
  IF (X(i).gt.Y(i-1)) THEN
    X(i) = 0
  ELSE
    Y(i) = Y(i-1) + 1
    Z(i,1:n) = Y(i)
  END IF
END DO
```

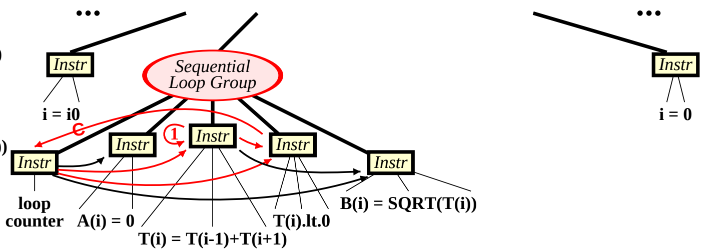



```
DO i=1,1000
  IF (X(i).le.Y(i-1)) THEN
    CTR(i) = .FALSE.
    Y(i) = Y(i-1) + 1
  END IF
END DO
PARALLEL DO i=1,1000
  IF (CTR(i)) THEN
    X(i) = 0
  ELSE
    Z(i,1:n) = Y(i)
  ENDIF
END DO
```

Example: Fission of natural loops

Nested Groups representation can handle arbitrary flow of control, and find parallelism in it.

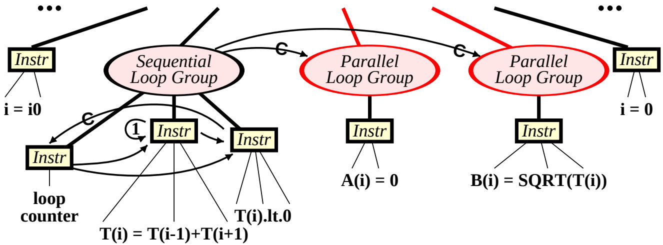
```
     i = i0
400  A(i) = 0
     T(i) = T(i-1)+T(i+1)
     IF (T(i).lt.0) THEN
        GOTO 500
     ELSE
        B(i) = SQRT(T(i))
     ENDIF
     i = i+1
     GOTO 400
500  i = 0
```



Example: Fission of natural loops

Nested Groups representation can handle arbitrary flow of control, and find parallelism in it.

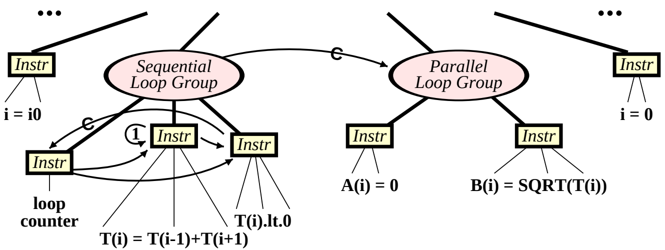
```
      i = i0
400   A(i) = 0
      T(i) = T(i-1)+T(i+1)
      IF (T(i).lt.0) THEN
         GOTO 500
      ELSE
         B(i) = SQRT(T(i))
      ENDIF
      i = i+1
      GOTO 400
500   i = 0
```



Example: Fission of natural loops

Nested Groups representation can handle arbitrary flow of control, and find parallelism in it.

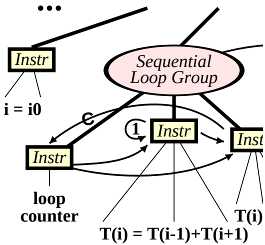
```
        i = i0
400     A(i) = 0
        T(i) = T(i-1)+T(i+1)
        IF (T(i).lt.0) THEN
           GOTO 500
        ELSE
           B(i) = SQRT(T(i))
        ENDIF
        i = i+1
        GOTO 400
500     i = 0
```



Example: Fission of natural loops

Nested Groups representation can handle arbitrary flow of control, and find parallelism in it.

```
        i = i0
400     A(i) = 0
        T(i) = T(i-1)+T(i+1)
        IF (T(i).lt.0) THEN
          GOTO 500
        ELSE
          B(i) = SQRT(T(i))
        ENDIF
        i = i+1
        GOTO 400
500     i = 0
```




```
        i = i0
        lc = 0
400     T(i) = T(i-1)+T(i+1)
        IF (T(i).lt.0) THEN
          GOTO 500
        ENDIF
        i = i+1
        lc = lc+1
        GOTO 400
500     PARALLEL DO i=i0,i0+lc-1
          A(i) = 0
          B(i) = SQRT(T(i))
        ENDDO
        A(i0+lc) = 0
        i = 0
```

Transformations captured by Nested Groups

Nested Groups capture the most useful parallelizing transformations:

- Loop fission and fusion
- Variable Expansion and Localization
- Reduction detection
- Invariant code motion
- Loop exchange
- Vectorial/Parallel distinction

Don't go back and forth to source program level.
In some (rare) situations, sophisticated (unimodular) transformations may be subcontracted to other tools ("bouclette")

Parametrization wrt the target architecture

A tactic combines the Group transformations, following a user-modifiable description of the target (*"F95"*, *"F95light"*, *"HPF"*, *"OpenMP"*, . . .):

```
 DO i = 1,n
   T(i) = Func2(T(i+2) * T(i+3)
&                    * U(i)/V(2*i)  )
 END DO
```

F95

```
tmp0(1:n) = T(3:n+2) * T(4:n+3) * U(1:n) / V(2:2*n:2)
DO i = 1,n
  T(i) = Func2(tmp0(i))
END DO
```

OpenMP

```
!$omp parallel do private(tmp0)
 DO i = 1,n
   tmp0 = Func2(T(i+2)*T(i+3)*U(i)/V(2*i))
!$omp barrier
   T(i) = tmp0
 END DO
```

HPF

```
FORALL ( i=1:n )
 T(i) = Func2(T(i+2)*T(i+3)*U(i)/V(2*i))
END FORALL
```

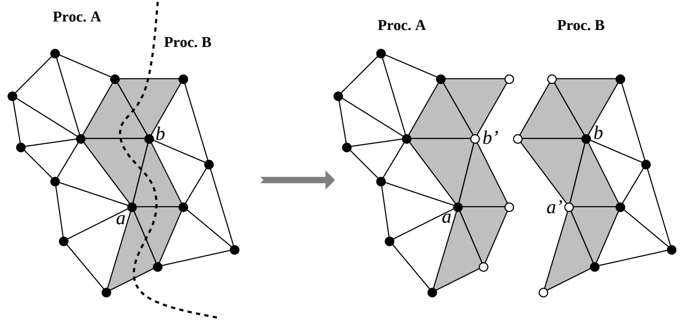
Outline

Consider a typical program on unstructured mesh:

```
New_Values = initialization
Repeat
  Old_Values = New_Values
  New_Values = 0
  Foreach Element ∈ Mesh
    gather the Old_Values from neighbors of Element
    compute the contribution of Element
    assemble into New_Values for neighbors of Element
  End Foreach
Until || New_Values − Old_Values || < ε
```

SPMD parallelization: mesh partition

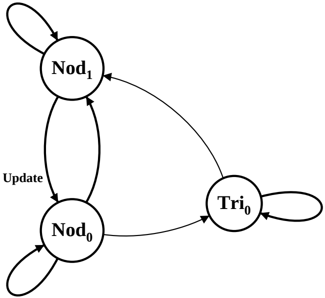
SPMD parallelization partitions the mesh wrt processors.



SPMD parallelization: boundary status

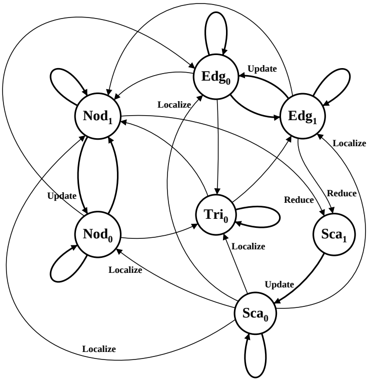
In a SPMD program, variables based on Mesh elements are duplicated on boundaries.

Values on the boundary switch from up-to-date state to out-of-date state, following well-known transitions:



SPMD parallelization: overlap automata

All possible transitions together form an finite-state automaton:



Mapping this automaton on the Dependence Graph gives the locations where synchronizations must be set.

Outline

Definition and Objectives of AD

- Given a program that computes F, AD builds a program that computes derivatives of F, analytically
- The derivatives have many uses in Scientific Computing, such as Inverse Problems and Optimization.
- For instance, AD of a Simulation program can return a gradient, used inside an Optimization program.
- Two main approaches: AD by Overloading (flexible) and AD by Program transformation (efficient).

Tangent AD by program transformation

```
SUBROUTINE FOO(v1,    v2,    v4,    p1)

  REAL v1,v2,v3,v4,p1


  v3 = 2.0*v1 + 5.0


  v4 = v3 + p1*v2/v3
END
```

Tangent AD by program transformation

```
SUBROUTINE FOO(v1,    v2,    v4,    p1)

  REAL v1,v2,v3,v4,p1

  v3d = 2.0*v1d
  v3 = 2.0*v1 + 5.0
  v4d = v3d + p1*(v2d*v3-v2*v3d)/(v3*v3)
  v4 = v3 + p1*v2/v3
END
```

Tangent AD by program transformation

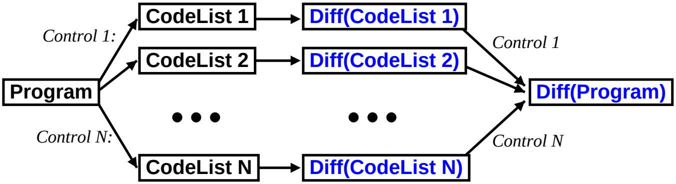
```
SUBROUTINE FOO'(v1,v1d,v2,v2d,v4,v4d,p1)
  REAL v1d,v2d,v3d,v4d
  REAL v1,v2,v3,v4,p1

  v3d = 2.0*v1d
  v3 = 2.0*v1 + 5.0
  v4d = v3d + p1*(v2d*v3-v2*v3d)/(v3*v3)
  v4 = v3 + p1*v2/v3
END
```

Just inserts "differentiated instructions" into FOO

Outline

The control flow problem / Discontinuities

For one given control, the program becomes a simple list of instructions $\Rightarrow$ AD can differentiate those.



Final program must reproduce all run-time control flows

Caution: the program is only piecewise differentiable !

AD should evaluate the "distance" to next discontinuity.

Constraints from Control switches

Consider one control switch: $\mathtt{P} : \{U; (T >= 0); D\}$
For initial input X, $T = T_x$ and constraint is:
$\delta T \geq -T_x$ or $\delta T < -T_x$

From $T = f_T \circ f_U(X)$, we get at first order

$$\delta T = f'_T \times f'_U \times \delta X$$

and the constraint on δX is thus $(K_T \cdot \delta X) \geq -1$ with:

$$K_T = f'_T \times f'_U / T_x$$

which is a gradient

Outline

1. Introduction
2. (Semi-)Automatic Parallelization
   - Parallelization and Data Dependencies
   - Parallelization strategy in PARTITA
   - SPMD parallellization
3. (Semi-)Automatic Differentiation
   - Control Flow and Discontinuities
   - Reverse AD: the quest for cheap gradients
   - Reverse AD: trajectory inversion problem
   - Static Analyses for Reverse AD
   - Applications of Data Dependencies
   - TAPENADE
4. Conclusion

AD formalization using the chain rule

- Sequences of instructions $\rightarrow$ composed functions:

$$\mathrm{P} : \{I_1; I_2; \ldots I_{p-1}; I_p;\} \rightarrow f = f_p \circ f_{p-1} \circ \cdots \circ f_1$$

- Each simple instruction

$$I_k : \quad \texttt{v4 = v3 + v2/v3}$$

  is a function $f_k : \boldsymbol{R}^q \rightarrow \boldsymbol{R}^q$ where
  - The output `v4` is built from the input `v2` and `v3`
  - All other variable are passed unchanged
- We define for short $W_0 = X$ and $W_k = f_k(W_{k-1})$. The chain rule yields:

$$f'(X) = f'_p(W_{p-1}) \times f'_{p-1}(W_{p-2}) \times \cdots \times f'_1(W_0)$$

Tangent and Reverse AD

The full $f'(X)$ is expensive and often not needed.
We'd better compute useful projections of $f'(X)$.

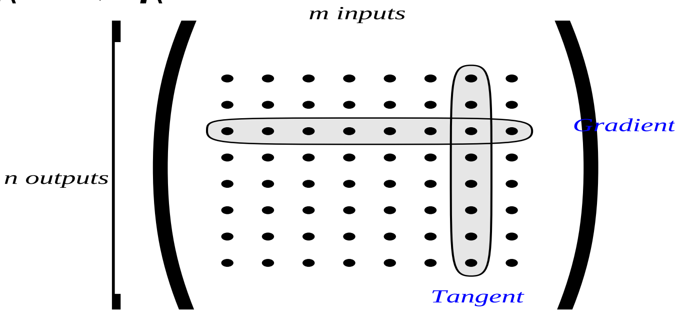
$$\text{tangent AD :}$$
$$\dot{Y} = f'(X).\dot{X} = \ f'_p(W_{p-1}).f'_{p-1}(W_{p-2}) \ldots f'_1(W_0).\dot{X}$$
$$\text{reverse AD :}$$
$$\overline{X} = f'^t(X).\overline{Y} = \ f'^t_1(W_0)\ldots . f'^t_{p-1}(W_{p-2}).f'^t_p(W_{p-1}).\overline{Y}$$

Evaluate both from right to left:
$\Rightarrow$ always matrix $\times$ vector

Theoretical cost is about 4 times the cost of P

Costs of Tangent and Reverse AD

$F \quad : \quad \boldsymbol{R}^m \quad \rightarrow \quad \boldsymbol{R}^n$



- J costs $m * 4 * \mathrm{P}$ using the tangent mode
  Good if $m <= n$
- J costs $n * 4 * \mathrm{P}$ using the reverse mode
  Good if $m >> n$ (e.g $n = 1$ in optimization)

Focus on the Reverse mode

$$\overline{X} = f'^t(X).\overline{Y} = f_1'^t(W_0).\ldots.f_{p-1}'^t(W_{p-2}).f_p'^t(W_{p-1}).\overline{Y}$$

$\overline{W}$ = $\overline{Y}$;

Focus on the Reverse mode

$$\overline{X} = f'^t(X).\overline{Y} = f_1'^t(W_0)\ldots f_{p-1}'^t(W_{p-2}).f_p'^t(W_{p-1}).\overline{Y}$$

$$\begin{array}{l} I_{p-1}\ ; \\ \overline{W} = \overline{Y}\ ; \\ \overline{W} = f_p'^t(W_{p-1}) * \overline{W}\ ; \end{array}$$

Focus on the Reverse mode

$$\overline{X} = f'^t(X).\overline{Y} = f_1'^t(W_0)\dots f_{p-1}'^t(W_{p-2}).f_p'^t(W_{p-1}).\overline{Y}$$

$$\begin{array}{l}
I_{p-2}\ ;\\
I_{p-1}\ ;\\
\overline{W} = \overline{Y}\ ;\\
\overline{W} = f_p'^t(W_{p-1}) * \overline{W}\ ;\\
\textit{Restore } W_{p-2} \textit{ before } I_{p-2}\ ;\\
\overline{W} = f_{p-1}'^t(W_{p-2}) * \overline{W}\ ;
\end{array}$$

Focus on the Reverse mode

$$\overline{X} = f'^t(X).\overline{Y} = f_1'^t(W_0).\ldots.f_{p-1}'^t(W_{p-2}).f_p'^t(W_{p-1}).\overline{Y}$$

I_1 ;
...
I_{p-2} ;
I_{p-1} ;
$\overline{W} = \overline{Y}$;
$\overline{W} = f_p'^t(W_{p-1}) * \overline{W}$;
Restore W_{p-2} *before* I_{p-2} ;
$\overline{W} = f_{p-1}'^t(W_{p-2}) * \overline{W}$;
...
Restore W_0 *before* I_1 ;
$\overline{W} = f_1'^t(W_0) * \overline{W}$;
$\overline{X} = \overline{W}$;

Instructions differentiated in the reverse order !

Reverse AD: back to the example

```
           ...
v3 = 2.0*v1 + 5.0
v4 = v3 + p1*v2/v3
           ...
```

Transposed Jacobian matrices:

$$f'^t(X) = \dots \begin{pmatrix} 1 & & 2 & \\ & 1 & & \\ & & 0 & \\ & & & 1 \end{pmatrix} \begin{pmatrix} 1 & & & 0 \\ & 1 & & \frac{p_1}{v_3} \\ & & 1 & 1 - \frac{p_1 * v_2}{v_3^2} \\ & & & 0 \end{pmatrix} \dots$$

$$\begin{aligned} & \dots \\ \overline{v}_1 &= \overline{v}_1 + 2 * \overline{v}_3 \\ \overline{v}_3 &= 0 \\ & \dots \end{aligned}$$

Reverse AD: continued example

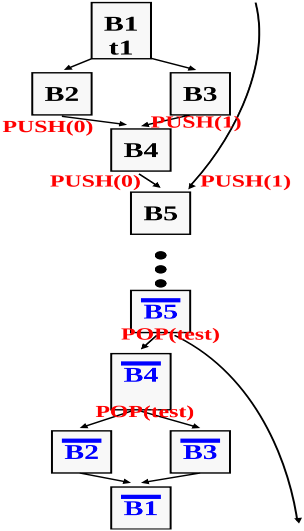
Reverse AD inverses P's control flow:

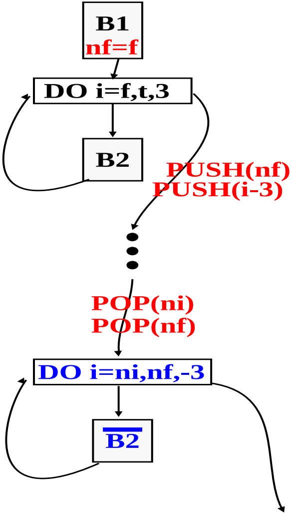
```
          ...
 v3 = 2.0*v1 + 5.0
 v4 = v3 + p1*v2/v3
          ...
          ...
........................../*restore previous state*/
 v2b = v2b + p1*v4b/v3
 v3b = v3b + (1-p1*v2/(v3*v3))*v4b
 v4b = 0.0
........................../*restore previous state*/
 v1b = v1b + 2.0*v3b
 v3b = 0.0
........................../*restore previous state*/
          ...
```

Differentiated instructions must be controlled by the inverse of P's original control flow.

Reverse AD: reversing the control flow

Flow reversal better expressed on the control flow graph:





Outline

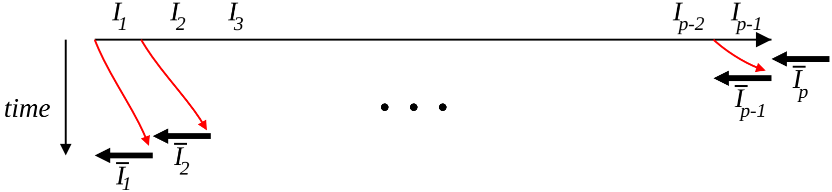
1. Introduction
2. (Semi-)Automatic Parallelization
   - Parallelization and Data Dependencies
   - Parallelization strategy in PARTITA
   - SPMD parallellization
3. (Semi-)Automatic Differentiation
   - Control Flow and Discontinuities
   - Reverse AD: the quest for cheap gradients
   - Reverse AD: trajectory inversion problem
   - Static Analyses for Reverse AD
   - Applications of Data Dependencies
   - TAPENADE
4. Conclusion

Reverse AD: Time/Memory tradeoffs

From the definition of the gradient $\overline{X}$

$$\overline{X} = f'^t(X).\overline{Y} = f_1'^t(W_0) \dots f_p'^t(W_{p-1}).\overline{Y}$$

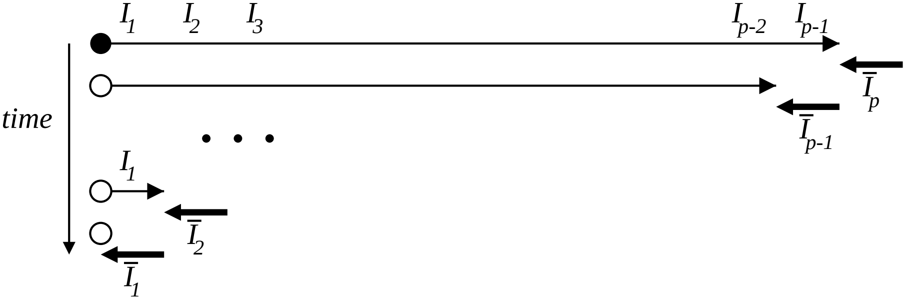
we get the general shape of reverse AD program:



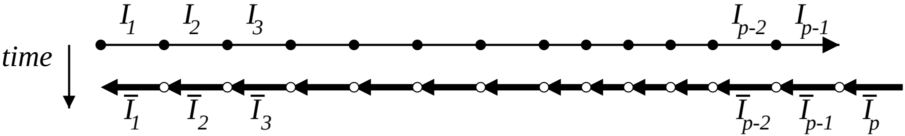
$\Rightarrow$ How can we restore previous values?

Reverse AD: Store-all *vs* Recompute-all

Recompute-all from initial state (CPU $\nearrow$; Mem $\searrow$) :

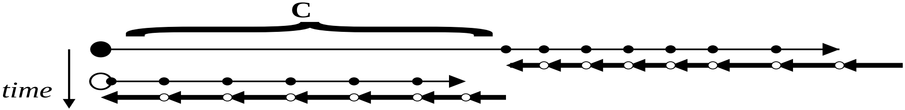


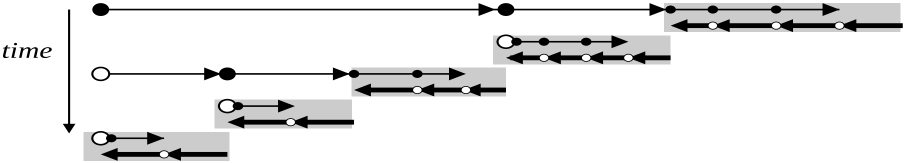
Store-all and undo progressively (CPU $\searrow$; Mem $\nearrow$) :



Reverse AD: Checkpointing (on Store-all)

On selected pieces of the program, possibly nested, don't store intermediate values and re-execute the piece, using a snapshot, when backwards sweep comes back.

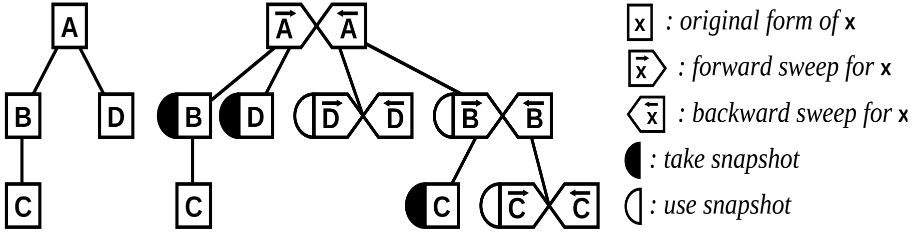




Memory and CPU grow like *log*(*size*(P))

Reverse AD: Checkpointing on calls (SA)

A classical choice: checkpoint procedure calls !



Memory and CPU grow like *log*(*size*(P)) when call tree well balanced.

Ill-balanced call trees require not checkpointing some calls

Careful analyses keep the snapshots small...

Outline

Adjoint Use, Adjoint Live, and Adjoint Out

$$\overline{I; D} = \overrightarrow{I}; \overline{D}; \overleftarrow{I} = \text{PUSH}(\mathbf{out}(I)); I; \overline{D}; \text{POP}(\mathbf{out}(I)); I'$$

is a too simple model of the SA-reverse mode, lacking:

- **Adjoint Use (TBR):** Only restore variables necessary in the sequel, i.e. $\mathbf{out}(I) \cap \mathbf{use}(I'; \overleftarrow{U})$.
- **Adjoint Live:** Execute I only if its output is needed in $\overline{D}$, i.e. $\mathbf{out}(I) \cap \mathbf{live}(\overline{D}) \neq \emptyset$ i.e. $\mathit{adj\text{-}live}(I, D)$

$$\begin{aligned} U \vdash \overline{I; D} = \; & [\text{PUSH}(\mathbf{out}(I) \cap \mathbf{use}(I'; \overleftarrow{U})); I;] \;\; \textit{if adj-live}(I, D) \\ & \{U; I\} \vdash \overline{D}; \\ & [\text{POP}(\mathbf{out}(I) \cap \mathbf{use}(I'; \overleftarrow{U}));] \;\; \textit{if adj-live}(I, D) \\ & I' \end{aligned}$$

Adjoint Live

On the complete model of SA-reverse AD, using only the standard axioms of **use**, **live**, and **out** analyses, we can derive specialized rules for reverse programs, for example:

$$\begin{cases} \mathbf{live}(\overline{\{\,\}}) & = \emptyset \\ \mathbf{live}(\overline{I;D}) & = \mathbf{live}(I') \cup (\mathbf{live}(\overline{D}) \otimes \mathbf{Dep}(I)) \end{cases}$$

We can turn them into analyses on original program P, writing $\overline{\mathbf{live}}(Z) = \mathbf{live}(\overline{Z})$, $\overleftarrow{\mathbf{use}}(Y) = \mathbf{use}(\overleftarrow{Y})$, ...

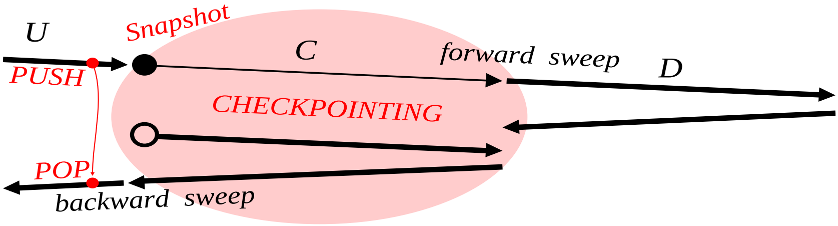
Adjoint Out and Snapshots

We similarly derive the rules for the **out** set of adjoint programs:

$$\textbf{out}(U \vdash \overline{I; D}) = \begin{cases} (\textbf{out}(I) \cup \textbf{out}(\{U; I\} \vdash \overline{D})) \setminus \\ \quad (\textbf{kill}(I) \cap \textbf{use}(I'; \overleftarrow{U})) & \textit{if adj-live}(I, D) \\ \\ \textbf{out}(\{U; I\} \vdash \overline{D}) & \textit{otherwise.} \end{cases}$$

which is used to get reduced snapshots

Reduced Snapshots



$$\textbf{Snapshot}(U, C, D) = \textbf{live}(\overline{C}) \cap (\textbf{out}(\overline{D}) \cup \textbf{out}(C))$$

used to get the variables overwritten by $\overline{C; D}$:

$$\begin{aligned}\textbf{out}(U \vdash \overline{C; D}) = \\ &(((\textbf{out}(\overline{D}) \cup \textbf{out}(C)) \setminus \textbf{Snapshot}) \cup \textbf{out}(\overline{C})) \\ &\setminus (\textbf{out}(C) \cap \textbf{use}(\overleftarrow{U}))\end{aligned}$$

Adjoint Data-Flow Analyses: results

```
subroutine FLW2D(...,g3,g3,g4,g4,rh3,rh3,rh4,rh4,...)
    ...
do iseg=nsg1,nsg2
    is1 = nubo(1,iseg)
    ...
    qs = t3(is2)*vnocl(2,iseg)
    dplim = qsor*g4(is1) + qs*g4(is2)
    rh4(is2) = rh4(is2) - dplim
    pm = pres(is1) + pres(is2)
    dplim = qsor*g3(is1)+qs*g3(is2)+pm*vnocl(2,iseg)
    rh3(is1) = rh3(is1) + dplim
    call PUSH(pm, sq)
    call LSTCHK(pm, sq)
    call POP(pm, sq)
    call LSTCHK(pm, pm, sq, sq)
    dplim = rh3(is1) - rh3(is2)
    ...
    vnocl(2,iseg) = vnocl(2,iseg)+t3(is2)*qs+t3(is1)*qsor
    t3(is1) = t3(is1) + vnocl(2,iseg)*qsor
enddo
```

Adjoint Data-Flow Analyses: results

```
subroutine FLW2D(...,g3,g3,g4,g4,rh3,rh3,rh4,rh4,...)
    ...
do iseg=nsg1,nsg2
    is1 = nubo(1,iseg)
    ...
    qs = t3(is2)*vnocl(2,iseg)
    dplim = qsor*g4(is1) + qs*g4(is2)
    rh4(is2) = rh4(is2) - dplim
    pm = pres(is1) + pres(is2)
    dplim = qsor*g3(is1)+qs*g3(is2)+pm*vnocl(2,iseg)
    rh3(is1) = rh3(is1) + dplim
    call PUSH(pm, sq)
    call LSTCHK(pm, sq)
    call POP(pm, sq)
    call LSTCHK(pm, pm, sq, sq)
    dplim = rh3(is1) - rh3(is2)
    ...
    vnocl(2,iseg) = vnocl(2,iseg)+t3(is2)*qs+t3(is1)*qsor
    t3(is1) = t3(is1) + vnocl(2,iseg)*qsor
enddo
```

Adjoint Data-Flow Analyses: measurements

Adjoint DFA progressively implemented in TAPENADE:

	ALYA (*CFD*)	UNS2D (*CFD*)	THYC (*Thermo*)	LIDAR (*Optics*)	STICS (*Agro*)
t(P):	0.85	2.39	2.67	11.22	1.80
t($\overline{\mathrm{P}}$):	5.65	29.70	11.91	23.17	42.60
new t:	4.62	24.78	10.99	22.99	35.7
improvmt:	18%	16%	8%	7%	16%
M($\overline{\mathrm{P}}$):	10.9	260	3614	16.5	456
new M:	9.4	259	3334	16.5	230
improvmt:	14%	0%	8%	0%	49%

Outline

Reverse mode (SA): the loops problem

```
DO i=1,N
  PUSH(v1); v1 = ...
  PUSH(v2); v2 = ...
  PUSH(v3); v3 = ...
  PUSH(v4); v4 = ...
  ...
ENDDO
...        ...
DO i=N,1,-1
  ...
  POP(v4); ...
  POP(v3); ...
  POP(v2); ...
  POP(v1); ...
ENDDO
```

- Stack size grows with `N`
- Can do much better on special cases:
  - iterative fixpoint computations,
  - fixed-length evolutions (treeverse),
  - parallel loops (e.g. gather-scatter)

The Dependence Graph of backward sweeps (1)

Given the dependence graph $\mathcal{G}$ of a program $\mathtt{P}$, whose nodes are the instructions I_k,

we study the dependence graph $\overleftarrow{\mathcal{G}}$ of the reverse sweep $\overleftarrow{\mathtt{P}}$, whose nodes are the I'_k (bunches of) instructions.

We observe that (roughly):

$$
\begin{array}{rcc}
I_k \;\text{ writes }\; \mathtt{v} & \Longleftrightarrow & I'_k \;\text{ writes }\; \overline{\mathtt{v}} \\
I_k \;\text{ reads }\; \mathtt{v} & \Longleftrightarrow & I'_k \;\text{ increments }\; \overline{\mathtt{v}} \\
I_k \;\text{ increments }\; \mathtt{v} & \Longleftrightarrow & I'_k \;\text{ reads }\; \overline{\mathtt{v}}
\end{array}
$$

The Dependence Graph of backward sweeps (2)

On the other hand, we refine the notion of dependence:
there is no dependence between two successive increments
so that dependences exist only between:

	w	r	i
w	*dep!*	*dep!*	*dep!*
r	*dep!*		*dep!*
i	*dep!*	*dep!*	

"read" r and "increment" i play interchangeable roles.
Therefore

$$I'_{k2} \xrightarrow{dep} I'_{k1} \quad \text{in } \overleftarrow{\mathcal{G}} \qquad \Longrightarrow \qquad I_{k1} \xrightarrow{dep} I_{k2} \quad \text{in } \mathcal{G}$$

Adjoint of Independent-Iterations loops

Thus if loop has data-independent iterations:

```
DO i=1,N
  PUSH(v1); v1 = ...
  PUSH(v2); v2 = ...
  PUSH(v3); v3 = ...
  PUSH(v4); v4 = ...
  ...
ENDDO

DO i=N,1,-1
  ...
  POP(v4); ...
  POP(v3); ...
  POP(v2); ...
  POP(v1); ...
ENDDO
```

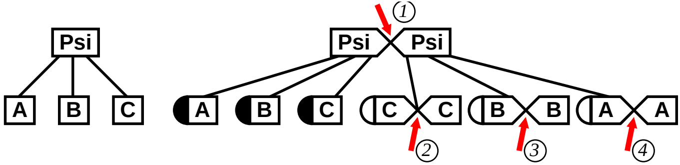
$\Longleftrightarrow$

```
DO i=1,N
  PUSH(v1); v1 = ...
  PUSH(v2); v2 = ...
  PUSH(v3); v3 = ...
  PUSH(v4); v4 = ...
  ...

  ...
  POP(v4); ...
  POP(v3); ...
  POP(v2); ...
  POP(v1); ...
ENDDO
```

which uses far less memory!

Application: gradient of an Euler flow

One time step is a sequence of large gather-scatter loops:



Tape local max	**1**	**2**	**3**	**4**
No modification:	12.40	12.37	13.60	9.66
Snapshot reduction:	1.02	0.85	9.70	9.33
//-loops improvmt:	12.38	7.98	4.10	0.02
Both:	1.02	0.61	0.22	0.02

//-loops optim and Adjoint DFA combined do the job!

Outline

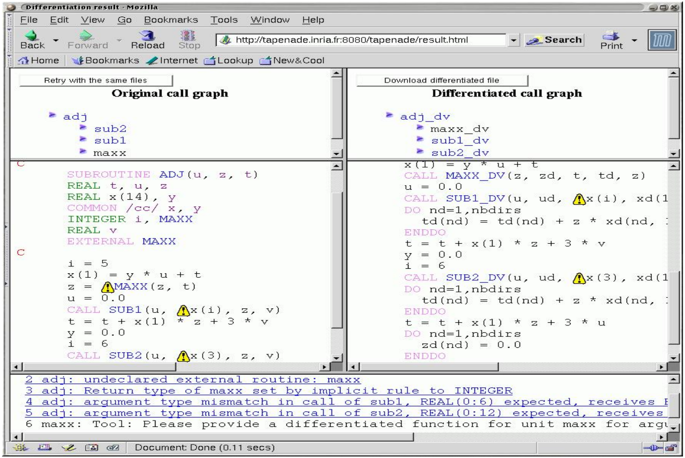
A word on TAPENADE

TAPENADE 2.1:

- AD tool, Program transformation approach
- Differentiates Fortran77, Fortran95, ...
- Tangent and Reverse mode, option for "multi-directional"
- Web server or command-line or "GUI"
- Linux, SunOS, Windows
- On-line doc, User's guide, users mailing list.

TAPENADE on the web

http://www-sop.inria.fr/tropics



4 years old, applied to industrial and academic codes:
Aeronautics, Hydrology, Chemistry, Biology, Agronomy...

Outline

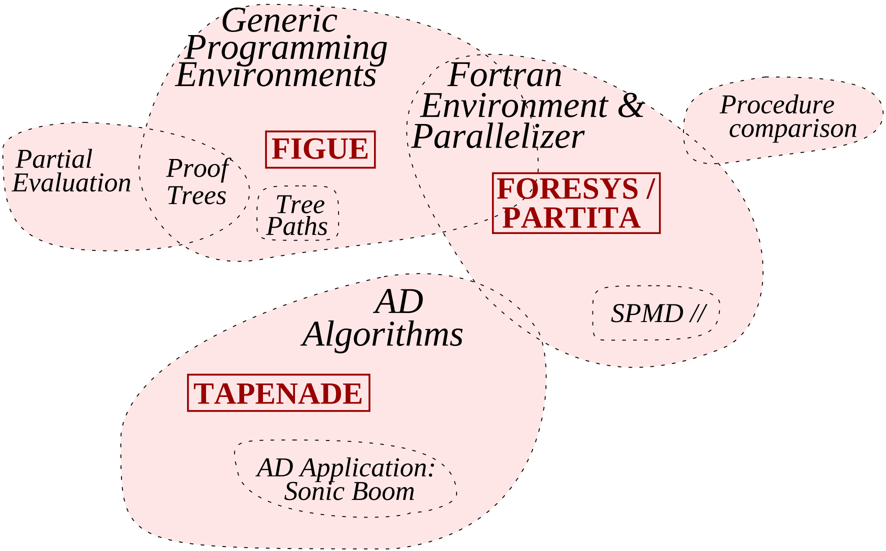
Conclusions

What I found most challenging in the latest period:

- Computer Science and Scientific Computing meet in AD, even more than in //
- Real usage requires an efficient design of tools, and precise models of the AD transformation.
- As AD model grows more complex, it takes more advantage of advanced concepts of compiler theory for a sound formalization:

  (Flow Graphs $\rightarrow$ Data Flow Eqs $\rightarrow$ Data Dependences)
- The benefit is by no means marginal !

Map of Personal Contributions



Directions for Future Work

- Unify RA and SA $\Rightarrow$ "ROSA" framework.
- Optimal Checkpointing on arbitrary control.
- Interaction with end-user + Build adapted AD for specific program types.
- Reverse AD and memory allocation,
  Rev. AD and asynchronous communications.

- Reverse AD for optimization of unsteady processes.
- Higher-Order AD for nonlinear optimization or gradient sensitivity.