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 ➔ 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
3.5: Application Tool 2: SPMD Parallelization

4.6: Application Tool: TAPENADE
1 Introduction

2 (Semi-)Automatic Parallelization
   - Parallelization and Data Dependencies
   - Parallelization strategy in PARTITA
   - SPMD parallelization

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
• 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!
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 targeted.

Classically done with 3 levels:
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

1. Introduction

2. (Semi-)Automatic Parallelization
   - Parallelization and Data Dependencies
   - Parallelization strategy in PARTITA
   - SPMD parallelization

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
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

1 Introduction

2 (Semi-)Automatic Parallelization
   - Parallelization and Data Dependencies
   - Parallelization strategy in PARTITA
   - SPMD parallelization

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
All languages guarantee some **execution order**:

- **Sequential:**
  
  \[
  \text{DO } i=4,7 \\
  A(i) = C(i) \\
  C(i+2) = B(i) \\
  \text{ENDDO}
  \]

- **Vectorial:**
  
  \[
  A(4:7) = C(4:7) \\
  C(6:9) = B(4:7)
  \]

- **Parallel:**
  
  \[
  \text{PARALLEL DO } i=4,7 \\
  A(i) = C(i) \\
  C(i+2) = B(i) \\
  \text{END PARALLEL DO}
  \]

which strongly impacts the semantics!
What can’t be changed in the execution order?

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

\begin{verbatim}
DO i=4,7
  A(i) = C(i)
  C(i+2)=B(i)
ENDDO
\end{verbatim}
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

\[ j = j' + g' \]
\[ k = k' + 6 \]

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

\[ -3 \cdot lc_1 + 3 \cdot lc_1' = 6 \]
\[ 2 \cdot lc_2 - 2 \cdot lc_2' - g = 0 \]
\[ g \geq 7 \]
\[ 2 \leq 2 \cdot lc_2 + 1 \leq m \]
\[ 0 \leq lc_1 \leq lc_1' \]
solved with an integer programming system:

\[ 2 \leq lc_1' - lc_1 \leq 2 \]
\[ -\infty < lc_2' - lc_2 \leq -4 \]
\[ \Rightarrow \text{dependency, with distance } (2, ]-\infty, -4]) \]
Static Data-Flow analyses reduce dependencies

- The fewer dependencies, the better
- Data-Flow analyses help especially:
  - constraint propagation
  - induction detection
  - used/killed variables
For (nested) loops, cycles in the dependence graph correspond to non-parallel parts.

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 parallelization

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
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

Sequential Loop Group

G1:

Vectorial Loop Group

Instr

X(i).gt.Y(i-1)  X(i) = 0  Y(i) = Y(i-1) + 1
Z(i,1:n) = Y(i)
Example: Loop fission and fusion

\[
\begin{align*}
\text{DO } & \text{i=1,1000} \\
\text{IF (X(i).gt.Y(i-1)) THEN} & \\
& \text{X(i) = 0} \\
\text{ELSE} & \\
& \text{Y(i) = Y(i-1) + 1} \\
& \text{Z(i,1:n) = Y(i)} \\
\text{END IF} & \\
\text{END DO}
\end{align*}
\]
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

Sequential Loop Group

Parallel Loop Group

Vectorial Loop Group

Hascoët (INRIA Sophia Antipolis) From Parallelization to Differentiation January 28, 2005 20 / 67
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

Sequential Loop Group

Vectorial Loop Group

Parallel Loop Group

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

Example: Fission of natural loops

\[ i = i_0 \]

\[ 400 \]

\[ A(i) = 0 \]

\[ T(i) = T(i-1)+T(i+1) \]

\[ \text{IF} (T(i) < 0) \text{ THEN} \]

\[ \text{GOTO} 500 \]

\[ \text{ELSE} \]

\[ B(i) = \sqrt{T(i)} \]

\[ \text{ENDIF} \]

\[ i = i+1 \]

\[ \text{GOTO} 400 \]

\[ 500 \]

\[ i = 0 \]

Hascoët (INRIA Sophia Antipolis) From Parallelization to Differentiation January 28, 2005 21 / 67
Nested Groups representation can handle arbitrary flow of control, and find parallelism in it.

Example: Fission of natural loops

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

\begin{align*}
i & = i_0 \\
400 \quad & A(i) = 0 \\
& T(i) = T(i-1)+T(i+1) \\
& \text{IF (} T(i) \text{.lt.0) THEN} \\
& \quad \text{GOTO 500} \\
& \text{ELSE} \\
& \quad B(i) = \sqrt{T(i)} \\
& \quad \text{ENDIF} \\
& i = i+1 \\
& \text{GOTO 400} \\
500 \quad & i = 0
\end{align*}

\begin{align*}
i & = i_0 \\
lc & = 0 \\
400 \quad & T(i) = T(i-1)+T(i+1) \\
& \text{IF (} T(i) \text{.lt.0) THEN} \\
& \quad \text{GOTO 500} \\
& \quad \text{ENDIF} \\
& i = i+1 \\
lc & = lc+1 \\
& \text{GOTO 400} \\
500 \quad & \text{PARALLEL DO i=i0,i0+lc-1} \\
& \quad A(i) = 0 \\
& \quad B(i) = \sqrt{T(i)} \\
& \quad \text{ENDDO} \\
& A(i0+lc) = 0 \\
& i = 0
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")
A tactic combines the Group transformations, following a user-modifiable description of the target ("F95", "F95light", "HPF", "OpenMP", . . .):

```plaintext
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

1 Introduction

2 (Semi-)Automatic Parallelization
   - Parallelization and Data Dependencies
   - Parallelization strategy in PARTITA
   - SPMD parallelization

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
Consider a typical program on unstructured mesh:

\[
\text{New\_Values} = \text{initialization}
\]

\begin{verbatim}
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 \[\| \text{New\_Values} - \text{Old\_Values} \| < \varepsilon\]
SPMD parallelization partitions the mesh wrt processors.
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:
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

1. Introduction

2. (Semi-)Automatic Parallelization
   - Parallelization and Data Dependencies
   - Parallelization strategy in PARTITA
   - SPMD parallelization

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

REAL v1, v2, v3, v4, p1

v3 = 2.0*v1 + 5.0

v4 = v3 + p1*v2/v3

END
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
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
1 Introduction

2 (Semi-)Automatic Parallelization
   - Parallelization and Data Dependencies
   - Parallelization strategy in PARTITA
   - SPMD parallelization

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
For one given control, the program becomes a simple list of instructions ⇒ 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.
Consider one control switch: $P : \{ U; (T \geq 0); D \}$

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 $\delta X$ 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 parallelization

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 → composed functions:
  \[ P : \{ I_1; I_2; \ldots; I_{p-1}; I_p; \} \rightarrow f = f_p \circ f_{p-1} \circ \cdots \circ f_1 \]

- Each simple instruction
  \[ I_k : \quad v_4 = v_3 + v_2/v_3 \]
  is a function \( f_k : \mathbb{R}^q \rightarrow \mathbb{R}^q \) where
  - The output \( v_4 \) is built from the input \( v_2 \) and \( v_3 \)
  - 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:
  \[ f'(X) = f'_p(W_{p-1}) \times f'_{p-1}(W_{p-2}) \times \cdots \times f'_1(W_0) \]
The full $f'(X)$ is expensive and often not needed. We’d better compute useful projections of $f'(X)$.

**tangent AD:**

$$\dot{Y} = f'(X).\dot{X} = f'_p(W_{p-1}).f'_{p-1}(W_{p-2}) \ldots f'_1(W_0).\dot{X}$$

**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

\[ F : \mathbb{R}^m \rightarrow \mathbb{R}^n \]

- \( J \) costs \( m \times 4 \times P \) using the tangent mode.
  Good if \( m \leq n \)

- \( J \) costs \( n \times 4 \times P \) using the reverse mode.
  Good if \( m >> n \) (e.g. \( n = 1 \) in optimization)
Focus on the Reverse mode

\[ \bar{X} = f'^t(X), \bar{Y} = f'_1^t(W_0) \ldots f'_{p-1}^t(W_{p-2})f'_p(W_{p-1}) \bar{Y} \]

\[ \bar{W} = \bar{Y} ; \]
Focus on the Reverse mode

\[ \overline{X} = f'^t(X) \cdot \overline{Y} = f'_1^t(W_0) \cdots f'_{p-1}^t(W_{p-2}) \cdot f'_p^t(W_{p-1}) \cdot \overline{Y} \]

\[ l_{p-1}; \]
\[ \overline{W} = \overline{Y}; \]
\[ \overline{W} = f'_{p-1}^t(W_{p-1}) \ast \overline{W}; \]
Focus on the Reverse mode

\[ \overline{X} = f'^t(X). \overline{Y} = f'_1t(W_0) \ldots f'_{p-1}(W_{p-2}). f'_p(W_{p-1}). \overline{Y} \]

\[ I_{p-2} ; \]
\[ I_{p-1} ; \]
\[ \overline{W} = \overline{Y} ; \]
\[ \overline{W} = f'_p(W_{p-1}) * \overline{W} ; \]

*Restore \( W_{p-2} \) before \( I_{p-2} \) ;*
\[ \overline{W} = f'_{p-1}(W_{p-2}) * \overline{W} ; \]
Focus on the Reverse mode

\[
\bar{X} = f'^t(X). \bar{Y} = f'_1^t(W_0) \ldots f'_{p-1}(W_{p-2}). f'_p(W_{p-1}). \bar{Y}
\]

\[
l_1 ;
\]
\[
\ldots
\]
\[
l_{p-2} ;
\]
\[
l_{p-1} ;
\]
\[
\bar{W} = \bar{Y} ;
\]
\[
\bar{W} = f'^t(W_{p-1}) \ast \bar{W} ;
\]
\[
\text{Restore } W_{p-2} \text{ before } l_{p-2} ;
\]
\[
\bar{W} = f'^t(W_{p-2}) \ast \bar{W} ;
\]
\[
\ldots
\]
\[
\text{Restore } W_0 \text{ before } l_1 ;
\]
\[
\bar{W} = f'_1^t(W_0) \ast \bar{W} ;
\]
\[
\bar{X} = \bar{W} ;
\]

Instructions differentiated in the reverse order!
Reverse AD: back to the example

\[ v_3 = 2.0 \times v_1 + 5.0 \]
\[ v_4 = v_3 + \frac{p_1 \times v_2}{v_3} \]

Transposed Jacobian matrices:

\[
f^{t'}(X) = \ldots \begin{pmatrix} 1 & 2 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & \frac{p_1}{v_3} \\ 1 & \frac{p_1 \times v_2}{v_3^2} \\ 0 & 0 \end{pmatrix} \ldots
\]

\[ \bar{v}_1 = \bar{v}_1 + 2 \times \bar{v}_3 \]
\[ \bar{v}_3 = 0 \]
\[ \ldots \]
Reverse AD inverses $P$’s control flow:

\[
\begin{align*}
\ldots \\
v3 &= 2.0 \times v1 + 5.0 \\
v4 &= v3 + p1 \times v2/v3 \\
\ldots \\
......................../*\text{restore previous state}*/ \\
v2b &= v2b + p1 \times v4b/v3 \\
v3b &= v3b + (1 - p1 \times v2/(v3 \times v3)) \times v4b \\
v4b &= 0.0 \\
......................../*\text{restore previous state}*/ \\
v1b &= v1b + 2.0 \times v3b \\
v3b &= 0.0 \\
......................../*\text{restore previous state}*/ \\
\ldots
\end{align*}
\]

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 parallelization

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 $\overline{X}$

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

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 ↗; Mem ↘):

Store-all and undo progressively (CPU ↘; Mem ↗):
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 $\log(\text{size}(P))$
A classical choice: checkpoint procedure calls!

Memory and CPU grow like $\log(\text{size}(P))$ when call tree well balanced.

Ill-balanced call trees require not checkpointing some calls

Careful analyses keep the snapshots small...
Outline

1. Introduction

2. (Semi-)Automatic Parallelization
   - Parallelization and Data Dependencies
   - Parallelization strategy in PARTITA
   - SPMD parallelization

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
Adjoint Use, Adjoint Live, and Adjoint Out

$$I; D = I; D; \uparrow I = \text{PUSH}\left(\text{out}(I)\right); I; D; \text{POP}\left(\text{out}(I)\right); I'$$

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

- **Adjoint Use** (TBR): Only restore variables necessary in the sequel, i.e. $$\text{out}(I) \cap \text{use}(I'; \uparrow U)$$.

- **Adjoint Live**: Execute $$I$$ only if its output is needed in $$D$$, i.e. $$\text{out}(I) \cap \text{live}(D) \neq \emptyset$$ i.e. $$\text{adj-live}(I, D)$$

$$U \vdash I; D = \left[\text{PUSH}\left(\text{out}(I) \cap \text{use}(I'; \uparrow U)\right); I; \right] \text{ if } \text{adj-live}(I, D)$$
$$\{U; I\} \vdash \overline{D};$$
$$\left[\text{POP}\left(\text{out}(I) \cap \text{use}(I'; \uparrow U)\right)\right] \text{ if } \text{adj-live}(I, D)$$

$$I'$$
On the complete model of SA-reverse AD, using only the standard axioms of \texttt{use}, \texttt{live}, and \texttt{out} analyses, we can derive specialized rules for reverse programs, for example:

\[
\begin{align*}
\text{live}(&\{\}\}) = \emptyset \\
\text{live}(I; D) &= \text{live}(I') \cup (\text{live}(D) \otimes \text{Dep}(I))
\end{align*}
\]

We can turn them into analyses on original program $P$, writing $\overline{\text{live}}(Z) = \text{live}(\overline{Z})$, $\overline{\text{use}}(Y) = \text{use}(\overline{Y})$, ...
We similarly derive the rules for the `out` set of adjoint programs:

\[
\text{out}(U \vdash I; D) = \begin{cases} 
\text{out}(I) \cup \text{out}(\{U; I\} \vdash \overline{D}) \\
(kill(I) \cap \text{use}(I'; \overline{U})) \ & \text{if} \ \text{adj-live}(I, D) \\
\text{out}(\{U; I\} \vdash \overline{D}) \ & \text{otherwise}.
\end{cases}
\]

which is used to get reduced snapshots.
**Reduced Snapshots**

\[
\text{Snapshot}(U, C, D) = \text{live}(\overline{C}) \cap (\text{out}(\overline{D}) \cup \text{out}(C))
\]

used to get the variables overwritten by \( \overline{C}, \overline{D} \):

\[
\text{out}(U \vdash \overline{C}, \overline{D}) =
(\left((\text{out}(\overline{D}) \cup \text{out}(C)) \setminus \text{Snapshot}\right) \cup \text{out}(\overline{C}))
\setminus (\text{out}(C) \cap \text{use}(\overline{U}))
\]
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
subroutine FLW2D(...,g3,g3,g4,g4,rh3,rh3,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

endo
Adjoint Data-Flow Analyses: measurements

Adjoint DFA progressively implemented in TAPENADE:

<table>
<thead>
<tr>
<th></th>
<th>ALYA (CFD)</th>
<th>UNS2D (CFD)</th>
<th>THYC (Thermo)</th>
<th>LIDAR (Optics)</th>
<th>STICS (Agro)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t(P) )</td>
<td>0.85</td>
<td>2.39</td>
<td>2.67</td>
<td>11.22</td>
<td>1.80</td>
</tr>
<tr>
<td>( t(\bar{P}) )</td>
<td>5.65</td>
<td>29.70</td>
<td>11.91</td>
<td>23.17</td>
<td>42.60</td>
</tr>
<tr>
<td>new ( t )</td>
<td>4.62</td>
<td>24.78</td>
<td>10.99</td>
<td>22.99</td>
<td>35.7</td>
</tr>
<tr>
<td>improvmt:</td>
<td>18%</td>
<td>16%</td>
<td>8%</td>
<td>7%</td>
<td>16%</td>
</tr>
<tr>
<td>( M(P) )</td>
<td>10.9</td>
<td>260</td>
<td>3614</td>
<td>16.5</td>
<td>456</td>
</tr>
<tr>
<td>new ( M )</td>
<td>9.4</td>
<td>259</td>
<td>3334</td>
<td>16.5</td>
<td>230</td>
</tr>
<tr>
<td>improvmt:</td>
<td>14%</td>
<td>0%</td>
<td>8%</td>
<td>0%</td>
<td>49%</td>
</tr>
</tbody>
</table>
Outline

1 Introduction

2 (Semi-)Automatic Parallelization
   - Parallelization and Data Dependencies
   - Parallelization strategy in PARTITA
   - SPMD parallelization

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 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 $P$, whose nodes are the instructions $I_k$,

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

We observe that (roughly):

- $I_k$ writes $v$ $\iff$ $I_k'$ writes $\overline{v}$
- $I_k$ reads $v$ $\iff$ $I_k'$ increments $\overline{v}$
- $I_k$ increments $v$ $\iff$ $I_k'$ reads $\overline{v}$
On the other hand, we refine the notion of dependence: there is no dependence between two successive increments so that dependences exist only between:

```
<table>
<thead>
<tr>
<th></th>
<th>①</th>
<th>②</th>
<th>③</th>
</tr>
</thead>
<tbody>
<tr>
<td>①</td>
<td>dep!</td>
<td>dep!</td>
<td>dep!</td>
</tr>
<tr>
<td>②</td>
<td>dep!</td>
<td></td>
<td>dep!</td>
</tr>
<tr>
<td>③</td>
<td>dep!</td>
<td>dep!</td>
<td></td>
</tr>
</tbody>
</table>
```

“read” ② and “increment” ③ play interchangeable roles. Therefore

\[ I'_k_2 \xrightarrow{dep} I'_k_1 \quad \text{in} \quad \overrightarrow{G} \quad \implies \quad I_k_1 \xrightarrow{dep} I_k_2 \quad \text{in} \quad \overrightarrow{G} \]
Thus if loop has data-independent iterations:

\[
\begin{align*}
&\text{DO } i=1,N \\
&\quad \text{PUSH}(v1); \quad v1 = \ldots \\
&\quad \text{PUSH}(v2); \quad v2 = \ldots \\
&\quad \text{PUSH}(v3); \quad v3 = \ldots \\
&\quad \text{PUSH}(v4); \quad v4 = \ldots \\
&\quad \ldots \\
&\quad \text{ENDDO} \\
&\text{DO } i=N,1,-1 \\
&\quad \ldots \\
&\quad \text{POP}(v4); \ldots \\
&\quad \text{POP}(v3); \ldots \\
&\quad \text{POP}(v2); \ldots \\
&\quad \text{POP}(v1); \ldots \\
&\quad \text{ENDDO}
\end{align*}
\]

\[\Rightarrow\]

\[
\begin{align*}
&\text{DO } i=1,N \\
&\quad \text{PUSH}(v1); \quad v1 = \ldots \\
&\quad \text{PUSH}(v2); \quad v2 = \ldots \\
&\quad \text{PUSH}(v3); \quad v3 = \ldots \\
&\quad \text{PUSH}(v4); \quad v4 = \ldots \\
&\quad \ldots \\
&\quad \text{ENDDO} \\
&\text{DO } i=N,1,-1 \\
&\quad \ldots \\
&\quad \text{POP}(v4); \ldots \\
&\quad \text{POP}(v3); \ldots \\
&\quad \text{POP}(v2); \ldots \\
&\quad \text{POP}(v1); \ldots \\
&\quad \text{ENDDO}
\end{align*}
\]

which uses far less memory!
Application: gradient of an Euler flow

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

<table>
<thead>
<tr>
<th>Tape local max</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>No modification:</td>
<td>12.40</td>
<td>12.37</td>
<td>13.60</td>
<td>9.66</td>
</tr>
<tr>
<td>Snapshot reduction:</td>
<td>1.02</td>
<td>0.85</td>
<td>9.70</td>
<td>9.33</td>
</tr>
<tr>
<td>II-loops improvmt:</td>
<td>12.38</td>
<td>7.98</td>
<td>4.10</td>
<td>0.02</td>
</tr>
<tr>
<td>Both:</td>
<td>1.02</td>
<td>0.61</td>
<td>0.22</td>
<td>0.02</td>
</tr>
</tbody>
</table>

II-loops optim and Adjoint DFA combined do the job!
Outline

1 Introduction

2 (Semi-)Automatic Parallelization
   - Parallelization and Data Dependencies
   - Parallelization strategy in PARTITA
   - SPMD parallelization

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
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...
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

- Generic Programming Environments
- Proof Trees
- Partial Evaluation
- FIGUE
- Tree Paths
- Fortran Environment & Parallelizer
- Procedure comparison
- FORESYS / PARTITA
- SPMD //
- AD Algorithms
- TAPENADE
- AD Application: Sonic Boom

Hascoët (INRIA Sophia Antipolis)
Directions for Future Work

- Unify RA and SA ⇒ “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.