OpenAD Tutorial 2/2

Jean Utke[1]

[1]University of Chicago and Argonne National Laboratory

CMG Workshop
Sept. 9/10, 2009

outline

- part 1
  - motivation
  - AD basics and examples
  - reversal schemes and checkpointing
  - concerns for the AD user (model developer)
- part 2
  - non smooth models
  - checkpointing with Revolve
  - adjoinable MPI

is the model smooth?

- `y=abs(x);` gives a kink

is the model smooth?

- `y=abs(x);` gives a kink
- `y=(x>0)?3*x:2*x+2;` gives a discontinuity

is the model smooth?

- `y=abs(x);` gives a kink
- `y=(x>0)?3*x:2*x+2;` gives a discontinuity
- `y=floor(x);` same

is the model smooth?

- `y=abs(x);` gives a kink
- `y=(x>0)?3*x:2*x+2;` gives a discontinuity
- `y=floor(x);` same
- `Y=REAL(Z);` what about `IMAG(Z)`

is the model smooth?

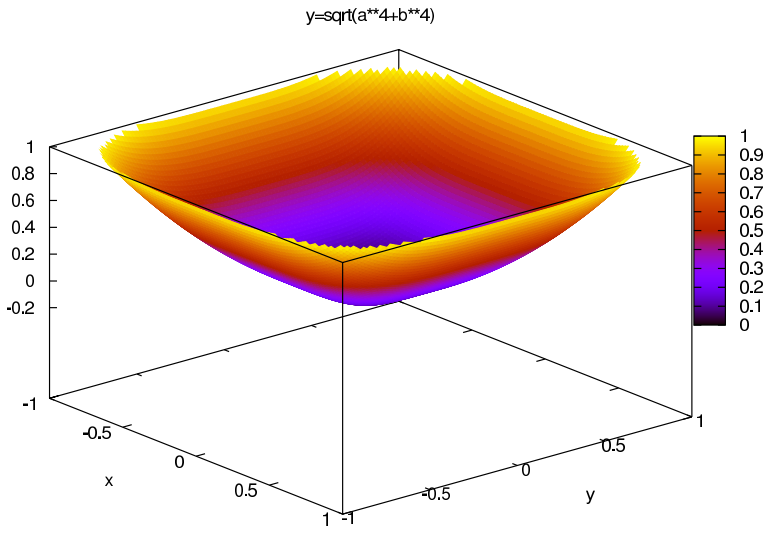
- `y=abs(x);` gives a kink
- `y=(x>0)?3*x:2*x+2;` gives a discontinuity
- `y=floor(x);` same
- `Y=REAL(Z);` what about `IMAG(Z)`
-
```
if (a == 1.0)
   y = b;
else if (a == 0.0) then
   y = 0;
else
   y = a*b;
```

is the model smooth?

- `y=abs(x);` gives a kink
- `y=(x>0)?3*x:2*x+2;` gives a discontinuity
- `y=floor(x);` same
- `Y=REAL(Z);` what about `IMAG(Z)`
-
```
if (a == 1.0)
   y = b;
else if (a == 0.0) then
   y = 0;
else
   y = a*b;
```
  intended: $\dot{y}$=a*$\dot{b}$+b*$\dot{a}$

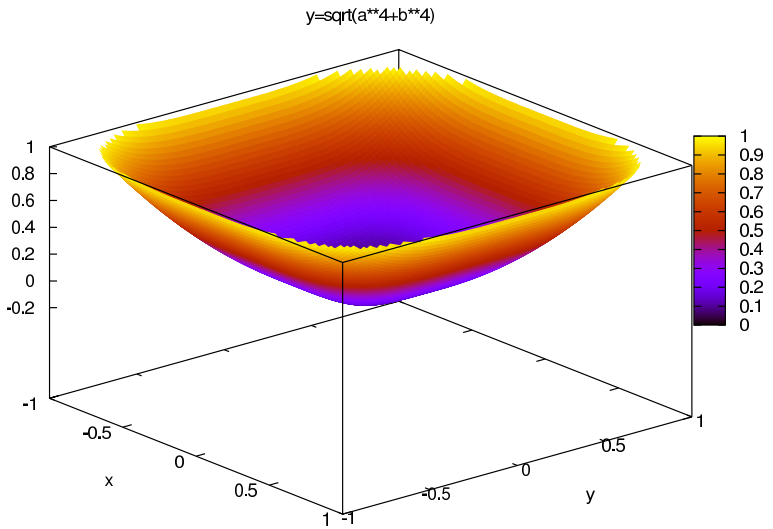
is the model smooth?

- `y=abs(x);` gives a kink
- `y=(x>0)?3*x:2*x+2;` gives a discontinuity
- `y=floor(x);` same
- `Y=REAL(Z);` what about `IMAG(Z)`
-
```
if (a == 1.0)
   y = b;
else if (a == 0.0) then
   y = 0;
else
   y = a*b;
```
  intended: $\dot{y}$=a*$\dot{b}$+b*$\dot{a}$
- `y = sqrt(a**4 + b**4);`



is the model smooth?

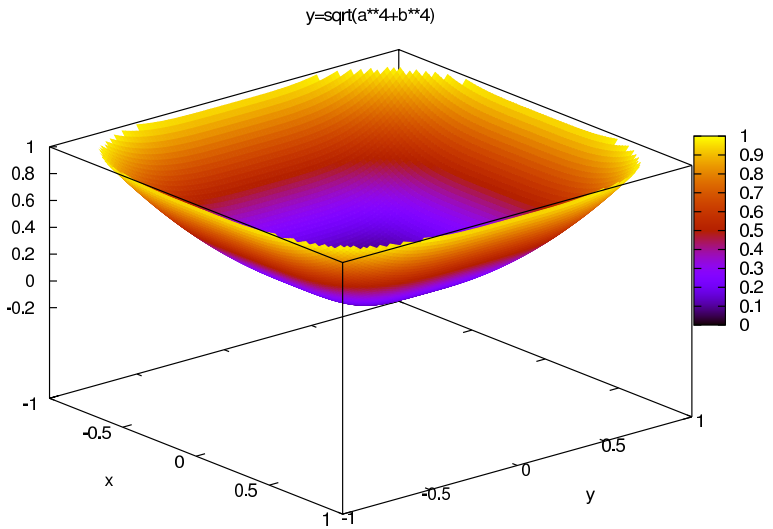
- `y=abs(x);` gives a kink
- `y=(x>0)?3*x:2*x+2;` gives a discontinuity
- `y=floor(x);` same
- `Y=REAL(Z);` what about `IMAG(Z)`
-
```
if (a == 1.0)
   y = b;
else if (a == 0.0) then
   y = 0;
else
   y = a*b;
```
  intended: $\dot{y}$=a*$\dot{b}$+b*$\dot{a}$
- `y = sqrt(a**4 + b**4);`



AD does not perform algebraic simplification, i.e. for a,b $\rightarrow$ 0 it does $\left(\frac{d\sqrt{t}}{dt}\right) \overset{t\rightarrow+0}{=} +\infty$.

is the model smooth?

- `y=abs(x);` gives a kink
- `y=(x>0)?3*x:2*x+2;` gives a discontinuity
- `y=floor(x);` same
- `Y=REAL(Z);` what about `IMAG(Z)`
-
```
if (a == 1.0)
   y = b;
else if (a == 0.0) then
   y = 0;
else
   y = a*b;
```
  intended: $\dot{y}$=a*$\dot{b}$+b*$\dot{a}$
- `y = sqrt(a**4 + b**4);`



AD does not perform algebraic simplification, i.e. for a,b $\rightarrow$ 0 it does $\left(\frac{d\sqrt{t}}{dt}\right) \overset{t\rightarrow+0}{=} +\infty$.

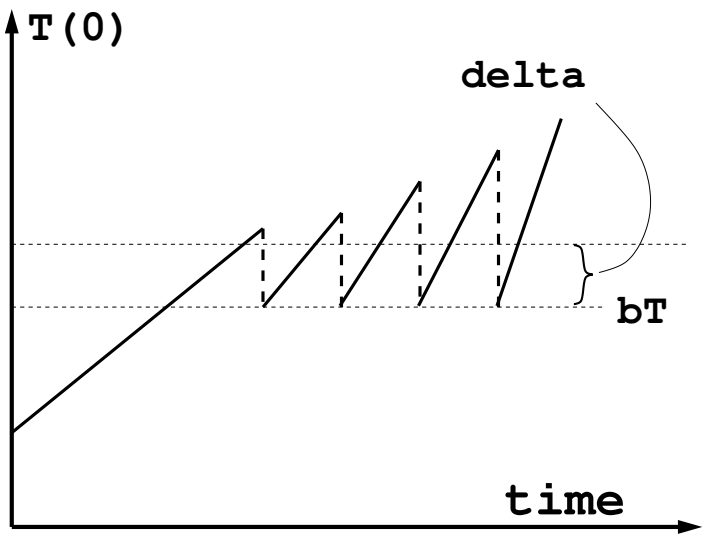
AD computes derivatives of programs(!)

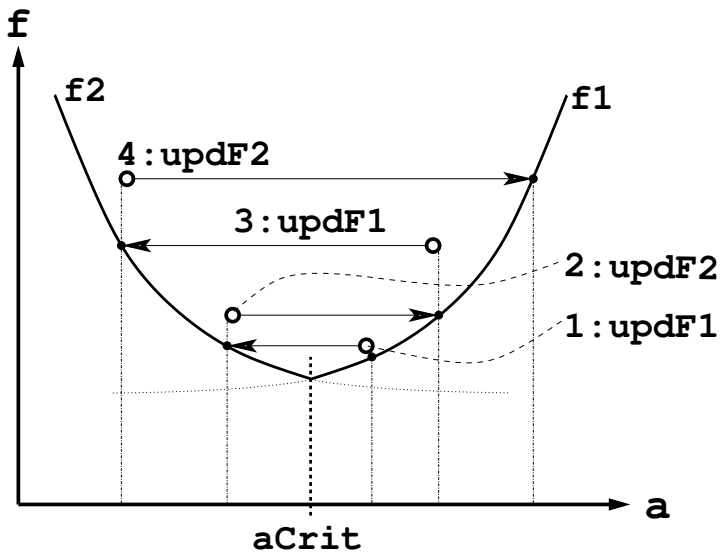
know your application e.g. fix point iteration, self adjoint, step size computation, convergence criteria

nonsmooth models

observed:

- `INF`, `NaN`
- oscillating derivatives (may be glossed over by FD) or derivatives growing out of bounds





nonsmooth models II

- blame AD tool - verification problem

nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)

nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD

nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool

nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool
- blame code, model's built-in numerical approximations, external optimization scheme or inherent in the physics?

nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool
- blame code, model's built-in numerical approximations, external optimization scheme or inherent in the physics?
- higher order models in mech. engineering, beam physics, AtomFT explicit g-stop facility for ODEs, DAEs

nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool
- blame code, model's built-in numerical approximations, external optimization scheme or inherent in the physics?
- higher order models in mech. engineering, beam physics, AtomFT explicit g-stop facility for ODEs, DAEs
- what to do about first order

nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool
- blame code, model's built-in numerical approximations, external optimization scheme or inherent in the physics?
- higher order models in mech. engineering, beam physics, AtomFT explicit g-stop facility for ODEs, DAEs
- what to do about first order
  - Adifor: optionally catches intrinsic problems via exception handling

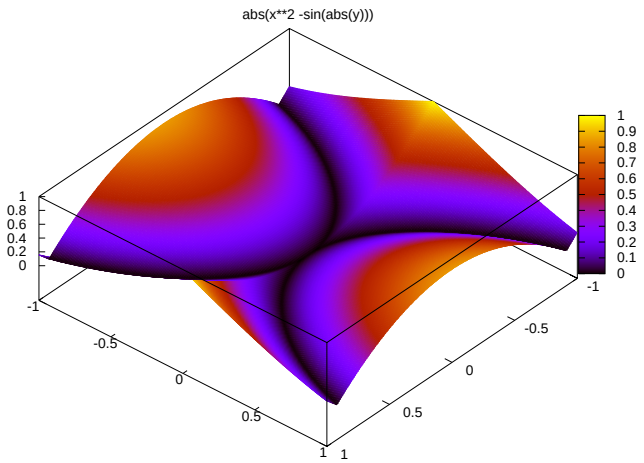
nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool
- blame code, model's built-in numerical approximations, external optimization scheme or inherent in the physics?
- higher order models in mech. engineering, beam physics, AtomFT explicit g-stop facility for ODEs, DAEs
- what to do about first order
  - Adifor: optionally catches intrinsic problems via exception handling
  - Adol-C: tape verification and intrinsic handling

nonsmooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot product check)
  - compare to FD
  - compare to other AD tool
- blame code, model's built-in numerical approximations, external optimization scheme or inherent in the physics?
- higher order models in mech. engineering, beam physics, AtomFT explicit g-stop facility for ODEs, DAEs
- what to do about first order
  - Adifor: optionally catches intrinsic problems via exception handling
  - Adol-C: tape verification and intrinsic handling
  - OpenAD (comparative tracing)

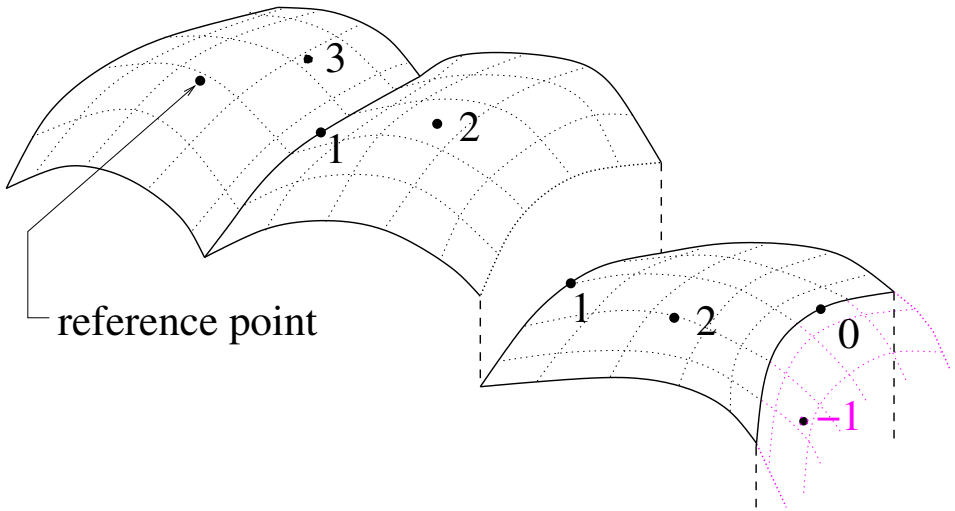
differentiability



piecewise differentiable function:
$|x^2 - sin(|y|)|$
is (locally) Lipschitz continuous; almost everywhere differentiable (except on the 6 critical paths)

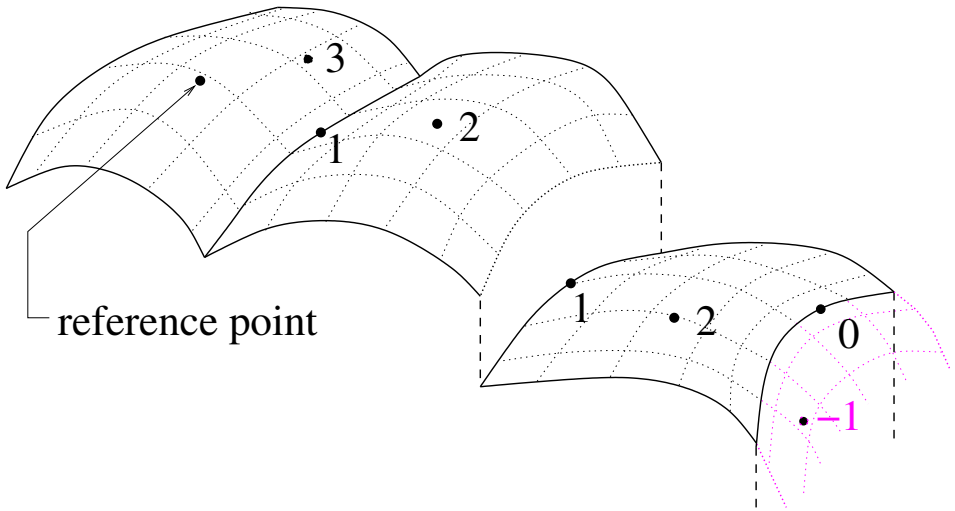
- Gâteaux: if $\exists \quad df(x,\dot{x}) = \lim_{\tau \to 0} \frac{f(x+\tau\dot{x})-f(x)}{\tau}$ for all directions $\dot{x}$
- Bouligand: Lipschitz continuous and Gâteaux
- Fréchet: $df(.,\dot{x})$ continuous for every fixed $\dot{x}$ ... not generally
- in practice: often benign behavior, directional derivative exists and is an element of the generalized gradient.

case distinction



case distinction

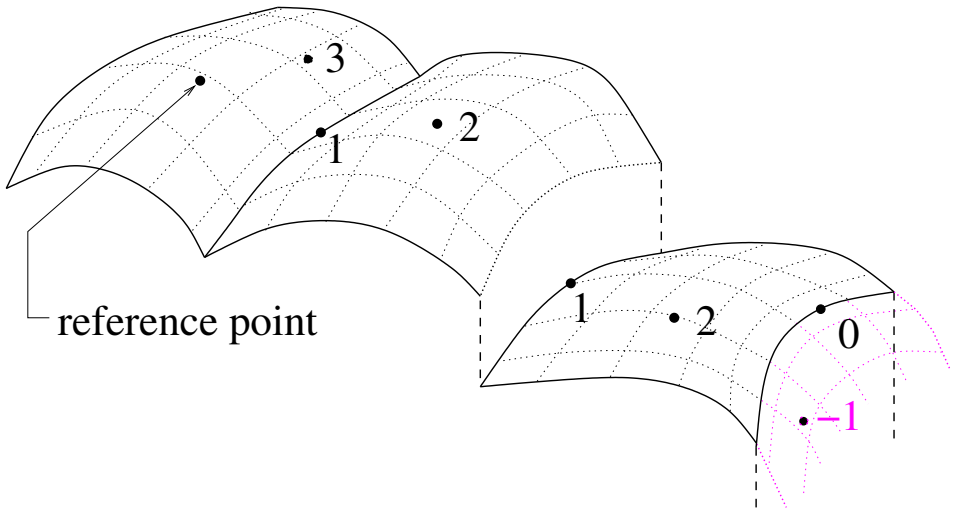
3 locally analytic



case distinction

3 locally analytic

2 locally analytic but crossed a (potential) kink (`min`,`max`,`abs`,`...`) or discontinuity (`ceil`,`...`) [for source transformation: also different control flow]

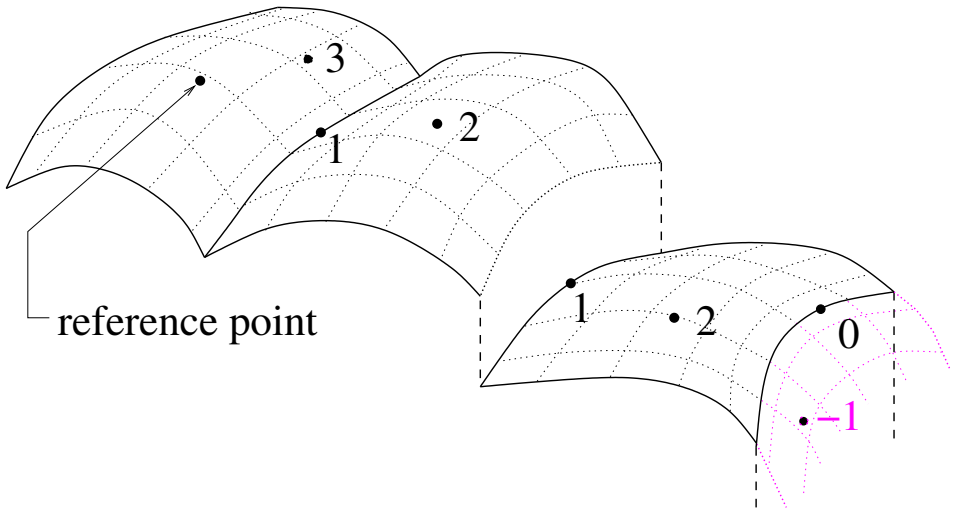


case distinction

3 locally analytic

2 locally analytic but crossed a (potential) kink (`min`,`max`,`abs`,`...`) or discontinuity (`ceil`,`...`) [for source transformation: also different control flow]

1 we are exactly at a (potential) kink, discontinuity



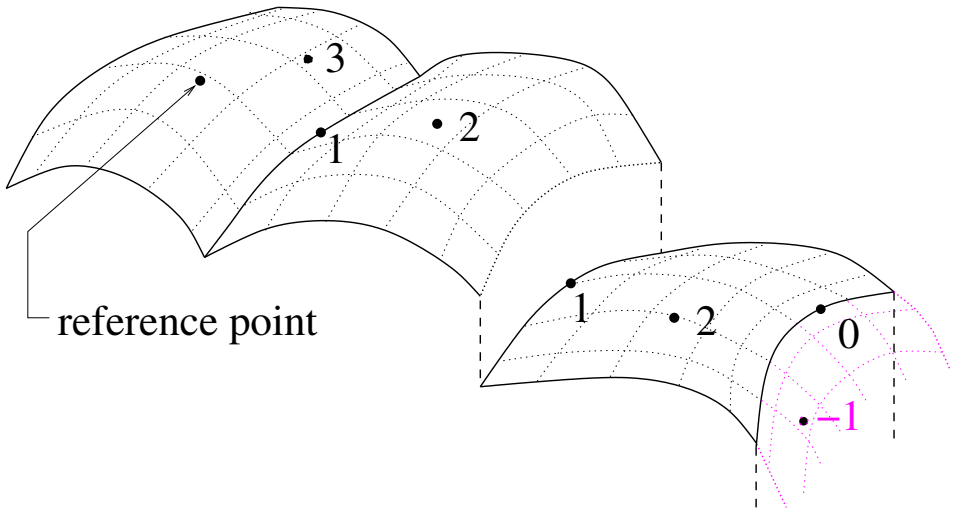
case distinction

3 locally analytic

2 locally analytic but crossed a (potential) kink (`min,max,abs,...`) or discontinuity (`ceil,...`) [for source transformation: also different control flow]

1 we are exactly at a (potential) kink, discontinuity

0 tie on arithmetic comparison (e.g. a branch condition) → potentially discontinuous (can only be determined for some special cases)



case distinction

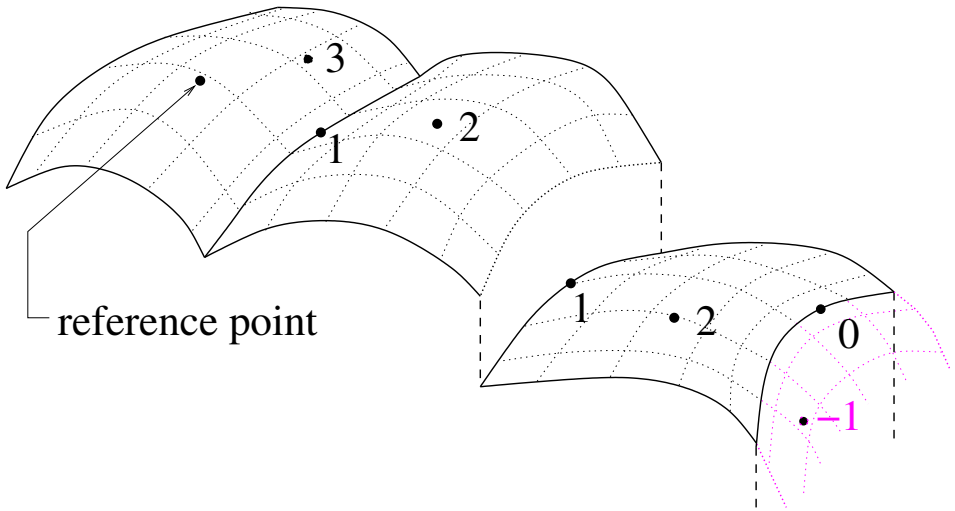
3 locally analytic

2 locally analytic but crossed a (potential) kink (`min,max,abs,...`) or discontinuity (`ceil,...`) [for source transformation: also different control flow]

1 we are exactly at a (potential) kink, discontinuity

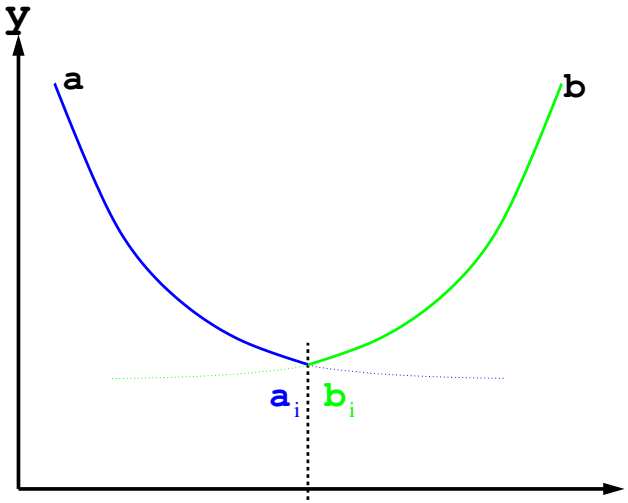
0 tie on arithmetic comparison (e.g. a branch condition) → potentially discontinuous (can only be determined for some special cases)

[-1 (operator overloading specific) arithmetic comparison yields a different value than before (tape invalid → sparsity pattern may be changed,...)]



Should AD make educated guesses?

consider `y=max(a(x),b(x))`
at the tie



pick direction from Taylor coefficients of first non-tied $\max(\mathtt{a}_i, \mathtt{b}_i)$?
consistency for unresolved ties:
take $\dot{\mathtt{a}}$ or $\dot{\mathtt{b}}$
and compare that to an adjoint split:
$\bar{\mathtt{a}} += \frac{\bar{\mathtt{y}}}{2}$ and $\bar{\mathtt{b}} += \frac{\bar{\mathtt{y}}}{2}$

consider $y = \sqrt{x}$ and $\dot{y}|_{x=+0} = \begin{cases} 0 & \text{if} \quad \dot{x} = 0 \\ \mathtt{+INF} & \text{if} \quad \dot{x} > 0 \\ \mathtt{NaN} & \text{if} \quad \dot{x} < 0 \end{cases}$

consider `maxloc`: tie-breaking argument `maxval` may differ from argument identified by `maxloc`

tracing facility - example

```
subroutine head(x,y)
  double precision :: x
  double precision :: y
!$openad INDEPENDENT(x)
  y=tan(x)
!$openad DEPENDENT(y)
end subroutine
```

driver →

```
program driver
  use OAD_active
  use OAD_rev
  use OAD_trace

  type(active) :: x, y

  x%v=.5D0
  ! first trace
  call oad_trace_init()
  call oad_trace_open()
  call head(x,y)
  call oad_trace_close()

  x%v=x%v+3.0D0
  ! second trace
  call oad_trace_open()
  call head(x,y)
  call oad_trace_close()
end program driver
```

output:

```
<Trace number="1">
<Call name="tan_scal" line="5">
</Call>
<Tan sd="0"/>
</Trace>

<Trace number="2">
<Call name="tan_scal" line="5">
</Call>
<Tan sd="1"/>
</Trace>
```

indicates subdomain of $\tan(x)$ is `sd=`k with integer $k = \lfloor \frac{x+\pi/2}{\pi} \rfloor$

tracing facility - control flow

check *active* control flow decisions:

test routine:

```
subroutine head(x1,x2,y)
  real,intent(in) :: x1,x2
  real,intent(out) :: y
  integer i
!$openad INDEPENDENT(x1)
!$openad INDEPENDENT(x2)
  y=x1
  do i=int(x1),int(x2)+2
     y=y*x2
     if (y>1.0) then
        y=y*2.0
     end if
  end do
!$openad DEPENDENT(y)
end subroutine head
```

trace at x=[0.5, 0.75]

```
<Trace number="1">
 <Loop line="8">
  <Branch line="10">
   <Cfval val="0"/>
  </Branch>
  <Branch line="10">
   <Cfval val="0"/>
  </Branch>
  <Branch line="10">
   <Cfval val="0"/>
  </Branch>
  <Cfval val="3"/>
 </Loop>
</Trace>
```

trace at x=[0.5, 1.75]

```
<Trace number="2">
 <Loop line="8">
  <Branch line="10">
   <Cfval val="0"/>
  </Branch>
  <Branch line="10">
   <Cfval val="1"/>
  </Branch>
  <Branch line="10">
   <Cfval val="1"/>
  </Branch>
  <Branch line="10">
   <Cfval val="1"/>
  </Branch>
  <Cfval val="4"/>
 </Loop>
</Trace>
```

note: difference between *active* and *varied* program variables.

tracing facility - data

associating events with program data:

test routine:

```
subroutine head(x,y)
  real :: x(2),y
!$openad INDEPENDENT(x)
  y=0.0
  do i=1,2
    y=y+sin(x(i))+ &
      tan(x(i))
  end do
!$openad DEPENDENT(y)
end subroutine
```

trace at x=[0.5, 0.75]

```
<Trace number="1">
 <Call name="tan_scal" line="6">
  <Arg name="X">
   <Index val="1"/>
  </Arg>
 </Call>
 <Tan sd="0"/>
 <Call name="tan_scal" line="6">
  <Arg name="X">
   <Index val="2"/>
  </Arg>
 </Call>
 <Tan sd="0"/>
</Trace>
```

trace at x=[0.5, 3.75]

```
<Trace number="2">
 <Call name="tan_scal" line="6">
  <Arg name="X">
   <Index val="1"/>
  </Arg>
 </Call>
 <Tan sd="0"/>
 <Call name="tan_scal" line="6">
  <Arg name="X">
   <Index val="2"/>
  </Arg>
 </Call>
 <Tan sd="1"/>
</Trace>
```

note: no arguments recorded w/o address computation...

tracing facility - call stack

need call stack context (shown by nesting):

test routine:

```
subroutine foo(t)
  real :: t
  call bar(t)
end subroutine
subroutine bar(t)
  real :: t
  t=tan(t)
end subroutine
subroutine head(x,y)
  real :: x
  real :: y
!$openad INDEPENDENT(x)
  call foo(x)
  call bar(x)
  y=x
!$openad DEPENDENT(y)
end subroutine
```

trace at x=0.5

```
<Trace number="1">
 <Call name="foo" line="13">
  <Call name="bar" line="3">
   <Call name="tan_scal" line="7">
   </Call>
   <Tan sd="0"/>
  </Call>
 </Call>
 <Call name="bar" line="14">
  <Call name="tan_scal" line="7">
  </Call>
  <Tan sd="0"/>
 </Call>
</Trace>
```

trace at x=1.0

```
<Trace number="2">
 <Call name="foo" line="13">
  <Call name="bar" line="3">
   <Call name="tan_scal" line="7">
   </Call>
   <Tan sd="0"/>
  </Call>
 </Call>
 <Call name="bar" line="14">
  <Call name="tan_scal" line="7">
  </Call>
  <Tan sd="1"/>
 </Call>
</Trace>
```

note: tracing difference only for the direct call from `head`, not from `foo`

conclusions

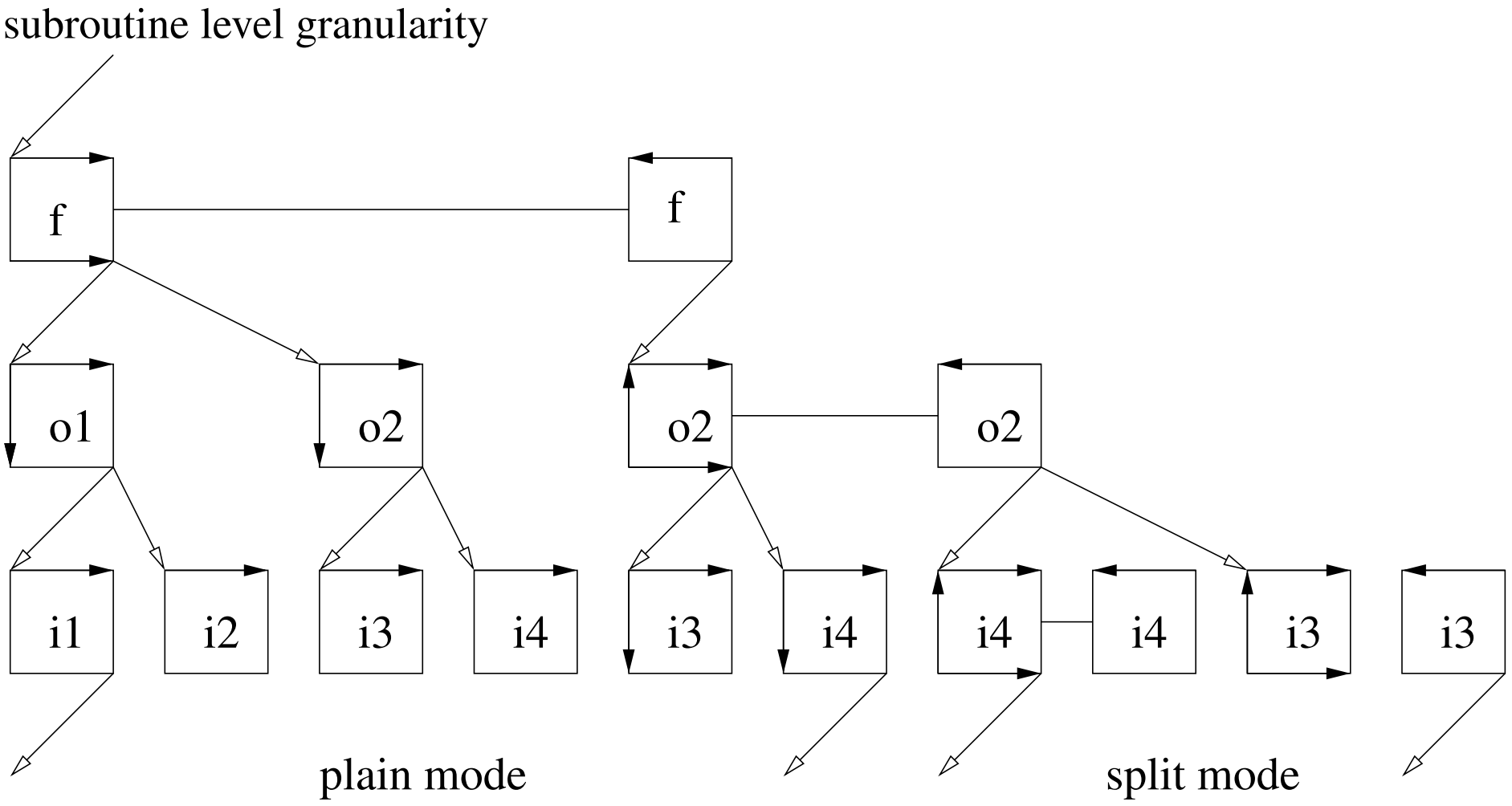
obvious (by now) recommendations regarding smoothness:

- avoid introducing numerical special cases
- pathological cases at domain boundaries, initial conditions
- filter out computations outside of the actual domain (e.g. $\sqrt{0}$)
- consider explicit logic to smooth (e.g. C^1 ?) kinks and discontinuities

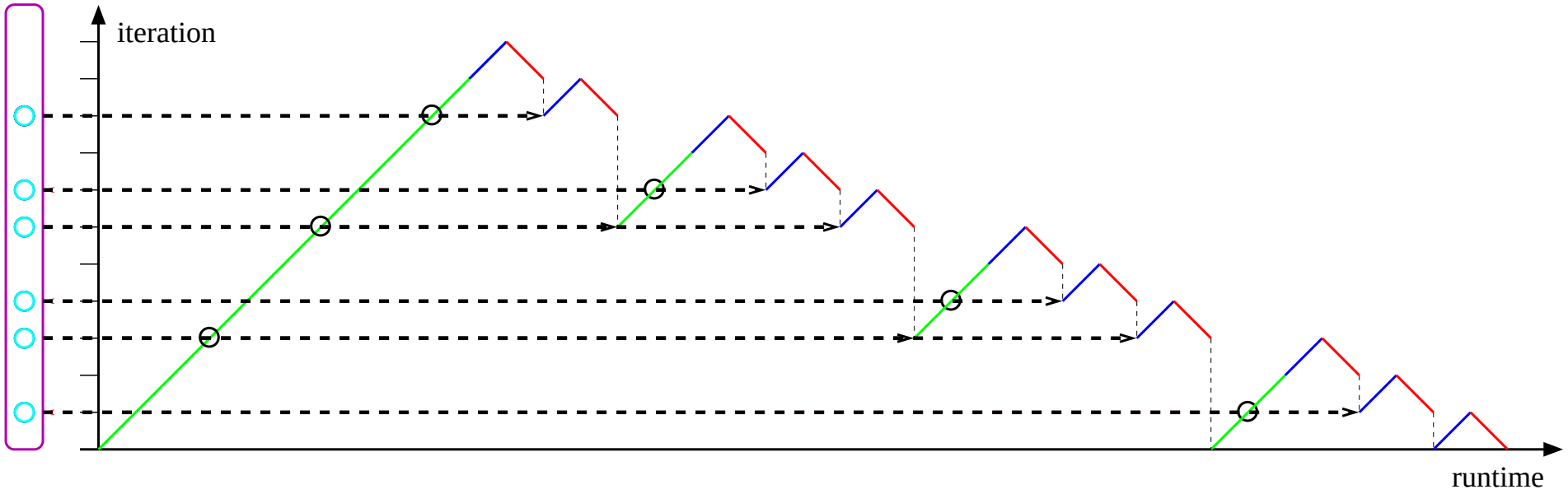
alternative (unimplemented) approaches:

- slopes (interval based)
- Laurent series (w different rules regarding $\pm$`INF` and `NaN`

reversal scheme with nested checkpointing

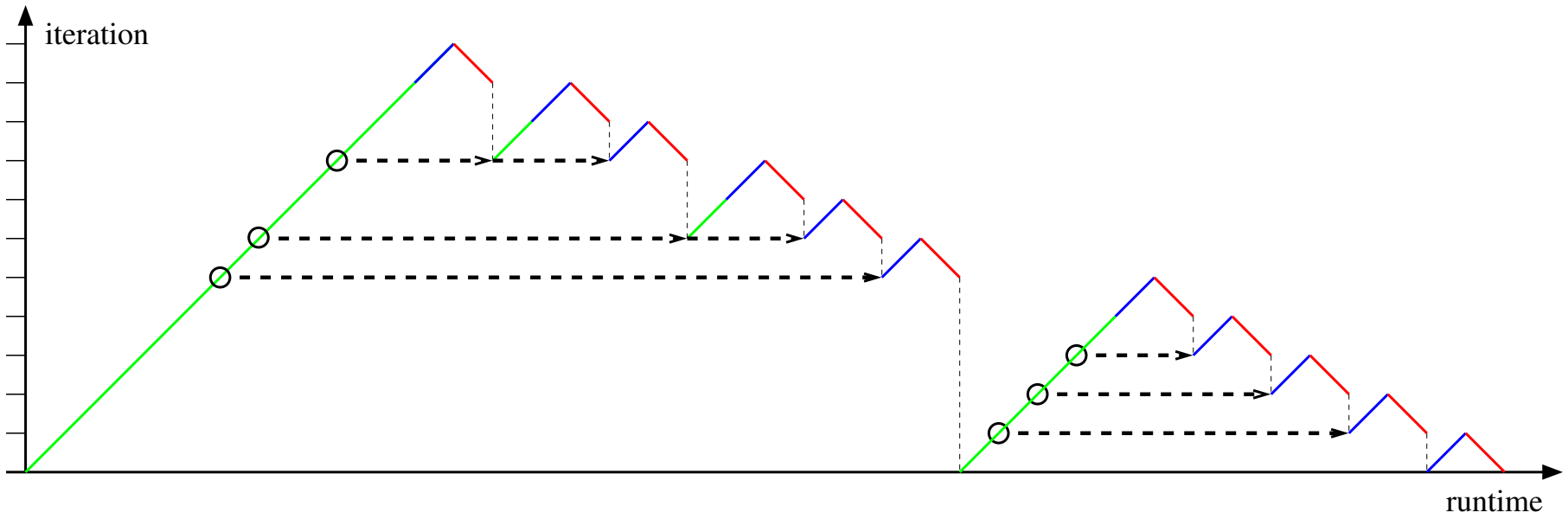


use of checkpointing to mitigate storage requirements



- 11 iters., memory limited to one iter. of storing $\boldsymbol{J}_i$ & 3 checkpoints

use of checkpointing to mitigate storage requirements



- 11 iters., memory limited to one iter. of storing $\boldsymbol{J}_i$ & 3 checkpoints
- optimal (binomial) scheme encoded in `revolve`; F9X implementation available at `http://mercurial.mcs.anl.gov/ad/RevolveF9X`

OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
   rvAct=rvNextAction()
   select case (rvAct%actionFlag)
   case (rvStore)
      call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
      call cp_close()
   case (rvRestore)
      call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
      currIter=rvAct%iteration
      call cp_close()
   case (rvForward)
      oadRevMod%plain=.TRUE.
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.FALSE.
      do while (currIter<rvAct%iteration)
        call loopBody(x)
        currIter=currIter+1
      end do
    case (rvFirstUTurn)
      oadRevMod%plain=.FALSE.
      oadRevMod%tape=.TRUE.
      oadRevMod%adjoint=.FALSE.
      call loopBody(x)
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.TRUE.
      call loopBody(x)
    case (rvUTurn)...
    end select
end do
```

OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

- init `revolve`

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
   rvAct=rvNextAction()
   select case (rvAct%actionFlag)
   case (rvStore)
      call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
      call cp_close()
   case (rvRestore)
      call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
      currIter=rvAct%iteration
      call cp_close()
   case (rvForward)
      oadRevMod%plain=.TRUE.
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.FALSE.
      do while (currIter<rvAct%iteration)
        call loopBody(x)
        currIter=currIter+1
      end do
    case (rvFirstUTurn)
      oadRevMod%plain=.FALSE.
      oadRevMod%tape=.TRUE.
      oadRevMod%adjoint=.FALSE.
      call loopBody(x)
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.TRUE.
      call loopBody(x)
    case (rvUTurn)...
    end select
end do
```

OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

- init `revolve`
- revolve loop

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
   rvAct=rvNextAction()
   select case (rvAct%actionFlag)
   case (rvStore)
      call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
      call cp_close()
   case (rvRestore)
      call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
      currIter=rvAct%iteration
      call cp_close()
   case (rvForward)
      oadRevMod%plain=.TRUE.
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.FALSE.
      do while (currIter<rvAct%iteration)
        call loopBody(x)
        currIter=currIter+1
      end do
    case (rvFirstUTurn)
      oadRevMod%plain=.FALSE.
      oadRevMod%tape=.TRUE.
      oadRevMod%adjoint=.FALSE.
      call loopBody(x)
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.TRUE.
      call loopBody(x)
    case (rvUTurn)...
    end select
end do
```

OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

- init `revolve`
- revolve loop
- get the action

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
   rvAct=rvNextAction()
   select case (rvAct%actionFlag)
   case (rvStore)
      call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
      call cp_close()
   case (rvRestore)
      call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
      currIter=rvAct%iteration
      call cp_close()
   case (rvForward)
      oadRevMod%plain=.TRUE.
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.FALSE.
      do while (currIter<rvAct%iteration)
        call loopBody(x)
        currIter=currIter+1
      end do
    case (rvFirstUTurn)
      oadRevMod%plain=.FALSE.
      oadRevMod%tape=.TRUE.
      oadRevMod%adjoint=.FALSE.
      call loopBody(x)
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.TRUE.
      call loopBody(x)
    case (rvUTurn)...
    end select
end do
```

OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

- init `revolve`
- revolve loop
- get the action
- transformation provides:
  - store checkpoint

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
   rvAct=rvNextAction()
   select case (rvAct%actionFlag)
   case (rvStore)
      call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
      call cp_close()
   case (rvRestore)
      call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
      currIter=rvAct%iteration
      call cp_close()
   case (rvForward)
      oadRevMod%plain=.TRUE.
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.FALSE.
      do while (currIter<rvAct%iteration)
        call loopBody(x)
        currIter=currIter+1
      end do
    case (rvFirstUTurn)
      oadRevMod%plain=.FALSE.
      oadRevMod%tape=.TRUE.
      oadRevMod%adjoint=.FALSE.
      call loopBody(x)
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.TRUE.
      call loopBody(x)
    case (rvUTurn)...
    end select
end do
```

OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

- init `revolve`
- revolve loop
- get the action
- transformation provides:
  - store checkpoint
  - restore checkpoint

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
   rvAct=rvNextAction()
   select case (rvAct%actionFlag)
   case (rvStore)
      call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
      call cp_close()
   case (rvRestore)
      call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
      currIter=rvAct%iteration
      call cp_close()
   case (rvForward)
      oadRevMod%plain=.TRUE.
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.FALSE.
      do while (currIter<rvAct%iteration)
        call loopBody(x)
        currIter=currIter+1
      end do
    case (rvFirstUTurn)
      oadRevMod%plain=.FALSE.
      oadRevMod%tape=.TRUE.
      oadRevMod%adjoint=.FALSE.
      call loopBody(x)
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.TRUE.
      call loopBody(x)
    case (rvUTurn)...
    end select
end do
```

OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

- init `revolve`
- revolve loop
- get the action
- transformation provides:
  - store checkpoint
  - restore checkpoint
  - forward to a iteration

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
   rvAct=rvNextAction()
   select case (rvAct%actionFlag)
   case (rvStore)
      call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
      call cp_close()
   case (rvRestore)
      call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
      currIter=rvAct%iteration
      call cp_close()
   case (rvForward)
      oadRevMod%plain=.TRUE.
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.FALSE.
      do while (currIter<rvAct%iteration)
        call loopBody(x)
        currIter=currIter+1
      end do
    case (rvFirstUTurn)
      oadRevMod%plain=.FALSE.
      oadRevMod%tape=.TRUE.
      oadRevMod%adjoint=.FALSE.
      call loopBody(x)
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.TRUE.
      call loopBody(x)
    case (rvUTurn)...
    end select
end do
```

OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

- init `revolve`
- revolve loop
- get the action
- transformation provides:
  - store checkpoint
  - restore checkpoint
  - forward to a iteration
  - store & adjoin

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
   rvAct=rvNextAction()
   select case (rvAct%actionFlag)
   case (rvStore)
      call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
      call cp_close()
   case (rvRestore)
      call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
      currIter=rvAct%iteration
      call cp_close()
   case (rvForward)
      oadRevMod%plain=.TRUE.
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.FALSE.
      do while (currIter<rvAct%iteration)
        call loopBody(x)
        currIter=currIter+1
      end do
    case (rvFirstUTurn)
      oadRevMod%plain=.FALSE.
      oadRevMod%tape=.TRUE.
      oadRevMod%adjoint=.FALSE.
      call loopBody(x)
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.TRUE.
      call loopBody(x)
    case (rvUTurn)...
    end select
end do
```

OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

- init `revolve`
- revolve loop
- get the action
- transformation provides:
  - store checkpoint
  - restore checkpoint
  - forward to a iteration
  - store & adjoin
- OpenAD has documented examples

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
   rvAct=rvNextAction()
   select case (rvAct%actionFlag)
   case (rvStore)
      call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
      call cp_close()
   case (rvRestore)
      call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
      currIter=rvAct%iteration
      call cp_close()
   case (rvForward)
      oadRevMod%plain=.TRUE.
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.FALSE.
      do while (currIter<rvAct%iteration)
        call loopBody(x)
        currIter=currIter+1
      end do
    case (rvFirstUTurn)
      oadRevMod%plain=.FALSE.
      oadRevMod%tape=.TRUE.
      oadRevMod%adjoint=.FALSE.
      call loopBody(x)
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.TRUE.
      call loopBody(x)
    case (rvUTurn)...
    end select
end do
```

MPI - data transfer between processes

- simple MPI program needs 6 calls :

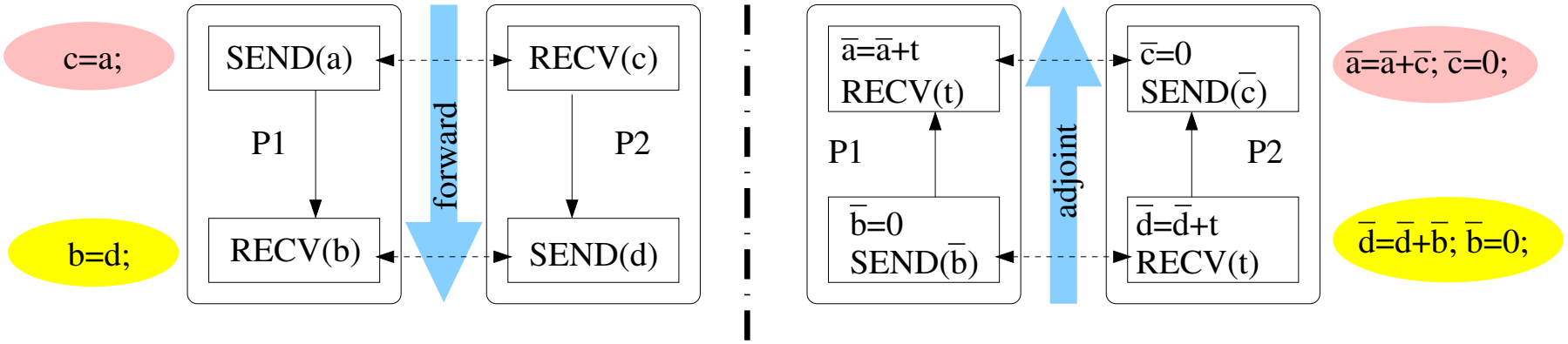
```
mpi_init      // initialize the environment
mpi_comm_size// number of processes in the communicator
mpi_comm_rank// rank of this process in the communicator
mpi_send      // send (blocking)
mpi_recv      // receive (blocking)
mpi_finalize // cleanup
```

MPI - data transfer between processes

- simple MPI program needs 6 calls :

```
mpi_init      // initialize the environment
mpi_comm_size// number of processes in the communicator
mpi_comm_rank// rank of this process in the communicator
mpi_send      // send (blocking)
mpi_recv      // receive (blocking)
mpi_finalize // cleanup
```

- example adjoining blocking communication between 2 processes:

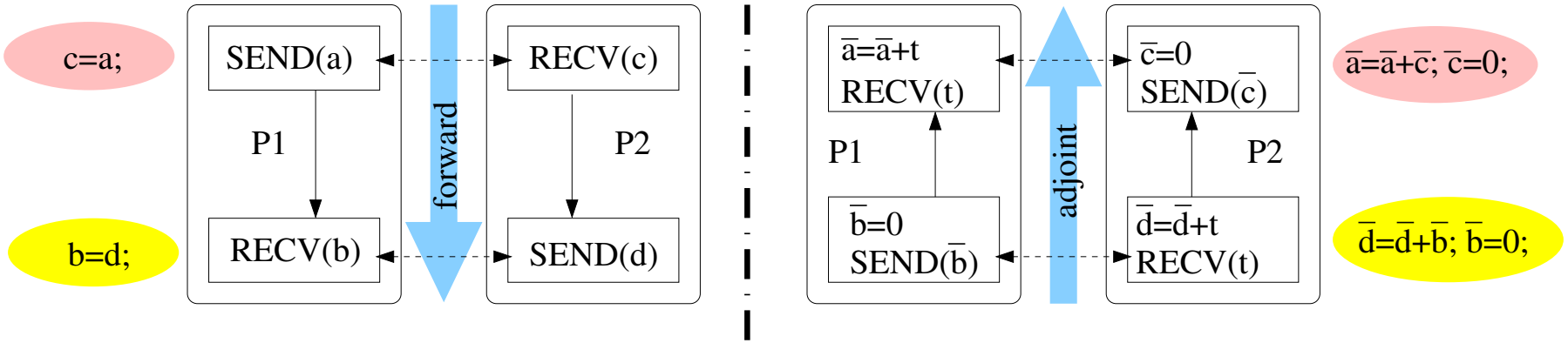


MPI - data transfer between processes

- simple MPI program needs 6 calls :

```
mpi_init      // initialize the environment
mpi_comm_size// number of processes in the communicator
mpi_comm_rank// rank of this process in the communicator
mpi_send      // send (blocking)
mpi_recv      // receive (blocking)
mpi_finalize // cleanup
```

- example adjoining blocking communication between 2 processes:



- use the communcation graph as model

MPI contd.

- MPI usage in the MITgcm ocean model: exchange tile halos, reductions operations. synchronization...

MPI contd.

- MPI usage in the MITgcm ocean model: exchange tile halos, reductions operations. synchronization...
- interfaces to various languages (here using Fortran), C implementation

MPI contd.

- MPI usage in the MITgcm ocean model: exchange tile halos, reductions operations. synchronization...
- interfaces to various languages (here using Fortran), C implementation
- total 287 routines (MPI-2)

MPI contd.

- MPI usage in the MITgcm ocean model: exchange tile halos, reductions operations. synchronization...
- interfaces to various languages (here using Fortran), C implementation
- total 287 routines (MPI-2)
- covering: communication, setup, grouping of processes, I/O, status queries, topologies, debugging,...

MPI contd.

- MPI usage in the MITgcm ocean model: exchange tile halos, reductions operations. synchronization...
- interfaces to various languages (here using Fortran), C implementation
- total 287 routines (MPI-2)
- covering: communication, setup, grouping of processes, I/O, status queries, topologies, debugging,...
- concentrate on portion “relevant” for AD (e.g. ignore one-sided comm. ?)

MPI contd.

- MPI usage in the MITgcm ocean model: exchange tile halos, reductions operations. synchronization...
- interfaces to various languages (here using Fortran), C implementation
- total 287 routines (MPI-2)
- covering: communication, setup, grouping of processes, I/O, status queries, topologies, debugging,...
- concentrate on portion "relevant" for AD (e.g. ignore one-sided comm. ?)
- need to enable activity analysis

MPI contd.

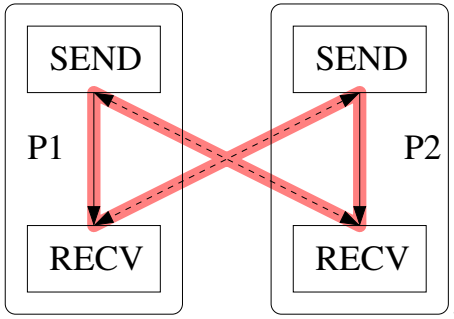
- MPI usage in the MITgcm ocean model: exchange tile halos, reductions operations. synchronization...
- interfaces to various languages (here using Fortran), C implementation
- total 287 routines (MPI-2)
- covering: communication, setup, grouping of processes, I/O, status queries, topologies, debugging,...
- concentrate on portion "relevant" for AD (e.g. ignore one-sided comm. ?)
- need to enable activity analysis
- consider the *communication modes*:
  - for send: `mpi_[i][b|s|r]send`
  - for receive: `mpi_[i]recv`

MPI contd.

- MPI usage in the MITgcm ocean model: exchange tile halos, reductions operations. synchronization...
- interfaces to various languages (here using Fortran), C implementation
- total 287 routines (MPI-2)
- covering: communication, setup, grouping of processes, I/O, status queries, topologies, debugging,...
- concentrate on portion "relevant" for AD (e.g. ignore one-sided comm. ?)
- need to enable activity analysis
- consider the *communication modes*:
  - for send: `mpi_[i][b|s|r]send`
  - for receive: `mpi_[i]recv`
- ensure correctness and aim at improving efficiency
  $\Rightarrow$ want the same for adjoints

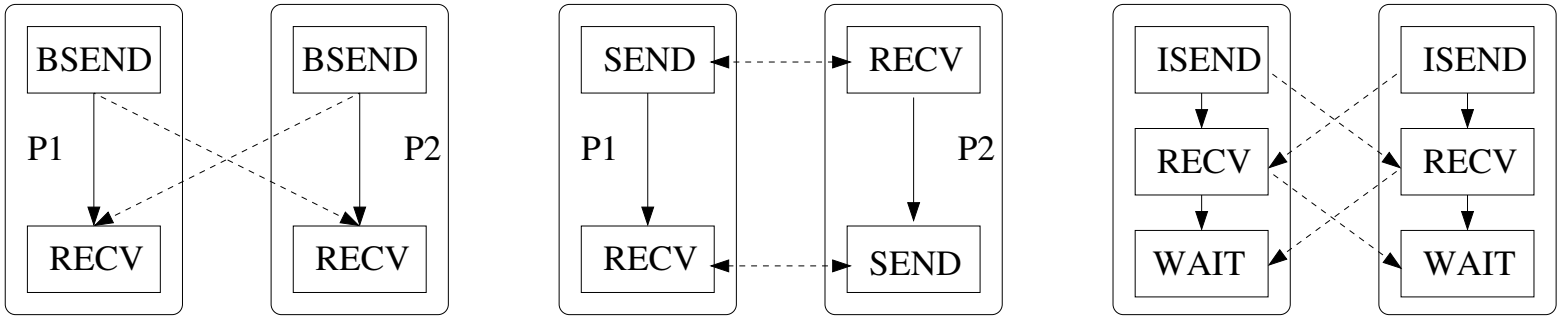
MPI idioms and correctness

- correct parameters ? (data,endpoints)
- no deadlocks ? (look at communication graphs)
- for example: data exchange between P1 and P2



... has a cycle (involving comm.edges)

- break with buffered* sends, reordering, non-blocking sends, ...

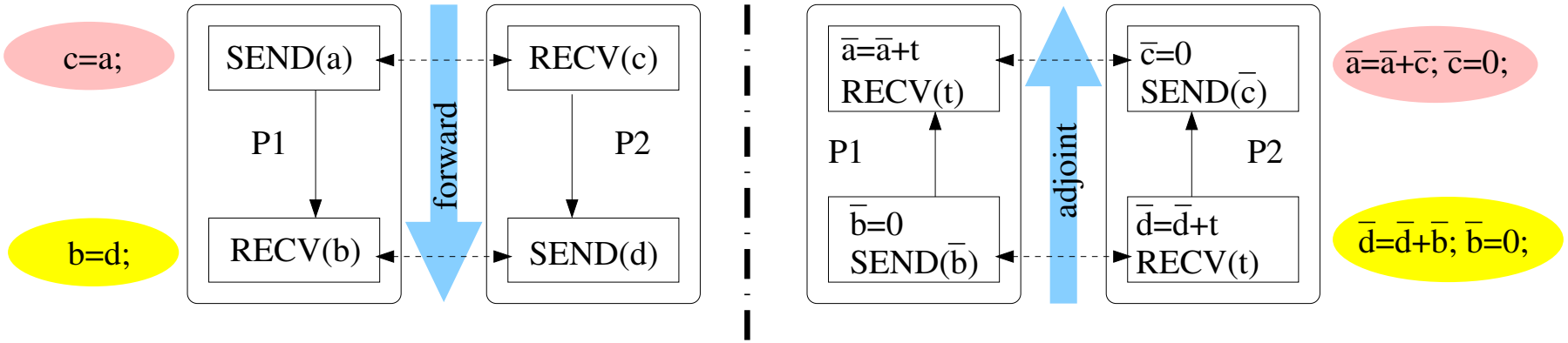


the last idiom is used in MITgcm

* resource starvation?

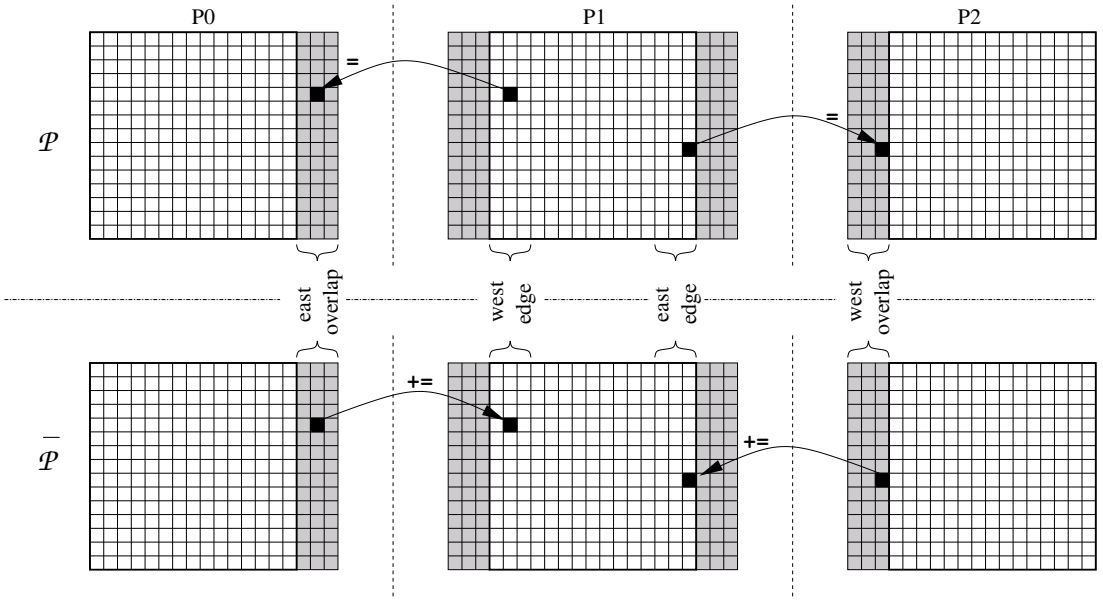
easy adjoints for blocking calls

- *easy* adjoint transformation: `send` $\mapsto$ `recv` and `recv` $\mapsto$ `send`



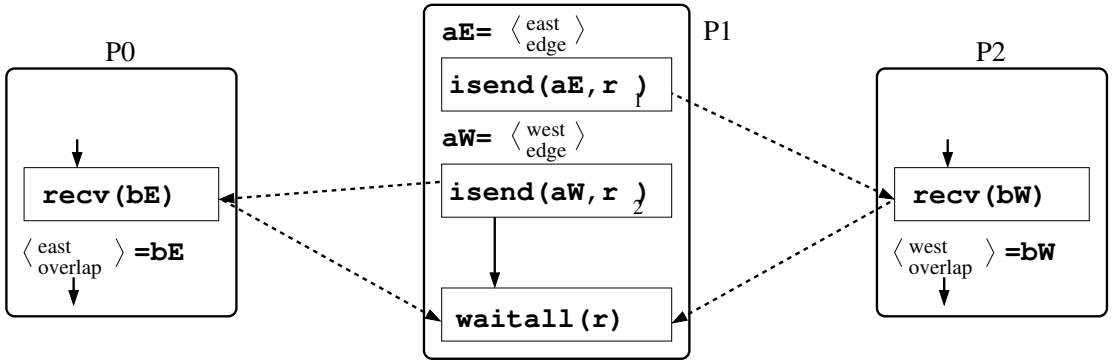
- hyp.: if the forward communication graph is acyclic, so is the adjoint; look at the communication graph with reversed edges
- for activity analysis: difficult to statically determine send/recv pairs; e.g. consider set of all possible dynamic comm. graphs
- with wildcards (but no threads): record actual sources/tags on receive and send with recorded tag to recorded source in the adjoint sweep
- hyp.: no forward deadlock $\equiv$ no cycle in current dynamic comm. graph $\Rightarrow$ no cycle in inverted dynamic comm. graph $\equiv$ no adjoint deadlock

MITgcm uses manual adjoints for halo exchange



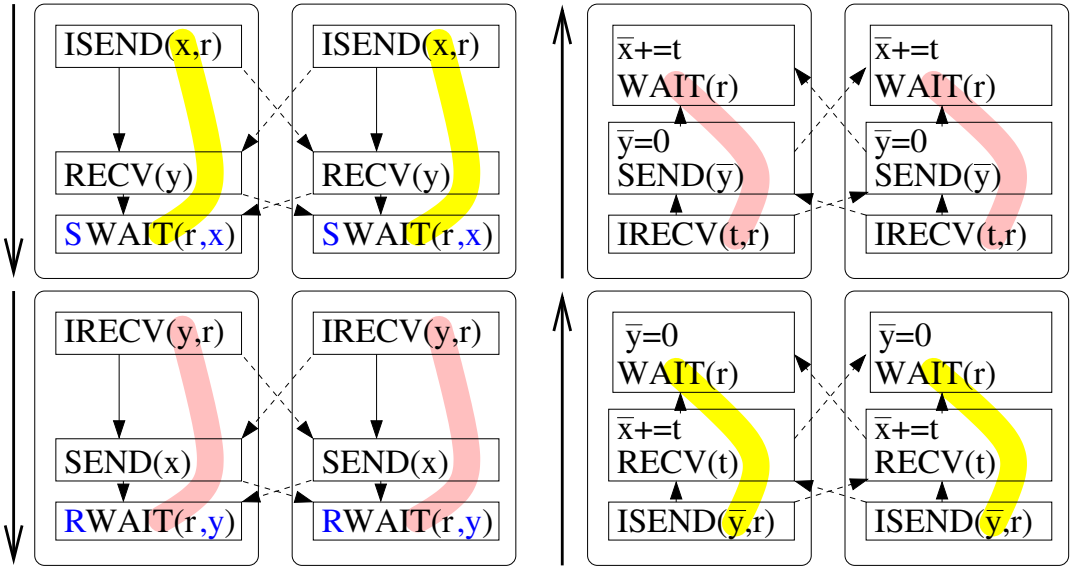
actually in E-W and N-S; need to consider corners...

- every tile needs to talk to its neighbors
- prevent deadlocks by imposing ordering ? (requires high-level view and imposes order)
- prevent deadlocks by buffered communication? (can run out of bufferspace)
- use non-blocking calls, the idiom used here is `ISEND*` – `RECV*` – `WAITALL`
- no "easy" transformation; multiple in-edges at `waitall`

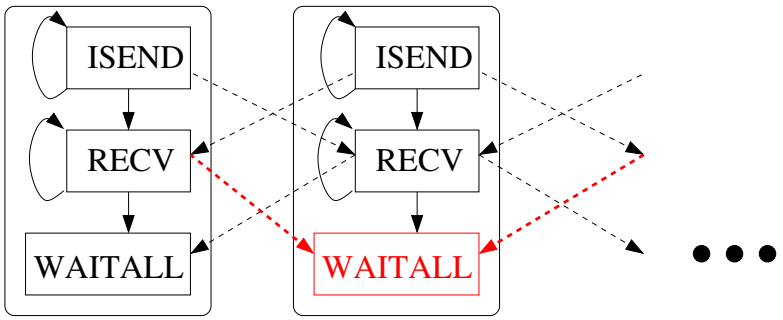


no easy transformation because ...

- consider the communication graphs for simple nonblocking idioms
- need to retain correctness, i.e. use nonblocking calls in the adjoint

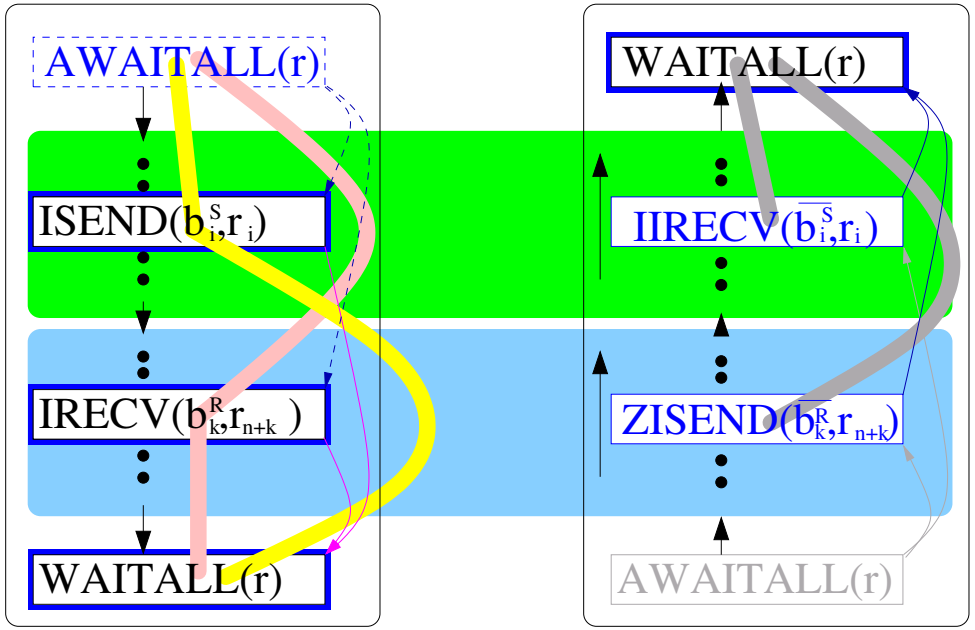


- the above transformations are provably correct
- **extensions** to convey context
  $\Rightarrow$ enables a transformation recipe per call
- promises to not read or **write** the respective buffer



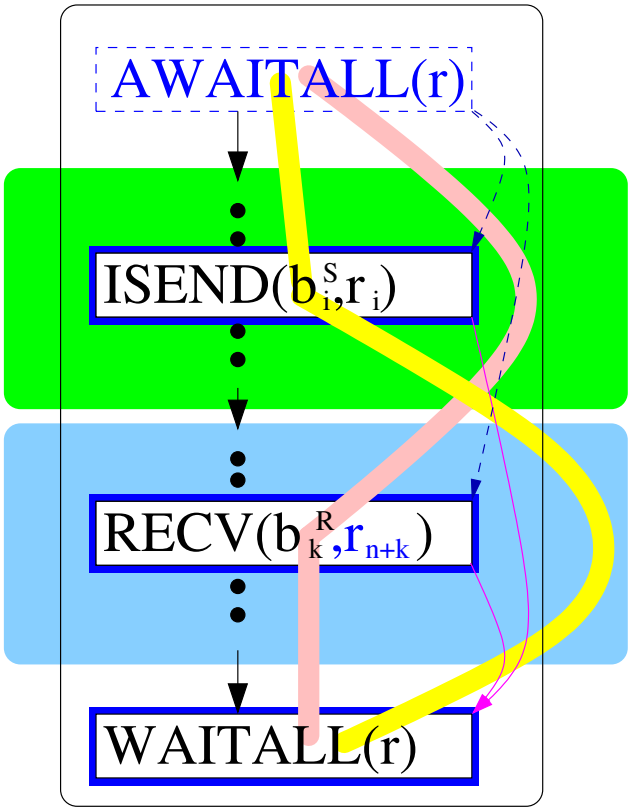
a solution

- require a symmetric counterpart to the `waitall`; think *"anti_waitall"*
- retains more efficient pattern ☺
- extend promises symmetrically to `AWAITALL`



- wrapped MPI calls have logic to delay zeroing/increment buffers
- goal: "simple" adjoint interpretation for each call

in the OpenAD prototype




```
call ampi_awaitall(exchNReqsX(1,bi,bj),&
                   exchReqIdX(1,1,bi,bj), &
                   mpiStatus, mpiRC)
```

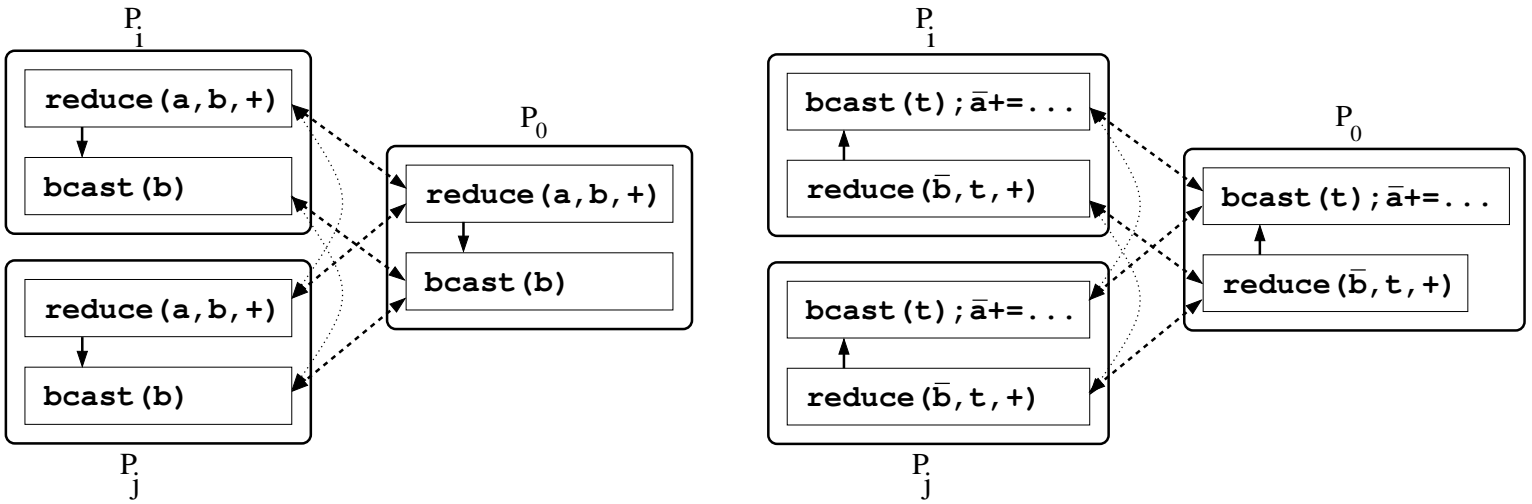
```
call ampi_isend(westSendBuf_RL(1,eBl,bi,bj),&
                theSize, theType, theProc, theTag, &
                MPI_COMM_MODEL,&
                exchReqIdX(pReqI,1,bi,bj), &
                exchNReqsX(1,bi,bj),&
                mpiStatus , mpiRc)
```

```
call ampi_wrecv(westRecvBuf_RL(1,eBl,bi,bj),&
                theSize, theType, theProc, theTag,&
                MPI_COMM_MODEL ,&
                exchReqIdX(pReqI,1,bi,bj), &
                exchNReqsX(1,bi,bj), &
                mpiStatus, mpiRc)
```

```
call ampi_waitall(exchNReqsX(1,bi,bj),&
                  exchReqIdX(1,1,bi,bj), &
                  mpiStatus, mpiRC)
```

collective communication

- example: reduction followed by broadcast
  $\mathtt{b}_0 = \sum \mathtt{a}_i$ followed by $\mathtt{b}_i = \mathtt{b}_0 \forall i$
- conceptually simple; `reduce` $\mapsto$ `bcast` and `bcast` $\mapsto$ `reduce`



- adjoint: $\mathtt{t}_0 = \sum \bar{\mathtt{b}}_i$ followed by $\bar{\mathtt{a}}_i \mathtt{+=} \mathtt{t}_0 \forall i$
- has single transformation points (connected by hyper communication edge)
- efficiency for product reduction because of increment $\bar{\mathtt{a}}_i \mathtt{+=} \frac{\partial \mathtt{b}_0}{\partial \mathtt{a}_i} \mathtt{t}_0, \forall i$

summary

- prerequisites: communication edges not interrupted by checkpoints
- resource problem for communicators (topology descriptions)
- initial wrapped MPI solution
- later integrated MPI solution + analysis support
- goal standardize (small) set of guaranteed adjoinable MPI calls
  $\Longrightarrow$ ongoing discussion with MPI group at Argonne
- final remark: adjoining OpenMP has problems too, e.g.
  the adjoint of `omp parallel do` *isn't necessarily parallel too*

```
DO I=2, N-1
   A(I)=X(I-1)-2*X(I)+X(I+1)
   B(I)=A(I)+SQRT(X(I))
ENDDO
```

```
DO I=N-1,2, -1
   Ā(I)   =Ā(I)   +B̄(I)
   X̄(I)   =X̄(I)   +B̄(I)*1./(2*SQRT(X(I))
   X̄(I-1)=X̄(I-1)+Ā(I)
   X̄(I)   =X̄(I)   +2*Ā(I)
   X̄(I+1)=X̄(I+1)+Ā(I)
ENDDO
```

race between reads and writes on the $\bar{X}$

source transformation vs. operator overloading

- complicated implementation of tools
- especially for reverse mode
- full front end, back end, analysis
- efficiency gains from
  - compile time AD optimizations
  - activity analysis
  - explicit control flow reversal
- source transformation based type change & overloaded operators appropriate for higher-order derivatives.
- efficiency depends on analysis accuracy

- simple tool implementation
- reverse mode: generate & reinterpret an execution trace → inefficient
- implemented as a library
- efficiency gains from:
  - runtime AD optimization
  - optimized library
  - inlining (for low order)
- manual type change
  - ↯ formatted I/O, allocation,...
  - matching signatures (Fortran)
  - easier with templates

---

higher-order derivatives ⇒ source transformation based type change
+ overloaded operators.

forward vs. reverse

- simplest rule: given $y = f(x) : \mathbb{R}^n \mapsto \mathbb{R}^m$ use reverse if $n \gg m$ (gradient)
- what if $n \approx m$ and large
  - want only projections, e.g. $J\dot{x}$
  - sparsity (e.g. of the Jacobian)
  - partial separability (e.g. $f(x) = \sum(f_i(x_i)), x_i \in \mathcal{D}_i \Subset \mathcal{D} \ni x$)
  - intermediate interfaces of different size
- the above may make forward mode feasible
  (projection $\bar{y}^T J$ requires reverse)
- higher order tensors (practically feasible for small n) $\rightarrow$ forward mode
  (reverse mode saves factor n in effort only once)
- this determines overall propagation direction, not necessarily the local preaccumulation (combinatorial problem)