Designing and implementing a tool-indedependent, adjoinable MPI wrapper library

J. Utke[1], L. Hascoët[2], M. Schanen[3]

[1] Argonne National Laboratory

[2]INRIA Sophia-Antipolis

[3]RWTH Aachen

Petascale Computing Joint Lab workshop June/2013 - Lyon

outline

- the algorithmic differentiation (AD) context
- concepts of adjoining numerical models with MPI communication
- objectives of a reusable implementation for adjoinable MPI
- limitations imposed by the AD tools
- the current state and the path forward.

Adjoining with algorithmic differentiation (1)

algorithmic differentiation (AD) aka automatic differentiation

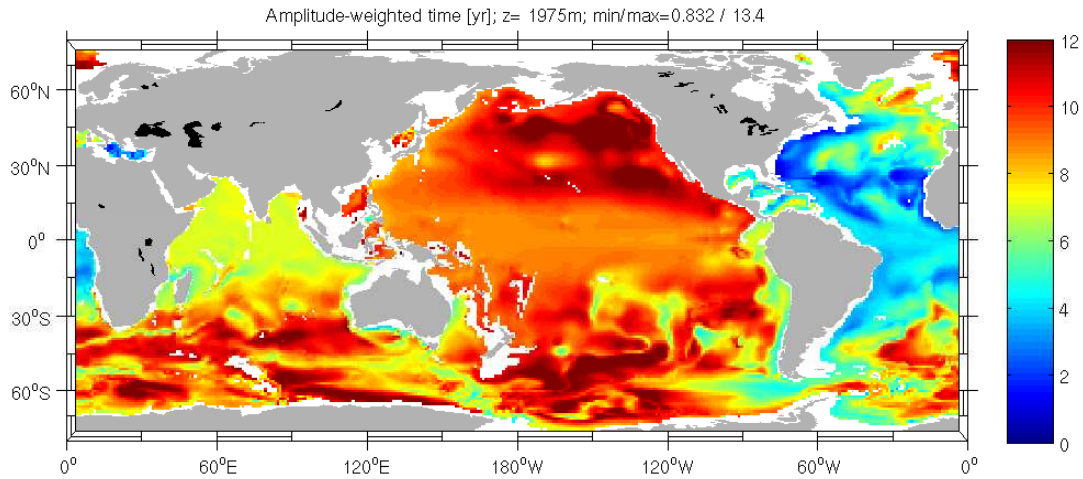
given: $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$

wanted: machine precision derivatives (of the algorithm) e.g.

- Jacobian projections forward: $\dot{\boldsymbol{y}} = \boldsymbol{J}\dot{\boldsymbol{x}}$,
- reverse (adjoint): $\bar{\boldsymbol{x}} = \boldsymbol{J}^T\bar{\boldsymbol{y}}$
- especially $\nabla \boldsymbol{f}$ for $m = 1$

why adjoint: $\nabla \boldsymbol{f}$ is computed at a small fixed factor over the cost of f, independent of the size of $\nabla \boldsymbol{f}$

uses: sensitivity analysis, optimization, state estimation



Adjoining with algorithmic differentiation (2)

how does it work?

- view $\boldsymbol{f}$ as a program $\mathcal{P}$ executing a sequence $[\phi_1, \phi_2, \ldots]$ of elemental operations $v_k = \phi(v_i, v_j, \ldots)$
- think of the v_l as values assigned to program variables; follow the data dependencies
- forward propagation $\dot{v}_k = \phi_{v_i}\dot{v}_i + \phi_{v_j}\dot{v}_j + \ldots$ with partials ϕ_{v_l}
- reverse (adjoint) propagation:
  $\bar{v}_i = \bar{v}_i + \phi_{v_i}\bar{v}_k; \bar{v}_j = \bar{v}_j + \phi_{v_j}\bar{v}_k; \ldots; \bar{v}_k = 0;$
- note the reversal of the data dependencies
- in particular for assignments: $t = s$ adjoint propagation implies $\bar{s} = \bar{s} + \bar{t}; \bar{t} = 0$
- important because assignments are the model for MPI send-recv from source s to target t

Adjoining with algorithmic differentiation (3)

how is it implemented?

by *semantic augmentation* of the original program.

- *data augmentation* adds a $\bar{v}_l$ to each v_l via
  - *association by name*: create a `b_v` for each `v`, as e.g. in Tapenade, Adifor
  - *association by address*: redeclare `v` to be of a structured type containing the value and an adjoint (OpenAD) or a (virtual) address or a mix (overloading tools); issues with non-contiguous data
- *logic augmentation* adds the adjoint propagation statements $\bar{v}_i = \bar{v}_i + \phi_{v_i}\bar{v}_k$ etc. to the original code via
  - *source transformation:* creates a reverted control flow with basic blocks into which the respective adjoint statements are inserted, e.g. OpenAD, Tapenade (in reverse order)
  - *operator overloading:* create a trace during the forward execution which then is played backward and interpreted, e.g. Adol-C, dco

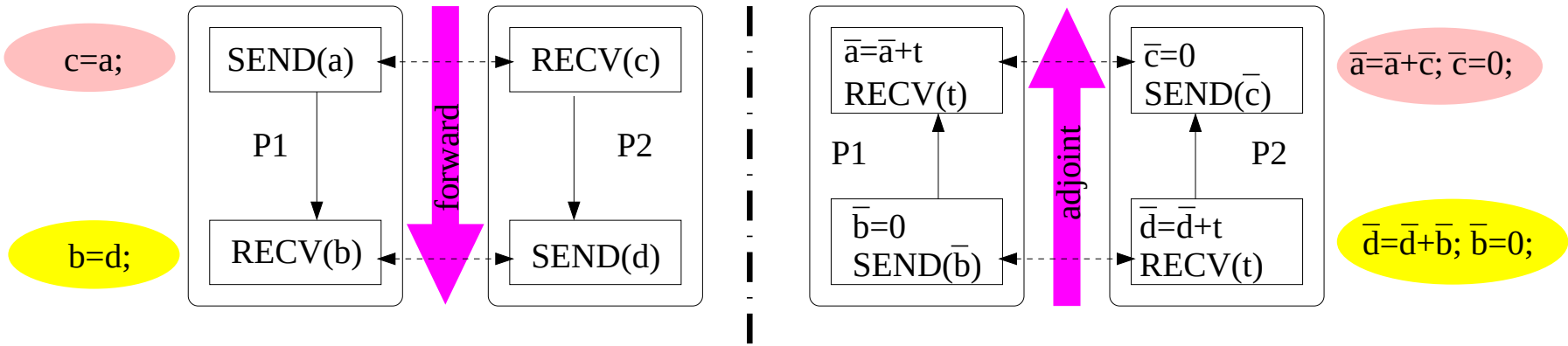
Implementation choices impact the MPI adjoint implementation!

Simple MPI

- 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 and interpret as assignments



- use the communication graph as model

previously on "AD and MPI"

not exhaustive and in no particular order:

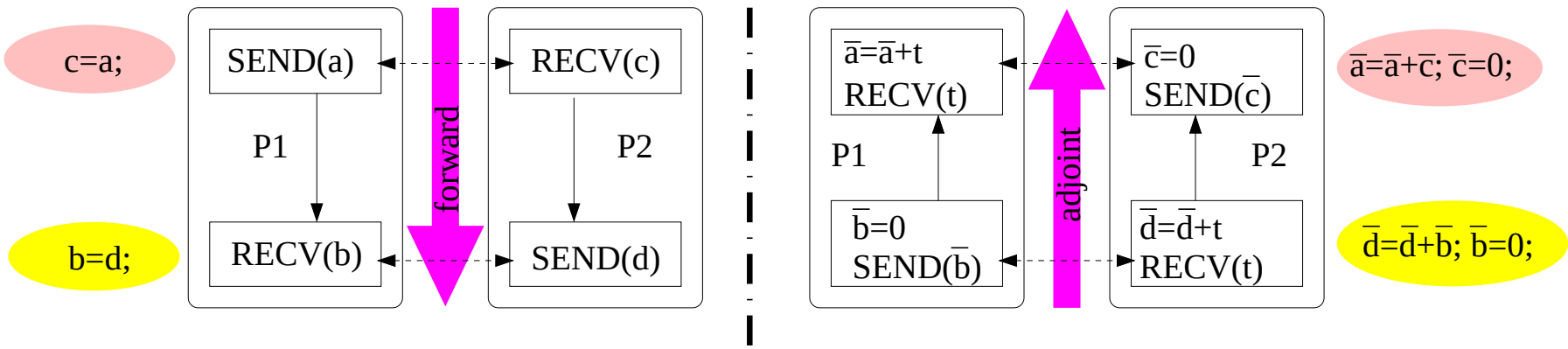
- Hovland: thesis *"AD of parallel programs"* - mostly forward
- Hovland/Bischof: *"Automatic Differentiation for Message-Passing Parallel Programs"* - association between value and derivative
- Carle/Fagan: *"Automatically Differentiating MPI-1 Datatypes"* - ditto
- Faure/Dutto: *"Extension of Odyssée to the MPI library -Reverse mode"* - plain send/recv
- Cheng: *"A Duality between Forward and Adjoint MPI Communication Routines"* - plain send/recv
- Carle: in ch. 24 of *"Sourcebook of Parallel Computing"* - 4 pgs on analysis, plain send/recv
- Strout/Hovland/Kreaseck: *"Data flow analysis for MPI programs"*
- Heimbach/Hill/Giering: *"Automatic generation of efficient adjoint code for a parallel Navier-Stokes Solver"*
  - hand-written communication adjoints in MITgcm
- Griewank: first ed. of "the book" had 2 pages on parallel programs; second edition has more

scope of consideration

- "typical" MPI usage in the MITgcm ocean model: exchange tile halos, reductions, synchronization.
- separating adjoinable communication from setup, grouping of processes, I/O, status queries, topologies, debugging,...
- concentrate on portion "relevant" for AD
- consider the *communication modes*:
  - for send: `mpi_[i][b|s|r]send`
    - i: nonblocking
    - b: buffered
    - s: synchronous
    - r: ready
  - for receive: `mpi_[i]recv`

requirements/goals for the adjoinable MPI concept (1)

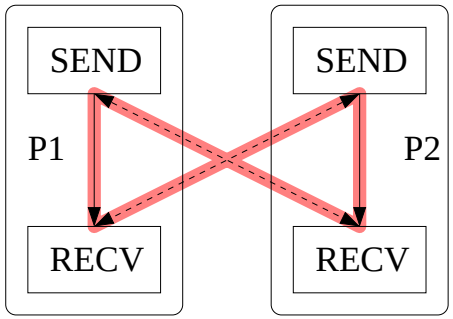
- ensure correctness of the adjoint, i.e. correct endpoints source $\leftrightarrow$ target, correct increment (adjoint of send) / nullification (adjoint of receive)
- *easy* adjoint transformation for blocking calls: `send` $\mapsto$ `recv` and `recv` $\mapsto$ `send`



- has to remain deadlock free (do not shadow messages)
- works for forward mode too

requirements/goals (2)

- look at communication graphs; example: data exchange between P1 and P2

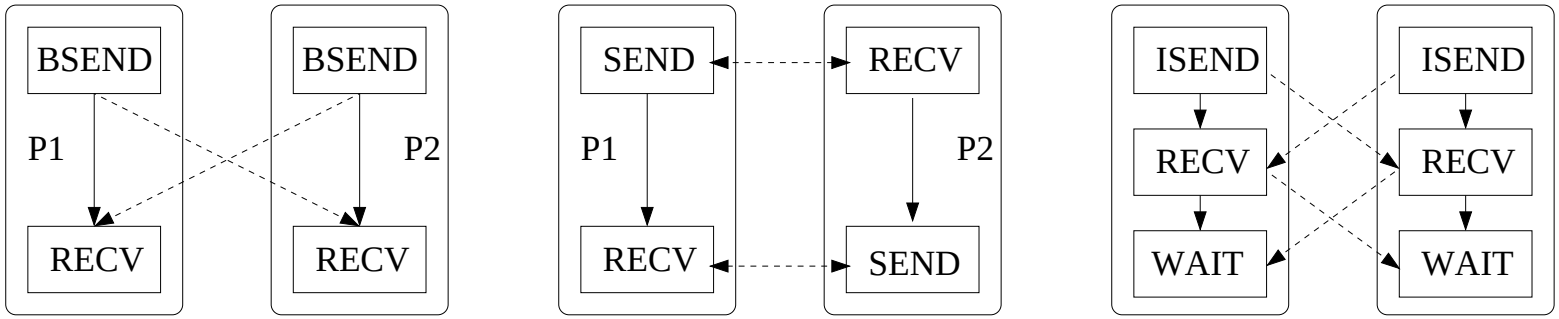


... has a cycle (involving comm.edges)

- hyp.: if the forward communication graph is acyclic, so is the adjoint; look at the communication graph with reversed edges
- with wildcards (but no threads): record actual sources/tags on receive and send with recorded tag to recorded source in the adjoint sweep
- deadlock free hypothesis extends from static to dynamic call graphs

requirements/goals (3)

- other modes for efficiency or to break deadlocks: with buffered* sends, reordering, non-blocking sends, ...



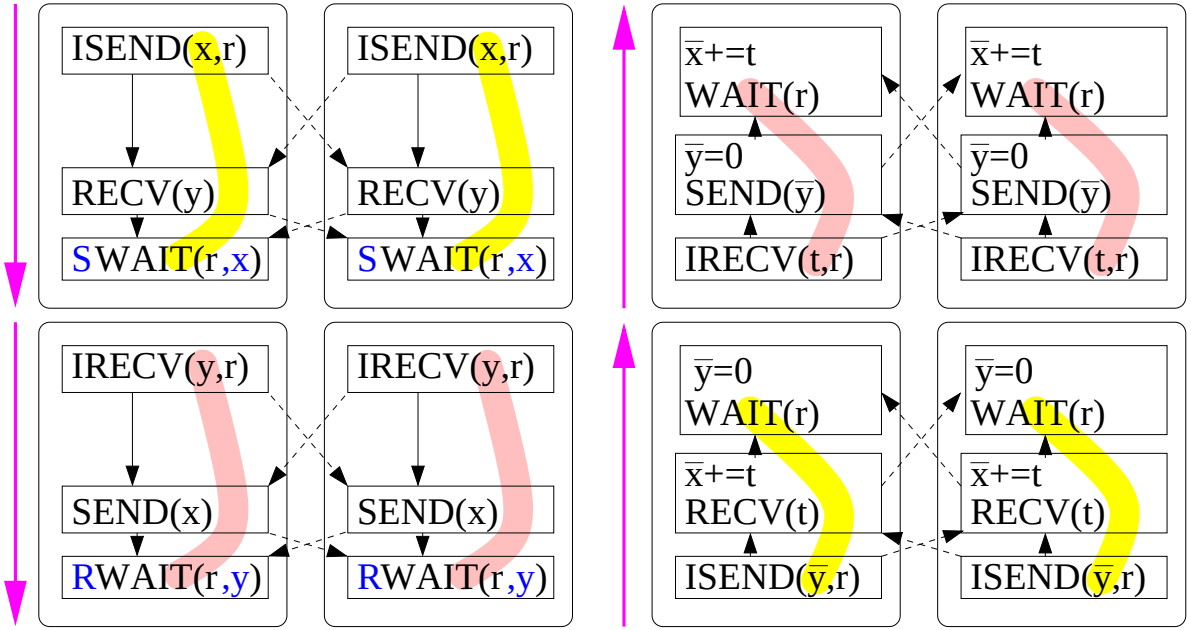
the last idiom is used in MITgcm

*resource starvation?

- justification to consider all the communication modes
- nonblocking for performance enhancements $\rightarrow$ adjoint performance ?

options for non-blocking reversal

- ensure correctness ⇒ use nonblocking calls in the adjoint



- transformations are provably correct
- convey context ⇒ enables a transformation recipe per call (extra parameters and/or split interfaces into variants)
- promises to not read or **write** the respective buffer

recipes

f as program $\mathcal{P}$ with adjoint part $\bar{\mathcal{P}}$

in $\mathcal{P}$		in $\bar{\mathcal{P}}$	
call	paired with	call	paired with
`isend(a,r)`	`wait(r)`	`wait(r);`$\bar{\texttt{a}}$`+=t`	`irecv(t,r)`
`wait(r)`	`isend(a,r)`	`irecv(t,r)`	`wait(r)`
`irecv(b,r)`	`wait(r)`	`wait(r);`$\bar{\texttt{b}}$`=0`	`isend(`$\bar{\texttt{b}}$`,r)`
`wait(r)`	`irecv(b,r)`	`isend(`$\bar{\texttt{b}}$`,r)`	`wait(r)`
`bsend(a)`	`recv(b)`	`recv(t);`$\bar{\texttt{a}}$`+=t`	`bsend(`$\bar{\texttt{b}}$`)`
`recv(b)`	`bsend(a)`	`bsend(`$\bar{\texttt{b}}$`);`$\bar{\texttt{b}}$`=0`	`recv(t)`
`ssend(a)`	`recv(b)`	`recv(t);`$\bar{\texttt{a}}$`+=t`	`ssend(`$\bar{\texttt{b}}$`)`
`recv(b)`	`ssend(a)`	`ssend(`$\bar{\texttt{b}}$`);`$\bar{\texttt{b}}$`=0`	`recv(t)`

communication patterns use multiple "rules"

e.g., the adjoint of `ibsend(a,r)` $\rightarrow$ `recv(b)` $\rightarrow$ `wait(r)` follows rule 2 for `wait` and rule 5 for `recv` to yield

`irecv(t,r)` $\rightarrow$ `bsend(`$\bar{\texttt{b}}$`);`$\bar{\texttt{b}}$`=0` $\rightarrow$ `wait(r);`$\bar{\texttt{a}}$`+=t`.

papers at EuroPVM/MPI 2008, PDSEC 2009, ICCS 2010,...

so, what is the problem?

since the theory papers were published, there has been no comprehensive implementation of the adjoining recipes

- what does exist are prototypes for:
  - specific AD tools
  - certain communication patterns
  - specific MPI implementations
  - specific target languages
- implementations are fragile
- but all prototypes agree on the design - as a wrapper library

What do we need?

- a common set of interfaces, the *"adjoinable MPI"*, promising standardized behavior
- independence from the MPI implementation
- bindings for the target languages C, C++, Fortran (incl. F77).
- independence from the AD tool implementation:
  - source transformation vs. operator overloading
  - association by name vs. by address
- a shared implementation of the common parts

new implementation started in Fall 2012

involves:

- Hascoët (INRIA Sophia Antipolis / Tapenade)
- Naumann/Schanen (RWTH Aachen / dco)
- Utke (Adol-C, OpenAD)

early identified constraints:

- cannot pass buffer arrays or contexts as structured type in F77
- be able to mix adjoinable and "passive" communications
- preserve option to recompute MPI call parameters

a first example...

example - original code

```
1   #include <mpi.h>
2   int head(double* x, double *y) {
3     MPI_Request r;
4     int world_rank;
5     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
6     if (world_rank == 0) {
7       *x=*x*2;
8       MPI_Isend (x, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD,&r);
9       MPI_Recv (y, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
10      MPI_Wait (&r,MPI_STATUS_IGNORE);
11      *y=*y*3;
12    } else if (world_rank == 1) {
13      double local;
14      MPI_Recv (&local, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
15      local=sin(local);
16      MPI_Isend (&local, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,&r);
17      MPI_Wait (&r,MPI_STATUS_IGNORE);
18    }
19  }
```

example - adapted to adjoinable MPI (and Adol-C)

```
1  #include "ampi/ampi.h"
2  #include "adolc/adolc.h"
3  int_head(adouble* x, adouble *y) {
4    AMPI_Request r;
5    int world_rank;
6    AMPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
7    if (world_rank == 0) {
8      *x=*x*2;
9      AMPI_Isend(x, 1, AMPI_ADOUBLE, 1, 0, AMPI_RECV, MPI_COMM_WORLD,&r);
10     AMPI_Recv (y, 1, AMPI_ADOUBLE, 1, 0, AMPI_ISEND_WAIT, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
11     AMPI_Wait(&r,MPI_STATUS_IGNORE);
12     *y=*y*3;
13   } else if (world_rank == 1) {
14     adouble local;
15     AMPI_Recv (&local, 1, AMPI_ADOUBLE, 0, 0, AMPI_ISEND_WAIT, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
16     local=sin(local);
17     AMPI_Isend(&local, 1, AMPI_ADOUBLE, 0, 0, AMPI_RECV, MPI_COMM_WORLD,&r);
18     AMPI_Wait(&r,MPI_STATUS_IGNORE);
19   }
20 }
```

- added pairing enumeration as context parameter
- `AMPI_Request` can be just an `MPI_Request`
- typing distinguishes active and passive

map to common implementation part (I)

via mapping layer (generic for operator overloading)

```
1  int AMPI_Isend (void* buf,
2                  int count,
3                  MPI_Datatype datatype,
4                  int dest,
5                  int tag,
6                  AMPI_PairedWith
                           pairedWith,
7                  MPI_Comm comm,
8                  AMPI_Request* request) {
9    return FW_AMPI_Isend(buf,
10                          count,
11                          datatype,
12                          dest,
13                          tag,
14                          pairedWith,
15                          comm,
16                          request);
17 }
```

- obtain a contiguous array of values from `buf`; depends on the implementation of the active type
- do the send of the contiguous array
- keep internal information about the request
- create a trace entry and retain **all** the information needed for the adjoint in an augmented request

source transformation creates the call directly

```
1  FW_AMPI_Isend(x,
2    1,
3    AMPI_ADOUBLE,
4    1,
5    0,
6    AMPI_RECV,
7    MPI_COMM_WORLD,
8    &r);
```

- may not need to extract the values if using association by name
- do the send
- keep internal information about the request
- store parameters that can't be recomputed for the call to be generated in $\bar{\mathcal{P}}$

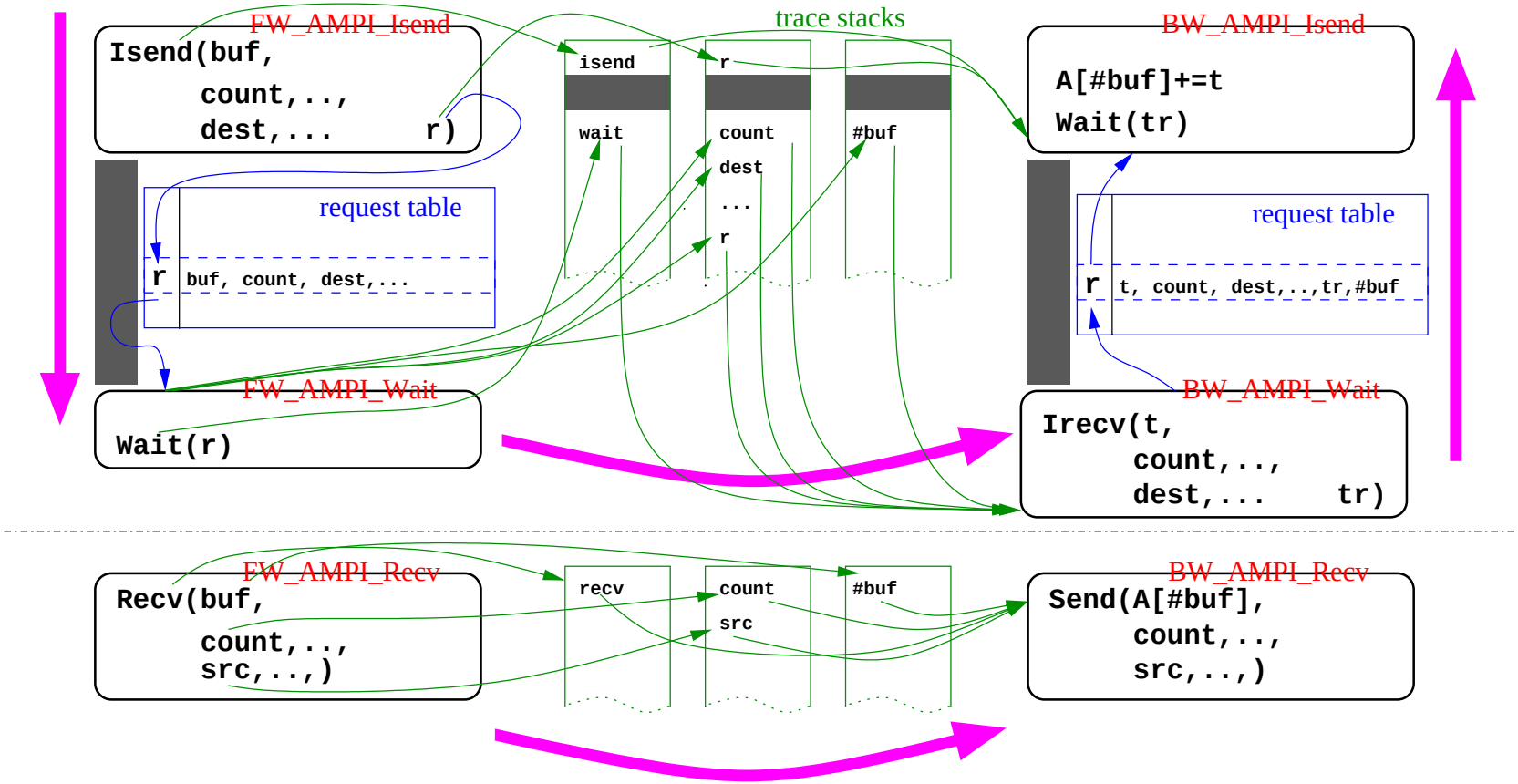
map to common implementation part (II)

```
int FW_AMPI_Isend (void* buf,
                   int count,
                   MPI_Datatype datatype,
                   int dest,
                   int tag,
                   AMPI_PairedWith pairedWith,
                   MPI_Comm comm,
                   AMPI_Request* request) {
  int rc;
  if (!(pairedWith==AMPI_RECV || .... ))
    rc=MPI_Abort(comm, MPI_ERR_ARG);
  else {
    rc= MPI_Isend(ADTOOL_AMPI_rawData(buf
          ,&count),
                  count,
                  datatype,
                  dest,
                  tag,
                  comm,
#ifdef AMPI_FORTRANCOMPATIBLE
                  request
#else
                  &(request->plainRequest)
#endif
                  );
```

... continue ...

```
    struct AMPI_Request_S *ampiRequest;
#ifdef AMPI_FORTRANCOMPATIBLE
    struct AMPI_Request_S ampiRequestInst;
    ampiRequest=&ampiRequestInst;
    ampiRequest->plainRequest=*request;
#else
    ampiRequest=request;
#endif
    /* fill in the other info */
    ampiRequest->endPoint=dest;
    ampiRequest->tag=tag;....
    ADTOOL_AMPI_mapBufForAdjoint(
          ampiRequest,buf);
    ampiRequest->tracedRequest=ampiRequest
          ->plainRequest;
#ifdef AMPI_FORTRANCOMPATIBLE
    BK_AMPI_put_AMPI_Request(ampiRequest);
#endif
    if (ADTOOL_AMPI_isActiveType(datatype)==
          AMPI_ACTIVE) {
      ADTOOL_AMPI_push_CallCode(
          AMPI_ISEND);
#ifdef AMPI_REQUESTONTRACE
      ADTOOL_AMPI_push_request(ampiRequest
          ->tracedRequest);
#endif
    }
  }
  return rc;
}
```

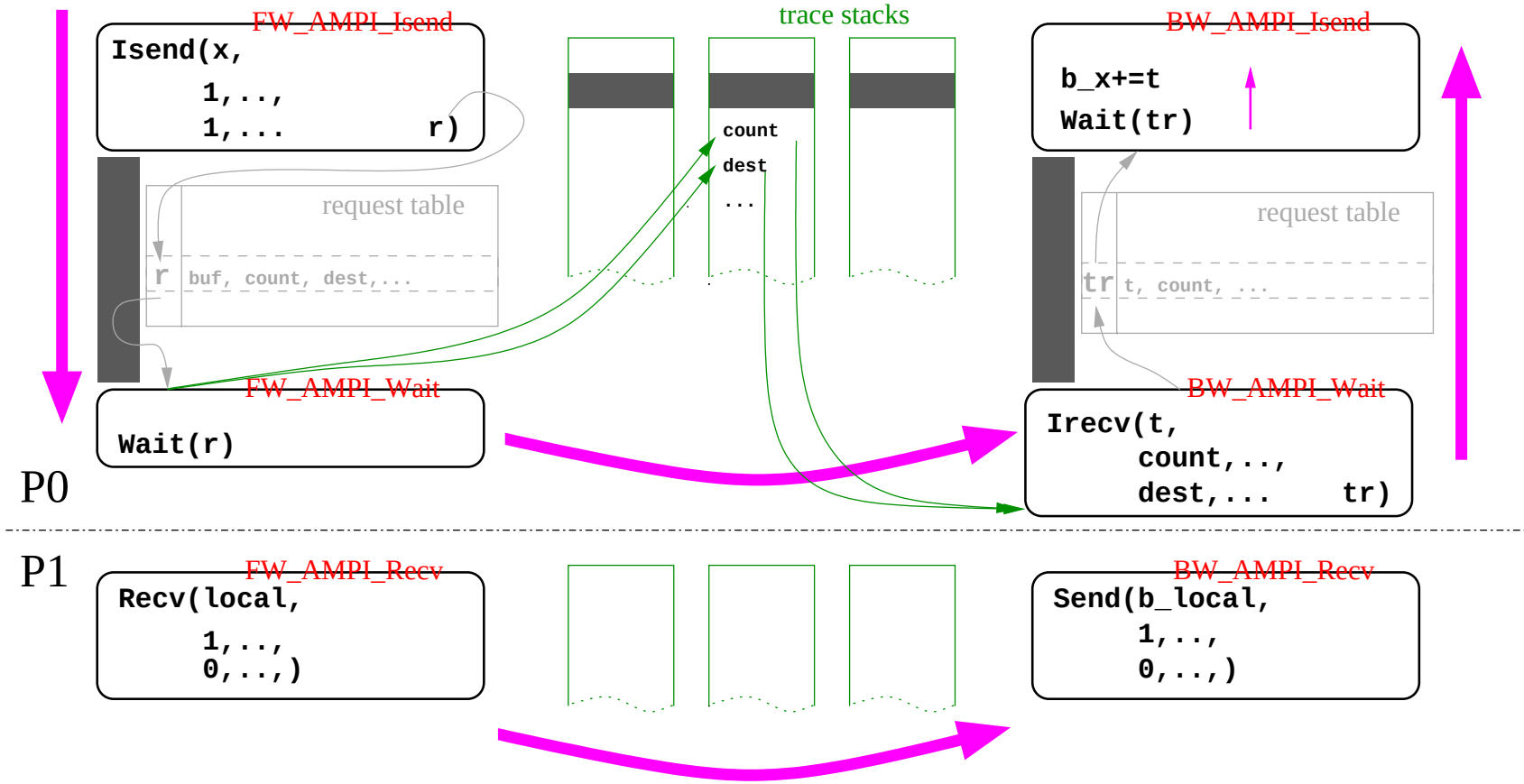
how to handle the parameters with operator overloading ?



intend for sharing:

- FW/BW implementations
- bookkeeping in the request table; put/get
- stack of MPI call parameters (with opaque type)

less tracing for source transformation



- optional actions by call backs, e.g.
  `ADTOOL_AMPI_push_CallCode(AMPI_ISEND);`
- optional request bookkeeping configurable in the common implementation

but there is a caveat

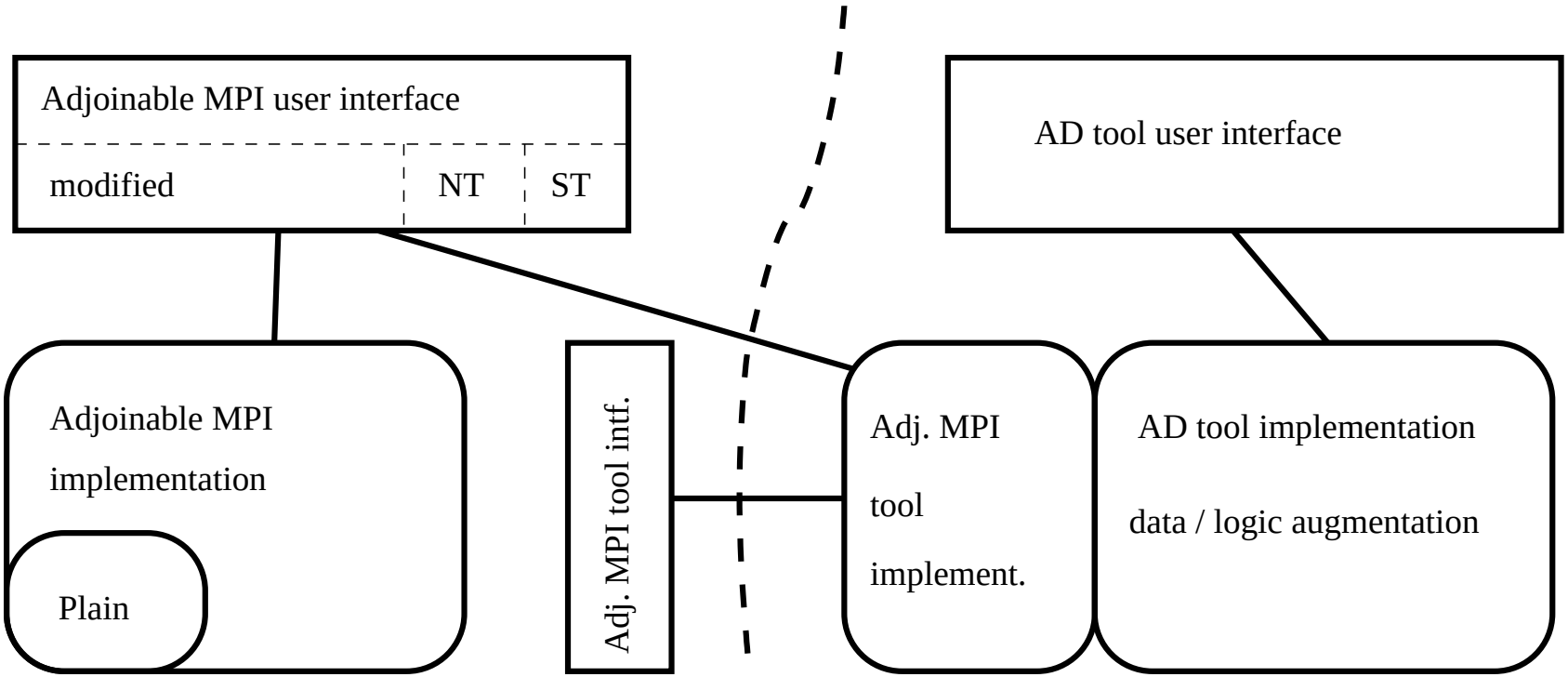
map to common implementation part (III)

```
int BW_AMPI_Isend (void* buf,
                   int count,
                   MPI_Datatype datatype,
                   int dest,
                   int tag,
                   AMPI_PairedWith pairedWith,
                   MPI_Comm comm,
                 za AMPI_Request* request) {
  int rc;
  MPI_Request *plainRequest;
  struct AMPI_Request_S *ampiRequest;
#ifdef AMPI_REQUESTONTRACE
  MPI_Request tracedRequest;
#endif
#ifdef AMPI_FORTRANCOMPATIBLE
  struct AMPI_Request_S ampiRequestInst;
  ampiRequest=&ampiRequestInst;
  plainRequest=request;
#else
  ampiRequest=request;
  plainRequest=&(ampiRequest->plainRequest);
#endif
#if defined AMPI_FORTRANCOMPATIBLE || defined AMPI_REQUESTONTRACE
#ifdef AMPI_REQUESTONTRACE
  tracedRequest=ADTOOL_AMPI_pop_request();
  BK_AMPI_get_AMPI_Request(&tracedRequest,ampiRequest,1);
#else
  BK_AMPI_get_AMPI_Request(plainRequest,ampiRequest,0);
#endif
#endif
  assert(ampiRequest->origin==AMPI_SEND_ORIGIN) ;
```

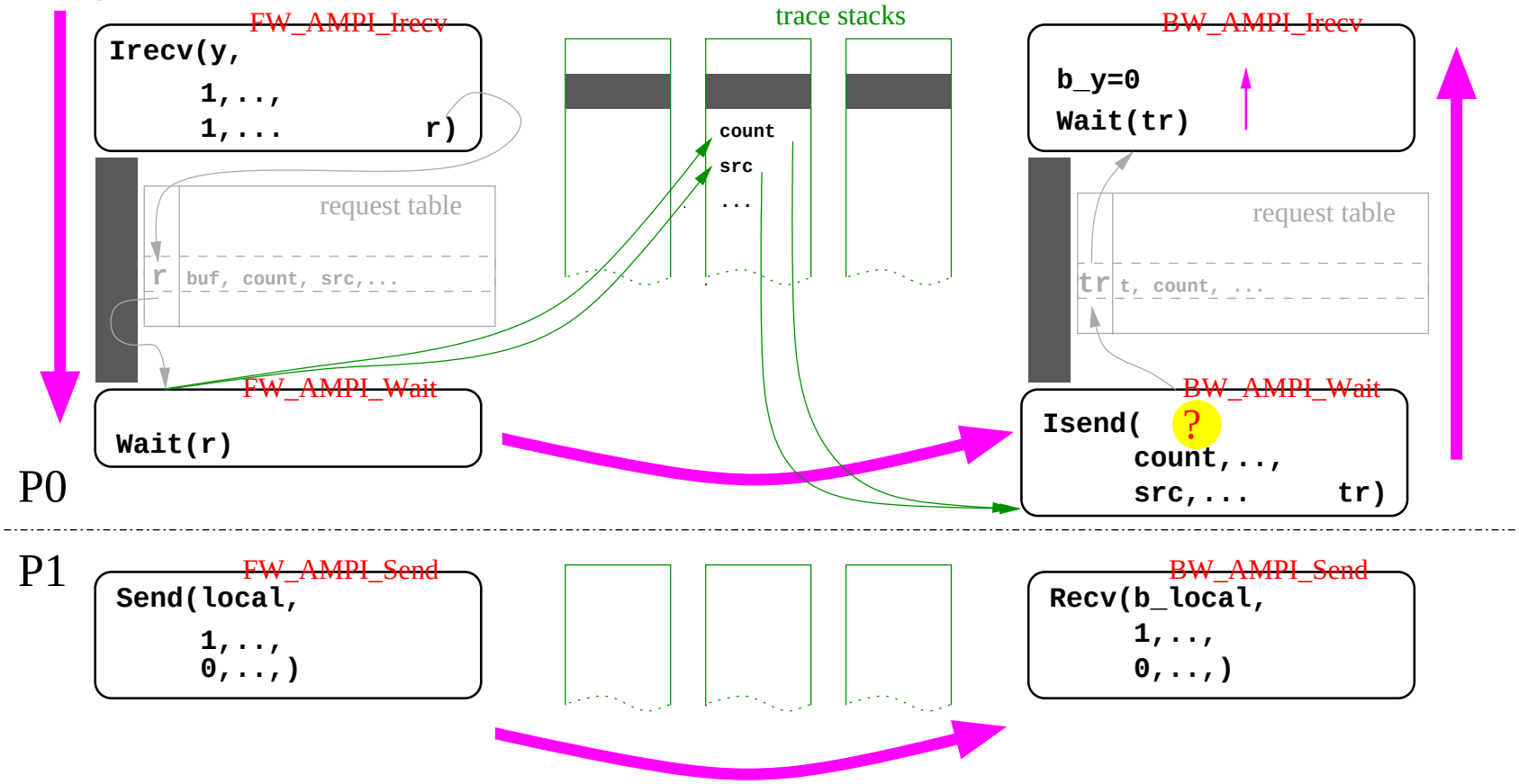
map to common implementation part (IV)

```
if (!(
        ampiRequest->pairedWith==AMPI_RECV
        ||
        ampiRequest->pairedWith==AMPI_IRECV_WAIT
        ||
        ampiRequest->pairedWith==AMPI_IRECV_WAITALL
        )) rc=MPI_Abort(comm, MPI_ERR_ARG);
else {
  switch(ampiRequest->pairedWith) {
  case AMPI_RECV:
  case AMPI_IRECV_WAIT: {
    rc=MPI_Wait(plainRequest, MPI_STATUS_IGNORE);
    ADTOOL_AMPI_adjointIncrement(ampiRequest->adjointCount,
                                 ampiRequest->datatype,
                                 ampiRequest->comm,
                                 ampiRequest->buf,
                                 ampiRequest->adjointBuf,
                                 buf,
                                 ampiRequest->adjointTempBuf,
                                 ampiRequestampiRequest->idx);
    ADTOOL_AMPI_releaseAdjointTempBuf(ampiRequest->adjointTempBuf);
    break;
  }
  default:
    rc=MPI_Abort(ampiRequest->comm, MPI_ERR_TYPE);
    break;
  }
}
  return rc;
}
```

library structure



missing the buffer association

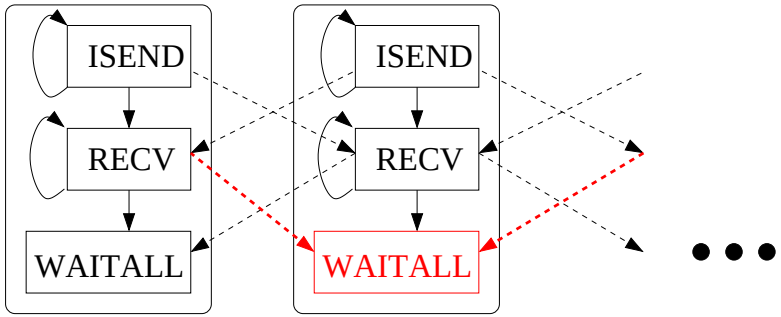


- not an issue with operator overloading; dynamic mapping y→buf→#buf→A[#buf]
- source transformation does static mapping y→b_y
- stop gap - add the buffer as parameter to the AMPI_Wait ?

expand the scope

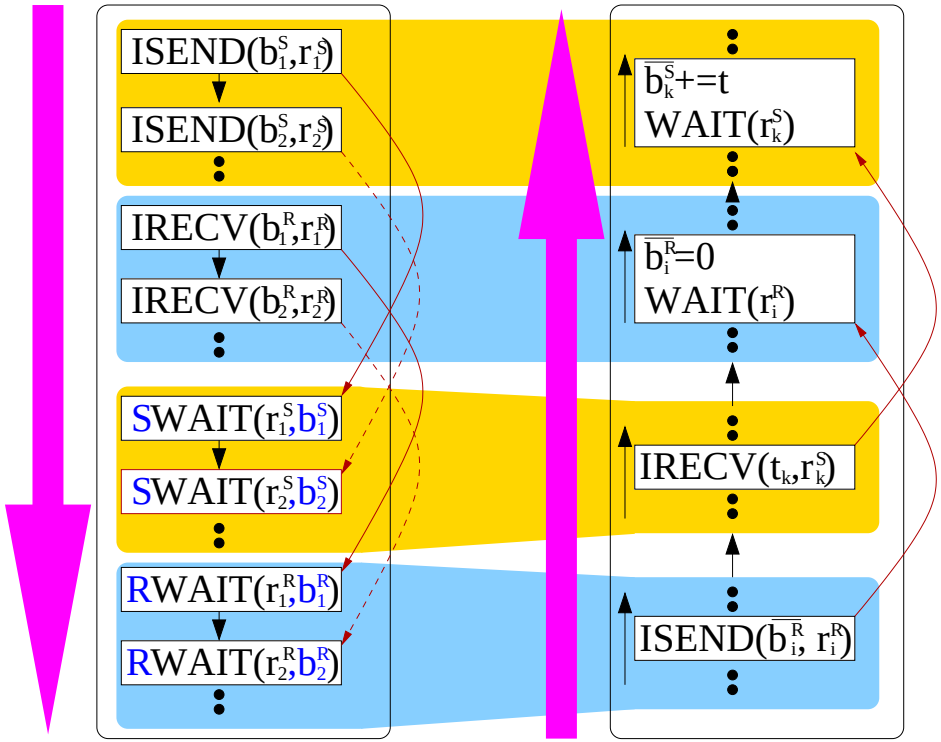
consider collective waits

- would have to pass buffer array
- knew the problem (started with AMPI for OpenAD)
- band aids vs solutions?



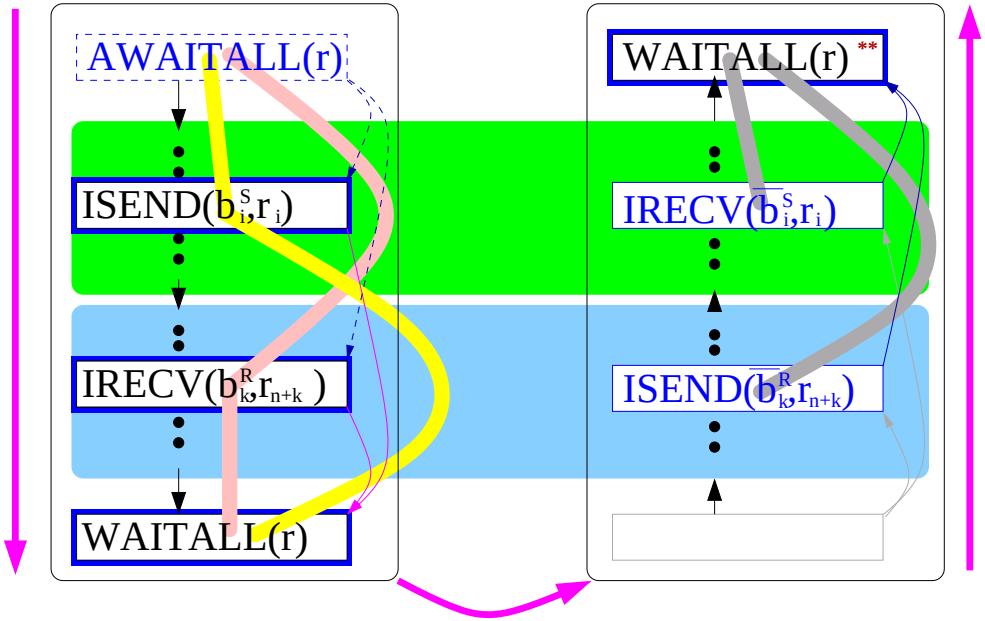
band aid "A":

- split the collective wait
- pass individual buffers
- imposes artificial order
- can be a nontrivial code change



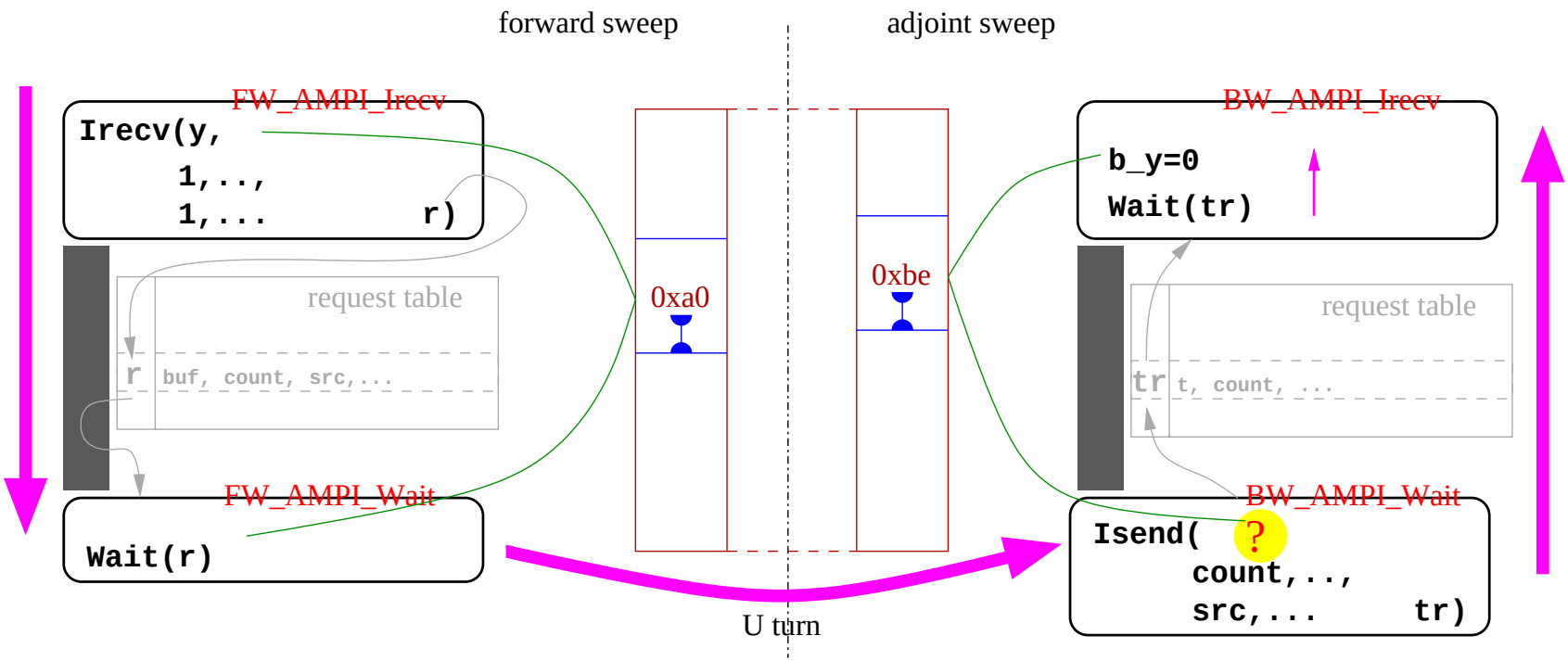
option “B”

- introduce an *anti_wait*
- requires backward extension of promises re. buffers to anti_wait
- symmetric communication patterns between processes are “easily” adapted to the symmetric *anti_wait*



- maximizing adjoint communication/computation overlap requires rearranging code; possible if we have a symmetric `anti_wait–wait` section
- deriving adjoint `anti_wait` recipes and proving correctness is hard if there is no symmetric “representer” pattern
- non-symmetric cases perhaps not so relevant for our class of applications

option “C” - dynamically mapping memory



relies on a pointer mapping algorithm; abstract description developed in 2012 (with Hascoët) for general purpose adjoining in the presence of pointers

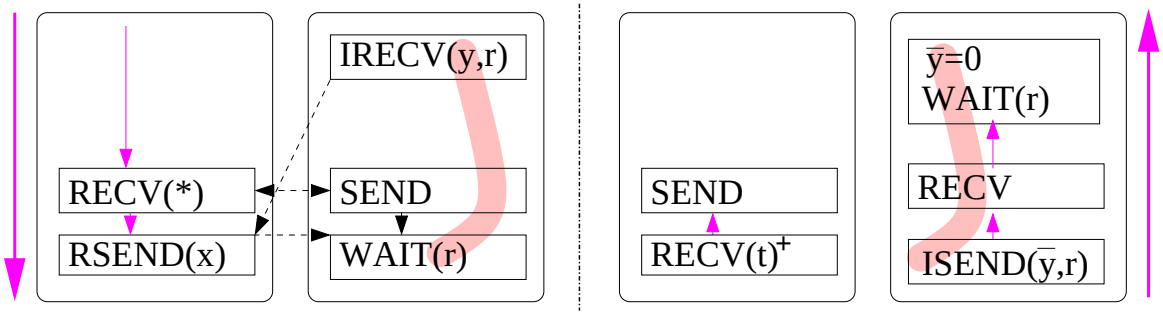
about the pointer mapping algorithm

- track base address and offset of pointer values
- maintain addresses map from forward to adjoint sweep (for symbols, dynamic memory)
  - implies runtime overhead for address mapping & offset tracking
  - needs (static) source code analysis to separate benign pointer uses from the ones that need mapping and trigger mapping when pointee becomes unavailable ("last chance") rather than mapping for each pointee reference
  - not yet implemented (because it is a significant effort)
  - but needed for most uses of pointers in adjoints
- possible simplifications allowing pointer values to be used without mapping in the adjoint sweep
  - F77 static allocation mode
  - "joint" reversal, i.e. U-turn always happens before leaving the stack frame; must not deallocate heap memory; implies recomputations with overhead depending on depth of callstack

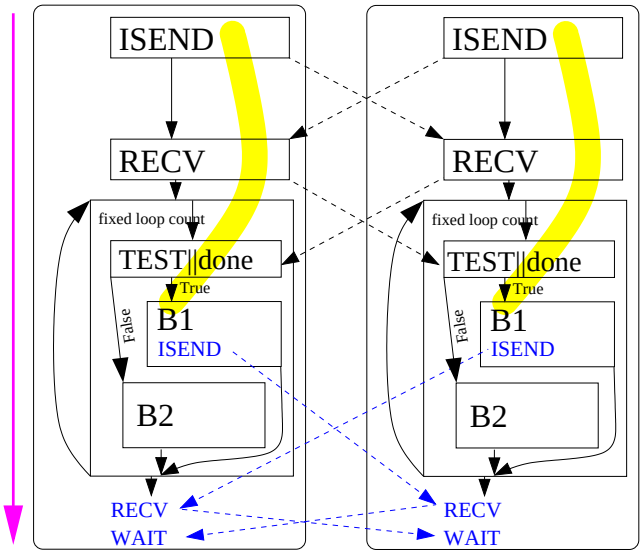
including option "B" until pointer mapping is robust

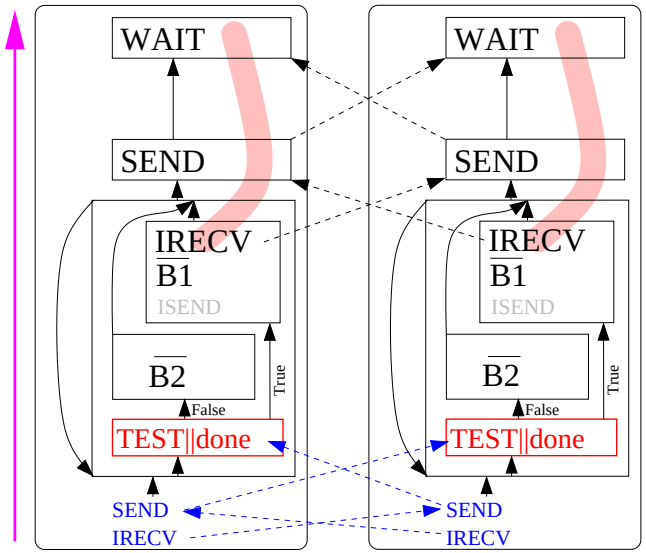
limitations & difficult MPI features (1)

- cannot retain efficiency advantage of `MPI_Rsend`; no `MPI_Rrecv`; adjoint sends back to source value of wildcard recv



- two communication phases using `MPI_Test` to maximize overlap - for the adjoint this requires capturing both phases as context





limitations & difficult MPI features (2)

- dynamically acquired/released resources (but treatable like dynamic memory)
- wildcard receives are recorded as sequence of sources and the adjoint will replay that sequence in that (artificial) order
- for user-defined MPI data types integrate with active MPI type defined in lib. interface
- user defined `MPI_op` (tricky for operator overloading)
- one-sided communication (next talk)
- ...

things that are relatively "easy"

- single call collective communications (e.g. `MPI_Bcast`, `MPI_Reduce`) with standard operations; have some efficiency implications
- global setup/teardown - is transparent if it encloses both the forward and adjoint sweep (NT version)

Summary:

- turned earlier theoretical concept of adjoinable MPI into an transparent implementation
- account for statically undecidable communication graph
- account for different AD tool implementation options / target languages
- avoid non-scalable internal data
- common interface and partially shared implementation is possible and evolving

  (http://trac.mcs.anl.gov/projects/AdjoinableMPI)