Adjoints of MPI programs

- based on discussions with Paul Hovland, Laurent Hascoët, Uwe Naumann, Bill Gropp, Darius Buntinas
- wanted by: Chris and Patrick
- general things about MPI
- MPI use in MITgcm
- a solution prototype in OpenAD

UChicago ▶ Argonne LLC

MPI - data transfer between processes

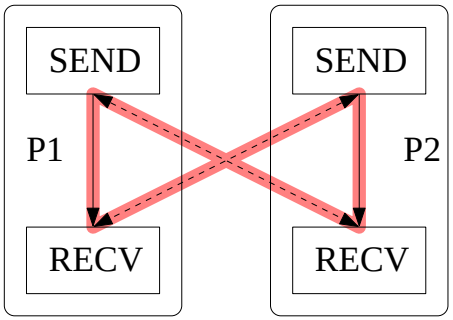
- in MITgcm: exchange tile halos, reductions operations. synchronization...
- simple MPI program needs 6 calls :

```
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
```

- 287 routines standardized by MPI-2
- covering: communication, setup, grouping of processes, I/O, status queries, topologies, debugging,...
- send modes: `mpi_[i][b|s|r]send`
- receive modes: `mpi_[i]recv`
- ensure correctness and improve efficiency $\Rightarrow$ want the same for adjoints

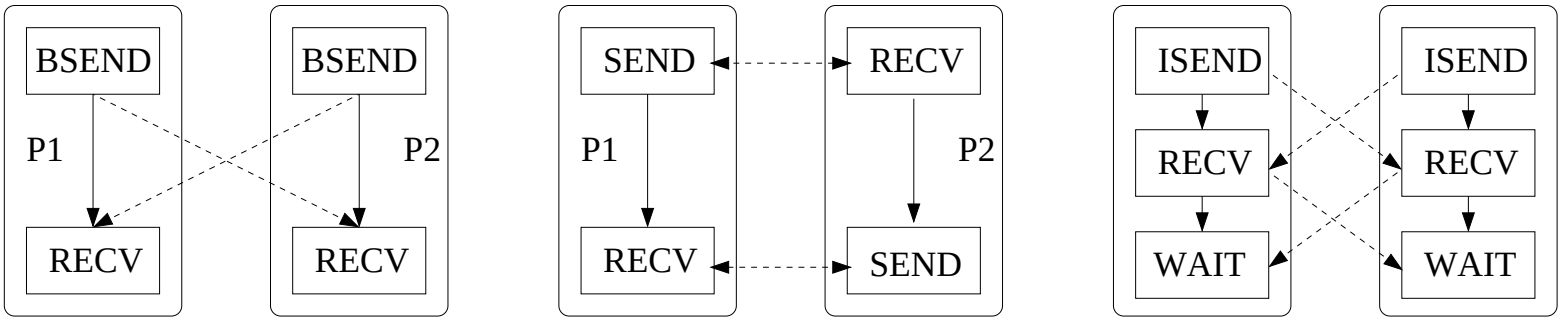
correctness as in ...

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



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

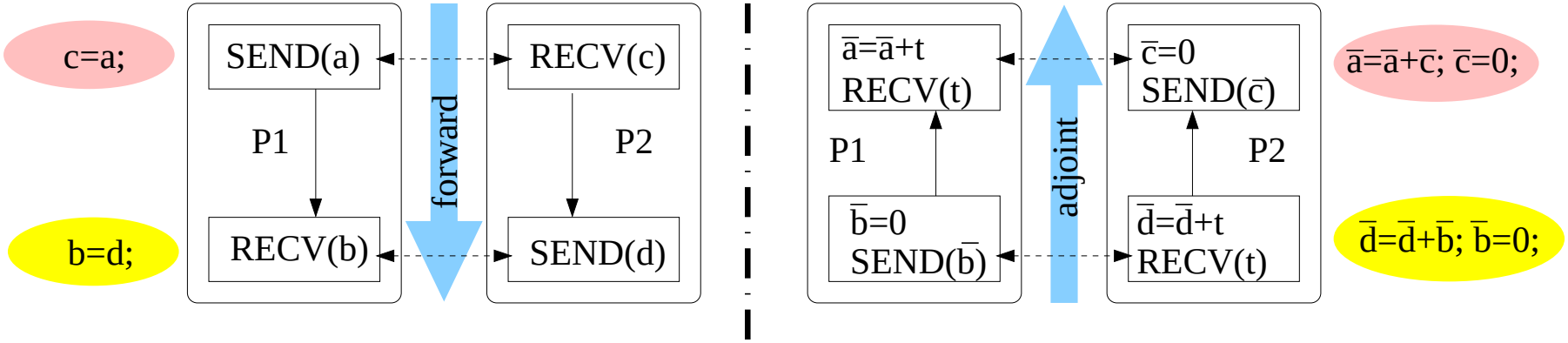
- break with buffered* sends, reordering, non-blocking sends, ...



the last idiom is used in MITgcm

*resource starvation?

easy adjoints for blocking calls



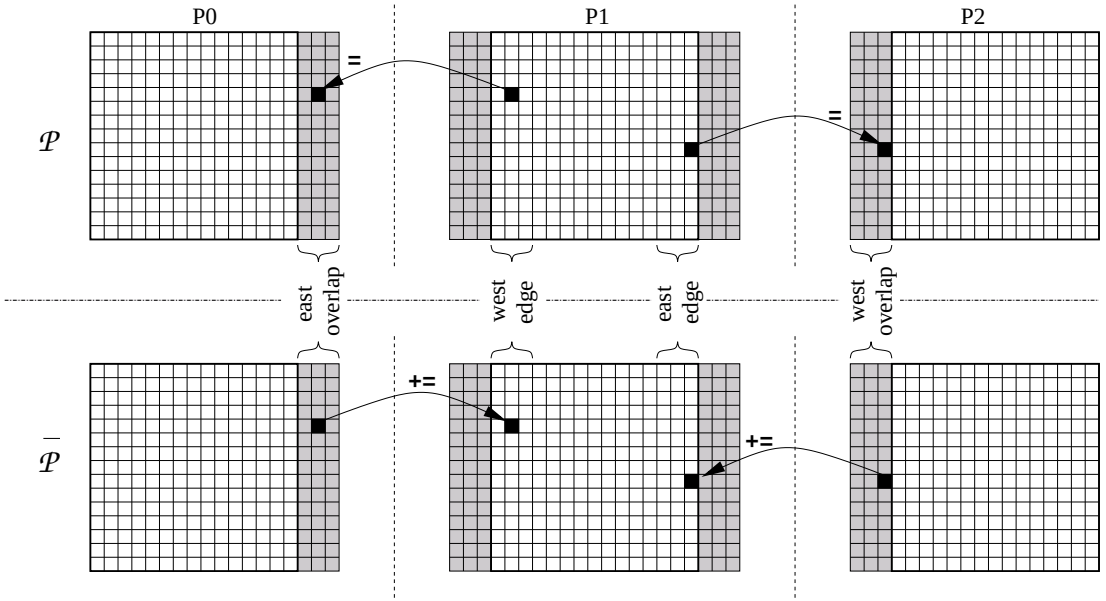
- *easy* adjoint transformation: `send` $\mapsto$ `recv` and `recv` $\mapsto$ `send`
- hyp.: if the forward communication graph is acyclic, so is the adjoint; look at the communication graph with reversed edges
- for activity analysis: difficult to statically determine send/recv pairs; e.g. consider 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

previously on "AD and MPI"

not exhaustive and in no particular order:

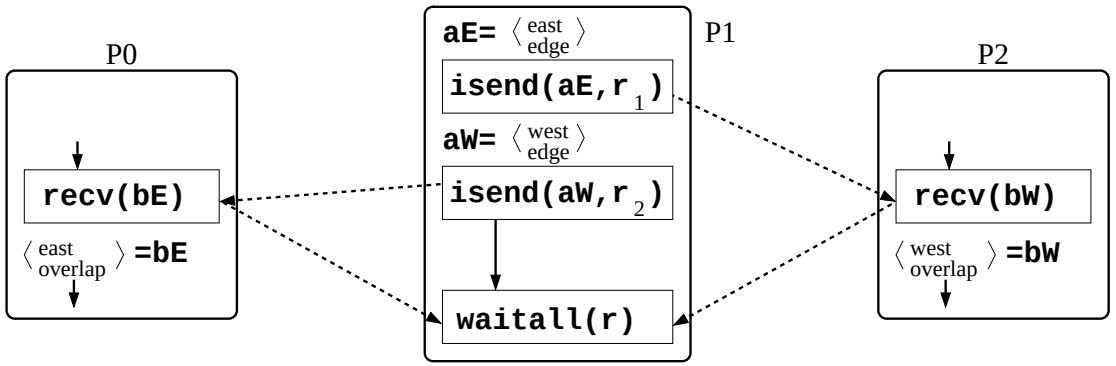
- Griewank: first ed. of "the book" had 2 pages on parallel programs; second edition has more
- Hovland: thesis *"AD of parallel programs"* - mostly forward
- Hovland/Bischof: *"Automatic Differentiation for Message-Passing Parallel Programs"* - association between value and derivative
- Carle/Fagan: *"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"* - ditto
- 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

why the wrappers in MITgcm?



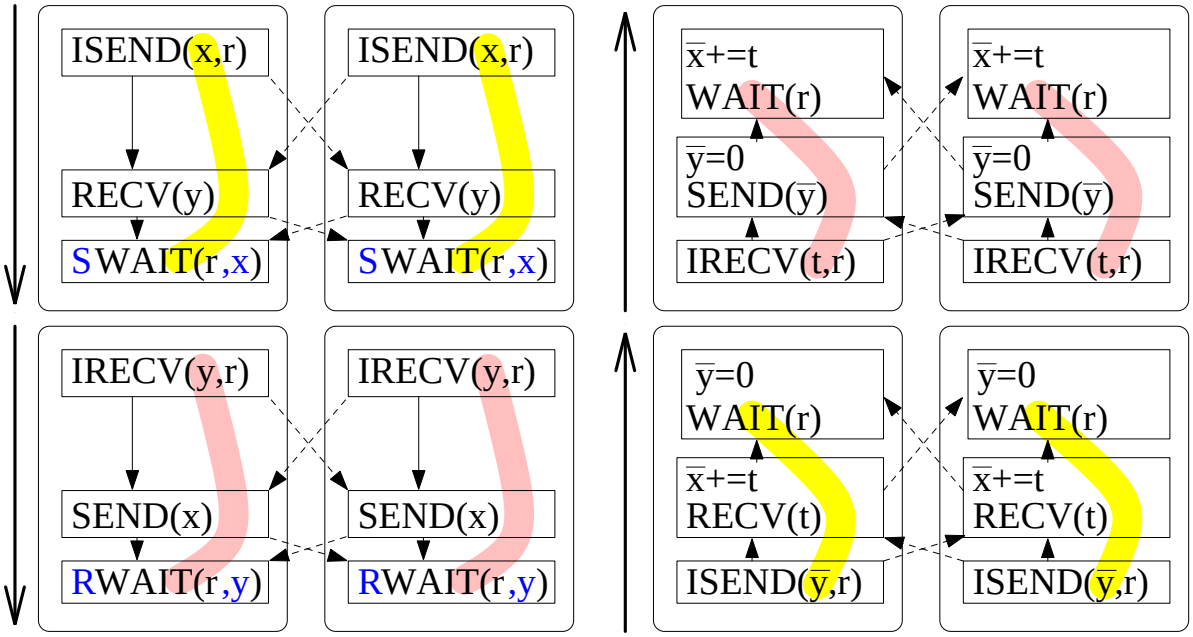
actually in E-W and N-S; need to consider corners...

- every tile needs to talk to its neighbors
- prevent deadlocks by imposing ordering ? (can be a bit complicated depending on the decomposition)
- 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; in MITgcm EXCH routines have hand-written adjoints

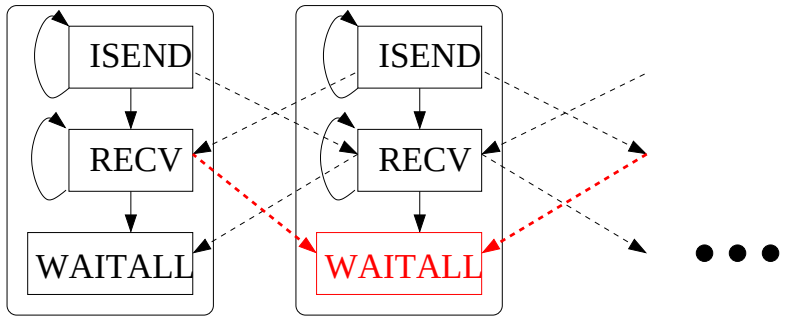


no easy transformation because ...

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

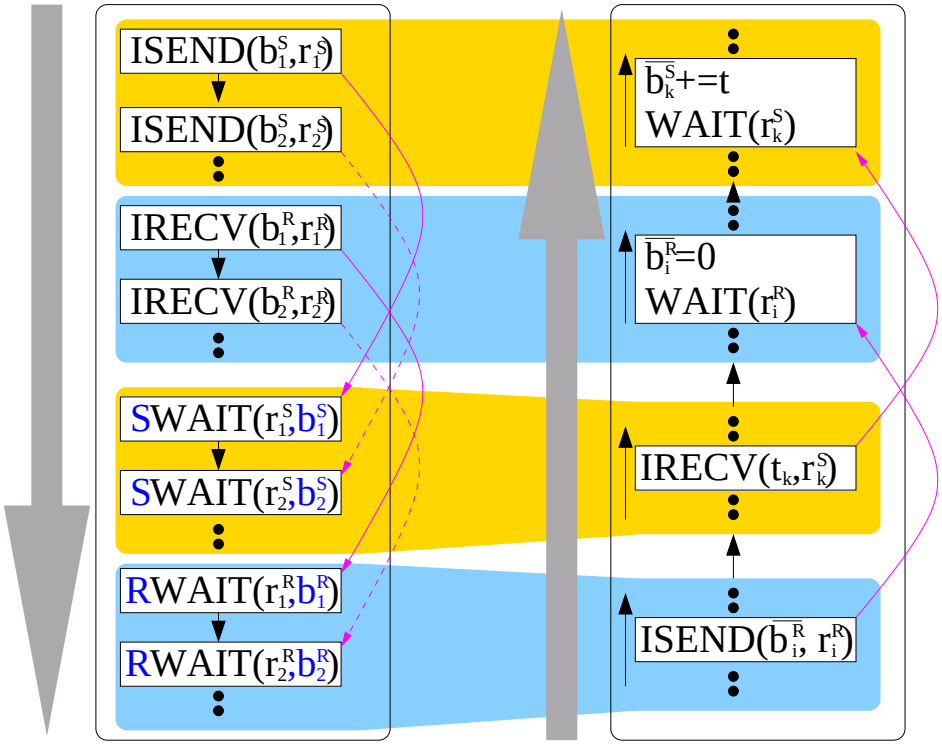


- the above transformations are provably correct
- **extensions** to convey context
  ⇒ enables a transformation recipe per call
- promises to not read or **write** the respective buffer



...solution A

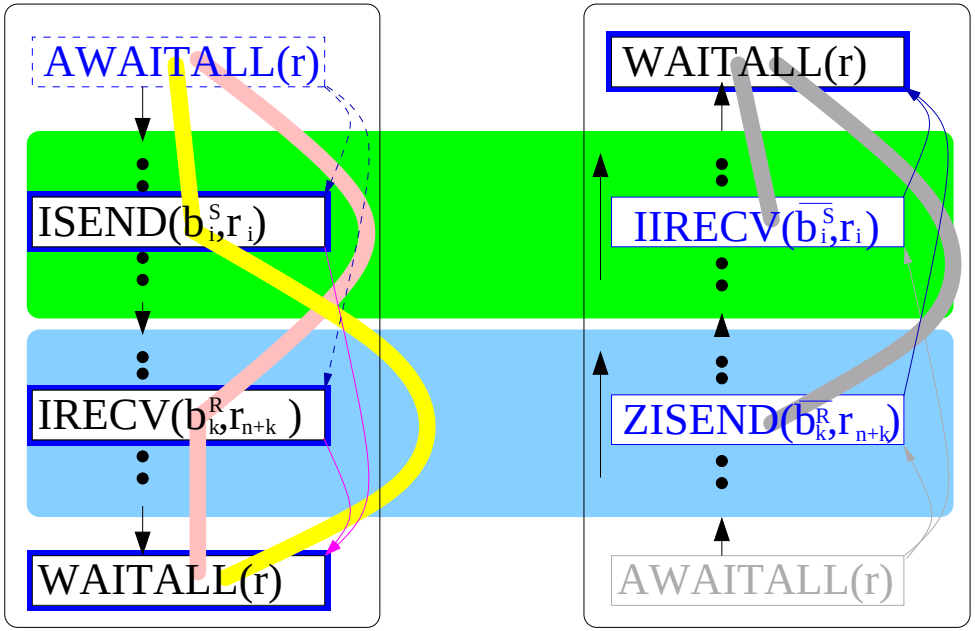
- require individual, marked WAITs (achieves 1 on 1 comm edges)



- extra arguments permit simple transformation recipe with (otherwise) indistinguishable request and buffer arrays
- but individual WAITs impose artificial order ⇒ bad for performance

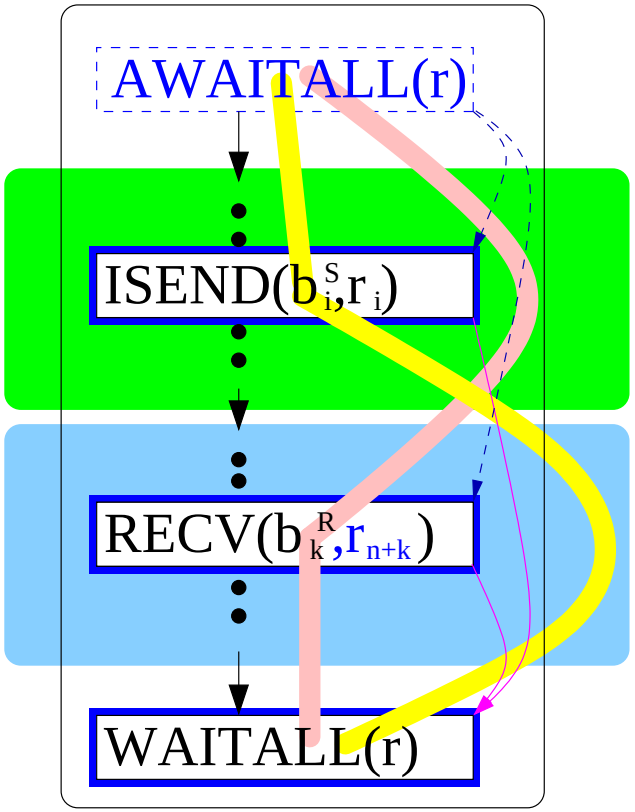
...solution B

- 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

in the OpenAD prototype




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

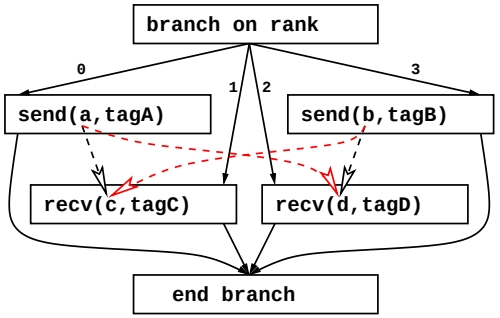
```
call ampi_isend(westSendBuf_RL(1,eBl,bi,bj),&
                theSize, theType, theProc, theTag, &
                MPI_COMM_MODEL,&
                exchReqIdX(pReqI,1,bi,bj), &
                exchNReqsX(1,bi,bj),&
                mpiStatus , mpiRc)
```

```
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)
```

what about <*insert problem here*> ?

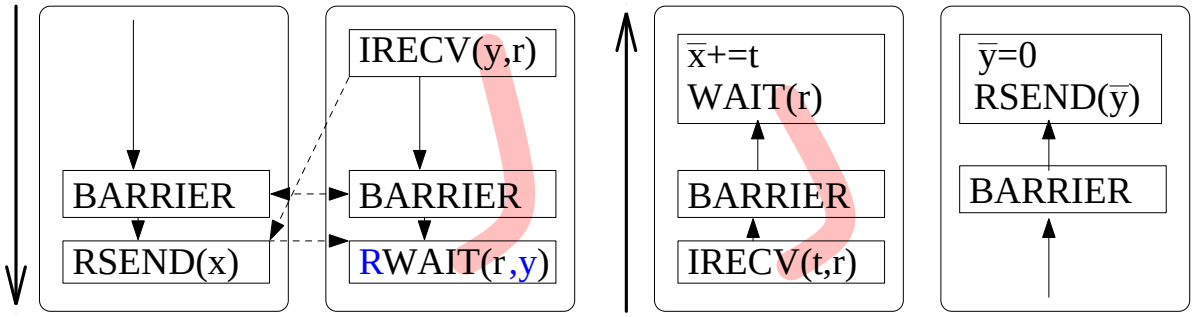
- solutions A & B disambiguate edges in dynamic comm. graph
- don't do much for static analysis:
  - e.g. modeled with MPI-enhanced CFG (Strout/Hovland/Kreaseck)
  - useful for activity analysis



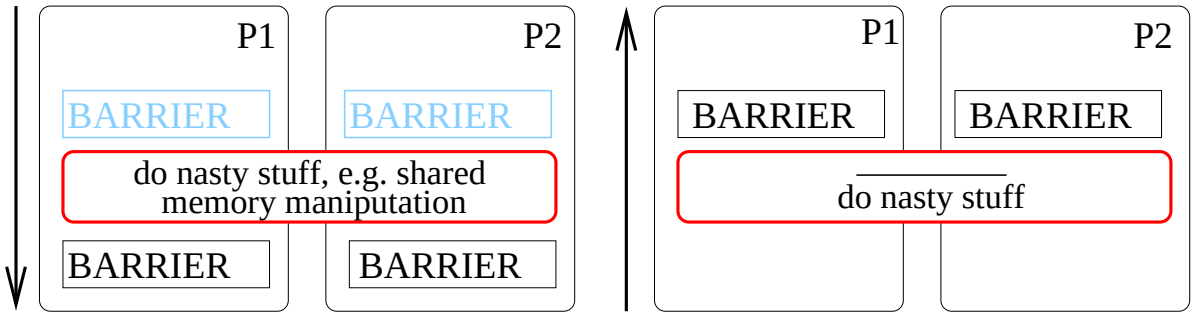
- would like to identify communication "channels" using
  - pragmas (can make up anything but no external support), or
  - aspects (existing systems w support, suggested by B. Gropp; no *"Aspecttran"*)
- group certain send - recv - wait / certain collective and barrier calls
  - would like runtime validity checks
  - reusable for analysis, debugging, adjoint generation
  - don't assume locality of channels in the source or single call location of collective ops

what about barrier ?

- simple synch point ⇒ same for adjoint, example:



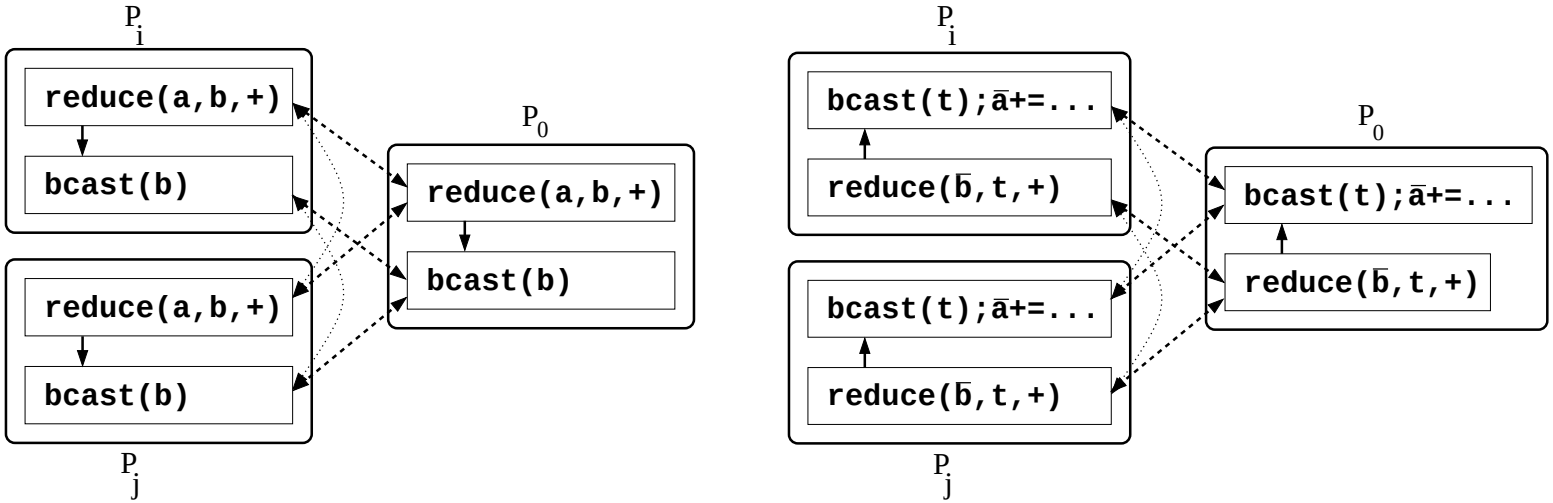
- retains pattern (rsend may improve performance by avoiding hand shake)
- barrier itself does not conceptualize a critical section



- can rationalize the need for barrier enclosed section for correctness of original program
- note - MPI's one-sided commun. has "fence" to demark section ☺

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

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
```

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

```
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
```

current developments in OpenAD

- debugging 1x1 MITgcm setup
- scarcity-based heuristics
- tracing of non-smooth model behavior
- redesign push/pop
  - to minimize tape
  - allow a variant to locally recompute during the adjoint sweep
  - framework to optimize push/pop placement
- misc. front-end fixes
- investigate candidates to replace the Open64 front-end
  (need long-term support for Fortran 200X)

aspect oriented programming

- a typical program modularization is I/O, solver, communication
- logging is a typical example of a "cross cutting concern" (not encapsulated in OOP)
- AOP aims to aid separation of concerns
- principle approach:
  - aspect: bits of new code (advice) injected into certain spots (join points) in the given program.
  - systems differ in how to specify join points (aka "point cuts"), what can be an advice (executable code, new members to classes)
  - tightly integrated with compilers
  - example 1: `int C::%(...)` matches all member functions of the class C that return an int
  - example 2: `call("void draw()") && within("Shape")` describes the set of calls to the function `draw` that are within methods of the class `Shape`
- while channels may be a cross-cutting concern they may not always be amenable to point cut specs ☹