Refactoring a Climate Simulation Miniapp to use a PGAS programming model

Andy Stone

NCAR Intern, SiParCS 2010

PhD student: Colorado State University

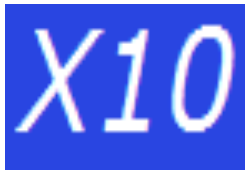
Internship Mentor: John Dennis

PhD advisor: Michelle Strout

NCAR

What are programming models

Ct



Cilk

CHAPEL

CUDA

Map-Reduce

STAPL

Parallel programming models: technologies to express parallel computations. **(languages and libraries)**

Lots of models exist

Why use them:

Improved productivity:

Fewer bugs

Smaller development time

The predominance of MPI

Predominent model is MPI

"the assembly language of parallel programming"

PGAS: partitioned global address space

- No explicit message passing
- One-sided communication
- More natural way to express parallelism

Adoption of models effected by:

- Need to work with existing code
- Need to maintain high performance
- Lack of knowledge of applicability with real-world codes
- Lack of developers experienced with these models

Studying applicability with "Real world" codes

Ideally:

Port lots of full-scale real world codes
Study these ported implementations

Realistically:

Real world codes can be huge (a million lines in climate sims.)

Experts with the code would need to do the port or offer a huge amount of support.

This support won't come unless the benefits are evident.

But "the benefits" are what we're trying to study!

Studying Benchmarks

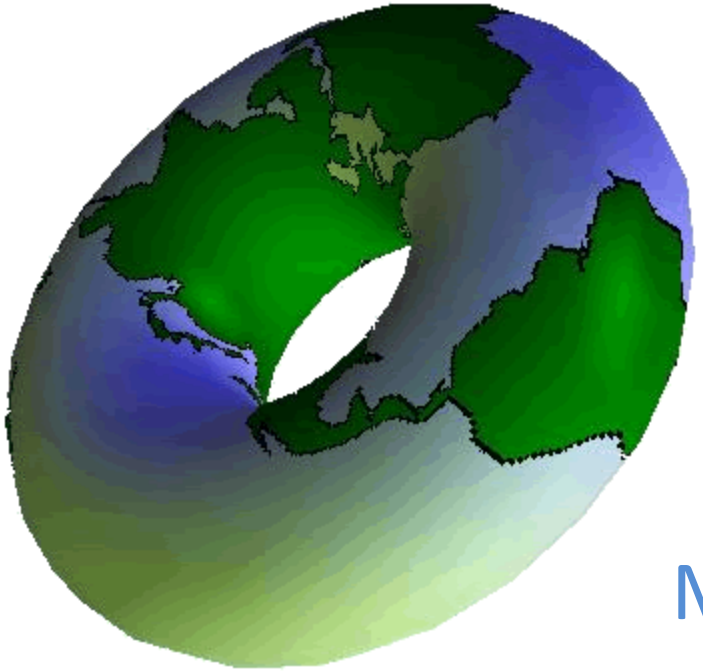
Benchmarks are small programs that perform some common type of computation.

High-performance LINPACK
NAS parallel benchmarks
HPC Challenge benchmarks

Sure, these benchmarks model different ways of number crunching but do they really test model applicability?

In many ways these benchmarks are "idealized" and the devil is in the details.

Some details



Data might not be so simple:

We don't live in donut world!

Data structures may be domain or application specific

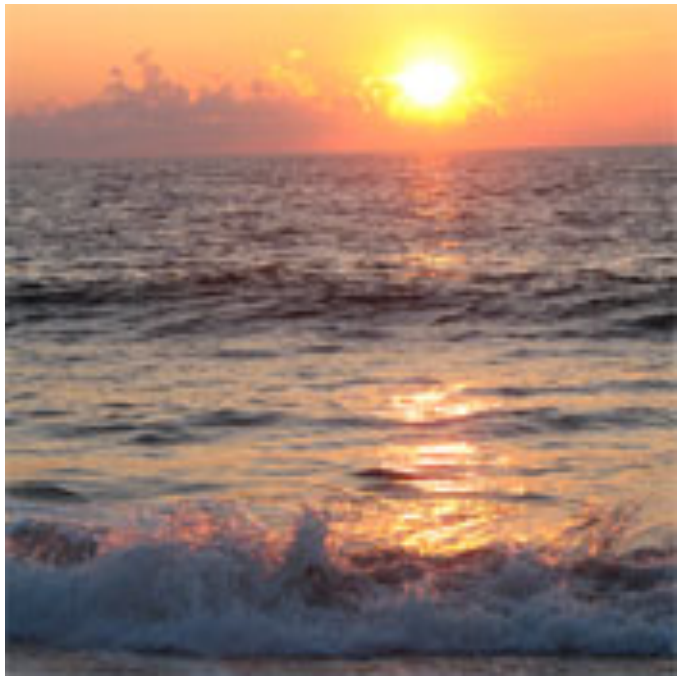
Metadata might not be so simple:

It might not be globally replicated

Solution: Miniapps!

Small pieces of code (1k - 10k SLOC) extracted from real-world applications.

The CGPOP Miniapp



This was created for this internship.

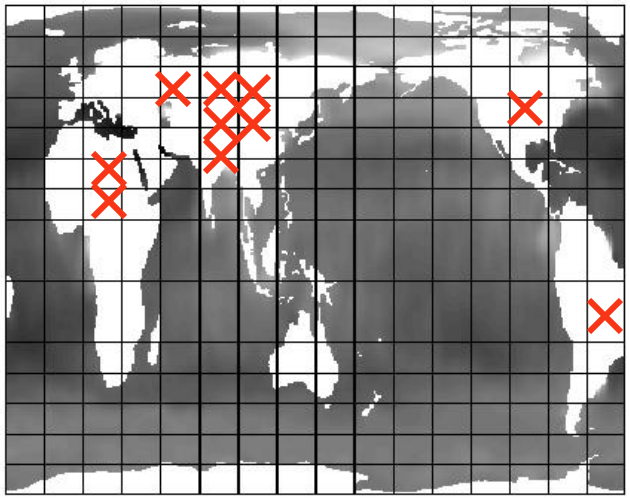
Conjugate gradient solver in POP (parallel ocean program), which is part of CCSM

About 10 KLOC

Let's look at some complicating details of CGPOP

Initial CGPOP implementation

How data is stored



Two versions:

- World is conceptually big 2D array
- World split into blocks
- Processes take ownership of blocks
- Land blocks are eliminated

In 2d version:

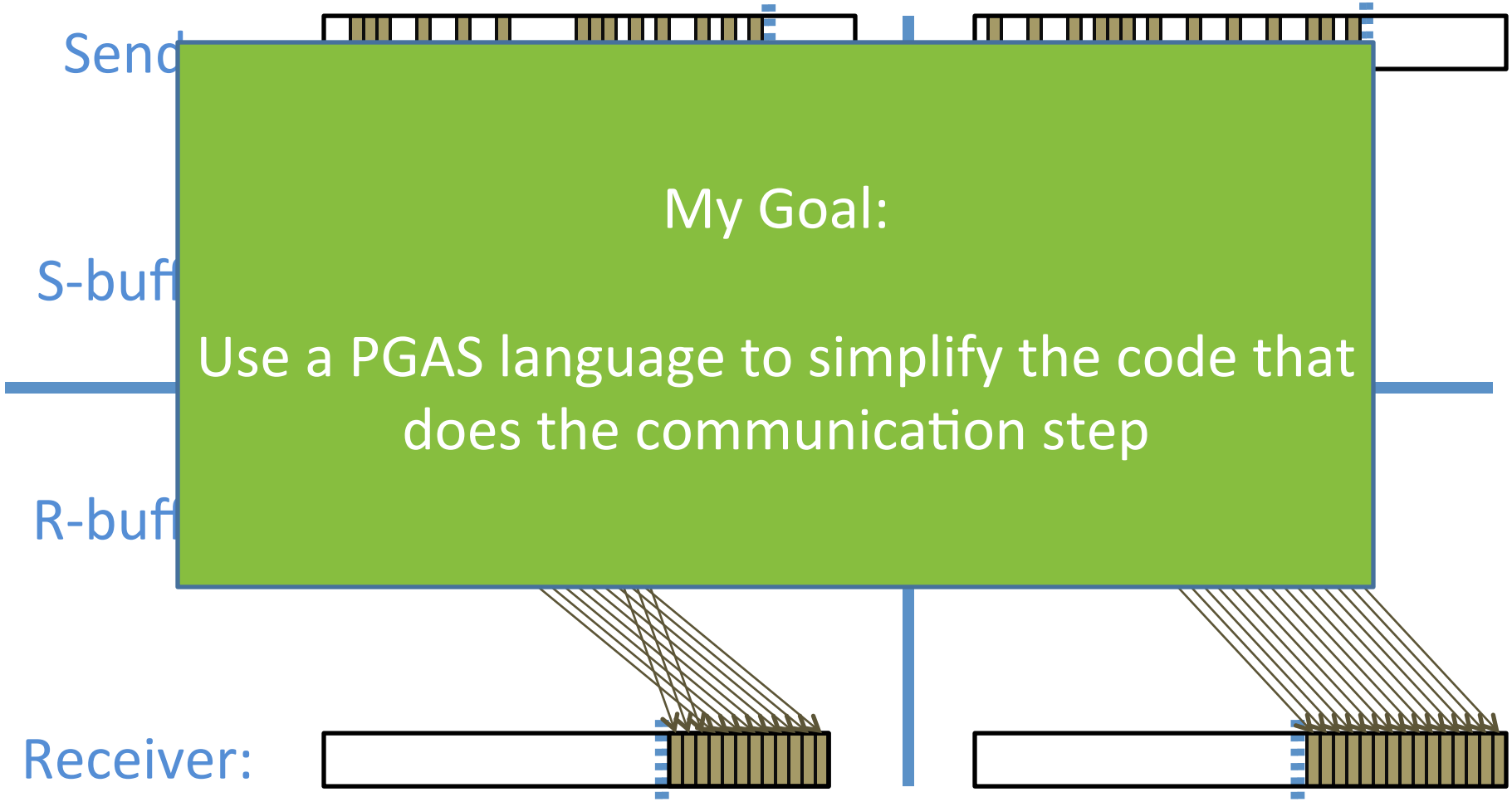
- 2D array
- Neighboring points stored in ghost cells

In 1d version:

- 1D array with indirect access
- Only ocean points are stored
- Neighboring points stored in halo

Initial CGPOP implementation

Communication to populate halo



Co-Array Fortran

Integrates into Cray Fortran compiler
SPMD execution (like MPI)
Co-arrays have extra dimension for processor
CAF also include some synchronization primitives

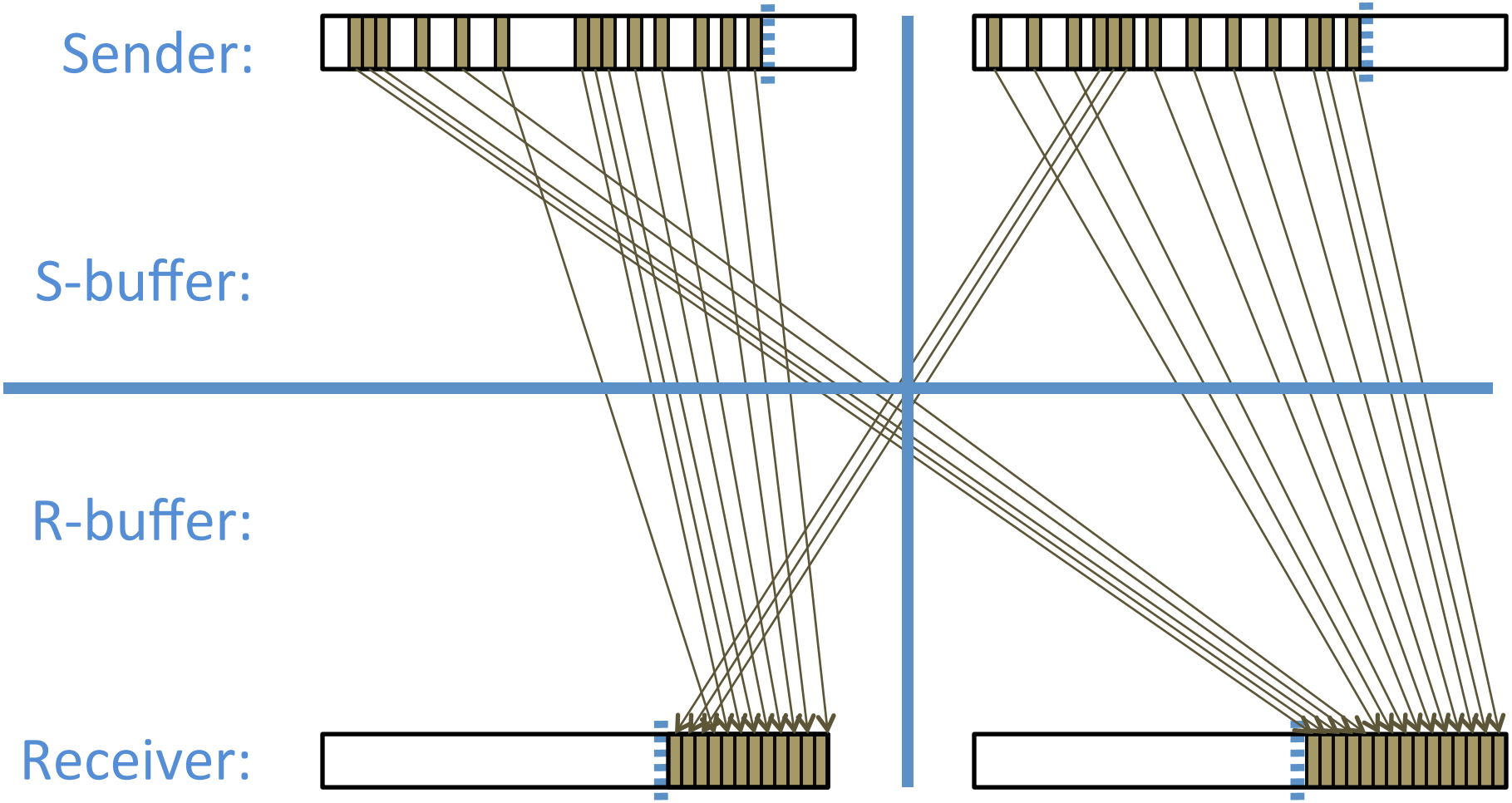
! Declaring a co-array:

```
integer :: A(10)[*]
```

! Set A(5) on right neighbor

```
integer :: rightNeigh = (this_image()  % num_images()) +1;
A(5)[rightNeigh] = 12345
```

One-sided communication (push)



This should be easy right?

Not so fast! Think about what metadata is needed to do communication:

Sending proc ID
Receiving proc ID
Address to pull data from on sender
Address to place data into on receiver

Metadata needed for communication

With 2-sided communication:

Sender	Receiver
Receiving proc ID	Sending proc ID
Address to pull from	Address to place into

With 1-sided pull communication:

Sender	Receiver
	Sending proc ID
	Address to pull from
	Address to place into

Lines of code

Original MPI version:

Init (calculate metadata):	266
Update (do communication):	33

1D pull version:

Init (calculate metadata):	33
Update (do communication):	14

Performance (on Lynx)

Original MPI version: 0.70 secs.

1D pull version: 36.08 secs.

Why poor performance?

Compiled to send one message per word

The compiler could help, but it doesn't

No automatic pre-fetching or message aggregation

What can be done:

Expression that access contiguous indices pass one message

A(lowIdx:highIdx)[recvProc]=
B(lowIdx:highIdx)[sendProc]

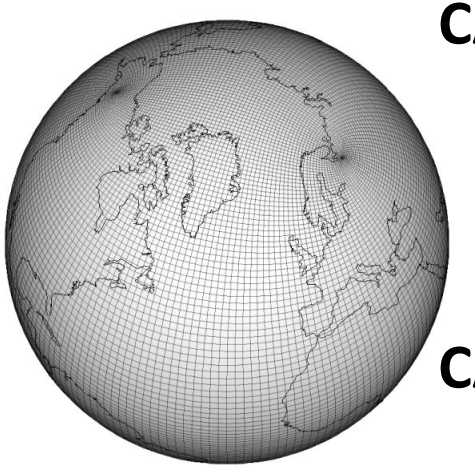
Looks like we're stuck using buffers for performance.

Performance w/ buffers (on Lynx)

Using buffers -- performance is better but not perfect:

Original MPI version:	0.70 secs.
Buffered push:	1.21 secs.
Buffered pull:	1.43 secs.

Conclusions



CAF's strong points:

- Decreased line count with non-buffered version
- Can compile with existing Fortran compiler
- Very easy learning curve

CAF's weak points:

- Immature compiler (found 2 compiler bugs)

Adoption: a chicken and egg scenario

- Immature compiler negatively impacts adoption
- Poor adoption doesn't help mature compiler
- Miniapps can help break this cycle

Lessons learned:

- Metadata placement can have a big impact on model refactorability
- Miniapp approach revealed issues

Conclusions

Open questions:

About my miniapp implementation:

- Refactoring vs. from scratch
- Maybe code-size / performance could be improved by changing data-layout?
- What about other models? (UPC, X10, Titanium, Chapel, etc.)

About evaluating with miniapps:

- Better metric than SLOC?
- What to include in a miniapp

About the future:

How mature will compilers be?

How much will future interconnects help?

[Bonus slides]

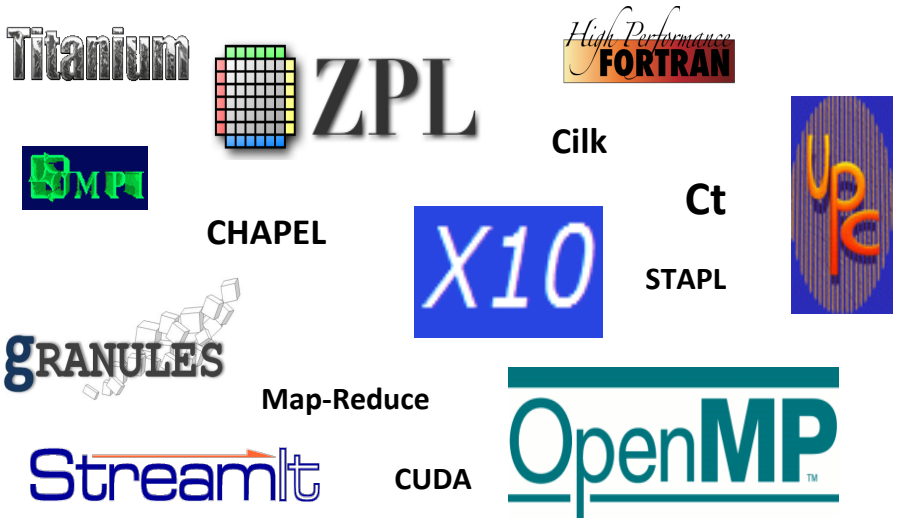
What is PGAS?

- **Parallel programming models:**
  technologies to express parallel computations.
  (languages and libraries)
- **PGAS**: partitioned global address space
  No explicit message passing, 1-sided communication

Why use it?

- Improved productivity:
  Fewer bugs
  Frees up scientists/programmers to do other things
  Quicker time-to-solution

Programming Models



There are many parallel programming models

MPI is predominantly used

MPI: "the assembly language of parallel programming"

Adoption of models effected by:

- Need to work with existing code
- Need to maintain high performance
- Lack of knowledge of applicability with real-world codes
- Lack of developers experienced with these models

Lines of code

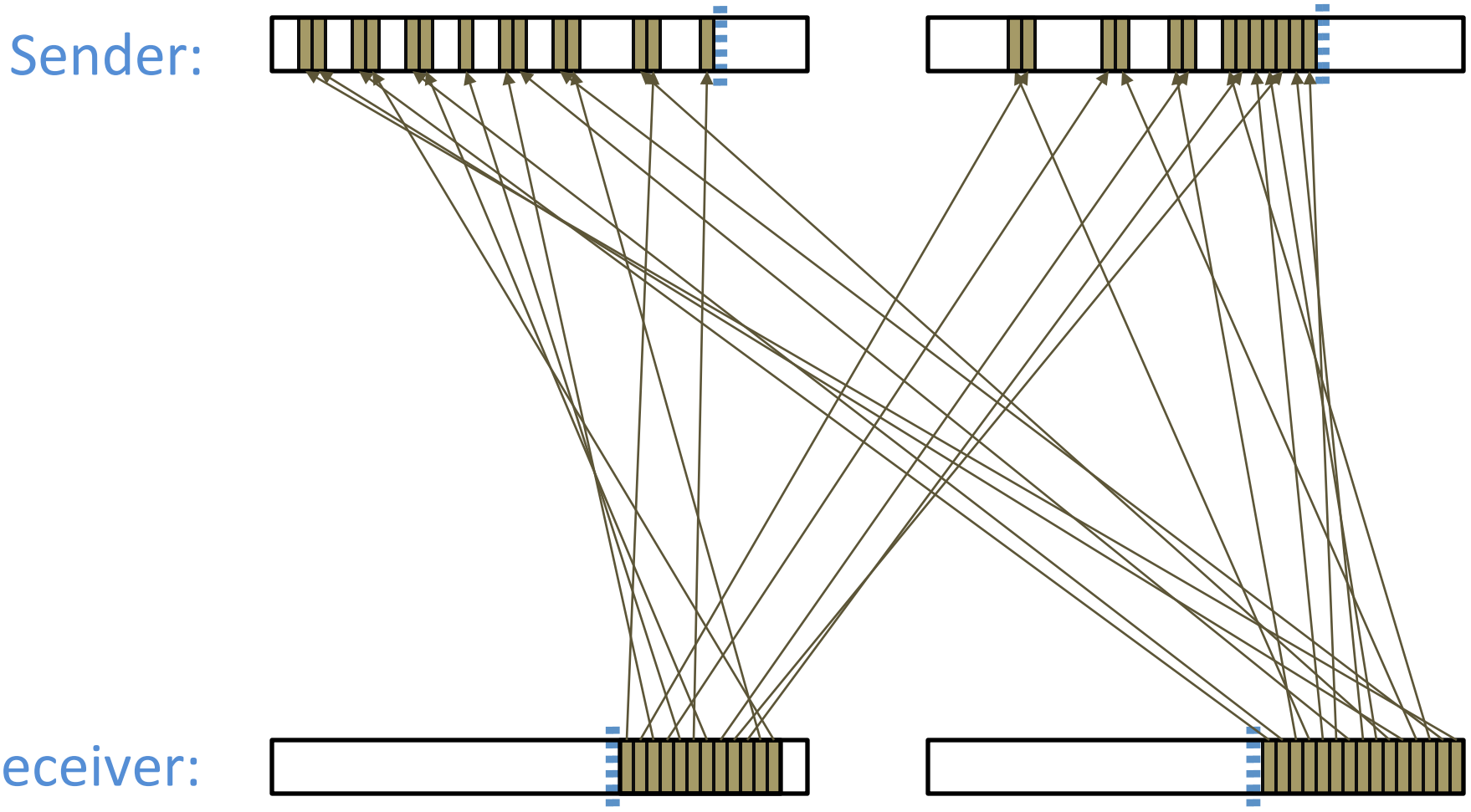
Original MPI version:

Init (calculate metadata):	266
Update (do communication):	33

2D push/pull buffered version:

Init (calculate metadata):	299
Update (do communication):	24

One-sided communication (pull)



Results

Some more subjective statements:

- Learning curve
- Thoughts on compiler

Results: learning curve

CAF very easy to learn
Need to be careful with off by one error
Not a lot of documentation available

Conclusion: CAF is a very learnable model given that it is a small extension to Fortran. Better debugger support would improve learnability.

Results: compiler difficulties

2 Compiler bugs:

- Passing parameters
- Optimizations with derived types

Conclusion: CAF compilers still not mature, however miniapp implementation reveals these issues

One parallel programming model

Can't study every model under the sun; must narrow to one:

Co-array Fortran

Benefits:

Commercial compiler (Cray)
Integrates into Fortran (don't have to rewrite entire code base)
One-sided communication (data is pushed or pulled)

Specifically looking at:

Communication step
CAF (and PGAS in general) is all about one-sided communication

Index spaces

Data resides in the following index spaces:

- Halo indices
- Receive buffer indices
- GDOF indices

Results: SLOC 1D

SLOC calculated via sloccount tool

Original MPI version:

Init: 266

Update: 33

	Non-buffered		Buffered	
	init	**update**	**init**	**update**
Push	--	--	299	24
Pull	33	14	297	24

Communication Metadata: original: receiver

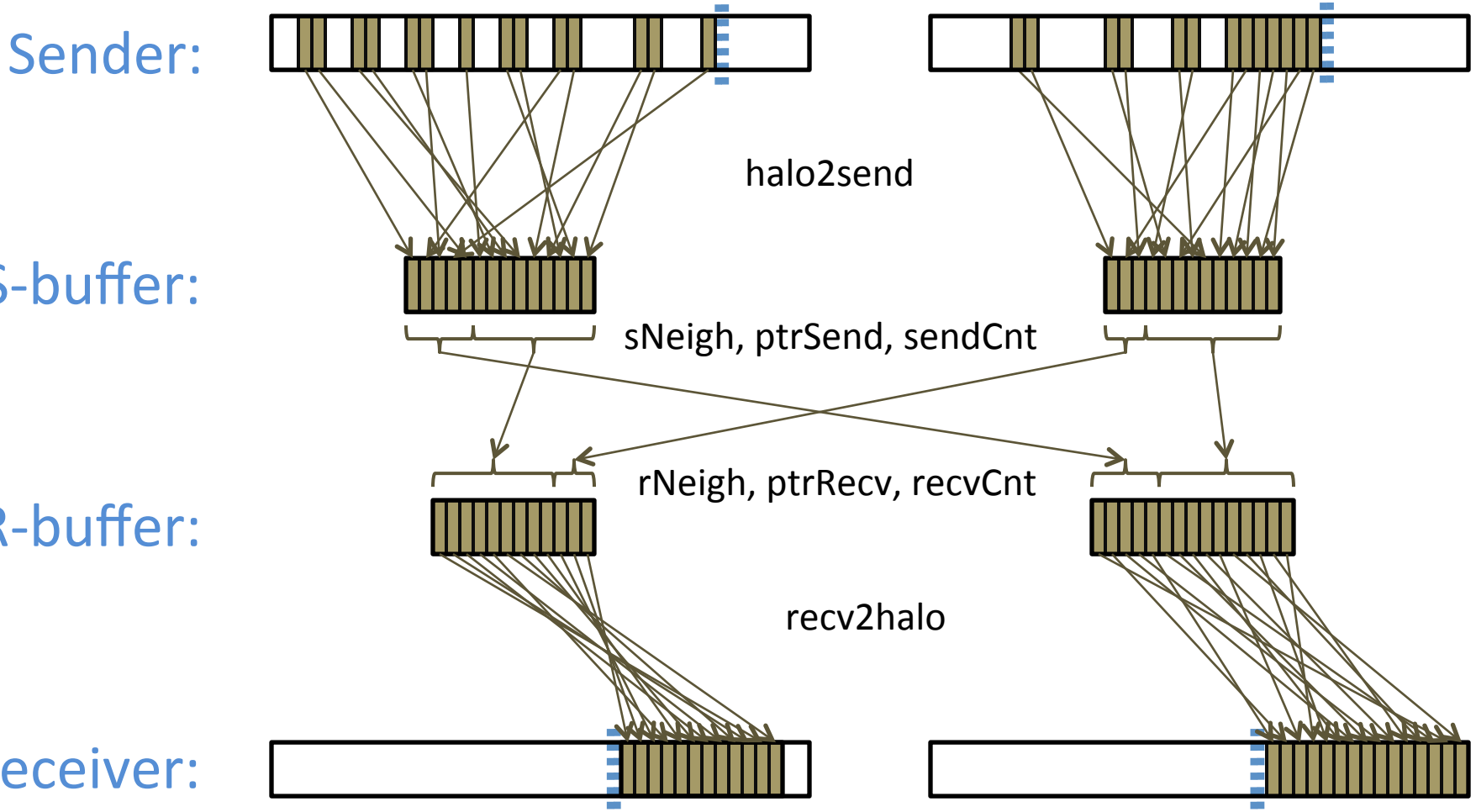
Variable	Description
nRecv	Number of neighbors to receive data from
rNeigh[]	List of neighbors to receive data from (by MPI rank)
ptrRecv[]	Array of offsets in receive buffer where data received from each neighbor in rNeigh should be placed.
recvCnt[]	Number of values that should be received from each neighbor in rNeigh
recv2halo[]	Maps indices in buffer to indices in halo

Communication Metadata: original: sender

Variable	Description
nSend	Number of neighbors to send data to
sNeigh[]	List of neighbors to send data to (by MPI rank)
ptrSend[]	Array of offsets in send buffer where data to be send to each neighbor in sNeigh should be placed.
sendCnt[]	Number of values that should be received from each neighbor in rNeigh
halo2send[]	Maps indices in buffer to indices in halo

Initial CGPOP implementation

Pattern of communication with metadata annotations



Initial CGPOP implementation

It helps to understand:

Distribution of computation:

What processor does what work

Distribution of data:

What structure does data take and where is it stored

Pattern of communication

Who sends what, who recievies it, and when does communication occur

Initial CGPOP implementation

Distribution of computation:

Owner-computes

Results: SLOC 2D

SLOC calculated via sloccount tool

Original MPI version:

Init: 574

Update: 151

	init	**update**
Push	160	94
Pull	160	94

Results: Performance 1D

All times are in seconds

Original MPI version: 0.70

	Non-buffered	**Buffered**
Push	--	1.21
Pull	36.08	1.43