Gradient of MPI-parallel codes

Jean Utke, Paul Hovland — *ANL*
Laurent Hascoët, Valérie Pascual — *INRIA Sophia-Antipolis*
Patrick Heimbach, Chris Hill — *MIT Boston*
Uwe Naumann, Michel Schanen — *RWTH Aachen*

ANL, INRIA, MIT, RWTH

June 14th, 2012

Outline

Adjoint Algorithms

Given an **algorithm** P that computes a **function** f,
the **"adjoint algorithm"** $\overline{\text{P}}$ computes the **gradient** of f.

Adjoint Algorithms

- compute the gradient at a cost
  that is **independent** from the number of inputs of f.
- compute the gradient backwards
  $\Rightarrow$ **reversal** of the control-flow and of the data-flow.
- can be built from P by an **Automatic Differentiation** tool.

Structure of Adjoint Programs

Algorithm P:

```
input a,b,c
...
...
u = g(a,c)
...
...
r = h(u,v)
output r
```

Structure of Adjoint Programs

Algorithm P:

input a,b,c

...

...

u = g(a,c)

...

...

r = h(u,v)

output r

Algorithm $\overline{\mathrm{P}}$:

input $\overline{\mathrm{r}}$

Structure of Adjoint Programs

Algorithm P:

```
input a,b,c
...
...
u = g(a,c)
...
...
r = h(u,v)
output r
```

Algorithm $\overline{\text{P}}$:

input $\overline{r}$

$\overline{u} \mathrel{+}= \frac{\partial h}{\partial u} * \overline{r}$

$\overline{v} \mathrel{+}= \frac{\partial h}{\partial v} * \overline{r}$

$\overline{r} = 0.0$

Structure of Adjoint Programs

Algorithm P:

input a,b,c

...

...

u = g(a,c)

...

...

r = h(u,v)

output r

Algorithm $\overline{\text{P}}$:

$\bar{u}$ = 0.0

$\bar{c}$ += $\frac{\partial g}{\partial c}$ * $\bar{u}$

$\bar{a}$ += $\frac{\partial g}{\partial a}$ * $\bar{u}$

...

...

$\bar{r}$ = 0.0

$\bar{v}$ += $\frac{\partial h}{\partial v}$ * $\bar{r}$

$\bar{u}$ += $\frac{\partial h}{\partial u}$ * $\bar{r}$

input $\bar{r}$

Structure of Adjoint Programs

Algorithm P:

input a,b,c

...

...

$u = g(a,c)$

...

...

$r = h(u,v)$

output r

Algorithm $\overline{P}$:

output $\bar{a}$, $\bar{b}$, $\bar{c}$

...

...

$\bar{u} = 0.0$

$\bar{c} \mathrel{+}= \frac{\partial g}{\partial c} * \bar{u}$

$\bar{a} \mathrel{+}= \frac{\partial g}{\partial a} * \bar{u}$

...

...

$\bar{r} = 0.0$

$\bar{v} \mathrel{+}= \frac{\partial h}{\partial v} * \bar{r}$

$\bar{u} \mathrel{+}= \frac{\partial h}{\partial u} * \bar{r}$

input $\bar{r}$

Adjoining Simple Instructions

Original:	Adjoint:
`x = a + 2*b`	$\bar{a} = \bar{a} + \bar{x}$; $\bar{b} = \bar{b} + 2*\bar{x}$; $\bar{x} = 0$
`x = 2*x`	$\bar{x} = 2*\bar{x}$
`y = sin(x)`	$\bar{x} = \bar{x} + \cos(x)*\bar{y}$; $\bar{y} = 0$
`b = a`	$\bar{a} = \bar{a} + \bar{b}$; $\bar{b} = 0$
`s = SUM(T(:))`	$\bar{T}(:) = \bar{T}(:) + \bar{s}$; $\bar{s} = 0$
`U(2:9) = U(2:9) + x`	$\bar{x} = \bar{x} + \mathrm{SUM}(\bar{U}(2:9))$
`where(T>3) T = T - a`	$\bar{a} = \bar{a} - \mathrm{SUM}(\bar{T}, T>3)$

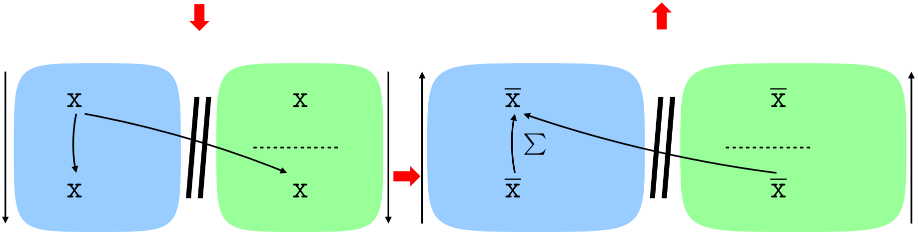
All these can be "proved" formally ... but a convenient justification is **backwards propagation of the influence** on the result.

Outline

Adjoining one Global Communication

```
bcast(x, ,P, )
```

```
reduce(x̄,t̄, , ,SUM,P, )
x̄ = 0.0
on P: x̄ = x̄ + t̄
```



Adjoining Global Communications

Original:	**Adjoint:**
bcast(x,,P,)	reduce($\bar{x}$,$\bar{t}$,,,SUM,P,) $\bar{x}$=0.0 ; on P:$\bar{x}$=$\bar{x}$+$\bar{t}$
reduce(x,y,,,SUM,P,)	on P:$\bar{t}$=$\bar{y}$; on P:$\bar{y}$=0.0 bcast($\bar{t}$,,P,) ; $\bar{x}$=$\bar{x}$+$\bar{t}$
allreduce(x,y,,,SUM,)	allreduce($\bar{y}$,$\bar{t}$,,,SUM,) $\bar{y}$=0.0 ; $\bar{x}$=$\bar{x}$+$\bar{t}$
gather(x,,y,,P,)	scatter($\bar{y}$,,$\bar{t}$,P,) on P:$\bar{y}$=0.0 ; $\bar{x}$=$\bar{x}$+$\bar{t}$
scatter(x,,y,,P,)	gather($\bar{y}$,,$\bar{t}$(:),,P,) $\bar{y}$=0.0 ; on P:$\bar{x}$=$\bar{x}$+$\bar{t}$(:)

Outline

Adjoining Point-to-point Communication

- Blocking ⇔ {Nonblocking ; `wait` }
- Analogy `send(a)/receive(b)` with `b = a`

`isend(a,r)` / `...` / `wait(r)` ∥ `irecv(b,r)` / `...` / `wait(r)`

`wait(r);` $\bar{a}$`+=t` / `...` / `irecv(t, r)` ∥ `wait(r);` $\bar{b}$`=0` / `...` / `isend(`$\bar{b}$`, r)`

`b = a` ➡ $\bar{a}$ `+=` $\bar{b}$`;` $\bar{b}$ `= 0`

Gives us the adjoints of MPI `isend`, `irecv`, and `wait`.

Rules for adjoining MPI calls

Adjoining rules depend on the context

- between nonblocking calls and their `wait`
- between the two communicating processes.

in P		in $\overline{\text{P}}$
call	paired with	call
`isend(a,r)`	`wait(r)`	`wait(r);ā+=t`
`wait(r)`	`isend(a,r)`	`irecv(t,r)`
`irecv(b,r)`	`wait(r)`	`wait(r);b̄=0`
`wait(r)`	`irecv(b,r)`	`isend(b̄,r)`
`bsend(a)`	`recv(b)`	`recv(t);ā+=t`
`recv(b)`	`bsend(a)`	`bsend(b̄);b̄=0`
`ssend(a)`	`recv(b)`	`recv(t);ā+=t`
`recv(b)`	`ssend(a)`	`ssend(b̄);b̄=0`

Adjoinable MPI requires more information

Adjoining a `wait` requires some context:

- Each `wait` must know what it waits for (`isend` or `irecv`)
- Each `wait` must know and **see** the travelling variable

Source analysis could find matching `wait` of a non-blocking call, but

- not always, and
- making the travelling variable visible is harder.

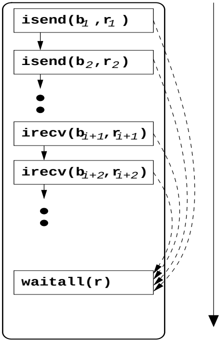
Instead, use an adjoinable MPI such as:

- `waitrecv(b,r)`, whose adjoint is `isend(`$\bar{\mathtt{b}}$`, r)`
- `waitsend(a,r)`, whose adjoint is `irecv(t, r)`

Outline

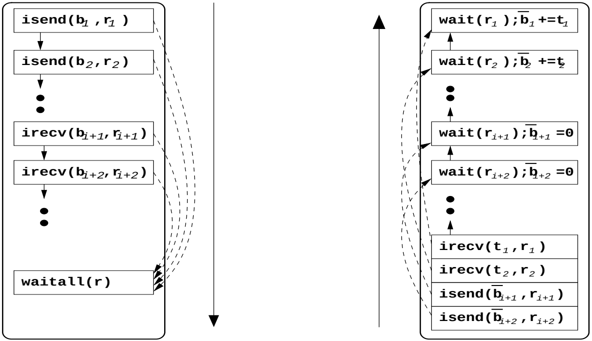
Open question: grouped wait's

"waitall" groups "wait" operations. Improves efficiency.



Open question: grouped wait's

"waitall" groups "wait" operations. Improves efficiency.



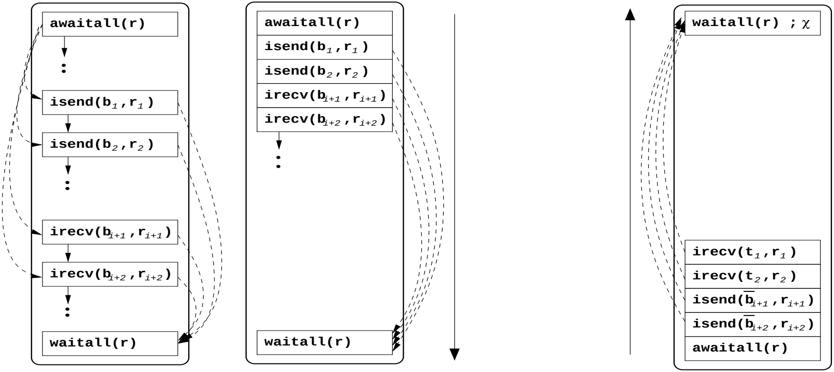
Adjoining re-introduces separate wait's !

⇒ can we get a waitall() back?

The “anti waitall”

Sometimes, we may introduce a `awaitall()`

- nonoperational
- placed by the end-user.

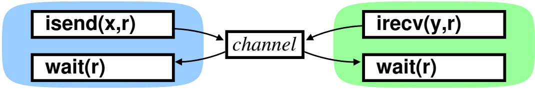


⇒ Allows the adjoint to use a `waitall()` again.

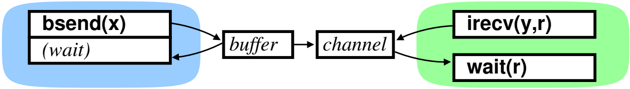
Outline

Data dependence graph for Point-to-Point

- Introduce "*channel*" pseudo variables.
- Nonblocking `isend`/`irecv`

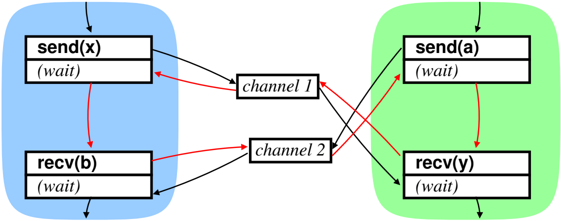


- `send`⇔{`isend; wait`}   `recv`⇔{`irecv; wait`}
- Buffered `bsend` uses an intermediate copy buffer
  ⇒ immediate return.



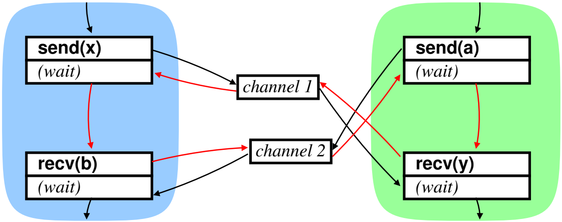
Deadlocks ; Blocking vs Nonblocking

Deadlocks are cycles in the data dependence graph:

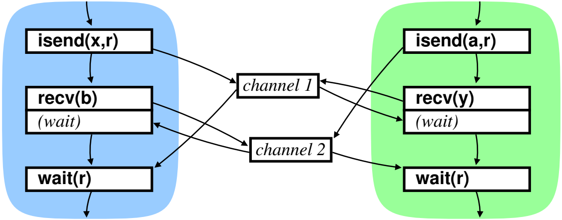


Deadlocks ; Blocking vs Nonblocking

Deadlocks are cycles in the data dependence graph:



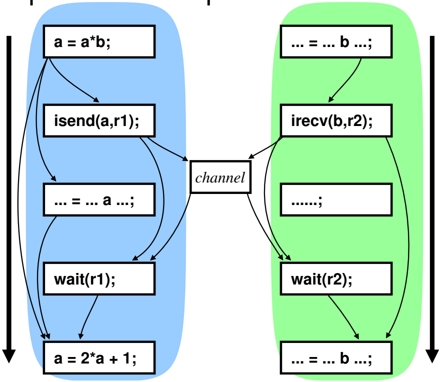
Splitting the `wait` from the `isend`/`irecv` can solve the problem:



Otherwise, use `bsend`'s.

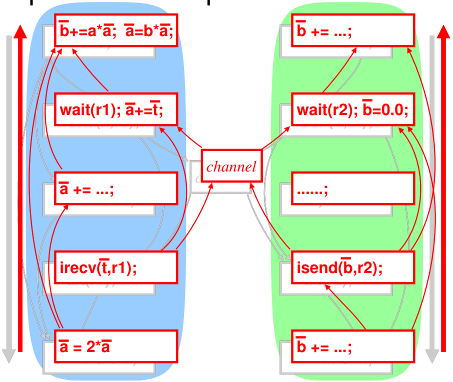
Data Dependence Graph of the Adjoint Algorithm

Consider the Data Dependence Graph of an MPI communication.



Data Dependence Graph of the Adjoint Algorithm

Consider the Data Dependence Graph of an MPI communication.



The Data Dependence Graph of the adjoint communication seems to just reverse the arrows.

$\Rightarrow$ Adjoining does not introduce deadlocks.

Outline

Impact on Data-Flow analyses

AD, like other program transformations, needs preliminary data-flow analyses

Message-passing modifies data-flow $\Rightarrow$ we must adapt analyses.

- If Interprocedural Control-Flow Graph $\longrightarrow$ introduce special flow arrows that only convey messages.
- If Call Graph of Flow Graphs $\longrightarrow$ introduce special "channel" variables and organize additional fixed-point iterations.

In any case, increases complexity/cost of analyses.

Outline

References

- Utke, Hascoët, Heimbach, Hill, Hovland, Naumann **"Towards Adjoinable MPI"** *IEEE IPDPS, 2009*
- Schanen, Naumann, Hascoët, Utke **"Interpretative Adjoints for Numerical Simulation Codes using MPI"** *Procedia Computer Science, 2010*
- Schanen, Naumann, Hascoët, Utke **"Interpretative Adjoints for Numerical Simulation Codes using MPI"** *ICCS 2010*
- Schanen, Förster, Naumann **"Second-Order Algorithmic Differentiation by Source Transformation of MPI Code"** *EuroMPI 2010*
- Naumann, Hascoët, Hill, Hovland, Riehme, Utke **"A Framework for Proving Correctness of Adjoint Message-Passing Programs"** *PVM/MPI Users' Group Meeting, 2008*
- Strout, Kreaseck, Hovland **"Data-Flow analysis for MPI programs"** *ICPP, 2006*
- Kreaseck, Strout, Hovland **"Depth Analysis of MPI programs"** *AMP workshop, PLDI 2010*
- Pascual, Hascoët **"Native handling of Message-Passing communication in Data-Flow analysis"** *AD2012*

Outline

Conclusion: an adjoinable MPI ?

- Unlike previous black-box approaches,
  we consider the level of the **individual MPI calls**.
- Our approach requires the **correspondence** between MPI calls
- Static data-flow analysis will not find it in general:
  $\Rightarrow$ **User input** is necessary:
  - could be pragmas
  - could be an “adjoinable MPI” library

Outlook

- Ongoing **application to the MITgcm**: validates the adjoining rules of MPI calls. Requires the `awaitall`.
- Look for a **general proof** of correctness, maybe based on PGAS or other abstractions e.g. Data Dependence graphs.
- Develop an **adjoinable MPI library**, to help/induce the user write an adjoinable code..

Outlook

- Ongoing **application to the MITgcm**: validates the adjoining rules of MPI calls. Requires the `awaitall`.
- Look for a **general proof** of correctness, maybe based on PGAS or other abstractions e.g. Data Dependence graphs.
- Develop an **adjoinable MPI library**, to help/induce the user write an adjoinable code..

Thank you for your attention !