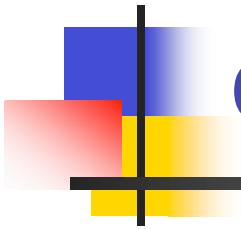
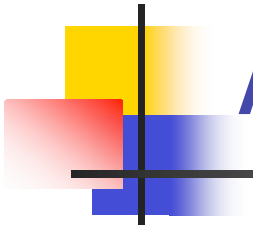


Retro dataflow: Accomplishments of the Sisal Language Project

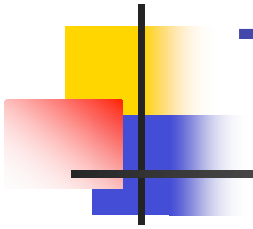
Pat Miller[1]
patmiller@users.sourceforge.net
John Feo[2]
Tom Deboni[3]

Current Affiliations: 1 D.E. Shaw Research 2 Microsoft Inc. 3 LBL/NERSC



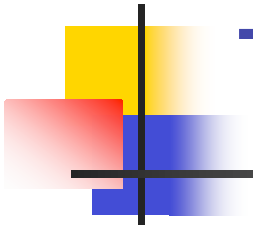
Abstract

The Sisal Language Project was a joint effort by Lawrence Livermore National Laboratory, Colorado State University, and University of Manchester to develop a high performance functional programming language for conventional parallel computer systems. The main project began in the early '80s and concluded in 1996. The project was successful in that many Sisal programs ran as fast as their Fortran or C equivalent on parallel systems of the time such as the SGI Challenge, multi-processor VAXs, and Cray vector computers. In this talk, we review the language's syntax and semantics, optimizing compiler, and high-performance runtime system. We describe two critical optimizations: memory pre-allocation and update-in-place. We conclude with comparative performance numbers, and give the availability of reports and source code.



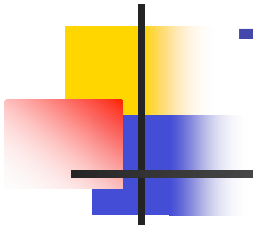
The birth of Sisal

- Sisal is a descendant of MIT's VAL [Akerman/Dennis79] and came to Livermore Labs via Jim McGraw
- Was a joint project with LLNL, Colorado State University, University of Manchester, and DEC
- We wanted to write implicitly parallel code in a high level language since we believed that you couldn't trust programmers to create correct parallel and vector code.



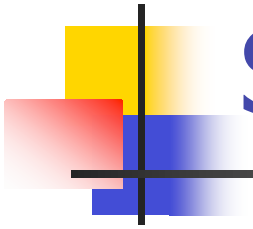
The birth (continued)

- Needed for novel parallel machines
  - Denelcor HEP, Manchester Dataflow Machine, DEC circus/dumbo, CRAY 2, Sequent Balance, Alliant FX, Encore Multimax, Warp Systolic Array, etc…
  - These machines were hard to program
- The language supported the streaming/vector style + coarse task model of the Crays **and** the fine grained active memory of experimental dataflow machines



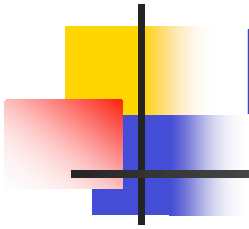
The Death of Sisal

- The project had several near death experiences and suffered a steady funding decline and the loss of industrial partners
- Funding was cut off in April 1996 (mid-year)
- Coincided with the new emergence of distributed memory "killer micros"



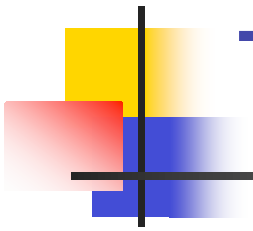
Syntax and Semantics

- Single Assignment
- Side effect free
- Referencial Transparency
- Applicative rather than functional
- Groovy Pascal-like syntax
- Strongly (though implicitly) typed
- Implicit parallelism



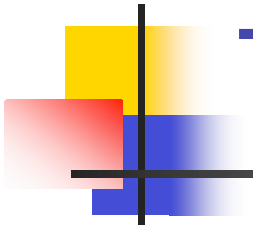
Hello World

```
define main

type string = array[character];

function main(returns string)
  "hello world!"
end function
```



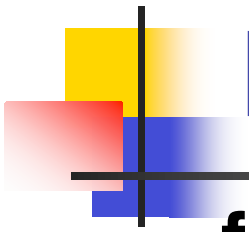
The LET functional form

```
function f(a,b,c,d : real returns real,real,real)
  let
    mean := (a+b+c+d)/4.0;
    % exp(a,b) means a**b,
    variance := exp(a-main,2) + exp(b-main,2) +
                        exp(c-main,2) + exp(d-main,2);
  in
      mean,variance,sqrt(variance)
  end let
end function
```



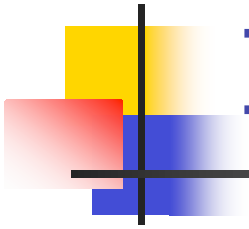
The IF functional form

```
if x < 10 then
    x+5,f(y)
elseif x > 20 then
    x,g(y)
else
    x,0
end if
```



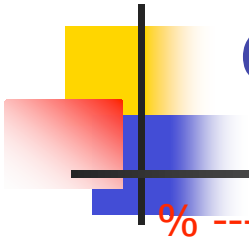
Loops with carried state

```
for initial
  i := 0;
  a := initialize(x,y);
  epsilon := 1e-6;
while err(a) < epsilon repeat
  a := evolve(old a);
returns
  value of a
  value of sum g(a)
  array of f(i,a)
end for
```



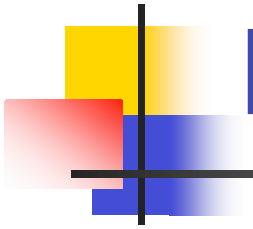
Implicitly parallel loops

```
for a in x dot b in y
      term := a*b;
   returns value of sum term;
end for

for   x_row in x                          % for all rows of
x
   cross y_col in y_transposed  % all columns of
y
   returns array of dot_product(x_row, y_col)
end for
```



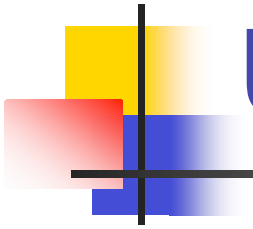
99 bottles of beer

```
% ---------------------------------------------------------
% Produce one stanza of the 99 bottles of beer song. Some care
% is taken to keep it grammatical
% ---------------------------------------------------------
function BottlesOfBeer(i : integer returns array[string])
  let
    s,bottles,preface,n,nextbottles :=
    if i = 1 then
      "1"," bottle","If that bottle","No more"," bottles"
    elseif i = 2 then
      itoa(2)," bottles","If one of those bottles",itoa(1)," bottle"
    else
      itoa(i)," bottles","If one of those bottles",itoa(i-1)," bottles"
    end if;
  in
    array[1: s || bottles || " of beer on the wall", s || bottles || " of beer!",
        preface || " should happen to fall... ", n || nextbottles || " of beer on the wall!", "" ]
  end let
end function
```



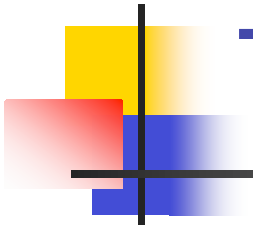
Missing bits

- 1st class functions
- No strings!
- No complex type
- No power operator (we use "exp")
- Reductions
  - Minat, maxat
  - Histograms
  - User defined
- Regular n-D arrays



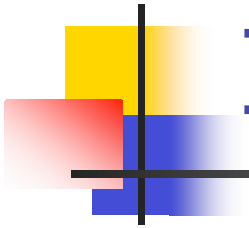
Ugly bits

- FIBRE I/O
- Extension/Embedding API
- Error value algebra and error handling
- For-initial form is awkward
- Let-in form can be awkward
- Array bounds algebra
- Stream implementations
- No mixed mode arithmetic
- Ugly to use math intrinsics
- Keyword heavy
- Case insensitivity



The Intermediate Forms

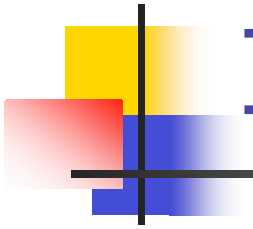
- IF1 (pure dataflow)
- IF2 (dataflow with memory and artificially enforced ordering)
- Structured type description
- Directed graphs with operations on nodes and values on edges
- Simple nodes like IFPlus, IFCall, etc..
- Compound nodes like IFSelect, IFForall, IFLoopA, IFLoopB



IF1 Types

```
T 1 1 0    %na=Boolean
T 2 1 1    %na=Character
T 3 1 2    %na=Double
T 4 1 3    %na=Integer
T 5 1 4    %na=Null
T 6 1 5    %na=Real
T 7 1 6    %na=WildBasic
T 8 10
T 9 0    4              % array[Integer]
T 10 8   9        0
T 11 8   4        10
T 12 8   4        11     % integer,integer,array[integer]
T 13 8   4        0
T 14 8   4        13     % integer,integer
T 15 3   12       14     % integer,integer,array[integer] → integer, integer
```

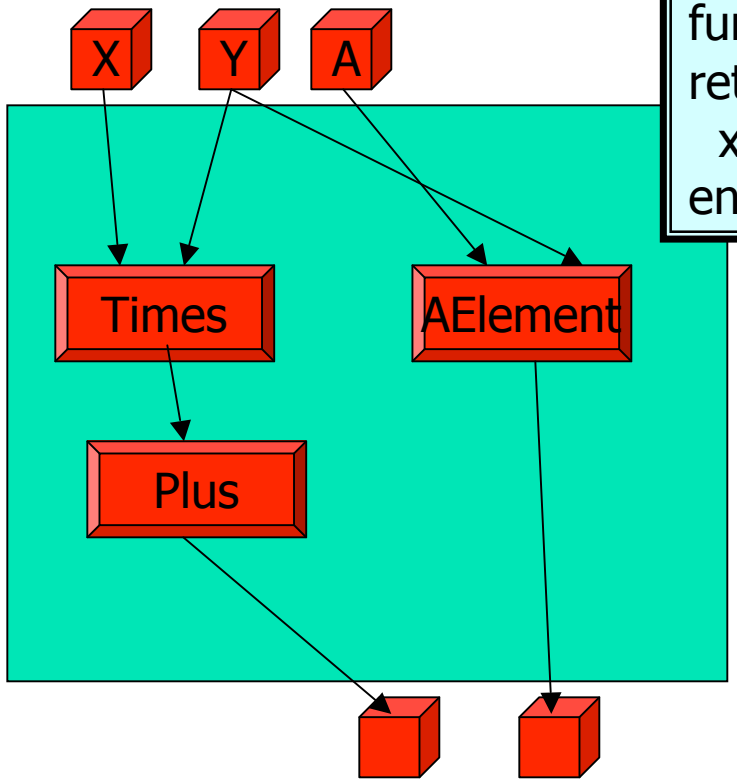
```
function main(x,y: integer; A : array[integer]
returns integer,integer)
  x+y*3,A[y]
end function
```



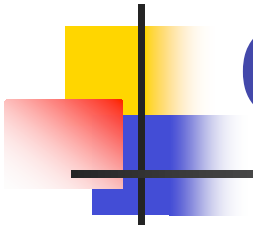
IF1 Function body

```
X       15      "main"  %sl=3   %sf=foo.sis
N 1     152     %sl=4   %sf=foo.sis
E       0 2     1 1     4       %na=y           %mk=V
%sl=4
L               1 2     4 "3"   %mk=V
N 2     105     %sl=5   %sf=foo.sis
E       0 3     2 1     9       %na=a  %mk=V        %sl=4
E       0 2     2 2     4       %na=y  %mk=V        %sl=4
N 3     141     %sl=4   %sf=foo.sis
E       0 1     3 1     4       %na=x %mk=V         %sl=4
E       1 1     3 2     4       %mk=V

E       3 1     0 1     4       %mk=V
E       2 1     0 2     4       %mk=V
```

As a dataflow graph…

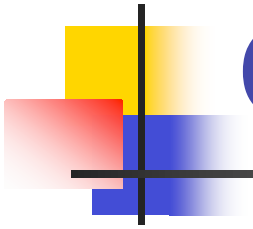



```
function main(x,y: integer; A : array[integer]
returns integer,integer)
  x+y*3,A[y]
end function
```



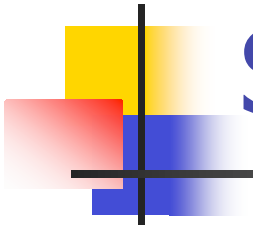
Optimizing compiler

- Developed at Colorado State by David Cann (thesis) and extended by David and Pat Miller at Livermore
- Consists of a Frontend to convert to the intermediate form and then a six-pass backend to generate executables



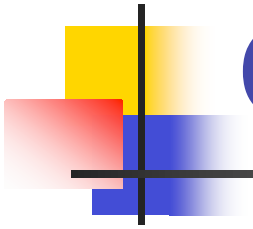
OSC

- if1ld – merge separately compiled .if1 files
- if1opt – architecture independent optimizations
- if2mem – add if2 buffers
- if2up – update in place analysis
- if2part – parallel (threaded) partitioning and vectorization
- cgen – Final code generation



Sisal Runtime

- Wild and woolly days before standards!
- Had to write our own thread-safe malloc, thread pool, and synchronization library
  - Deallocation is tied to idle workers
- Needed a startup/shutdown to talk with the FLI (Foreign language interface)

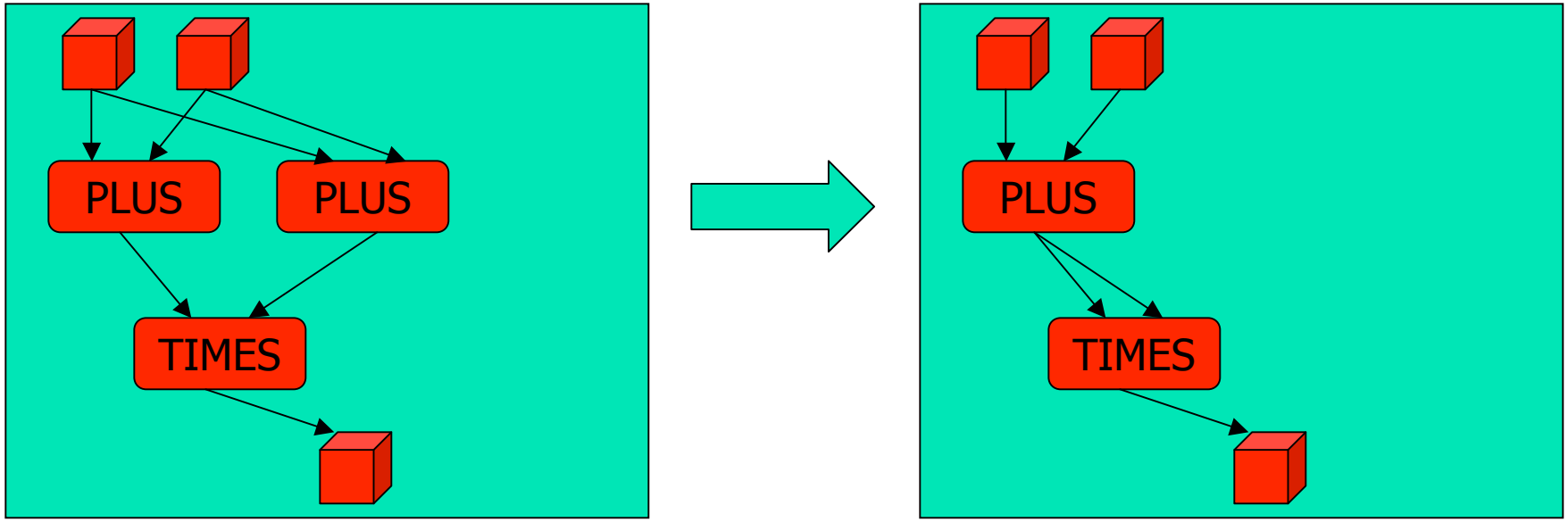


Optimizations

- Many optimizations are easy in pure functional code
- The error semantics allow free reordering
- Values are not tied to memory by the language, only late in the implementation
  - Simplifies vectorization, array-of-structs vs struct of arrays

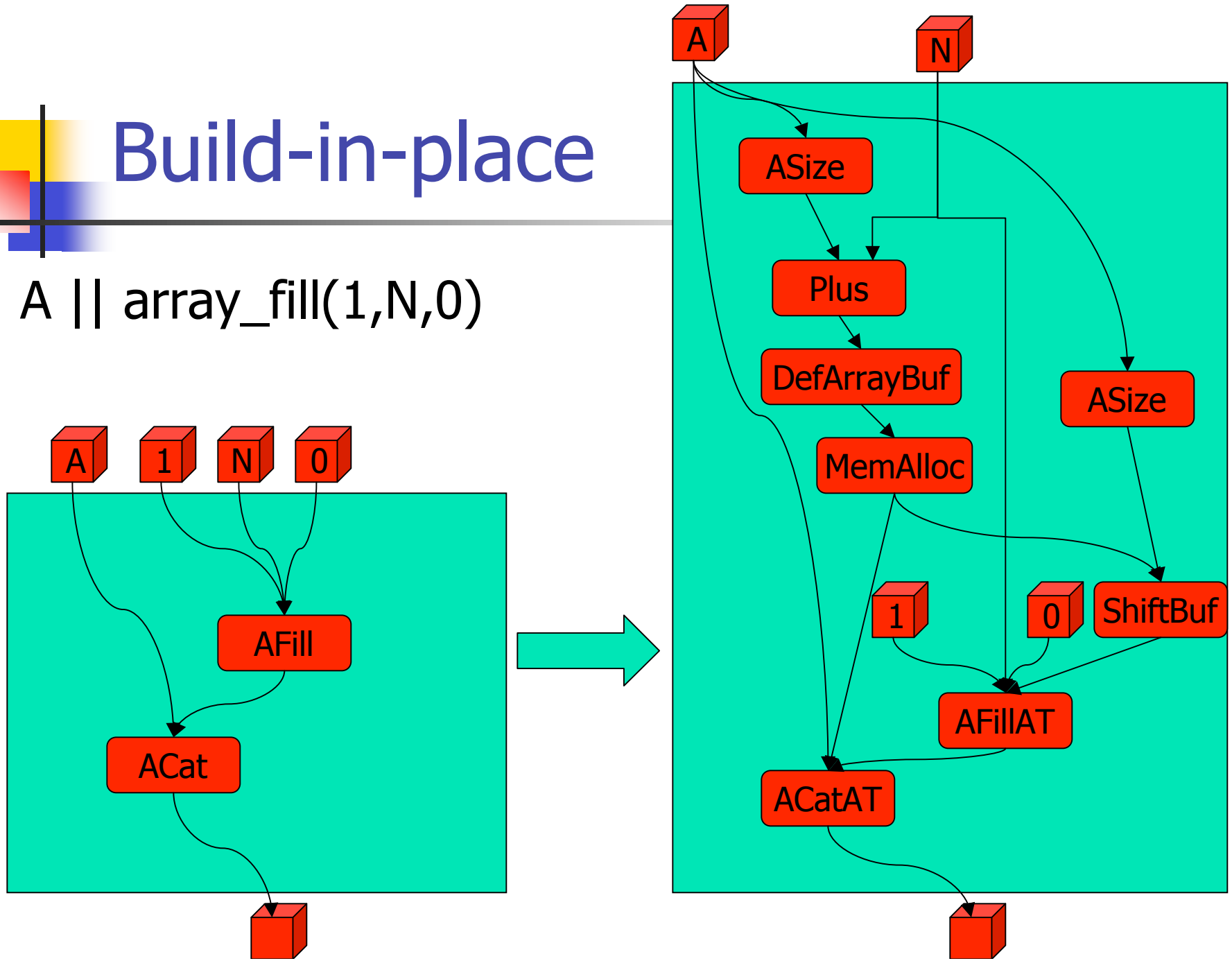
CSE

- If two simple nodes have the same opcode and input edges from the same sources, they are identical and can be merged. Period!



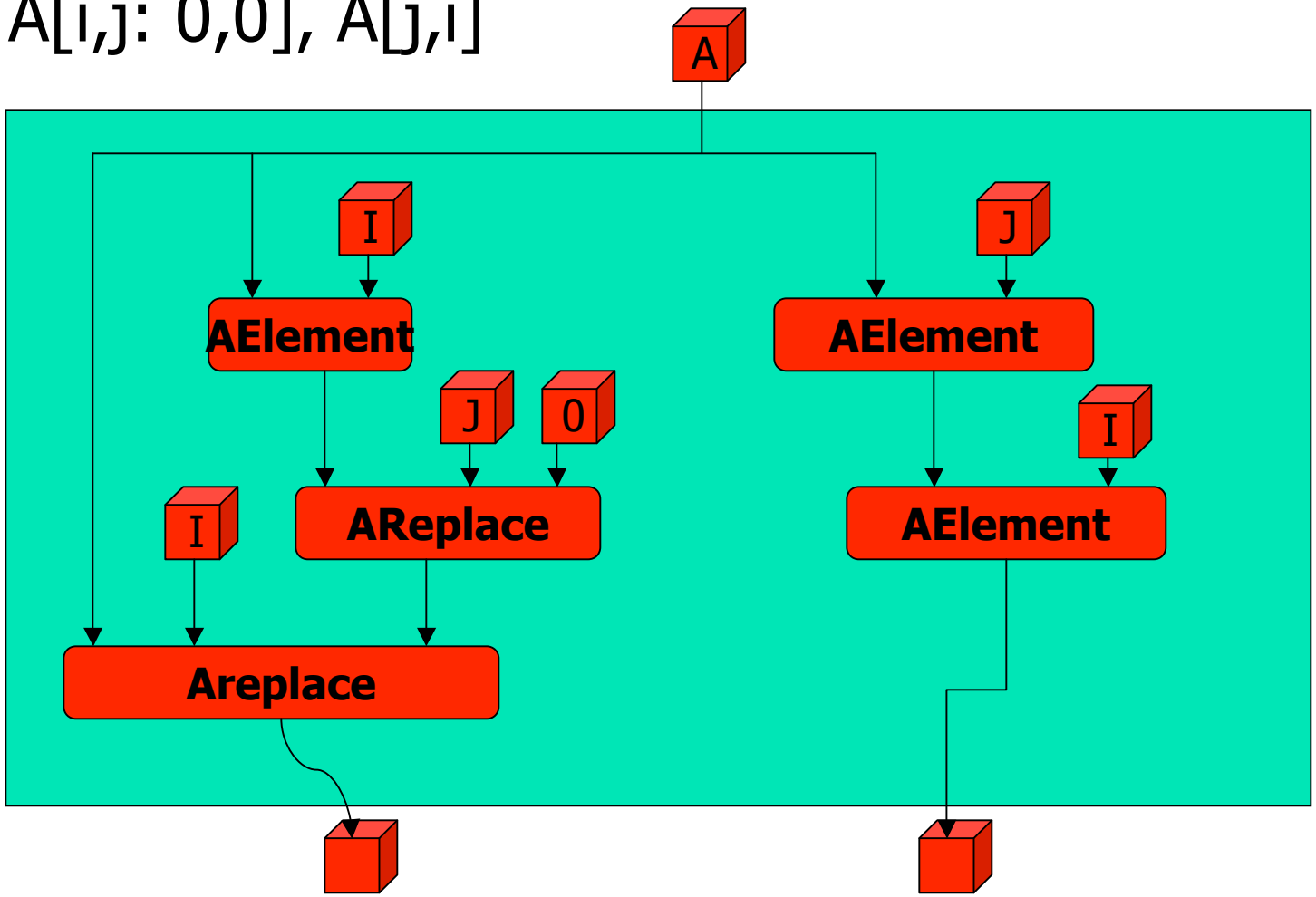
Build-in-place

A || array_fill(1,N,0)

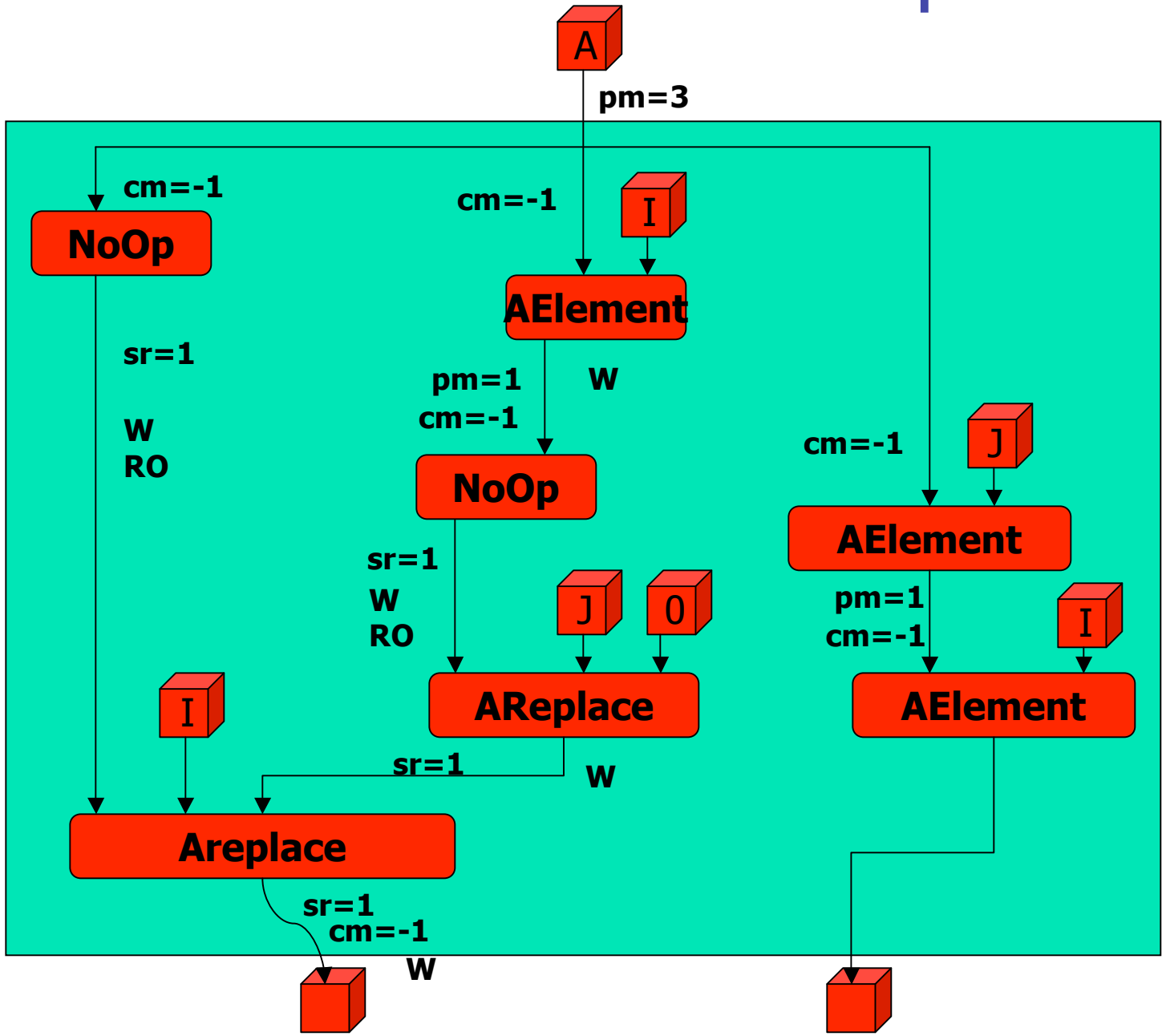


Update-in-place

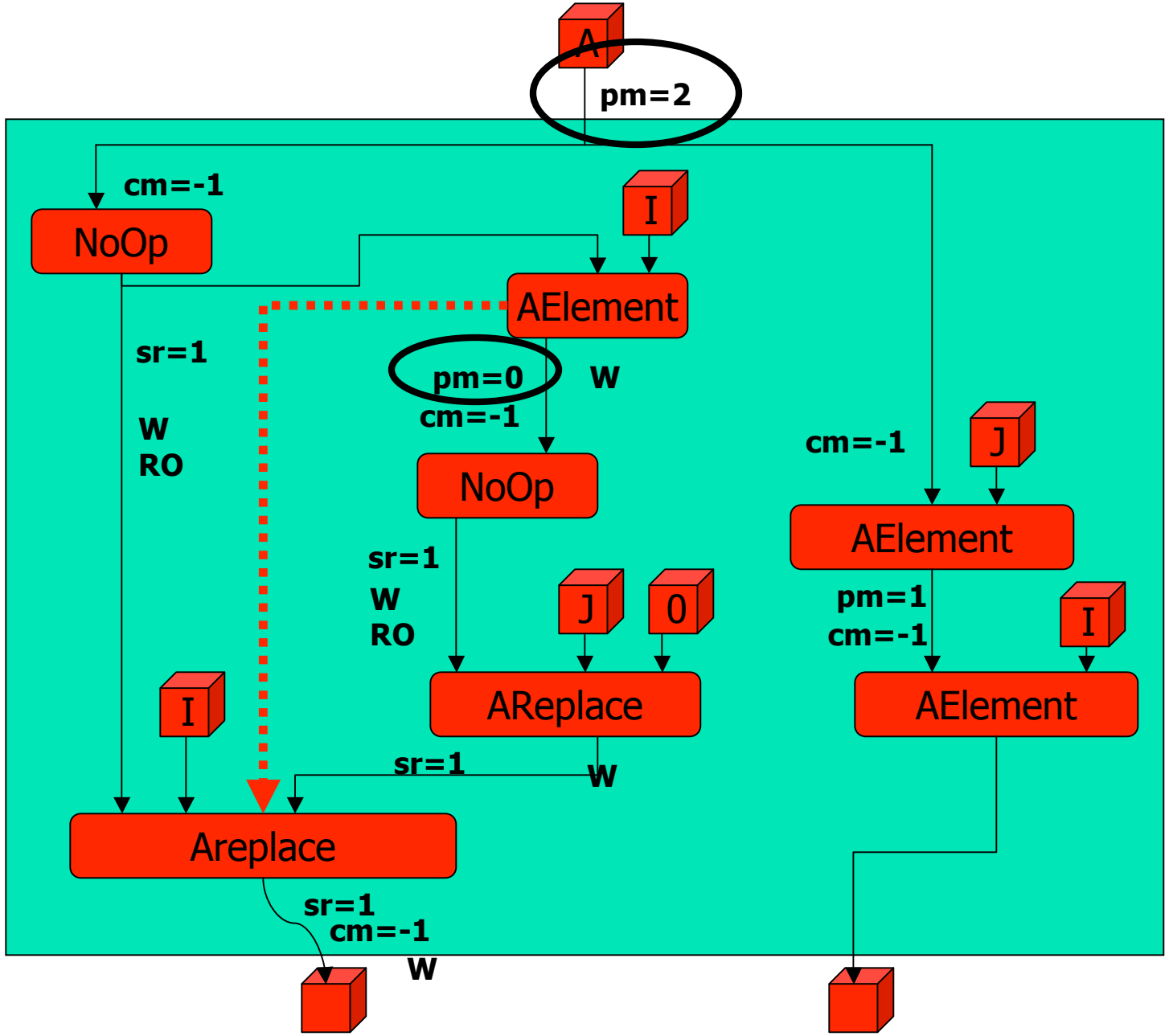
A[i,j: 0,0], A[j,i]



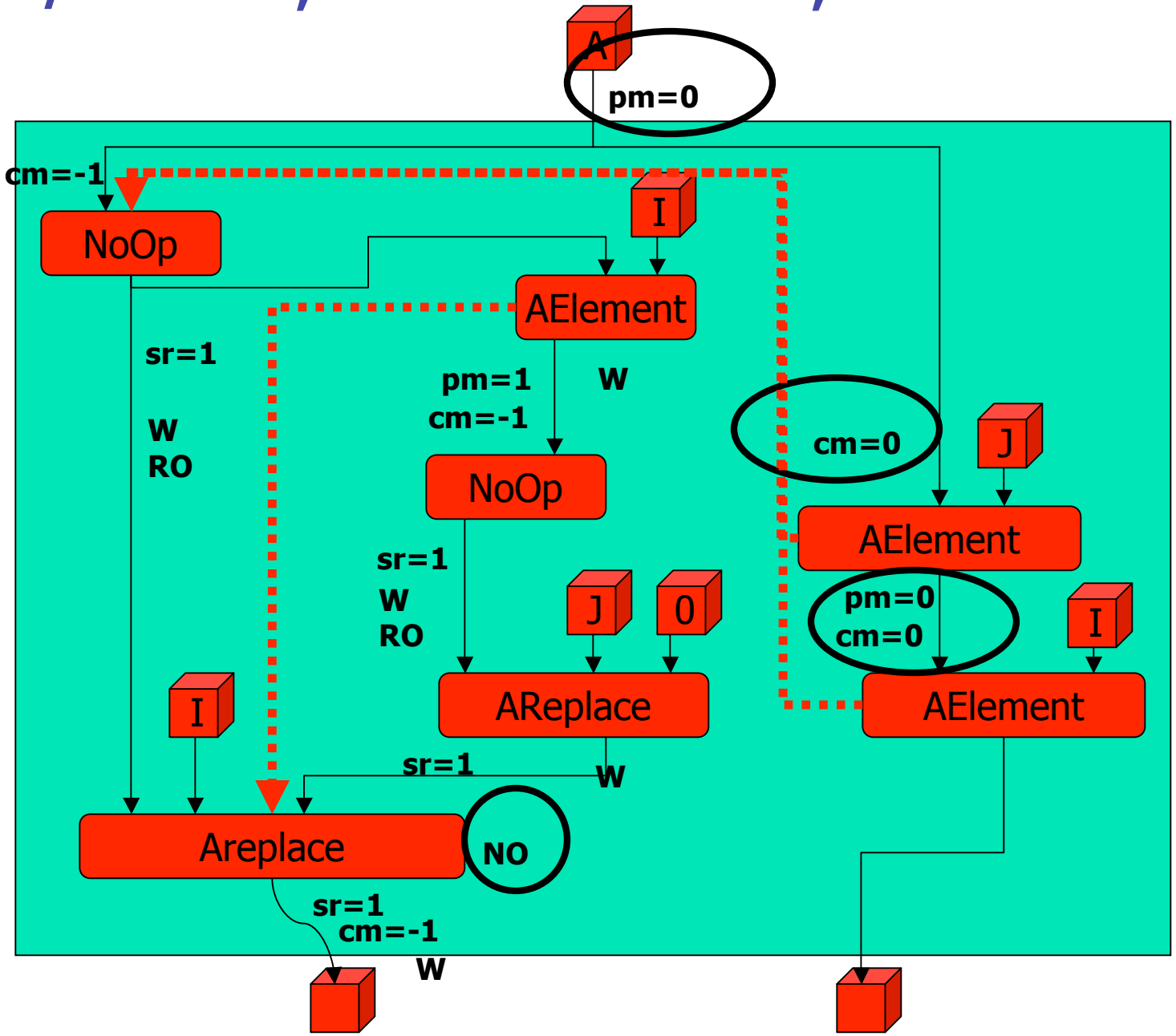
Reference counts and copies

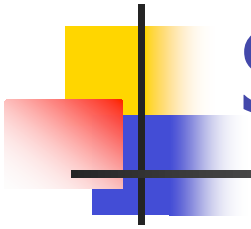


Reference inheritance



R/W set,node reorder, refcnt elim



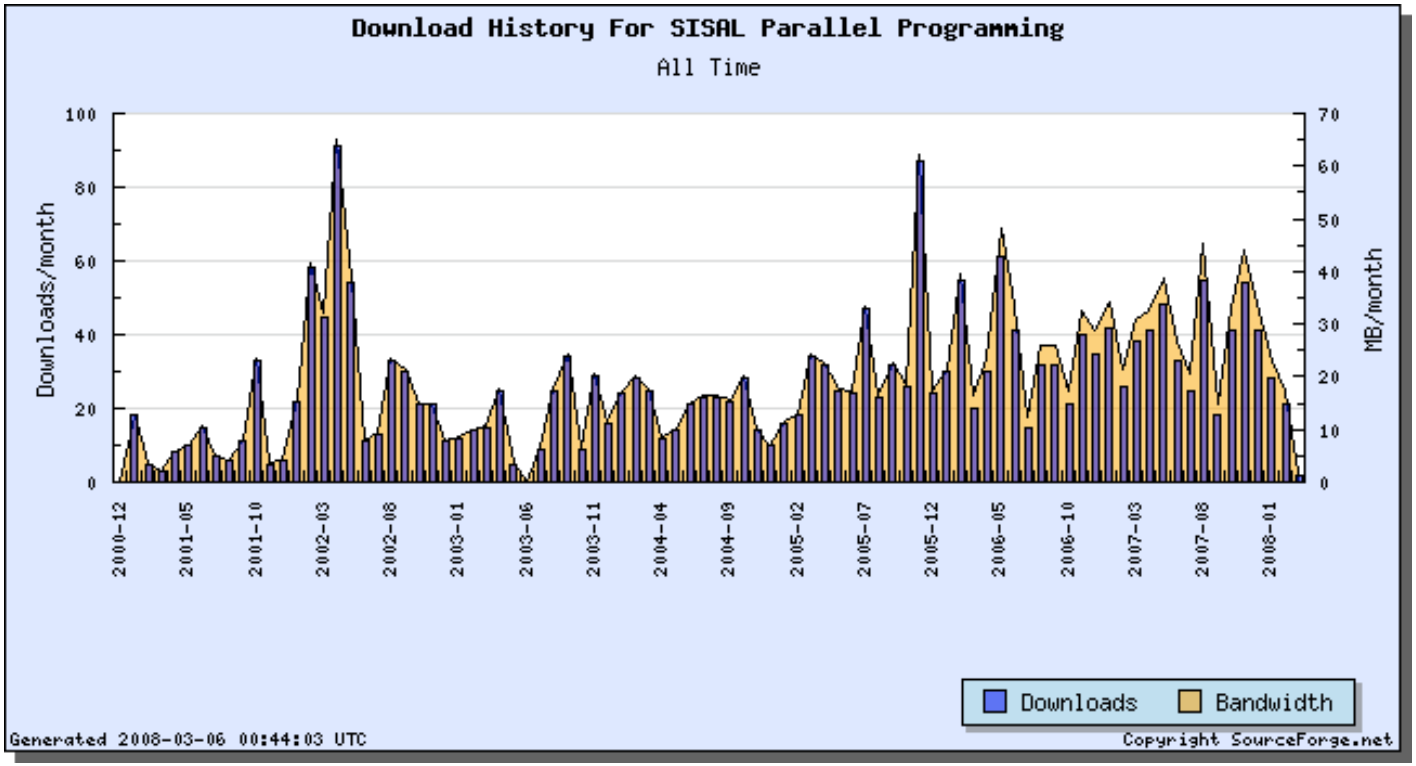


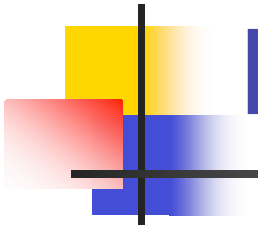
Some takeaways

- Late binding of memory and tight integration of memory allocation with compiler are crucial to getting high efficiency with implicit parallel structures
  - Compare to boost::shared_ptr<> or expression templates
- Plenty of information is available to the compiler
- Need to support legacy languages at least for setup and and I/O support

Sisal is alive

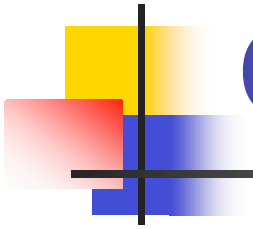
- http://sourceforge.net/projects/sisal/
  - 2,286 downloads





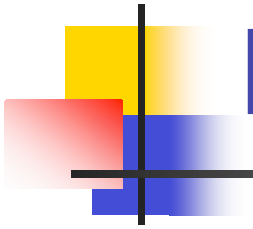
Maintenance and future work

- SLAg (Sisal Lives AGain)
- Ground up rewrite and modernization
- C-like, single assignment syntax
- Interested in distributed processing streams and splitting work with a GPU co-processor
- It's a hobby – don't hold your breath ☺



Closing thoughts…

- Being right doesn't mean squat!
- In 1995 I predicted the only thing that could save Sisal was a multithreaded game platform or an SMP on everyone's desktop ☺
- We learned many painful lessons in squeezing efficient implementations out of dataflow code. We published them too! Remember to look as far back as the early 1980's.



References and info

Ackerman, W.B. and J.B. Dennis. *Val – A value oriented algorithmic language.* MIT Technical Report LCS/TR-218, MIT, Cambridge, MA, June 1979

Feo, J.T. and D.C. Cann and Rodney Oldehoeft. *A report on the Sisal Language Project*. UCRL-102440. Journal of Parallel and Distributed Computing. 1990.

http://tamanoir.ece.uci.edu/projects/sisal/sisaltutorial

http://www.westnet.com/mirrors/99bottles/

I also have paper copies of all the seminal Sisal papers and reports. Most are also available from www.llnl.gov's TID (though many are missing)