The New, Fantastic R-Array

Tom Epperly, Gary Kumfert & Jim Leek

Center for Applied Scientific Computing
January 27, 2005

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

UCRL-PRES-209156
Outline

- What's the big deal?
- What's the catch?
- How is this possible?
- How can I get one?
- Yes, but what's my code going to look like?
- How did this go from an idea to reality in 1.5 months?
What's the big deal?

- Arrays appear more “natural” in C, C++, Fortran 90 and particularly Fortran 77
- Developers need less or no code to translate between their array data structures to SIDL's data structure
- SIDL generated APIs can match signatures from well known legacy APIs
- Less performance overhead due to avoiding a malloc & free calls
What's the catch?

- Only in & inout modes supported
- R-arrays must be contiguous and column-major ordered
- No NULL r-arrays
- Implementation cannot reshape or replace an inout r-array
- R-arrays are limited to int, long, float and double
- Lower index is always 0
How is this possible?

- Changing the semantics of inout makes it possible
- Normal SIDL arrays have identical semantics to SIDL objects
  - ability to deleteRef and replace the array severely constrains how arrays must be passed
- inout for r-arrays means the data is passed from caller to callee and back
How can I get one?

- Download and install Babel 0.10.0 (or later)
- Modify your SIDL files to use the new r-array syntax
- Regenerate your client and server code to use the new API
- Edit your client code and impls
New r-array SIDL syntax

- in rarray\textless type[, dimension]\textgreater arg(indices)
  inout rarray\textless type[, dimension]\textgreater arg(indices)
- The SIDL declaration also must include the declarations of the index variables
- Example:
  ```c
  void solve(in rarray\textless double, 2\textgreater A(m,n),
             inout rarray\textless double\textgreater x(n),
             in rarray\textless double\textgreater b(m),
             in int m,
             in int n);
  ```
Additional notes on r-array syntax

- Number of index variables must match the dimension of the array
- Index variables can be reused for other arguments
- Index arguments can appear anywhere in the argument list
- Values of index variables determine size of array
Yes, but what's my code going to look like?

- Watch out, A is in column-major order
- C client-side signature for solve
- Macros for column-major are available

```c
/** C client-side API for solve method */
void num_Linsol_solve(/*in*/ num_Linsol self,
    /*in*/ double* A,
    /*inout*/ double* x,
    /*in*/ double* b,
    /*in*/ int32_t m,
    /*in*/ int32_t n);
```
C server-side signature

```c
void
impl_num_Linsol_solve(/*in*/ num_Linsol self,
    /*in*/ double* A, /*inout*/ double* x,
    /*in*/ double* b,
    /*in*/ int32_t m, /*in*/ int32_t n)
{
    /* DO-NOT-DELETE splicer.begin(num.Linsol.solve) */
    /* Insert the implementation of the solve method here... */
    /* DO-NOT-DELETE splicer.end(num.Linsol.solve) */
}
```
C++ client-side signature

- C++ provides overloaded stub methods
- Note m & n don't appear in 2nd method

```c++
void solve (/*in*/ double* A,
            /*inout*/ double* x,
            /*in*/ double* b,
            /*in*/ int32_t m,
            /*in*/ int32_t n) throw ();

void solve (/*in*/ ::sidl::array<double> A,
            /*inout*/ ::sidl::array<double>& x,
            /*in*/ ::sidl::array<double> b)
    throw();
```
Fortran 77 client-side binding!

- Note array lower index is 0

```fortran
subroutine num_Linsol_solve_f(self, $  A, x, b, m, n)
  implicit none
C in num.Linsol self
  integer*8 self
  integer*4 m, n
C in rarray<double,2> A(m,n)
  double precision A(0:m-1, 0:n-1)
C inout rarray<double> x(n)
  double precision x(0:n-1)
C in rarray<double> b(m)
  double precision b(0:m-1)
end
```
subroutine num_Linsol_solve_fi(self, A, x, b, m, n)
implicit none

C in num.Linsol self
integer*8 self

C in int m
integer*4 m

C in int n
integer*4 n

C in rarray<double,2> A(m,n)
double precision A(0:m-1, 0:n-1)

C inout rarray<double> x(n)
double precision x(0:n-1)

C in rarray<double> b(m)
double precision b(0:m-1)

C DO-NOT-DELETE splicer.begin(num.Linsol.solve)
C Insert the implementation here...
C DO-NOT-DELETE splicer.end(num.Linsol.solve)
end
Like C++, F90 provides an overloaded client-side signature (no m & n args)

private :: solve_1s, solve_2s
interface solve
    module procedure solve_1s, solve_2s
end interface

recursive subroutine solve_1s(self, A, x, b)
    implicit none
    type(num_Linsol_t) , intent(in) :: self ! in num.Linsol self
    ! in array<double,2,column-major> A
    type(sidl_double_2d) , intent(in) :: A
    ! inout array<double,column-major> x
    type(sidl_double_1d) , intent(inout) :: x
    ! in array<double,column-major> b
    type(sidl_double_1d) , intent(in) :: b
    ! details deleted
end subroutine solve_1s
Here is the one that takes native Fortran 90 as arguments (m & n don't appear)

```fortran
recursive subroutine solve_2s(self, A, x, b)
    implicit none
    type(num_Linsol_t), intent(in) :: self ! in num.Linsol self
    ! in rarray<double,2> A(m,n)
    real (selected_real_kind(15, 307)), intent(in), dimension(:, :) :: A
    ! inout rarray<double> x(n)
    real (selected_real_kind(15, 307)), intent(inout), dimension(:) :: x
    ! in rarray<double> b(m)
    real (selected_real_kind(15, 307)), intent(in), dimension(:) :: b
    ! details deleted
end subroutine solve_2s
```
recursive subroutine num_Linsol_solve_mi(self, A, x, b, m, n)
    use num_Linsol
    use sidl_double_array
    use num_Linsol_impl
    ! DO-NOT-DELETE splicer.begin(num.Linsol.solve.use)
    ! DO-NOT-DELETE splicer.end(num.Linsol.solve.use)
    implicit none
    type(num_Linsol_t) :: self ! in
    integer (selected_int_kind(9)) :: m ! in
    integer (selected_int_kind(9)) :: n ! in
    real (selected_real_kind(15, 307)), dimension(0:m-1, 0:n-1) :: A ! in
    real (selected_real_kind(15, 307)), dimension(0:n-1) :: x ! inout
    real (selected_real_kind(15, 307)), dimension(0:m-1) :: b ! in

    ! DO-NOT-DELETE splicer.begin(num.Linsol.solve)
    ! Insert the implementation here...
    ! DO-NOT-DELETE splicer.end(num.Linsol.solve)
end subroutine num_Linsol_solve_mi
R-arrays for other languages

- In Java and Python, r-arrays are treated just like normal SIDL arrays
  - the index variables do not appear
How did this go from an idea to reality in 1.5 months?

- Babel users complained about having to wrap simple arrays as borrowed arrays
- LAPACK/Victor wanted simpler arrays
- Jeff Keasler (LLNL) suggested changing the array rules
- I flushed out the idea
- Gary came up with explicit variable declarations
- Jim and I coded it up