An Emerging, Portable Co-array Fortran Compiler for High-Performance Computing

John Mellor-Crummey, Yuri Dotsenko, Cristian Coarfa, Daniel Chavarría-Miranda
{johnmc, dotsenko, ccristi, danich}@cs.rice.edu

Programming Models for High-Performance Computing

Simple and expressive models for high performance programming based on extensions to widely used languages

- Performance: users control data and computation partitioning
- Portability: same language for SMPs, MPPs, and clusters
- Programmability: global address space for simplicity

Co-Array Fortran

A sensible alternative to these extremes

MPI

- Portable and widely used
- The programmer has explicit control over data locality and communication
- Using MPI can be difficult and error prone
- Most of the burden for communication optimization falls on application developers; compiler support is underutilized

HPF

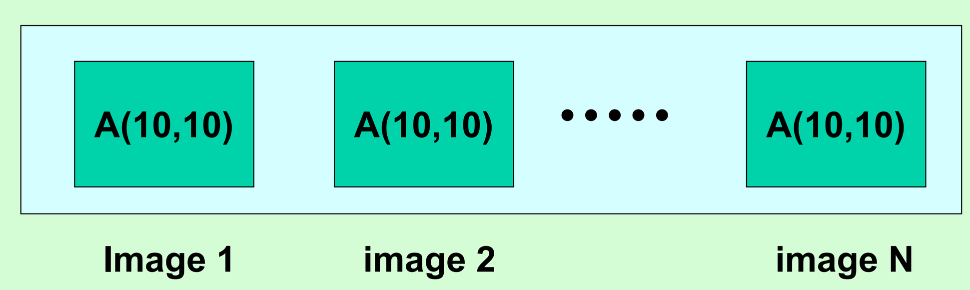
- The compiler is responsible for communication and data locality
- Annotated sequential code (semiautomatic parallelization)
- Requires heroic compiler technology
- The model limits the application paradigms: extensions to the standard are required for supporting irregular computation

Co-Array Fortran Language

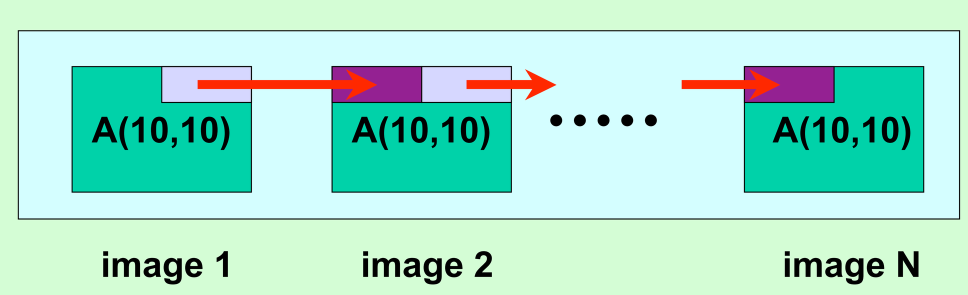
- SPMD process images
  - number of images fixed during execution
  - images operate asynchronously
- Both private and shared data
  - real a(20,20) — private: a 20x20 array in each image
  - real a(20,20) [*] — shared: a 20x20 array in each image
- Simple one-sided shared memory communication
  - x(:,j:j+2) = a(r,:) [p:p+2] — copy rows from p:p+2 into local columns
- Flexible synchronization
  - sync_team(team [,wait])
    - team = a vector of process ids to synchronize with
    - wait = a vector of processes to wait for (a subset of team)
- Pointers and dynamic allocation
- Parallel I/O

Explicit Data and Computation Partitioning

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




```
if (this_image() .lt. num_images()) then
  A(1:3,1:5)[this_image()+1] = A(1:3,6:10)
```



Finite Element Example

```
subroutine assemble(start, prin, ghost, neib, x)
  integer :: start(:), prin(:), ghost(:), neib(:)
  integer :: k1, k2, p
  real :: x(:) [*]
  call sync_all(neib)
  do p = 1, size(neib) ! Update from ghost regions
    k1 = start(p); k2 = start(p+1)-1
    x(prin(k1:k2)) = x(prin(k1:k2)) +
         x(ghost(k1:k2)) [neib(p)]
  enddo
  call sync_all(neib)
  do p = 1, size(neib) ! Update the ghost regions
    k1 = start(p); k2 = start(p+1)-1
    x(ghost(k1:k2)) [neib(p)] = x(prin(k1:k2))
  enddo
  call sync_all
end subroutine assemble
```

Co-Array Fortran enables simple expression of complicated communication patterns

Research Focus

- Enhancements to Co-Array Fortran model
  - Point-to-point one-way synchronization
  - Hints for matching synchronization events
  - Collective operation intrinsics
  - Split-phase primitives
- Synchronization strength-reduction
- Communication vectorization
- Platform-driven communication optimization
  - Transform as useful from 1-sided to two-sided and collective communication
  - Generate both fine-grain load/store and calls to communication libraries as necessary
  - Multi-model code for hierarchical architectures
  - Convert Gets into Puts
- Compiler-directed parallel I/O with UIUC
- Interoperability with other programming models

Implementation Status

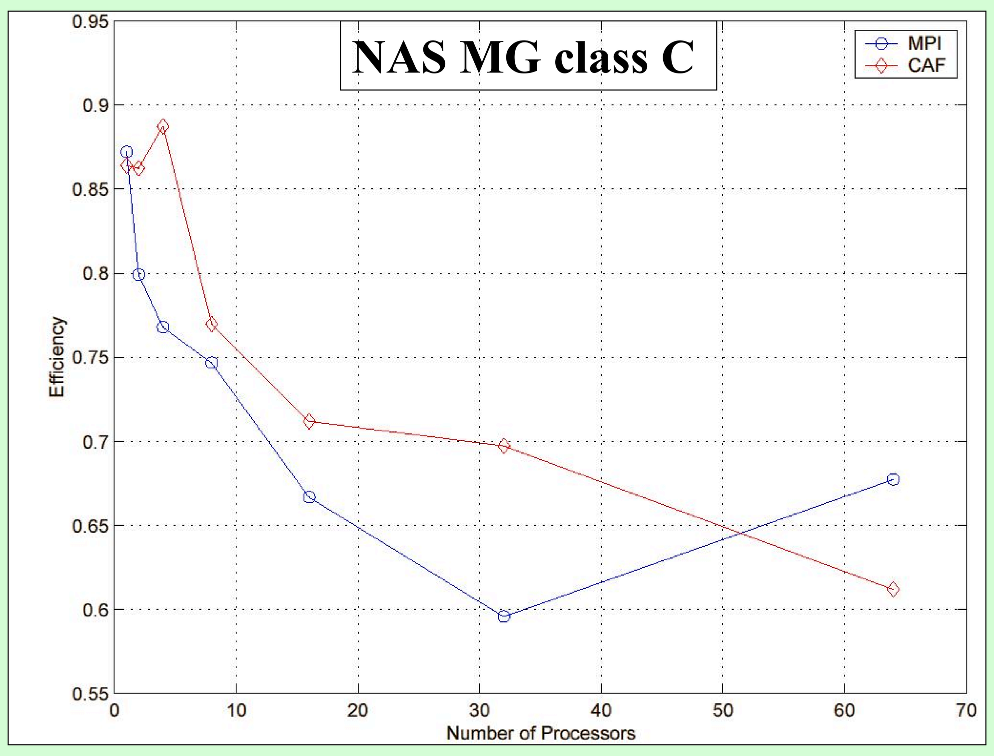
- Source-to-source code generation for wide portability
- Open source compiler will be available
- Working prototype for a subset of the language
- Current compiler implementation performs no optimization
  - each co-array access is transformed into a get/put operation at the same point in the code
- Code generation for the widely-portable ARMCI library for one-sided communication
- Front-end based on production-quality Open64 front end, modified to support source-to-source compilation

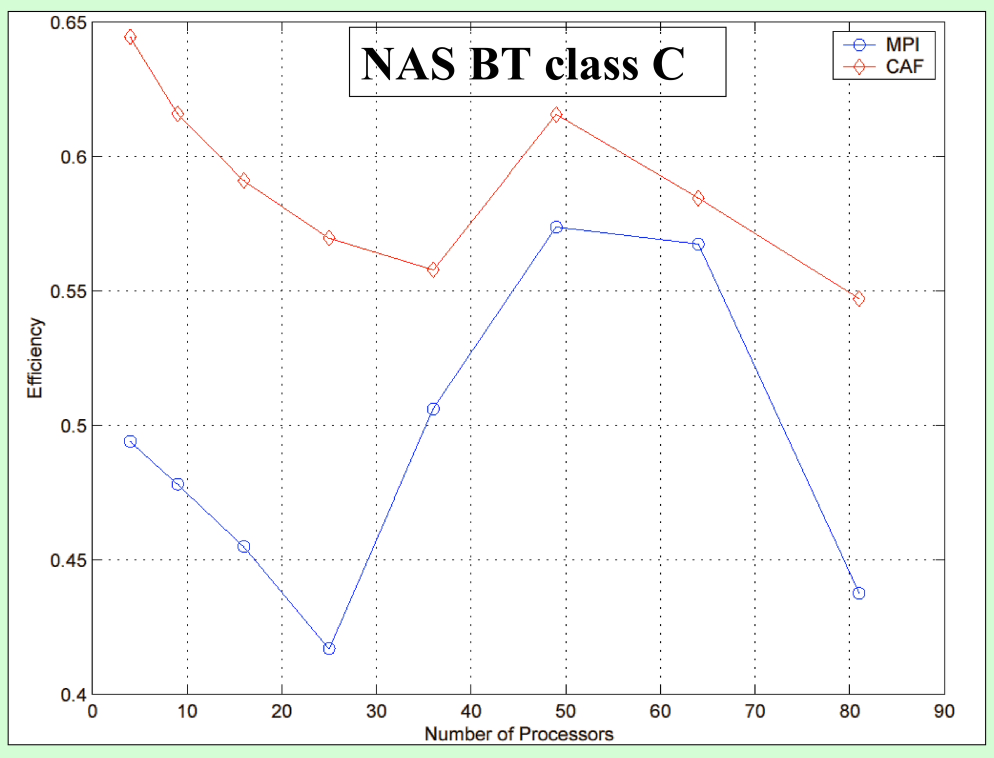
PUT Translation Example

```
. . .
real(8) a(0:N+1,0:N+1)[*]
me = this_image()
. . .
! ghost cell update
a(1:N,N+1)[left(me)] = a(1:N,0)
. . .
```

```
type CafHandleReal8
     integer:: handle
     real(8):: ptr(:,:)
end type
type(CafHandleReal8) a_caf
. . .
allocate( cafBuffer_1%ptr(1:N,0:0) )
cafBuffer_2%ptr => a_caf%ptr(1:N,N+1:N+1)
cafBuffer_1%ptr = a_caf%ptr(1:N,0)
call CafArmciPutS(a_caf%handle,left(me),
cafBuffer_1, cafBuffer_2)
deallocate( cafBuffer_1%ptr )
. . .
```

Early Performance Results





IA64 / Myrinet 2000