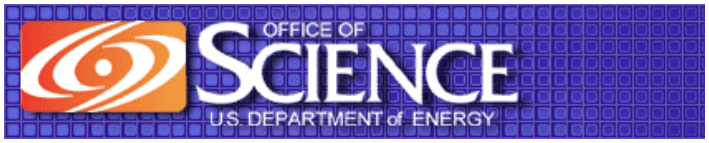
Co-array Fortran Performance and Potential: an NPB Experimental Study

Cristian Coarfa
Yuri Dotsenko
Jason Lee Eckhardt
John Mellor-Crummey

Department of Computer Science
Rice University



Parallel Programming Models

- Goals:
  - Expressiveness
  - Ease of use
  - Performance
  - Portability
- Current models:
  - ***OpenMP***: difficult to map on distributed memory platforms
  - ***HPF***: difficult to obtain high-performance on broad range of programs
  - ***MPI***: de facto standard; hard to program, assumptions about communication granularity are hard coded

Co-array Fortran – a Sensible Alternative

- ***Programming model overview***
- Co-array Fortran compiler
- Experiments and discussion
- Conclusions

Co-Array Fortran (CAF)

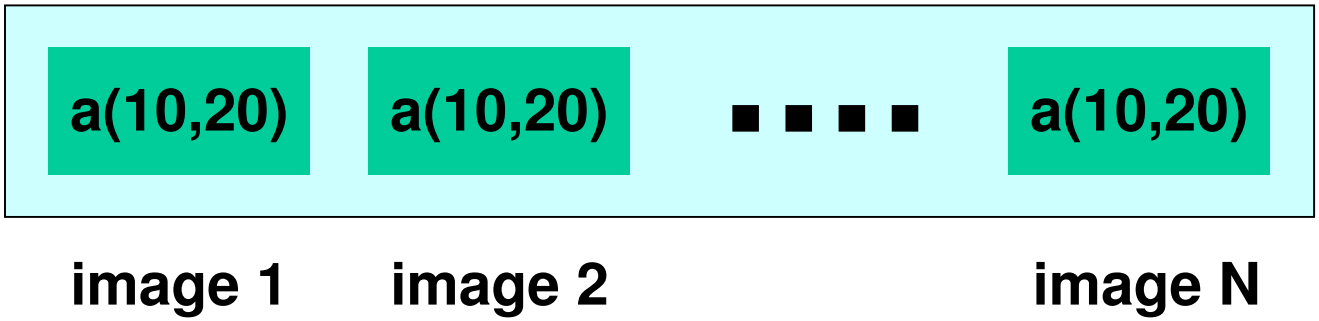
- Explicitly-parallel extension of Fortran 90/95 (Numrich & Reid)
- Global address space SPMD parallel programming model
  - one-sided communication
- Simple, two-level model that supports locality management
  - local vs. remote memory
- Programmer control over performance critical decisions
  - data partitioning
  - communication
- Suitable for mapping to a range of parallel architectures
  - shared memory, message passing, hybrid, PIM

CAF Programming Model Features

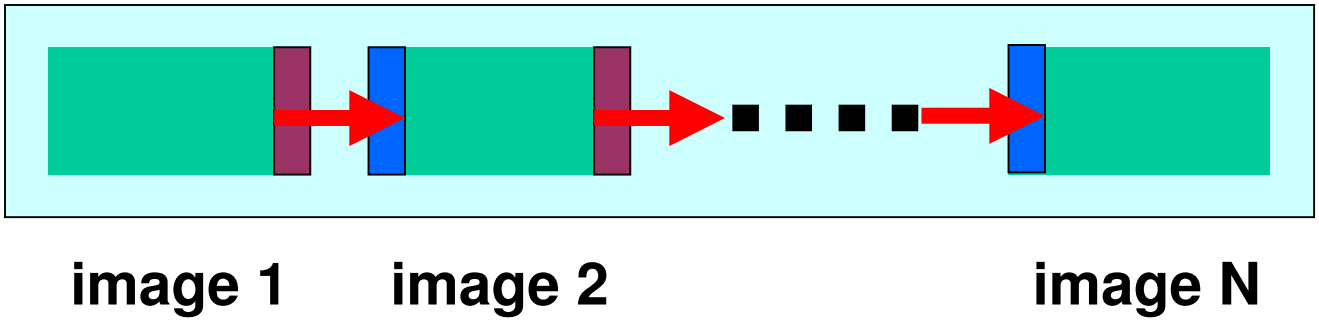
- SPMD process images
  - fixed number of images during execution
  - images operate asynchronously
- Both private and shared data
  - **real x(20, 20)** a private 20x20 array in each image
  - **real y(20, 20) [*]** a shared 20x20 array in each image
- Simple one-sided shared-memory communication
  - **x(:,j:j+2) = y(:,p:p+2) [r]** copy columns from p:p+2 into local columns
- Synchronization intrinsic functions
  - **sync_all** – a barrier
  - **sync_team([notify], [wait])**
    - **notify** = a vector of process ids to signal
    - **wait** = a vector of process ids to wait for, a subset of notify
- Pointers and (perhaps asymmetric) dynamic allocation
- Parallel I/O

One-sided Communication with Co-Arrays

```
integer a(10,20)[*]
```




```
if (this_image() > 1)
      a(1:10,1:2) = a(1:10,19:20)[this_image()-1]
```



Finite Element Example (Numrich)

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

Proposed CAF Model Refinements

- Initial implementations on Cray T3E and X1 led to features not suited for distributed memory platforms
- Key problems and solutions
  - Restrictive memory fence semantics for procedure calls
    - ⇨ goal: enable programmer to overlap one-sided communication with procedure calls
  - Overly restrictive synchronization primitives
    - ⇨ add unidirectional, point-to-point synchronization
  - No collective operations
    - ⇨ add CAF intrinsics for reductions, broadcast, etc.

Co-array Fortran – A Sensible Alternative

- Programming model overview
- ***Co-array Fortran compiler***
- Experiments and discussion
- Conclusions

Portable CAF Compiler

- Compile CAF to Fortran 90 + runtime support library
  - source-to-source code generation for wide portability
  - expect best performance by leveraging vendor F90 compiler
- Co-arrays
  - represent co-array data using F90 pointers
  - allocate storage with dope vector initialization outside F90
  - access co-array data through pointer dereference
- Porting to a new compiler / architecture
  - synthesize compatible dope vectors for co-array storage: CHASM library (LANL)
  - tailor communication to architecture
    - today: ARMCI (PNNL)
    - future: compiler and tailored communication library

CAF Compiler Status

- Near production-quality F90 front end from Open64
- Working prototype for CAF core features
  - allocate co-arrays using static constructor-like strategy
  - co-array access
    - access remote data using ARMCI get/put
    - access process local data using load/store
  - co-array parameter passing
  - sync_all, sync_team, sync_wait, sync_notify synchronization
  - multi-dimensional array section operations
- Co-array communication inserted around statements with co-array accesses
- Currently no communication optimizations

Co-array Fortran – A Sensible Alternative

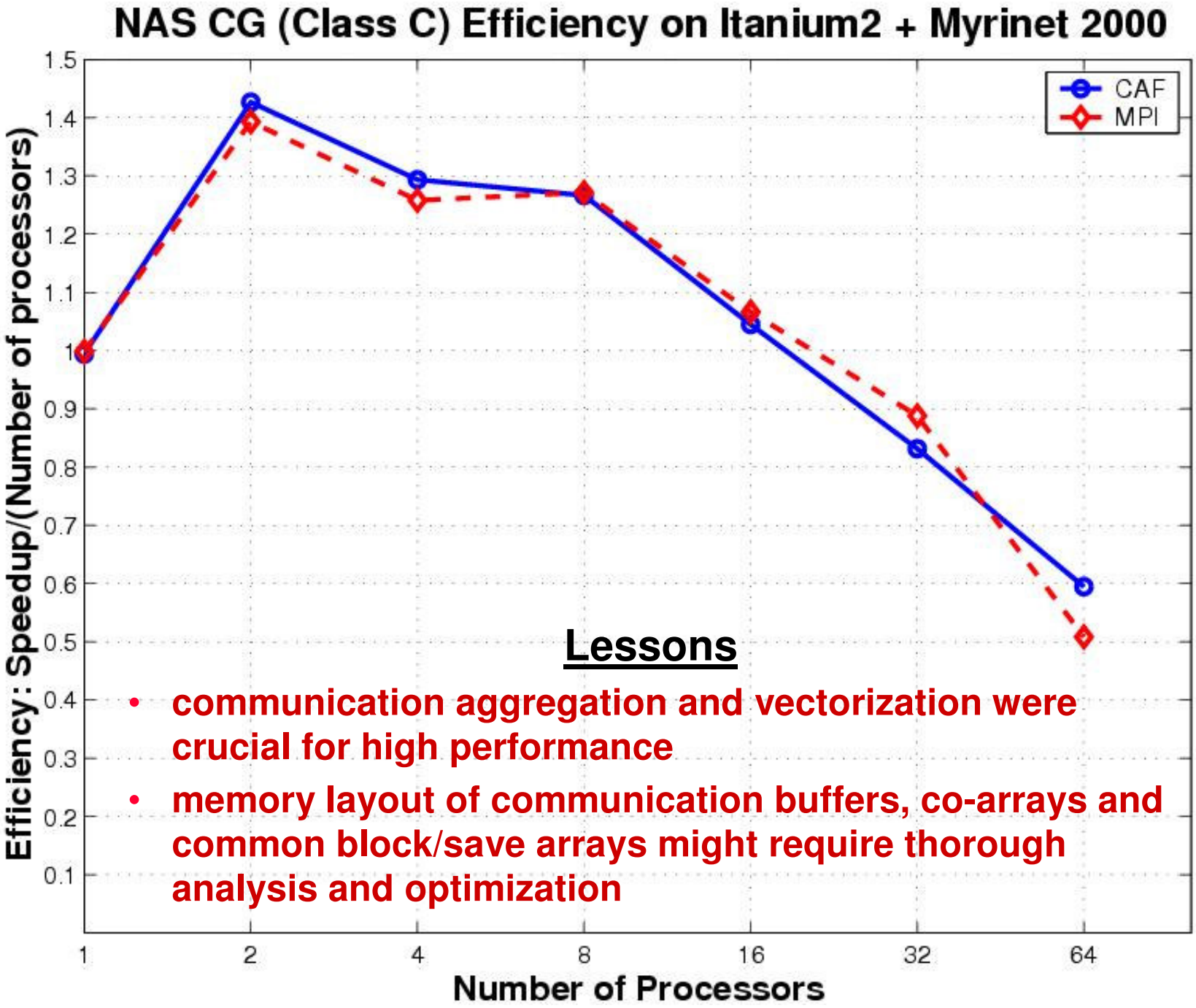
- Programming model overview
- Co-array Fortran compiler
- ***Experiments and Discussion***
- Conclusions

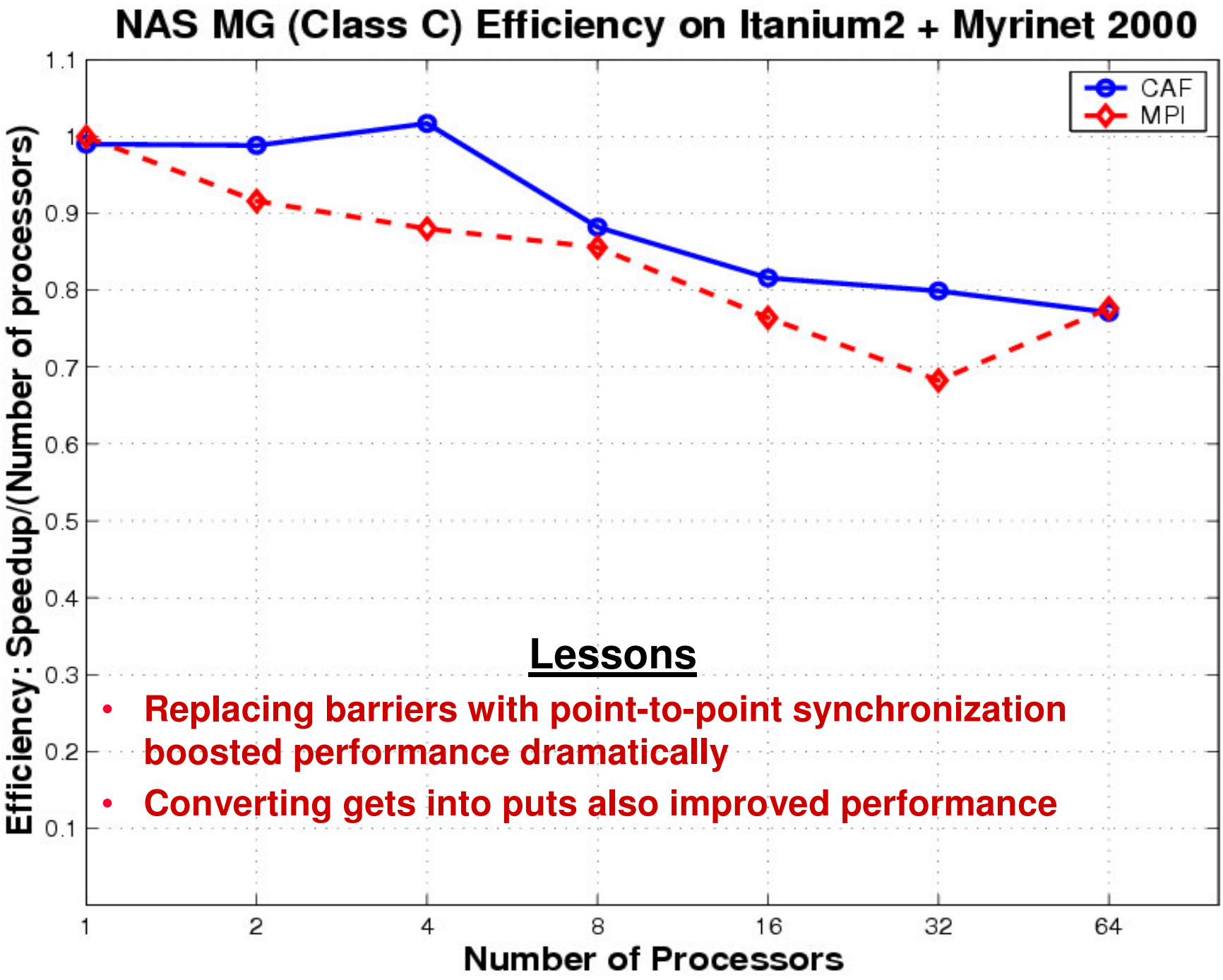
Platform

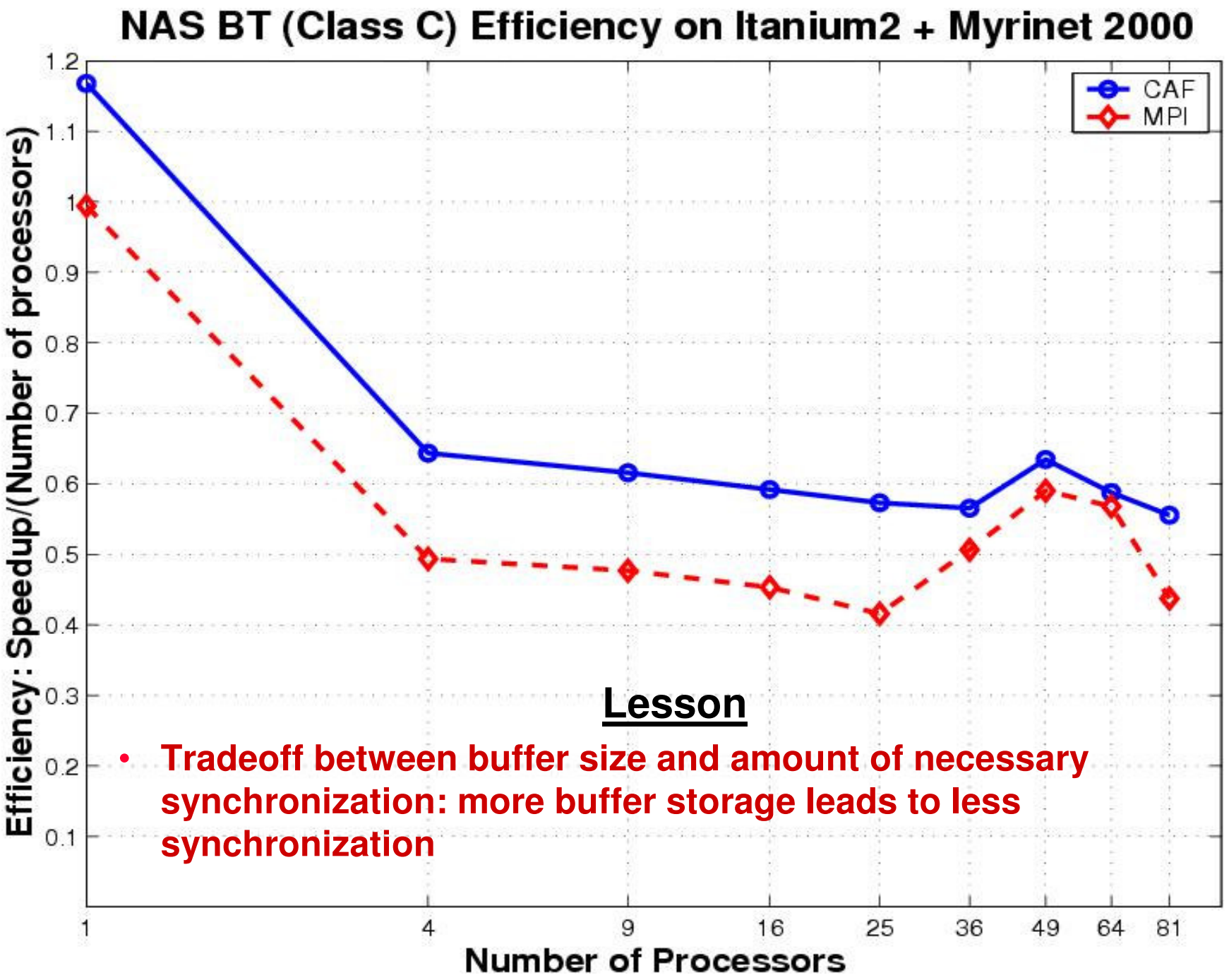
- 96 HP zx6000 workstations
  - dual 900Mhz Itanium2 32KB/256KB/1.5MB L1/L2/L3 cache
  - 4GB ram
- Myrinet 2000 interconnect
- Red Hat Linux, 2.4.20 kernel plus patches
- Intel Fortran Compiler v7.1
- ARMCI 1.1-beta
- MPICH-GM 1.2.5..10

NAS Benchmarks 2.3

- Benchmarks by NASA:
  - Regular, dense-matrix codes: MG, BT, SP
  - Irregular codes: CG
  - 2-3K lines each (Fortran 77)
  - Widely used to test parallel compiler performance
- NAS Versions:
  - NPB2.3b2 : Hand-coded MPI
  - NPB2.3-serial : Serial code extracted from MPI version
  - NPB2.3-CAF: CAF implementation, based on the MPI version

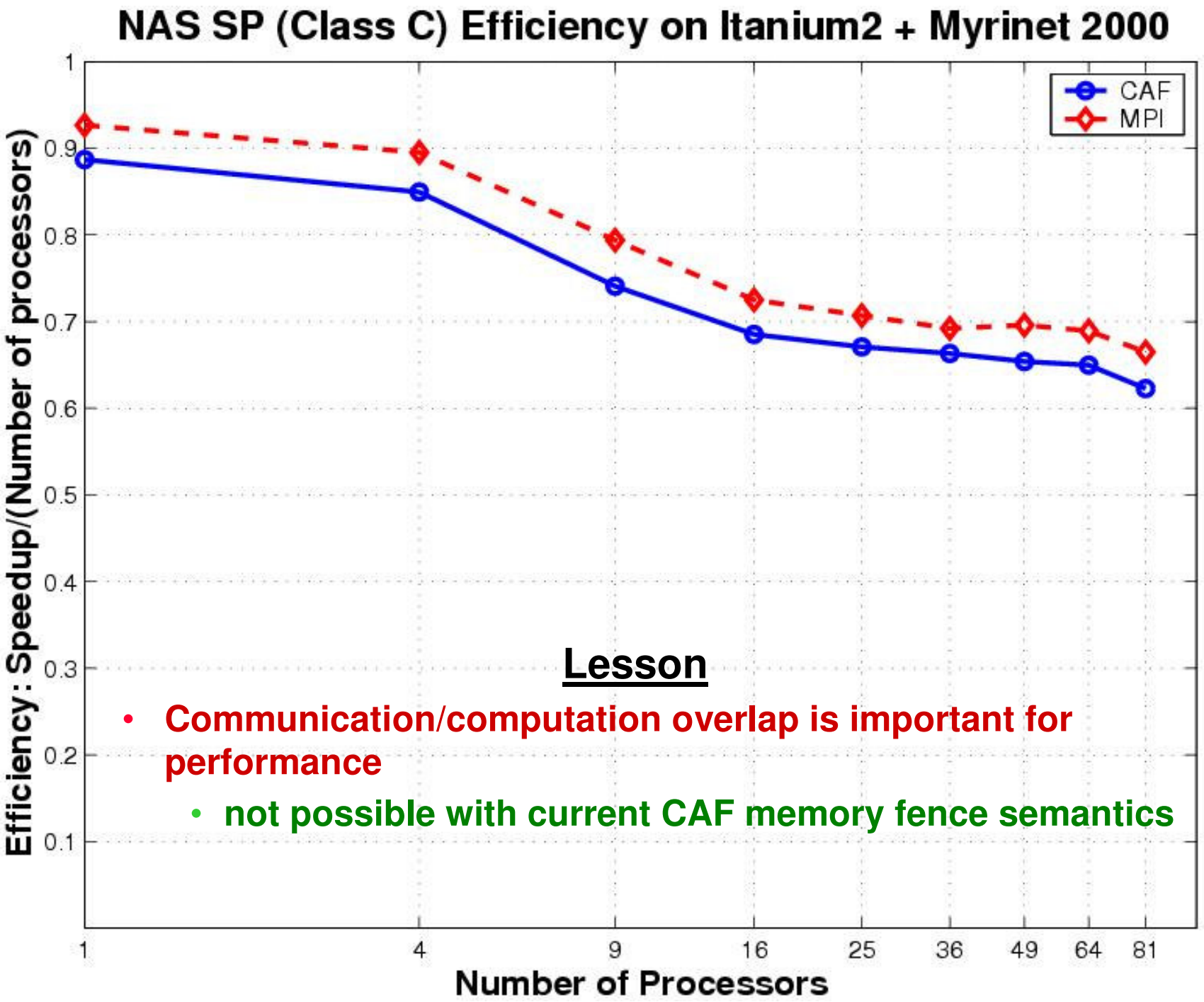

NAS CG (Class C) Efficiency on Itanium2 + Myrinet 2000
Efficiency: Speedup/(Number of processors)
Number of Processors
CAF
MPI
1.5
1.4
1.3
1.2
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
1
2
4
8
16
32
64
Lessons
• communication aggregation and vectorization were crucial for high performance
• memory layout of communication buffers, co-arrays and common block/save arrays might require thorough analysis and optimization


NAS MG (Class C) Efficiency on Itanium2 + Myrinet 2000
Efficiency: Speedup/(Number of processors)
Number of Processors
CAF
MPI
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
1
2
4
8
16
32
64
Lessons
• Replacing barriers with point-to-point synchronization boosted performance dramatically
• Converting gets into puts also improved performance



Lesson

- **Tradeoff between buffer size and amount of necessary synchronization: more buffer storage leads to less synchronization**



Lesson

- Communication/computation overlap is important for performance
  - not possible with current CAF memory fence semantics

Experiments Summary

- On cluster-based architectures, to achieve best performance with CAF, a user or compiler must
  - Vectorize (and perhaps aggregate) communication
  - Reduce synchronization strength
    - replace all-to-all with point-to-point where sensible
  - Convert gets into puts where gets are not a h/w primitive
  - Consider memory layout conflicts: co-array vs. regular data
  - Overlap communication with computation
- CAF language enables many of these to be performed manually at the source level
- Plan to automate optimizations, but compiler might need user hints

Conclusions

- CAF performance is comparable to highly tuned hand-coded MPI
  - even without compiler-based communication optimizations!
- CAF programming model enables optimization at the source level
  - communication vectorization
  - synchronization strength reduction

  ⇨ achieve performance today rather than waiting for tomorrow's compilers
- CAF is amenable to compiler analysis and optimization
  - significant communication optimization is feasible, unlike for MPI
  - optimizing compilers will help a wider range of programs achieve high performance
  - applications can be tailored to fully exploit architectural characteristics
    - e.g., shared memory vs. distributed memory vs. hybrid