Using Co-Array Fortran to Enhance the Scalability of the EPIC Code

Jef Dawson

Army High Performance Computing Research Center

jdawson@ahpcrc.org

Outline

- Project Description
- Co-Array Fortran (CAF) Overview
- EPIC Overview & Scaling Issues
- CAF Code Examples
- Performance Results
- Observations

Army High Performance Computing Research Center
Support Infrastructure Contract
Network Computing Services, Inc. , Prime Contractor
Contract DAAD19-03-D-0001

This presentation was developed in connection with contract DAAD19-03-D-0001 with the U.S. Army Research Laboratory. The views and conclusions contained in this presentation are those of the authors and should not be interpreted as presenting the official policies or positions, either expressed or implied, of the U.S. Army Research Laboratory or the U.S. Government unless so designated by other authorized documents. Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Project Description

- Investigate benefits of CAF
  - Replace MPI with CAF in key regions
- Begin to establish CAF ‘Best Practice’
- Improve EPIC performance for user base

CAF Overview

In a nutshell, CAF replaces this:

```
CALL MPI_SEND(A,N,MPI_REAL,MYPE+M,TAG, COMM,IERR)
CALL MPI_RECV(A,N,MPI_REAL,MYPE-M,TAG, COMM,ISTAT,IERR)
```

with this:

```
A(1:N)[MYPE+M]=A(1:N)
```

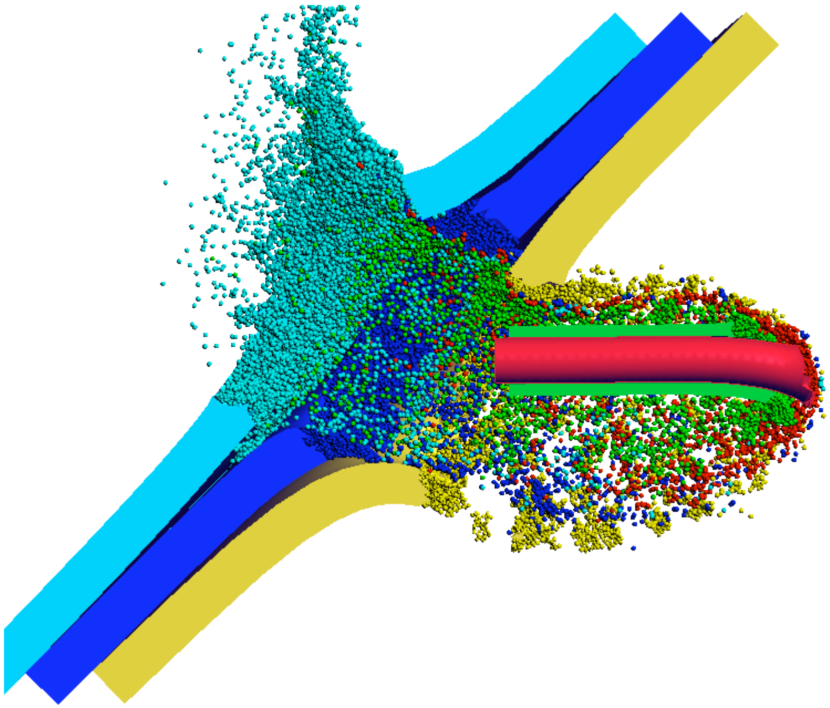
- **Both code fragments move N elements of A from CPU MYPE to CPU MYPE+M**
- **The CAF code is:**
  - **Actual language syntax, rather than library calls**
  - ***One-sided*, so that only one CPU needs to execute code for data to be communicated**
- **CAF on the X1E is *native***
  - **Hardware instructions directly load/store remote data**

Advantages of CAF

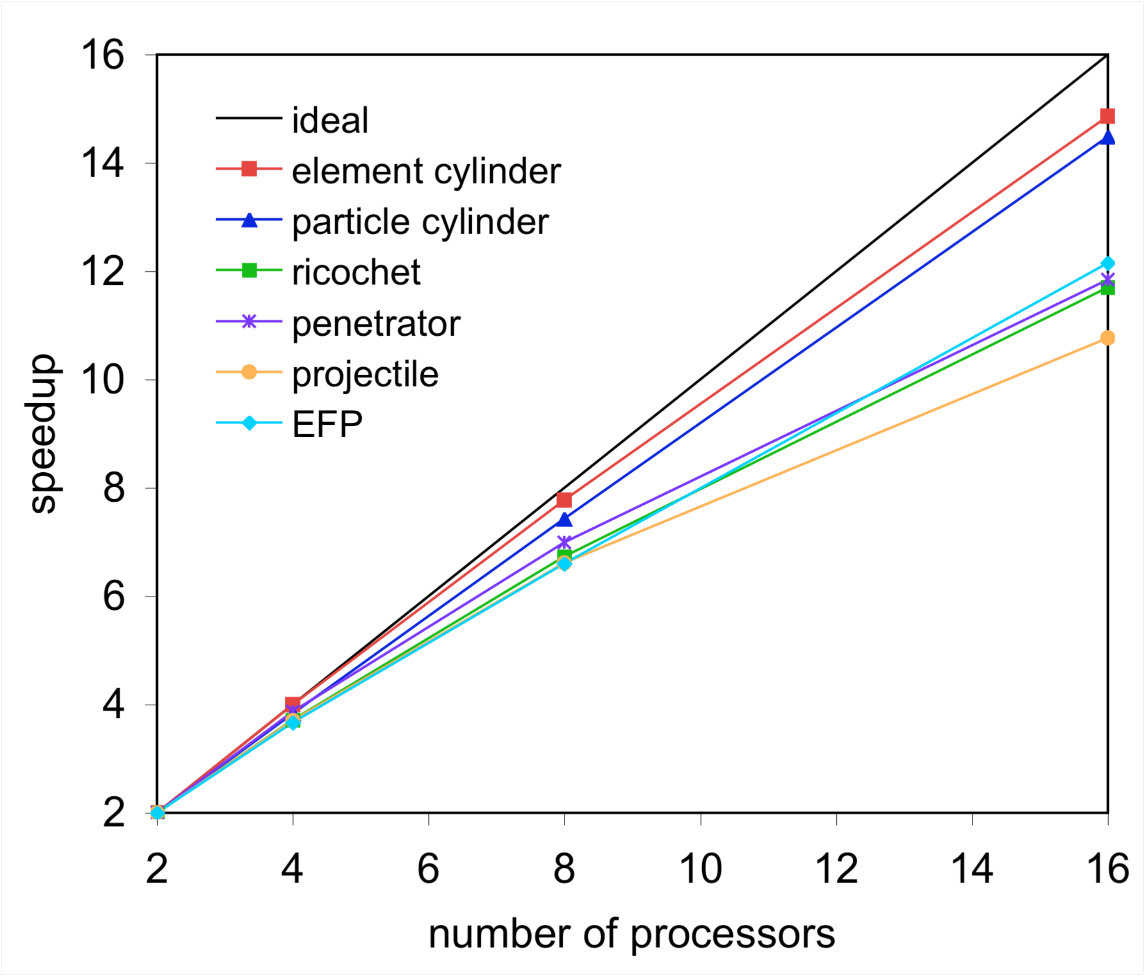
- **Performance**
  - **No function calls for communication**
    - No function call overhead
    - Compilers can optimize inter-processor communication
    - Non-native CAF compilers don’t have these advantages
- **Productivity**
  - **CAF Fortran syntax often more readable than complex MPI calling sequence**

EPIC Overview

- **Elastic-Plastic Impact Computations**
- A finite element code used to simulate projectile-target interaction
- Complex phenomena
- Computationally intensive
- Large legacy code initially developed for serial execution



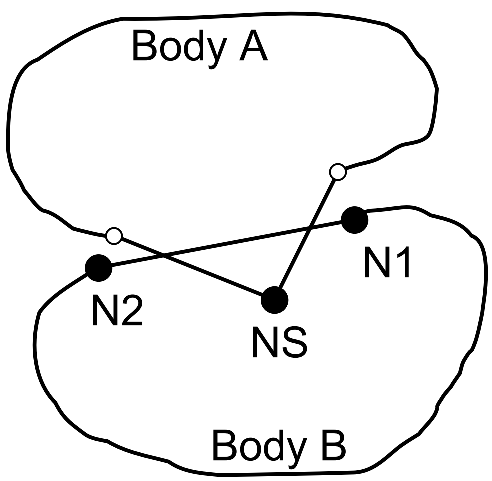
MPI Scaling of EPIC Cases



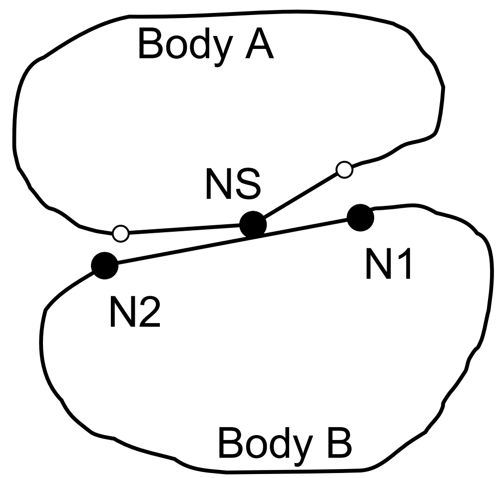
Contact Algorithm Overview

- Simulating contact of multiple bodies is a critical feature of EPIC
- When bodies make contact, nodes 'penetrate' surfaces
- The contact algorithm consists of iterative adjustment to eliminate penetration

Before Adjustment



After Adjustment



Contact Scalability Issues

- Contact between bodies is unpredictable
  - Good load balancing is tricky
  - Inter-processor communication is complex
- The contact algorithm is iterative
  - Inter-processor communication is frequent
  - Latency limits scaling

CAF Parallel Performance Optimizations

- Low latency communication
  - Critical when messages are small
- Fine-grain communication hiding
  - Work on incoming data as it arrives, rather than waiting
  - A form of *pipelining*
  - Uniquely valuable when no independent work is available to hide communication time

Example 1: Low Latency

- **CAF code 8 times faster than MPI**
  - **Latency advantage important for small messages**

Original Code:

```
CALL  MPI_ALLGATHER(NUMALTERED,1,
      MPI_INTEGER,ITMP,NUM,MPI_INTEGER,
       MPI_COMM_WORLD,IER)
```

CAF Code:

```
DO I=1,NPES
    ITMP(I)=NUMALTERED(1)[I]
ENDDO
```

Example #2: Fine-Grain Communication Hiding

Original Code:

- The computations in the loop cannot begin until the 4 MPI_ALLREDUCE calls finish
- Unnecessary computation & communication for many elements of PX, PY, and PZ whose values are 0.0

```
TEMP(1:SLDTOT)=PX(1:SLDTOT)
CALL MPI_ALLREDUCE(TEMP, PX, SLDTOT,   &
                   MPI_DOUBLE_PRECISION,   &
        MPI_SUM, MPI_COMM_WORLD, IER)
TEMP(1:SLDTOT)=PY(1:SLDTOT)
CALL MPI_ALLREDUCE(TEMP, PY, SLDTOT,    &
                   MPI_DOUBLE_PRECISION,   &
         MPI_SUM, MPI_COMM_WORLD, IER)
TEMP(1:SLDTOT)=PZ(1:SLDTOT)
CALL MPI_ALLREDUCE(TEMP, PZ, SLDTOT,     &
                   MPI_DOUBLE_PRECISION,    &
              MPI_SUM, MPI_COMM_WORLD, IER)
LTEMP(1:SLDTOT)=UPDATE(1:SLDTOT)
CALL MPI_ALLREDUCE(LTEMP,UPDATE,SLDTOT, &
                         MPI_LOGICAL,MPI_LOR, &
                        MPI_COMM_WORLD,IER)
DO M=1,SLDTOT
   IF (UPDATE(M)) THEN
      CALL NFIX(IXYZ(M),IRIG,IXX,IYY,IZZ)
      X(M) = X(M) + PX(M)*(1-IXX)
      Y(M) = Y(M) + PY(M)*(1-IYY)
      Z(M) = Z(M) + PZ(M)*(1-IZZ)
   ENDIF
ENDDO
```

Example #2: Fine-Grain Communication Hiding

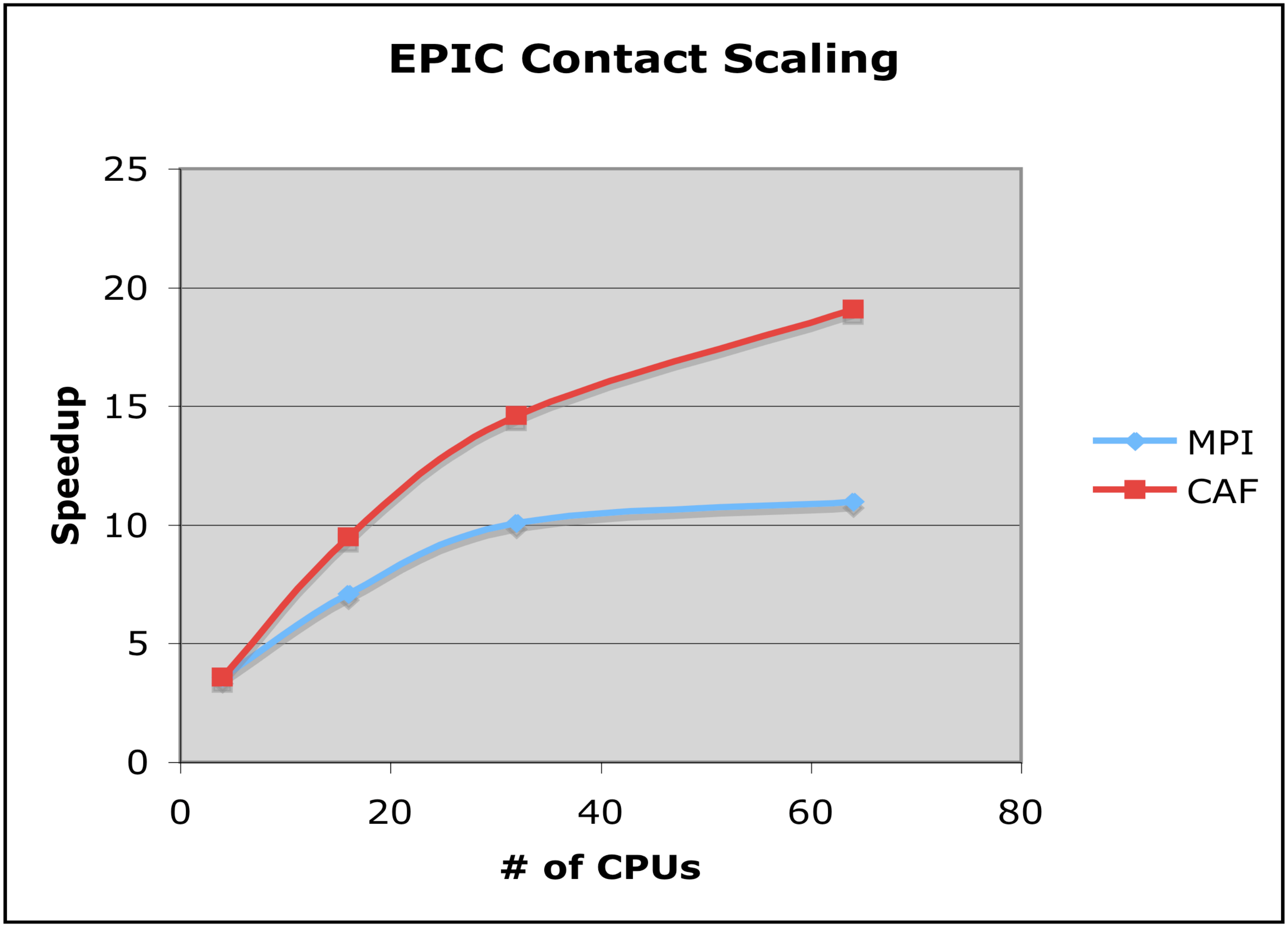
CAF Code:

```
ICOUNT=0
DO I=1,SLDTOT
  IF (UPDATE(I)) THEN
    ICOUNT=ICOUNT+1
    ICTEMP(ICOUNT)=I
    PXTEMP(ICOUNT)=PX(I)
    PYTEMP(ICOUNT)=PY(I)
    PZTEMP(ICOUNT)=PZ(I)
  ENDIF
ENDDO
CALL SYNC_IMAGES()

DO ITER=1,NPES
  ISRC=MYPN-ITER+1
  IF (ISRC.LE.0) ISRC=ISRC+NPES
    DO M=1,ICOUNT[ISRC]
      GLOB=ICTEMP(M)[ISRC]
      CALL NFIX(IXYZ(GLOB),IRIG,IXX,IYY,IZZ)
      X(GLOB)=X(GLOB)+PXTEMP(M)[ISRC]*(1-IXX)
      Y(GLOB)=Y(GLOB)+PYTEMP(M)[ISRC]*(1-IYY)
      Z(GLOB)=Z(GLOB)+PZTEMP(M)[ISRC]*(1-IZZ)
    ENDDO
ENDDO
```

- Only non-zero values are communicated
- Virtually all communication is 'hidden' by concurrent computations
  - Vector loads directly from remote memory
- 5 times as fast as the original MPI code

Performance Results



Observations

- CAF significantly faster for latency-sensitive communication
- CAF allows fine-grain communication hiding not possible with MPI
- The implementation matters
  - Requires underlying support for pipelining remote memory operations
- CAF offers a natural way to take full advantage of high performance hardware
- Adding CAF to MPI is painless (on the X1E)
- If pipelining is important to 'hide' local memory operations (it is), it must be *really* important for hiding remote memory operations