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

Jef Dawson
Army High Performance Computing Research Center,
Network Computing Services, Inc.

ABSTRACT: *Supercomputing users continually seek to solve more demanding problems by increasing the scalability of their applications. These efforts are often severely hindered by programming complexity and inter-processor communication costs. Co-Array Fortran (CAF) offers innate advantages over previous parallel programming models, both in programmer productivity and performance. This presentation will discuss the fundamental advantages of CAF, the importance of underlying hardware support, and several code examples from the EPIC code showing how CAF is being effectively used to improve scalability.*

KEYWORDS: Co-Array Fortran, EPIC, MPI, communication, latency, overlapping, pipelining

1. Introduction

The Army High Performance Computing Research Center (AHPCRC) is a Government-University-Industry partnership conducting high performance computing (HPC) research supporting the United States Army's HPC goals. A key area of activity at the Center is the investigation of methods to improve the scalability of complex, mission critical applications.

The Center operates a wide variety of advanced computing platforms, including a Cray X1E. The X1E features a completely global address space, allowing each CPU to directly load and store data to and from any other CPU's memory. This feature offers some fundamental advantages for the scaling of parallel applications.

The Message Passing Interface (MPI), used for inter-processor communication in most parallel programs, cannot take full advantage of a global address space, as will be discussed in more detail later. Co-Array Fortran (CAF), a simple extension to Fortran, has the potential to make the performance advantages of a global memory space more readily available to parallel programmers.

2. Project Description

This paper describes an ongoing project to improve the scalability of the EPIC (**E**lastic-**P**lastic **I**mpact **C**omputations) code, using CAF to replace MPI in selected areas.

This project has several related goals. The first is to investigate and demonstrate the benefits of CAF for parallel performance. The second is to begin developing a set of practical guidelines to help programmers obtain the best results when using CAF. The third is to deliver improved parallel performance for end users of EPIC.

3. Introduction to CAF

While a complete description of CAF is beyond the scope of this paper, a brief overview will be given, as background for the discussion that follows.

CAF is a small extension to the Fortran language, which will be included in the next revision of the Fortran standard. Most significantly, CAF provides syntax to express inter-processor communication without the use of function calls.

To illustrate this, consider the following MPI code:

```
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)
```

This simple code communicates 'N' elements of array 'A' from one CPU to another CPU, using the MPI_SEND and MPI_RECV functions. To achieve the same result using CAF, the following code could be used:

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

In this code, the inter-processor communication is expressed using a simple assignment statement. The array 'A' is a *co-array*. A co-array is an array that has a copy on every CPU running the program. Each CPU's copy of 'A' can be referenced using its *co-index*, specified in the square brackets. When a co-array is referenced with no co-index, the reference is to the copy on the CPU executing the statement. Thus, the assignment statement above results in the same inter-processor communication as the preceding MPI code.

The co-array extension is the primary feature of CAF. It provides an elegant and readable syntax to express inter-processor communication. It is also *one-sided*, in that only one CPU needs to execute code for communication to occur.

This syntax also offers a fundamental performance benefit. Because inter-processor communication is represented by simple assignment statements, a compiler can perform optimisations that would otherwise be prevented by the presence of function calls. This will be illustrated by some examples.

It should be noted that the performance characteristics of CAF are dependent on the underlying implementation. The work described here was carried out on a Cray X1E with a 'native' CAF compiler that takes full advantage of the system's global address space. This allows the instructions that move data between CPUs to be optimised in the same ways that local memory operations are optimised by most modern compilers. CAF implementations that rely on emulated global address space may not be able to offer the same performance benefits.

4. Overview of EPIC scalability issues

EPIC is a finite element application used to simulate a variety of complex problems involving extreme deformation of solids. One such problem is that of projectile-target interaction, as shown in Figure 1.

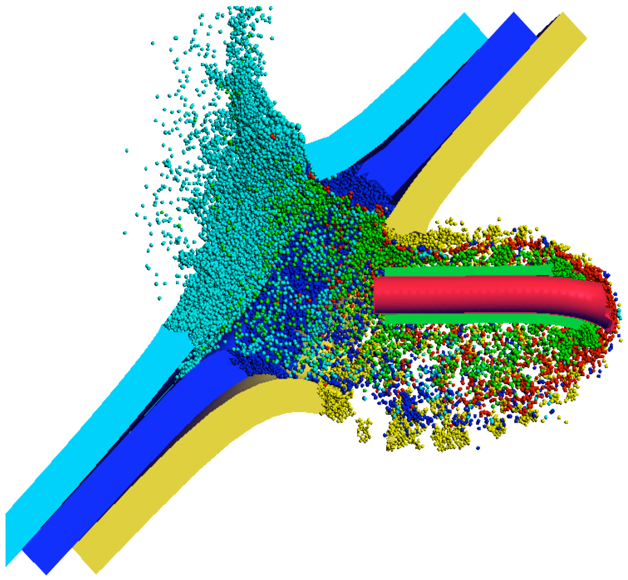

Figure 1. This image from an EPIC simulation shows a projectile just after passing through three layers of armor plate. The projectile has made a hole in all three plates, and has been severely damaged in the process.

The problems for which EPIC is used involve very complex physical phenomena. As a result, EPIC simulations tend to be very computationally intensive. To allow more demanding and realistic problems to be solved in shorter run-times, EPIC has been parallelized, using MPI.

One portion of the EPIC application is problematical to parallelize efficiently. This is the *contact* algorithm. This algorithm is responsible for correctly accounting for contact between distinct bodies during the simulation, as shown in Figure 2.

The nature of the contact problem makes efficient parallel execution inherently difficult. When a program is parallelized, the underlying problem is carefully divided into parts, or *sub-domains*. Ideally, the programmer knows in advance which sub-domains must communicate with each other, and can design the communication scheme for optimum performance.

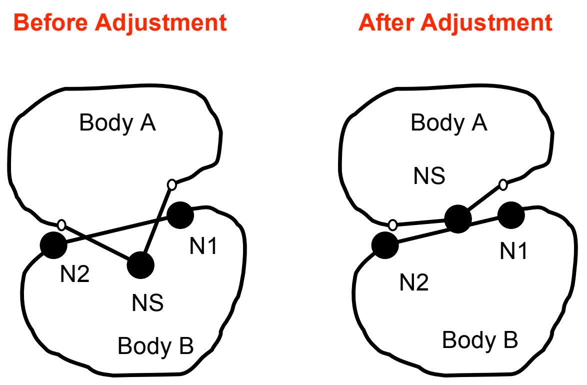

Figure 2. This schematic illustrates the operation of the EPIC contact algorithm. During the course of a simulation, body A and body B have moved in such a way that node NS, a part of body A, has penetrated the segment between nodes N1 and N2, which is part of body B. The contact algorithm rectifies this non-physical situation by iteratively deforming the two bodies such that the penetration does not occur, and such that the velocities and momenta of all bodies are physically correct.

In EPIC, it is not known in advance which sub-domains will be in contact with each other, so it is impossible to specify an optimal communication scheme in advance. Moreover, the contact algorithm requires several iterations, each of which involve inter-processor communication.

From a parallel programmer's perspective, latency is the main problem introduced by the frequent inter-processor communication required by the EPIC contact algorithm. Even though a relatively small amount of data is communicated with each MPI function call, the overhead incurred by the large number of calls limits scalability.

In this project, we are working to improve the scalability of the contact algorithm by reducing the latency penalty. CAF is helpful in two distinct ways. First, the minimum attainable latency is lower with CAF than with MPI. At least part of the reason for this is that CAF does not require a function call to transfer data between CPUs, so there is no function call overhead. Second, using CAF syntax allows the compiler to optimise the code to 'hide' much of the communication time.

Often, this optimisation takes the form of *pipelining*. Modern compilers often pipeline serial loops, by arranging to load data for a subsequent loop iteration while computations for the current iteration are in progress. This technique can overlap memory access time with computation, thus hiding the delays associated with memory access. CAF allows this same technique to be used for remote memory operations.

On the Cray X1E, accessing data from a remote CPU's memory can be accomplished with a standard vector memory reference. This allows the latency-hiding benefit of vector memory operations to be used for inter-processor communication. On a non-vector system with a global memory system, similar pipelining techniques could be applied.

5. Code examples

Two examples, drawn from EPIC, will be used to show how CAF was used to improve the scaling of the contact algorithm.

Consider this MPI call:

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

This call is used to 'gather' the first element of 'NUMALTERED' from each CPU into the array 'ITMP' on each CPU, in rank order. After this call, each CPU's copy of 'ITMP' will contain the values of 'NUMALTERED(1)' from all the CPUs, including itself. This is a common operation in parallel applications. Because each transfer of data between CPUs is so small, the time spent on this type of operation is dominated by function call overhead and communication latency.

The equivalent CAF code follows:

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

This simple loop vectorizes easily, which hides the latency of all but the first data transfer to each CPU. Also, there is no function call overhead to reduce performance.

For a variety of cases, the CAF code was at least 8 times faster than the MPI code.

The second example is a bit more complicated. In this case, three arrays, 'X', Y, and 'Z' are to be updated in a computational loop, using the arrays ‘PX’, ‘PY’, and ‘PZ’. Before the computation can be carried out, the current values of ‘PX’, ‘PY’, ‘PZ’, and ‘UPDATE’ must be obtained from all other CPUs. Below is the original MPI code:

```
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
```

There are several performance issues in this code. The reductions performed on arrays ‘PX’, ‘PY’, and ‘PZ’ result in unnecessarily communicating and summing many elements that won’t be used in the computations. Also, all four ‘MPI_REDUCE’ calls must complete before the following computations can begin. It would be preferable to communicate and sum only the needed elements of the arrays, and to pipeline the computations with the communication, thus ‘hiding‘ the communication time.

In the CAF version of the code, we are able to achieve both of these goals. First, each CPU executes a loop that determines which elements of ‘PX’, ‘PY’, and ‘PZ’ are needed, and packs those elements into temporary arrays ‘PXTEMP’, ‘PYTEMP’, and ‘PZTEMP’, which are co-arrays. The total number of elements packed in the arrays is stored in the co-array ‘ICOUNT’, and the original indices are stored in the co-array ‘ICTEMP’.

Then the computation loop is modified to access the necessary elements of ‘PX’, ‘PY’, and ‘PZ’, directly from the remote CPUs, using the appropriate co-indices. An additional loop is added outside the original loop, to access the contributions from each CPU.

In this code, only the elements of ‘PX’, ‘PY’, and ‘PZ’ that are needed are communicated between CPUs and summed. Also, the computation loop is completely vectorized, so that the computations begin as soon as the first element of data arrives from the remote CPU. This eliminates the need to wait for all of the data to be communicated before the computations begin.

The CAF code is below:

```
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
```

The CAF code runs at least 5 times as fast as the original code for all the cases tested.

6. Overall performance

CAF code has replaced MPI code in many parts of the contact algorithm, two of which were described in the preceding examples. The situations varied considerably, but generally the performance improvements to date are due to either a simple reduction in the latency of a communication operation, or significant overlapping of communication with computation through pipelining.

The improvement in performance is shown in Figure 3. This plot shows only the scaling of the contact algorithm itself. The scaling of the contact algorithm tends to be considerably worse than the scaling of the other portions of EPIC. Generally, the more time a problems spends in the contact algorithm, the worse it scales. The CAF version of the contact algorithm continues to benefit from the use of additional CPUs when run with 64 CPUs, long after the original version has levelled off. In practice, the scaling of contact-dominated EPIC jobs generally dictates that they be run with 16 CPUs or fewer, in order to make good use of resources.

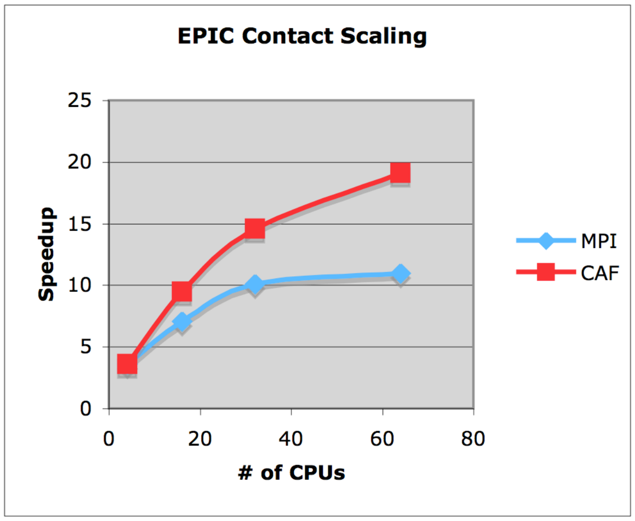

Figure 3. This plot shows the scaling of the original, MPI, version of the contact algorithm and the CAF version. The CAF version continues to benefit from additional CPUs after the performance of the original version has levelled off.

7. Observations

The performance of the CAF version of the EPIC contact algorithm demonstrates that significant performance improvements are achievable, compared to MPI. The performance of latency-dominated communication operations involving small messages can be dramatically improved. Similar improvements are possible when inter-processor communication can be overlapped with computation by accessing the remote data via vector loads.

While no testing was carried out with any other CAF implementations, it is clear that the performance benefits observed cannot be attributed to the use of CAF syntax per se. The underlying implementation is critical. The author has come to the view that CAF is a uniquely appropriate parallel programming model for taking advantage of advanced, high-performance hardware features such as global memory systems and pipeline-able remote memory instructions.

The ability to effectively pipeline remote memory operations seems to be a fundamentally enabling feature. All modern CPU architectures employ some form of pipelining to hide the latency of local memory operations. Since inter-processor latency is generally much higher than local latency, it seems that the ability to pipeline remote operations should be viewed as a key feature of advanced parallel systems.

In the course of optimising the contact algorithm, CAF code was used to replace MPI code only in specific places. The vast majority of the MPI code was left in place. Since many early users of CAF will likely be in this situation, it is important to note that CAF is quite compatible with MPI, although this might depend on the implementation. Since references to co-arrays without the co-indices simply refer to the local copy of the array, most of the code can remain unmodified.

About the Author

Jef Dawson is a Performance Analysis Specialist at the AHPCRC, employed by Network Computing Services, Inc. He optimises software for users of AHPCRC systems, carries out applied research into emerging HPC programming technologies, and teaches classes in CAF, Unified Parallel C, and MPI. He can be reached at 1200 Washington Avenue South, Minneapolis, MN 55415, USA. Email: jdawson@ahpcrc.org