AŞKIN ODABAŞI

M.S. Thesis

2011

PARALLEL PROGRAMMING TECHNIQUES BY USING CO-ARRAY FORTRAN

AŞKIN ODABAŞI

KADIR HAS UNIVERSITY

2011

PARALLEL PROGRAMMING TECHNIQUES BY USING CO-ARRAY FORTRAN

AŞKIN ODABAŞI

M.S., Computer Engineering, Kadir Has University, 2011

B.S., Computer Engineering, Sakarya University, 2003

Submitted to the Graduate School of Kadir Has University

In partial fulfillment of the requirements for the degree of

Master of Science

In

Computer Engineering

KADIR HAS UNIVERSITY

2011

KADIR HAS UNIVERSITY

COMPUTER ENGINEERING

PARALLEL PROGRAMMING TECHNIQUES BY USING CO-ARRAY FORTRAN

AŞKIN ODABAŞI

APPROVED BY:

_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

APPROVAL DATE:

PARALLEL PROGRAMMING TECHNIQUES BY USING CO-ARRAY FORTRAN

Abstract

Co-array Fortran (CAF) is a small set of extensions to Fortran 90. And also CAF is an emerging model for scalable, global address space paralel programming. CAF's global address space programming model simplifies the development of SPMD paralel programs by shifting the burden for managing the details of communication from developers to compilers.

In this study I introduce CAF's Programming Model, provide it's technical specifications, explain CAF's memory model and PGAS (Partitioned Global Address Space) , make comparsion between two SPMD language CAF and OpenMP.

In case, I select Matrix Multiplication as a problem and wrote Co Array Fortran code fort his problem. I ran it on Amazon EC2 Cluster with 16 CPU and CentOS operating system. Finally I showed the performance numbers fort his work.

Key words : Co-Array, Fortran, PGAS, SPDM, OpenMP

CO-ARRAY FORTRAN İLE PARALEL PROGRAMLAMA TEKNİKLERİ

Özet

Co-array Fortran (CAF) Fortran 90 uzantılarının küçük bir kümesidir. Ve aynı zamanda CAF, ölçeklenebilir, global adres alanlı paralel programlama için ortaya çıkan bir modeldir. CAF'ın global adres alanlı proramlama modeli compilerlarla geliştiricilerin iletişim detaylarını yönetmek için yükü kaydırarak SPDM paralel programların geliştirilmesini basitleştirir.

Bu çalışmada CAF'ın programlama modeli tanıtılmış, teknik spesifikasyonları sunulmuş, CAF'ın hafıza modeli ve PGAS (Partitioned Global Address Space) açıklanarak, iki farklı SPMD dili olan CAF ve OpenMP arasında karşılatırma yapılmıştır.

Örnek çalışmada, Co Array Fortran'da matrix çarpımı ele alındı ve yazılan program, Amzaon EC2 Cluster 16 CPU platfornunda CentOS işletim sistemi üzerinde çalıtırılarak performans değerleri elde edildi.

Anahtar Kelimeler: Co-Array, Fortran, PGAS, SPDM, OpenMP

Acknowledgements

This thesis was completed at the Faculty Engineering of the Kadir Has University in Istanbul, Turkey. In this project I received support from some special people.

I am greatly appreciative to my advisor Assistant Prof. Dr. Zeki Bozkuş for his guidance and support throughout my study.

Especially, I am grateful to my wife for all her support and impulsion in tihs project. Also I would like to thank my daughter Mihrişah Odabaşı giving me the happiness.

This thesis is dedicated to:

My Patient Wife

My Sweet Daughter

To giving meaning to my life…

Table of contents

List of Figures

List of Abbreviations

CAF	Co-array Fortran
ISO	International Organization for Standardization
PGAS	Partitioned Global Address Space
SPMD	Single Program Multiple Data
MPI	Message Passing Interface
I/O	Input / Output
RMA	Remote Memory Access
UPC	Unified Paralel C
HPF	High Performance Fortran
SMP	Symmetric Multiprocessor
MPP	Massively Parallel Processor
DSM	Distributed Shared Memory
NUMA	Non-Uniform Memory Access
API	Application Programming Interface
NLOM	NRL Layered Ocean Model
SHMEM	Shared memory

Chapter 1 Introduction

Co-array Fortran (CAF), formerly known as F--, is an extension of Fortran 95/2003 for parallel processing created by Robert Numrich and John Reid in 1990s. The Fortran 2008 standard (ISO/IEC 1539-1:2010) now includes coarrays (spelt without hyphen), as decided at the May 2005 meeting of the ISO Fortran Committee; the syntax in the Fortran 2008 standard is slightly different from the original CAF proposal.

A Co-array Fortran program is interpreted as if it were replicated a number of times and all copies were executed asynchronously. Each copy has its own set of data objects and is termed an image. The array syntax of Fortran is extended with additional trailing subscripts in square brackets to provide a concise representation of references to data that is spread across images.

The Co-array Fortran extension has been available for a long time and was implemented in some Fortran compilers such as those from Cray (since release 3.1). Since the inclusion of coarrays in the Fortran 2008 standard, the number of implementation is growing. The first open-source compiler which implemented coarrays as specified in the Fortran 2008 standard for Linux architectures is G95.

A group at Rice University is pursuing an alternate vision of coarray extensions for the Fortran language. Their perspective is that the Fortran 2008 standards committee's design choices were shaped more by the desire to introduce as few modifications to the language as possible than to assemble the best set of extensions to support parallel programming. They don't believe that the set of extensions agreed upon by the committee are the right ones. In their view, both Numrich and Reid's original design and the coarray extensions proposed for Fortran 2008, suffer from the following shortcomings:

- There is no support for processor subsets; for instance, coarrays must be allocated over all images.
- Coarrays must be declared as global variables; one cannot dynamically allocate a coarray into a locally scoped variable.
- The co-array extensions lack any notion of global pointers, which are essential for creating and manipulating any kind of linked data structure.
- Reliance on named critical sections for mutual exclusion hinders scalable parallelism by associating mutual exclusion with code regions rather than data objects.
- Fortran 2008's sync images statement doesn't provide a safe synchronization space. As a result, synchronization operations in user's code that are pending when a library call is made can interfere with synchronization in the library call.
- There are no mechanisms to avoid or tolerate latency when manipulating data on remote images.
- There is no support for collective communication.

To address these shortcomings, Rice University is developing a clean-slate redesign of the Co-array Fortran programming model. Rice's new design for Co-array Fortran, which they call Co-array Fortran 2.0, is an expressive set of coarray-based extensions to Fortran designed to provide a productive parallel programming model. Compared to the emerging Fortran 2008, Rice's new coarray-based language extensions include some additional features:

- Process subsets known as teams, which support coarrays, collective communication, and relative indexing of process images for pair-wise operations,
- Topologies, which augment teams with a logical communication structure,
- Dynamic allocation/deallocation of coarrays and other shared data,
- Local variables within subroutines: declaration and allocation of coarrays within the scope of a procedure is critical for library based-code,
- Team-based coarray allocation and deallocation,
- Global pointers in support of dynamic data structures, and

- Enhanced support for synchronization for fine control over program execution,
- Safe and scalable support for mutual exclusion, including locks and lock sets; and
- Events, which provide a safe space for point-to-point synchronization.

This study is efort on paralel programming with Co-Array Fortran (CAF). In next chapter, I give a brief overview of Co-Array Fortran, It's syntax and semantic. In chapter 3, I explain Co-Array Fortran Programming Model and CAF's Memory Model. Also chapter 3 includes Partitioned Global Address Space (PGAS). Chapter 4 contains a complete technical specifications. Chapter 5 includes comparison of two PGAS languages CAF and OpenMP. And last chapter is all about case study.

Chapter 2 A Brief Overview of Co-Array Fortran

Co-Array Fortran, formally called F--, is a small set of extensions to Fortran 95 for Single Program Multiple Data, SPMD, parallel processing.

Co-Array Fortran is a simple syntactic extension to Fortran 95 that converts it into a robust, efficient parallel language. It looks and feels like Fortran and requires Fortran programmers to learn only a few new rules. The few new rules are related to two fundamental issues that any parallel programming model must resolve, work distribution and data distribution.

First, consider work distribution. A single program is replicated a fixed number of times, each replication having its own set of data objects. Each replication of the program is called an image. Each image executes asynchronously and the normal rules of Fortran apply, so the execution path may differ from image to image. The programmer determines the actual path for the image with the help of a unique image index, by using normal Fortran control constructs and by explicit synchronizations. For code between synchronizations, the compiler is free to use all its normal optimation techniques, as if only one image is present.

Second, consider data distribution. The co-array extension to the language allows the programmer to express data distribution by specifying the relationship among memory images in a syntax very much like normal Fortran array syntax. One new object, the co-array, is added to the language. For example,

```
REAL, DIMENSION (N) [*] :: X,Y
X(:) = Y(:) [Q]
```

declares that each image has two real arrays of size N. If Q has the same value on each image, the effect of the assignment statement is that each image copies the array Y from image Q and makes a local copy in array X.

Array indices in parentheses follow the normal Fortran rules within one memory image. Array indices in square brackets provide an equally convenient notation for accessing objects across images and follow similar rules. Bounds in square brackets in co-array declarations follow the rules of assumed-size arrays since co-arrays are always spread over all the images. The programmer uses co-array syntax only where it is needed. A reference to a co-array with no square brackets attached to it is a reference to the object in the local memory of the executing image. Since most references to data objects in a parallel code should be to the local part, co-array syntax should appear only in isolated parts of the code. If not, the syntax acts as a visual flag to the programmer that too much communication among images may be taking place. It also acts as a flag to the compiler to generate code that avoids latency whenever possible.

Fortran 90 array syntax, extended to co-arrays, provides a very powerful and concise way of expressing remote memory operations. Here are some simple examples:

```
X       = Y[PE] ! get from Y[PE]

Y[PE]   = X     ! put into Y[PE]
Y[:]    = X     ! broadcast X
Y[LIST] = X     ! broadcast X over subset of PE's in array LIST
Z(:)    = Y[:]  ! collect all Y
S=MINVAL(Y[:])  ! min (reduce) all Y
B(1:M)[1:N]=S   ! S scalar,promoted to array of shape (1:M,1:N)
```

Input/output has been a problem with previous SPMD programming models, such as MPI, because standard Fortran I/O assumes dedicated single-process access to an open file and this constraint is often violated when it is assumed that I/O from each image is completely independent. Co-Array Fortran includes only minor extensions to Fortran 95 I/O, but all the inconsistencies of earlier programming models have been avoided and there is explicit support for parallel I/O. In addition I/O is compatible with both process-based and thread-based implementations.

The only other additions to Fortran 95 are several intrinsics. For example: the integer function NUM_IMAGES() returns the number of images, the integer function THIS_IMAGE() returns this image's index between 1 and NUM_IMAGES(), and the subroutine SYNC_ALL() is a global barrier which requires all operations before the

call on all images to be completed before any image advances beyond the call. In practice it is often sufficient, and faster, to only wait for the relevant images to arrive. SYNC_ALL(WAIT=LIST) provides this functionality.

There is also SYNC_TEAM(TEAM=TEAM) and SYNC_TEAM(TEAM=TEAM, WAIT=LIST) for cases where only a subset, TEAM, of all images are involved in the synchronization. The intrinsics START_CRITICAL and END_CRITICAL provide a basic critical region capability. It is also possible to write your own synchronization routines, using the basic intrinsic SYNC_MEMORY. This routine forces the local image to both complete any outstanding co-array writes into ``global'' memory and refresh from global memory any local copies of co-array data it might be holding (in registers for example). A call to SYNC_MEMORY is rarely required in Co-Array Fortran, because there is an implicit call to this routine before and after virtually all procedure calls including Co-Array's built in image synchronization intrinsics. This allows the programmer to assume that image synchronization implies co-array synchronization.

Image and co-array synchronization is at the heart of the typical Co-Array Fortran program. For example, here is how to exchange an array with your north and south neighbors:

```
      COMMON/XCTILB4/ B(N,4)[*]
      SAVE  /XCTILB4/
C
      CALL SYNC_ALL( WAIT=(/IMG_S,IMG_N/) )
      B(:,3) = B(:,1)[IMG_S]
      B(:,4) = B(:,2)[IMG_N]
      CALL SYNC_ALL( WAIT=(/IMG_S,IMG_N/) )
```

The first SYNC_ALL waits until the remote B(:,1:2) is ready to be copied, and the second waits until it is safe to overwrite the local B(:,1:2). Only nearest neighbors are involved in the sync. It is always safe to replace SYNC_ALL(WAIT=LIST) calls with global SYNC_ALL() calls, but this will often be significantly slower. In some cases, either the preceeding or succeeding synchronization can be avoided. Communication load balancing can sometimes be important, but the majority of remote co-array access optimization consists of minimizing the frequency of synchronization and having synchronization cover the minimum number of images. If the program is likely to run on machines without global memory hardware, then

array syntax (rather than DO loops) should always be used to express remote memory operations and copying co-array's into local temporary buffers well before they are required might be appropriate (although the compiler may do this for you).

In data parallel programs, each image is either performing the same operation or is idle. For example here is a data parallel fixed order cumulative sum:

```
REAL SUM[*]
CALL SYNC_ALL( WAIT=1 )
DO IMG= 2,NUM_IMAGES()
   IF (IMG==THIS_IMAGE()) THEN
      SUM = SUM + SUM[IMG-1]
   ENDIF
   CALL SYNC_ALL( WAIT=IMG )
ENDDO
```

Having each SYNC_ALL wait on just the active image improves performance, but there are still NUM_IMAGES() global sync's. In this case a better alternative is probably to minimize synchronization by avoiding the data parallel overhead entirely:

```
REAL SUM[*]
   ME = THIS_IMAGE()
   IF (ME.GT.1) THEN
     CALL SYNC_TEAM( TEAM=(/ME-1,ME/) )
     SUM = SUM + SUM[ME-1]
   ENDIF
   IF (ME.LT.NUM_IMAGES()) THEN
    CALL SYNC_TEAM( TEAM=(/ME,ME+1/) )
   ENDIF
```

Now each image is involved in at most two sync's, and only with the images just before and just after it in image order. Note that the first SYNC_TEAM call on one image is matched by the second SYNC_TEAM call on the previous image. This illustrates the power of the Co-Array Fortran synchronization intrinsics. They can improve the performance of data parallel algorithms, or provide implicit program execution control as an alternative to the data parallel approach.

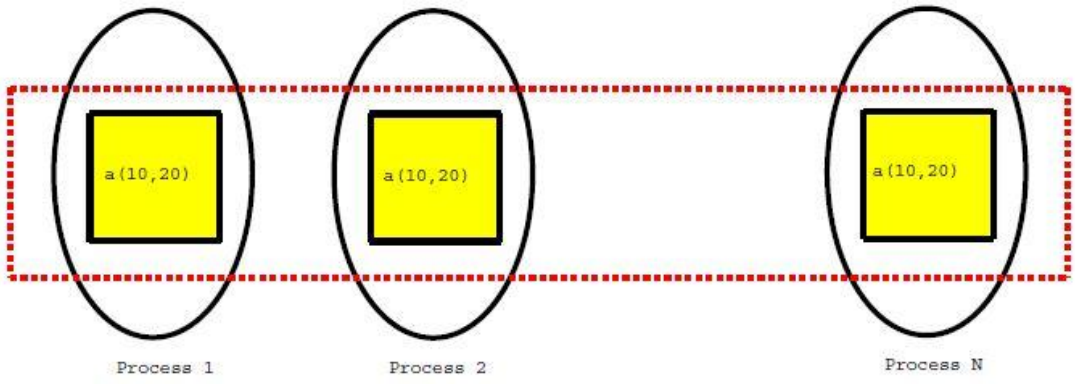
Several non-trivial Co-Array Fortran programs are included as examples with the caf2omp translator, and with the Cray T3E intrinsics.

Chapter 3 Co-Array Fortran Programming Model

Co-array Fortran supports SPMD parallel programming through a small set of language extensions to Fortran 95. An executing CAF program consists of a static collection of asynchronous process images. Similar to MPI, CAF programs explicitly distribute data and computation. However, CAF belongs to the family of Global Address Space Programming languages and provides the abstraction of globally accessible memory for both distributed and shared memory architectures [4].

CAF supports distributed data using a natural extension to Fortran 95 syntax. For example, the declaration presented and graphically represented in Figure 3.1 creates a shared co-array a with 10 × 20 integers local to each process image [5].

Figure 3.1: Graphical representation of co-array



Dimensions inside square brackets are called co-dimensions. Co-arrays may be declared for user-defined types as well as primitive types. A local section of a co-array may be a singleton instance of a type rather than an array of type instances. Co-arrays can be static objects, such as COMMON or SAVE variables, or can be declared as ALLOCATABLE variables and allocated and deallocated dynamically during program execution, using collective calls. Co-arrays of user-defined types may contain allocatable components, which can be allocated at runtime

independently by each process image. Finally, co-array objects can be passed as procedure arguments [4].

Instead of explicitly coding message exchanges to access data belonging to other processes, a CAF program can directly reference non-local values using an extension to the Fortran 95 syntax for subscripted references. For instance, process p can read the first column of co-array a from process p+1 referencing a(:,1)[p+1].

CAF has several synchronization primitives. sync all implements a synchronous barrier across all images; sync team is used for barrier-style synchronization among dynamically-formed teams of two or more processes; and sync memory implements a local memory fence and ensures the consistency of a process image's memory by completing all of the outstanding communication requests issued by this image.

Since both remote data access and synchronization are language primitives in CAF, communication and synchronization are amenable to compiler-based optimization. In contrast, communication in MPI programs is expressed in a more detailed form, which makes effective compiler transformations much more difficult [7].

3.1 PGAS

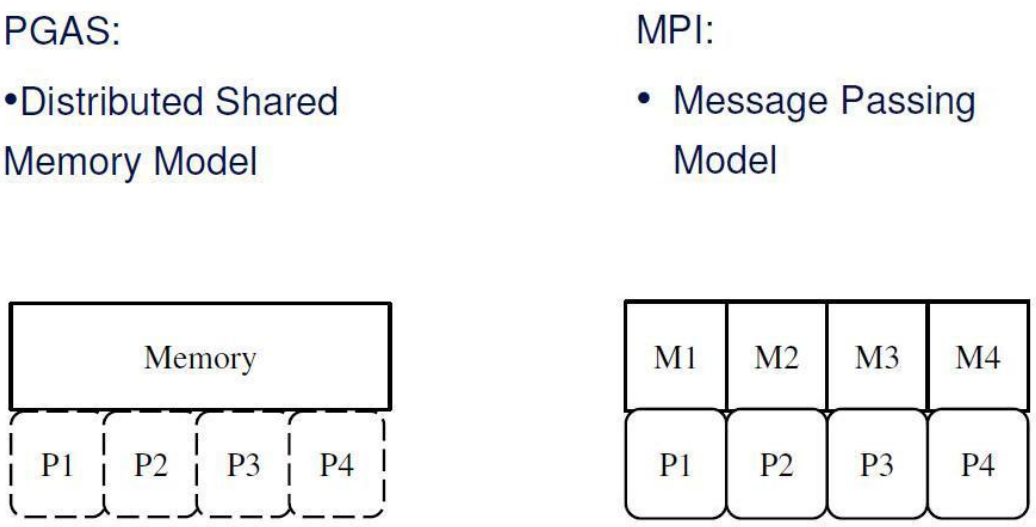
As has been discussed, it is currently popular for computers to have a 'hybrid' architecture where processing nodes are connected in a distributed memory architecture, but contain multiple cores which share memory. This trend is reflected in software through the increasing popularity of Partitioned Global Address Space (PGAS) languages.

These languages combine features of both message passing languages, as used with distributed memory architectures, and shared memory languages.

As the name suggests, PGAS languages feature a global address space that is partitioned logically between processors. As a result each processor has its own local portion of the memory space, similar to a Distributed Memory paradigm as implemented in MPI programs.

However unlike MPI programs processes need not communicate via messages; they can directly access each other's data via the global address space. A schematic illustration of the difference between the PGAS paradigm and a Distributed Memory paradigm is shown in Figure 3.2.

Figure 3.2: The PGAS paradigm and the Distributed Memory paradigm



A key difference between the distributed memory paradigm and the PGAS paradigm is in the communication between processors. The global address space of the PGAS paradigm allows single-sided communications. This means that the target processor, from which data is being read or to which data is being written, does not need to be interrupted during the communication.

The PGAS and shared memory paradigms both share the feature of simple data referencing between processors as they both use a global address space. The partitioning of the global address space is what distinguishes the PGAS paradigm from the shared memory paradigm and allows for better scaling on distributed memory machines [6].

Thus it can be seen that the global address space of the PGAS paradigm allows for positive features of both the shared memory paradigm and the distributed memory paradigm. Data accesses remain simple as in the shared memory paradigm, but

scaling on distributed memory machines is possible by making a distinction between accesses local and remote data.

However, one of the challenges of programming in any of the languages that implement the PGAS paradigm is that because the Remote Memory Access (RMA) calls are single sided there is no synchronisation implied by communications. This means that synchronisation must be explicitly declared by the programmer. Care must be taken with synchronisation to ensure that difficult to debug errors such as race conditions are avoided.

Another constraint imposed by the PGAS paradigm is that data structures that are shared between threads must have the same size on each thread. This ensures that the location of data on a thread is known by another thread when it tries to access that data remotely.

PGAS programming languages can get around this using pointers or derived data types.

In a derived data type data size can be changed internally, hiding differently sized data from the compiler [6].

3.1.1 Why PGAS?

The PGAS is the best of both worlds. This parallel programming model combined the performance and data locality (partitioning) features of distributed memory with the programmability and data referencing simplicity of a shared-memory (global address space) model. The PGAS programming model aims to achieve these characteristics by providing:

1. A local-view programming style (which differentiates between local and remote data partitions).
2. A global address space (which is directly accessible by any process).
3. Compiler-introduced communication to resolve remote references.
4. One-sided communication for improved inter-process performance.
5. Support for distributed data structures.

In this model variables and arrays can be either shared or local. Each process has private memory for local data items and shared memory for globally shared data values. While the shared-memory is partitioned among the cooperating processes (each process will contribute memory to the shared global memory), a process can directly access any data item within the global address space with a single address. Languages of PGAS Currently there are three PGAS programming languages that are becoming commonplace on modern computing systems:

1. Unified Parallel C (UPC)
2. Co-Array Fortran (CAF)
3. Titanium

3.2 Memory Models

There are 2 models for memory usage:

- Shared Memory Model.
- Distributed Memory Model

3.2.1 Shared Memory Model

The shared-memory programming model typically exploits a shared memory system, where any memory location is directly accessible by any of the computing processes (i.e. there is a single global address space). This programming model is similar in some respects to the sequential single-processor programming model with the addition of new constructs for synchronizing multiple access to shared variables and memory locations[8].

3.2.2 Distributed Memory Model

The distributed-memory programming model exploits a distributed-memory system where each processor maintains its own local memory and has no direct knowledge about another processor's memory (a "share nothing" approach). For data to be shared, it must be passed from one processor to another as a message.

3.3 CAF Memory Model

The CAF is a simple extension to Fortran 90 that allows programmers to write efficient parallel applications using a Fortran-like syntax. It also assumes the SPMD programming model with replicated data objects called co-arrays. Co-array objects are visible to all processors and each processor can read and write data belonging to any other processor by setting the index of the co-dimension to the appropriate value. The CAF creates multiple images of the same program where text and data are replicated in each image. it marks some variables with co-dimensions that behave like normal dimensions and express a logical problem decomposition. It also allows one sided data exchange between co-arrays using a Fortran like syntax [9].

On the other hand, CAF requires the underlying run-time system to map the logical problem decomposition onto specific hardware.

CAF Syntax: The CAF syntax is a simple parallel extension to normal Fortran syntax, where it uses normal rounded brackets () to point data in local memory, and square brackets [] to point data in remote memory [10]**.**

CAF Execution Model: The number of images is fixed and each image has its own index, retrievable at run-time. Each image executes the same program independently of the others and works on its own local data. An image moves remote data to local data through explicit CAF syntax while an "object" has the same name in each image. The programmer inserts explicit synchronization and branching as needed [10]**.**

CAF Memory Model: There are 4 memory models: [11, 12, 13]

1. One to one model
2. Many to one model
3. One to many model
4. Many to many model

Figure 3.3: One to one memory model

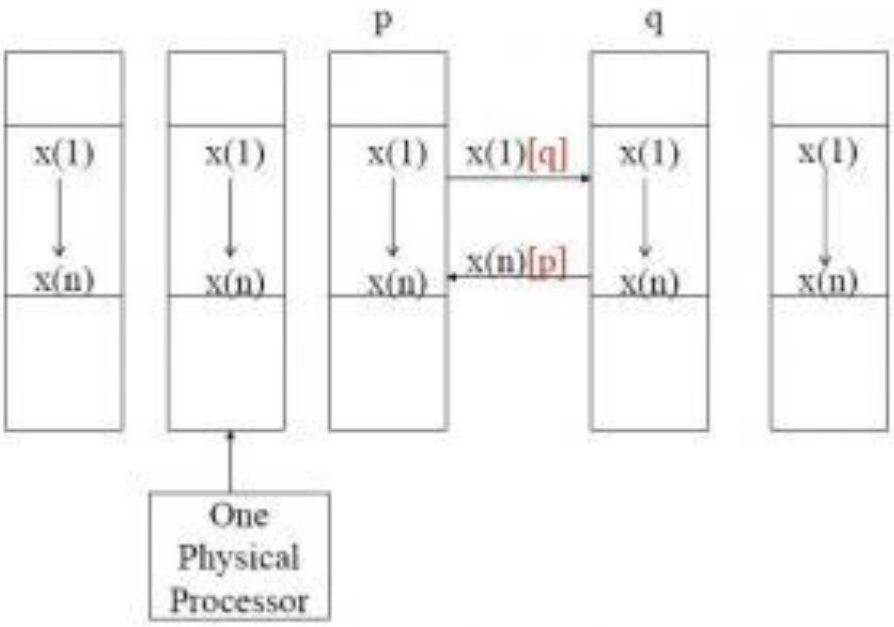


Figure 3.4: Many to one memory model

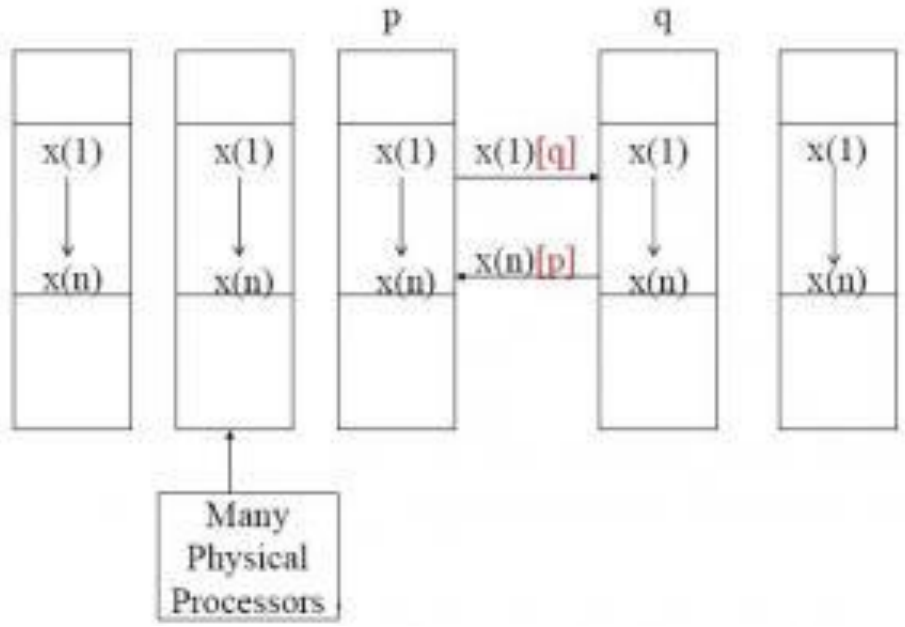


Figure 3.5: One to many memory model

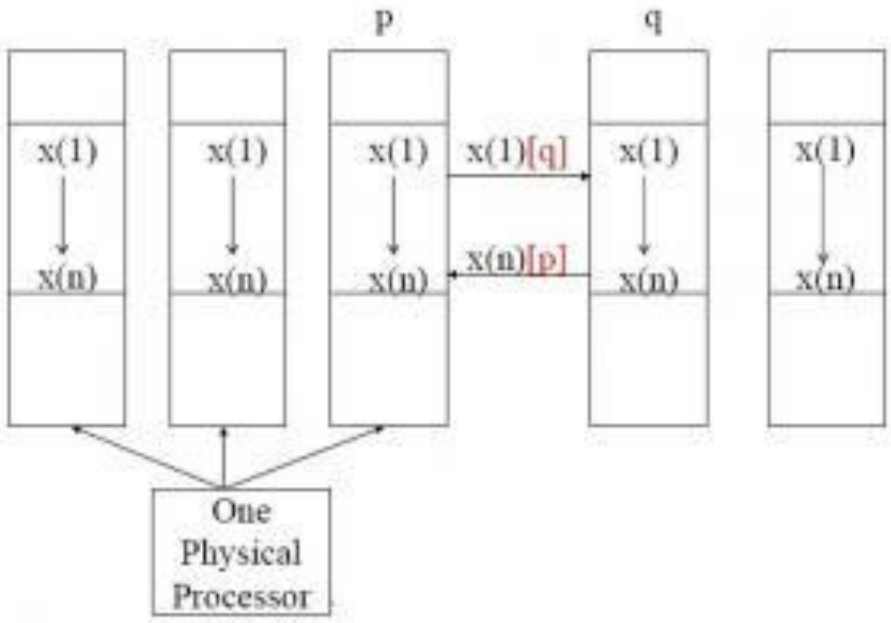
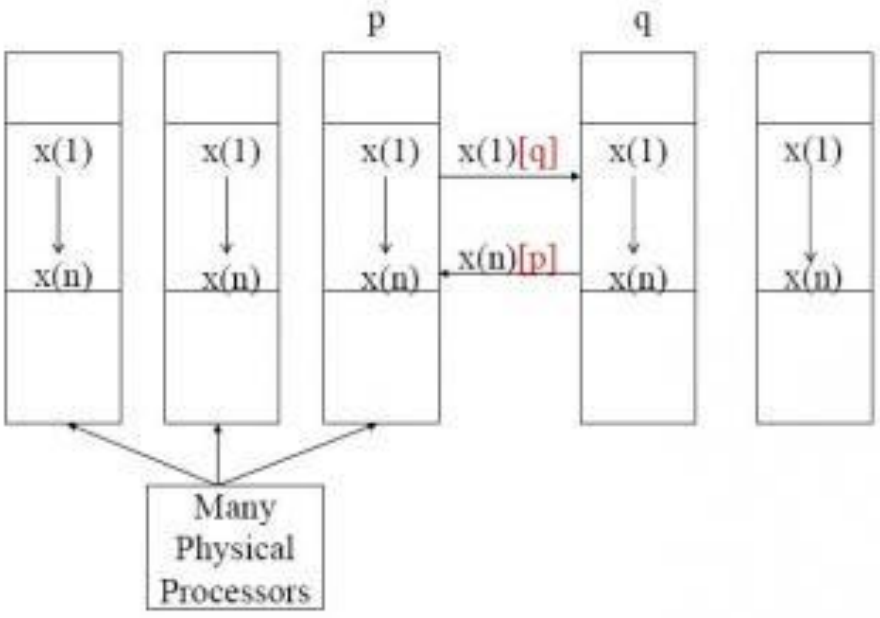


Figure 3.6: Many to many memory model



Chapter 4 Technical Specification

4.1 Program Images

A Co-Array Fortran program executes as if it were replicated a number of times, the number of replications remaining fixed during execution of the program. Each copy is called an image and each image executes asynchronously. A particular implementation of Co-Array Fortran may permit the number of images to be chosen at compile time, at link time, or at execute time. The number of images may be the same as the number of physical processors, or it may be more, or it may be less. The programmer may retrieve the number of images at run time by invoking the intrinsic function num_images(). Images are indexed starting from one and the programmer may retrieve the index of the invoking image through the intrinsic function this_image(). The programmer controls the execution sequence in each image through explicit use of Fortran 95 control constructs and through explicit use of intrinsic synchronization procedures.

4.2 Specifying Data Objects

Each image has its own set of data objects, all of which may be accessed in the normal Fortran way. Some objects are declared with co-dimensions in square brackets immediately following dimensions in parentheses or in place of them, for example:

```
REAL, DIMENSION(20)[20,*]  :: A
REAL  :: C[*], D[*]
CHARACTER :: B(20)[20,0:*]
INTEGER :: IB(10)[*]
TYPE(INTERVAL) :: S
DIMENSION :: S[20,*]
```

Unless the array is allocatable (Chapter 4.6), the form for the dimensions in square brackets is the same as that for the dimensions in parentheses for an assumed-size

array. The set of objects on all the images is itself an array, called a co-array, which can be addressed with array syntax using subscripts in square brackets following any subscripts in parentheses (round brackets), for example:

```
A(5)[3,7] = IB(5)[3]
D[3] = C
A(:)[2,3] = C[1]
```

We call any object whose designator includes square brackets a co-array subobject; it may be a co-array element, a co-array section, or a co-array structure component. The subscripts in square brackets are mapped to images in the same way as Fortran array subscripts in parentheses are mapped to memory locations in a Fortran 95 program. The subscripts within an array that correspond to data for the current image are available from the intrinsic this_image with the co-array name as its argument.

The rank, extents, size, and shape of a co-array or co-array subobject are given as for Fortran 95 except that we include both the data in parentheses and the data in square brackets. The local rank, local extents, local size, and local shape are given by ignoring the data in square brackets. The co-rank, co-extents, co-size, and co-shape are given from the data in square brackets. For example, given the co-array declared thus

```
REAL, DIMENSION(10,20)[20,5,*]  :: A
```

a(:,:)[:,:,1:15] has rank 5, local rank 2, co-rank 3, shape (/10,20,20,5,15/), local shape (/10,20/), and co-shape (/20,5,15/).

The co-size of a co-array is always equal to the number of images. If the co-rank is one, the co-array has a co-extent equal to the number of images and it has co-shape (/num_images()/). If the co-rank is greater than one, the co-array has no final extent, no final upper bound, and no co-shape (and hence no shape).

The local rank and the co-rank are each limited to seven. The syntax automatically ensures that these are the same on all images. The rank of a co-array subobject (sum of local rank and co-rank) must not exceed seven.

For a co-array subobject, square brackets may never precede parentheses.

A co-array must have the same bounds (and hence the same extents) on all images.

For example, the subroutine

```
SUBROUTINE SOLVE(N,A,B)
INTEGER :: N
REAL :: A(N)[*], B(N)
```

must not be called on one image with n having the value 1000 and on another with n having the value 1001.

A co-array may be allocatable:

```
SUBROUTINE SOLVE(N,A,B)
INTEGER :: N
REAL :: A(N)[*], B(N)
REAL,ALLOCATABLE :: WORK(:)[:]
```

Allocatable arrays are discussed in Chapter 4.6.

There is no mechanism for assumed-co-shape arrays. A co-array is not permitted to be a pointer. Automatic co-arrays are not permitted; for example, the co-array work in the above code fragment is not permitted to be declared thus

```
SUBROUTINE SOLVE(N,A,B)
INTEGER :: N
REAL :: A(N)[*], B(N)
REAL :: WORK(N)[*]  ! NOT PERMITTED
```

A co-array is not permitted to be a constant.

A DATA statement initializes only local data. Therefore, co-array subobjects are not permitted in DATA statements. For example:

```
REAL :: A(10)[*]
DATA A(1)    /0.0/ ! PERMITTED
DATA A(1)[2] /0.0/ ! NOT PERMITTED
```

Unless it is allocatable or a dummy argument, a co-array always has the SAVE attribute.

The image indices of a co-array always form a sequence, without any gaps, commencing at one. This is true for any lower bounds. For example, for the array declared as

```
REAL :: A(10,20)[20,0:5,*]
```

A(:,:)[1,0,1] refers to the rank-two array a(:,:) in image one.

Co-arrays may be of derived type but components of derived types are not permitted to be co-arrays.

4.3 Accessing Data Objects

Each object exists on every image, whether or not it is a co-array. In an expression, a reference without square brackets is always a reference to the object on the invoking image. For example, size(b) for co-array b declared as

```
CHARACTER :: B(20)[20,0:*]
```

returns its local size, which is 20.

The subscript order value of the co-subscript list must never exceed the number of images. For example, if there are 16 images and the the co-array a is declared thus

```
REAL :: A(10)[5,*]
```

a(:)[1,4] is valid since it has co-subscript order value 16, but a(:)[2,4] is invalid.

Two arrays conform if they have the same shape. Co-array subobjects may be used in intrinsic operations and assignments in the usual way, for example,

```
B(:,1:M) = A[:,1:M]*C(:)[1:M] ! ALL HAVE RANK TWO.
B(J,:) = A[:,K]               ! BOTH HAVE RANK ONE.
C[1:P:3] = D(1:P:3)[2]        ! BOTH HAVE RANK ONE.
```

Square brackets attached to objects in an expression or an assignment alert the reader to communication between images. Unless square brackets appear explicitly, all expressions and assignments refer to the invoking image. Communication may take place, however, within a procedure that is referenced, which might be a defined operation or assignment.

The rank of the result of an intrinsic operation is derived from the ranks of its operands by the usual rules, disregarding the distinction between local rank and co-

rank. The local rank of the result is equal to the rank. The co-rank is zero. Similarly, a parenthesized co-array subobject has co-rank zero.

For example 2.0*d(1:p:3)[2] and (d(1:p:3)[2]) each have rank 1, local rank 1, and co-rank 0.

4.4 Procedures

A co-array subobject is permitted only in intrinsic operations, intrinsic assignments, and input/output lists.

If a dummy argument has co-rank zero, the value of a co-array subobject may be passed by using parentheses to make an expression, for example,

```
C(1:P:2) = SIN( (D[1:P:2]) )
```

If a dummy argument has nonzero co-rank, the co-array properties are defined afresh and are completely independent of those of the actual argument. The interface must be explicit. The actual argument must be the name of a co-array or a subobject of a co-array without any square brackets, vector-valued subscripts, or pointer component selection; any subscript expressions must have the same value on all images. If the dummy argument has nonzero local rank and its local shape is not assumed, the actual argument shall not be an array section, involve component selection, be an assumed-shape array, or be a subobject of an assumed-shape array.

A function result is not permitted to be a co-array.

A pure or elemental procedure is not permitted to contain any Co-Array Fortran extensions.

The rules for resolving generic procedure references remain unchanged.

4.5 Sequence Association

COMMON and EQUIVALENCE statements are permitted for co-arrays and specify how the storage is arranged on each image (the same for every one). Therefore, co-array subobjects are not permitted in an EQUIVALENCE statement. For example

```
EQUIVALENCE (A[10],B[7]) ! NOT ALLOWED(COMPILE-TIME CONSTRAINT)
```

is not permitted. Appearing in a COMMON and EQUIVALENCE statement has no effect on whether an object is a co-array; it is a co-array only if declared with square brackets. An EQUIVALENCE statement is not permitted to associate a co-array with an object that is not a co-array. For example

```
INTEGER :: A,B[*]
EQUIVALENCE (A,B) ! NOT ALLOWED (COMPILE-TIME CONSTRAINT)
```

is not permitted. A COMMON block that contains a co-array always has the SAVE attribute. Which objects in the COMMON block are co-arrays may vary between scoping units. Since blank COMMON may vary in size between scoping units, co-arrays are not permitted in blank COMMON.

4.6 Allocatable Arrays

A co-array may be allocatable. The ALLOCATE statement is extended so that the co-extents can be specified, for example,

```
REAL, ALLOCATABLE :: A(:)[:], S[:,:]
:
ALLOCATE ( ARRAY(10)[*], S[34,*] )
```

The upper bound for the final co-dimension must always be given as an asterisk and values of all the other bounds are required to be the same on all images. For example, the following are not permitted

```
ALLOCATE(A(NUM_IMAGES()))     ! NOT ALLOWED (COMPILE-TIME CONSTRAINT)
ALLOCATE(A(THIS_IMAGE())[*]) ! NOT ALLOWED (RUN-TIME CONSTRAINT)
```

There is implicit synchronization of all images in association with each ALLOCATE statement that involves one or more co-arrays. Images do not commence executing subsequent statements until all images finish execution of an ALLOCATE statement for the same set of co-arrays. Similarly, for DEALLOCATE, all images delay

making the deallocations until they are all about to execute a DEALLOCATE statement for the same set of co-arrays.

An allocatable co-array without the SAVE attribute must not have the status of currently allocated if it goes out of scope when a procedure is exited by execution of a RETURN or END statement.

When an image executes an allocate statement, no communication is involved apart from any required for synchronization. The image allocates the local part and records how the corresponding parts on other images are to be addressed. The compiler, except perhaps in debug mode, is not required to enforce the rule that the bounds are the same on all images. Nor is the compiler responsible for detecting or resolving deadlock problems. For allocation of a co-array that is local to a recursive procedure, each image must descend to the same level of recursion or deadlock may occur.

4.7 Array Pointers

A co-array is not permitted to be a pointer.

A co-array may be of a derived type with pointer components. For example, if p is a pointer component, z[i]%p is a reference to the target of component p of z on image i. To avoid references with co-array syntax to data that is not in a co-array, we limit each pointer component of a co-array to the behaviour of an allocatable component of a co-array:

1. A pointer component of a co-array is not permitted on the left of a pointer assignment statement (compile-time constraint),
2. A pointer component of a co-array is not permitted as an actual argument that corresponds to a pointer dummy argument (compile-time constraint),
3. If an actual argument of a type with a pointer component is part of a co-array and is associated with a dummy argument that is not a co-array, the pointer association status of the pointer component must not be altered during execution of the procedure (this is not a compile-time constraint).

To avoid hidden references to co-arrays, the target in a pointer assignment statement is not permitted to be any part of a co-array. For example,

```
Q => Z[I]%P ! NOT ALLOWED (COMPILE-TIME CONSTRAINT)
```

is not permitted. Intrinsic assignments are not permitted for co-array subobjects of a derived type that has a pointer component, since they would involve a disallowed pointer assignment for the component:

```
Z[I] = Z ! NOT ALLOWED IF Z HAS A POINTER
Z = Z[I] ! COMPONENT (COMPILE-TIME CONSTRAINT)
```

Similarly, it is legal to allocate a co-array of a derived type that has pointer components, but it is illegal to allocate one of those pointer components on another image:

```
TYPE(SOMETHING), ALLOCATABLE ::  T[:]
...
ALLOCATE(T[*])           !  ALLOWED
ALLOCATE(T%PTR(N))       !  ALLOWED
ALLOCATE(T[Q]%PTR(N))    !  NOT ALLOWED (COMPILE-TIME CONSTRAINT)
```

4.8 Execution Control

Most of the time, each image executes on its own as a Fortran 95 program without regard to the execution of other images. It is the programmer's responsibility to ensure that whenever an image alters co-array data, no other image might still need the old value. Also, that whenever an image accesses co-array data, it is not an old value that needs to be updated by another image. The programmer uses invocations of the intrinsic synchronization procedures to do this, and the programmer should make no assumptions about the execution timing on different images. This obligation on the programmer provides the compiler with scope for optimization. When constructing code for execution on an image, it may assume that the image is the only image in execution until the next invocation of one of the intrinsic synchronization procedures and thus it may use all the optimization techniques available to a standard Fortran 95 compiler.

In particular, if the compiler employs temporary memory such as cache or registers (or even packets in transit between images) to hold co-array data, it must copy any such data it has defined to memory that can be accessed by another image to make it

visible to it. Also, if another image changes the co-array data, the executing image must recover the data from global memory to the temporary memory it is using. The intrinsic procedure sync_memory is provided for both purposes. It is concerned only with data held in temporary memory on the executing image for co-arrays in the local scope. Given this fundamental intrinsic procedure, the other synchronization procedures can be programmed in Co-Array Fortran, but the intrinsic versions, which we describe next, are likely to be more efficient. In addition, the programmer may use it to express customized synchronization operations in Co-Array Fortran.

If data calculated on one image are to be accessed on another, the first image must call sync_memory after the calculation is complete and the second must call sync_memory before accessing the data. Synchronization is needed to ensure that sync_memory is called on the first before sync_memory is called on the second.

The subroutine sync_team provides synchronization for a team of images. The subroutine sync_all (see Chapter 4.10) provides a shortened call for the important case where the team contains all the images. Each invocation of sync_team or sync_all has the effect of sync_memory. The subroutine sync_all is not discussed further in this section.

For each invocation of sync_team on one image of a team, there shall be a corresponding invocation of sync_team on every other image of the team. The n-th invocation for the team on one image corresponds to the n-th invocation for the team on each other image of the team, n=1,2,... . The team is specified in an obligatory argument team.

The subroutine also has an optional argument wait. If this argument is absent from a call on one image it must be absent from all the corresponding calls on other images of the team. If wait is absent, each image of the team waits for all the other images of the team to make corresponding calls. If wait is present, the image is required to wait only for the images specified in wait to make corresponding calls.

Teams are permitted to overlap, but the following rule is needed to avoid any possibility of deadlock. If a call for one team is made ahead of a call for another team

on a single image, the corresponding calls shall be in the same order on all images in common to the two teams.

The intrinsic sync_file plays a similar role for file data to that of sync_memory for co-array data. Because of the high overheads associated with file operations, sync_team does not have the effect ofsync_file. If data written by one image to a file is to be read by another image without closing the connection and re-opening it on the other image, calls of sync_file on both images are needed (details in Chapter 4.9).

To avoid the need for the programmer to place invocations of sync_memory around many procedure invocations, these are implicitly placed around any procedure invocation that might involve any reference to sync_memory. Formally, we define a caf procedure as

1. An external procedure;
2. A dummy procedure;
3. A module procedure that is not in the same module;
4. Sync_all, sync_team, sync_file, start_critical, end_critical; or
5. A procedure whose scoping unit contains an invocation of sync_memory or a caf procedure reference.

Invocations of sync_memory are implicitly placed around every caf procedure reference.

Exceptionally, it may be necessary to limit execution of a piece of code to one image at a time. Such code is called a critical section. We provide the subroutine start_critical to mark the commencement of a critical region and the subroutine end_critical to mark its completion. Both have the effect of sync_memory. Each image maintains an integer called its critical count. Initially, all these counts are zero. On entry to start_critical, the image waits for the system to give it permission to continue, which will only happen when all other images have zero critical counts. The image then increments its critical count by one and returns. Having these counts permits nesting of critical regions. On entry to end_critical, the image decrements its critical count by one and returns.

The effect of a STOP statement is to cause all images to cease execution. If a delay is required until other images have completed execution, a synchronization statement should be employed.

4.9 Input / Output

Most of the time, each image executes its own read and write statements without regard for the execution of other images. However, Fortran 95 input and output processing cannot be used from more than one image without restrictions unless the images reference distinct file systems. Co-Array Fortran assumes that all images reference the same file system, but it avoids the problems that this can cause by specifying a single set of I/O units shared by all images and by extending the file connection statements to identify which images have access to the unit.

It is possible for several images to be connected on the same unit for direct-access input/output. The intrinsic sync_file may be used to ensure that any changed records in buffers that the image is using are copied to the file itself or to a replication of the file that other images access. This intrinsic plays the same role for I/O buffers as the intrinsic sync_memory does for temporary copies of co-array data. Execution of sync_file also has the effect of requiring the reloading of I/O buffers in case the file has been altered by another image. Because of the overheads of I/O, sync_file applies to a single file.

It is possible for several images to to be connected on the same unit for sequential output. The processor shall ensure that while one image is transfering the data of a record to the file, no other image transfers data to the file. Thus, each record in an external file arises from a single image. The processor is permitted to hold the data in a buffer and transfer several whole records on execution of sync_file.

The I/O keyword TEAM is used to specify an integer rank-one array, connect_team, for the images that are associated with the given unit. All elements of connect_team shall have values between 1 andnum_images() and there shall be no repeated values. One element shall have the value this_image(). The default connect_team is (/this_image()/).

The keyword TEAM is a connection specifier for the OPEN statement. All images in connect_team, and no others, shall invoke OPEN with an identical connection-spec-list. There is an implied call tosync_team with the single argument connect_team before and after the OPEN statement. The OPEN statement connects the file on the invoking images only, and the unit becomes unavailable on all other images. If the OPEN statement is associated with a processor dependent file, the file is the same for all images in connect_team. If connect_team contains more than one image, the OPEN shall haveACCESS=DIRECT or ACTION=WRITE.

An OPEN on a unit already connected to a file must have the same connect_team as currently in effect.

A file shall not be connected to more than one unit, even if the connect_teams for the units have no images in common.

Pre-connected units that allow sequential read shall be accessible on the first image only. All other pre-connected units have a connect_team containing all the images.

CLOSE has a TEAM= specifier. If the unit exists and is connected on more than one image, the CLOSE statement must have the same connect_team as currently in effect. There is an implied call tosync_file for the unit before CLOSE. There are implied calls to sync_team with single argument connect_team before and after the implied sync_file and before and after the CLOSE.

BACKSPACE, REWIND, and ENDFILE have a TEAM= specifier. If the unit exists and is connected on at least one image, the file positioning statement must have the same connect_team as currently in effect. There is an implied call to sync_file for the unit before the file positioning statement. There are implied calls to sync_team with single argument connect_team before and after the impliedsync_file and before and after the file positioning statement.

4.10 Intrinsic Procedures

Co-Array Fortran adds the following intrinsic procedures. Only num_images, log2_images, and rem_images are permitted in specification expressions. None are permitted in initialization expressions. We use italic square brackets, *[ ]*, to indicate optional arguments.

end_critical() is a subroutine for limiting parallel execution. Each image holds an integer called its critical count. On entry, the count for the image shall be positive. The subroutine decrements this count by one. end_critical has the effect of sync_memory.

log2_images() returns the base-2 logarithm of the number of images, truncated to an integer. It is an inquiry function whose result is a scalar of type default integer.

num_images() returns the number of images. It is an inquiry function whose result is a scalar of type default integer.

rem_images() returns mod(num_images(),2**log2_images()). It is an inquiry function whose result is a scalar of type default integer.

start_critical() is a subroutine for limiting parallel execution. Each image holds an integer called its critical count. Initially all these counts are zero. The image waits for the system to give it permission to continue, which will only happen when all other images have zero critical counts. The image then increments its critical count by one and returns. start_critical has the effect ofsync_memory.

sync_all(*[*wait*]*) is a subroutine that synchronizes all images. sync_all() is treated as sync_team(all) and sync_all(wait) is treated as sync_team(all,wait), where all has the value(/ (I,I=1,num_images()) /).

sync_all(*[*wait*]*) has the effect of sync_memory.

sync_file(unit) is a subroutine for marking the progress of input-output on a unit. unit is an INTENT(IN) scalar argument of type integer and specifies the unit.

The subroutine affects only the data for the file connected to the unit. If the unit is not connected on this image or does not exist, the subroutine has no effect. Before return from the subroutine, any file records that are held by the image in temporary storage and for which WRITE statements have been executed since the previous call of sync_file on the image (or since execution of OPEN in the case of the first sync_file call) shall be placed in the file itself or a replication of the file that other images access. The first subsequent access by the image to file data in temporary storage shall be preceded by data recovery from the file itself or its replication. If the unit is connected for sequential access, the previous WRITE statement shall have been for advancing input/output.

sync_team(team *[*,wait*]*) is a subroutine that synchronizes images. team is an INTENT(IN) argument that is of type integer and is scalar or of rank one. The scalar case is treated as if the argument were the array (/this_image(),team/); in this case, team must not have the value this_image(). All elements of team shall have values in the range 1<=team(i)<=num_images() and there shall be no repeated values. One element of team shall have the value this_image(). wait is an optional INTENT(IN) argument that is of type integer and is scalar or of rank one. Each element, if any, of wait shall have a value equal to that of an element of team. The scalar case is treated as if the argument were the array (/wait/).

The argument team specifies a team of images that includes the invoking image. For each invocation of sync_team on one image, there shall be a corresponding invocation of sync_team for the same team on every other image of the team. The n-th invocation for the team on one image corresponds to the n-th invocation for the team on each other image of the team, n=1, 2, ... If a call for one team is made ahead of a call for another team on a single image, the corresponding calls shall be in the same order on all images in common to the two teams.

If wait is absent on one image it must be absent in all the corresponding calls on the other images of the team. In this case, wait is treated as if it were equal to team and

all images of the team wait until all other images of the team are executing corresponding calls. If wait is present, the image waits for all the images specifed by wait to execute corresponding calls.

sync_team(team*[*,wait*]*) has the effect of sync_memory.

sync_memory() is a subroutine for marking the progress of the execution sequence. Before return from the subroutine, any co-array data that is accessible in the scoping unit of the invocation and is held by the image in temporary storage and has been defined there shall be placed in the storage that other images access. The first subsequent access by the image to co-array data in this temporary storage shall be preceded by data recovery from the storage that other images access.

this_image(*[*array*[*,dim*]]*) returns the index of the invoking image, or the set of co-subscripts of array that denotes data on the invoking image. The type of the result is always default integer. There are four cases:

- Case (i). If array is absent, the result is a scalar with value equal to the index of the invoking image. It is in the range 1, 2, ..., num_images().
- Case (ii). If array is present with co-rank 1 and dim is absent, the result is a scalar with value equal to co-subscript of the element of array that resides on the invoking image.
- Case (iii). If array is present with co-rank greater than 1 and dim is absent, the result is an array of size equal to the co-rank of array. Element k of the result has value equal to co-subscript k of the element of array that resides on the invoking image.
- Case (iv). If array and dim are present, the result is a scalar with value equal to co-subscript dim of the element of array that resides on the invoking image.

Chapter 5 A Comparsion of Co-Array Fortran and OpenMP Fortran

5.1 OpenMP Fortran

OpenMP Fortran is a set of compiler directives that provide a high level interface to threads in Fortran, with both thread-local and thread-shared memory. Most compilers are now complient with version 1.1 of the specification [28], which will be discussed here unless otherwise noted. Version 2.0 [29] was released in November 2000 but is not yet widely available. OpenMP can also be used for loop-level directive based parallelization, but in SPMD-mode *N* threads are spawned as soon as the program starts and exist for the duration of the run. The threads act like Co-Array images (or MPI processes), with some memory private to a single thread and other memory shared by all threads. Variables in shared memory play the role of co-arrays in Co-Array Fortran, i.e. if two threads need to "communicate" they do so via variables in shared memory. Local non-saved variables are thread private, and all other variables are shared by default. The directive !$OMP THREADPRIVATE can make a named common private to each thread.

Threaded I/O is well understood in C [21], and many of the same issues arise with OpenMP Fortran I/O. A single process necessarily has one set of I/O files and pointers. This means that Fortran's single process model of I/O is appropriate. I/O is "thread safe" if multiple threads can be doing I/O (i.e., making calls to I/O library routines) at the same time. OpenMP Fortran requires thread safety for I/O to distinct unit numbers (and therefore to distinct files), but not to the same I/O unit number. A SPMD program that writes to the same file from several threads will have to put all such I/O operations in critical regions. It is therefore not possible in OpenMP to perform parallel I/O to a single file.

The integer function OMP_GET_NUM_THREADS() returns the number of threads, the integer function OMP_GET_THREAD_NUM() returns this thread's index

etween 0 and OMP_GET_NUM_THREADS()-1. The compiler directive !$OMP BARRIER is a global barrier which requires all operations before the barrier on all threads to be completed before any thread advances beyond the call. The directives !$OMP CRITICAL and !$OMP END CRITICAL provide a critical region capability, with more flexiblity than that in Co-Array Fortran, and in addition there are intrinsic routines for shared locks that can be used for the fine grain synchronization typical of threaded programs [21]. The directives !$OMP MASTER and !$OMP END MASTER provide a region that is executed by the master thread only, !$OMP SINGLE and !$OMP END SINGLE identify a region executed by a single thread. Note that all directive defined regions must start and end in the same lexical scope. It is possible to write your own synchronization routines, using the basic directive !$OMP FLUSH. This routine forces the thread to both complete any outstanding writes into memory and refresh from memory any local copies of data it might be holding (in registers for example). It only applies to "thread visible" variables in the local scope, and can optionally include a list of exactly which variables it should be applied to. BARRIER, CRITICAL, and END CRITICAL all imply FLUSH, but unlike Co-Array Fortran it is not automatically applied around subroutine calls. This means that the programmer has to be very careful about making assumptions that thread visible variables are current. Any user-written synchronization routine should be preceeded by a FLUSH directive every time it is called.

A subset of OpenMP's loop-level directives, that automate the allocation of loop iterations between threads, are also available to SPMD programs but are not typically used.

Unlike High Performance Fortran (HPF) [22], which has compiler directives that are carefully designed to not alter the meaning of the underlying program, the OpenMP directives used in SPMD-threaded programming are declaration attributes or executable statements. They are still properly expressible as structured comments, starting with the string "!$OMP", because they have no effect when the program has exactly one thread. But they are not "directives" in the conventional sense. For example "!$OMP BARRIER" does not allow any thread to continue until all have reached the statement. When there is more than one thread, SPMD OpenMP defines

a new language that is different from uni-processor Fortran in ways that are not obvious by inspection of the source code. For example:

1. Saved local variables are always shared and non-saved local variables are always threadprivate. It is all too easy to inadvertently create a saved variable. For example, in Fortran 90/95 initializing a local variable, e.g., INTEGER :: I=0, creates a saved variable. A DATA statement has a similar effect in both Fortran 77 and Fortran 90/95. In OpenMP such variables are always shared, but often the programmer's intent was to initialize a threadprivate variable (which is not possible with local variables in version 1.1).
2. In version 1.1, only common can be either private or shared under programmer control. Module variables, often used to replace common variables in Fortran 90/95, are always shared. Version 2.0 allows individual saved and module variables to be declared private.
3. ALLOCATE is required to be thread safe, but because only common variables can be both private and non-local, it is difficult to use ALLOCATABLE for private variables. A pointer in THREADPRIVATE common may work, but is not a safe alternative to an allocatable array.
4. It is up to the programmer to avoid race conditions caused by the compiler using copy-in/copy-out of thread-shared array section subroutine arguments.
5. There is no way to document the default case using compiler directives. There is a !$OMP THREADPRIVATE directive but no matching optional !$OMP THREADSHARED directive. Directives that imply a barrier have an option, NOWAIT, to skip the barrier but no option, WAIT, to document the default barrier.
6. Sequential reads from multiple threads must be in a critical region for thread safety and provide a different record to each thread. In all process-based SPMD models sequential reads from multiple processes provide the same record to each process.

SPMD OpenMP is not a large extension to Fortran but OpenMP programs cannot be maintained by Fortran programmers unfamiliar with OpenMP. For example, a programmer has to be aware that adding a DATA statement to a subroutine could

change the multi-thread behavior of that subroutine. In contrast, adding a DATA statement, or making any other modifications, to a Co-Array Fortran program is identical in effect to making the same change to a Fortran 90/95 program providing no co-arrays are involved (i.e., providing no square brackets are associated with the variable in the local scope).

Version 2.0 of the specification adds relatively few capabilities for SPMD programs, but the extension of THREADPRIVATE from named common blocks to saved and module variables will provide a significantly improved environment particularly for Fortran 90 programmers. It is unfortunate that there is still no way to document the default via a similar THREADSHARED directive. If this existed, the default status of variables would cease to be an issue because it could be confirmed or overridden with compiler directives. The lack of fully thread safe I/O places an unnecessary burden on the SPMD programmer. The standard should at least require that thread safe I/O be available as a compile time option. This is much easier for the compiler writer to provide, either as a thread-safe I/O library or by automatically inserting a critical region around every I/O statement, than the application programmer. The sequential read limitation is a basic property of threads, and is primarily an issue because many Fortran programmers are familiar with process-based SPMD APIs. Version 2.0 has a COPYPRIVATE directive qualifier that handles this situation cleanly. For example:

```
!$OMP SINGLE
       READ(11) A,B,C
!$OMP END SINGLE, COPYPRIVATE(A,B,C)
```

Here "A,B,C" are threadprivate variables that are read on one thread and then copied to all other threads by the COPYPRIVATE clause at the end of the single section. Co-Array Fortran I/O is designed to work with threads or processes, and a proposed extension can handle this case:

```
READ(11,TEAM=ALL) A,B,C
```

All images in the team perform the identical read and there is implied synchronization before and after the read. If images are implemented as threads, the I/O library could establish a separate file pointer for each thread and have each thread read the file independently or the read could be performed on one thread and the result copied to all others.

The limitations of OpenMP are more apparent for SPMD programs than for those using loop-level directives, which are probably the primary target of the language. SPMD programs are using orphan directives, outside the lexical scope of the parallel construct that created the threads [28]. OpenMP provides a richer set of directives within a single lexical scope, which allow a more complete documentation of the exact state of all variables. However, it is common to call a subroutine from within a do loop that has been parallelized and the variables in that subroutine have the same status as those in a SPMD subroutine. Also, almost all OpenMP compilers support Fortran 90 or 95, rather than Fortran 77, but version 1.1 directives largely ignore Fortran 95 constructs. Version 2.0 has more complete Fortran 95 support, which provides an incentive for compilers to be updated to version 2.0.

5.2 A simple example

The calculation of π was used as an example in the original OpenMP proposal [25], which presented three versions using OpenMP's loop level parallelization constructs, using MPI, and using pthreads. SPMD versions using Co-Array Fortran and OpenMP Fortran are presented here. First Co-Array Fortran:

```
program compute_pi
double precision :: mypi[*],pi,psum,x,w
integer          :: n[*],me,nimg,i
nimg = num_images()
me   = this_image()
if (me==1) then
  write(6,*) 'Enter number of intervals'; read(5,*) n
  write(6,*) 'number of intervals = ',n
  n[:] = n
endif
call sync_all(1)
w = 1.d0/n; psum = 0.d0
do i= me,n,nimg
   x = w * (i - 0.5d0); psum = psum + 4.d0/(1.d0+x*x)
enddo
mypi = w * psum
call sync_all()
if (me==1) then
  pi = sum(mypi[:]); write(6,*) 'computed pi = ',pi
endif
call sync_all(1)
end
```

The number of intervals and the partial sums of π are declared as co-arrays, because these must be communicated between images. All other variables are local to each image. The number of intervals is input on image 1 and broadcast to all images. Note that n without square brackets refers to the local part, n[me]. All images wait at the first sync_all for image 1 to arrive, signaling that n is safe to use. Each image then waits at the second sync_all for all images to complete the calculation. Finally, the first image adds the co-array of partial sums and writes out the result. The final sync_all prevents the other images from terminating the program before image 1 completes the write.

In OpenMP Fortran this becomes:

```
        program main
        call omp_set_dynamic( .false.)
        call omp_set_nested( .false.)
!$omp parallel
        call compute_pi
!$omp end parallel
        stop
        end
        subroutine compute_pi
        double precision :: psum,x,w ! threadprivate
        integer :: me,nimg,i ! threadprivate
        double precision :: pi
        integer :: n
        common /pin/ pi,n
!*omp threadshared(/pin/)
        integer omp_get_num_threads,omp_get_thread_num
        nimg = omp_get_num_threads()
        me = omp_get_thread_num() + 1
!$omp master
        write(6,*) 'Enter number of intervals'; read(5,*) n
        write(6,*) 'number of intervals = ',n
        pi = 0.d0
!$omp end master
!$omp barrier
        w = 1.d0/n; psum = 0.d0
        do i= me,n,nimg
        x = w * (i - 0.5d0); psum = psum + 4.d0/(1.d0+x*x)
        enddo
!$omp critical
        pi = pi + (w * psum)
!$omp end critical
!$omp barrier
!$omp master
        write(6,*) 'computed pi = ',pi
```

```
!$omp end master
!$omp barrier
      End
```

All SPMD OpenMP programs start with the same main program. It spawns the number of threads specified by the environment variable OMP_NUM_THREADS, then immediately calls the top level subroutine that represents the actual program to replicate. On exit from this subroutine all threads except the master thread are freed and the program then exits. The number of intervals and _ are declared in named common and are therefore global (thread-shared) variables by default. There is no compiler directive available to confirm the default, so a pseudo-directive, !*omp threadshared, is used to document that the common is shared. All other variables are local to the subroutine and therefore private to each thread (no saved variables). The number of intervals is input on the master thread, and since n is a global variable it is automatically available on all threads. All threads wait at the first !$omp barrier for the master thread to arrive, signaling that n is safe to use. Each thread then independently calculates its part of π and adds it to the total π. Updating π is in a critical region, so that only one thread at a time can access π. Each thread then waits at the second !$omp barrier for all threads to complete the calculation. Finally, the master thread writes out the result. The final !$omp barrier prevents the other threads from terminating the program before the master completes the write. This is probably unnecessary, since it is the master that will execute stop in the main program.

A relatively minor difference between the two versions is that Co-Array Fortran has a richer set of synchronization operations. In many cases, sync_all(1) is significantly faster than sync_all() because the former allows the image to continue as soon as image 1 arrives and the latter requires the image to wait for all images to arrive. OpenMP's !$omp barrier is the only synchronization of its kind provided by OpenMP and is equivalent to sync_all(). A synchronization routine like sync_all(wait) can be written in OpenMP, provided it is always called in conjunction with a !$omp flush directive. The primary difference between the two versions is that global variables are co-arrays spread across all images in Co-Array Fortran, but are standard variables in global memory (not assigned to any particular thread) in OpenMP Fortran. However, the difference is more one of style than substance. The

OpenMP version can be rewritten in Co-Array Fortran, by only using the part of each co-array on image 1:

```
program compute_pi
double precision :: psum,x,w
integer :: me,nimg,i
double precision :: pi[*] ! only use pi[1]
integer :: n[*] ! only use n[1]

nimg = num_images()
me = this_image()

if (me==1) then
        write(6,*) 'Enter number of intervals'; read( 5,*) n
        write(6,*) 'number of intervals = ',n
        pi = 0.d0
endif
call sync_all()

w = 1.d0/n[1]; psum = 0.d0
do i= me,(n[1]),nimg
        x = w * (i - 0.5d0); psum = psum + 4.d0/(1.d0+x*x)
enddo
call start_critical()
        pi[1] = pi[1] + (w * psum)
call end_critical()
call sync_all()

if (me==1) then
        write(6,*) 'computed pi = ',pi
endif
call sync_all()
end
```

In order to emulate shared variables, the Co-Array Fortran code replicates them on all images but only uses the part on image 1. All references to such variables must end in[16]. In the case of large shared arrays, it would be possible to avoid the space this wastes by defining a co-array of a derived type with a pointer component and then only allocating an array to the pointer on image 1. This sounds complicated, but is in fact the standard way for Fortran 90/95 to handle an array of arrays (or in this case a co-array of arrays). The master directive in OpenMP is replaced by a test for the first image. Co-Array Fortran does not treat the first image any differently than the others (i.e., it has no master image). However, standard input is available on the first image only, so if the master's tasks include reading standard input Co-Array Fortran must use the first image as the master.

Similarly, the Co-Array version can be expressed in OpenMP by adding a perthread dimension to each shared variable:

```
      program main
      call omp_set_dynamic( .false.)
      call omp_set_nested( .false.)
!$omp parallel
      call compute_pi
!$omp end parallel
      stop
      end
      subroutine compute_pi
      integer, parameter :: max_threads=128
      double precision :: pi,psum,x,w
      integer :: me,nimg,i
      double precision :: mypi
      integer :: n
      common /pin/ mypi(max_threads),n(max_threads)
!*omp threadshared(/pin/)
      integer omp_get_num_threads,omp_get_thread_num

      me = omp_get_thread_num() + 1
      nimg = omp_get_num_threads()
      if (me==1) then
            if (nimg>max_threads) then
                  write(6,*) 'error - too many threads ',nimg
                  stop
            endif
            write(6,*) 'Enter number of intervals';
                  read(5,*) n(me)
            write(6,*) 'number of intervals = ',n(me)
            n(1:nimg) = n(me)
      endif
!$omp flush
      call caf_sync_all(1)

      w = 1.d0/n(me); psum = 0.d0
      do i= me,n(me),nimg
            x = w * (i - 0.5d0); psum = psum + 4.d0/(1.d0+x*x)
      enddo
      mypi(me) = w * psum
!$omp barrier
      if (me==1) then
            pi=sum(mypi(1:nimg)); write(6,*) 'computed pi=',pi
      endif
!$omp flush
      call caf_sync_all(1)
      end
```

In order to emulate co-arrays, the OpenMP code puts them in named common (i.e. makes them shared variables) and converts co-array dimensions into additional regular array dimensions. Since array size has to be known at compile time, the parameter max_threads is introduced which has to be no smaller than the actual number of threads at run time. If this is set to a safe value, e.g., the number of processors on the machine, it is probably an over estimate and hence wastes memory. Co-Array Fortran allows the local part of a co-array to be referenced without square brackets, but all references to emulated co-arrays must include the co-dimensions, e.g. n(me). It also "knows" that the co-size is num_images(), so mypi[:] is legal Co-Array Fortran but must become mypi(1:nimg) in OpenMP Fortran. The routine caf_sync_all is assumed to be an OpenMP implementation of sync_all but it can only synchronize threads, the !$omp flush is also required to synchronize shared objects.

5.3 A comparison

The features of SPMD OpenMP Fortran and Co-Array Fortran are summarized in Figure 5.1. OpenMP Fortran is only applicable to systems with a single global memory space, and perhaps only to those with flat single level addressing and cache coherence across the entire memory space (i.e., systems such as the Cray T3E are not candidates for OpenMP). However, this includes a wide range of SMP and DSM systems with from 2 to 256 processors. OpenMP is a relatively new "standard," but it has wide vendor and third party support is available on almost all machines with a suitable global shared memory, from PC's to MPP's. Compilers with partial support for OpenMP typically do not support it in SPMD-mode, but most compilers now claim full version 1.1 complience. Version 2.0 complient compilers are not yet typically available, but for SPMD programmers only the extension of THREADPRIVATE to saved and module variables and the new COPYPRIVATE clause are significant, so even partial support for version 2.0 may be sufficient.

Co-Array Fortran can take full advantage of a hardware global memory, but it can also be used on shared nothing systems with physically distinct memories connected by a network. However, performance is expected to only be about as good as MPI on such systems. Co-Array Fortran can be implemented using threads or processes, or on a cluster of SMP systems it could even use threads within a SMP system and

processes between systems. It is therefore more widely applicable than OpenMP Fortran. However, the Cray T3E is the only machine with a Co-Array Fortran compiler today and it implements only a subset of the language. There is

Figure 5.1: Features of SPMD OpenMP Fortran and Co-Array Fortran

Feature	SPMD OpenMP Fortran	Co-Array Fortran
availability	wide-spread	Cray T3E only
implementable using	threads	threads and/or processes
target memory architecture	cache-coherent global	any
on single thread/image, get	standard Fortran	extended Fortran
incompatible Fortran extension	copy-in/out	none
local variables	private	private
saved and module variables	shared	private (or co-array)
common variables	shared (or private)	private (or co-array)
pointers	local or global	local
communication	shared variables	co-arrays
memory synchronization	local scope	global scope
memory layout control	automatic for private none for shared	automatic for private automatic for co-arrays
synchronization	global critical regions, locks	global or team (local short cut) critical region
I/O namespace	single and shared	single but private
I/O operations	unsafe to same unit	safe

a definite need for a source to source compiler that will allow Co-Array Fortran to run on the same systems as OpenMP Fortran. This is discussed in more detail in Chapter 5.4.

Any Co-Array Fortran program is translatable into OpenMP Fortran and vice versa. However, the differences between co-arrays and shared arrays tend to steer programmers to alternative solutions to the same problem. For example, suppose there are P images or threads and we need to perform operations both on an array, A(1:M,1:N), and its transpose, AT(1:N,1:M). For simplicity further assume that both M and N are multiples of P. In Co-Array Fortran we would probably store both as co-arrays, A(1:M,1:N/P)[*] and AT(1:N,1:M/P)[*], and write a routine to copy between them. Since A and AT are co-arrays, the routine can do a direct copy without using intermediate buffers. In OpenMP, for efficiency of memory layout we might similarly store both as private arrays on each thread, A(1:M,1:N/P) and AT(1:N,1:M/P), but now we have to provide a shared buffer to copy between them.

The simplest shared buffer to use is the entire array, B(1:M,1:N). Then the copy routine is just copy each private A into the shared B, barrier, copy the shared B into each private AT. An alternative in OpenMP is to always store the array as a whole shared array, A(1:M,1:N). It may then be unnecessary to store the transpose at all, although cache effects may make it advisable to also have a shared transpose, AT(1:N,1:M). The shared array approach is also available in Co-Array Fortran by placing arrays on one image, but to avoid wasting memory a co-array of arrays, CA[1]%A(1:M,1:N), would probably be used rather than a simple co-array, A(1:M,1:N)[1]. In either case, the co-array syntax makes clear that accessing the "shared" array is a potentially expensive remote memory operation. The shared array approach is sometimes the easiest to use, and is more cleanly expressible in OpenMP Fortran, but it comes at the cost of less programmer control over performance.

OpenMP Fortran contains no directives to control the layout of shared arrays in memory. This is not an issue on SMP systems with uniform memory access, but where in memory shared arrays are placed may have a large effect on performance on non-uniform memory access (NUMA) systems. This limits OpenMP's scalability to large numbers of nodes, since large node-count systems tend to be NUMA. OpenMP Fortran is primarily designed for fine grained loop parallelization, which is typically appropriate for small node counts. Therefore the lack of layout control is less of an issue for OpenMP in its primary domain of interest, but it is a concern for SPMD programs. All memory in Co-Array Fortran is associated with a local image, so memory placement on NUMA systems is simple to arrange and does not effect scalability. Since Co-Array Fortran always knows when remote communication is involved, the global memory does not need to be cache coherent and in fact each image's memory can be physically and logically distinct with only a fast network connecting them. Overall, Co-Array Fortran has clear advantages on systems with large node counts (above about 32 processors).

Co-Array Fortran is a simple set of extensions to Fortran 90/95. The features that are compatible with Fortran 77 do not produce a viable subset language. OpenMP compilers typically support Fortran 90 or 95, but the version 1.1 compiler directives really only apply well to Fortran 77 programs. The lack of support for thread private module variables and for ALLOCATABLE are two examples of this. Fortran 77 is

probably still the dominant variant for SPMD programs, but large projects, in particular, are increasingly migrating to Fortran 95 and will need version 2.0 OpenMP compilers.

One example of Co-Array Fortran's reliance on Fortran 90/95 features is that any subroutine with a co-array dummy argument must have an explicit interface. Hence a Fortran 77 subset would either have to ban co-array dummy arguments or provide an extension to the existing language that distinguishes between co-array actual arguments and local part actual arguments without an explicit interface. Explicit interfaces make Co-Array Fortran significantly safer to use. The compiler always has complete knowledge of co-arrays except in the special case when the local part of a co-array is passed to a dummy argument of co-rank zero. The programmer is responsible for co-array safety in this special case, and must make sure that no other image references the passed piece of the co-array between the subroutine call and return. A synchronization call in Co-Array Fortran always implies that all co-arrays are up to date (except those passed to co-rank zero dummies, and these must not be referenced from other images anyway). All the co-array "machinery" works behind the scenes to allow the programmer to do the obvious thing and in fact get the expected result.

OpenMP Fortran provides a significantly lower level programmer interface. Once an object has been passed to a procedure through its argument list there is no way to tell if it is a shared object or a private object. Pointers can be shared or private and both can reference either shared or private variables, so it is possible (although unsafe) for one thread to access the private memory of another thread. Also, synchronization primitives only apply to shared objects in the local scope. All subroutine arguments are potentially "thread visible" (i.e. shared), so all have to be flushed even though some may actually be private. Local scope synchronization has several pitfalls to trap the unwary programmer. One of the most obvious is
copy-in/copy-out:

```
      common/shared/ i(100)
!$omp master
      i(1:100) = 0
!$omp end master
!$omp barrier
      call sub1(i(2:100:2))
```

```
!$omp master
      write(6,*) i(4)
!$omp end master
      end
      subroutine sub1(i2)
      integer :: i2(0:49),omp_get_thread_num
      i2(omp_get_thread_num()) = omp_get_thread_num()
!$omp barrier
      end
```

The barrier in sub1 synchronizes i2, but, since it is not an assumed shape array, i2 is probably only a local contiguous copy of i(2:100:2) and a different local copy on every thread (the only practical alternative a compiler has is to in-line sub1). If a local copy is used, the barrier has no effect on i and when each thread returns from sub1 they will independently copy back their entire local version of i(2:100:2) into the shared original and perhaps also update a register holding i(4) from the local version. This means that there is no way to tell if the write prints the value 1, as expected, or 0. This is not just a local scope issue, since the problem remains even if the second barrier is moved from inside sub1 to just after the call to sub1. The value of i(4) then depends on which thread exits sub1 last. The only safe approach is for the programmer to manually implement steps similar to those that Co-Array Fortran takes. Issue a !$omp flush before and after every subroutine call that might contain synchronization, and if a shared array section that is not contiguous in array element order is passed to a subroutine the associated dummy array argument must be assumed shape (and the subroutine interface therefore explicit). The OpenMP specification makes the above example illegal, i.e. it places the responsibility onto the programmer to avoid such copyin/ copy-out race conditions. Co-Array Fortran guarantees that copy-in/copy-out is never required for co-array dummy arguments, e.g., if i2 were a co-array an array section actual argument would be illegal (and detectable as an error at compile time) unless i2 is declared assumed shape. All library-based SPMD APIs have similar consistency problems. The MPI-2 standard [23] has a good discussion of these issues, which can cause optimization problems in Fortran 77 but are much more serious for Fortran 90/95. Co-Array Fortran may be unique among SPMD APIs in having no known conflicts with Fortran 90/95. High Performance Fortran is also consistent with Fortran 95, but is not formally a SPMD API although it is often implemented using SPMD.

The previously listed limitations make OpenMP Fortran a less than optimal choice for very large SPMD programs, but it has the important advantage of being widely available. There is a preliminary port of the NRL Layered Ocean Model (NLOM) to OpenMP Fortran [32]. NLOM already ran in SPMD-mode using MPI or SHMEM. The original code is 69,000 lines of Fortran 77 including 22,000 comment lines of which 500 are compiler directives (many are repeats in different dialects). Ignoring communication routines, adding support for OpenMP required 900 OpenMP compiler directives, 500 to characterize all COMMON's (could be reduced using INCLUDE) and 400 primarily to handle I/O. This illustrates a general property of compiler directive based APIs, they are very verbose. Other, extensive, changes were required to allow sequential I/O to be compatible with either SPMD processes or SPMD threads. Programs that do all sequential I/O from a single image would not require these modifications. If the COPYPRIVATE directive qualifier had been available the sequential I/O modifications would have been greatly simplified. Shared variables were added to handle sequential I/O, but in general variables outside communication routines are THREADPRIVATE. Fortunately, NLOM does not use modules and most saved local variables had already been placed in common. However, DATA statement initialization had to be modified to make sure there were no implied saved local variables. Since this was a prototype port, the required communication routines were generated by replicating the existing 4,000 line SHMEM version and making as few changes as possible to support OpenMP.

The native OpenMP port of NLOM has been made obsolete by adding support to NLOM for a dialect of Co-Array Fortran that can be automatically translated into OpenMP Fortran using a nawk [18] script (described in more detail below). Outside communication routines, this involved adding macros (that are null except when using Co-Array Fortran) to 230 I/O statements and adding Co-Array syntax to a single subroutine that defines arrays (co-arrays) used by communication routines. Inside communication routines, this involved replicating and modifying SHMEM-specific code fragments for Co-Array Fortran (with each SHMEM library call mapping to a single co-array assignment statement), and adding a macro identifying the local part of a co-array to 375 assignment statements. The latter are only required due to limitations in the nawk script and are null when using a true Co-Array Fortran compiler. The total effort involved in writing the nawk script and adding Co-Array

Fortran support to NLOM was significantly less than required to add native OpenMP support. The nawk script adds all required OpenMP compiler directives and emulates Co-Array Fortran I/O, thus removing the two most time consuming aspects of the port.

NLOM is typical of many SPMD codes in using a program-specific interface to handle all communication. This greatly simplifies porting to a new SPMD API, but reduces the opportunity for optimization with a low latency API (such as either Co-Array and OpenMP Fortran). Porting an existing SPMD program, e.g. one using MPI, that did not separate out communication to OpenMP Fortran would be difficult, because MPI allows communication between what are in OpenMP terms THREADPRIVATE objects and in fact has no concept of THREADSHARED objects. Porting any existing SPMD program to Co-Array Fortran would be much easier, because all objects including co-arrays can be treated as local to an image and co-arrays need only be introduced at all for objects that are involved in communication.

5.4 Translation

5.4.1 Subset Co-Array Fortran

The full Co-Array Fortran language [24] provides support for legacy SPMD programs based on Cray's SHMEM put and get library [16]. It requires that all variables in named COMMON be treatable either as standard variables or as co-arrays, and which objects in the COMMON block are co-arrays is allowed to vary between scoping units. This makes it difficult to implement co-arrays as if they were Fortran arrays of higher rank. The following relatively minor restrictions on the full language define a formal Subset that significantly widens the implementation choices, and in particular allows a simple mapping from Co-Array Fortran to OpenMP Fortran.

1. If a named COMMON includes a co-array, every object in that COMMON must be a co-array. The objects in the COMMON must agree in size, type, shape, co-rank and co-extents in all scoping units that contain the COMMON.
2. The EQUIVALENCE statement is not permitted for co-arrays.

3. The sum of the local rank plus the co-rank of a co-array is limited to seven.
4. A dummy co-array argument cannot have assumed size local dimensions.
5. A dummy co-array argument cannot have assumed shape local dimensions, unless the co-rank is one. The actual argument shall then also have co-rank one.
6. If a dummy argument has both nonzero local rank and nonzero co-rank and does not have assumed shape local dimensions, the actual argument must agree in size and type with the dummy argument.

The restrictions on COMMON are similar to nonsequence COMMON in HPF [22]. The restrictions on EQUIVALENCE are more severe than in HPF, for simplicity, but in Subset Co-Array Fortran the restrictions only apply to co-arrays. COMMON can be largely replaced by MODULE for new programs, so the restrictions are easily met except when migrating legacy Fortran 77 programs that make heavy use of COMMON and either EQUIVALENCE or different layouts for the same named common in different scopes. If the objects in a legacy COMMON already agree in size, type, and shape in all scoping units (which is good programming practice), then every object in that COMMON can be converted to a co-array without changing the meaning of the program. This is because a reference to a co-array without square brackets is always a reference to the local part of the co-array. Migration to the Subset is therefore easy in this case.

In Co-Array Fortran both the local rank and the co-rank of a co-array can be seven, but the local rank plus co-rank of any co-array subobject that is actually used in an executable statement must be no more than seven (because the rank and co-rank are merged and the object treated as a standard array subobject). Thus the Subset's restriction on local rank plus co-rank to seven is not typically a severe additional constraint. It would be helpful if Fortran 2000 increased the limit on the rank from seven to, say, ten, since this would give more room for rank plus co-rank.

The restrictions on dummy co-array arguments may require the programmer to explicitly pass additional array dimension information through the argument list. The restriction on assumed shape is probably the most severe of all for new programs, since assumed shape arrays are a significant simplifying factor in Fortran 90/95 programs and Co-Array Fortran requires some kinds of co-array actual arguments to

only be associated with assumed shape dummy arguments. It is a consequence of the fact that co-size is always NUM_IMAGES() and therefore that, when the co-rank is greater than one, the co-array has no final extent, no final upper bound, and no co-shape.

The Subset does not allow any kind of array element sequence association for co-arrays. It therefore prohibits an element of a co-array being passed to a subroutine and treated there as a co-array of non-zero rank. Only entire co-arrays can be passed to explicit-shape co-array dummy arguments and the size of the actual and dummy argument must be identical.

5.4.2 Subset Co-Array Fortran into OpenMP Fortran

Subset Co-Array Fortran has been designed to be implementable by mapping co-arrays onto arrays of higher rank. In particular, they are implementable as shared OpenMP Fortran arrays. The translation of the Co-Array Fortran π program into OpenMP Fortran presented in Chapter 5.3 illustrates what a compiler is required to do. Any saved or module local variables must be placed in THREADPRIVATE named common (or just declared THREADPRIVATE in version 2.0). Any named common that does not contain co-arrays must be made THREADPRIVATE. All co-arrays must be shared objects. Square brackets are merged to create arrays of higher rank. The convention that references to a co-array without square brackets is a reference to the local part of the co-array requires first expanding the reference to include both round brackets and square brackets, and then merging square brackets to create an array subobject. References to co-arrays in procedure calls do not typically include square brackets, but the intent is always unambiguous because the interface must be explicit when the dummy argument is a co-array. If the dummy argument is not a co-array, the reference must be expanded to explicitly pass the local part of the co-array to the procedure. If the dummy argument is an assumed shape co-array (with co-rank one), the dummy is translated to an assumed shape array with one higher rank and special handling may also be required on the calling side. Co-Array intrinsic procedures can be implemented as an OpenMP module. All caf-procedure calls [24], i.e., calls to procedures that could contain synchronization, must be bracketed by FLUSH directives that explicitly name all actual co-arrays in the local

scope. A generic FLUSH without arguments would also be sufficient, but is less efficient because OpenMP would then flush objects that the original Co-Array source has identified as not being thread visible. All of Co-Array I/O maps directly onto threadsafe OpenMP I/O, so the translator may have to explicitly make I/O thread safe, using critical directives, but the mapping is otherwise straight forward. The translation process has been presented as if performed by a Subset Co-Array Fortran source to OpenMP Fortran source compiler. Many of the steps are trivial if actually performed by retargeting an existing native OpenMP Fortran compiler to support Subset Co-Array Fortran. So on machines with a cache-coherent shared memory and an OpenMP compiler it would take very little effort on the vendors part to support Subset Co-Array Fortran. A single compiler is typically already used for standard Fortran and OpenMP Fortran, with the target language specified at compile and link time. With minor upgrades the same compiler can also support Subset Co-Array Fortran. There would be a single compiler but three distinct languages, so linking standard Fortran and Subset Co-Array Fortran objects together would not be supported (just as linking standard Fortran and OpenMP Fortran objects is not supported now).

As a “proof of concept” a nawk script has been developed to translate Subset Co-Array Fortran directly into OpenMP Fortran. Since this is a pattern matching script, rather than a compiler, it treats some keywords as reserved and requires some statements be expressed in one of the several alternatives that Fortran provides. In order to implement TEAM read, I/O unit numbers are restricted to be less than 100,000. The only other significant variances from the Subset Co-Array Fortran language are those made necessary by a lack of a symbol table identifying modules and co-arrays by name. The most serious of these is that the local part of a co-array cannot be referenced without square brackets. To simplify local parts, the script will automatically translate a copy of the co-array’s declaration square brackets, with "*" replaced by "@" into square brackets identifying the local part. For example:

```
COMMON/XCTILB4/ B(N,4)[0:MP-1,0:*]
SAVE /XCTILB4/
CALL SYNC_ALL( WAIT=(/IMG_S,IMG_N/) )
B(:,3)[0:MP-1,0:@] = B(:,1)[I_IMG_S,J_IMG_S]
B(:,4)[0:MP-1,0:@] = B(:,2)[I_IMG_S,J_IMG_N]
CALL SYNC_ALL( WAIT=(/IMG_S,IMG_N/) )
```

Only the local part of B(:,3) and B(:,4) is used, but square brackets are still required and have been provided by replicating the square bracket declaration of B with "*" replaced by "@". The advantage of this extension to the language is that these square brackets can be removed by a batch stream editor to produce a legal Subset program.

Absent a symbol table tracking explicit interfaces, passing co-arrays to subroutines also requires extensions to the Subset language. A whole co-array can be passed to a co-array dummy just as in the Subset, but all other cases rely on an extension to the Subset to allow co-array sections to be passed as arguments. A co-array section passed to a co-array dummy must include square brackets that cover the entire co-extent. A local part passed to a dummy of co-rank zero must use square brackets to form the corresponding co-array section.

The nawk script obviously provides a way of running Co-Array Fortran programs (after some manual tweaking) via an OpenMP compiler. But it can also be simply viewed as a pre-processor that provides an improved SPMD interface for OpenMP. It has several major advantages over native OpenMP Fortran for SPMD programs. For example, I/O is consistent with process-based SPMD APIs and the mapping of variables onto shared and private memory is greatly enhanced (because the script automatically places variables in COMMON as necessary). Co-Array Fortran intrinsic procedures provide a much richer set of synchronization options than OpenMP, and the special handling of caf-procedures ensures that synchronization of threads implies synchronization of co-arrays. A disadvantage of the nawk script is that it provides no error checking. Legal Co-Array Fortran programs are translated to legal OpenMP Fortran programs, but illegal programs will also be translated and it is up to the OpenMP compiler to detect the error. Error messages are likely to be obscure, but relatively few lines are modified in the translation so inspection of the OpenMP source should provide an indications of the error. A true Subset Co-Array compiler, either provided as an addition to an OpenMP compiler or as a stand-alone source to source compiler, would not have any of the restrictions of the nawk script and would be able to provide clear and relevant error diagnostics for non-conforming syntax.

One advantage that OpenMP has over Co-Array Fortran is that if an OpenMP program is designed to work when there is exactly one thread, it is then also a legal Fortran 90/95 program. The compiler directives have no effect on one thread, and are ignored by the Fortran 90/95 compiler. A library of a few standard procedures is required, but is trivial to implement for a single thread. This is not the case for Co-Array Fortran. Obviously, a Subset Co-Array Fortran source to OpenMP Fortran source compiler would also be a Subset Co-Array Fortran source to Fortran 90/95 source compiler in the special case of one image. A much simpler source to source compiler is sufficient in this special case, and a public domain implementation would provide a useful service to the Co-Array Fortran programming community. This is not quite just a matter of deleting all references to square brackets, because the effective rank of a co-array subobject is the sum of its local rank and co-rank. If the square brackets are deleted the effective rank may change, giving rise to illegal ranks for intrinsic procedure arguments and non-conforming ranks in some array assignment statements involving co-arrays.

Chapter 6 Related Work

This chapter starts with an overview of existing parallel programming languages; MPI and UPC, Next it gives example of theese two languages programming code with the same problem in the case study. And shows how the parallel matrix multiplication code looks like in MPI and UPC.

6.1 MPI

MPI is a language-independent communications protocol used to program parallel computers. Both point-to-point and collective communication are supported. MPI "is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation." MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today.

6.2 UPC

Unified Parallel C (UPC) is an extension of the C programming language designed for high-performance computing on large-scale parallel machines, including those with a common global address space (SMP and NUMA) and those with distributed memory (e.g. clusters). Theprogrammer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single execution per processor.
In order to express parallelism, UPC extends ISO C 99 with the following constructs:

- An explicitly parallel execution model
- A shared address space
- Synchronization primitives and a memory consistency model
- Memory management primitives

processor. UPC uses a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time, typically with a single thread of.

The UPC language evolved from experiences with three other earlier languages that proposed parallel extensions to ISO C 99: AC, Split-C, and Parallel C Preprocessor (PCP). UPC is not a superset of these three languages, but rather an attempt to distill the best characteristics of each. UPC combines the programmability advantages of the shared memory programming paradigm and the control over data layout and performance of the message passing programming paradigm.

6.3 Matrix Multiplication in MPI and UPC

6.3.1 MPI code for matrix multiplication;

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define ms 2

int main(int argc,char* argv[])
{
    int i,j,k;
    int x,c;
    int matrix_a[ms][ms];
    int matrix_b[ms][ms];
    int matrix_c[ms][ms];
    int myrank, p;
    int NRPE;
    double starttime, endtime;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Status status;

    NRPE = ms / p;

    if(myrank == 0)
    {
         printf("\nTCE 3411 Parellel Processing : Matrix
Multiplication\n");

         printf("\nDate : %s",__DATE__);
         printf("\nTime : %s",__TIME__);

printf("\n=====================================================\
n");


        printf("\n   matrix a \n");
        printf("-------------\n");
```

```
        for(i=0; i<ms; ++i)
            for(j=0; j<ms; ++j)
                matrix_a[i][j] = rand() % 10;

        for(i=0; i<ms; ++i)
        {
            for(j=0; j<ms; ++j)
                printf("%3d", matrix_a[i][j]);
            printf("\n");
        }

        printf("\n   matrix b \n");
        printf("--------------\n");

        for(x=0; x<ms; ++x)
            for(c=0; c<ms; ++c)
                matrix_b[x][c] = rand() % 10;

        for(x=0; x<ms; ++x)
        {
            for(c=0; c<ms; ++c)
                printf("%3d", matrix_b[x][c]);
            printf("\n");
        }

    }

        for(i=0; i < ms; i++)
    {
        MPI_Bcast(matrix_b[i], ms*ms, MPI_INT, 0,
MPI_COMM_WORLD);
    }
    printf("\n MATRIX B by Process: %d\n", myrank);
    for(x=0; x<ms; ++x)
    {
        for(c=0; c<ms; ++c)
            printf("%3d", matrix_b[x][c]);
        printf("\n");
    }

    for(i=0; i<p; i++)
    {
      for(j=0; j<ms; j++)
       {
            MPI_Send(&matrix_a[j], ms*NRPE, MPI_INT, i, 0,
MPI_COMM_WORLD);
            NRPE++;
       }
    }

    starttime = MPI_Wtime();
    for (k=0; k<ms; k++)
    for (i=0; i<ms; i++) {
      matrix_c[i][k] = 0;
      for (j=0; j<ms; j++)
        matrix_c[i][k] = matrix_c[i][k] + matrix_a[i][j] *
matrix_b[j][k];
      }
      endtime   = MPI_Wtime();
    MPI_Send(&matrix_c[i][k], ms*ms, MPI_INT, 0, 0,
MPI_COMM_WORLD);
```

```
       if(myrank == 0)

     printf("\n\nParellel Time %f seconds\n",endtime-
starttime);
    }
  printf ("\n");

    return 0;
    MPI_Finalize();
}
```

6.3.2 UPC code for matrix multiplication;

```
#define M 200
#define N 250
#define P 50
shared double A[M][P];
shared double B[P][N];
shared double C[M][N];
static double timer(){
  struct timeval tv;
  gettimeofday(&tv,NULL);
  return (double)tv.tv_sec + 1e-6*(double)tv.tv_usec;
}
void verify(int niter){
  int i,j;
  if (MYTHREAD == 0) {
    for (i=2; i<M; i++)
      for (j=0; j<N; j++)
        {
          double i1 = 1.0/((double)i+1);
          double shb = niter * (pow (i1, (double)(j+1))-1) /
(i1-
1);
          double diff = (C[i][j]-shb)/shb; if (diff < 0) diff =
-
diff;
          if (diff > 1e-8) {
                       printf("Verification FAILED\n");
                       return;
                }
        }
  }
}
void init_matrix(){
  int i, j, k;
  if (MYTHREAD == 0) {
    for (i=0; i<M; i++) for (k=0; k<P; k++) A[i][k] = pow (1.0/
(double)(i+1), (double)k);
    for (k=0; k<P; k++) for (j=0; j<N; j++) B[k][j] = k<=j;
    for (i=0; i<M; i++) for (j=0; j<N; j++) C[i][j] = 0;
  }
  upc_barrier;
void matmult_naive(){
      int i,j,k;
      upc_forall (i=0; i<M; i++; continue)
        upc_forall (j=0; j<N; j++; &C[i][j])
          {
            double s = C[i][j];
            for (k=0; k<P; k++) s += A[i][k] * B[k][j];
```

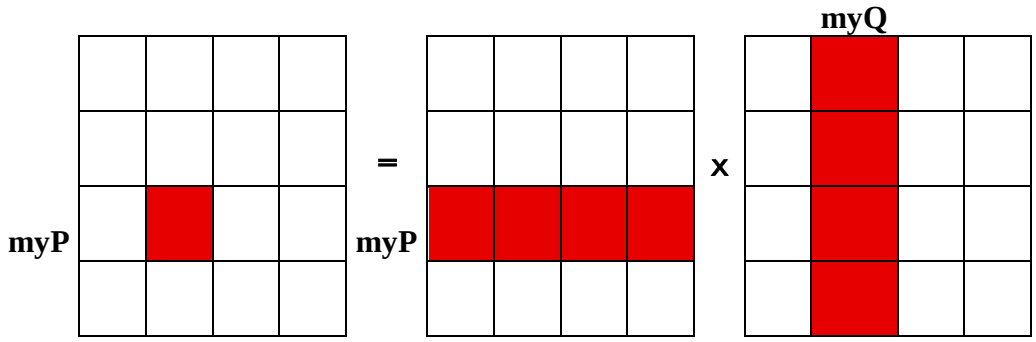
```
            C[i][j] = s;
          }
}
int main(){
  int niter = 20;
  double flops1;
  int i, j, k, iter;
  init_matrix();
  upc_barrier;
  double t1 = timer();
  for (iter = 0; iter < niter; iter++)
        matmult_naive();
  double t2 = timer();
  upc_barrier;
  verify(niter);
  upc_barrier;
  double t = t2 - t1;
  flops1 = (double)M * N * P * 2 * niter / t;
  upc_barrier;
  if (MYTHREAD==0)
        printf ("Naive matmult UPC: %g GFlops\n", flops1*1e-9);
  upc_barrier;
  return 0;
```

Chapter 7 Case Study

7.1 Matrix multiplication in Co Array Fortran

For this study our platform is Amazon EC2 Cluster system with 16 CPU and CentOS operating system.

For this study our problem is nxn matrix multiplication.



7.2 Co Array Fortran Code

```
program matmul
implicit none
real, allocatable,dimension(:,:), codimension[:,:] :: a,b,c
integer :: i
integer :: j
integer :: k
integer :: l
integer,parameter :: n = 10
integer :: p
integer :: q
integer :: iAm
integer :: myP
integer :: myQ
p = num_images()
q = int(sqrt(float(p)))
iAm = this_image()
if (q*q /= p) then
if(iAm == 1) write (*,"('num_images must be square: p=',i5)") p
stop
end if
allocate(a(n,n)[q,*])
allocate(b(n,n)[q,*])
```

```
allocate(c(n,n)[q,*])
myP = this_image(c,1)
myQ = this_image(c,2)
a = 1.0
b = 1.0
c = 0.0
sync all
do i=1,n
do j=1,n
do k=1,n
do l=1,q
c(i,j) = c(i,j) + a(i,k)[myP, l]*b(k,j)[l,myQ]
end do
end do
end do
end do
if (any(c /= n*q)) write(*,"('error on image: ',2i5,e20.10)")
myP, myQ, c(1,1)
write(*,"('check sum[',i5',',i5,']',e20.10)") myP, myQ, sum(c)
- q*n**3
deallocate(a,b,c)
end program matmul
```

7.3 Performance Analysis

As I defined in Chapter 7.1 my platform for this case study is Amazon EC2 Cluster system with 16 x Intel Xeon X5570, quad core"Nehalem" architecture CPU and CentOS operating system.

I ran the matrix multiplication program code for 120x120, 200x200, 320x320, 400x400, 520x520, 600x600, 720x720, 800x800, 920x920 and 1000x1000 matrices each on 1CPU, 2 CPU, 4 CPU, 8 CPU, 16 CPU.
Performance figures are shown below. Figure 7.1 shows performance table of CAF code. Figure 7.2 shows, performance chart of CAF code. Figure 7.3 shows performance chart of 120x120 matrix on 1CPU, 2 CPU, 4 CPU, 8 CPU and 16 CPU. Figure 7.4 shows performance chart of 520x520 matrix on 1 CPU, 2 CPU, 4 CPU, 8 CPU, 16 CPU. An the last figüre, Figure 7.5 shows performance chart of 1000x1000 matrix on 1 CPU, 2 CPU, 4 CPU, 8 CPU, 16 CPU.

Figure 7.1: Performance table of Co Array Fortran Code

NxN	1 CPU	2 CPU	4 CPU	8 CPU	16 CPU
120x120	0,0592	0,0312	0,0198	0,0162	0,0098
200x200	0,1245	0,0834	0,0600	0,0424	0,0211
320x320	0,3039	0,1821	0,1498	0,1268	0,0763
400x400	0,5983	0,3876	0,2581	0,2136	0,1632
520x520	3,1265	1,7426	1,2646	0,9828	0,6139
600x600	4,0256	2,1674	1,5658	1,2318	0,0892
720x720	5,4328	2,8710	2,0234	1,8341	1,3487
800x800	6,1390	3,1438	2,2717	1,9732	1,6034
920x920	7,1845	3,9048	3,1249	2,8645	2,2431
1000x1000	8,8240	4,8296	3,5477	3,1241	2,4987

Figure 7.2: Performance chart of CAF code for NxN matrix

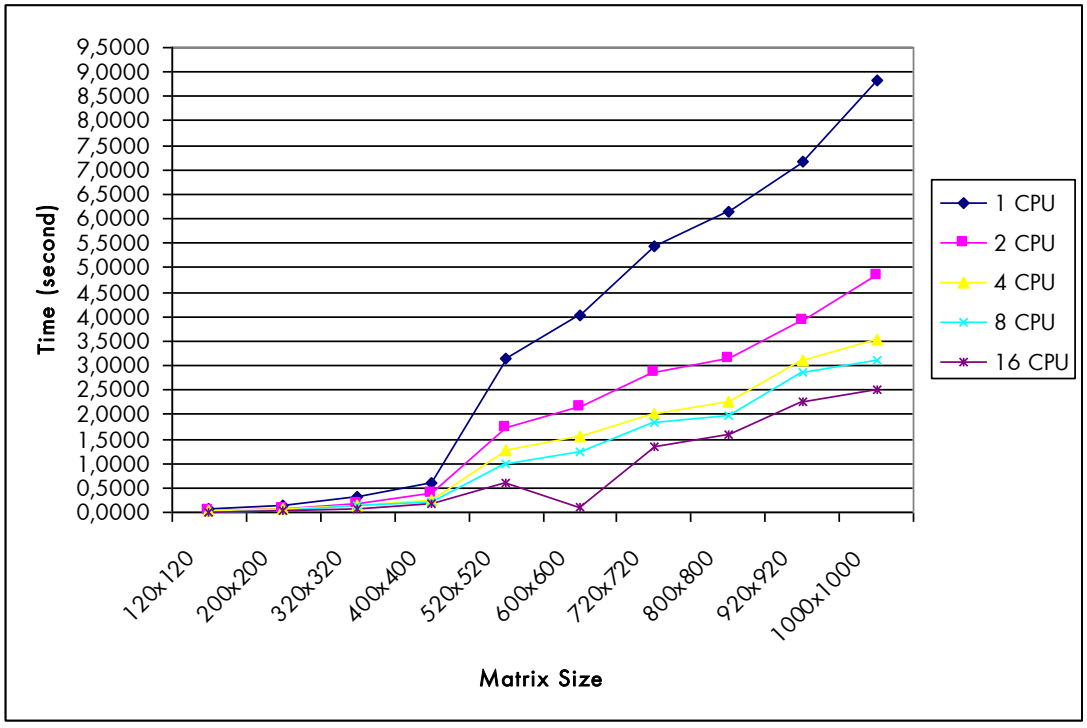


Figure 7.3: Performance chart for 120x120 matrix

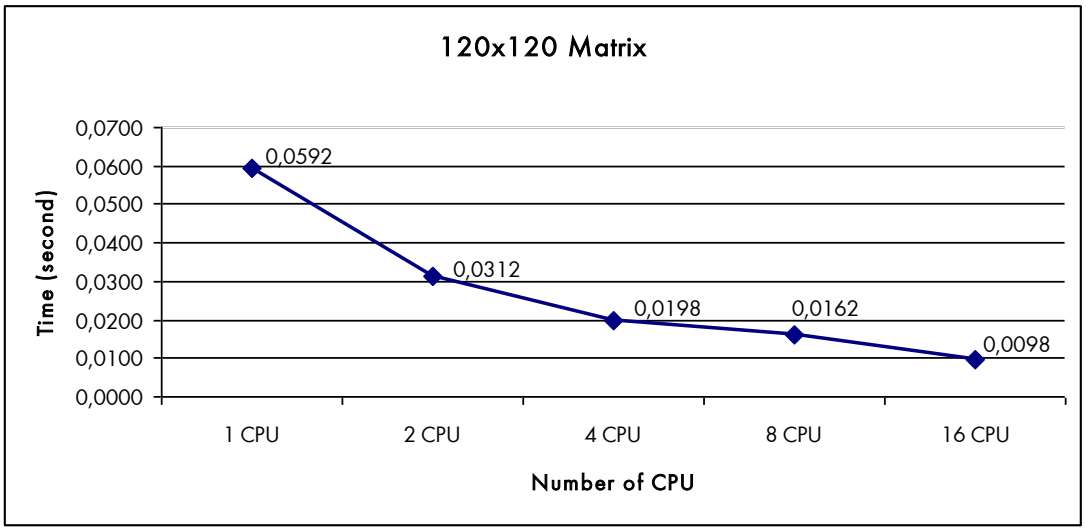


Figure 7.4: Performance chart for 520x520 matrix

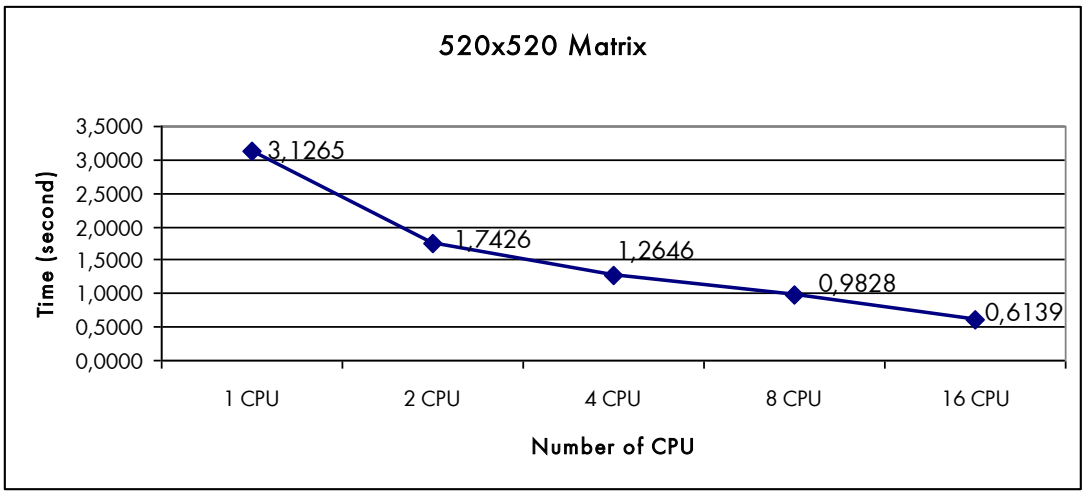
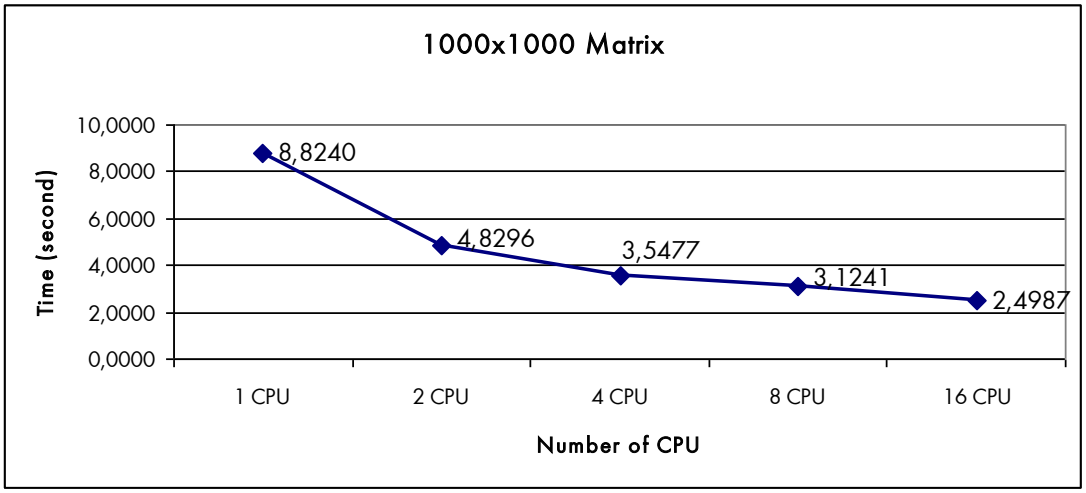


Figure 7.5: Performance chart for 1000x1000 matrix



7.4. Conclusion

Co-array Fortran looks and feels like Fortran and requires Fortran programmers to learn only a few new rules. The CAF syntax gives the programmer more control and flexibility. In Co-array fortran easy to write programs this increase the productivity. Co-array Fortran is a PGAS language and able to take adventages of PGAS. There is no subroutine calls in Co-array Fortran compiler can optimize across assignment. The Co-array Fortran performance is better than the library based models.

Curriculum Vitae

Aşkın ODABAŞI was born on 02 August 1977, in Köln. He received B.S. degree in 2003 in Computer Engineering from Sakarya University. Since 2008, He has been an IT specialist at a private company.

Education&Training

MS, Computer Engineering, Kadir Has University, Istanbul (Ongoing)

BS, Computer Engineering, Sakarya University, Sakarya (2003-2000)

Vocational High School, Computer Programming, Çanakkale 18 Mart University (1999-1997)

Atatürk Lisesi, Ordu (1996-1993)

References

[1] Co Array Fortran.
http://neptune.ce.ncsu.edu/~sarat/sc07/PGAS-SC2007/CAF/caf_intro.htm

[2] Co Array Fortran 2.0.
http://caf.rice.edu

[3] Co Array.
http://www.coarray.org

[4] C. Coarfa. *Portable High Performance and Scalability of Partitioned Global Address Space Language*. Doctor's Degree thesis, Rice University, Houston, Texas, Jan. 2007

[5] R.W. Numrich and J. Reid. Co-Arrays in the next fortran standard. *ACM Fortran Forum*, 24(2):4-17, Aug. 2005.

[6] *The PGAS Programming Model and Coarray Fortran* – a lacture in the EPCC course: Parallel Decomposition, Dr. Michele Weiland.

[7] R.W. Numrich and J. Reid. Co-Arrays in the next fortran standard. *ACM Fortran Forum*, 17(2):1-31, Aug. 1998.

[8] Partitioned Global Address Space
http://mohamedfahmed.wordpress.com/2010/05/06/partitioned-global-address-space-pgas/

[9] J. Reid and R.W. Numrich. Co-Arrays in the next Fortran Standard, *Scientific Programming* 15(1), 9-26 (2007).

[10] R.W. Numrich, A Parallel Numerical Library for Co-Array Fortran, S*pringer Lecture Notes in Computer Science 3911*, 960-969 (2005).

[11] R.W. Numrich, Parallel numerical algorithms based on tensor notation and Co-Array Fortran syntax, *Parallel Computing 31, 588-607* (2005).

[12] R.W. Numrich and J.K. Reid, Co-Array Fortran for Parallel Programming, *ACM Fortran Forum* 17(2):1-31 (1998).

[13] R.W. Numrich, J. Reid and K. Kim, Writing a Multigrid Solver Using Co-Array Fortran, *Springer Lecture Notes in Computer Science 1541*, 390-399 (1998).

[14] R.W. Numrich, F--: A Parallel Extension to Cray Fortran, *Scientific Programming* 6(3), 275-284 (1997).

[15] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[16] Cray Research Inc. *Application Programmer's Library Reference Manual*. Cray Research SR-2165, 1996.

[17] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 2001.

[18] D. Dougherty. *Sed & Awk*. O'Reilly and Assoc., Sebastopol, CA, 1990.

[19] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, New York, NY, USA, 1997. ACM Press.

[20] T. E. Anderson and E. D. Lazowska. Quartz: a tool for tuning parallel program performance. In *SIGMETRICS '90: Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–125, New York, NY, USA, 1990. ACM Press.

[21] S. Kleiman, D. Shah, and B. Smaalders. *Programming with Threads*. SunSoft Press, Prentice Hall, Upper Saddle River, NJ, 1996.

[22] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, 1994.

[23] The Message Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface, http://www.mpi-forum.org/docs/docs.html, 1997.

[24] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *Fortran Forum*, 17(2):1–31, 1998.

[25] The OpenMP Organization. OpenMP: A Proposed Industry Standard API for Shared Memory Programming, http://www.openmp.org, 1997.

[26] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, 2006.

[27] C. Bell, W.-Y. Chen, D. Bonachea, and K. Yelick. Evaluating support for global address space languages on the Cray X1. In *Proceedings of the 18th ACMInternational Conference on Supercomputing*, Saint Malo, France, June 2004.

[28] The OpenMP Organization. OpenMP Fortran Application Programming Interface version 1.1,
http://www.openmp.org, 1999.

[29] The OpenMP Organization. OpenMP Fortran Application Programming Interface version 2.0,
http://www.openmp.org, 2000.

[30] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.

[31] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Comple te Reference*. MIT Press, Cambridge, MA, 1996.

[32] A. J. Wallcraft. SPMD OpenMP vs MPI for ocean models. *Concurrency: Practice and Experience*, 12:1155–1164, 2000.

[33] D. Bonachea. GASNet specification v 1.1. Technical Report UCB/CSD-02-1207, University of California at Berkeley, 2002.

[34] D. Bonachea. Proposal for extending the UPC memory copy library functions and supporting extensions to GASNet, v1.0. Technical Report LBNL-56495, Lawrence Berkeley National, October 2004.

[35] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. PGHPF – an optimizing High Performance Fortran compiler for distributed memory machines. *Scientific Programming*, 6(1):29–40, 1997.

[36] S. J. D. Bradford L. Chamberlain, Sung-Eun Choi and L. Snyder.

[37] T. Brandes. Compiling data parallel programs to message passing programs for massively parallel MIMD systems. In *Working Conference on Massively Parallel Programming Models*, Berlin, 1993.

[38] T. Brandes. Adaptor: A compilation system for data parallel Fortran programs. In C. W. Kessler, editor, *Automatic Parallelization— New Approaches to Code Generation, Data Distribution, and Performance Prediction*. Vieweg,Wiesbaden, 1994.

[39] P. G. Bridges and A. B. Maccabe. Mpulse: Integrated monitoring and profiling for large-scale environments. In *Proceedings of the Seventh Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Houston,TX, October 2004.

[40] D. Cann and J. Feo. SISAL versus FORTRAN: A comparison using the Livermore loops. In *Proceedings of Supercomputing 1990*, pages 626–636, NY, November 1990.

[41] D. C. Cann. The optimizing SISAL compiler. Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, April 1992.

[42] F. Cantonnet and T. El-Ghazawi. UPC performance and potential: A NPB experimental study. In *Proceedings of Supercomputing 2002*, Baltimore, MD, 2002.

[43] F. Cantonnet, T. El-Ghazawi, P. Lorenz, and J. Gaber. Fast address translation techniques for distributed shared memory compilers. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, Denver, CO, 2005.

[44] F. Cantonnet, Y. Yao, S. Annareddy, A. S. Mohamed, T. El-Ghazawi, P. Lorenz, and J. Gaber. Performance monitoring and evaluation of a UPC implementation on a NUMA architecture. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, 2003.

[45] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, May 1999.

[46] J. Caubet, J. Gimenez, J. Labarta, L. D. Rose, and J. S. Vetter. A dynamic tracing mechanism for performance analysis of OpenMP applications. In *WOMPAT '01: Proceedings of the InternationalWorkshop on OpenMP Applications and Tools*, pages 53–67, London, UK, 2001. Springer-Verlag.

[47] C. Coarfa. *Portable High Performance and Scalability of Partitioned Global Address Space Language*. Doctor's Degree thesis, Rice University, Houston, Texas, Jan. 2007

[48] B. L. Chamberlain, S. J. Deitz, and L. Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *Proceedings of Supercomputing 2000*, Dallas, November 2000.

[49] D. Chavarr´ıa-Miranda and J. Mellor-Crummey. An evaluation of data-parallel compiler support for line-sweep applications. In *Proceedings of the Eleventh International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, VA, Sept. 2002.

[50] D. Chavarr´ıa-Miranda and J. Mellor-Crummey. An evaluation of data-parallel compiler support for line-sweep applications. *Journal of Instruction Level Parallelism*, 5, feb 2003.

[51] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation.
http://www-128.ibm.com/developerworks/power/library/pa-cellperf/, nov 2005.

[52] W. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grain

UPC applications. In *Proceedings of the 14th International Conference of Parallel Architectures and Compilation Techniques*, Saint-Louis,MO, 2005.

[53] W. Chen, A. Krishnamurthy, and K. Yelick. Polynomial-time algorithms for enforcing sequential consistency in SPMD programs with arrays. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*, number 2958 in LNCS. Springer-Verlag, October 2-4, 2003.

[54] W.-Y. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC compiler. In *Proceedings of the 17th ACM International Conference on Supercomputing*, San Francisco, California, June 2003.

[55] I.-H. Chung and J. K. Hollingsworth. Using information from prior runs to improve automated tuning systems. In *Proceedings of Supercomputing 2004*, Pittsburgh, PA, 2004.

[56] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-Array Fortran Performance and Potential: An NPB Experimental Study. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*, number 2958 in LNCS. Springer-Verlag, October 2-4, 2003.

[57] C. Coarfa, Y. Dotsenko, and J. Mellor-Crummey. Experiences with Sweep3D implementations in Co-Array Fortran. In *Proceedings of the Los Alamos Computer Science Institute Fifth Annual Symposium*, Santa Fe, NM, Oct. 2004. Distributed on CD-ROM.

[58] Cray Research, Inc. Application programmer's library reference manual. Technical Report SR-2165, Cray Research, Inc., 1994.

[59] A. Darte, D. Chavarr´ıa-Miranda, R. Fowler, and J. Mellor-Crummey. Generalized multipartitioning for multi-dimensional arrays. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, Apr. 2002.

[60] A. Darte, J. Mellor-Crummey, R. Fowler, and D. Chavarr´ıa-Miranda. Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, Apr. 2002.

[61] R. Das, P. Havlak, J. Saltz, and K. Kennedy. Index array flattening through program transformation. In *Proceedings of Supercomputing 1995*, San Diego, CA, December 1995.