History of Coarrays and SPMD Parallelism in Fortran

JOHN REID, JKR Associates and Rutherford Appleton Laboratory, UK
BILL LONG, HPE Inc., formerly Cray Inc., USA
JON STEIDEL, Intel Inc., USA

Shepherd: Barbara Ryder, Virginia Tech, USA

The coarray programming model is an expression of the Single-Program-Multiple-Data (SPMD) programming model through the simple device of adding a codimension to the Fortran language. A data object declared with a codimension is a coarray object. Codimensions express the idea that some objects are located in local memory while others are located in remote memory. Coarray syntax obeys most of the same rules for normal array syntax. It is familiar to the Fortran programmer so the use of coarray syntax is natural and intuitive. Although the basic idea is quite simple, inserting it into the language definition turned out to be difficult.

In addition, the process was complicated by rapidly changing hardware and heated arguments over whether parallelism should be supported best as an interface to language-independent libraries, as a set of directives superimposed on languages, or as a set of specific extensions to existing languages.

In this paper, we review both the early history of coarrays and also their development into a part of Fortran 2008 and eventually into a larger part of Fortran 2018. Coarrays have been used, for example, in weather forecasting and in neural networks and deep learning.

CCS Concepts: • Computing methodologies → Parallel programming languages; Distributed programming languages; • Applied computing → Physical sciences and engineering.

Additional Key Words and Phrases: Parallel programming, Fortran, Single Program Multiple Data (SPMD), High Performance Computing (HPC), Coarrays, Massively Parallel Processors (MPP).

ACM Reference Format:
John Reid, Bill Long, and Jon Steidel. 2020. History of Coarrays and SPMD Parallelism in Fortran. Proc. ACM Program. Lang. 4, HOPL, Article 72 (June 2020), 30 pages. https://doi.org/10.1145/3386322

Authors’ addresses: John Reid, JKR Associates and Rutherford Appleton Laboratory, UK, John.Reid@stfc.ac.uk; Bill Long, HPE Inc., formerly Cray Inc., USA, billlong@acm.org; Jon Steidel, Intel Inc., USA, jon.l.steidel@intel.com.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2020 Copyright held by the owner/author(s).
## Contents

| Section | Title                                      | Page |
|---------|--------------------------------------------|------|
| 1       | The Evolution of Fortran                   | 3    |
| 2       | Shared-Memory Machines                     | 3    |
| 3       | Distributed-Memory Machines                | 4    |
| 4       | The MPP Revolution                         | 5    |
| 5       | Get and Put Operations                      | 6    |
| 6       | Early Implementation                        | 7    |
| 7       | Coarrays in Fortran 2008                   | 9    |
| 7.1     | Introduction                                | 9    |
| 7.2     | Early Work on Coarrays in the Fortran Standard | 11  |
| 7.3     | Coarrays at the February 2006 Joint Meeting of WG5 and J3 | 12  |
| 7.4     | Signs of Controversy                        | 15   |
| 7.5     | The WG5 and J3 Meetings in 2008             | 16   |
| 7.6     | Completion of the Fortran 2008 Coarray Features | 18  |
| 8       | Coarrays in Fortran 2018                   | 20   |
| 8.1     | Introduction                                | 20   |
| 8.2     | The Requirements Phase, 2011–2012           | 20   |
| 8.3     | The Development Phase, 2013–2014            | 22   |
| 8.4     | Completion 2015–2018                        | 24   |
| 9       | Implementation History                      | 25   |
| 9.1     | Implementation in Fortran Compilers         | 25   |
| 9.2     | Implementation of Coarrays in C and as a Class in C++ | 26  |
| 9.3     | Implementation in Python                    | 27   |
| 10      | Conclusion                                 | 27   |
|         | References                                  | 28   |
1 THE EVOLUTION OF FORTRAN

When the original version of Fortran was introduced by IBM in 1956 it was a high-level computer language designed for what was at that time “High Performance Computing” (HPC), namely, the use of computers to solve extensive numerical analysis problems, mainly from physics or disciplines derived from physics. Apart from direct business applications (for which the COBOL language was later introduced in 1959, see Sammet [1981]), this was the main use of computers in the 1950s and 1960s. There was no World Wide Web, Facebook, or computerized financial trading. So there was no need for a label to identify HPC as a subdomain of computing. By today’s catalog of computing subdomains, Fortran was the original HPC language. It has remained the dominant HPC language ever since, and has become effectively a niche language for this field of computing. From the beginning, it was designed for efficient execution of the compiled code (see Backus [1981]).

Fortran has stayed relevant in the HPC arena by continuous evolution to accommodate developments in HPC hardware. Changes in Fortran intentionally lag the initial stage of a hardware evolution in order to avoid chasing passing fads. Vector processors appeared in computers starting in the 1970s, in recognition of the fact that many HPC applications involved numerical methods based on matrices and vectors, and that custom hardware to perform these calculations offered a performance boost to applications.

Fortran standardization was begun as a US National Standard [American Standards Association 1966] that was adopted in other countries. In 1988 the American Fortran Committee X3J3 (later known as J3 and now known as PL22.31) became deadlocked because the rules under which it operated required a 2/3 majority to make changes and about half the committee wanted a very modest revision while about half wanted a significant revision. The deadlock was broken by the ISO Fortran Committee WG5. Since then the committees have agreed that WG5 should decide on the strategy and choose the major features, but most of the detailed work should be done by J3.

Arrays were an integral part of Fortran from the start; whole arrays have always been allowed in input/output statements, see IBM FORTRAN manual [International Business Machines Corporation 1954, page 16], and were allowed as procedure arguments in the first Fortran standard [American Standards Association 1966] (which was the first standard for a programming language). The Fortran 90 standard [ISO 1991] added array expressions and assignments. Coarrays [Numrich 1997] are a natural extension of arrays for parallel processing. A coarray exists on all processors and another set of subscripts (known as cosubscripts) is used to identify the processor involved in a reference.

Parallel computing for HPC machines began in the 1980s. When J3 met in Los Alamos in August 1983, users there made a strong plea for parallel computing to be included in the revision that was then being planned for 1985. Unfortunately, the future of parallel computing was uncertain at that time, and no ideas were exposed that were thought likely to pass the test of time. Therefore no features for parallel computing were added to Fortran then.

2 SHARED-MEMORY MACHINES

During the 1980s, many HPC systems were parallel computers that used the shared-memory model. The processors executed asynchronously, each having access to shared memory. There were hardware locks to prevent more than one processor updating a data element at the same time. Codes were based on FORTRAN 77 [ISO 1978] and consisted of sequential parts that were executed on a single processor and parallel constructs in which the work was shared among processors. Nesting of parallel constructs was allowed.

---

1We will refer to it as J3 in this paper because it was known as J3 during most of the time covered by this paper.
PCF (Parallel Computing Forum) was an informal body hosted by Rice University and informally headed by Ken Kennedy. It attempted to define a standard programming paradigm for shared-memory parallelism. This effort began in the mid 1980s and in 1991 led to a formal extension [Parallel Computing Forum 1991] for FORTRAN 77 known as PCF. There was not widespread experience in parallel programming at the time, and much of the work in PCF was theoretical and academic. Vendors continued to enhance and tune their own proprietary models as they gained experience and understood what was required and gave good performance. Vendors also had concerns that PCF was not developed through a formal standards body in which any interested party could participate, and worried about anti-trust claims that non-participating vendors might make in the future. As talk of formalizing the standards process increased, Kennedy shifted his focus to the distributed-memory massively parallel systems under development and HPF (High Performance Fortran), see the end of Section 3.

PCF dissolved, and the ANSI X3H5 committee was formed to standardize a parallel model for shared-memory systems. The committee did not maintain a strong focus on existing or common practices, and often reopened debate about things previously decided, making little progress. Committee members became discouraged, and membership began to decline, until the committee disbanded.

By the mid 1990s, several vendors were producing shared-memory parallel systems, and the collective knowledge of how to program them was growing. The US national laboratories, notably Lawrence Livermore National Laboratory (LLNL), held a meeting in Berkeley, California, attended by several HPC vendors. LLNL made a strong plea that the HPC vendors and users adopt a common parallel model across architectures. Each vendor’s model had matured over the over the previous decade and the models shared many common elements, but their extensions differed. The concerns about anti-trust claims were again raised, but this time the vendors’ lawyers got involved. There result was the formation of the OpenMP Architecture Review Board (ARB) which is funded by members, and is the executive body of the OpenMP committee. The OpenMP effort was advertised in trade journals and trade shows as an open committee welcoming all interested parties. With weekly teleconferences, the OpenMP 1.0 specification [OpenMP Architecture Review Board 1997] was drafted standardizing basic parallel constructs in just over a year. The current release is 5.0 [OpenMP Architecture Review Board 2018], and provides partial support of Fortran 2008 [ISO/IEC 2010]. OpenMP has been widely adopted and implemented in compilers, see [OpenMP 2020].

3 DISTRIBUTED-MEMORY MACHINES

Another hardware mechanism for parallel computing that was developed in the 1980s involved distributed memory. Here a number of uni-processors are linked together in a network. They are usually all given the same program to execute (SPMD\(^2\)). They exchange data by message passing. One processor would send a tagged message and another would receive it using the same tag. These would need to be carefully synchronized to ensure that the message was available when it was needed without a long wait.

PVM (Parallel Virtual Machine), described by Sunderam [1990], was first written at Oak Ridge National Laboratory in 1989. It addresses this issue by providing a library of procedures that is available on all the processors to provide message passing and synchronization. The system was developed to Version 3 in 1993, but was supplanted by MPI (Message Passing Interface) which is an informal standard written by an international forum. Like PVM, MPI is a library of procedures but differs in that it does not allow for nodes in the system to have different architectures. Because of the overhead in PVM caused by this generality (encoding and decoding numeric data at the

\(^2\)SPMD stands for Single Program Multiple Data.
endpoints of transmissions), MPI performed better than PVM. With the widespread adoption of the IEEE standard for binary floating-point arithmetic [IEEE 1985] and two’s-complement for integer data, the generality of PVM became uninteresting and the better performance of MPI was valued. The latest version of MPI is 3.1 [Message Passing Interface Forum 2015]. It has grown over the years and now has good implementations on many platforms.

MPI has proved very successful. A reason for its success is that it is implemented as a module\(^3\) including a set of procedures and makes no special demands on the compiler. The procedures can be written independently of the compilers, though vendors need to take account of the specific hardware. The fundamental operation is the send/receive pair, \textit{MPI\_send}/\textit{MPI\_recv}, illustrated by the code fragment in Figure 1. Here \textit{comm} is a variable that holds information about a set

\begin{verbatim}
if (p==p1) then
    call MPI\_send (array1,size(array1),MPI\_real,p2,tag,comm,ierr)
else if (p==p2) then
    call MPI\_recv (array2,size(array1),MPI\_real,p1,tag,comm,status,ierr)
end if
\end{verbatim}

Fig. 1. The MPI send/receive pair.

of processors that are communicating with each other, \(p\) is an integer whose value identifies the executing processor within this set, \(p1\) and \(p2\) are integer values identifying two particular processors in the set, \textit{array1} is a real array whose value on \(p1\) is needed by \(p2\) in another real array called \textit{array2}, \textit{tag} is an integer whose value identifies the message, \textit{status} is a derived-type\(^4\) variable that holds information about the message, and \textit{ierr} is an integer error flag.

Another reason for the success of MPI is that the library is designed to work well with C and C++ (and now Python). Other than SHMEM (see the start of Section 6), there are not many alternatives for C/C++ programmers.

An alternative is to work with a single-threaded model. Arrays would be distributed over processors and the compiler would arrange for the computations to be performed as much as possible where the data are stored, using message passing only when necessary. The programmer would provide guidance over how best to distribute the arrays, but this would affect only the speed of execution not the correctness of the computation. Kennedy, Koelbel, and Zima [2007] reviewed these efforts and also carefully reviewed Fortran D, Vienna Fortran, and CM Fortran. In 1992, the High Performance Fortran Forum (HPFF) was formed to standardize this approach and the first version of the specification was published in 1993 as HPF 1.0 [High Performance Fortran Forum 1993]. A final revised version, HPF 2.0 was published in 1997. Implementation of HPF made high demands on the compiler, which meant that efficient execution was not available early on and enthusiasm waned in most countries. However, it was successfully used in Japan on the Earth Simulator, see Uno [2005]. HPF is easier to program than MPI, but much less flexible.

4 THE MPP REVOLUTION

In the 1990s, the limitations of small numbers of powerful processors sharing a common memory gave way to the massively parallel processors (MPP) common today. MPP systems are characterized by a collection of physical nodes with local memory and ever more advanced microprocessors.

\(^3\)A module is a program unit containing (or accessing from other modules) definitions that are to be made accessible to other program units.

\(^4\)A derived type is a type that is defined by the program and has components of intrinsic type or other derived types. The program may also specify operations and assignments for variables of the type.
The nodes are interconnected by an internal network used for transferring data between nodes. This design allowed for very large and theoretically powerful systems by increasing the number of nodes along with scaling up the network. Power was also obtained within the nodes by making them shared-memory processors.

In the mid-1990s Cray Research, then the dominant maker of powerful shared-memory vector systems, became convinced that MPP was the inevitable future and began investigating the field. The hardware produced was a series of systems starting with the Cray T3D, whose name was derived from the logical geometry of the internal interconnect network, a Torus in 3 Dimensions.

Concurrently, the compiler developers worked on understanding how to program these sorts of systems. They developed a system known as CRAFT, which was explained by Pase, MacDonald, and Meltzer [1994]. Execution would begin in SPMD mode with the program replicated to all the processors and PVM available for communication. A data object could be declared as private, so that each processor had its own object with its own value, as expected by PVM. Alternatively, an object could be declared as shared. A shared scalar would be held on a single processor while a shared array would be distributed by blocks of contiguous elements of a size declared by the programmer. Special directives for barriers allow the program to change to executing as a single thread, as in HPF. Work-sharing constructs can then allow the work to be shared across processors.

5 GET AND PUT OPERATIONS

Robert Numrich, a physical chemist at Cray, who was involved with programming models for large systems, looked at the existing models for parallel programming and realized the importance of a small set of elementary operations. These involved getting data from another processor’s memory or putting data into another processor’s memory, see Numrich, Springer, and Peterson [1994]. He disliked the data distribution directives of the single-threaded models, which put the burden of resolving remote vs. local references on the compiler, and found programming with message passing burdensome and error prone (see Figure 1). Seeking something simpler, he developed a model that allowed a processor to request data from another processor, a simple GET operation. This model had a simple barrier which allowed synchronization between processors. At the urging of colleagues, a PUT function was added to the model. The initial software platform was the mature FORTRAN 77 compiler, as Fortran 90 compilers were just being developed.

Two essential ingredients formed the eventual design. First, to directly get data from the memory of a remote processor (or put data into a remote processor), it was necessary to know the address of the data on that processor. FORTRAN 77 had a data structure called a COMMON block. It was a “common” set of data items accessible to multiple subprograms in a program. While not required by the FORTRAN 77 standard, almost all compilers accessed data in a COMMON block based on offsets from the starting address of the COMMON block. In a lucky coincidence, FORTRAN 77 compilers placed COMMON blocks starting at known static addresses. As a result, the instance of the program on each processor had the same address for any particular data item stored in a COMMON block. This feature made it trivial to determine the addresses of a data item in a COMMON block on a remote processor, as it was the same as the address of the corresponding item on the local processor. Memory layout such as this is referred to as symmetric memory.

The second ingredient involved a visit to the T3D hardware designers. The question Numrich asked was, on the surface, a simple one: if the address of an object on a remote processor is known, how do I access that data from my processor? The ability to do this underpinned most of the MPP

---

5We believe that this term originates from the fact that each individual coarray reference is from one image and to another and there is symmetry between the layouts on these two images.
programming models, so not surprisingly there was a positive answer. The hardware designers supplied a sequence of machine instructions that accomplished what was needed.

Thus enough was in place to write a simple GetPut library for the T3D. For example, the code fragment in Figure 1 is replaced by the statement

```fortran
if (p==p1) call put(array1,array2,size(array1),p2)
```

or the statement

```fortran
if (p==p2) call get(array1,array2,size(array1),p1)
```

Note that only one of the processors has to execute the code and the programmer has to write the information only once. Testing showed it was significantly more efficient than the existing alternatives, see Sawdey, O’Keefe, Bleck, and Numrich [1994].

6 EARLY IMPLEMENTATION

In 1996, Silicon Graphics acquired Cray Research. MPI was the message-passing method of choice. Starting with Numrich’s GetPut library, Feind [1995] added functionality to create a one-sided communication library, SHMEM, for parallelism in a shared-memory hardware environment typical of SGI systems then being sold. Like MPI, it was designed to scale to a large number of processors, and like Numrich’s GetPut library, it was designed for one-sided communication. Numrich was not a fan of the added complexity of SHMEM, and intentionally went for simplicity.

Numrich proposed to incorporate the Get and Put operations directly into the Fortran syntax. There was little support for making such a change to the FORTRAN 77 compiler from Cray since it was being replaced by a new compiler that supported the recent Fortran 90 standard. So Numrich’s attention shifted to the new Cray Fortran 90 compiler. The front end of this compiler was attached to the back end of the SGI compiler to target SGI hardware.

One of the signature features of Fortran 90 was to allow whole arrays in expressions and assignments. The original COMMON block layout evolved into the concept of each processor having a corresponding array of exactly the same size. Syntax for accessing an array of another processor was just an extension of the ordinary array notation already present in Fortran.

The original design of Numrich [1997] used a vertical line to separate ordinary subscripts from subscripts in a processor grid. For example,

```
a(i,j|k,l)
```

was a reference to element \((i,j)\) on processor \((k,l)\). This was revised by Numrich and Steidel [1997b] to use square brackets for subscripts in the processor grid, for example,

```
a(i,j)[k,l]
```

and this notation has persisted.

For illustration, the code in Figure 1 becomes

```fortran
if (p==p1) array2(:)[p2] = array1(:)
```

or

```fortran
if (p==p2) array2(:) = array1(:)[p1]
```

Note that the programmer’s intention is expressed much more clearly.

The result was dubbed \(F^-\) by Numrich [1997]. The name \(F^-\) was intended to denote that this was a simple change to Fortran, in contrast to C++ which had been a significant complication of the C language. There was excitement about \(F^-\) within the Cray Fortran development team. Brian Johnson of that team implemented the \(F^-\) syntax into the compiler front end and translated it to calls to a GetPut/SHMEM-like library Karl Feind provided for the SGI Origin 2000 platform. Performance testing of \(F^-\) by Numrich [1997] showed speed gains over CRAFT (see Section 4) by
factors of between 2 and 4. Jon Steidel provided the support within management that enabled these developments to take place.

Numrich invented the term co-array or coarray for the distributed object consisting of the collection of corresponding arrays (or scalars) on all the processors, but it was not used in the papers describing F++. There, each processor has access to data on a single remote processor using scalar or array syntax, including array sections. During the standardization of coarrays, it was realized that the original design was better, see item (2) in the list in Section 7.2 and the final paragraph of Section 7.3. Furthermore, there were language problems over exactly what was meant by a “coarray”; for example, was a subobject a coarray if it involved a single processor? The standards use the term “coarray” for an object on a processor that may be addressed from another processor by using cosubscripts in square brackets and we will use it with this meaning in this paper.

An important element of F++ that has been maintained in all versions of coarrays in Fortran is that the local coarray may be addressed without cosubscripts, as we did in the example earlier in this section. While cosubscripts that specify the executing processor are allowed, the form without cosubscripts is recommended. This tells both the reader and the compiler that no remote access is involved. The key to efficient execution is that as much computation as possible is done locally, that is, without cosubscripts.

Early papers on F++ talked about the program being replicated on processors. Numrich and Steidel [1997b] mentioned images instead of processors. This is in recognition that replication need not be to processors, though it often is. Instead, it may be to threads, processes, nodes, or some other mechanism. It might involve more than one image executing on a processor, useful for testing a code. The term has stuck and we will use it henceforth in this paper. It does accurately express the fact that the program is replicated a number of times, usually as an executable code. Each image has its own set of variables.

The standards have always retained the property of symmetric memory, introduced in Section 5, which is that an image can always calculate the address of a coarray on another image (it is usually the same as that of the corresponding coarray on itself).

It has always been assumed that the number of images is fixed during program execution. Originally, this was the number of hardware processors. To allow the program to be tested on a small number of images but then run on more, the number of images is not fixed at compile time. Rather, it is specified when the program is initiated and is available to the code by a function call, num_images() in the standard. The index of the executing image is available as this_image() and lies in the range 1 ≤ this_image() ≤ num_images().

Synchronization is the responsibility of the programmer. F++ provided a simple barrier and synchronization with a single other image or a set of images. In the standard, these became the statements sync all and sync images(list), where list is an integer holding an image index or an integer array holding image indices. For example, the code in Figure 2 arranges for the value of a to

```fortran
if (this_image()==1) read (*, *) a
sync all
a = a[1]
```

Fig. 2. Barrier synchronization.

---

6In 2008, Malcolm Cohen [08-125] suggested that hyphens be removed from all the coarray words starting “co-”, such as “co-array” and “co-rank”. Just as “cosine” and “cotangent” are not hyphenated, it seemed appropriate not to hyphenate “coarray”. Robert Numrich was consulted and agreed, but he now prefers the original hyphenated form co-array and uses it in his book [Numrich 2018].
be read on image 1 and copied by the other images. The sync all is needed to ensure that the other images do not try to access the value before it has been read. Each image continues with its work without waiting for the other images to receive the value. To require all images to wait until they receive the value, another sync all statement would be needed at the end of the code fragment. If the value is to be used only on images 1 to 16, the sync images statement is suitable, see Figure 3.

```fortran
if (this_image()==1) then
    read (*, *) a
    sync images ( [(i,i=2,16)] )
else if (this_image()<=16) then
    sync images ( 1 )
    a = a[1]
end if
```

Fig. 3. Synchronizing some of the images.

The article by Numrich and Steidel [1997a] in SIAM News sparked John Reid’s interest. It struck him as a much clearer way of expressing parallelism than anything else he had seen. For illustration, compare the code in Figure 1 with the statement

```fortran
if (p==p1) array2(:,p2) = array1(:)
```

However, there were several aspects of the definition of F−− that Reid did not quite understand or that he felt had not been specified clearly or precisely enough. This led to a collaboration resulting in a paper [Numrich and Reid 1998] that was taken for several years as the definition of coarrays in Fortran. This paper contains several examples illustrating the utility of coarrays. It was in this paper that Numrich and Reid pointed out that for code between synchronizations, the compiler is free to use all its normal optimizations as if only one image were present. Note that this includes holding remote data in registers and writing changed values back to their home image only at the next synchronization. Reid suggested at this time that a non-allocatable\(^7\) coarray be declared with an asterisk in its last codimension, for example

```fortran
real :: coarray2(n)[m,*], coarray1(p)[*]
```

This mirrors the syntax of an assumed-size array, whose size is determined by the size of the corresponding actual argument. For a coarray the cosize is determined from the environment at run time.

Numrich, Reid, and Kim [1998] compared coarrays in Fortran with MPI for a multigrid solver on a shared-memory machine (ORIGIN 2000) and a distributed-memory machine (T3E). They found quite similar timings for the coarrays and MPI versions, but with coarrays scaling a little better as the number of processors increased.

### 7 COARRAYS IN FORTRAN 2008

#### 7.1 Introduction

John Reid presented a tutorial on coarrays at the 1998 meeting of the ISO Fortran Committee WG5 in Trollhättan, Sweden, and was well received. However, a long list of features had already been accepted for the next revision of the Fortran standard, which was intended for completion in 2000, so he did not propose the inclusion of coarrays.

\(^7\)An allocatable array or coarray is declared with a given number of dimensions and codimensions but its bounds and cobounds depend on the execution of allocate and deallocate statements. A non-allocatable coarray has bounds and cobounds that do not vary during its lifetime.
Work on the revision was completed in 2003. It became known as Fortran 2003, although the standard was actually published in 2004 [ISO/IEC 2004]. It was a huge revision. Its size can be seen from its description by Metcalf, Reid, and Cohen [2004] where Fortran 95 occupies the first 218 pages and the Fortran 2003 additions occupy a further 128 pages. It became apparent later that allowing the revision to be so large was a big mistake because it led to a long delay in the availability of compilers. Some features, particularly parameterized derived types and derived-type i/o were costly to implement in compilers and not much requested by users. The size of Fortran 2003 both delayed the start of consideration of coarrays and seriously delayed their implementation.

In 2004, the Fortran committees began considering the content of the next revision. They were determined that the revision would be small. By 2004, parallel computing was prevalent, even on personal computers, and the Fortran committee felt under pressure to include a feature to support it. It was 21 years since such a feature had been requested by users at Los Alamos (see Section 1).

Coarrays were well established in the Cray compiler. Implementation was not enormously difficult and users were happy with their experience. However, users were wary of using coarrays for major codes because of the lack of portability. Other vendors were wary of implementing a Cray extension that had not been standardized. In an initiative led by John Reid, the UK proposed the addition of coarrays. Reid was a coarrays enthusiast with no personal commercial interest in them; he was encouraged by Bill Long of Cray. Long was a determined advocate of coarrays because of the considerable demand from Cray users who saw the advantages of coarrays and wanted to be able to use them in their code without loss of portability.

Despite the wish to keep the revision small, there was a strong consensus within WG5 at its meeting in May 2005 in Delft, Netherlands to include coarrays. The straw vote was Yes-16, No-2, Undecided-1. The size of revisions were judged on an 8-point scale from 1 meaning "minor editorial change requiring less than 10 lines to be altered" to 8 meaning "technical change likely to need more than 3 years to develop". Coarrays were rated 6 meaning "technical change likely to need more than a year to develop" and no other feature rated 6 or more was accepted. It ended up by being the main change between Fortran 2003 [ISO/IEC 2004] and Fortran 2008 [ISO/IEC 2010].

John Reid was Convenor of WG5 from 1999 to 2017, the entire period in which coarrays were developed within Fortran 2008 and Fortran 2018. The main task of the Convenor is to secure consensus, though chairing meetings and providing leadership is included in the role. “Consensus” does not mean unanimity—there are times when a small number of people are so adamantly opposed that the only way forward is to agree to differ. He says “I did my best to separate the role of Convenor, which required me to balance the interests of those in favour and those against coarrays, from my role as an advocate for coarrays. Indeed, I sometimes declared that I was now wearing my Convenor’s hat or my coarrays’ hat. I am very proud of the fact that I managed to achieve consensus throughout the period. Mostly, this was easy, but at times it was hard.”

By ISO rules, all votes within WG5 are straw votes to indicate the extent of consensus. Formal voting is by country at the level of the parent committee, ISO/IEC JTC1/SC22. In practice, it is the people who attend WG5 that largely determine these country votes, so sometimes the Convenor took straw votes by country to judge how future formal votes at the level of SC22 might go. For most of the period considered in this paper, proposals for additions in the early stages of the development of a revision of the standard were made by countries, so that there was an element of filtering applied before WG5 considered the suggestions.

---

8The intrinsic types are parameterized by integer constants known as "kind" and "len" values. For example, the kind value for reals specifies the precision and the len value for characters specifies the length. Fortran 2003 extended this idea to derived types.

9Edit descriptors allow the programmer to control the output format for data of intrinsic type. Fortran 2003 allows programmers to specify the output format for data of a derived type by procedures associated with the type.
7.2 Early Work on Coarrays in the Fortran Standard

As proposed by Reid, coarrays were little changed from the definition of Numrich and Reid [1998]. However, there was intense discussion of its features at the 2005 meeting of WG5 as the experts on the committee began to think seriously about the functionality and ease of implementation of the features. There were no serious differences of opinion and many changes were agreed. A revised definition, noting all the changes, was written by Reid [N1642]. Numrich and Reid [2005] provided a new description, which ended with an appendix containing a summary of the changes. The major changes were:

1. Rather than limit both the number of subscripts (rank) and the number of cosubscripts (corank) to 7, the total rank and corank was limited to 15. This was to allow a large rank—WG5 did not anticipate that a large corank would ever be needed because the maximum array size was usually very much greater than the number of images.

2. Cosubscripts were limited to scalars to simplify the extension. The compiler writers on the committee agreed that there would be serious implementation problems over calculating a distributed array expression and assigning a distributed value; it would not be efficient for such a calculation to be performed by a single processor, but very hard to see how an implementation could in general orchestrate the activity of all the processors involved. WG5 thought that this limitation could be removed later, when there would be time to consider all the ramifications fully, but there has been no demand for it.

3. For the sake of symmetric memory (defined in the middle of Section 6), a coarray dummy argument such as coarry in the code

```fortran
subroutine work (m,n,coarry, ...) 
real coarry(m)[n,*] : 
```

was required to have the same bounds and cobounds (m and n in the above example) on all executing images. It was realized that this restriction was unnecessary. Provided the corresponding actual argument is a coarray, the image can properly interpret any reference to the remote object. The restriction was removed, which allows different execution threads to be performing quite different actions.

4. A derived type had not been allowed to have a coarray component because that would have led to references such as z[p]%x[q], which can be read as "go to image p and find component x on image q" and this can be written (more clearly) as z[q]%x. There was much discussion of this, led by Aleksander Donev, then a Ph.D. student at Princeton University, who wanted a more convenient way to express parallelism in his fluid flow calculations. He felt that there should at least be an ability to group coarrays and other objects into a single object. Others were concerned about the potential increase in the size and complexity of the language and the burden on implementers. It was decided to allow allocatable coarrays as components of derived types, but not allow an object of the type to be an array or a coarray.

5. To avoid references with coarray syntax to data that is not in a coarray, Numrich and Reid [1998] restricted pointer components of coarrays to the functionality of allocatable components, that is, a pointer component could be allocated and deallocated on its own image but not pointer assigned. Had allocatable components been part of Fortran 95, Numrich and Reid would have allowed these as components of coarrays and disallowed pointer components. With the addition of allocatable components in Fortran 2003, pointer components of coarrays

---

10 We will reference WG5 papers here by "N" followed by four digits. They are available from the WG5 website [ISO/IEC JTC1/SC22/WG5 2020].
were freed of these restrictions and references with coarray syntax to data that is not in a coarray were allowed. However, pointer assignment was limited to local data to avoid references to remote data without the presence of square brackets.

(6) A new form of synchronization was added. It involved the subroutines called notify_team, ready_team, and wait_team, which enabled images to notify other images in a team\(^\text{11}\) of their progress, query the progress of other images in a team, or wait for other images in a team. Each team was identified by an integer array of image indices. These were soon replaced by the notify statement and two forms of query statement, which made their descriptions more self-contained allowed dynamic restrictions to be imposed to avoid race conditions.

(7) The collective functions co_all, co_any, co_count, \ldots were added to provide over the cosubscripts of a coarray the equivalent of all, any, count, \ldots for the subscripts of an array. For example

\[
\text{real } x[*], y \\
\quad : \\
\quad y = \text{co_sum}(x)
\]

gives the value \(\sum_i x[i]\) to the variable \(y\) on every image.

A draft set of edits for adding coarrays to the standard was prepared by Bill Long and John Reid [05-272r2]\(^\text{12}\) and presented to the meeting of J3 in November 2005. This was discussed at the meeting and some minor changes were suggested but it was deferred to the next meeting for adoption.

### 7.3 Coarrays at the February 2006 Joint Meeting of WG5 and J3

A new version [06-122] was prepared by Long and Reid for the joint WG5/J3 meeting in February 2006 in Fairfax, VA, USA. A notable change was the introduction of the term image control statement for any statement that might affect the execution ordering between images together with the concept of a segment to denote the sequence of statements executed by an image between its execution of two image control statements (or at the start or end of execution). This allows far more precise statements to be made about ordering of statements and the availability of remote data. The segments executed by a single image are ordered, making data changed in one statement available in a later statement. A pair of segments on different images may be ordered as a consequence of the execution of image control statements and the ordering of segments on individual images. For example, in the code of Figure 2, each image executes two segments separated by a sync all statement and each first segment precedes every second segment. Data altered in any segment must not be accessed in another segment if the segments are unordered. For example, if the sync all statement is removed from the code of Figure 2, the rule would be broken because there would be only one segment on each image and they would be unordered.

Robert Numrich attended the meeting in February 2006 and presented a tutorial [06-145]. He wrote [06-134]

“Six basic features define the essence of the model:

1. a syntax for data distribution,
2. a syntax for data communication,
3. a full barrier,

\(^{11}\)A team is a set of images that are collaborating in a calculation. For example, one team might be performing an atmospheric calculation while another is performing an oceanographic calculation.

\(^{12}\)We will reference J3 papers here by the year as two digits, followed by a hyphen and three digits, and sometimes characters labelling a revision. They are available from the J3 website [INCITS PL22.3 2020].
(4) a partial barrier between pairs of images,
(5) allocatable co-arrays, and
(6) allocatable components of co-array derived types.

Although these features may not address every problem encountered in parallel applications, they cover a very large percentage of current practice. We should be careful adding complicated features before we have solid implementations of the simple features and good evidence of what more is needed.

Numrich was concerned about five extensions:

(1) [06-120] The addition of a new atomic construct that specified a scalar variable of type integer, real, or logical, and whose body contained statements involving the variable to be executed atomically, that is, by one image at a time. An example is shown in Figure 4.

```fortran
atomic(x[1])
  if(x[1] <= num_images()) x[1] = x[1] + 1
end atomic
```

Fig. 4. An example of the suggested atomic construct.

(2) [06-121] The addition of a new collective construct that would execute the statements in its body collectively on all the images of a team. An example is shown in Figure 5. Numrich was opposed to this, saying [06-134] “The co-array model is a local model. It is deliberately designed so that the compiler is not required to perform global optimization, is not expected to perform global optimization, and is in fact prevented from performing global optimization. Adding anything to the model that requires the compiler to analyze global information (J3/06-121) destroys the local model, makes it difficult to implement, and deters the compiler from performing its more important responsibility for local optimization.”

```fortran
collective(*) ! * specifies a team of all images
  a[2:n-1] = (a[1:n-2] + a[3:n]) / 2
  a[1] = (a[n] + a[2]) / 2
  a[n] = (a[n-1] + a[1]) / 2
end collective
```

Fig. 5. An example of the suggested collective construct.

(3) [06-129] That coarrays of a derived type with an allocatable component that might vary in size from image to image be replaced by "ragged allocatable coarrays". One might be declared thus:

```fortran
real, allocatable, dimension(:)[*] :: ragged_coarray
```

and allocated independently on each image without synchronization.

(4) [06-130] That the "owner computes" semantics for the assignment statement be adopted. For example, the statement

```fortran
a[1] = a[1] + 1
```

would be equivalent to the statement

```fortran
if (me == 1) a[1] = a[1] + 1
```

(5) [06-132] That intrinsic functions be used to form and destroy a team.
None of these were adopted, mainly because the size of the coarray proposal was already of concern. However, within the extensions adopted for Fortran 2018 (see the list in Section 8.4) there were more atomics, collectives, and a team feature that included a derived type for holding data about a team. Numrich emphasized [06-135] the importance of allowing coarrays to be of a derived type with an allocatable component that is not a coarray. It is a simple way to allow access to remote arrays that do not have the same size and was not available in other models. A simple example is shown in Figure 6.13 Shterenlikht, Margetts, and Cebamanos [2018] show the utility of this when modelling fracture in heterogeneous materials.

Associated with the segment model was the concept of user-defined ordering. The image control statement sync memory was available to separate a segment into two segments on each image that executes it. The first segment on one image might then be ordered in a user-defined fashion with respect to the second image. Numrich [06-134] wanted to remove this concept and the sync memory statement because he thought that it was not needed given the addition of further image control statements. However, the committee saw it as an established part of coarrays in Fortran and kept it.

Numrich [06-134] wanted to remove the collective functions from the language because they could be provided in a library and added to the language later. However, the committee thought it was important to have efficient implementations of these and that neither defining them in the standard nor implementing them would be difficult.

Numrich also expressed reservations in [06-134] about notify/wait, see item (6) in the list in Section 7.2, and suggested their replacement by events. They were later removed, see item (3) in the first list in Section 7.5, and eventually replaced by events, see Section 8.2.

There was not time at this meeting (of WG5/J3 in February 2006) to reach agreement on all the edits to the standard, but coarrays were confirmed as a top priority item and an amended set of edits [06-122r2] was constructed by the HPC (High Performance Computing) subgroup of J3. The subgroup decided that a coarray should be an object on a single image rather than a collection of objects on all the images. This was an editorial change that led to considerable simplification in the descriptions. The collective functions were changed to be subroutines because they involve image synchronization and this is very awkward to define precisely within a statement that involves several collective function calls. A significant change the subgroup made was to add an opaque derived type image_team and a form_team collective subroutine to calculate values of this type for a set of images on all those images. It was intended to allow implementations to store information

Note that any text following the character ! in a line of Fortran is a comment.
in these variables, varying from image to image, about how best to communicate with the other members of the team. The improvement in performance when the number of images is very large was expected to be great, mainly by better distribution of the load on the network. The team in a sync_team statement, a collective subroutine call, or an open statement for a file was required to be specified by a scalar of type image_team instead of an integer array.

7.4 Signs of Controversy

Work continued on the edits between the February and May 2006 meetings, but reservations about coarrays were expressed by Malcolm Cohen, Miles Ellis, David Muxworthy, and Lawrie Schonfelder [06-165]. They felt that coarrays were of minority interest and untested in practice, so should be an optional part of the standard to allow those vendors whose primary markets were for single processors to avoid the burden of implementing features of little interest to their customers. This argument was strongly countered by Bill Long [06-180]. He wrote that the “minority interest” would grow rapidly to a majority, that coarrays had been tested extensively in practice and proven valuable, and that failure to mandate coarrays would defeat the central point of standardization which is to provide portability. The two points of view were discussed extensively at the May 2006 meeting of J3 and Long’s view prevailed in a straw vote (Yes-6, No-3, Undecided-2). The set of edits to the standard to add coarrays was revised [06-174r3] and it was agreed unanimously that it was time to incorporate them in the draft standard.

There followed four meetings of J3 between August 2006 and May 2007 at which much work was performed on the draft for the new standard. A large number of changes were made to coarrays. Of note are:

1. Input/output units are integers used to identify devices in input/output statements. They were global (only one instance for all the images) because the model was of a single input/output system. It was realized that this model did not make it necessary for the units to be global, that it was inefficient, and it was inconsistent with each image having its own instance of each variable. Instead, each image was made to have its own units.

2. The collective co_findloc was added for consistency with the intrinsic findloc, which was part of the draft standard and returned the location within an array of a specified value.

3. In most circumstances, a Fortran implementation is permitted to make a copy of an actual argument on entry to a procedure and copy it back on return. This may be desirable on efficiency grounds, particularly to allow the procedure to work on an array held in contiguous storage when the actual argument is an array that is not held in contiguous storage, for example the array section a(1:10:2). It was realized that if the dummy argument is a coarray, copy-in copy-out would prevent access to the corresponding coarray on another image. Restrictions were added to make it unnecessary ever to perform copy-in copy-out for a coarray dummy argument.

4. An object with a cosubscript, such as a(1:n)[i] (known as a coindexed object), was permitted to be associated as an actual argument with a dummy argument that is not a coarray. This is very important because it makes existing codes accessible for objects from other images. The code that is called will not know that a remote object is involved, so copy-in copy-out must be expected. The code will be passed the address of the copy and the result will be copied back on return.

5. Normal termination of execution was clarified to ensure that an image continues to support access by other images to its data after it has executed a stop statement. All images synchronize as they execute a stop statement and only after they have all done so actually cease to execute. The statement all stop was added for rapid error termination of all images.
(6) For the sake of symmetric memory (defined in the middle of Section 6), allocating or deallo-
cating an allocatable coarray required synchronization of all the images, so that they always
all had the same memory requirements. A restriction was added to allow the allocation or
deallocation of a dummy argument that is an allocatable coarray only if it is associated with
the same coarray on all images.

(7) Pointers were restricted to pointing to objects on the same image. Instead of trying to disallow
all copying of pointers from one image to another, it was decided to treat such a copy as
undefined.

Aleksander Donev played a large role in the development of coarrays, particularly at this time. His
summary [Donev 2007] of the rationale for coarrays explains the reasoning behind these decisions
in more detail.

In August 2007 there was a joint meeting of WG5 and J3 in London. The UK National Activity
Report [N1682] expressed concern that the new standard might be published before there were
any implementations of the existing standard and that it was not the small revision that was
intended. A strong majority of the UK panel proposed that the schedule for the revision be delayed
by two years and the revision itself be reduced by removing all of the big items (coarrays, bits, and
intelligent macros) and possibly some of the middle-sized items. This was countered by a strong
plea from John Wallin, a professor in Computational Science at George Mason University [N1689]
to retain coarrays because he expected parallel computing to become dominant in the near future
and saw the need for a language that can express parallel concepts easily. These papers promoted a
wide-ranging discussion.

After a long discussion of the schedule and features, a straw vote on keeping coarrays in the
standard was taken and the result was Yes-13, No-1, Undecided-2. On bits, it was decided to remove
the bits type but keep the intrinsics and some other features. It was decided to remove intelligent
macros from the new standard. For the schedule, there was sentiment for some delay, but not for as
much as 2 years. A revised schedule [N1693], involving a delay of 10 months was agreed, which
required holding an extra WG5 meeting in 2008. There was sentiment for reducing the size of the
cooray feature, but after careful consideration only the recently-added collective _co_findLoc was
removed.

7.5 The WG5 and J3 Meetings in 2008

WG5 and J3 met again in February 2008 in Las Vegas. Those attending began very split regarding
coarrays between those worried about the size of the language with likely delays on its implemen-
tation and those who felt that it was very important to include a feature for parallel computing.
Tutorials on the need for parallel programming were given by Dan Nagle (Chair of J3) and John
Wallin. At the London meeting in 2007 there were three abstentions in the country straw votes.
The convenor (John Reid) therefore invited the three heads of delegations to indicate what they
required in order to vote yes on the standard. He collaborated with two of them in writing papers
that set out their suggestions for reaching consensus. The paper from David Muxworthy and John
Reid [N1710] suggested removing coarrays from the draft standard and placing them in a Technical
Report (TR) with the intention that if coarrays were included in a future revision, they would be as
given there, apart from the correction of errors. Jim Xia and John Reid [N1712] suggested reducing
the size by limiting the corank to one and the lower cobound to one because this was the most
common case. They also suggested removing the ability to have an I/O device connected on a team
of images (team I/O). Masayuki Takata [N1706] suggested moving coarrays to another part of the
standard.
There was much discussion on the first day without reaching any conclusion. Taking team I/O out was favoured (Yes-8, No-2, Undecided-7) but reducing the corank to one was not (Yes-3, No-12, Undecided-2). There was no progress on the second day but on the third day the US delegation softened its position and suggested a compromise: keep the core features in the standard and develop a TR for the other features plus possibly some suggested by public comments. A straw vote on this (Yes-8, No-2, Undecided-7 by individuals and Yes-5, No-1, Undecided-0 by countries) was far more favourable than the straw votes on other suggestions for changes. This approach was adopted unanimously. It should be remarked that although the members of WG5 were very divided technically on this issue, personal relationships remained amicable and there was a huge sigh of relief when a solution that everyone could accept was found.

The set of features that were moved out of the standard [08-131r1] were:

1. All the collective subroutines. They could all be provided within a library, though they would probably not be as efficient as vendor-written code would be.
2. Teams and the features that require teams. These were still under active development and a far more satisfactory feature was eventually constructed for the TR, see Section 8.2.
3. The notify and query statements. These were image control statements that allowed one image to proceed with calculations while another is catching up. Segments on an image executed before the execution of a notify statement precede the segments on other another image that follow the corresponding query statement on that image. These were still under development and were eventually replaced by events, see Section 8.2.
4. Files connected on more than one image, except for the files preconnected as output_unit and error_unit. Parallel I/O for direct-access files had been developed but was felt not to be adequate. We discuss this further in Section 8.2.

Bill Long volunteered to edit the TR. The aim was for publication approximately a year after Fortran 2008, that is, in June 2011.

A very tight schedule for the editor to make a new version of the draft standard and have it checked by a small subcommittee was set and a CD (Committee Draft) ballot was held from March 20 to August 31.

A critique [08-126] of coarray features in the draft standard was sent to J3 by John Mellor-Crummey, Laksono Adhianto, and Bill Scherer of Rice University during the February meeting. They had been working independently on coarrays in an academic environment, see Mellor-Crummey, Adhianto, Scherer, and Jin [2009]. It was not possible to consider their paper during the meeting, but a response [08-167] was written by Bill Long afterwards.

The Rice critique convinced J3 that locks were needed in the core part of the standard to provide better control over access to data by more than one image. The feature was worked on by Aleksander Donev and Bill Long over two meetings and adopted [08-168r2] in August 2008. A lock is a scalar coarray variable of the derived type lock_type. It has one of two states: locked and unlocked and the only way to change its value is by executing the image-control statements lock and unlock. Each lock variable is successively locked and unlocked by a sequence of lock and unlock statements. The segment that precedes the execution on an image of one of these unlock statements precedes the segment that follows the execution on another image of the next lock statement in the sequence.

The Rice critique also convinced J3 that a generalized facility for performing reductions across images would be valuable and it was decided that the collective co_reduce should be included in the TR.
During the May and August meetings of J3, several glitches were repaired:

1. A noncoarray allocatable dummy argument must not correspond to a coarray actual argument because its allocation or deallocation would occur only on the executing image and thereby upset the symmetric memory property.

2. An intent(out) dummy argument must not be an allocatable coarray or have a subobject that is an allocatable coarray because it would be automatically deallocated on entry on the executing image, which would require the procedure to be called on all the images so that they could synchronize.

3. A procedure reference through a coindexed object, for example
   
   ```
   call x[j]%proc()
   ```

   was defined as taking place on the executing image. This may be hard to implement and is not what the programmer expects. It was decided to disallow it.

4. Image control statements were disallowed in a do concurrent construct, which was a new feature in Fortran 2008.

5. The value attribute was disallowed for a coarray or an object with a coarray ultimate component because it involves a temporary copy only on the executing image.

6. A coindexed object was not permitted to be polymorphic or to have a polymorphic ultimate component because of implementation difficulties.

These are typical of the detailed changes that needed to be made during the development of coarrays in the standard.

WG5 and J3 met in November 2008 in Tokyo to consider all the comments from the CD ballot. There was one no vote, from the UK. It expressed concern that there were no full implementations of Fortran 2003 and wanted the revision to be minor. Coarrays were a large feature with ramifications throughout the language. They should be developed separately as an optional feature, either as a Technical Report or as an optional part of the standard. Japan abstained because they were undecided, given the rejection of their wish to move coarrays to an optional part of the standard. The four other countries that were active in the Fortran standard approved, three with comments. Thus the meeting began with disagreement not unlike that at the February meeting. There were extensive discussions during the week about the differing positions, but it was eventually agreed that the compromise chosen at the February meeting was the only way forward.

Nick Maclaren was concerned about the memory model of coarrays in Fortran. In particular, he did not like the exclusion of volatile variables from the segment ordering rules and the need of them for user-defined segment ordering (see Section 7.3). It was decided instead to add the intrinsic subroutines `atomic_define` and `atomic_ref` for atomically defining integers of kind `atomic_int_kind` or logicals of kind `atomic_logical_kind`. Only calls of these subroutines for these kinds of arguments would be allowed to break the segment ordering rules. It was expected that hardware would often be available to support these atomic operations. Maclaren’s other concerns were addressed, including a clear definition of user-defined ordering of segments and the requirement that unordered segments be given equitable shares of resources to ensure that there is progress.

### 7.6 Completion of the Fortran 2008 Coarray Features

Further changes were made to the coarray features during the February 2009 meeting of J3, notably:

1. Many Fortran programs rely on a feature that dates back to FORTRAN 66 [American Standards Association 1966]: an array element actual argument may be associated with an array dummy

---

14Actually both Cray and IBM had nearly full implementations, see Chivers and Sleightholme [2008].
argument, which is interpreted as an association with all the array elements from the referenced element onwards. This had been allowed for a noncoarray array dummy argument and a coindexed actual argument, but the usual implementation device of passing the address would not work because it would be on another image, so copy-in copy-out of the subarray would have been needed. It was decided not to allow this.

(2) Rules were needed for output on files preconnected as output_unit and error_unit. It was decided that each image would treat these files as separate and that the system would be required to merge the records into a sequence for the physical device, in an order of its choice.

(3) The wording for describing variables of type lock_type was made precise and clear.

While the magnitude of the changes was diminishing, the view taken by a majority in the UK and by the Japanese Fortran group was that it would be premature to proceed to Final Committee Draft (FCD) ballot while some coarray features are still actively being developed and that it would be inappropriate to proceed to FCD before the full Fortran 2003 language has been exposed to a wider range of users. Thus the May 2009 meeting of WG5 and J3 in Las Vegas began on a note of disagreement. In the end it was agreed that the whole schedule would be delayed to allow one more J3 meeting to hone the draft, but that additional meeting was explicitly disallowed from making any further changes to the technical content.

The May 2009 meeting of WG5 and J3 did make some further changes, notably:

(1) To allow for the case of a coarray having a noncorray component of type lock_type.

(2) Require the atom argument of atomic_define or atomic_ref to be a coarray or coindexed object because there is no need for these subroutines for other objects.

The changes made at the August 2009 meeting of J3 were all very minor and the draft was sent for FCD vote, which took place between August 29 and December 29. The result was approval, but the UK voted "no" and there were three "yes with comments" votes (USA, Japan, Canada). WG5 was instructed to take the comments into consideration and provide a revised text for Final Draft International Standard (FDIS) ballot. WG5 and J3 met in February 2010 again in Las Vegas. One of the UK reasons for the "no" vote was again to move coarrays out of the base standard. This was not discussed afresh. Most of the changes made in response to the ballot were minor but these changes are of note:

(1) The all stop statement was renamed error stop to recognize that it is very different from normal termination with stop when the stopped images continue to support access to their data by other images.

(2) Text was added on the behaviour of the intrinsics random_seed and random_number on multiple images. If random_seed is called in a segment A, and either random_seed or random_number is called in segment B, then segments A and B must be ordered. It is processor dependent whether each image uses a separate random number generator, or if some or all images use common random number generators. On images that use a common generator, the interleaving of calls to random_number in unordered segments is processor dependent.

The FDIS ballot was held between 1 July and 1 September 2010. All votes were in favour and the standard was published on 6 October 2010. A summary of the core set of coarray features that were included in Fortran 2008 was written by Reid [N1824]. These features have been used, for example, in weather forecasting [Mozdzynski, Hamrud, Wedi, Doleschal, and Richardson 2012], neural networks and deep learning [Curcic 2019], atmospheric research [Rasmussen, Gutmann, Friesen, Rouson, Filippone, and Moultitas 2018], direct numerical simulation of turbulence [Ashby and Reid 2008], Computational Fluid Dynamics [Garain, Balsara, and Reid 2015], and modelling fracture in heterogeneous materials [Shterenlikht, Margetts, and Cebamanos 2018].
8 COARRAYS IN FORTRAN 2018

8.1 Introduction

WG5 decided in 2008 to construct a Technical Report on the coarray features that were removed from the standard then (see the first list in Section 7.5) with a promise that they would be included in the next revision, subject to correction of any errors that became apparent. Work did not start in earnest until 2011, by which time ISO/IEC had decided to rename Technical Reports as Technical Specifications (TSs) without changing their purpose. The first draft [N1858] contained exactly those features that had been removed in 2008, but various suggestions had been made for other features by Long, Numrich, Bader, and Xia, see [N1835], and by Mellor-Crummey and coworkers at Rice University [N1856]. WG5 decided that the overall complexity of the TS (from the point of view of both implementation and edits to the standard) should be as planned in 2008, but the public should be invited to comment on the technical contents.

8.2 The Requirements Phase, 2011–2012

At the J3 meeting in October 2011 all the suggestions were discussed, including new suggestions [N1883] from the invitation to comment. These discussions were mainly in the HPC subgroup and the conclusions are summarized in the first coarray requirements document [11-256r2].

Bill Long suggested [N1835] that the number of collectives be reduced to those most wanted by users: co_bcast, co_max, co_min, co_reduce, and co_sum. The collective co_reduce was new; it behaves like co_sum but provides generality by supporting a user-written function with two scalar arguments. The collective co_bcast, soon renamed to co_broadcast, was also new. It broadcasts a value to all images or all images of a team. For example, the code in Figure 2 can be rewritten

\[
\text{if (this_image() == 1) read (*, *) a call co_broadcast(a, 1)}
\]

which is simpler and better performing as it is designed specifically for this task and avoids having all the images trying to access image 1 concurrently, which creates network congestion near the node running image 1. The Rice group commented that “the full set of collectives supported by MPI seems worth considering” but this was out of tune with the wish to limit the size of the TS and was not accepted.

Bill Scherer from Rice University was actively involved in these discussions. He suggested important changes to the teams feature including:

1. The use of a block construct within which images indices should be relative to a team so that existing code that was written for all the images can run unchanged on the team.
2. A command to form subteams by splitting an existing team into parts.
3. An intrinsic to return the index of an image within a team.
4. An intrinsic to return the number of images within a team.

These ideas were based on the experience of the Rice Group on an extended version of coarrays in Fortran, see Mellor-Crummey, Adhianto, Scherer, and Jin [2009], and were all accepted eventually. The form subteam statement was an image-control statement that subdivided the current team, and the change team construct changed the current team to that already constructed by a form subteam statement.

The Rice Group suggested the addition of an event type and this was accepted. Events allow an action to be delayed on one image until one or more actions have been performed on other images. An image records that it has performed an action by executing an event post statement involving a scalar coarray variable of derived type event_type, known as an event variable. An
image executes an event wait statement involving the event variable if it needs to delay its action for one or more posted actions on other images.

We illustrate the use of events in Figure 7, which shows an outline of a multifrontal code for solving a sparse set of symmetric linear equations by using a tree. Such sets of equations occur in a huge range of applications, including structural analysis, chemical process engineering, optimal control, power flow, partial differential equations, and non-linear optimization. The structure of such a set of equations can always be represented by a tree, which is a graph in which every node is a “child” of a single “parent” node except for the “root” which has no parent. The root is at level 1, its children are at level 2, and so on. Work at a node can start after all the work of its children is complete and their data have been passed to it. Each node is assigned to an image and each image has a list of its nodes, ordered in decreasing tree level. For each node, array elements hold the number of children, an event variable, the image that holds its parent, and the position of the parent in that image’s node list. This allows the processing to proceed asynchronously.

```fortran
integer, allocatable :: node(:) ! Tree nodes that this image handles.
integer, allocatable :: nc(:) ! node(i) has nc(i) children.
integer, allocatable :: parent (:), sub (:)
! The parent of node (i) is node(sub(i))[parent(i)].
type (event_type), allocatable :: done (:) [:]
integer :: i, j, status
... ! Set up the tree, including allocation of all arrays.
do i = 1, size(node)
  if (nc (i) > 0) then
    ! Wait for children to complete
    event wait (done(i), until_count=nc(i), stat=status)
    if (status/=0) exit
  end if
... ! Process node, using data from children, if any.
if (parent(i)>0) then
  ! node is not the root.
  ... ! Place result on image parent(i) for node(sub(i))[parent(i)]
  ! Tell image parent(i) that this has been done.
  event post (done(sub(i))[parent(i)], stat=status)
  if (status/=0) exit
end if
end do
```

Fig. 7. Outline multifrontal code.

In [N1835] Bill Long asked for nine additional atomic subroutines, all of which existed in the Cray coarray implementation from the beginning as a result of specific customer demands. Four perform the operations add, and, or, and xor on two integers atomically, overwriting one of them. Another four perform the same operations while copying the overwritten value to another variable. The ninth subroutine compares two integers and gives one of them another value if they are equal. The Rice group requested the same nine atomics in [N1883].

The requirements document [11-256r2] was discussed further at the February 2012 meeting of J3. It was decided [12-136r2] that the scope should be limited, in keeping with the general purpose of a TS, which is to address circumstances “when there is an urgent market requirement for such
documents”. J3 recommended that the only features to be supported should be those of the previous four paragraphs plus parallel I/O as originally proposed for the standard. This included the idea of connecting a direct-access file on every image of a team. The images of a team were permitted to access distinct records asynchronously, but for data written by one to be accessed by another the images were required to execute flush statements and then synchronize.

For teams, it was agreed that while an image executes a statement it should be a member of exactly one team and access to variables on images outside the current team should not be permitted, but it was later realized (see Section 8.3) that this was too restrictive.

To maintain symmetric memory, defined in Section 6, it was decided that an allocatable coarray that is allocated before entry to a change team construct must remain allocated during its execution and an allocatable coarray that is allocated during its execution and remains allocated on leaving should be automatically deallocated then, even if it has the save attribute. This allows each image to hold its allocatable coarrays in a stack with those allocated in the initial team at the bottom and those allocated in nested constructs in successively higher parts.

These decisions were endorsed by WG5 at its June 2012 meeting in Markham, Canada, see [N1930], except that it was decided to remove parallel I/O for direct-access files. The feature was seen as inadequate and it was felt that it would be better to delay, but no comparable feature has been added since.

At the meeting of J3 in November 2012, full details of all five collectives were agreed [12-178r5] and discussions on teams and events were had. It was here [12-199] that the concept of continued execution in the presence of failed images was introduced. It was anticipated that a coarray program might execute on a huge number of images. While the likelihood of a particular image failing during the execution of a program would be small, the likelihood that one of them might fail could be significant. It was thought that this would require a small change to the teams feature. It was decided that once an image had failed it would continue to be regarded as failed for the rest of the execution of the program, that is, no attempt be made to support intermittent failure.

8.3 The Development Phase, 2013–2014

The teams and events features were polished at the J3 meeting in February 2013. Following this meeting a Draft TS (DTS) [N1967] was constructed and was the subject of an informal WG5 straw vote. It had been planned at the 2012 meeting of WG5 that the DTS would be constructed at the June 2013 meeting of WG5 in Delft, Netherlands, but the result of the ballot [N1971] showed that more work was needed. The main changes made to the DTS following the ballot were:

1. For the sake of performance, the synchronizations at the start and end of a change team construct were made to involve each of the new teams separately, so that they would not all have to wait for the slowest.
2. Syntax was added to make it possible to access data of any image of an ancestor team. Without this, it would be necessary to exit the change team construct, access the data, then re-enter the construct, which would be a slow process.
3. Optional syntax was added to specify cosubscripts relative to a given team, for example

   \[ a(i, j)[\text{ocean\_team}::k,l] \]

   (but changed later, see end of this subsection).
4. One of the two types for holding events was removed, keeping the one that leads to efficient execution.
5. Processors should not be obliged to support detection and reporting of failed images.
(6) Users should be allowed to provide the mapping of image indices from the team to the
subteams in the `form subteam` statement. A `new_index=` specifier was added to provide the
image index that the executing image will have in its subteam.

The plan at the June 2013 meeting of WG5 was for the DTS to be ready in December 2013 but
this was too optimistic. New drafts [N1996 and N2007] were constructed after the J3 meetings in
October 2013 and February 2014. They were the subjects of more informal WG5 straw votes, see
[N1999 and N2013]. There was no major change in functionality but some quite significant changes
in the syntax and semantics were made, notably:

(1) The term “subteam” was considered confusing because what was a subteam became a team
within a change team construct. It was decided to use “team” for both. For example, the
`form subteam` statement became the `form team` statement.

(2) A `get_team` intrinsic was added to provide the value of the team variable of an ancestor of
the current team, which could be the initial team.

(3) More consideration was given to continued execution in the presence of a failed image. For
example, `sync all` synchronizes the nonfailed images (unless there is some other error).

(4) The `fail image` statement was added to simulate the failure of an image, which allows a
recovery process to be tested.

(5) The executions of `event post` and `event wait` statements were defined to perform atomic
operations on the event variable.

Almost all the comments in ballots were considered and answered within J3 papers. Many edits to
the DTS were made.

The result of the informal WG5 straw vote [N2013] was Yes-6 (2 with comments), No-4, Abs-2.
All the comments were considered at the June 2014 meeting of WG5 in Las Vegas. The document
was clearly not ready yet to be a DTS and the target was moved to December 2014. Many changes
were made, notably:

(1) Most of the collectives had a nonoptional argument `source` and an optional argument
`result`. If `result` was present it was given the computed value and `source` was not changed;
otherwise, `source` was overwritten by the computed value. This was simplified by removing
the `result` argument. If the original value of `source` is needed after the call to the collective,
it can be copied to a temporary variable and the collective applied to the temporary variable
instead of `source`. An example is shown in Figure 8.

```fortran
real :: a[*], new_a[*]
 : new_a = a
 call co_sum (new_a)
```

Fig. 8. Keeping the original of a collective invocation.

(2) The presence of a failed image among the images involved in an image control statement was
deemed to be an error; the presence of a `stat=` specifier is needed for execution to continue.

(3) An ancestor team was sometimes identified by a team variable and sometimes by its depth of
nesting in change team constructs. For simplification this was changed to use a team variable
in every case. It was decided that it was sufficient for the `get_team` intrinsic to get the team
value only for the initial team, the parent team, or the current team.

(4) An optional `stat` argument was added to the atomic subroutines to allow execution to
continue if they accessed a failed image.
(5) The concept of a stalled image was added. This is an image that tries to access data on a failed image of a team other than the current team. For a stalled image, there is a transfer of control to the end team statement of the team construct of the team of the failed image. The assumption is that the whole of that team calculation has failed and a recovery process is needed.

December 2014 turned out to be far too optimistic a target for the DTS. More work was done at the October 2014 and February 2015 J3 meetings and further informal WG5 straw votes were taken after each without consensus being reached. The main changes to the DTS were:

1. Stalled images were made to support finalizations and deallocations that would have occurred if they had not stalled.
2. Later, the concept of stalling was removed altogether because implementation difficulties were foreseen. It was replaced by access to a data value on a failed image resulting in the provision of a processor-dependent value and continued execution.
3. The syntax for cosubscripts was changed to allow an optional stat= specifier, for example
   \[ a[i, \text{team=parent}, \text{stat=svar}] \]
   The presence of a stat= specifier gave the program the possibility of resuming execution after an error including accessing a failed image.
4. A stat argument in a reference to an intrinsic procedure or the variable in a stat= specifier was not permitted to be coindexed.

### 8.4 Completion 2015–2018

Another new draft TS [N2048] was constructed and was the subject of another informal WG5 straw vote, which was deemed by the Convenor to have passed with the result Yes-6, Yes, but recommend changes-3, No-3, Abs-2 [N2054]. The “No” votes mostly recommended changes that were felt to be too important for a vote of “Yes, but recommend changes”. The Convenor requested a subgroup led by the editor Bill Long and working by email to consider all the comments and prepare a revised version with no syntax changes as the DTS version. They responded to the comments [N2055] and constructed a new draft [N2056]. The changes were editorial in nature, but some were quite important, such replacing the term “team identifier” by “team number” and clarifying some of the definitions of terms. The new draft [N2056] was the subject of a DTS ballot 26 May to 27 July 2015 and the result [N2065] (by countries) was Approval:10, Disapproval:0, Abs:8, with three countries (Germany, Japan, UK) commenting. The most significant comment came from the UK: “The content of the TS should not be added to the full Fortran language until it there is evidence that it can be implemented to work portably and reliably on most of the parallel systems used by the Fortran community”, and was accompanied by the note “This comment does not reflect all opinion in the panel but it is strongly held by some members”. The remaining comments were editorial in nature.

All the comments were considered at the meeting of WG5 and J3 in London in August 2015. The responses are contained in [N2076] and a new version of the draft [N2074] was constructed. Another WG5 straw vote agreed, Yes:12, No:0, Abs:2 [N2081], that it was ready for publication. It was published on 19 November 2015. Note that this was more than four years later than had been planned in 2008 (see paragraph 3 of Section 7.5). Bill Long was formally thanked as Project Editor by WG5 for “producing the various drafts of the TS and for preparing it for SC22 ballot” [Resolution L3 in N2077]. His perseverance was notable—he made 11 versions of the TS.

The most important features of the TS were

1. Teams. With the change team construct, this was a far better feature than the team feature removed in 2008.
(2) The concept of supporting continued execution after image failure. This was not a coarray feature in Fortran 2008 and is important as the numbers of images in execution get huge.

(3) The collective functions. There were less of them than in the set that was removed in 2008, but the additions (co_broadcast and co_reduce) were important. For example, Curcic [2019] obtains good performance with a Fortran program that uses co_broadcast and co_sum as its only coarray features.

(4) Events, which provide a more satisfactory feature than the notify and query statements that were removed in 2008.

(5) Nine additional atomic subroutines.

In accord with the promise in 2008 (see Section 7.5), the contents of the TS were incorporated in the new draft standard for Fortran. There have been some technical changes made, but they are too minor to be worthy of mention here. The standard [ISO/IEC 2018] was published in November 2018 and is known informally as Fortran 2018. A summary of all the new features has been provided by Reid [N2161]. For an informal description of the whole language, see Metcalf, Reid, and Cohen [2018].

9 IMPLEMENTATION HISTORY

9.1 Implementation in Fortran Compilers

The Cray Fortran compiler has supported the evolving specification of coarrays for twenty years. However, many of their users were reluctant to employ coarrays because of having huge programs with a very long life. Portability was very important to them. They hoped for other implementations because they wanted their code to run on other vendors’ hardware.

The Cray compiler served as the demonstration implementation during the development of the Fortran 2008 and Fortran 2018 specifications of coarrays. Early versions used the [ ] notation for objects on a different image, but depended on library routines (such as sync_all) for what became statements (such as sync_all) in the final standard. In early implementations for the Cray T3E through Cray X2 systems, inline code was generated for remote references, as there was good native hardware support. With the Cray compiler’s support of commodity processors (AMD, Intel, ARM, . . .) an internal runtime library has been used to implement coarray features. Version 8.1, released in September, 2012, supported all of Fortran 2008, including the coarray features. At that point the default for the compiler’s -hcaf/-hnocaf option was changed to make recognition of coarray syntax (-hcaf) the default. The 2019 Cray Fortran compiler (version 9.0.1) provided some features of Fortran 2018, including events. Additional Fortran 2018 features are planned for late 2019, and a complete implementation is planned for 2020.

A free implementation of much of the Fortran 2008 coarray features was provided in g95 by Andrew Vaught in October 2008, running on Linux platforms including cygwin. He provided very impressive support of this until 2010 when he moved to PathScale. John Reid recalls more than once reporting a bug one day and having a corrected version the next day. A stable version was released in October 2012 and is available from Vaught [2012].

Intel recognized the importance of coarrays for parallel processing and HPC applications. The 12.0 release of ifort in 2010 was the first release of coarrays in an Intel product, and was built upon MPI. This release was a full implementation of Fortran 2008 coarrays. Subsequent releases 13.0 and 14.0 contained numerous bug fixes. Significant performance improvements followed in releases 16.0, and 17.0. The first Fortran 2018 coarray features, events and the coshape intrinsic, were part of the 19.0 release in September 2018. The 19.1 compiler release contains significant Fortran 2018 feature support. The new coarray features include the FAIL IMAGE statement, the
The Fortran compiler that is part of the open source Gnu Compiler Collection (GCC), gfortran, provided partial support on one image only from 2010 and supported most of the coarray features in Fortran 2008 on multiple images from the 5.1 release in April, 2015, see [GNU 2020]. The underlying runtime library associated with this compiler is OpenCoarrays sponsored by the Sourcery Institute, headed by Damian Rouson. This implementation of coarrays was largely the result of heroic efforts in 2014 by Tobias Burnus of Germany (for the changes to the gfortran compiler), and Alessandro Fanfarillo of Italy (who wrote most of the runtime library). The OpenCoarray library used MPI or GASNet [Bonachea and Hargrove 2017] as a low level transport layer. Fanfarillo, Burnus, Cardellini, Filippone, Nagle, and Rouson [2014] ran performance comparisons with the Cray and Intel implementations. Both were sometimes outperformed by gfortran. Currently, an OpenSHMEM layer replaces GASNet. Rasmussen, Gutmann, Friesen, Rouson, Filippone, and Moulitsas [2018] found that the OpenSHMEM version outperformed the equivalent pure MPI code on an atmospheric research model and continued to work with more than 2,000 processes, whereas the MPI back end did not. However, the MPI option is more often used, since it is likely to be already installed on the system. However, the gfortran implementation is significant because free versions of the compiler and runtime are available for a wide variety of computer systems. It now provides partial support for events, teams, collectives, and failed images of Fortran 2018. Fanfarillo, Garain, Balsara, and Nagle [2019] show how to assign some of the images as spares to be used to replace failed images. A team of fixed size implements a robust Jacobi iteration and they give performance figures.

The Fujitsu Fortran compiler for its supercomputers supports the coarray features of Fortran 2008, see [Fujitsu 2017].

On shared-memory machines, the 2020 version of the NAG Fortran compiler, Release 7.0, supports the coarray features of Fortran 2008 and many of the coarray features of Fortran 2018, see [NAG 2020].

IBM has not implemented coarrays into its compiler but has been actively monitoring the development of coarrays in the Fortran standard.

9.2 Implementation of Coarrays in C and as a Class in C++

The advent of parallel computing also prompted the C programming community to investigate changes to C for parallel programming, resulting in Unified Parallel C (UPC), which was first proposed by Carlson, Draper, Culler, Yelick, Brooks, and Warren [1999]. Like coarrays in Fortran, the basic model was SPMD. The concept of a “distributed array” over many processors was more general in UPC in that consecutive array elements could be distributed among processors in a round-robin fashion. While this was useful for a particular type of application, there was substantial interest in “block” distribution of data in a manner very similar to coarrays in Fortran. Eleftheriou, Chatterjee, and Moreira [2002] provided an implementation of coarrays in C++ for the IBM Blue Gene/L, except that the order of subscripts and cosubscripts was reversed. More recently [Johnson 2013] implemented Coarray C++ (CAC++), which has much of the nature of coarrays in Fortran, except that the roles of [ ] and ( ) in coarray references are interchanged, as C normally uses [ ] for array subscripting. So cosubscripts appear in ( ) in the CAC++ implementation. C++ has substantial flexibility in the ability to redefine parts of the language syntax, including ( ) following an array name. As a result, it is possible to implement coarrays in C++ only by writing a class library, and using the same runtime library for remote communication that is used for Fortran. This implementation does not require modification of the underlying C++ compiler. CAC++ is available.
as part of Cray’s C++ compiler. It has not been ported to other platforms yet, mainly because of the proprietary runtime library.

9.3 Implementation in Python

Rasmussen, Sottile, Nieplocha, Numrich, and Jones [2004] developed a Python module that provided coarray syntax that allows a Python programmer to address coarray data on a remote processor. An example of Jacobi iteration using the module shows good performance results.

10 CONCLUSION

We have described the early history of parallel programming and the realization of the importance of get/put operations by Robert Numrich in the early 1990s. This led him to invent the concept of a coarray (originally written co-array) to express these operations more clearly by allowing data on another processor to be addressed directly. To emphasize that it need only be a small addition to Fortran, Numrich originally called his Fortran extension $F^{-\infty}$.

The idea was taken up by Cray after realizing that the future of High Performance Computing lay in the use of large numbers of quite powerful shared-memory processors executing asynchronously and connected by a good network. For programming such a machine, PVM and MPI were available but programs using these libraries are hard to write and maintain.

Discussions on the inclusion of coarray features in the Fortran standard began in 2004. They had been delayed while the huge revision to Fortran 2003 was completed, now recognized by the committee as a big mistake because it took a long time for vendors to implement all the features. There were difficulties over integration of coarrays with the additional features of Fortran 2003 and controversy over how many new features to add, but agreement that coarrays were the way to provide support within Fortran for parallel programming. There was pressure to keep the feature small because Fortran had become so large, but there was considerable demand from users because they saw its advantages and wanted to be able to use it in their code without loss of portability. We have explained that a compromise was eventually reached, with a subset of coarray features included, together with a promise that more would be part of the next standard. It was later recognized that making Fortran 2003 so large was a mistake.

Included in Fortran 2008 were some key coarray features:

1. The concept of an image as a replication of the program that has its own data and executes asynchronously.
2. Accessing data on another image with coarray syntax and data on its own image without coarray syntax, for example
   \[ a[p] = \sin(b[q]) + c \]
3. The concept of symmetric memory: implementations able to arrange that one image would know the address of a coarray object data on another image.
4. Programmers are responsible for including image control statements such as `sync all` to avoid data races, guided by the concept of a segment to define the requirements.
5. Allocatable coarrays and allocatable components for dynamic data without loss of efficiency.

While there is the potential for compilers to optimize the execution in ways that are not available when calling a collection of procedures such as those of MPI, in practice the performance has been quite similar to that of MPI. The advantage lies in the clarity of expression which makes for much easier development and maintenance.
As promised in 2008, further features were added in Fortran 2018:

1. Teams.
2. Support for continued execution after image failure.
3. Collective functions.
4. Events.
5. Atomic subroutines.

Cray Fortran compilers have supported the evolving specification of coarrays for twenty years. A free implementation of much of the Fortran 2008 coarray features was provided in g95 by Andrew Vaught in October 2008, running on Linux platforms including cygwin. Intel has recognized the importance of coarrays for parallel processing and HPC applications and since 2010 has supported coarrays in its Fortran compilers. The Fortran compiler in the open source Gnu Compiler Collection (GCC), gfortran, has supported most of the coarray features in Fortran 2008 since 2015. The Fujitsu Fortran compiler for its supercomputers supports the coarray features of Fortran 2008. On shared-memory machines, the 2020 version of the NAG Fortran compiler, Release 7.0, supports the coarray features of Fortran 2008 and many of the coarray features of Fortran 2018.

REFERENCES

American Standards Association. 1966. American Standard FORTRAN X3.9-1966.
John V Ashby and John K Reid. 2008. Migrating a Scientific Application from MPI to Coarrays. CUG 2008 Proceedings (2008), 1–8.
John Backus. 1981. The History of Fortran I, II, and III. In History of Programming Languages I, Richard L. Wexelblat (Ed.). ACM, New York, NY, USA, 25–74. https://doi.org/10.1145/800025.1198345
Dan Bonachea and Paul H. Hargrove. 2017. GASNet Specification, v1.8.1. Technical Report LBNL-2001064. Lawrence Berkeley National Laboratory. https://doi.org/10.2172/1398512
William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. 1999. Introduction to UPC and language specification. CCS-TR-99-157 (May 1999). Available at https://www.researchgate.net/.
Ian D. Chivers and Jane Sleightholme. 2008. Compiler support for the Fortran 2003 Standard. ACM Fortran Forum 27, 2 (2008), 22–24.
Milan Curcic. 2019. A parallel Fortran framework for neural networks and deep learning. ACM Fortran Forum 38, 1 (2019), 4–21.
Aleksander Donev. 2007. Rationale for Co-arrays in Fortran 2008. ACM Fortran Forum 26, 3 (2007), 9–19.
Maria Eleftheriou, Siddhartha Chatterjee, and José Moreira. 2002. A C++ Implementation of the Co-Array Programming Model for Blue Gene/L. https://doi.org/10.1109/IPDPS.2002.1016489
Alessandro Fanfarillo, Tobias Burnus, Valeria Cardellini, Salvatore Filippone, Dan Nagle, and Damian Rouson. 2014. Open-Coarrays: open-source transport layers supporting coarray Fortran compilers. In Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models. ACM, 4.
Alessandro Fanfarillo, Sudip Kumar Garain, Dinshaw Balsara, and Daniel Nagle. 2019. Resilient computational applications using Coarray Fortran. Parallel Comput. 81 (2019), 58–67. https://doi.org/10.1016/j.parco.2018.12.002
Karl Feind. 1995. Shared Memory Access (SHMEM) Routines. CUG 1995 Proceedings (1995), 303–308.
Fujitsu. 2017. Advanced Software for the FUJITSU Supercomputer PRIMEHPC FX100. https://web.archive.org/web/2017/http://www.fujitsu.com/global/Images/primehpc-fx100-soft-en.pdf Also at NA http://www.fujitsu.com/global/Images/primehpc-fx100-soft-en.pdf
Sudip Garain, Dinshaw S. Balsara, and John Reid. 2015. Comparing Coarray Fortran (CAF) with MPI for Several Structured Mesh PDE Applications. J. Comput. Phys. 297, C (Sept. 2015), 237–253. https://doi.org/10.1016/j.jcp.2015.05.020
GNU. 2020. Fortran 2008 Features supported by GNU Fortran. https://web.archive.org/web/20191221152824/http://fortranwiki.org/fortran/show/Fortran+2008+status Also at NA http://fortranwiki.org/fortran/show/Fortran+2008+status.

IEEE. 1985. IEEE Standard 754–1985 for binary floating-point arithmetic. Institute of Electrical and Electronics Engineers, New York.
INCITS PL22.3. 2020. Website. https://j3-fortran.org
International Business Machines Corporation. 1954. Specifications for The IBM Mathematical FORmula TRANSlating System, FORTRAN. Available at https://www.computerhistory.org/collections/catalog/102679231.
ISO. 1978. American Standard programming language FORTRAN ANSI X3.9-1978.
Aaron Sawdey, Matthew O’Keefe, Rainer Bleck, and Robert W. Numrich. 1994. The Design, Implementation, and Performance of a Parallel Ocean Circulation Model. In Proceedings of the Sixth ECMWF Workshop on the Use of Parallel Processors in Meteorology. 523–550.

A. Shterenlikht, L. Margetts, and L. Cebamanos. 2018. Modelling fracture in heterogeneous materials on HPC systems using a hybrid MPI/Fortran coarray multi-scale CAFE framework. Advances in Engineering Software 125 (2018), 155–166. 
https://doi.org/10.1016/j.advengsoft.2018.05.008

V. S. Sunderam. 1990. PVM: A framework for parallel distributed computing. Concurrency and Computation Practice and Experience 2, 4 (Dec. 1990), 315–339.

Atsuya Uno. 2005. Software of the Earth Simulator. Journal of the Earth Simulator 3 (2005), 52–59.

Andrew Vaught. 2012. G95 Website. https://web.archive.org/web/20191210041340/http://www.g95.org/ Also at NA http://www.g95.org.