Postprint of Fortran' s Parallel Extensions

Authors: Su Le, Fang Shuangde, Huang Yuanjie

Date: 2017-03-10T00:00:00+00:00

Abstract

Fortran, as the world' s first general-purpose high-level programming language, has experienced over 50 years of development and evolution across multiple versions, and has been widely applied in scientific and engineering numerical computation, establishing itself as one of the most popular programming languages in the high-performance computing domain to date. Constrained by its era of invention, the original Fortran language did not support parallel computing. Consequently, to augment and enhance Fortran' s capabilities for parallel computing and ensure program portability, academia and industry have proposed numerous parallel extensions for previous Fortran versions across different eras, some of which have become formal or de facto standards and have achieved widespread adoption. This paper provides a comprehensive review of the parallel extensions of the Fortran language that have been widely supported and accepted by the industry since Fortran 77, along with their characteristics, implementation status, and application effectiveness.

Full Text

Preamble

Vol. 11 No. 1 Information Technology Letter Vol. 11 No. 1
Parallel Extensions to Fortran
Su Le, Fang Shuangde, Huang Yuanjie

Abstract

Fortran, as the world' s first general-purpose high-level programming language, has undergone over 50 years of development and multiple version evolutions. It has been widely applied in scientific and engineering numerical computing and remains one of the most popular programming languages in high-performance computing to date. Constrained by its era of invention, the original Fortran language did not support parallel computing. Consequently, to enhance Fortran'

s capabilities for parallel computing and ensure program portability, academia and industry have proposed numerous parallel extensions to previous Fortran versions across different eras, some of which have become formal or de facto standards and gained widespread adoption. This paper surveys the parallel extensions to the Fortran language since Fortran 77 that have received broad industry support and acceptance, along with their characteristics, implementation status, and application effectiveness.

Keywords: Fortran, parallel computing, programming

1.1 The Birth of Fortran

Before 1954, almost all programs were written in machine language or assembly language. At that time, programmers regarded creating an effective program as a complex and creative artistic endeavor, devoting most of their energy to overcoming various technical limitations of contemporary computers, such as the lack of index registers and floating-point operations. The "automatic programming" systems provided for programmers at that time primarily focused on overcoming these shortcomings. These systems allowed for floating-point instructions, index registers, and improved input/output instructions, essentially creating a "virtual computer" with different operation codes from the actual machine. While this virtual machine was easier to program than the actual hardware, all these early automatic programming systems incurred excessive overhead, typically reducing machine execution speed by 80% to 90%.

By around 1954, investment in programmers had already approached the value of the computers themselves. Moreover, one-quarter to one-half of computer usage time was spent on program debugging. Thus, programming and debugging accounted for three-quarters of a computer' s operational investment. As computers became increasingly affordable, this situation grew even more severe.

Driven by these factors, in late 1953, John Backus [1] proposed to his employer, IBM, the formation of a small research group to develop a more efficient and economical programming method to address the low programming efficiency of IBM' s 704 computer. The proposal was adopted by IBM' s then-boss Cuthbert Hurd, and the research group began work under Backus' s leadership. By mid-1954, they had produced a preliminary programming language specification with considerable functionality and flexibility, initially called the IBM Mathematical FORmula TRANslation System (FORTRAN). Since computers were primarily used for scientific calculations at the time, the language' s design goal was to translate programs rich in mathematical expressions into sufficiently efficient object programs at adequately low cost. The project was originally intended purely for internal IBM research, but after the release of the language' s interim report, it generated tremendous interest among IBM customers. Consequently, IBM decided to ensure that every customer purchasing a 704 computer could use the Fortran language on their machine. This version was called Fortran I.

1.2 Fortran Language Standardization Activities

Since its inception, Fortran has evolved through several developments, forming numerous versions [2~8]. Fortran I eventually spawned many different versions, with Fortran II, which appeared in 1958, being the most popular. Fortran II expanded Fortran I considerably (for example, by introducing subroutines) and was implemented on many machines. The subsequent Fortran III was never implemented on any computer. Fortran IV, which appeared in 1962, made some changes to the original Fortran, causing Fortran II source programs to not be directly usable under the Fortran IV compiler, creating language incompatibility issues and resulting in the coexistence of Fortran II and Fortran IV as two distinct programming languages.

Because Fortran met real-world needs, it spread rapidly. During its dissemination and use, multiple dialect versions inevitably emerged. The semantic and syntactic specifications of various Fortran dialects were not entirely consistent, causing great inconvenience to users who urgently desired a Fortran language that could be interchanged and used universally across different machine types. Consequently, Fortran language standardization became imperative. In May 1962, the American Standards Association (ASA, later renamed ANSI—American National Standards Institute, now NIST—National Institute of Standards and Technology) established a working group to undertake this task. In 1966, they officially published two American standard texts: Standard Basic Fortran X3.10-1966 (equivalent to Fortran II) and Standard Fortran X3.9-1966 (equivalent to Fortran IV).

Due to Fortran' s widespread international use, the International Standard Organization (ISO) published the ISO Fortran standard in 1972, namely *Programming Language Fortran ISO 1539-1972*, which was divided into three levels: Level 1 Fortran was equivalent to Fortran IV, Level 2 Fortran fell between Fortran II and Fortran IV, and Level 3 Fortran was equivalent to Fortran II.

Fortran IV (i.e., Fortran 66) was popular for over a decade, virtually dominating all numerical computing fields. Many application programs and libraries were written in Fortran IV. However, many compilers did not adhere to this standard, often ignoring it to implement useful features. Additionally, after the structured programming methodology was proposed, people began to feel that Fortran IV could no longer meet requirements. Fortran IV was not a structured language and lacked statements to directly implement the three basic control structures, often requiring GOTO statements in programs to achieve specific algorithms. Moreover, to enable portability of non-standard Fortran source programs, "preprocessors" were created that would read non-standard Fortran source programs and generate standard Fortran text, but such automatically generated Fortran programs were typically difficult for humans to read and understand.

In 1976, the American National Standards Association revised ANSI X3.9-1966 Fortran, essentially absorbing all effective features from various vendors and adding many new ones. In April 1978, ANSI officially published it as the U.S.

national standard, ANSI X3.9-1978 Fortran, called Fortran 77. In 1980, Fortran 77 was accepted as an international standard, *Programming Language Fortran ISO 1539-1980*. This new standard was not a common subset of various non-standard Fortrans but a self-contained new language. China's Fortran standard, which basically adopted the international standard (i.e., Fortran 77), was published and implemented in May 1983, with the standard number GB3057-82.

Fortran 77 was not yet a fully structured language, but by adding some structured statements, it could be used to write structured programs. Additionally, it expanded character processing capabilities, enabling Fortran to be applicable not only in numerical computing but also in non-numerical computing domains.

Because Fortran 77 had obvious limitations, ANSI began preparing to formulate the Fortran 8x standard in the early 1980s to introduce new functions and adapt to language development. Initially, to correspond with the previous standard, the assumption was that x = 8. However, because Fortran 77 had to be included as a subset while ensuring high program efficiency, the standardization work took more than a decade. Finally, in 1991, the new Fortran 90 standard ANSI X3.198-1991 was passed, with the corresponding International Organization for Standardization number ISO/IEC1539:1991. The new Fortran standard abandoned outdated strict source program formatting, improved language regularity, enhanced program safety, and provided greater functional expansion, making it a modern programming language that could adapt to modern programming concepts. To protect the huge investment in software development by Fortran 77 users, the entire Fortran 77 was included as a strict subset of Fortran 90.

With the rapid development of other programming languages, Fortran was no longer the only widely used programming language. However, although other programming languages might be more suitable for some special domains, Fortran still holds a strong advantage in scientific and engineering technical numerical computing. Its strong vitality lies in its ability to keep updating standards with the times, with each new text providing breakthrough progress in functionality. For example, Fortran 90 not only standardized existing languages but more importantly absorbed advantages from other languages. Therefore, although Fortran has a long history, it continues to develop with each passing day.

With the sudden emergence of giant computers (vector machines and parallel machines), a new High-Performance Fortran language (HPF) appeared. HPF is an extended subset of Fortran 90, primarily used for programming on distributed-memory computers to reduce the burden on users writing message-passing programs. The HPF-1.0 language definition was made at the 1992 International Conference on Supercomputing, with the official text published in 1993. In subsequent years, it was modified, redefined, and annotated at conferences, and the HPF 2.0 language definition was released in 1997. Fortran 95 incorporated many new features from HPF. Before Fortran 90, running programs on parallel machines required combining specialized vectorization subroutine libraries or re-

lying on Fortran compiler systems for automatic vectorization. After Fortran 90, programmers could purposefully control parallelization during programming.

After 2004, the Fortran 2003 standard supporting object-oriented programming was successively published, followed later by the Fortran 2008 standard.

2 Parallel Extensions Since Fortran 77

The development process of the Fortran language has been a gradual evolution alongside the rapid development of programming methodologies. The number of extensions directly related to Fortran standards and parallel computing is not large. Below is an overview of the ANSI/ISO Fortran standard development.

2.1 Fortran 77

Fortran 77 made important improvements over Fortran 66 in many aspects. Originally designed for numerical computing, Fortran 77 expanded character processing capabilities, enabling its application in non-numerical computing domains. Fortran 77 also added block IF statements, ELSE statements, END IF statements, etc., making programs more structured and readable. Additionally, Fortran 77 enhanced input/output capabilities and file processing, improving many parts of the Fortran 66 standard (such as allowing mixed operations of different variable types and values, allowing array lower bounds to be negative or zero, and allowing array subscript expressions to be any integer expression).

Specific extensions are as follows [4][9]: - CHARACTER data type (greatly expanded tools for character input/output and processing of character-based data) - IMPLICIT statement - IF block statements, with optional ELSE and ELSE IF clauses (providing improved language support for structured programming) - OPEN, CLOSE, and INQUIRE statements (to improve read/write (I/O) capabilities) - Direct access file read/write - PARAMETER statement (to specify constants) - SAVE statement (to preserve local variables) - Generic names for intrinsic functions - DO WHILE and END DO statements - INCLUDE statement - IMPLICIT NONE variable (for IMPLICIT statements) - Bit manipulation intrinsic functions (based on similar functions included in Industrial Real-Time Fortran (ANSI/ISA S61.1 (1976)))

The 1991 IEEE 1003.9 POSIX standard version provided Fortran 77 programmers with calls on POSIX systems, defining over one hundred functional calls for files, allowing access to POSIX-compliant process control, signal handling, file system control, device control, procedure pointing, and stream I/O.

Because this version successfully changed the Fortran 77 development process, it enabled originally slow and repetitive programming design to smoothly cope with rapid changes in the computer field.

2.2 PCF Parallel Fortran

PCF Parallel Fortran was a parallel extension submitted to the Fortran standardization committee in 1991 by Bruce Leasure et al. [10]. Based on Fortran 77, it added statements supporting shared-memory multiprocessor architectures, enabling programmers to specify the number of threads, designate private/shared data, lock resources, and control thread ordering.

For computation partitioning, it primarily added two constructs: PARALLEL DO and PARALLEL SECTION. PARALLEL DO is used to mark loops without dependencies, without guaranteeing the order of loop indices during execution. When synchronization control or communication is needed during parallel section execution, locking, specifying CRITICAL SECTION, and global EVENT are provided for implementation.

Notably, PCF Parallel Fortran did not require compilers to handle deadlocks, so programmers needed to consider potential deadlock situations caused by locks or events themselves. PCF Parallel Fortran was implemented on early machines such as the Cray T3E, but its extensions ultimately did not enter the Fortran standard.

2.3 Fortran 90

Fortran 90, the successor to Fortran 77, was officially released in 1992 after a long delay. Fortran 90' s modifications to Fortran 77 focused on programming methodology [5][11,12]: - Allowed free-format source code input and lowercase Fortran keywords - Introduced modules to group related procedures and data together, making them callable by other program units, including allowing restriction of access to specific parts of modules - Added RECURSIVE procedures - Greatly improved parameter passing mechanisms, allowing interface checking at compile time - Allowed user-defined interfaces for generic procedures - Allowed operator overloading - Introduced derived/abstract data types - Added new data type definition syntax to specify other attributes of data types and variables - Allowed whole-array (or array section) operations in expressions and assignment statements, thereby greatly simplifying programming for mathematical and engineering calculations. These features include whole, partial, and wildcard array assignments (such as using WHERE statements for selective assignment), array constants and expressions, user-defined array functions, and array constructors - Dynamic memory allocation could be achieved through the ALLOCATABLE attribute and ALLOCATE and DEALLOCATE statements - Introduced the POINTER attribute, pointer assignment, and NULLIFY statements to facilitate creating and manipulating dynamic data structures - Introduced the CASE structure to support multi-branch selection - Introduced EXIT and CYCLE statements for sequentially "breaking out" of normal DO loop iterations - Extended identifier length to a maximum of 31 characters - Added inline comments - Added user-controllable definable numeric precision - Added new and enhanced intrinsic procedures

The underlying model for parallel computing in Fortran 90 is data parallelism, which needs to be implemented through the use of data arrays, providing operations and intrinsic functions acting on those parallel arrays, with compilers optimizing for each specific hardware architecture. By definition, there is not much difference between "data parallelism" and the so-called SIMD[1] (Single Instruction Multiple Data) programming style. From some perspectives, the two terms almost mean the same thing: the programmer writes a single operation, such as "+" , and then the compiler executes it on multiple data blocks in as parallel a manner as the hardware allows.

Any parallel computing method that is not SIMD is generally called MIMD[2] (Multiple Instruction Multiple Data). A parallel programming language with MIMD characteristics would allow several different subroutines (acting on different parts of data) to be called and executed simultaneously. Fortran 90 has almost no MIMD structures. A Fortran 90 compiler might execute MIMD code on some machines by implementing some Fortran 90 intrinsic functions (such as pack or unpack), but this is hidden from the Fortran 90 user. Some extended versions of Fortran 90, such as HPF (High Performance Fortran), explicitly implement MIMD features.

2.4 High Performance Fortran (HPF)

As supercomputing systems have evolved, they have formed various system structures, such as distributed-memory MPP[3], shared-memory multiprocessor systems, vector machines, mainframes, and workstations. To fully utilize the capabilities of these different computer systems, programs need to provide more information than traditional Fortran and Fortran 90 programs. Prior to this, many research institutions had conducted extensive research on this topic, with Rice University' s Fortran D [13] language and the University of Vienna' s Vienna Fortran [14] language having the greatest influence. At the end of 1991, Ken Kennedy and Geoffery Fox suggested establishing an unofficial organization to further define this language and carry out standardization. Thus, an organization composed of industry and academia—the High Performance Fortran Forum [15]—was established. After more than two years of effort, a new Fortran language standard that could meet the above requirements—High Performance Fortran (HPF)—was finally introduced in 1993.

The goals of HPF [15~17] were to define a set of language extension standards for Fortran (mainly Fortran 90) to: - Support data-parallel programming design - Achieve maximum performance on SIMD or MIMD computers with non-uniform memory access overhead - Facilitate code portability between computers with different architectures

2.4.1 Essential Features The essential features of the HPF parallel model are single-threaded, global memory space, and loosely synchronous. Although HPF initially targeted NUMA machines, its description of data parallelism on non-uniform memory access machines could also guide compilers in generating

code for distributed-memory machines. When the compiler targets distributed-memory machines, HPF is compiled into SPMD programs.

New syntactic structures FORALL and intrinsic functions in HPF are used to support data parallelism, while data distribution directives are used to achieve high performance on NUMA machines. External functions can be used for architecture-specific low-level optimizations.

2.4.2 HPF Data Distribution Directives HPF has a set of compiler directives to help programmers express how data is stored. These directives do not affect computation results but do affect execution efficiency. Their introduction is based on two important but simple observations: - If the same operation processes multiple data objects, executing these data objects on the same processor will be more efficient - If operations execute simultaneously on different processors, overall execution efficiency will be higher

Using these data storage compiler directives, we can reasonably distribute data across multiple processors, and the compiler will perform data-parallel optimization to improve program execution efficiency with runtime support.

The main objects of operation in HPF are arrays. Compiler directives can divide data into equal-sized blocks, called BLOCK distribution, or align them with other arrays in a round-robin fashion, called CYCLIC distribution (or use more complex methods for array data partitioning). These data objects (arrays) are then aligned to some abstract templates, which are distributed across abstract node processors. The compiler maps these abstract processors to physical processors, thereby completing the entire data mapping process, as shown in Figure 1 [Figure 1: see original paper]:

Figure 2 [Figure 2: see original paper] shows the first line of HPF directive information that instructs the compiler to map A(K) and B(K-1) to the same virtual processor for any K. The second line of HPF directive information requires copying array C(I, 1:100) to the virtual processor storing any element of D(I, 1:100); data copying operations and data consistency guarantees are both handled by the compiler.

Figure 3 [Figure 3: see original paper] demonstrates two different DISTRIBUTE methods: block and cyclic distribution. If the target platform is a four-processor machine, the first processor will store the following portions of arrays: A(1:25,1:100), B(1:100,1:97:4), and C(1:5), C(21:25), C(41:45), C(61:65), C(81:85).

2.4.4 Other HPF Features HPF also provides some built-in functions for programmers to use, such as MAXLOC and MINLOC for finding indices of maximum and minimum values.

In terms of language feature modifications, Fortran 77' s sequential storage requirements conflicted with locality requirements in modern architectures. There-

fore, HPF eliminated sequential storage requirements unless specified by directives, allowing compilers to modify storage distribution. Meanwhile, compilers could use directive information to optimize data organization and distribution.

HPF is a high-level language independent of specific machines. To effectively support fine-grained optimizations such as systolic communication, it introduced an interface for external functions. HPF 2.0 also added HPF I/O extensions, though vendors could still define their own read/write (I/O) extensions.

2.4.5 HPF Implementation Status As HPF evolved from 1.0 to 1.1 and 2.0, more and more compilers supporting HPF became widely used, with more research institutions and vendors investing in it. The following are some HPF-supporting compilers: - Absoft Pro Fortran from Absoft Corporation - DIGITAL Fortran from Digital Corporation (no longer available) - Fortran 90 from Parsec Technology Corporation (no longer available) - Fortran Power Station 4.0 from Compaq Corporation (no longer available) - HPF Plus from NA Software Ltd - Limaoshan from the Chinese Academy of Sciences (no longer available) - PGHPF from Portland Group Corporation - VAST-HPF from Crescent Bay Software Corporation (formerly Pacific-Sierra Research Institute) - xHPF from Applied Parallel Research Institute (no longer available) - xlhpf from IBM Corporation - Adaptor from GMD-SCAI Corporation - G95 Fortran 95 compiler - SHPF from the University of Southampton and VCPC (Vienna Parallel Computing European Center)

Many important applications have also benefited from HPF, mainly concentrated in computation-intensive, highly data-parallel applications such as physics and mathematics: - Three-dimensional magnetohydrodynamics simulation - Biomembrane simulation - Computational accelerator physics simulation - Rice University' s HPFt project—Generic Crash Kernels for crashworthiness simulation - MCMC for carcinogenesis models in cancer process simulation - Neural networks for hydrocarbon net pay prediction - Princeton Ocean Model

2.4.6 HPF Application Performance Dale Shires et al. [18] compared the efficiency of HPF and MPI approaches using unstructured finite element simulations. The experiments used a Cray T3E supercomputer with the PGHPF 3.2 compiler. The T3E is a parallel distributed-memory platform using a tightly coupled 3D bidirectional torus configuration with 1,088 processing elements, each with 500MB of memory. The experimental results are shown in Figure 6 [Figure 6: see original paper].

In Figure 6(a), with 2 processors, the MPI program was 4.5 times faster than the HPF program; with 32 processors, it was 4.3 times faster. In Figure 6(b), with 2 processors, the MPI program was 3.8 times faster than HPF; with 128 processors, it was 2.7 times faster.

The experimental conclusion was that both HPF and MPI have good scalability for unstructured finite element applications, and both parallel programming

approaches are suitable for most parallel systems such as SUN, SGI, IBM, etc. HPF is a higher-level parallel programming abstraction, and achieving such execution efficiency is already very remarkable for HPF. Although there is still a gap compared to MPI application execution efficiency, HPF programming is greatly simplified compared to MPI, which significantly lowers the barrier to parallel program design and has greatly promoted the popularization of parallel systems.

Hitoshi Murai et al. [19] evaluated HPF application performance on an 8-processor machine using the HPF/SX V2 compiler across HPFBench [20], APR Benchmarks [21], GENESIS Benchmarks [22], and NAS Parallel Benchmarks [23]. Figure 7 [Figure 7: see original paper] shows the results on HPFBench, with the baseline being F90 without parallel extensions.

Figures 7(a)-(c) show evaluation results for linear algebra libraries; (d)-(f) show evaluation results for application kernels. From Figures 7(a)-(f), the overall results show that when using HPF extensions in a single-processor environment, performance slightly decreases or remains unchanged, while in multi-processor environments, HPF extensions can achieve good execution efficiency and scalability.

2.5 Fortran 95

Fortran 95 [6][12][24] was only a minor revision, with most changes correcting some more significant issues in the Fortran 90 standard. Nevertheless, Fortran 95 still had considerable expansions, especially in HPF specifications: - Added FORALL and nested WHERE structures to aid vectorization - Introduced user-defined PURE and ELEMENTAL procedures

An important supplement to Fortran 95 was ISO Technical Report TR-15581: Enhanced Data Type Facilities, informally known as the Allocatable TR. This standard defined enhanced applications of ALLOCATABLE arrays, appearing before fully Fortran 2003-compatible Fortran compilers, allowing users to use ALLOCATABLE arrays in procedure dummy argument lists and as derived type components in function return values. (ALLOCATABLE arrays are preferred over POINTER-based arrays because ALLOCATABLE arrays are guaranteed by Fortran 95 to be automatically deallocated when they go out of scope, avoiding the possibility of memory leaks. Additionally, aliasing is no longer an issue when optimizing array references, allowing compilers to generate code that runs faster than code using pointers.)

A second supplement to Fortran 95 was ISO Technical Report TR-15580: Floating-point exception handling, informally known as the IEEE TR. This standard defined support for IEEE floating-point arithmetic and floating-point exception handling.

2.6 Fortran 2003

Fortran 2003 [7][24~26] represented a tremendous change in Fortran' s development, introducing the then-popular object-oriented programming methodology. Major improvements included: - Enhanced derived types: derived types with parameters, improved controllability, improved structured creation and destruction - Support for object-oriented programming: extended types and inheritance, polymorphism, dynamic type allocation, and type-bound procedures - Improved data manipulation: support for allocatable components (incorporated into IEEE TR 15581), deferred type parameters, VOLATILE attribute (in concurrent systems, programmers can use this attribute to know whether shared data values have been correctly refreshed), support for explicitly defining types in array constructors and allocation statements, support for enhanced pointers, extended initialization expressions, enhanced intrinsic procedures - Enhanced input/output: support for asynchronous transfer, stream access, allowing users to specify transfer operations for derived types, allowing users to specify rounding control during format conversion, specifying constants for pre-connected units, using the FLUSH statement, defining keyword specifications and accessing error information - Support for procedure pointers - Support for IEEE floating-point arithmetic and floating-point exception handling (incorporated into IEEE TR 15580) - Support for interoperability with C language - Support for internationalization: access to ISO 106464 byte characters and selection of decimal points or commas in formatted numeric input/output - Enhanced integration with the host operating system: access to command-line arguments, environment variables, and processor error information

2.7 Fortran 2008

The version after Fortran 2003 was Fortran 2008 [8][25,26], which, like Fortran 95, was only a minor revision, slightly correcting some issues with Fortran 2003 and merging the language features of TR-19767: - Co-array Fortran –parallel processing mode - BIT data type

In August 2007, the BIT data type was removed. In February 2008, the Co-array Fortran plan was scaled back to only Parallel I/O, and the development team was downsized.

2.8 Co-array Fortran

Co-Array Fortran (abbreviated "CAF") [27~32] is a set of SPMD parallel extensions to Fortran 95. CAF' s main parallel objects are also arrays, and it only makes very simple extensions to the original Fortran 95 syntax, so the learning burden on programmers is minimal.

A Fortran program segment written using CAF appears as if it has been copied multiple times, with each copy executing asynchronously. Each program copy has its own data objects, called images. CAF mainly extends array indexing syntax, using square brackets for cross-image access.

CAF is an SPMD parallel programming model based on the Fortran 95 language. Similar to MPI, CAF programs need to explicitly manage data and computation distribution, but CAF is a shared-memory programming model that does not require explicit data communication management. It only requires using extended Fortran 95 array subscript syntax to access data on other processors, with data communication and synchronization handled and optimized by the compiler. CAF is now mainly applied in scientific and engineering computing, with extensive use in supercomputers, implemented primarily in Cray Fortran 90 compiler version 3.1 and later. Additionally, the Los Alamos Computer Science Institute (LACSI) laboratory at Rice University [28] has also implemented CAF.

2.8.1 Main Features of Co-array Fortran First is CAF' s work distribution: a program is copied multiple times, with each copy having its own set of data objects. Each copy is called an image, and multiple images execute asynchronously, so each image has a different execution path. Programmers use image indices and explicit synchronization operations to determine the actual path of a particular image. The compiler is responsible for optimizing CAF.

Second, considering data distribution, CAF extends Fortran language syntax, allowing programmers to use data subscript-style syntax to distribute and access data across multiple processors. For example:

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

In the above code, each image is declared to have two real arrays X and Y, each of size N. If Q has the same value on every image, the second statement means that the X array on each image copies the value of the Y array from image Q. The array subscript in parentheses represents indexing of array elements within an image. The array subscript in square brackets represents the index of the image where the array resides. Using extended array subscript syntax makes it easier for programmers to write code that accesses data on other images.

Since many operations on data objects in a program are most efficient when performed locally, CAF syntax should appear in a small, relatively isolated section. Otherwise, more CAF code usage indicates more communication needed between images. Here is a simple CAF example:

```
6.  S = MINVAL(Y[:])
```

In the above example, line 1 assigns X the value of Y from the PE-th image; line 2 assigns the value of X to Y on the PE-th image; line 3 broadcasts the value of X in the current image to Y in all images; line 4 assigns the value of X in the current image to Y in the group of images specified by LIST; line 5 assigns the values of Y from all images to Z in the current image; line 6 obtains the values of Y from all images, finds the minimum value, and assigns it to S in the current image.

Previously, input/output was a troublesome issue for SPMD programming models such as MPI, because standard Fortran input/output assumes a dedicated process to read files, a restriction that is often violated, especially when each image needs independent input/output. CAF avoids some limitations faced by previous programming models with minimal extensions to Fortran 95 I/O, thereby explicitly providing parallel read/write capabilities that can be implemented based on processes and threads.

Figure 8 [Figure 8: see original paper] is a complete CAF program example for finding the maximum value:

```
Subroutine greatest(a,great)
! Find the maximum value of a(:) [*]
real, intent(in)::a(:)[*]
real, intent(out)::great[*]
real :: work (num_images()) ! Local work array
great = maxval (a(:))
call sync_all ! Wait for all images to arrive
if(this_image(great)==1)then
work(:)=great[:] ! Get local maximum values
great[:]=maxval(work) ! Broadcast global maximum value
end if
call sync_all
End subroutine greatest
```

Figure 9 [Figure 9: see original paper] shows an improved version using allocatable arrays:

```
Subroutine greatest (first,last,a,great)! Find the maximum value of a( : )[first:last]
integer,intent(in)::first,last
real,intent(in)::a (:)[*]
real,intent(out)::great(:)[*] ! Place results in great [first:last]
real, allocatable::work(:) ! Local work array
integer::i,this
this=this_image(great)
if (this.GE. fitst.and.This .LE.last) then
allocate (work(first:last))
great=maxval(a)
call sync_team((/(i,i=first,last)/))
if(this. EQ.first)then
work=great[first:last] ! Get local maximum values
great[first:last]=maxval(work) ! Broadcast global maximum value
deallocate (work)
end if
call sync_team((/(i,i=first,last)/))
end if
end subroutine greatest
```

As shown in the code in Figure 8: initially, all images find the local maximum value of array a within their own image. Then synchronization occurs, waiting for all images to complete finding the maximum value. The first image then collects local maximum values from all images, finds the global maximum value, and distributes it to all images. The `call sync_all` on line 6 is a method provided by CAF for synchronization between images.

One issue here is that, as shown on line 4 of the code, each image allocates a local array named work to store collected local minimum values. In fact, only the first image needs the work array. We can use CAF's allocatable array feature to allocate the work array only in image 1. Figure 9 shows the improved code. As indicated by the gray fill box in Figure 9, the work array is first declared as allocatable, then allocated and deallocated within the code that image 1 will execute.

2.8.2 Co-array Implementation Status Co-array extensions have been implemented in many Fortran compilers and corresponding systems for a long time. For example, Cray Fortran [27] has included Co-array support since version 3.1. Through continuous development and improvement, Co-array extensions have been incorporated into the Fortran 2008 standard and are being implemented and supported in more and more systems. The first open-source Co-array implementation was in G95, which implemented Co-array as a Fortran 2008 standard in the compiler and related Linux runtime systems.

Additionally, some research institutions and companies are actively implementing compilers and runtime libraries for the CAF2.0 standard. For example, as of 2011, Rice University was still developing a production-level open-source compiler supporting CAF2.0, whose CAF2.0 runtime library uses the University of California, Berkeley' s GASNet [33] as the underlying communication library. GASNet is a language-independent, low-level network communication layer that provides network-independent, efficient communication primitives to support common global address space SPMD languages such as UPC, Titanium, and Co-Array Fortran.

2.8.3 Co-array Application Performance Cristian Coarfă et al. [34] evaluated shared-address-space SPMD languages in 2005, including CAF. The experiments used NAS Parallel Benchmarks (NPB) MG, CG, SP, and BT, comparing CAF, UPC, and Fortran + MPI implementations. The experimental platforms included four types: 1. 92-node HP zx6000 workstation + Myrinet 2000 2. SGI Origin 2000

Figure 10 [Figure 10: see original paper] shows the performance comparison of MPI, CAF, and UPC on NAS MG, comparing MPI Fortran version, CAF, UPC, and their performance when using different variants, where BUPC represents compilation using BUPC, CAF-barrier refers to synchronization using barrier methods, BUPC-restrict refers to using the restrict keyword for alias optimization, BUPC-p2p refers to synchronization using point-to-point methods, and

BUPC-strided uses UPC extensions for bulk transfers of strided data.

Figure 11 [Figure 11: see original paper] shows the performance comparison of MPI, CAF, and UPC on NAS CG:

From Figures 10 and 11, it can be seen that although performance comparison results vary due to program differences, CAF shows good performance comparable to traditional MPI programs and the emerging UPC language. As a shared-address-space parallel language extension, CAF uses the most acceptable and easily writable way to extend Fortran language data object subscripts, achieves good performance results, and has good scalability and platform adaptability, which is an important reason why CAF could be added to the Fortran 2008 standard.

2.9.1 F-

As the predecessor of Co-Array Fortran, F-[35] is a parallel extension based on Fortran 77/90 language, also using the SPMD programming model. Its execution model (computation distribution) and data model (data distribution) are exactly the same as Co-Array, with only slightly different syntax, partly due to differences between Fortran 90 and 95 syntax.

2.9.2 Fortran D

Fortran D [13][36,37] is a set of efficient parallel extensions to Fortran 77 language that can be used to specify data partitioning on distributed-memory machines. The Fortran D compiler then automatically creates efficient parallel code based on data partitioning for computation partitioning. The compiler generates explicit communication (based on message passing) and optimizes this communication to generate efficient parallel code. Ultimately, data is distributed across nodes according to the specified data partitioning, and SPMD executable programs are distributed to all nodes for parallel execution.

Although Fortran D is an old language extension and compilation system that is gradually no longer used, its development has made tremendous contributions to parallel program research on distributed-memory parallel machines. For example, Fortran D had an important influence on the development of HPF. HPF adopted almost the same approach of leaving data partitioning tasks to programmers, and HPF' s backend also used many of the same optimization techniques. This is why HPF and Fortran D extensions are so similar.

2.9.3 MPF

The MPF [38] programming language is an extension to Fortran 90 proposed by A.Ya. Kalinov et al. for distributed parallel systems. MPF' s design experience comes from the development and application of the MPC programming language.

MPF is an explicit parallel programming method that attempts to find a balance between efficiency and ease of programming. In previous parallel extensions to Fortran languages, HPF was easy to program with high developer efficiency, but program compilation and optimization were more difficult, causing its performance to largely depend on compiler implementation and be lower than programs written in MPI + Fortran. Therefore, MPF' s purpose is to provide better programmability while improving efficiency over HPF.

MPF' s features are not detailed here. In comparison results with MPI programs on a small number of programs, MPI program performance is about 10% higher than MPF programs. However, MPF programs have better expressiveness and are easier to program. But MPF has not been widely used or accepted, remaining only in the research and experimental stage.

2.9.4 IPFortran

IPFortran [39] is a programming language that utilizes multiple processes to implement data parallelism, an extension to the Fortran language. When implementing data-parallel communication between multiple processes, send and receive operations are implicitly included in infix operators and reduce functions. Since system-related message communication code is transparent to programmers and handled by the compiler and runtime, and code is written serially, code development efficiency is improved. Moreover, the logic of message communication is automatically generated by the compiler, thereby reducing errors in development.

An important prerequisite for IPFortran programming is that each processor knows the names of variables on all processors. To achieve this prerequisite, all processors need to run the same IPFortran code, using the SPMD programming model. Programmers use a local view approach to write code for each processor explicitly and use corresponding control structures for data decomposition and propagation. Using this method, programmers can efficiently write IPFortran code, and much industrial-level code can be quickly ported to IPFortran.

2.9.5 Fortran M

Fortran M [40,41] is a set of extensions based on Fortran 77 that supports modular message-passing programming methodology. Fortran M has the following characteristics: - **Modular**: Programs communicate using explicitly defined communication channels. Fortran M programs are formed by connecting modules together through these communication channels. Each module is called a process, which can contain ordinary data, sub-processes, and internal communication. - **Safe**: Operations on channels are strictly restricted and regulated, avoiding ambiguous execution and incorrect results. Communication channels are typed, so compilers can check whether they are used correctly. - **Architecture-independent**: Process mapping can first be mapped to a virtual computer configuration different from the actual machine. Mapping can

be done through annotations, affecting only program performance, not correctness. - **Efficient**: Fortran M can be efficiently implemented on single processors, shared-memory systems, non-shared-memory distributed systems, and network workstations. Since communication is included in the code, compilers can also optimize communication when optimizing computation.

2.9.6 OpenMP Fortran

OpenMP [42] is not an exclusive Fortran extension but rather an industrial standard of directives, library functions, and environment variables aimed at providing a parallel program development method on shared-memory machines that is widely supported by compilers. Similar to HPF, OpenMP specifies parallel regions through directives, with code in parallel regions executing on threads. OpenMP parallel regions allow nesting.

OpenMP is supported by almost all mainstream compilers and has become the de facto standard for parallel programming in shared-memory environments. Due to its outstanding portability, it has gradually replaced many vendor-specific private extensions.

3.1 VPP Fortran

VPP Fortran [43~46] was originally developed for the VPP series supercomputers (1991). On one hand, because it uses a global namespace, VPP Fortran programming is much easier than MPI. Data can be decomposed and distributed across multiple processors, with runtime ensuring data consistency and synchronization. On the other hand, VPP Fortran provides relatively low-level methods to support pure data parallelism, thereby achieving better performance. These methods can explicitly access data on each processor, implementing broadcast, multicast, barrier, critical section, synchronization, and other operations. These low-level features are not always necessary, but they can be used to improve the performance of critical programs.

3.1.1 Data Distribution

The target hardware described by the VPP Fortran programming model is shown in Figure 12 [Figure 12: see original paper]: each processor has its own local memory, which is always the fastest memory. Global memory is physically distributed across each processor and shared by all processors. In the VPP Fortran programming model, each variable is either local or global attribute, depending on its location.

3.1.2 Execution Process

The execution process of VPP Fortran programs is illustrated using the code segment shown in Figure 13 [Figure 13: see original paper]:

Code segments contained within PARALLEL REGION are executed in parallel. PARALLEL REGION uses a fork function to create processes on each processor and calls join at END PARALLEL. This behavior is somewhat similar to OpenMP, but VPP Fortran is used on distributed-memory parallel systems where both control and data are distributed across each node processor, whereas OpenMP is used on shared-memory systems where only control is distributed across multiple processors.

VPP Fortran has two decomposition (SPREAD) constructs: one is SPREAD REGION, which describes how tasks are decomposed; the other is SPREAD DO, which describes how loops are decomposed.

A set of PARALLEL REGION constructs or SPREAD constructs corresponds to a set of processors, called a region, and regions can be nested. As shown in Figure 13: C is assigned to processors 1 and 2, while D is assigned to processors 3-8. SPREAD DO distributes loop iterations to all processors for parallel execution. Additionally, VPP Fortran can implement partitioned indexing and explicit communication.

3.2 HPF/JA Extension

HPF/JA extensions [47~50] are a series of extensions based on VPP Fortran features, built upon HPF 2.0. Their features include asynchronous communication between processing stages, explicit compiler directives for shadow, and local directives. These features are also very useful in VPP Fortran when dealing with real applications. Here is an example of HPF/JA to explain its features:

Figure 14 [Figure 14: see original paper] shows an example of how to access data on adjacent processors. The left side is the HPF implementation of the BT benchmark program. Arrays X and U have their second dimension distributed in BLOCK mode, and the following loop is parallelized. Accessing array X is local access, but array U is not necessarily so, because U array subscript access uses J-1, J-2, J+1, J+2 instead of J, and these elements happen to be mapped to adjacent processors. HPF 2.0 defines shadow compiler directives for such loops. Shadow specifies that a processor has a memory block on its local storage that can hold part of the data from adjacent processors for convenient access.

Figure 14(b) shows how to use the shadow compiler directive, where the second dimension of U has shadow areas of 2 elements at both upper and lower boundaries. The Reflect directive is used before the loop to copy boundary elements from adjacent processors to the current processor' s local storage, ensuring that U is local before being accessed, allowing us to use the local compiler directive.

The local compiler directive was introduced for the following reason: even when data is stored on the same processor during execution, if the compiler doesn' t know where the data is at compile time, it may still cause inefficient access. The reason is that finding data location at runtime involves executing many instructions, which is relatively expensive. Although data location can often be

analyzed at compile time, there remains some uncertainty. The local compiler directive is introduced to tell the compiler that data is definitely stored in the current processor' s local memory, thereby optimizing data access and improving execution efficiency.

3.3 ICL DAP Fortran

International Computing Limited produced the earliest parallel processor DAP and developed DAP Fortran for it. DAP Fortran [51] is an extension to the non-I/O parts of FORTRAN 77 to support array and matrix calculations.

3.4 PGI CUDA Fortran

CUDA (Compute Unified Device Architecture) [52] is a computing platform launched by graphics card manufacturer NVIDIA. CUDA is a general-purpose parallel computing architecture that includes the CUDA instruction set architecture (ISA) and parallel computing engines inside GPUs, enabling GPUs to solve complex computing problems. Developers can now write programs for the CUDA architecture using the C language. The written programs can run at ultra-high performance on CUDA-supported processors.

CUDA supports both FORTRAN and C++. CUDA Fortran, jointly defined by NVIDIA and PGI, is implemented in PGI' s FORTRAN compiler. It is a directive-based interface similar to OpenMP. It still follows a simple heterogeneous environment programming interface where CPU and GPU are different devices with different address spaces. Host code runs on the CPU, while device code runs on the GPU.

The Portland Group Corporation and NVIDIA Corporation defined a small set of Fortran extensions that allow programmers to program the CUDA platform directly in Fortran, naturally introducing CUDA support by extending Fortran 90' s syntax for defining variable attributes, memory allocation statements, and array assignments.

Figures 15 [Figure 15: see original paper] and 16 [Figure 16: see original paper] show an example of blocked matrix multiplication:

3.5 Cray Parallelization Directives

Cray Corporation' s Cray Fortran [53] compiler toolchain includes support for a large number of the company' s private directives. These directives can be used not only to control compiler features but also to mark code that needs automatic thread parallelization.

3.6 Sun Parallelization Directives

Oracle' s Sun Studio development environment [54], in addition to supporting standard OpenMP, also has limited support for Cray and Sun private-format

parallel directives on FORALL structures after Fortran 95.

4 Summary

This paper surveys parallel extensions to the Fortran language since Fortran 77, with the most important being HPF and Co-array Fortran, which are widely applied and well-supported.

Fortran is a language with great development potential. During its global popularization, Fortran language standardization has continuously absorbed new features from modern programming languages and still occupies an important position in engineering computing.

In numerical computing, Fortran remains irreplaceable. The Fortran 90 standard introduced array operations and other features very conducive to matrix operations. During array operations, Fortran can automatically perform parallel operations, which many programming languages cannot do. Fortran allows users to utilize many existing function software packages, making it very convenient.

Fortran has not declined because of its long history and inability to keep up with development. Instead, it has gradually introduced features from new languages, added support for new hardware, and improved old parallel extensions. For example, Co-Array has received widespread support in recent years and become part of the Fortran 2008 standard. Fortran' s support for emerging accelerators such as GPU CUDA programming demonstrates its development with the times. Moreover, because Fortran has always occupied a major position in high-performance computing, various tools and high-performance computing software packages and mathematical libraries are relatively complete, providing a good foundation for its future development.

References

[1] Backus, J. W., H. Stern, I. Ziller, et al. (1957). "The FORTRAN Automatic Coding System" . Western joint computer conference: Techniques for reliability (Los Angeles, California: Institute of Radio Engineers, American Institute of Electrical Engineers, ACM): 188–198. doi:10.1145/1455567.1455599.

[2] Duan Zhaohui. 1995. The Development History of FORTRAN Language. *Computer World*, Issue 8. National Research Center for Intelligent Computing Systems.

[3] ANSI x3.9-1966. USA Standard FORTRAN. American National Standards Institute. Informally known as FORTRAN 66.

[4] ANSI x3.9-1978. American National Standard -Programming Language FORTRAN. American National Standards Institute. Also known as ISO 1539-1980, informally known as FORTRAN 77.

[5] ANSI X3.198-1992 (R1997) / ISO/IEC 1539:1991. American National Standard –Programming Language Fortran Extended. American National Standards Institute / ISO/IEC. Informally known as Fortran 90.

[6] ISO/IEC 1539-1:1997. Information technology –Programming languages –Fortran –Part 1: Base language. Informally known as Fortran 95. There are a further two parts to this standard. Part 1 has been formally adopted by ANSI.

[7] ISO/IEC 1539-1:2004. Information technology –Programming languages –Fortran –Part 1: Base language. Informally known as Fortran 2003.

[8] ISO/IEC 1539-1:2010 (Final Draft International Standard). Information technology –Programming languages –Fortran –Part 1: Base language. Informally known as Fortran 2008.

[9] Page, Clive G. (1988). *Professional Programmer' s Guide to Fortran77* (7 June 2005 ed.). London: Pitman. ISBN 0-273-02856-1. Retrieved 4 May 2010.

[10] CORPORATE The Parallel Computing Forum (1991). ACM SIGPLAN Fortran Forum. Volume 10 Issue 3, Sept. 1991, Pages 1 –57, ACM New York, NY, USA.

[11] Press, William H. (1996). *Numerical Recipes in Fortran 90: The Art of Parallel Scientific Computing*. Cambridge, UK: Cambridge University Press. ISBN 0-521-57439-0.

[12] Akin, Ed (2003). *Object Oriented Programming via Fortran 90/95* (1st ed.). Cambridge University Press. ISBN 0-521-52408-3.

[13] D. Tam, Prof. Abdelrahman, (2000). Fortran D - Final Report. Compilation Techniques for Parallel Processors ECE 1754. Friday May 12, 2000.

[14] M. Ujaldon, E. L. Zapata, B. M. Chapman, et al. (1997). Vienna-Fortran/HPF Extensions for Sparse and Irregular Problems and Their Compilation. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, vol. 8, pages: 1068 –1083, 1997.

[15] HPFF - Rice University HPF Forum, http://hpff.rice.edu/ (http://dacnet.rice.edu/).

[16] High Performance Fortran Forum (1994). *High Performance Fortran Language Specification.*

[17] C. H. Koelbel. (1997). *The high performance Fortran handbook –2nd edition.* MIT press.

[18] Shires Dale, Mohan Ram. An Evaluation of HPF and MPI Approaches and Performance in Unstructured Finite Element Simulations. *Journal of Mathematical Modeling and Algorithms* 2002-09-01.

[19] Murai, Hitoshi, Araki, et al. Implementation and evaluation of HPF/SX V2. *Concurrency and Computation: Practice and experience.*

[20] Y. C. Hu, G. H. Jin, S. L. Johnsson, et al. HPFBench: A High Performance Fortran Benchmark Suite.

[21] Applied Parallel Research, Inc., APR' s public domain benchmark suite, ftp://ftp.infomall.org/tenants/apri/Benchmarks, 1996.

[22] C. A. Addison, et al., The GENESIS Distributed-memory Benchmarks, *Computer Benchmarks*, J.J. Dongara and W. Gentzsch (Eds), Advances in Parallel Computing, Vol. 8, Elsevier Science Publications, BV (North-Holland), Amsterdam, The Netherlands, pp. 257-271, 1991.

[23] D. E. Bailey, et al., The NAS Parallel Bench-marks, Technical Report RNR-94-007, NASA Ames Research Center, 1994.

[24] Chapman, Stephen J. (2007). *Fortran 95/2003 for Scientists and Engineers* (3rd ed.). McGraw-Hill. ISBN

[25] Chivers, Ian; Sleightholme, Jane (2012). *Introduction to Programming with Fortran* (2nd ed.). Springer. ISBN 978-0-85729-232-2.

[26] Metcalf, Michael; John Reid, Malcolm Cohen (2011). *Modern Fortran Explained*. Oxford University Press. ISBN 0-19-960142-9.

[27] Co-array Fortran: http://www.co-array.org/.

[28] Co-array Fortran at Rice: http://caf.rice.edu/.

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

[30] R. W. Numrich, J. Reid, (2005). Coarrays in the next Fortran Standard. *SIGPLAN Fortran Forum*, 24(2): 4-17, Aug. 2005.

[31] C. Coarfa, Y. Dotsenko, J. Eckhardt, et al. Co-array Fortran performance and potential: An NPB experimental study. In 16th International Workshop on Languages and Compilers for Parallel Processing (LCPC), October 2003.

[32] John Mellor-Crummey, Laksono Adhianto, William Scherer III, and Guohua Jin, A New Vision for Co-array Fortran, Proceedings PGAS09, 2009.

[33] GASNet: http://gasnet.cs.berkeley.edu/.

[34] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, et al. 2005. An evaluation of global address space languages: co-array Fortran and unified parallel C. In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming Pages 36 –47, ACM New York, NY, USA (PPoPP ' 05).

[35] R. W. Numrich, J. L. Steidel, B. H. Johnson,et al. (1997). Definition of the F–extension to Fortran 90. In Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing (LCPC ' 97), Pages 292 – 306, Springer-Verlag London, UK.

[36] S. Hiranandani, K. Kenney, C. Koelbel, et al. (1991) An Overview of the Fortran D Programming System. In Proceedings of the Fourth International

Workshop on Languages and Compilers for Parallel Computing, Pages 18 –34, Springer-Verlag London, UK, 1991.

[37] R. V. Hanxleden, K. Kennedy, C. Koelbel, et al. (1992). Compiler Analysis for Irregular Problems in Fortran D. In Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, Pages 97 –111, Springer-Verlag London, UK, 1992.

[38] Kalinov, Alexey, Ledovskih, et al. (2004). An Extension of Fortran for High Performance Parallel Computing. *Programming and Computer Software*. 2004.

[39] B. Bagheri, T. W. Clark, L. R. Scott. 1992. IPFortran: a parallel dialect of Fortran. *SIGPLAN Fortran Forum* 11, 3 (September 1992), 20-31. DOI=10.1145/141438.141450.

[40] I. T. Foster, K. M. Chandy. (1995). Fortran M: A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, Volume 26, Issue 1, 1 April 1995, Pages 24–35.

[41] I. T. Foster, D. W. Walker. (1994). Paradigms and Strategies for scientific computing on distributed memory concurrent computers. In Proceedings of the High Performance Computing 1994 Conference, La Jolla, CA, April 11-15, 1994. Published by the Society for Computer Simulation, San Diego, CA.

[42] OpenMP: http://openmp.org/wp/.

[43] H. Iwashita, S. Okada, M. Nakanishi, et al. (1994). VPP Fortran and Parallel Programming on the VPP500 Supercomputer. In Proceedings of the 1994 International Symposium on Parallel Architectures, Algorithms and Networks (poster session papers), pages 165-172. Japan, December 1994.

[44] E. Yamanaka, T. Shindo. (1997). Parallel Language Processing System for High-Performance Computing. *Fujitsu Sci. Tech. J.*, 33(1): 39-51, June 1997.

[45] H. Iwashita. (1997). A view of VPP Fortran in contrast with HPF. *IPSJ Magazine*, 38(2), February 1997. (in Japanese)

[46] Fortran VPP300 User guide: Programming in VPP Fortran. http://anusf.anu.edu.au/VPP/Userguide/VPI

[47] HPF/JA Language Speciation, JAHPF (Japan Association for High Performance Fortran), January 31, 1999 Version 1.0.

[48] Y. Seo1, H. Ohta, H. Sakagami, et al. (2002). HPF/JA: extensions of High Performance Fortran for accelerating real-world applications. *Concurrency and Computation: Practice and Experience* Volume 14, Issue 8-9, pages 555–573, 2002.

[49] H. Iwashita1, N. Sueyasu1, S. Kamiya1, et al. (2002). VPP Fortran and the design of HPF/JA extensions. *Concurrency and Computation: Practice and Experience* Special Issue: High Performance Fortran Volume 14, Issue 8-9, pages 575–588, 2002.

[50] Asaoka, Kae, Hirano, et al. Evaluation of the HPF/JA Extensions on Fujitsu VPP Using the NAS Parallel Benchmarks. *High Performance Computing.*

[51] ICL. Icl dap Fortran: http://www.hpjava.org/talks/beijing/hpf/introduction/node5.html.

[52] CUDA: http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation.

[53] Cray Fortran: www.ccs.ornl.gov/Phoenix/ftn.html.

[54] Oracle Sun Studio: www.oracle.com/.

Author Biographies

Su Le: Ph.D. candidate, Virtual Reality Technology Group, Forward-Looking Technology Laboratory, Institute of Computing Technology, Chinese Academy of Sciences. sule@ict.ac.cn

Fang Shuangde: Ph.D. candidate, Advanced Compilation Technology Group, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences.

Huang Yuanjie: Ph.D. candidate, Advanced Compilation Technology Group, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences.

Note: Figure translations are in progress. See original paper for figures.