



Fortran 90 for Scientific Computing

A.G. Sunderland, I.J. Bush, M. Plummer and M. Ashworth
Computational Science and Engineering Department
CCLRC Daresbury Laboratory, Warrington, WA4 4AD, UK

Abstract

The Fortran language is widely considered to provide the best facilities for dealing with the numerical data associated with engineering and scientific application codes. Consequently, most large-scale scientific computer programs are written in Fortran. Traditionally, Fortran IV or Fortran 66 was used, but today most codes have migrated to, or are being written in, either Fortran 77 or Fortran 90/95. In many cases Fortran 77, Fortran 90 and C are combined together and the codes also may contain pre-processing commands and instructions for implementation in parallel such as MPI and Open MP. This publication aims to i) summarise the main new features of Fortran 90, Fortran 95 and Fortran 200x; ii) discuss backward compatibility issues with legacy Fortran source code; iii) highlight any performance issues associated with Fortran 90 usage on high end computing platforms.

This is a Technical Report of the UKHEC Collaboration.

Report available from

http://www.ukhec.ac.uk/publications/reports/f90_experiences.pdf

© UKHEC 2001.

Neither the UKHEC Collaboration nor its members separately accept any responsibility for loss or damage arising from the use of information contained in any of their reports or in any communication about their tests or investigations

1	<i>Fortran 90 / 95 Overview and Code Compatibility</i>	3
2	<i>Fortran Standards</i>	3
2.1	Fortran 95 Language Standard	4
2.2	Object-Oriented Fortran 90	5
3	<i>Modules</i>	5
3.1	Modules for Global Data	6
3.2	Modules for Derived Types	6
3.3	Modules for Explicit Procedure Interfaces	6
3.4	Modules Containing Procedures	6
4	<i>Dynamic Memory</i>	6
5	<i>Performance of Intrinsic Functions</i>	8
6	<i>Performance of External Subroutines</i>	9
7	<i>Problems with Word-size and Addressing Modes</i>	9
8	<i>Undefined Variables</i>	10
9	<i>Performance Analysis Tools</i>	10
10	<i>Information on Compilers</i>	10
11	<i>Survey of Fortran 90 / Fortran 95 Compilers</i>	10
11.1	Compaq/HP	10
11.2	IBM	11
11.3	SGI	11
11.4	Portland Group	11
11.5	CRAY	11
11.6	INTEL	12
11.7	SUN	12
11.8	ABSOFIT	12
11.9	SALFORD	12
11.10	NAG (Numerical Algorithms Group)	13
12	<i>Fortran 200x</i>	13

1 Fortran 90 / 95 Overview and Code Compatibility

The CSE introductory web article ‘Fortran 95 for Fortran 77 programmers’ gives a general description of the Fortran 90 and Fortran 95 standards and their many improvements over Fortran 77. New codes written at Daresbury take full advantage of opportunities for modularization, dynamic memory allocation and moves towards object oriented programming. Established scientific application codes and numerical libraries are usually supplied with a variety of makefile options for different systems with ‘best-optimised’ compiler flags. A separate report on building portable Fortran 90 applications is available [2].

A few Fortran 66 features have been made incompatible with Fortran 95: these features are regarded as poor practice and are rejected at compilation time. Many Fortran 77 scientific codes make use of some of the new features from Fortran 90. Fortran 77 codes are generally accepted by Fortran 90 compilers, as the 77 standard is a sub-set of the 90 and 95 standards. The ‘fixed’ source form is either recognised through the use of a .f (as opposed to .f90) file extension, or may be specified using a compiler option.

The Fortran 90 and Fortran 95 standards have expanded considerably the namespace associated with the language and this can cause conflicts when compiling Fortran 77 and Fortran 66 codes. This often occurs when a Fortran 77 variable name conflicts with Fortran 90 syntax or an intrinsic function. Examples include the COUNT and MAXVAL intrinsic functions that are commonly used as descriptive variable names in Fortran 77. Users of Fortran 77 codes may also routinely use extensions to the language (without necessarily being aware of this) which become redundant, or cause a conflict in Fortran 90 and Fortran 95.

2 Fortran Standards

The ISO Fortran Standards Technical Committee WG5 <http://www.nag.co.uk/sc22wg5> is the working group committee responsible for the development and maintenance of Fortran language standards. In order to understand fully the behaviour of scientific programs it is often important for Fortran programmers to be able to distinguish between what the standard prescribes and what it does not. For instance, the Fortran 95 standard specifies the following information:

- Syntax of Fortran statements and forms for Fortran programs
- Semantics of Fortran statements and Fortran programs
- Specifications for correct input data
- Appearance of standard output data.

The standard does not specify the following information:

- The way in which each Fortran compiler is written

- Operating system facilities defining the computing system
- Methods used to transfer data to and from peripheral storage devices and the nature of the peripheral devices
- Behaviour of vendor extensions
- Size and complexity of a Fortran program and its data
- Hardware or firmware used to run the program
- The way values are represented and the way numeric values are computed
- Physical representation of data
- Characteristics of tapes, disks, and various storage media.

2.1 Fortran 95 Language Standard

Fortran 95 continues the evolutionary model introduced in Fortran 90 by deleting several of the features marked as obsolescent in Fortran 90 and identifying a few new obsolescent features. For information on these features, see the Fortran Language Reference Manual, Volume 3. Fortran 95 is a relatively minor evolution of standard Fortran, with the emphasis in this revision being upon correcting defects in the Fortran 90 standard. The ISO standards committee had also endeavoured to keep Fortran in step with the work in the HPF area. This new standard also provides interpretation for a number of questions that have arisen concerning Fortran 90 semantics and syntax. For example, the Fortran 95 SIGN intrinsic function behaves differently from the Fortran 90 SIGN function if the second argument is negative real zero.

In addition to corrections and clarifications, Fortran 95 contains several extensions to Fortran 90. The major extensions are as follows:

- The FORALL statement and construct.
- PURE and ELEMENTAL procedures.
- Pointer initialisation and structure default initialisation.
- Implicit initialisation of derived type objects.
- Additional intrinsic procedures. A Fortran 90 program may have a different interpretation under the Fortran 95 standard if it invokes an external procedure that has the same name as one of the new standard intrinsic procedures unless that procedure is specified in an EXTERNAL statement or an interface body.

Minor features include:

- new intrinsic function NULL
- new intrinsic function CPU_TIME
- automatic deallocation of allocatable arrays

- SIGN can distinguish between +0 and -0
- comments in namelist input data
- references to pure functions in specification expressions
- changes to some intrinsic functions.

2.2 Object-Oriented Fortran 90

Fortran 90 is a language that introduces many new features beneficial for producing reliable maintainable and portable scientific programs. Whilst array-syntax and dynamic memory capabilities are well known beneficial features of Fortran 90, the language supports many other modern software concepts, included object-oriented programming.

While Fortran 90 is not a full object-oriented language it can directly support many of the important concepts of such languages including abstract data types, encapsulation, function overloading, and classes. Other concepts, such as inheritance and dynamic dispatching, are not supported directly, but can be emulated (direct support is a [Fortran 2000](#) requirement (see below).) Since Fortran 90 is backward compatible with Fortran 77, new concepts can be introduced into existing programs in a controlled manner. This allows experienced Fortran 77 programmers to modernise their software, making it easier to understand, modify, share, and extend based on the benefits modern programming principles provide. An article outlining the use of object-oriented concepts in Fortran 90 can be accessed at http://www.cs.rpi.edu/~szymansk/OOF90/F90_Objects.html [7].

The addition of higher level abstractions can create difficulties for compilers when optimising code and this in turn can lead to a degradation of performance. Some performance loss using these abstract data types is expected due to pointer aliasing (multiple pointers referring to the same memory location). The compiler cannot always determine if two different pointer references refer to the same memory location, and therefore must produce safer, slower code. In Fortran 77 style arrays such overlapping references are forbidden, allowing the compiler to produce faster code. This issue is common to the C programming language where excessive use of pointers can degrade performance due to aliasing.

Moreover, object-oriented codes often test compilers in new and unusual ways, and this can sometimes lead to compilation failures or incorrect calculations. Decyk, Norton and Szymanski [8] have measured these effects for a range of Fortran 90 compilers and results are presented at <http://www.cs.rpi.edu/~szymansk/oof90.html>.

3 Modules

The module is one of the most important features of Fortran 90. Modules are a type of program unit where everything required by more than one program unit may be packaged and made available where required. They are particularly useful for the following :-

- Global data;

- Derived types;
- Explicit interfaces;
- Containing procedures.

3.1 Modules for Global Data

Modules are used in Fortran 90 to supersede the use of common blocks for data that is shared between different subprograms. The best way of managing common blocks in Fortran 77 was to place common declarations in an include file and to include this file into each subprogram using Fortran include statements (non-standard but commonly available) or using C pre-processor (`CPP`) include directives. With Fortran 90 we can regularise this process. Variables that are declared in Fortran 90 modules are automatically made available by the compiler to all subprograms which 'USE' the module. Modules also allow passing of dynamic data type whilst common blocks do not.

3.2 Modules for Derived Types

When passing derived data types as arguments to subroutines, both the actual arguments and dummy arguments must be of the same type, that is they must be declared with reference to the same type definition. This is best achieved by using modules. The user-defined type is declared in a module and each program unit that requires that type uses the module.

3.3 Modules for Explicit Procedure Interfaces

Interface blocks are mandatory in Fortran 90 under certain conditions, e.g. pointers, generic procedures, operator overloading. In large programs with many procedure references it can become tedious to repeat interface blocks throughout. A more convenient solution is to place one or more procedure interfaces in a module

3.4 Modules Containing Procedures

When designing programs, related subprograms, subroutines and functions can be packaged into the same module. These procedures are *contained* within the module as *module procedures* and have access to any data and type definitions declared in the module and their interfaces are explicit to each other. The `PRIVATE` and `PUBLIC` keywords are useful features for hiding non-relevant data and procedures from module users.

4 Dynamic Memory

One of the major new features of Fortran 90 that has benefits for scientific codes is that of dynamic memory management. Fortran 77 codes have to use static allocation. This means that codes must be recompiled for different problem sizes or different numbers of processors. The only alternative is to declare a single large array at compile time and then, at run time, assign indexes to point at different

locations within the array for use by different quantities. This results in code that is very cluttered and difficult to maintain.

With Fortran 90 we can declare the major arrays of a code as `allocatable`. Then the problem size, number of processors etc. can be read in at run-time and arrays allocated to the size required. According to the decomposition of the problem, arrays may be different sizes on different processes. Ensuring that arrays are the correct size for the problem, rather than using a part of an over-sized array, can bring performance benefits through improved cache utilisation as well as saving memory. With Fortran 90 we can declare the major arrays of a code as `allocatable`. Then the problem size, number of processors etc. can be read in at run-time and arrays allocated to the size required. According to the decomposition of the problem, arrays may be different sizes on different processes. Ensuring that arrays are the correct size for the problem, rather than using a part of an over-sized array, can bring performance benefits through improved cache utilisation as well as saving memory. Temporary arrays can also be declared to the correct size at the top of a subroutine (automatic arrays). In this case their space in memory is automatically returned and available for re-use when control leaves the subroutine (this process is similar to the 'garbage collection' available in object-oriented languages such as C++ and Java).

Typically on modern UNIX platforms, a regular, statically allocated Fortran 77 array resides in the heap (or data) region of memory. However a Fortran 90 regular array is often in the stack region of memory unless the array appears in a `SAVE` statement or a `COMMON` block. Fortran 90 `allocatable` arrays reside in the heap region whereas a Fortran 90 automatic array is in the stack region.

With these improvements in memory management, there is now little excuse for the use of `equivalence` with all the dangers and performance degradation that it can bring.

However, it is prudent also to sound a note of caution. The Direct Numerical Simulation CFD code, `pdns3d`, (which originates from the Aeronautics group at the University of Southampton) has been designed to use either static or dynamic (`allocatable`) array allocation. The choice is made at compile time using a `cpp` option. Performance results show (Figure 1) that for small numbers of processors on the Cray T3E and in all cases on the SGI Origin 3800 there is a performance penalty associated with dynamic array allocation that can be as high as a factor of six. The cause is not clear, but we believe that such a large performance effect must be related to compiler optimisation. In some cases, compilers can perform more effective loop optimisations when they know the sizes of the arrays. The fact that the performance degradation on the T3E only occurs at small numbers of processors, when there is an issue about whether arrays and columns of arrays fit into cache, suggests that it is a matter of how well the compiler generated code utilises cache. Setting the environment variables `SMA_GLOBAL_ALLOC=1` and `SMA_GLOBAL_HEAP_SIZE=1000000000` on the Origin may help improve performance.

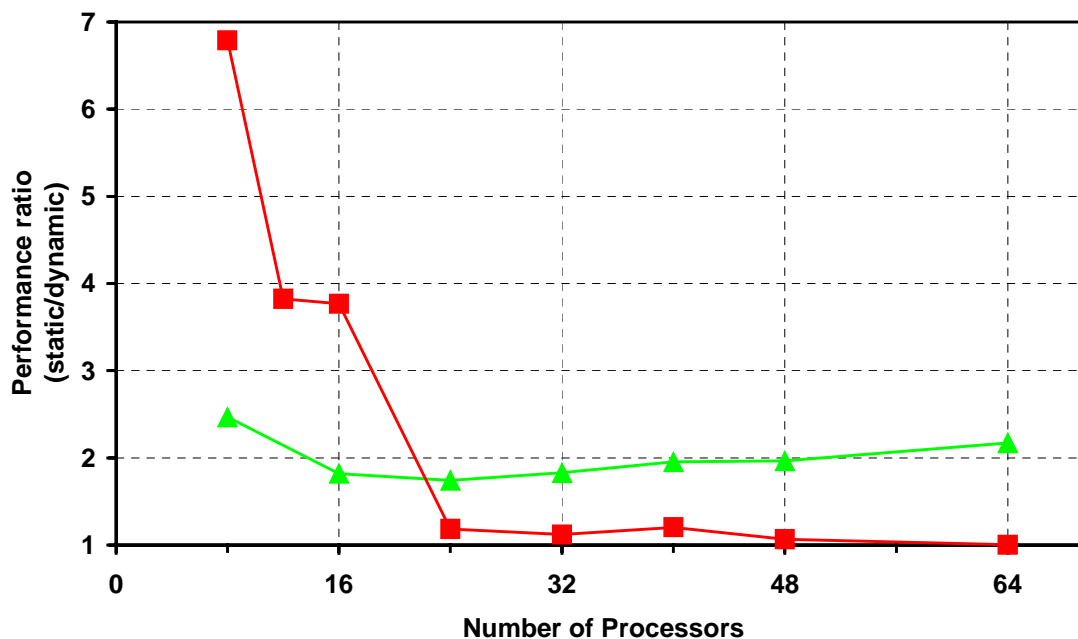


Figure 1: Performance ratio using static and dynamic memory allocation of the `pdns3d` code on the Cray T3E (red squares) and the SGI Origin 3800 (green triangles). A ratio greater than one indicates that the static code is faster.

5 Performance of Intrinsic Functions

Performance related compiler problems also occur for new Fortran 90 features with which the compiler has not dealt efficiently. A prime example is the matrix-matrix multiply intrinsic function, `matmul`. Using `matmul` may be much slower than calling the equivalent LAPACK and BLAS routines, `xGEMM` and `xGEMV` (see [9]). Example i) in the appendix lists the assembler code generated by the current F90 Cray compiler. This maps the source code construct `c = matmul(a,b)` to the highly-optimised external BLAS routine `SGEMM`, resulting in a good serial performance of around 600 Mflop/s. However, it is shown in example ii) that the assembler code produced from the construct `c = alpha * matmul(a,b)` does not reference `SGEMM`, resulting in a reduced serial performance of around 100 Mflop/s. To counter this problem the new codes can make use of a generic interface to a matrix multiplication module which can be set up to use either `matmul` or the BLAS routines directly without disturbing the rest of the code.

On IBM Power4 p690 based systems, the F90 intrinsic `matmul` can yield poor performance if the following conditions are satisfied:

- Used in a parallel program

- Source code compilation used reentrant compilers (e.g. mpxf90_r).
- Size of the matrix is 108 or larger.

Under these circumstances the `matmul` intrinsic becomes multi-threaded and therefore performance across multiple SMP nodes can slow down noticeably. IBM advises that this performance problem can be corrected by recompiling with `-qhot`. An alternative solution is to replace calls to `matmul` with calls to `xGEMV` and `xGEMM`.

6 Performance of External Subroutines

If there is no explicit interface available for a routine that is called from a Fortran 90 program the compiler may make local copies of the variables passed to the routine. This is because the program may be passing assumed-shape arrays (e.g. array sections with elements in non-contiguous memory storage). This will also depend on the calling routine and may vary from compiler to compiler. This 'copy-in/copy-out' overhead has the potential to degrade the performance benefits of external optimised numerical library routines, such as the BLAS. In addition, it may affect asynchronous MPI communications [2].

The extent to which F90 compilers overcome the performance of non-uniform strides in memory can also vary. For example, the use of the transpose ('T') option in calls to `xGEMM` (and `matmul`) can significantly affect performance on the Cray T3E. Generally, the matrix operation $\mathbf{A}^T\mathbf{B}$ is the simplest for the F90 compiler to optimise, as all data is accessed contiguously in memory, and thus good performance is ensured. On the Cray T3E, the matrix operation \mathbf{AB} gives almost identical performance, indicating that the compiler has optimised the non-unit strides in \mathbf{A} . It achieves this by holding sections of one matrix in cache and using the T3E's stream buffers for the other. However the performance of \mathbf{AB}^T is generally lower and highly erratic (Fig.4.5 in [9]), as this optimisation strategy is much less effective in cases where **both** matrices are accessed non-contiguously. It is therefore recommended that when undertaking matrix-multiply operations of the type \mathbf{AB}^T , \mathbf{B}^T is transposed before the matrix-multiply subroutine call. In comparison, the optimisation of matrix-multiply operations on the IBM SP and SGI Origin series yields good performance regardless of any matrix transpose options selected.

7 Problems with Word-size and Addressing Modes

The main porting idiosyncrasies we have dealt with are in switching codes between the Cray T3E and the SGI Origin systems at Manchester. These problems are often related to the default sizes of variables and can be dealt with in pre-processing.

Another issue was raised on porting the POLCOMS shelf seas modelling code from the T3E to the Origin. The code used `MPI_Address` in order to combine variables of different type into a single derived type for efficient message passing. When compiled with 64-bit addressing (compiler option `-64`) the code fails. This is because the `MPI_Address` function tries to return a 64-bit address into a standard-sized 32-bit integer. The problem was fixed by removing the use of `MPI_Address` as this code only appeared in the initialisation phase of the code. A fix which

restores the capability to construct derived data types using addresses will be available using the MPI-2 subroutine `MPI_Get_Address` (due to be made available on the Origin with MPT 1.5.2) as this allows MPI to specify the kind of the integer result.

8 Undefined Variables

Running a code on a range of different machines can be very helpful for improving the code's robustness by showing up undefined variables: some compilers set these to zero, others give them random values or leave them undefined, leading to runtime crashes. In general it is highly inadvisable to access variables before they have been defined. Compiler options may help. For example, on the Cray T3E try running with the compiler option `-ei`, which sets all local variables to undefined, and with a loader CLD directive file containing the command `preset=nan`, which sets all global variables to undefined. This has the effect of generating an exception if a variable is referenced before it is defined and allows these situations to be easily detected and eliminated.

9 Performance Analysis Tools

Not all performance analysis tools are designed for Fortran 90 and may either fail or point inaccurately at the source code.

10 Information on Compilers

For Windows and Linux based systems, an exhaustive guide to Fortran compilers, including detailed comparisons of diagnostic capabilities and some benchmarking is available at <http://www.polyhedron.co.uk/>. The UKHEC report on Beowulf systems built from commodity components for scientific applications [1] includes a short section on compilers. The national resource at Manchester Computing Centre provides extensive web pages and links describing the Cray, SGI and other compilers and pre-processors (<http://www.csar.cfs.ac.uk/>). Other sites collating information on Fortran may be found at <http://www.fortran.com/> and <http://www.kcl.ac.uk/kis/support/cit/fortran/>.

11 Survey of Fortran 90 / Fortran 95 Compilers

11.1 Compaq/HP

Compaq Fortran is a full-language implementation of the Fortran 95 standard. Current implementations include Compaq Fortran for Tru64 UNIX Alpha systems (v5.5), Compaq Fortran for Linux Alpha systems (v1.2) and Compaq Visual Fortran for Windows 95/98/Me/NT/2000/XP systems. Highlights include:

- Support for the multi-vendor OpenMP Fortran specification;
- Support for High Performance Fortran (HPF V2.0) (UNIX Alpha systems only);

- The Compaq Extended Math Library CXML;
- Support for the proposed Fortran 2000 feature “allocatable components of derived types”;
- Free download of Compaq Fortran for Linux Alpha systems (v1.2) under an Enthusiast and Education license for non-commercial use.

See URL <http://www.compaq.com/fortran>

11.2 IBM

XL Fortran v7.1 is fully compliant with the Fortran 95 standard. IBM Fortran compilers are available for AIX systems only. Highlights include:

- Fortran 90 intrinsic performance enhancements;
- Full support for the OpenMP Fortran API Version 1.0 specification;
- POSIX pthreads support;
- Support for automatic and explicit parallelism;
- Portability enhancements, including:
 - Structure
 - Union
 - SIZEOF intrinsic
- Portability compiler options.

See URL <http://www-3.ibm.com/software/ad/fortran/xlfortran>

11.3 SGI

MIPSpro Fortran 95 Compiler v7.3 for all SGI IRIX systems. It supports the Fortran 95 ISO/ANSI standard. Highlights include:

- Automatic parallelizer helps take advantage of the parallelism in programs to provide better performance on multiprocessor systems;
- Fortran OpenMP specification supported in current compilers;
- Mixed language support (C, Fortran).

See URL <http://www.sgi.com/developers/devtools/languages/mipspro.html>

11.4 Portland Group

PGF90 *Workstation Class* compilers v4.0 are for x86-32 32-bit processor based workstations with up to 4 CPUs running Linux or W98/NT/w2000 O/S. Highlights include:

- Automatic or user-directed parallelization;
- Support for OpenMP or HPF user directed parallelization;
- Extensions for Fortran 95.

See URL <http://www.pgroup.com/products/workpgi.htm>

11.5 CRAY

CF90 Compilers for the Cray J90, C90, T90, T3E and SV1 systems.

- Fortran 95 standard compliant;
- Highly optimised scientific libraries.

See URL <http://www.cray.com/products/software/cf90.html>

11.6 INTEL

The Intel® Fortran Compiler 6.0 for Linux on IA32 and IA64 architectures.

- OpenMP support;
- Auto-parallelization;
- Conforms to ISO Fortran 95 standard;
- Mixed language support (C, Fortran).

See URL <http://developer.intel.com/software/products/compilers/f60/>

11.7 SUN

Sun Fortran 95 compiler, as part of the Sun ONE Studio 7 compiler collection, offers:

- Automatic loop parallelization;
- Full implementation of ISO Fortran 95 standard;
- Support for emerging Fortran 2000 standard;
- OpenMP Fortran v2.0 API support;
- Cray extensions, such as Cray Pointer.

See URL <http://www.sun.com/software/sundev/suncc/index.html>

11.8 ABSOFT

Absoft Fortran 95 compiler v7.5 for Linux/Windows/OX systems is a result of a five year joint effort with Cray Research.

- Complete ANSI/ISO F95 implementation.
- Source compatible with CF90 2.0 and earlier Cray releases.

See URL <http://www.absoft.com/>

11.9 SALFORD

Salford FTN95 compiler for Microsoft®.NET and Win32

- Full Fortran 95 source language and backwards compatibility with all commonly available F77 source extensions.
- Mixed Fortran 77 and Fortran 95 programs.
- Optimised Intrinsic Library.
- Optimised array features.
- Full screen debuggers for DOS and GUI debugger for Windows

See URL <http://www.salfordsoftware.co.uk>

11.10 NAG (Numerical Algorithms Group)

NAGWare f95 Compiler Release 4.2.

- Thread safe (reentrant) Run Time Library
- Full traceback with -gline
- Many common extensions of Fortran 77 are allowed.
- New Features of Fortran 2000 added:
 - Allocatable Components.
 - IEEE Floating Point Exception Handling.
- 30 day trial version available.

See URL <http://www.nag.co.uk>

12 Fortran 200x

Whereas Fortran 95 is a relatively minor enhancement of Fortran 90, the Fortran Standards Technical Committee Working Group 5 is currently working on a major revision of the language, with a target publication date of 2004. New features include:

- Derived Types enhancements: parameterized derived types (e.g. kind, length, shape of components to be chosen when derived type is used), allocatable components, mixed component accessibility, public entities of private type, improved structure constructors and finalizers.
- Exception Handling: support for the five exceptions of the IEC 60559 (IEEE 754) standard.
- Interoperability with the C Language: in order to enable Fortran programs to access and interact with the low-level facilities provided by C and operating systems. This feature also allows C programs to take advantage of Fortran numerical libraries.
- Support for object oriented programming: type extension and inheritance, polymorphism, dynamic type allocation and type-bound procedures.
- I/O enhancements: asynchronous transfer, derived type I/O.

References

1. R.J. Allan, S.J. Andrews, M.F. Guest, P.M. Oliver, D. Henty, L. Smith, S. Telford and S. Booth, Design and Building of Beowulf-class Cluster Computers. Technical Report (UKHEC 2000), <http://www.ukhec.ac.uk/publications/reports/beowulf.pdf>.
2. R.J. Allan and Y.F. Hu, Portable Application Compilation and Building for Fortran 90. Technical Report (UKHEC, 2001), http://www.ukhec.ac.uk/publications/reports/build_doc.pdf.
3. R.J. Allan and C.J. Müller, Tutorial on Shared-Memory Programming Paradigms (DL 1999).

4. S.J. Andrews and M.F. Guest, Proceedings of the 13th Daresbury Machine Evaluation Workshop. (DL 2003),
<http://www.cse.clrc.ac.uk/disco/mew/talks.html>.
5. L. Smith, Mixed Mode MPI / Open MP Programming. Technical Report (UKHEC 2000), <http://www.ukhec.ac.uk/publications/tw/mixed.pdf>.
6. R.J.Allan and Y.F.Hu, CLIPS: The CLRC Library of Parallel Subroutines, Technical Report (DL 1999).
7. V.K.Decyk, C.D.Norton and B.K.Szymanski, Introduction to Object-Oriented Concepts using Fortran 90. <http://www.cs.rpi.edu/~szymansk/oof90.html>.
8. V.K.Decyk, C.D.Norton and B.K.Szymanski, Performance Comparison of Data Structure Implementations in Fortran 90.
<http://www.cs.rpi.edu/~szymansk/oof90.html>.
9. Graeme Watt, Benchmarking OpenMP Implementations of a Finite Element Analysis Code. MRCCS Report, (2000).
10. Ian Bush, F95 for F77 Programmers. CSE Technical Report (2001). Publication at <http://www.cse.clrc.ac.uk>

Appendix

Cray assembler listing for calls to F90 intrinsic function **matmul** :

i) **c = matmul(a,b)**

```

      lda      r3, 120(r31)           ; Ln 13 0x00000144 (In 85)
      stq     r3, -136(r15)         ; Ln 13 0x00000148 @W0 (In
86)
      lda      r3, 120(r31)           ; Ln 13 0x0000014c (In 87)
      stq     r3, -128(r15)         ; Ln 13 0x00000150 @W1 (In
88)
      lda      r3, 120(r31)           ; Ln 13 0x00000154 (In 89)
      stq     r3, -120(r15)         ; Ln 13 0x00000158 @W2 (In
90)
      ldt     f10, (r11)             ; Ln 13 0x0000015c $FP000001
(In 91)
      stt     f10, -112(r15)         ; Ln 13 0x00000160 @W3 (In
92)
      lda      r3, 1(r31)            ; Ln 13 0x00000164 (In 93)
      stq     r3, -104(r15)         ; Ln 13 0x00000168 @W4 (In
94)
      lda      r3, 120(r31)           ; Ln 13 0x0000016c (In 95)
      stq     r3, -96(r15)          ; Ln 13 0x00000170 @W5 (In
96)
      lda      r3, 1(r31)            ; Ln 13 0x00000174 (In 97)

```

```

98)      stq      r3,  -88(r15)          ; Ln 13 0x00000178 @W6 (In
        lda      r3,  120(r31)         ; Ln 13 0x0000017c (In 99)
        stq      r3,  -80(r15)         ; Ln 13 0x00000180 @W7 (In
100)
        stt      f31, -72(r15)         ; Ln 13 0x00000184 @W8 (In
101)
        lda      r3,  1(r31)           ; Ln 13 0x00000188 (In 102)
        stq      r3,  -64(r15)         ; Ln 13 0x0000018c @W9 (In
103)
        lda      r3,  120(r31)         ; Ln 13 0x00000190 (In 104)
        stq      r3,  -56(r15)         ; Ln 13 0x00000194 @W10 (In
105)
        lda      r3,  -136(r15)        ; Ln 13 0x00000198 @W0 (In
106)
        lda      r18, -128(r15)        ; Ln 13 0x0000019c @W1 (In
107)
        lda      r5,  -120(r15)        ; Ln 13 0x000001a0 @W2 (In
108)
        lda      r2,  -112(r15)        ; Ln 13 0x000001a4 @W3 (In
109)
        lalm     r1,  ^Xffffffffffffe3d58(r31); Ln 13 0x000001a8 A (In 110)
        lal      r1,  ^Xffffffffffffe3d58(r1) ; Ln 13 0x000001ac A (In 111)
        addq     r1,  r15,      r1      ; Ln 13 0x000001b0 (In 112)
        lda      r21, -104(r15)        ; Ln 13 0x000001b4 @W4 (In
113)
        lda      r22, -96(r15)         ; Ln 13 0x000001b8 @W5 (In
114)
        lalm     r7,  ^Xffffffffffffc7b58(r31); Ln 13 0x000001bc B (In 115)
        lal      r7,  ^Xffffffffffffc7b58(r7) ; Ln 13 0x000001c0 B (In 116)
        addq     r7,  r15,      r7      ; Ln 13 0x000001c4 (In 117)
        lda      r23, -88(r15)         ; Ln 13 0x000001c8 @W6 (In
118)
        lda      r27, -80(r15)         ; Ln 13 0x000001cc @W7 (In
119)
        lda      r17, -72(r15)         ; Ln 13 0x000001d0 @W8 (In
120)
        lalm     r4,  ^Xfffffffffffffab958(r31); Ln 13 0x000001d4 C (In 121)
        lal      r4,  ^Xfffffffffffffab958(r4) ; Ln 13 0x000001d8 C (In 122)
        addq     r4,  r15,      r4      ; Ln 13 0x000001dc (In 123)
        lda      r6,  -64(r15)         ; Ln 13 0x000001e0 @W9 (In
124)
        lda      r8,  -56(r15)         ; Ln 13 0x000001e4 @W10 (In
125)
        stq      r22, 48(r30)          ; Ln 13 0x000001e8 (In 126)
        stq      r7,  56(r30)          ; Ln 13 0x000001ec (In 127)
        stq      r23, 64(r30)          ; Ln 13 0x000001f0 (In 128)
        stq      r27, 72(r30)          ; Ln 13 0x000001f4 (In 129)
        stq      r17, 80(r30)          ; Ln 13 0x000001f8 (In 130)
        stq      r4,  88(r30)          ; Ln 13 0x000001fc (In 131)
        stq      r6,  96(r30)          ; Ln 13 0x00000200 (In 132)
        stq      r8,  104(r30)         ; Ln 13 0x00000204 (In 133)
        bis      r3,  r31,      r16     ; Ln 13 0x00000208 (In 134)
        bis      r18, r31,      r17     ; Ln 13 0x0000020c (In 135)
        bis      r5,  r31,      r18     ; Ln 13 0x00000210 (In 136)
        bis      r2,  r31,      r19     ; Ln 13 0x00000214 (In 137)
        bis      r1,  r31,      r20     ; Ln 13 0x00000218 (In 138)
        lda      r25, 13(r31)          ; Ln 13 0x0000021c (In 140)
        laum     r9,  SGEMMX@(r31)     ; Ln 13 0x00000220 (In 141)
        sll     r9,  32,      r9       ; Ln 13 0x00000224 (In 142)
        lalm     r9,  SGEMMX@(r9)     ; Ln 13 0x00000228 (In 143)
        lal      r9,  SGEMMX@(r9)     ; Ln 13 0x0000022c (In 144)
        jsr     r26, (r9),      SGEMMX@ ; Ln 13 0x00000230 (In 145)

```

ii) $c = \alpha * \text{matmul}(a,b)$

```

    lalm      r24, ^Xffffffffffffc7bb0(r31); Ln 13 0x00000138 B (In 85)
    lal       r24, ^Xffffffffffffc7bb0(r24); Ln 13 0x0000013c B (In 86)
    addq      r24, r15,      r24      ; Ln 13 0x00000140 (In 87)
    bis       r24, r31,      r22      ; Ln 13 0x00000144 (In 88)
    lalm      r24, ^Xfffffffffffffab9b0(r31); Ln 13 0x00000148 C (In 89)
    lal       r24, ^Xfffffffffffffab9b0(r24); Ln 13 0x0000014c C (In 90)
    addq      r24, r15,      r24      ; Ln 13 0x00000150 (In 91)
    bis       r24, r31,      r4       ; Ln 13 0x00000154 (In 92)
    lda       r5, 120(r31)          ; Ln 13 0x00000158 (In 321)
$LL00007:
    ; Codeblock 8
    lalm      r0, ^Xffffffffffffe3db0(r31); Ln 13 0x0000015c A (In 96)
    lal       r0, ^Xffffffffffffe3db0(r0) ; Ln 13 0x00000160 A (In 97)
    addq      r0, r15,      r0       ; Ln 13 0x00000164 (In 98)
    bis       r0, r31,      r6       ; Ln 13 0x00000168 (In 99)
    bis       r4, r31,      r7       ; Ln 13 0x0000016c (In 100)
    lda       r8, 120(r31)          ; Ln 13 0x00000170 (In 323)
$LL00008:
    ; Codeblock 9
    bis       r22, r31,      r1      ; Ln 13 0x00000174 (In 106)
    ldt       f2, (r1)            ; Ln 13 0x00000178 BLOAD (In
347)
    lda       r2, 119(r31)          ; Ln 13 0x0000017c (In 324)
    cpys      f31, f31,      f3      ; Ln 13 0x00000180 (In 104)
    bis       r6, r31,      r3      ; Ln 13 0x00000184 (In 105)
$LL00014:
    ; Codeblock 15
    ble       r2, $LL00013        ; Ln 0 0x00000188 (In 346)
$LL00009:
    ; Codeblock 10
    subq      r2, 1,      r2       ; Ln 13 0x0000018c (In 116)
    ldt       f1, (r3)            ; Ln 13 0x00000190 (In 110)
    lda       r3, 960(r3)          ; Ln 13 0x00000194 (In 114)
    mult/d    f1, f2,      f0       ; Ln 13 0x00000198 (In 112)
    ldt       f2, 8(r1)           ; Ln 13 0x0000019c BLOAD (In
348)
    addq      r1, 8,      r1       ; Ln 13 0x000001a0 (In 115)
    addt/d    f3, f0,      f3      ; Ln 13 0x000001a4 (In 113)
    bgt       r2, $LL00009        ; Ln 13 0x000001a8 (In 117)
$LL00013:
    ; Codeblock 14
    ldt       f1, (r3)            ; Ln 13 0x000001ac (In 339)
    mult/d    f1, f2,      f0       ; Ln 13 0x000001b0 (In 340)
    subq      r2, 1,      r1       ; Ln 13 0x000001b4 (In 341)
    lda       r1, 960(r3)          ; Ln 13 0x000001b8 (In 343)
    addt/d    f3, f0,      f2       ; Ln 13 0x000001bc (In 344)
$LL00010:
    ; Codeblock 11
    mult/d    f4, f2,      f0       ; Ln 13 0x000001c0 (In 119)
    subq      r8, 1,      r8       ; Ln 13 0x000001c4 (In 123)
    stt       f0, (r7)            ; Ln 13 0x000001c8 (In 120)
    addq      r6, 8,      r6       ; Ln 13 0x000001cc (In 121)
    addq      r7, 8,      r7       ; Ln 13 0x000001d0 (In 122)
    bgt       r8, $LL00008        ; Ln 13 0x000001d4 (In 124)
$LL00011:
    ; Codeblock 12
    lda       r22, 960(r22)        ; Ln 13 0x000001d8 (In 126)
    lda       r4, 960(r4)          ; Ln 13 0x000001dc (In 127)
    subq      r5, 1,      r5       ; Ln 13 0x000001e0 (In 128)
    bgt       r5, $LL00007        ; Ln 13 0x000001e4 (In 129)

```