MIPSpro 7 Fortran 90 Commands and Directives Reference Manual

SR–3907 3.0.1

Copyright © *1997* Cray Research, Inc. All Rights Reserved. This manual or parts thereof may not be reproduced in any form unless permitted by contract or by written permission of Cray Research, Inc.

Portions of this product may still be in development. The existence of those portions still in development is not a commitment of actual release or support by Cray Research, Inc. Cray Research, Inc. assumes no liability for any damages resulting from attempts to use any functionality or documentation not officially released and supported. If it is released, the final form and the time of official release and start of support is at the discretion of Cray Research, Inc.

Autotasking, CF77, CRAY, Cray Ada, CraySoft, CRAY Y-MP, CRAY-1, CRInform, CRI/*Turbo*Kiva, HSX, LibSci, MPP Apprentice, SSD, SUPERCLUSTER, UNICOS, and X-MP EA are federally registered trademarks and Because no workstation is an island, CCI, CCMT, CF90, CFT, CFT2, CFT77, ConCurrent Maintenance Tools, COS, Cray Animation Theater, CRAY APP, CRAY C90, CRAY C90D, Cray C++ Compiling System, CrayDoc, CRAY EL, CRAY J90, CRAY J90se, CrayLink, Cray NQS, Cray/REELlibrarian, CRAY S-MP, CRAY SSD-T90, CRAY T90, CRAY T3D, CRAY T3E, CrayTutor, CRAY X-MP, CRAY XMS, CRAY-2, CSIM, CVT, Delivering the power . . ., DGauss, Docview, EMDS, GigaRing, HEXAR, IOS, ND Series Network Disk Array, Network Queuing Environment, Network Queuing Tools, OLNET, RQS, SEGLDR, SMARTE, SUPERLINK, System Maintenance and Remote Testing Environment, Trusted UNICOS, UNICOS MAX, and UNICOS/mk are trademarks of Cray Research, Inc.

ProDev and Silicon Graphics are trademarks of Silicon Graphics, Inc. DynaWeb is a trademark of Electronic Book Technologies, Inc. MIPSpro is a trademark of MIPS Technologies, Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. X/Open is a registered trademark, and the X device is a trademark, of X/Open Company Ltd.

The UNICOS operating system is derived from UNIX® System V. The UNICOS operating system is also based in part on the Fourth Berkeley Software Distribution (BSD) under license from The Regents of the University of California.

Record of Revision

Version	*Description*
3.0	August 1997. Original Printing. This printing supports the MIPSpro 7 Fortran 90 compiler, release 7.2, running on IRIX systems.

Contents

Page

Preface **xvii**

Related MIPSpro 7 Fortran 90 publications xvii

MIPSpro 7 Fortran 90 messages xviii

MIPSpro 7 Fortran 90 man pages xviii

Related Fortran publications xviii

Related publications xix

Ordering publications xix

Conventions xx

Reader comments xx

Introduction [1] **1**

The f90(1) command 1

The MIPSpro 7 Fortran 90 programming environment 3

Invoking MIPSpro 7 Fortran 90 [2] **5**

-64, -n32 6

-align*n* 8

-ansi 9

-avoid_gp_overflow 9

-C 9

-c 9

-chunk=*integer* 10

-cif 10

-col*n* 10

-cord 10

-cpp 10

Page

-cray_mp (deferred implementation) 10
-d*n* 11
-D*var*[=*def*][,*var*[=*def*]] 11
-default64 11
-dsm 11
-E 12
-extend_source 12
-fb*file*.cfb 12
-feedback*file* 12
-fixedform 12
-freeform 12
-ftpp 13
-g*debug_lvl* 13
-help 13
-I*dir* 13
-INLINE:… 14
-IPA[:…] 14
-i*n* 14
-ignore_suffix 14
-KPIC 15
-keep 15
-L*directory* 15
-LIST:... 15
  -LIST:=*setting* 15
  -LIST:all_options=*setting* 16
  -LIST:notes=*setting* 16
  -LIST:options=*setting* 16
  -LIST:symbols=*setting* 16
-LNO:… 16

Page

General options 17
-LNO:auto_dist=*setting* (Origin series only) 17
-LNO:fission=*n* 17
-LNO:fusion=*n* 18
-LNO:fusion_peeling_limit=*n* 19
-LNO:gather_scatter=*n* 19
-LNO:ignore_pragmas=*setting* 20
-LNO:oinvar=*setting* 20
-LNO:opt=*n* 20
-LNO:outer=*setting* 20
-LNO:vintr=*setting* 20
Transformation options 20
-LNO:blocking=*setting* 21
-LNO:blocking_size=*n1*[,*n2*] 21
-LNO:interchange=*setting* 21
-LNO:ou=*n*, ou_max=*n*, and ou_prod_max=*n* 22
-LNO:ou_further=*n* 22
-LNO:ou_deep=*setting* 23
Cache memory management options 23
-LNO:assoc1=*n*, assoc2=*n*, assoc3=*n*, assoc4=*n* 23
-LNO:cmp1=*n*, cmp2=*n*, cmp3=*n*, cmp4=*n* and dmp1=*n*, dmp2=*n*, dmp3=*n*, dmp4=*n* . . 23
-LNO:cs1=*n*, cs2=n, cs3=*n*, cs4=*n* 24
-LNO:is_mem1=*setting*, is_mem2=*setting*, is_mem3=*setting*, is_mem4=*setting* 24
-LNO:ls1=*n*, ls2=*n*, ls3=*n*, ls4=*n* 24
TLB options 24
-LNO:ps1=*n*, ps2=*n*, ps3=*n*, ps4=*n* 24
-LNO:tlb1=*n*, tlb2=*n*, tlb3=*n*, tlb4=*n* 24
-LNO:tlbcmp1=*n*, tlbcmp2=*n*, tlbcmp3=*n*, tlbcmp4=*n* and tlbdmp1=*n*, tlbdmp2=*n*, tlbdmp3=*n*, tlbdmp4=*n* 25

Page

Prefetch options 25
-LNO:pf*n*=*setting* 25
-LNO:prefetch=*n* 25
-LNO:prefetch_ahead=*n* 25
-LNO:prefetch_manual=*setting* 26
-l*library* 26
-listing 27
-MDupdate[*file*] 27
-macro_expand 28
-mips*n* 28
-MP:... 28
-MP:check_reshape=*setting* 28
-MP:clone=*setting* 29
-MP:dsm=*setting* (Origin series systems only) 29
-mp 29
-mp_schedtype=*mode* 29
-nocpp 30
-noextend_source 30
-nostdinc 30
-OPT:... 31
-OPT:alias=*name* 31
-OPT:cis=*setting* 31
-OPT:cray_ivdep=*setting* 31
-OPT:div_split=*setting* 32
-OPT:fast_bit_intrinsics=*setting* 32
-OPT:fast_complex=*setting* 32
-OPT:fast_exp=*setting* 32
-OPT:fast_nint=*setting* 33
-OPT:fast_sqrt=*setting* 33

Page

-OPT:fast_trunc=*setting* 33
-OPT:fold_reassociate=*setting* 33
-OPT:fold_unsafe_relops=*setting* 34
-OPT:fold_unsigned_relops=*setting* 34
-OPT:got_call_conversion=*setting* 34
-OPT:IEEE_arithmetic=*n* 34
-OPT:IEEE_comparisons=*setting* 35
-OPT:inline_intrinsics=*setting* 35
-OPT:liberal_ivdep=*setting* 35
-OPT:Olimit=*n* 35
-OPT:pad_common=*setting* 35
-OPT:recip=*setting* 36
-OPT:reorg_common=*setting* 36
-OPT:roundoff=*n* 36
-OPT:rsqrt=*setting* 37
-OPT:space=*setting* 37
-OPT:swp=*setting* 37
-OPT:unroll_analysis=*setting* 37
-OPT:unroll_size=*n* 38
-OPT:unroll_times_max=*n* 38
-OPT:wrap_around_unsafe_opt=*setting* 38
-o*out_file* 38
-O*level* 39
-P 40
-pfa, -pfalist 40
-r*real_spec* 40
-r*processor* 41
-S 41

Page

-static 41
-TARG:... 42
-TARG:fp_precise=*setting* 42
-TARG:madd=*setting* 42
-TARG:platform=*ipxx* 43
-TARG:processor=*processor* 43
-TARG:r4krev22=*setting* 43
CPU targeting (cross compiling) using the compiler.defaults file 43
-TENV:... 44
-TENV:align_aggregate=*bytes* 44
-TENV:check_div=*n* 44
-TENV:large_GOT=*setting* 44
-TENV:small_GOT=*setting* 44
-TENV:trapuv=*setting* 45
-TENV:X=*n* 45
-trapuv 46
-U*var* 46
-version 46
-w[*arg*] 46
-woff*num* 46
-x*dirlist* 47
-xgot 47
-- 47
file.suffix[90][*file.suffix*[90]...] 48

Directives [3] **49**

Using directives 49
Directives and command line options 50
Directive range 51

Page

Directive continuation and other considerations 51
LNO directives 51
AGGRESSIVEINNERLOOPFISSION 52
BLOCKABLE 52
BLOCKINGSIZE, NOBLOCKING 53
FISSION, FISSIONABLE, NOFISSION 54
FUSE, FUSABLE, NOFUSION 55
INTERCHANGE, NOINTERCHANGE 56
PREFETCH 57
PREFETCH_MANUAL 58
PREFETCH_REF 58
PREFETCH_REF_DISABLE 59
UNROLL 59
Argument aliasing directives 60
Symbol storage directives 61
Inlining and IPA directives 63

Multiprocessing Directives [4] **65**

Using directives 65
Directive range 65
Directive continuation 66
Loop-level multiprocessing directives: DOACROSS, CHUNK, MP_SCHEDTYPE, and !$ 66
DOACROSS directive 67
AFFINITY clause 68
BLOCKED clause 69
CHUNK clause 69
IF clause 69
LASTLOCAL, LOCAL, and SHARED clauses 70
MP_SCHEDTYPE clause 71
NEST clause 72

Page

REDUCTION clause . . . 72
CHUNK directive . . . 73
MP_SCHEDTYPE directive . . . 74
!$ directive . . . 74
DOACROSS directive examples . . . 75
Analyzing data dependencies for multiprocessing . . . 77
Dependency analysis examples . . . 78
Rewriting data dependencies . . . 81
Work quantum . . . 86
Cache effects and optimization . . . 88
Performing a matrix multiply . . . 88
Optimization costs . . . 89
Load balancing . . . 90
Local common blocks . . . 92
PCF directives . . . 93
BARRIER directive . . . 94
CRITICALSECTION and ENDCRITICALSECTION directives . . . 94
PARALLEL and ENDPARALLEL directives . . . 95
PARALLELDO directive . . . 96
PDO and ENDPDO directives . . . 96
PSECTION[S], SECTION, and ENDPSECTION[S] directives . . . 98
SINGLEPROCESS and ENDSINGLEPROCESS directives . . . 100
Restrictions on the PCF directives . . . 102

Parallel Programming on Origin series systems [5] **105**

Performance tuning on Origin series systems . . . 105
Improving program performance . . . 106
Choosing a tuning method . . . 109
Directives for performance tuning . . . 110
DISTRIBUTE, DISTRIBUTE_RESHAPE, and REDISTRIBUTE . . . 111

Page

DOACROSS . . . 112
AFFINITY clause . . . 112
NEST clause . . . 114
DYNAMIC . . . 116
PAGE_PLACE . . . 117
Using the data distribution directives . . . 118
Regular data distribution . . . 119
Data distribution with reshaping . . . 120
Restrictions on Reshaped Arrays . . . 120
Error detection for reshaped arrays . . . 121
Implementation of reshaped arrays . . . 121
Regular versus reshaped data distribution . . . 124
Examples . . . 125
Distributing columns of a matrix . . . 125
Using data distribution and data affinity scheduling . . . 126
Argument passing . . . 127
Redistributed arrays . . . 128
Irregular distributions and thread affinity . . . 129

CF90 Directives [6] **131**

Using directives . . . 131
Directive continuation . . . 132
Directive range and placement . . . 132
Interaction of directives with the -x command line option . . . 133
Optimization directives . . . 133
Local use of compiler features . . . 133
Check array bounds: BOUNDS and NOBOUNDS . . . 133
Specify source form: FREE and FIXED . . . 135
Autotasking directives (deferred implementation) . . . 135

Page

Mark parallel loop: DOALL (deferred implementation) 136
Mark parallel loop: DOPARALLEL and ENDDO (deferred implementation) 138
Critical region: GUARD and ENDGUARD (deferred implementation) 139
Mark parallel region: PARALLEL and ENDPARALLEL (deferred implementation) 140
Examples (deferred implementation) 140
Read-only variables 140
Array indexed by loop index 141
Read-then-write variables 141
Write-then-read variables and arrays 141
Miscellaneous directives 142
Create identification string: ID 142
Ignore dependencies: IVDEP 144
External name mapping directive: NAME 144

Source Preprocessing [7] 147

General rules 147
Directives 148
#include directive 148
#define directive 149
#undef directive 151
(null) directive 151
Conditional directives 151
#if directive 152
#ifdef directive 153
#ifndef directive 153
#elif directive 153
#else directive 154
#endif directive 154
Predefined macros 154
Command line options 155

Page

Interlanguage Calling [8] 157
External and public names . . . 157
How Fortran 90 handles external and public names . . . 158
Calling a Fortran 90 subprogram from C . . . 158
Calling a C function from Fortran 90 . . . 159
Correspondence of Fortran 90 and C data types . . . 159
Corresponding scalar types . . . 159
Corresponding character types . . . 160
Corresponding array elements . . . 161
Unsupported array arguments . . . 162
How Fortran 90 passes arguments . . . 162
Calling Fortran 90 from C . . . 164
Calling a Fortran 90 subroutine from C . . . 164
Calling a Fortran 90 function from C . . . 166
Calling C from Fortran 90 . . . 167
Calls to C functions . . . 168
Using Fortran 90 common blocks in C code . . . 169
Using Fortran 90 arrays in C code . . . 170
Calls to C using %LOC and %VAL . . . 171
Using %VAL . . . 171
Using %LOC . . . 172
Making C wrappers with mkf2c(1) . . . 172
mkf2c(1) argument assumptions . . . 172
mkf2c(1) character string treatment . . . 173
mkf2c(1) restrictions . . . 175
Using mkf2c(1) and extcentry(1) . . . 175
Makefile considerations . . . 176
Calling assembly language from Fortran 90 . . . 177

Page

Appendix A Library Usage 179

The assign command . . . 179

Options to the assign command . . . 179

Supported FFIO layers . . . 182

FFIO and asynchronous I/O . . . 182

Intrinsic procedures . . . 183

Library routines . . . 183

Library functions . . . 188

Compatibility with sproc . . . 195

Communicating between threads . . . 195

Appendix B Debugging 199

Compiling and running parallel Fortran . . . 199

Using the -static option . . . 199

Profiling a parallel Fortran program . . . 200

Debugging parallel Fortran . . . 201

Other debugging tips for multiprocessed loops . . . 202

Appendix C Differences 205

Model differences . . . 205

Fortran 90 statement differences . . . 205

Functions and procedures . . . 206

Modules . . . 206

I/O library . . . 206

Library functions and procedures . . . 207

Math library . . . 207

Index 209

Page

Figures

Figure 1. f90(1) command example 2

Figure 2. Origin series memory hierarchy 106

Figure 3. Cache behavior and solutions 108

Figure 4. Block distribution 118

Figure 5. Cyclic distribution 119

Figure 6. Implementation of BLOCK distribution 122

Figure 7. Implementation of CYCLIC(1) distribution 123

Figure 8. Implementation of BLOCK-CYCLIC Distribution 124

Figure 9. Correspondence between C and Fortran 90 subscripts 162

Tables

Table 1. Directives 131

Table 2. Autotasking directive *parameter* 137

Table 3. Corresponding Fortran 90 and C Data Types 160

Table 4. How mkf2c(1) treats function arguments 173

Table 5. Summary of System Interface Library Routines 188

Preface

This manual describes the commands and directives for using the MIPSpro 7 Fortran 90 compiler, which is invoked through the `f90`(1) command. The `f90`(1) command can also invoke a source preprocessor, a source lister, and the loader.

The MIPSpro 7 Fortran 90 compiler runs under the IRIX operating system, version 6.2 and later, on Silicon Graphics and Cray Research computer systems.

The MIPSpro 7 Fortran 90 compiler was developed to support the Fortran standards adopted by the American National Standards Institute (ANSI) and the International Standards Organization (ISO). These standards, commonly referred to as *the Fortran 90 standard*, are ANSI X3.198–1992 and ISO/IEC 1539:1991–1. Because the ANSI Fortran 90 standard is a superset of the FORTRAN 77 standard, the MIPSpro 7 Fortran 90 compiler will compile code written to the FORTRAN 77 standard.

> **Note:** The Fortran 90 standard is a substantial revision to the FORTRAN 77 language standard. Because of the number and complexity of the features, the standards organizations are continuing to interpret the Fortran 90 standard for Silicon Graphics, Cray Research, and for other vendors. To maintain conformance to the Fortran 90 standard, Silicon Graphics and Cray Research may need to change the behavior of certain MIPSpro 7 Fortran 90 features in future releases based upon the outcome of the outstanding interpretations to the standard.

Related MIPSpro 7 Fortran 90 publications

This manual is one of a set of manuals that describes the MIPSpro 7 Fortran 90 compiler. The other manuals in the set are as follows:

- *Intrinsic Procedures Reference Manual*, publication SR–2138 (Cray Research publication)
- *Fortran Language Reference Manual, Volume 1*, publication SR–3902 (Cray Research publication)
- *Fortran Language Reference Manual, Volume 2*, publication SR–3903 (Cray Research publication)
- *Fortran Language Reference Manual, Volume 3*, publication SR–3905 (Cray Research publication)

MIPSpro 7 Fortran 90 messages

You can obtain explanations for MIPSpro 7 Fortran 90 compiler messages by using the online `explain`(1) command.

MIPSpro 7 Fortran 90 man pages

In addition to printed and online prose documentation, several online man pages describe aspects of the MIPSpro 7 Fortran 90 compiler. Man pages exist for the library routines, the intrinsic procedures, and several programming environment tools.

You can print copies of online man pages by using the pipe symbol with the `man`(1), `col`(1), and `lpr`(1) commands. In the following example, these commands are used to print a copy of the `explain`(1) man page:

```
% man explain | col -b | lpr
```

Each man page includes a general description of one or more commands, routines, system calls, or other topics, and provides details of their usage (command syntax, routine parameters, system call arguments, and so on). If more than one topic appears on a page, the entry in the printed manual is alphabetized under its primary name; online, secondary entry names are linked to these primary names. For example, `egrep` is a secondary entry on the page with a primary entry name of `grep`. To access `grep` online, you can type `man grep`. To access `egrep` online, you can type either `man grep` or `man egrep`. Both commands display the `grep` man page on your terminal.

Related Fortran publications

The following commercially available reference books are among those that you should consult for more information on the history of Fortran and the Fortran 90 language itself:

- Adams, J., W. Brainerd, J. Martin, B. Smith, and J. Wagener. *Fortran 90 Handbook — Complete ANSI/ISO Reference.* New York, NY: Intertext Publications/Multiscience Press, Inc., 1990.
- Metcalf, M. and J. Reid. *Fortran 90 Explained.* Oxford, UK: Oxford University Press, 1990.
- American National Standards Institute. *American National Standard Programming Language Fortran*, ANSI X3.198–1992. New York, 1992.

- International Standards Organization. *ISO/IEC 1539:1991, Information technology — Programming languages — Fortran.* Geneva, 1991.

Related publications

The following documents contain information that may be useful when using the MIPSpro 7 Fortran 90 compiler:

- *Application Programmer's I/O Guide*, publication SG–2168 (Cray Research publication)
- *MIPSpro Assembly Language Programmer's Guide*, SGI publication 007-2418-003
- *MIPSpro Automatic Parallelizer Programmer's Guide*, SGI publication 007–3572–001
- *MIPSpro Compiling and Performance Tuning Guide*, SGI publication 007-2360-007
- *MIPSpro Fortran 77 Programmer's Guide*, SGI publication 007-2361-005
- *MIPSpro 64-bit Porting and Transition Guide*, SGI publication 007-2391-004
- *Performance Tuning Optimization for Origin2000 and Onyx2*, SGI publication 007-3430-001
- *SpeedShop User's Guide*, SGI publication 007–3311–002

Ordering publications

Silicon Graphics maintains publications information at the following URL:

`http://techpubs.sgi.com/library`

The preceding website contains information that allows you to browse documents online, order documents, and send feedback to Silicon Graphics.

The *User Publications Catalog*, publication CP–0099, describes the availability and content of all Cray Research hardware and software documents that are available to customers. Cray Research customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

Cray Research also has documents available online at the following URL:

`http://www.cray.com/swpubs`

To order a Cray Research or Silicon Graphics document, either call the Distribution Center in Mendota Heights, Minnesota, at +1–612–683–5907, or send a facsimile of your request to fax number +1–612–452–0141.

Cray Research employees may send their orders via electronic mail to `orderdsk` (UNIX system users).

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

Reader comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. You can contact us in any of the following ways:

- Send us electronic mail at the following address:

  `publications@cray.com`

- Contact your customer service representative and ask that an SPR or PV be filed. If filing an SPR, use `PUBLICATIONS` for the group name, `PUBS` for the command, and `NO-LICENSE` for the release name.
- Call our Software Publications Group in Eagan, Minnesota, through the Customer Service Call Center, using either of the following numbers:

  1–800–950–2729 (toll free from the United States and Canada)

  +1–612–683–5600
- Send a facsimile of your comments to the attention of “Software Publications Group” in Eagan, Minnesota, at fax number +1–612–683–5599.

We value your comments and will respond to them promptly.

Introduction [1]

This manual is organized into the following chapters:

- Chapter 1 introduces the content of the manual and provides a general description of the compiler.
- Chapter 2, page 5, describes the `f90`(1) command, which you use to invoke the compiler. This chapter includes information about using the `f90`(1) command line options, CPU targeting, obtaining a listing, and other aspects of compiling with the MIPSpro 7 Fortran 90 compiler.
- Chapter 3, page 49, introduces the compiler directives and describes the general compiler directives that the MIPSpro 7 Fortran 90 compiler recognizes.
- Chapter 4, page 65, describes the multiprocessing directives.
- Chapter 5, page 105, describes the directives that are available to you if you are running the MIPSpro 7 Fortran 90 compiler on an Origin 2000, Origin 200, or Cray Origin 2000 system.
- Chapter 6, page 131, describes Cray Research CF90 compiler directives that are also supported by the MIPSpro 7 Fortran 90 compiler.
- Chapter 7, page 147, describes the source preprocessor.
- Chapter 8, page 157, describes the interlanguage calling conventions used when calling a C/C++ function from a Fortran 90 procedure and a Fortran 90 procedure from a C function.
- Appendix A, page 179, describes library routines available to you from Fortran 90 programs.
- Appendix B, page 199, describes debugging Fortran 90 programs.
- Appendix C, page 205, describes differences between the CF90 compiler, which runs on Cray Research's UNICOS and UNICOS/mk systems, and the MIPSpro 7 Fortran 90 compiler, which runs on IRIX systems.

1.1 The `f90`(1) command

In the following example, the `f90`(1) command is used to invoke the compiler. The `-listing` option is specified to generate a source listing and a cross

reference. File `pgm.f` is the input file. After compilation, you can run this program by entering the output file name as a command. In this example, the default output file name, `a.out`, is used. Figure 1 illustrates this example:

```
% f90 -listing pgm.f
% ./a.out
```

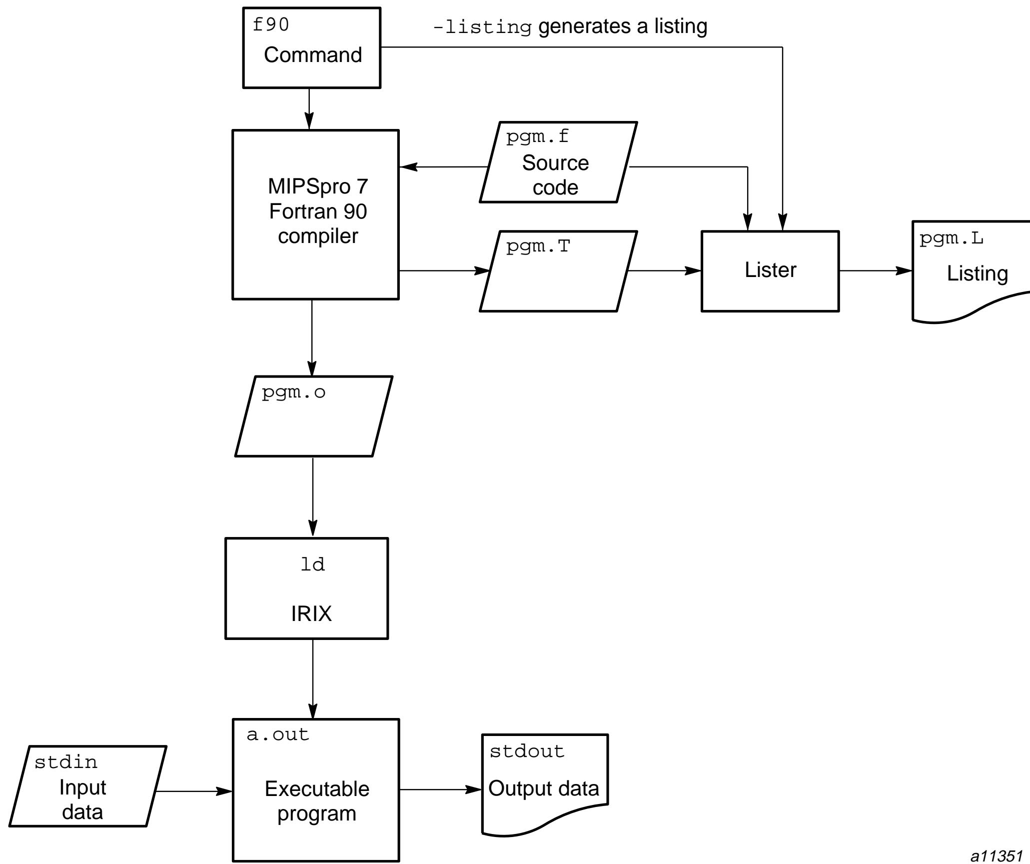


Figure 1. `f90`(1) command example

You can use the options on the `f90`(1) command line to modify the default actions; for example, you can disable the load step. For more information on `f90`(1) command line options, see Chapter 2, page 5.

1.2 The MIPSpro 7 Fortran 90 programming environment

The MIPSpro 7 Fortran 90 compiler is one of many products that form the IRIX programming environment. This environment allows you to develop, debug, and run Fortran 90 codes on your computer system. It includes the following products:

- A loader. By default, the IRIX loader, `ld`(1), is invoked and your program is automatically loaded.
- A preprocessor. You can use the `-ftpp` or `-cpp` options on the `f90`(1) command line to invoke a preprocessor.
- A lister. You can specify the `-listing` option on the `f90`(1) command line to obtain a source listing and a cross reference. You can also invoke a separate lister, `ftnlist`(1).
- The `ftnlint`(1) utility, which checks Fortran 90 programs for possible errors.
- The compiler information file (CIF) tools, which include the `cifconv`(1) command and the libraries. For more information on these see the *Compiler Information File (CIF) Reference Manual*, publication SR–2401. SR–2401 is a Cray Research publication.
- The libraries, which include functions optimized for use on IRIX systems. Information on the individual library routines can be found in the online man pages for each routine. In addition to online man pages, the *Application Programmer's Library Reference Manual*, publication SR–2165, contains printed copies of the library routine man pages and other library information. SR-2165 is a Cray Research publication.

  The intrinsic procedures are implemented within the math library (`libm`), within `libfortran`, and within the compiler itself. The *Intrinsic Procedures Reference Manual*, publication SR–2138, contains printed copies of the online man pages for all the intrinsic procedures. SR–2138 is a Cray Research publication.
- The performance tools contained in SpeedShop. For more information, see the *SpeedShop User's Guide*, a Silicon Graphics publication.
- The archiving tool. An *archive library* is a file that contains one or more routines in object file format (*file*`.o`). When a program calls an object file that is not explicitly included in the program, the loader, `ld`(1), looks for that object file in an archive library. The loader then loads only that object file, not the whole library, and loads it with the calling program.

The archiver creates and maintains archive libraries. It allows you to copy new objects into the library, replace existing objects in the library, move objects within the library, and copy individual objects from the library into individual object files. For more information on the archive library, see the ar(1) man page.

- Object file tools, which allow you to disassemble object files into machine instructions, print information about archive files, and perform other tasks. For more information on these tools, see the following man pages: dis(1), elfdump(1), file(1), nm(1), size(1), and strip(1).
- ftnchop(1), ftnmgen(1) and ftnsplit(1). These commands invoke a program unit problem isolator, a Fortran makefile utility, and a split utility, respectively. For more information on these commands, see the man pages for each.
- Online documentation utilities. The man(1) command allows you to retrieve online man pages. Prose reference text, such as this manual, can be retrieved through the WWW browser supported at your site. Contact your support staff for specific information on retrieving information in this manner.

Invoking MIPSpro 7 Fortran 90 [2]

The `f90`(1) command invokes the MIPSpro 7 Fortran 90 compiler. The following syntax boxes show the `f90`(1) command syntax:

```
f90 [-64 | -n32][-mipsn] file.suffix[90] [file.suffix[90]]...
```

```
f90 [-64 | -n32] [-alignn] [-ansi] [-avoid_gp_overflow] [-C]
   [-c] [-chunk=integer] [-cif] [-coln] [-cord] [-cpp] [-cray_mp]
   [-Dvar[=def][, var[=def]] ...] [-dn] [-default64] [-E]
   [-extend_source] [-fbfile.cfb] [-fe] [-feedback] [-fixedform]
   [-freeform] [-ftpp] [-g[debug_lvl]] [-help] [-I[dir]] [-INLINE:...]
   [-IPA[:...]] [-in] [-ignore_suffix] [-KPIC] [-keep] [-Ldirectory]
   [-LIST:...] [-LNO:...] [-llibrary] [-listing] [-MDupdate[file]]
   [-macro_expand] [-mipsn] [-MP:...] [-mp] [-mp_keep]
   [-mp_schedtype=mode] [-nocpp] [-noextend_source] [-nostdinc]
   [-Olevel] [-OPT:...] [-oout_file] [-P] [-pfa[list]] [-rreal_spec]
   [-rprocessor] [-S] [-static] [-TARG:...] [-TENV:...] [-trapuv] [-Uvar]
   [-version] [-w[arg]] [-woffnum] [-xdirlist] [-xgot] [--] file.suffix[90]
   [file.suffix[90]...]
```

In some cases, multiple options can have an effect on a compiler feature. The following list shows some of the compiler features and the options that affect them:

- Listing control: `-listing`, `-LIST:`.
- Source preprocessing: `-cpp`, `-E`, `-ftpp`, `-macro_expand`, `-nocpp`.
- Setting the compilation environment: `-n32`, `-64`, `-mips`*n*, `-r`*processor*, `-TARG:`.
- Optimization: `-LNO:`, `-OPT:`, `-O`*level*.

Note: The MIPSpro Automatic Parallelization Option is invoked when you specify the `-pfa` command line option. You must be licensed for the MIPSpro Automatic Parallelization Option in order to be able to use this command line option.

Various environment variable settings can affect your compilation. For more information on the environment variables, see the `pe_environ`(5) man page.

The following sections describe the options for the f90(1) command. The last section in this chapter describes CPU targeting.

Note: Some f90(1) command options, for example, -LNO:..., -LIST:..., -MP:... , -OPT:..., -TARG:..., and -TENV:... accept several arguments and allow you to specify a setting for each argument. To specify multiple arguments, either use colons to separate each argument or specify multiple options on the command line. For example, the following command lines are equivalent:

```
f90 -LIST:notes=ON:options=OFF b.f
f90 -LIST:notes=ON -LIST:options=OFF b.f
```

Some argument to options of this type are specified with a setting that will either enable or disable the feature. To enable a feature, specify the argument either alone or with =1, =ON, or =TRUE. To disable a feature, specify the argument with either =0, =OFF, or =FALSE. For example, the following command lines are equivalent:

```
f90 -LNO:auto_dist:blocking=OFF:oinvar=FALSE a.f
f90 -LNO:auto_dist=1:blocking=0:oinvar=OFF a.f
```

For brevity, this manual shows only the ON or OFF settings to arguments, but the compiler also accepts 0, 1, TRUE, and FALSE as settings.

2.1 -64, -n32

Specifies the Application Binary Interface (ABI). Enter either -n32 or -64 to specify an ABI. Specifying -n32 generates 32–bit objects. Specifying -64 generates 64–bit objects.

Note: Certain predefined system defaults can greatly affect your compilation. These include system defaults for your ABI, Instruction Set Architecture (ISA), and processor type. To determine the default ABI for your system, look in file /etc/compiler.defaults. To determine your system's processor, use the hinv(1) command. The -64 and -n32 options can affect the Instruction Set Architecture (ISA) used during compilation. For more information on this interaction, see the -mips*n* option.

When -n32 is specified, the total memory allocation for a program and individual arrays cannot exceed 2 gigabytes (2 GB, or 2,048 MB).

When -64 is specified, the compiler supports arrays that are larger than 2 GB.

As the following example shows, the arrays can be local, global, or dynamically created when compiling with the following command line:

```
f90 -64 -i8 whale.f
```

```
MODULE DEFS
INTEGER, PARAMETER   :: ARRAY_SIZE = 4294967304     ! 0X100000008
INTEGER              :: I(ARRAY_SIZE)
END MODULE


PROGRAM MAIN
USE DEFS
INTEGER, ALLOCATABLE :: J(:)
INTEGER              :: STATUS

ALLOCATE(K(ARRAY_SIZE), STAT=STATUS)

IF (STATUS == 0) THEN
  I(ARRAY_SIZE) = 7;
  J(ARRAY_SIZE) = 8;
  PRINT *, 'I(ARRAY_SIZE) = ', I(ARRAY_SIZE)
  PRINT *, 'J(ARRAY_SIZE) = ', J(ARRAY_SIZE)
  CALL SUB
END IF

END PROGRAM


SUBROUTINE SUB
USE DEFS
INTEGER :: K(ARRAY_SIZE)

K(ARRAY_SIZE) = 9;

PRINT *, 'K(ARRAY_SIZE) = ', K(ARRAY_SIZE)

END SUBROUTINE
```

You must have enough swap space to support the working set size and you must have your shell limit datasize, stack size, and `vmemoryuse` variables set to values large enough to support the sizes of the arrays. For information on these settings, see the `sh`(1) man page.

The following example compiles and runs the preceding code after setting the stack size to a correct value:

```
$uname -a
IRIX64 cydrome 6.2 03131016 IP19
$f90 -64 -mips3 a2.f
$limit
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       65536 kbytes
coredumpsize    unlimited
memoryuse
descriptors     200
vmemoryuse      unlimited
$limit stacksize unlimited
$limit
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       unlimited
coredumpsize    unlimited
memoryuse       754544 kbytes
descriptors     200
vmemoryuse      unlimited
$a.out
7
8
9
```

2.2 -align*n*

Aligns object on specified boundaries. The `-align`*n* specifications are as follows:

Option	Action
`-align32`	Aligns objects larger than 32 bits long on 32-bit boundaries.

`-align64`	Aligns objects larger than 64 bits long on 64-bit boundaries. Default.

When an alignment is specified, objects smaller than the specification are aligned on boundaries that correspond to their sizes. For example, when `align64` is specified, 32-bit and larger objects are aligned on 32-bit boundaries; 16-bit and larger objects are aligned on 16-bit boundaries; and 8-bit and larger objects are aligned on 8-bit boundaries.

2.3 `-ansi`

Causes the compiler to generate messages when it encounters source code that does not conform to the Fortran 90 standard.

2.4 `-avoid_gp_overflow`

Adjusts internal settings with the intent of avoiding global symbol table (GOT) overflow. For more information on the GOT, see the `-xgot` option, the `gp_overflow`(5) man page, and the *What should I do about a GOT overflow?* question in the FAQ section of the `dso`(5) man page.

2.5 `-C`

Performs run-time subscript range checking. This functionality can also be obtained by specifying `-check_bounds`. Subscripts that are out of range cause fatal run–time errors.

2.6 `-c`

Disables the load step and writes the binary object file to ***file***`.o`.

For example, the following command line produces file `more.o`:

```
% f90 -c more.f
```

2.7 -chunk=*integer*

Specifies the number of loop iterations per chunk. For scheduling purposes, the iterations of a loop are broken up into pieces.

Enter a nonzero, unsigned, positive integer for *integer*.

2.8 -cif

Specifies that the compiler should write a compiler information file.

2.9 -col*n*

Specifies the line width for fixed-format source lines. Enter 72 or 80 for *n*. By default, fixed-format lines are 72 characters wide. For more information on specifying line length, see the -extend_source and -noextend_source options.

2.10 -cord

Runs the procedure rearranger, cord(1), on the resulting file after loading. The rearrangement is done to reduce virtual memory paging and/or instruction cache misses.

For more information on procedure rearranging, see the cord(1), pixie(1), and prof(1) man pages.

2.11 -cpp

Runs the cpp source preprocessor on all input source files before compiling. For more information on controlling preprocessing, see the -ftpp, -E, and -nocpp options. For information on enabling macro expansion, see the -macro_expand option. By default, no preprocessing is performed.

2.12 -cray_mp (deferred implementation)

Specifies that the Autotasking directives (with the !MIC prefix) should be honored. These directives are also implemented in the Cray Research CF90

compiler on UNICOS systems. For more information on these directives, see Chapter 6.

2.13 `-d`*n*

Specifies the `KIND` specification used for objects declared `DOUBLE COMPLEX` and `DOUBLE PRECISION`, as follows:

Option	`KIND` value
`-d8`	Uses `REAL(KIND=8)` for objects declared as `DOUBLE PRECISION`. Uses `COMPLEX(KIND=8)` for objects declared `DOUBLE COMPLEX`. Default.
`-d16`	Uses `REAL(KIND=16)` for objects declared as `DOUBLE PRECISION`. Uses `COMPLEX(KIND=16)` for objects declared `DOUBLE COMPLEX`.

2.14 `-D`*var*[=*def*][`,`*var*[=*def*]] ...

Defines variables used for source preprocessing as if they had been defined by a `#define` directive. If no *def* is specified, 1 is used. For information on undefining variables, see the `-U`*var* option.

2.15 `-default64`

Sets the sizes of default integer, real, logical, and double precision objects to be the same as if the program were executing on a Cray Research UNICOS system. This option causes the following options to go into effect: `-r8`, `-i8`, `-d16`, and `-64`.

2.16 `-dsm`

Specifies that directives specific to Origin series systems should be honored. For more information on these directives, see Chapter 5, page 105.

2.17 -E

Runs the ftpp source preprocessor on all input Fortran source files, before compiling, and writes the preprocessed output to stdout. The output file contains line directives. This option overrides the -nocpp option. For more information on controlling source preprocessing, see the -cpp, -ftpp, -macro_expand, and -nocpp options.

2.18 -extend_source

Specifies 132-character line length for fixed-format source lines. By default, fixed-format lines are 72 characters wide. For more information on controlling line length, see the -col*n* option

2.19 -fb*file*.cfb

Specifies the feedback file to be used. The file suffix must be .cfb. For more information on procedure rearranging and on producing feedback files, see cord(1), pixie(1), and prof(1).

2.20 -feedback*file*

Specifies the name of a feedback file. This option is used with the -cord option. For more information on procedure rearranging and on producing feedback files, see cord(1), pixie(1), and prof(1).

2.21 -fixedform

Treats all input source files, regardless of suffix, as if they were written in fixed source form. By default, only input files suffixed with .f or .F are assumed to be written in fixed source form.

2.22 -freeform

Treats all input source files, regardless of suffix, as if they were written in free source form. By default, only input files suffixed with .f90 or .F90 are assumed to be written in free source form.

2.23 -ftpp

Runs the ftpp source preprocessor on input Fortran source files that are suffixed with .F or .F90 before compiling. For more information on controlling preprocessing, see the -cpp, -E, and -nocpp options. For information on enabling macro expansion, see the -macro_expand option.

2.24 -g*debug_lvl*

Generates debugging information and establishes a debugging level. Enter one of the following:

Option	Support
-g0	No debugging information produced. Default.
-g2, -g	Information for symbolic debugging is produced, and optimization is disabled.
-g3	Information for symbolic debugging of fully optimized code is produced. The debugging information produced may be inaccurate. This option can be used in conjunction with the -O, -O1, -O2, and -O3 options.

2.25 -help

Lists all available options. The compiler is not invoked.

2.26 -I*dir*

Specifies a directory to be searched for files named in INCLUDE lines in the Fortran source file, for files named in #include source preprocessing directives, and *file*.mod files whose names do not begin with a slash (/) character. Files are searched in the following order: first, in the directory that contains the input file; second, in the directories specified by *dir*; and third, in the standard directory, /usr/include.

2.27 -INLINE:...

Specifies actions for the standalone inliner. These options control the application of intrafile subprogram inlining when interprocedural analysis (IPA) is not enabled.

If you have included inlining directives in your source code, the `-INLINE` option must be specified in order for those directives to be honored.

For more information on the individual options in this group, see `ipa`(5).

2.28 -IPA[:...]

Specifies actions for the interprocedural analyzer (IPA). These options control the application of interprocedural analysis and optimization, including inlining, common block array padding, constant propagation, dead function elimination, alias analysis, and other features. Specify `-IPA` with no arguments to invoke the interprocedural analysis phase with default options.

If you have included IPA directives in your source code, the `-IPA` option must be specified in order for those directives to be honored.

If you compile and load in distinct steps, you must use at least `-IPA` for the compile step, and you must specify `-IPA` and the individual options in the group for the load step. For more information on the individual options in this group, see `ipa`(5).

2.29 -i*n*

Specifies the length of default integer constants, default integer variables, and logical quantities. Specify one of the following:

Option	Action
`-i4`	Specifies 32-bit (4 byte) objects. Default.
`-i8`	Specifies 64-bit (8 byte) objects.

2.30 -ignore_suffix

Compiles all files as if they were Fortran source files. By default, the `f90`(1) command determines the type of processing necessary for an input file based in its suffix. Files that end in `.c`, for example, are compiled by `cc`(1). When

-ignore_suffix is specified, the compiler processes all files named as if they were all Fortran source files, regardless of suffix.

2.31 -KPIC

Generates position-independent code (PIC), which is necessary for programs loaded with dynamic shared libraries. Enabled by default.

To disable the generation of PIC, specify the -nonshared option.

2.32 -keep

Writes intermediate compilation files to *file*.B and *file*.s and retains them after compilation is finished.

2.33 -L*directory*

Changes the library search algorithm. For *directory*, specify the path name to a directory that should be searched before using the default system libraries. You can specify multiple -L options on the command line. The library search algorithm searches these directories in the order given.

2.34 -LIST:...

Specifies the information that is written to the listing file, *file*.l. This information is also written to the assembly listing file if the -S option is also in effect.

For an alternative method of obtaining a listing, see the -listing option.

The following sections describe the individual -LIST: options.

2.34.1 -LIST:=*setting*

Writes or suppresses the listing file. Enter ON or OFF for *setting*.

If one or more -LIST options are enabled, the listing file is written. By default, the listing file contains a list of options in effect during compilation.

2.34.2 -LIST:all_options=*setting*

Writes or suppresses the list of supported options in the listing file. Enter ON or OFF for *setting*. The default is OFF.

2.34.3 -LIST:notes=*setting*

Writes or suppresses notes regarding various optimization phases to the assembly listing file. Must be specified in conjunction with -S. Enter ON or OFF for *setting*. The default is ON.

2.34.4 -LIST:options=*setting*

Writes or suppresses a listing of the options in effect during compilation to the listing file. Enter ON or OFF for *setting*. The default is OFF.

2.34.5 -LIST:symbols=*setting*

Writes or suppresses a listing of the symbols (variables) used in the compilation to the listing file. Enter ON or OFF for *setting*. The default is OFF.

2.35 -LNO:...

Specifies options and transformation performed on loop nests. The -LNO options are performed only if -O3 is also specified on the f90(1) command line.

The arguments to -LNO are divided into the following groups:

- General options
- Transformation options
- Cache memory management options
- TLB options
- Prefetch options

For information on the LNO options that are in effect during a compilation, use the -LIST option.

The following sections describe the individual LNO options.

2.35.1 General options

The general options are as follows:

2.35.1.1 `-LNO:auto_dist=`*setting* (Origin series only)

Distributes local arrays and arrays in common blocks that are accessed in parallel. Enter `ON` or `OFF` for *setting*. The default is `OFF`.

When `-LNO:auto_dist=ON`, the compiler uses a heuristic to distribute local and `COMMON` arrays that are accessed in parallel. The heuristic is based on access patterns inside the routines that define the arrays; access patterns of arrays used as dummy arguments are ignored. This optimization works with either automatic parallelism or parallelism expressed through directives. This optimization is always safe, does not affect the layout of arrays in virtual space and does not incur addressing overhead.

Example:

```
      PROGRAM FRED
      REAL A(1000,100)
      COMMON A
!$DOACROSS LOCAL(J)
      DO I=1,N
        DO J=1,N
          A(J,I) = 0.0
        END DO
      END DO
      END
```

In the preceding code fragment, every processor accesses a block of iterations of parallel loop `I`. This implies that every processor will zero a block of columns of array `A`. When this option is enabled, the compiler distributes the array using the `!$DISTRIBUTE A(*,BLOCK)` directive so that each processor accesses data local to its own memory. The algorithm uses a heuristic that might not pick the best distribution. In particular, if arrays are accessed differently in different subroutines, a majority rules algorithm applies. This option is useful for programs that are not written with data distribution in mind. For more information on the `DISTRIBUTE` directive, see Section 5.2.1, page 111.

2.35.1.2 `-LNO:fission=`*n*

Controls loop fission. Enter 0, 1, or 2 for *n*. The default is 1.

Loop fission is an optimization process by which a loop is divided into smaller, independent loops. This can improve register use for large inner loops. It also enables other optimizations, such as loop interchange and blocking, to execute more efficiently. Consider the following loop:

```
DO I ...
   DO J1 ...
      ...
   ENDDO
   DO J2 ...
      ...
   ENDDO
ENDDO
```

With loop fission, the preceding loop is transformed into the following two loops:

```
DO I1 ...
   DO J1 ...
      ...
   ENDDO
ENDDO
DO I2 ...
   DO J2 ...
      ...
   ENDDO
ENDDO
```

`fission=0` disables loop fission. `fission=1` performs normal fission as necessary. `fission=2` specifies that fission be tried before fusion.

If `-LNO:fission=`*n* and `-LNO:fusion=`*n* are both set to 1 or to 2, fusion is performed.

2.35.1.3 `-LNO:fusion=`*n*

Controls loop fusion. *Loop fusion* is an optimization process by which two small loops are transformed into one big loop. Loop fusion can lower the number of memory references and improve cache behavior. It also enables other optimizations, such as loop interchange and cache blocking, to execute more efficiently. Enter 0, 1, or 2 for *n*. The default is 1. The loops to be fused need not have identical iteration counts, but the iteration counts should be approximately the same.

Consider the following loop:

```
DO I = 1,N
   DO J = 1,N
      A(I,J) = B(I,J) + B(I,J-1) + B(I,J+1)
   END DO
END DO
DO I = 1,N
   DO J = 1,N
      B(I,J) = A(I,J) + A(I,J-1) + A(I,J+1)
   END DO
END DO
```

With loop fusion, the preceding loops are transformed into the following loop:

```
DO I=1,N
   A(I,1) = B(I,0) + B(I,1) + B(I,2)
   DO J = 2,N
      A(I,J) = B(I,J) + B(I,J-1) + B(I,J+1)
      B(I,J-1) = A(I,J-2) + A(I,J-1) + A(I,J)
   END DO
   B(I,N) = A(I,N-1) + A(I,N) + A(I,N+1)
END DO
```

`fusion=0` disables loop fusion. `fusion=1` performs standard outer loop fusion. `fusion=2` specifies that outer loops should be fused, even if it means partial fusion. The compiler attempts fusion before fission. The compiler performs partial fusion if not all levels can be fused in the multiple-level fusion.

If `-LNO:fission=`***n*** and `-LNO:fusion=`***n*** are both set to 1 or to 2, fusion is performed.

The `fusion=` options affect the singly nested loops produced by the compiler.

2.35.1.4 `-LNO:fusion_peeling_limit=`*n*

Sets the limit for the number of iterations allowed to be peeled in fusion, where $n \geq 0$. By default, n=5.

2.35.1.5 `-LNO:gather_scatter=`*n*

Performs gather-scatter optimizations. Enter 0, 1, or 2 for *n*. The default is 1.

`gather_scatter=0` disables all gather-scatter optimization. `gather_scatter=1` performs gather-scatter optimizations on non-nested `IF` statements. `gather_scatter=2` performs multi-level gather-scatter optimizations.

2.35.1.6 `-LNO:ignore_pragmas=`*setting*

Specifies that the command line options override directives in the source file. Specify either `ON` or `OFF` for *setting*. The default is `ignore_pragmas=OFF`.

By default, directives within a file override command line options.

2.35.1.7 `-LNO:oinvar=`*setting*

Controls outer loop hoisting. *Hoisting* is the process by which invariant statements or expressions are taken out of a loop. The compiler looks for expressions that vary in the inner loop but are invariant in an outer loop. The compiler precomputes all the invariant expressions and stores them in a temporary vector. All references to the expression in the inner loop are replaced by loads from the vector. Enter `ON` or `OFF` for *setting*. The default is `oinvar=ON`.

2.35.1.8 `-LNO:opt=`*n*

Controls the LNO optimization level. Enter either 0 or 1 for *n*. The default is 1.

`opt=0` disables nearly all loop nest optimization. `opt=1` performs full LNO transformations.

2.35.1.9 `-LNO:outer=`*setting*

Enables or disables outer loop fusion. Enter `ON` or `OFF` for *setting*. The default is `outer=ON`.

2.35.1.10 `-LNO:vintr=`*setting*

Specifies that vectorizable versions of the math intrinsic functions should be used. Enter `ON` or `OFF` for *setting*. The default is `vintr=ON`.

For information on the math intrinsic functions, see `man`(3M).

2.35.2 Transformation options

The loop transformation options allow you to control cache blocking, loop unrolling, and loop interchange.

2.35.2.1 `-LNO:blocking=`*setting*

Specify `blocking=OFF` to disable cache blocking. Enter `ON` or `OFF` for *setting*. The default is `blocking=ON`.

2.35.2.2 `-LNO:blocking_size=`*n1*[,*n2*]

Specifies a blocking size that the compiler must use when performing any blocking. Specify a value for *n2* when using a 2-level cache. For *n1* or *n2*, enter an integer number that represents the number of iterations.

2.35.2.3 `-LNO:interchange=`*setting*

Specifies whether or not loop interchange optimizations are performed.

Loop nests such as the following benefit from loop interchange optimizations:

```
DO I ...
   DO J ...
      DO K ...
         A(J,K) = A(J, K) + B(I,K)
      END DO
   END DO
END DO
```

In the preceding loop, each iteration of loop `K` requires two loads and one store. Also, if the loop bounds are large, every memory reference results in a cache miss.

With `-LNO:interchange=ON`, the loop is transformed into the following loop:

```
DO K ...
   DO J ...
      DO I ...
         A(J,K) = A(J,K) + B(I,K)
      END DO
   END DO
END DO
```

In the new loop, note that `A(J,K)` is a loop invariant entity; only one load is needed per iteration. The new loop is also more efficient with regard to cache management.

Specifying `-LNO:interchange=OFF` disables loop interchange optimizations. Enter `ON` or `OFF` for *setting*. The default is `interchange=ON`.

2.35.2.4 `-LNO:ou=`*n*, `ou_max=`*n*, and `ou_prod_max=`*n*

Specifies aspects of loop unrolling. When a loop is *unrolled*, the compiler makes copies of the loop body and executes them in sequence. The compiler performs some loop unrolling by default, but this option let you override default system assumptions.

Specifying `ou=`*n* indicates that all outer loops for which unrolling is legal should be unrolled by *n*, where *n* is an integer. The compiler unrolls loops by this amount or not at all.

Specifying `ou_max=`*n* indicates that the compiler can unroll as many as *n* copies per loop, but no more.

Specifying `ou_prod_max=`*n* indicates that the product of unrolling of the various outer loops in a given loop nest is not to exceed *n*.

Example. The following loop is compiled with `-LNO:ou=2`:

```
DO I = 1,N
   DO J = 1,N
      A(J,I) = A(J,I) + B(J)
   END DO
END DO
```

After unrolling, the loop is as follows:

```
DO I = 1,N-1,2
   DO J = 1,N
      A(J,I) = A(J,I) + B(J)
      A(J,I+1) = A(J,I+1) + B(J)
   END DO
END DO
DO I = I,N
   DO J = 1,N
      A(J,I) = A(J,I) + B(J)
   END DO
END DO
```

The advantage of unrolling, in the example, is that there is no need to load `B(J)` `N` times but instead `N/2` times.

2.35.2.5 `-LNO:ou_further=`*n*

Specifies whether or not the compiler performs outer loop unrolling on wind-down loops. When unrolling a loop with *n* iterations *u* times, the

compiler must generate a wind–down loop to handle cases in which *n* is not a multiple of *u*. The ***wind-down loop*** handles the extra iterations at the end. The wind-down loop will have at most *u*-1 iterations. When the unrolling factor, *u*, is large, it may be beneficial to unroll the wind-down loop itself. When this option is set to *n*, the compiler unrolls a wind-down loop only if the original loop was unrolled by at least a factor of *n*.

You can disable additional unrolling by specifying `-LNO:ou_further=999999`. Unrolling is enabled as much as is sensible by specifying `-LNO:ou_further=3`.

2.35.2.6 `-LNO:ou_deep=`*setting*

Specifies that for loops with 3-deep, or deeper, loop nests, the compiler should outer unroll the wind-down loops that result from outer unrolling loops further out. This results in a large executable file, but it generates much better code whenever wind down loop execution costs are important. The default is `ou_deep=ON`.

2.35.3 Cache memory management options

LNO does several transformations, such as blocking and loop interchange, to improve the cache behavior of programs. When performing these transformations, LNO assumes that the target platform has certain cache characteristics. The following options allow advanced users to change the default cache characteristics, thereby giving finer control over the optimizations that LNO performs.

The numbering in these arguments starts with the cache level closest to the processor and works outward.

2.35.3.1 `-LNO:assoc1=`*n*, `assoc2=`*n*, `assoc3=`*n*, `assoc4=`*n*

Specifies cache set associativity. For a fully associative cache, such as main memory, set *n* to any sufficiently large number, such as 128. Specifying *n*=0 indicates that there is no cache at that level.

2.35.3.2 `-LNO:cmp1=`*n*, `cmp2=`*n*, `cmp3=`*n*, `cmp4=`*n* and `dmp1=`*n*, `dmp2=`*n*, `dmp3=`*n*, `dmp4=`*n*

Specifies, in processor cycles, the time for a clean or dirty miss to the next outer level of the memory hierarchy. This number is approximate because it depends upon a clean or dirty line, read or write miss, etc. Specifying *n*=0 indicates that there is no cache at that level.

2.35.3.3 `-LNO:cs1=`*n*, `cs2=n`, `cs3=`*n*, `cs4=`*n*

Specifies the cache size. The value *n* can be 0, or it can be a positive integer followed by one of the following letters: `k`, `K`, `m`, or `M`. This specifies the cache size in kilobytes or megabytes. Specifying *n*=0 indicates that there is no cache at that level. The default cache size depends on your system. You can use the `-LIST:` option to see the default cache sizes used during compilation.

2.35.3.4 `-LNO:is_mem1=`*setting*, `is_mem2=`*setting*, `is_mem3=`*setting*, `is_mem4=`*setting*

Specifies that certain memory hierarchies should be modeled as memory, not cache. Enter `ON` or `OFF` for *setting*. The default is `OFF` for each option.

If an `is_mem`*n*=*setting* setting is specified, the corresponding `assoc`*n*=*n* specification is ignored. Blocking can be attempted for this memory hierarchy level, and blocking appropriate for memory, rather than cache, is applied. No prefetching is performed, and any prefetching options are ignored. Any `cmp`*n*=*n* and `dmp`*n*=*n* options on the command line are ignored.

2.35.3.5 `-LNO:ls1=`*n*, `ls2=`*n*, `ls3=`*n*, `ls4=`*n*

Specifies the line size, in bytes. This is the number of bytes, specified in the form of an integer number, *n*, that are moved from the memory hierarchy level further out to this level on a miss. Specifying *n*=0 indicates that there is no cache at that level.

2.35.4 TLB options

The following options control the TLB. The TLB is a cache for the page table. Blocking for the TLB can improve cache performance. The following options control how the loop nest optimizer models the TLB when performing transformations. The TLB hardware is assumed to be fully associative.

2.35.4.1 `-LNO:ps1=`*n*, `ps2=`*n*, `ps3=`*n*, `ps4=`*n*

Specifies the number of bytes in a page. Enter an integer for *n*. The default *n* depends on your system hardware.

2.35.4.2 `-LNO:tlb1=`*n*, `tlb2=`*n*, `tlb3=`*n*, `tlb4=`*n*

Specifies the number of entries in the TLB for this cache level. Enter an integer for *n*. The default *n* depends on your system hardware.

2.35.4.3 `-LNO:tlbcmp1=`*n*, `tlbcmp2=`*n*, `tlbcmp3=`*n*, `tlbcmp4=`*n* and `tlbdmp1=`*n*, `tlbdmp2=`*n*, `tlbdmp3=`*n*, `tlbdmp4=`*n*

Specifies the number of processor cycles it takes to service a clean or dirty TLB miss. Enter an integer for *n*. The default *n* depends on your system hardware.

2.35.5 Prefetch options

The following options control use of the prefetch operation. When an LNO prefetch option is enabled, the compiler examines the source code for memory references that may cause cache misses. It then inserts prefetches into the generated code so that the prefetches are performed ahead of the corresponding memory references.

The `-mips4` and `-r10000` options must be in effect in order for the LNO prefetch options to be honored.

2.35.5.1 `-LNO:pf`*n*`=`*setting*

Selectively disables and enables prefetching for cache level *n*, where $1 \le n \le 4$. Enter `ON` or `OFF` for *setting*.

When `-r10000` is in effect, `pf1=ON` and `pf2=ON` by default. At any other `-r`*n* setting, `OFF` is in effect for all cache levels.

2.35.5.2 `-LNO:prefetch=`*n*

Specifies levels of prefetching.

`prefetch=0` disables all prefetching. This is the default when `-r4000`, `-r5000`, or `-r8000` is in effect.

`prefetch=1` enables conservative prefetching. This is the default when `-r10000` is in effect.

`prefetch=2` enables aggressive prefetching.

2.35.5.3 `-LNO:prefetch_ahead=`*n*

Prefetches the specified number of cache lines ahead of the reference. The default is 2.

2.35.5.4 `-LNO:prefetch_manual=`*setting*

Specifies whether manual prefetches (through directives) should be respected or ignored. Enter `ON` or `OFF` for *setting*.

`prefetch_manual=OFF` ignores manual prefetches. This is the default for R8000 and earlier processors.

`prefetch_manual=ON` respects manual prefetches. This is the default for R10000 and later processors.

2.36 `-l`*library*

Searches the library named `lib`*library*`.a` or `lib`*library*`.so`. Libraries are searched in the order given on the command line.

If you are using another compiler, for example the C compiler, to load Fortran 90 object files, you need to explicitly specify to the C compiler that the Fortran libraries be loaded.

The following table shows the Fortran libraries that the `f90`(1) command loads by default:

`-l` option	Link library	Content
`-l fortran`	`/usr/lib*/libfortran.so`	Intrinsic procedure, I/O, multiprocessing, IRIX interface, and indexed sequential access method library for shared loading and compiling.
`-l m`	`/usr/lib*/libm.so`	Mathematics library.

Example 1. In the following example, the `cc`(1) command loads Fortran 90 object files. The `-l` option loads the Fortran library files:

```
cc -o myprog main.o rest.o -lfortran -lm
```

See the `ld`(1) man page for information on specifying the `-l` option.

Example 2. You may need to specify libraries when you use IRIX system packages that are not part of a particular language. Most of the man pages for these packages list the required libraries. For example, the getwd(3C) subroutine requires the BSD compatibility library libbsd.a. Specify this library as follows:

```
% f90 main.o more.o rest.o -lbsd
```

Example 3. To load the SGI/Cray Scientific Library (SCSL), specify one of the following command lines:

```
% f90 -lscs sci.f
% f90 -lscs_mp mpsci.f
```

The -lscs_mp option used in the preceding command line loads the multiprocessed version of SCSL, which is supported on Origin series systems.

Example 4. To specify a library created with the archiver, type in the path name of the library as follows:

```
% f90 main.o more.o rest.o libfft.a
```

Note: The loader searches libraries in the order you specify. Therefore, if you have a library, for example, libfft.a that uses data or procedures from -lfourier, you **must** specify libfft.a first.

2.37 -listing

Writes a source code listing and a cross reference listing to *file*.L.

For an alternative method of obtaining a listing, see the -LIST: option.

2.38 -MDupdate[*file*]

Updates makefile dependencies in *file*. The file can be included by smake(1) and pmake(1) to get dependencies. Files named on INCLUDE statements and modules named on USE statements are updated.

When *file* is not specified, the lines updated are those that begin with the name of the output file followed by a colon and end with a distinctive make(1) comment.

When *file* is specified, *file* is updated during compilation to contain header, library, and run-time make(1) dependencies for the output file.

For example, assume that file `foo.f90` contains the following two lines:

```
INCLUDE "bar.h"
USE mod
```

The updated file will contain a line similar to the following:

```
foo.o : bar.h MOD.mod
```

2.39 -macro_expand

Enables macro expansion in preprocessed Fortran source files throughout each file.

When `-macro_expand` is specified, macro expansion occurs throughout the source file. When `-macro_expand` is not specified, macro expansion is limited to preprocessor (#) directives in files processed by FTPP.

2.40 -mips*n*

Specifies the Instruction Set Architecture (ISA). Enter `-mips3` to specify the MIPS III instruction set. Enter `-mips4` to specify the MIPS IV instruction set. For information on the default setting for your system, see file `/etc/compiler.defaults`.

The `-mips`*n* option interacts with the `-64` and `-n32` options.

2.41 -MP:...

Specifies individual multiprocessing options that provide fine control over certain optimizations.

To specify all the `-MP:`... options, specify the `-mp` option on the command line.

The following sections describe the `-MP:` options.

2.41.1 -MP:check_reshape=*setting*

Enables or disables run time consistency checks across procedure boundaries when passing reshaped arrays (or portions thereof) as actual arguments. Enter `ON` or `OFF` for *setting*. The default is `check_reshape=OFF`.

2.41.2 `-MP:clone=`*setting*

Enables or disables autocloning. Enter `ON` or `OFF` for *setting*. The compiler automatically duplicates procedures that are called with reshaped arrays as actual arguments for the incoming distribution. If you have explicitly specified the distribution on all relevant dummy arguments, you can disable autocloning. The consistency checking of the distribution between actual and dummy arguments is not affected by this option and is always enabled. The default is `clone=ON`.

2.41.3 `-MP:dsm=`*setting* (Origin series systems only)

Enables or disables recognition of the distributed shared memory directives described in Chapter 5, page 105. Enter `ON` or `OFF` for *setting*. When the `-mp` option is also in effect, the default is `dsm=ON`. When the `-mp` option is not in effect, the default is `dsm=OFF`.

When the `-mp` option is specified on the `f90`(1) command line, the compiler silently generates bookkeeping information in the `rii_files` directory. This information is used to implement data distribution directives, as well as perform consistency checks of these directives across multiple source files. To disable the processing of the data distribution directives and not generate the `rii_files`, compile the program with the `-MP:dsm=off` option.

2.42 `-mp`

Enables all the distributed shared memory directives described in Chapter 5, page 105, and all multiprocessing options described for the `-MP:` option.

2.43 `-mp_schedtype=`*mode*

Specifies a mode for scheduling work among the participating tasks in loops.

Specifying this option has the same effect as putting a `!$MP_SCHEDTYPE=`*mode* directive at the beginning of the file. Enter one of the following for *mode*:

mode	Action
`DYNAMIC`	Breaks the iterations into pieces; the size is specified by the `-chunk=`*integer* option. As each process finishes a piece, it enters a critical section

	and obtains the next available piece. For more information, see the `-chunk=`***integer*** option.
`GSS`	Schedules pieces according to the sizes of the pieces awaiting to execution.
`INTERLEAVE`	Breaks the iterations into pieces; the size is specified by the `-chunk=`***integer*** option. Execution of the pieces is interleaved among the processes. For more information, see the `-chunk=`***integer*** option.
`RUNTIME`	Schedules pieces according to information contained in the `MP_SCHEDTYPE` and `CHUNK` environment variables.
`SIMPLE`	Divides the iterations among processes by dividing them into contiguous pieces and assigning one piece to each process.

For more information on environment variables, these modes, and their effects, see `pe_environ`(5).

2.44 -nocpp

Disables the source preprocessor. See the `-cpp`, `-E`, and `-ftpp` options for more information on controlling preprocessing.

2.45 -noextend_source

Restricts Fortran source code lines to columns 1 through 72. See the `-col`***n*** and `-extend_source` options for more information on controlling line length.

2.46 -nostdinc

Directs the system to skip the standard directory, `/usr/include`, when searching for `#include` files.

2.47 -OPT:...

Controls miscellaneous optimizations. These options override defaults based on the main optimization level.

To enable any of these options that accept ON or OFF as arguments, you only need to specify the option name itself. The =ON is not needed. For example, the following specifications are equivalent:

-OPT:div_split=ON and -OPT:div_split

For information on inlining, see the -INLINE:... option. For information on loop nest optimization, see the -LNO:... option. For information on interprocedural optimization, see the -IPA:... option.

2.47.1 -OPT:alias=*name*

Specifies the pointer aliasing model to be used. By specifying one of the following for *name*, the compiler is able to make assumptions throughout the compilation:

name	Assumption
ANY	Any two memory references can be aliased unless the compiler can determine otherwise. This is the default.
COMMON_SCALAR	Scalar variables that are defined in a common block along with array variables are not referenced or modified by any accesses of the array variables.

2.47.2 -OPT:cis=*setting*

Converts SIN/COS pairs with the same argument to a single call that calculates both values at once. Enter ON or OFF for *setting*. The default is cis=ON.

2.47.3 -OPT:cray_ivdep=*setting*

Specifies that the compiler should use Cray Research semantics when a !DIR$ IVDEP directive is encountered. The compiler ignores all loop iteration dependencies. Enter ON or OFF for *setting*. The default is cray_ivdep=OFF, which directs the compiler to use Silicon Graphics semantics when a !DIR$ IVDEP directive is encountered.

For more information on the `!DIR$ IVDEP` directive, see Section 6.5.2, page 144.

2.47.4 -OPT:div_split=*setting*

Enables or disables the calculation of x/y as $x \times (1.0/y)$. Enter `ON` or `OFF` for *setting*. The default is `div_split=OFF`.

This is typically enabled by the `-OPT:IEEE_arithmetic=3` option. Also see the `-OPT:recip` option. This option should be used with caution because it produces less accurate results.

2.47.5 -OPT:fast_bit_intrinsics=*setting*

`fast_bit_intrinsics=ON` turns off the check for the bit count being within range for Fortran bit intrinsics (for example, `BTEST` and `ISHFT`). Enter `ON` or `OFF` for *setting*. The default is `fast_bit_intrinsics=OFF`.

2.47.6 -OPT:fast_complex=*setting*

`fast_complex=ON` enables fast calculations for values declared as type complex. When set to ON, complex absolute value (norm) and complex division calculations use fast algorithms that can cause overflow for an operand (divisor in the case of division) that has an absolute value larger than the square root of the largest representable floating-point number (or underflow for a value which is smaller than the square root of the smallest representable floating point number).

Enter `ON` or `OFF` for *setting*. The default is `fast_complex=OFF`. `fast_complex=ON` is enabled if `-OPT:roundoff=3` is in effect.

2.47.7 -OPT:fast_exp=*setting*

`fast_exp=ON` optimizes exponentiation by replacing the run-time call for exponentiation by multiplication and/or square root operations for certain compile-time constant exponents (integers and halves). This can produce differently rounded results than the run-time routine. `fast_exp=ON` is in effect unless `-OPT:roundoff=1` is in effect.

Enter `ON` or `OFF` for *setting*. The default is `fast_exp=ON`.

2.47.8 -OPT:fast_nint=*setting*

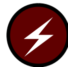
fast_nint=ON uses hardware features to implement NINT and ANINT (both single- and double-precision versions). Enter ON or OFF for *setting*. The default is fast_nint=OFF, but fast_nint=ON is enabled by default if -OPT:roundoff=3 is in effect. fast_nint=ON is also enabled when fast_trunc=ON is in effect.

When fast_nint=ON is in effect, rounding is performed according to the IEEE standard rather than the Fortran 90 standard. For example, the Fortran 90 standard requires that NINT(1.5)=2 and NINT(2.5)=3. The IEEE standard, however, rounds both of these to 2.

2.47.9 -OPT:fast_sqrt=*setting*

fast_sqrt=ON calculates square roots using the identity sqrt(x)=x*rsqrt(x), where rsqrt is the reciprocal square root operation. Enter ON or OFF for *setting*. The default is OFF.

This option is ignored unless the -mips4 option is in effect.



Warning: This option results in sqrt(0.0) producing a NaN result. Use it only when zero sqrt operands are not valid.

2.47.10 -OPT:fast_trunc=*setting*

fast_trunc=ON inlines the NINT, ANINT, AINT, and AMOD Fortran intrinsics, both single- and double-precision versions. Enter ON or OFF for *setting*. The default is fast_trunc=OFF. fast_trunc=ON is enabled automatically if -OPT:roundoff=1 (or greater) is in effect.

Although fully compliant with the Fortran 90 standard, fast_trunc=ON reduces the valid argument range somewhat.

If fast_trunc=ON is in effect, fast_nint=ON is also enabled.

2.47.11 -OPT:fold_reassociate=*setting*

fold_reassociate=ON allows optimizations involving reassociation of floating-point quantities. Enter ON or OFF for *setting*. The default is fold_reassociate=OFF. fold_reassociate=ON is enabled automatically when -O3 is in effect or when -OPT:roundoff=2 or greater is in effect.

2.47.12 -OPT:fold_unsafe_relops=*setting*

fold_unsafe_relops=ON folds relational operators in the presence of possible integer overflow. Enter ON or OFF for *setting*. The default is fold_unsafe_relops=ON.

2.47.13 -OPT:fold_unsigned_relops=*setting*

fold_unsigned_relops=ON folds unsigned relational operators in the presence of possible integer overflow. Enter ON or OFF for *setting*. The default is fold_unsigned_relops=OFF.

2.47.14 -OPT:got_call_conversion=*setting*

got_call_conversion=ON loads function addresses to be moved out of loops. The load is set up with the proper relocation so that the address is resolved at program start-up time. Enter ON or OFF for *setting*. got_call_conversion=OFF is the default when -O2 or lower is in effect. got_call_conversion=ON when -O3 is in effect.

Note: This option should be disabled when compiling shared objects that contain function addresses that may be preempted by rld(1). For more information, see dso(5).

2.47.15 -OPT:IEEE_arithmetic=*n*

These options specify the level of conformance to ANSI/IEEE 754-1985, the IEEE Standard for Binary Floating-point Arithmetic, which describes a standard for NaN and inf operands, arithmetic roundoff, and overflow. Enter 1, 2, or 3 for *n*. The default is IEEE_arithmetic=1.

IEEE_arithmetic=1 inhibits optimizations that produce less accurate results than required by ANSI/IEEE 754-1985, the IEEE Standard for Binary Floating-point Arithmetic.

IEEE_arithmetic=2 performs operations that can produce less accurate inexact results (but accurate exact results) on the target hardware. Examples are the recip and rsqrt operators for a MIPS IV target.

IEEE_arithmetic=3 performs arbitrary, mathematically valid transformations, even if they can produce inaccurate results for operations specified in ANSI/IEEE 754-1985, the IEEE Standard for Binary Floating-point Arithmetic. These transformations can cause overflow or underflow for a valid

operand range. An example is the conversion of `x/y to x*recip(y)` for MIPS IV targets. See also `roundoff=`***n*** in this list.

2.47.16 **-OPT:IEEE_comparisons=*setting***

Forces all comparisons to yield results that conform to ANSI/IEEE 754-1985, the IEEE Standard for Binary Floating-point Arithmetic, which describes a standard for NaN and inf operands. Enter `ON` or `OFF` for *setting*. The default is `IEEE_comparisons=OFF`.

`IEEE_comparisons=OFF` produces non-IEEE results for comparisons. For example, `x=x` is treated as TRUE without executing a test.

2.47.17 **-OPT:inline_intrinsics=*setting***

`inline_intrinsics=OFF` turns all Fortran intrinsics that have a library function into a call to that function. Enter `ON` or `OFF` for *setting*. The default is `inline_intrinsics=ON`.

2.47.18 **-OPT:liberal_ivdep=*setting***

Instruct the compiler to ignore all vector dependencies when encountering `!DIR$ IVDEP` statements. Enter `ON` or `OFF` for *setting*. The default is `liberal_ivdep=OFF`.

2.47.19 **-OPT:Olimit=*n***

Specifies that any routine bigger than *n* should not be optimized. If `-O2` or greater is in effect and a routine is so big that the compile speed may be slow, the compiler generates a message indicating the `Olimit` value that is needed to optimize. You can recompile with that value of *n* or you can recompile with `-OPT:Olimit=0` and avoid having any `Olimit` cutoff.

2.47.20 **-OPT:pad_common=*setting***

`pad_common=ON` reorganizes common blocks to improve the cache behavior of accesses to members of the common block. This may involve adding padding between members and/or breaking a common block into a collection of common blocks. Enter `ON` or `OFF` for *setting*. The default is `pad_common=OFF`.

This option should not be used unless the common block definitions (including EQUIVALENCE) are consistent among all sources comprising a program. In addition, pad_common=ON should not be specified if common blocks are initialized with DATA statements. If specified, pad_common=ON must be used for all source files in the program.

pad_common=ON is supported for Fortran only. It should not be used if a common block is referenced from C code.

2.47.21 -OPT:recip=*setting*

recip=ON specifies that faster, but potentially less accurate, reciprocal operations should be performed. Enter ON or OFF for *setting*. The default is recip=OFF. If -O3 or -OPT:IEEE_arithmetic=2 or above are in effect, recip=ON is enabled. recip=ON is effective only if -r8000 is in effect.

2.47.22 -OPT:reorg_common=*setting*

reorg_common=ON reorganizes common blocks to improve the cache behavior of accesses to members of the common block. The reorganization is performed only if the compiler detects that it is safe to do so. Enter ON or OFF for *setting*.

This option produces consistent results for programs that conform to the Fortran 90 standard. The optimizations performed are safe even if common blocks are declared differently in different subroutines or if elements in the common block are equivalenced. The optimizations performed with reorg_common=ON can lead to unexpected results if references to arrays in the common block are made outside the declared array bounds.

reorg_common=ON is enabled when -O3 is in effect and when all files that reference the common block are compiled at -O3. reorg_common=OFF is set when the file that contains the common block is compiled at -O2 (or below).

2.47.23 -OPT:roundoff=*n*

Specifies the level of acceptable departure from source language floating-point, round-off, and overflow semantics. Enter 0, 1, 2, or 3 for *n*.

roundoff=0 is the default when optimization levels -O0, -O1, and -O2 are in effect. This inhibits optimizations that might affect the floating-point behavior.

`roundoff=1` allows simple transformations that might cause limited round-off or overflow differences. Compounding such transformations could have more extensive effects.

`roundoff=2` is the default level when `-O3` is in effect. This level allows more extensive transformations, such as the reordering of reduction loops.

`roundoff=3` enables any mathematically valid transformation.

To obtain best performance in conjunction with software pipelining, specify `roundoff=2` or `roundoff=3`. This is because reassociation is required for many transformations to break recurrences in loops. Also see the descriptions for `-OPT:IEEE_arithmetic`, `-OPT:fast_complex`, `-OPT:fast_trunc`, and `-OPT:fast_nint`.

2.47.24 **`-OPT:rsqrt=`*setting***

`rsqrt=ON` specifies that faster, but potentially less accurate, square root operations should be performed. Enter `ON` or `OFF` for *setting*. The default is `rsqrt=OFF`.

If `-OPT:IEEE_arithmetic=2` or above or `-O3` are in effect, `rsqrt=ON` is enabled.

2.47.25 **`-OPT:space=`*setting***

`space=ON` specifies that code space is to be given priority in tradeoffs with execution time in optimization choices. For instance, this forces all exits from a function to go through a single exit block. Enter `ON` or `OFF` for *setting*. The default is `space=OFF`.

2.47.26 **`-OPT:swp=`*setting***

`swp=ON` enables software pipelining. Enter `ON` or `OFF` for *setting*. `swp=ON` is enabled when `-O3` is in effect. The default is `swp=OFF`.

2.47.27 **`-OPT:unroll_analysis=`*setting***

`unroll_analysis=ON` analyzes resource usage and recurrences in bodies of innermost loops that do not qualify for being fully unrolled. Such loops are unrolled only to the extent for which there is a potential benefit in doing so. A loop could be unrolled, for example, to decrease the shortest possible schedule

length per iteration. Enter ON or OFF for *setting*. The default is unroll_analysis=ON.

unroll_analysis=OFF can have the negative effect of unrolling loops less than the upper limit dictated by the OPT:unroll_times_max and OPT:unroll_size specifications.

2.47.28 -OPT:unroll_size=*n*

Specifies the maximum size (in instructions) of an unrolled loop. Enter an integer for *n*. The default is unroll_size=320.

This option indirectly determines which loops can be fully unrolled. See also -OPT:unroll_times_max in this list.

2.47.29 -OPT:unroll_times_max=*n*

Specifies the maximum number of times a loop will be unrolled if it is not going to be fully unrolled. Enter an integer for *n*. The default is unroll_times_max=2 when -mips4 is in effect. The default is unroll_times_max=4 when -mips3 is in effect. See also -OPT:unroll_size in this list.

2.47.30 -OPT:wrap_around_unsafe_opt=*setting*

wrap_around_unsafe_opt=OFF disables both the induction variable replacement and linear function test replacement optimizations. Enter ON or OFF for *setting*. By default, these optimizations are enabled at -O2 and -O3. These optimization are disabled by default at -O0.

This options's optimizations are unsafe because they can generate incorrect code when, for example, there are multiple induction variables in loops and their combined initial values overflow or wrap around in memory.

Using this option can degrade performance. It is provided as a diagnostic tool to identify the situation described previously.

2.48 -o*out_file*

Writes the executable file to *out_file* rather than to a.out. By default, the executable output file is written to a.out.

For example, the following command line loads object module `myprog.o` and produces an executable object named **myprog**:

```
% f90 -o myprog myprog.o
```

2.49 -O*level*

Specifies the basic optimization level, as follows:

Option	Action
`-O0`	No optimization. Default.
`-O1`	Local optimization.
`-O2, -O`	Extensive optimization. Optimizations performed at this level are almost always beneficial. The execution time is shortened, but compile time may be lengthened.
`-O3`	Aggressive optimization. Optimizations performed at this level may generate results that differ from those obtained when `-O2` is specified.
`-Ofast`[=*ipxx*]	Use optimizations selected to maximize performance for target platform *ipxx* processor type. To determine a platform *ipxx* designation, use the `hinv`(1) command. The optimizations performed may differ from release to release and among the supported platforms. The optimizations always enable the full instruction set of the target platform (for example, `-mips4` for an R10000). Although the optimizations are generally safe, they may affect floating-point accuracy due to operator reassociation. Typical optimizations selected include those performed at `-O3`. See the `-TARG:platform=`*ipxx* option for more

information on the *ipxx* argument. The default is an R10000 Power Challenge, IP25.

2.50 -P

Performs source preprocessing on *file*.f[90] or *file*.F[90] and puts the results in *file*.i. The *file*.i that is generated does not contain # lines.

2.51 -pfa, -pfalist

The -pfa option automatically converts sequential code into parallel code by inserting parallel directives where it it safe and beneficial to do so. Specifying this option also sets the -mp option to recognize parallel directives that you have already inserted into your code.

Note: This option is ignored unless you are licensed for the MIPSpro Automatic Parallelization Option. For more information on this product contact your sales representative.

When the -pfalist option is specified, the compiler produces *file*.l, which is a listing file. The listing file indicates the loops that were executed in parallel and explains why others were not executed in parallel.

For more information on parallel processing, see auto_p(5), or the *MIPSpro Automatic Parallelizer Programmer's Guide*, SGI publication 007–3572–001.

2.52 -r*real_spec*

Specifies the default kind specification for real values.

The -r option accepts 4 and 8 as arguments, as follows:

Option	Kind value
-r4	Uses REAL(KIND=4) and COMPLEX(KIND=4) for real and complex variables, respectively. Default.
-r8	Uses REAL(KIND=8) and COMPLEX(KIND=8) for real and complex variables, respectively. You can specify -r8 when porting programs from 64-bit machines to avoid convergence

problems and long execution times if the floating-point accuracy is inadequate.

2.53 -r*processor*

Specifies the code scheduler. The `-r` option accepts `4000`, `5000`, `8000`, and `10000` as arguments, as follows:

Option	Action
`-r4000`	Schedules code for the R4000 processor.
`-r5000`	Schedules code for the R5000 processor.
`-r8000`	Schedules code for the R8000 processor.
`-r10000`	Schedules code for the R10000 processor.

This option adds one of the following to the head of the library search path, where *processor* is as you specified:

- `-L/usr/lib32/mips3/`*processor*
- `-L/usr/lib32/mips4/`*processor*
- `-L/usr/lib64/mips3/`*processor*
- `-L/usr/lib64/mips4/`*processor*

The actual library search path that is added depends on the ABI that is specified or implied. See the `-64` and `-n32` options for information on specifying an ABI.

2.54 -S

Generates an assembly file, *file*`.s`, rather than an object file (*file*`.o`).

2.55 -static

Statically allocates all local variables. Statically allocated local variables are initialized to zero and exist for the life of the program. This option can be useful when porting programs from older systems in which all variables are statically allocated.

When compiling with the `-static` option, global data is allocated as part of the compiled object (*file*.o) file. The total size of any *file*.o file cannot exceed 2 GB, but the total size of a program loaded from multiple .o files can exceed 2 GB.

An individual common block may not exceed 2 GB, but you can declare multiple common blocks each having that size.

For more information on compiling with large files, see Section 2.1, page 6.

2.56 -TARG:...

Cross compiling is compiling a program on one system and executing it on another. To cross compile, you can either use the `-TARG:` command line options to control the target architecture and machine for which code is generated or you can set the `COMPILER_DEFAULTS_PATH` environment variable to specify the file that contains the default processor information needed to generate executable code for the target system.

The following subsections describe cross compiling using both the `-TARG:` options and the `COMPILER_DEFAULTS_PATH` environment variable.

2.56.1 -TARG:fp_precise=*setting*

Forces the target processor into precise floating-point mode at execution time. Using this option to compile any component source files of a program invokes this feature in the resulting program. Enter `ON` or `OFF` for *setting*. The default is `OFF`.

This option is only meaningful for R8000 target processors, and can cause significant performance degradation for programs with heavy floating-point usage. For more information on floating-point mode, see `fpmode`(1).

2.56.2 -TARG:madd=*setting*

Enables or prevents transformations from using multiply and add instructions. Enter `ON` or `OFF` for *setting*. The default is `ON`. This option is ignored unless `-mips4` is in effect.

These instructions perform a multiply/add with a single round off. They are more accurate than the usual discrete operations, and may cause results not to match baselines from other targets. Use this option to determine whether observed differences are due to multiply/add instructions.

2.56.3 `-TARG:platform=`*ipxx*

Specifies the target platform for compilation, choosing various internal parameters (such as cache sizes) appropriately. Supported values are as follows: `ip19`, `ip20`, `ip21`, `ip22_4k`, `ip22_5k`, `ip24`, `ip25`, `ip26`, `ip27`, `ip28`, `ip30`, `ip32_5k`, and `ip32_10k`. The appropriate selection for your platform can be determined by entering the following command:

```
hinv -c processor
```

The first line of output identifies the proper IP number. If a processor suffix (for example, `_4k`) is required, the next line identifies the processor (for example, R4000).

2.56.4 `-TARG:processor=`*processor*

Selects the processor for which to schedule code. The chosen processor must support the ISA specified (or implied by the ABI). Enter one of the following for *processor*: `r4000`, `r5000`, `r8000`, or `r10000`.

2.56.5 `-TARG:r4krev22=`*setting*

Generates code to work around bugs in the R4000 rev 2.2 chip. This currently means simulating 64-bit variable shifts in the software. Enter `ON` or `OFF` for *setting*. The default is `OFF`.

2.56.6 CPU targeting (cross compiling) using the `compiler.defaults` file

The MIPSpro 7 Fortran 90 compiler retrieves default information for the Application Binary Interface (ABI), instruction set architecture (ISA), and processor type from `/etc/compiler.defaults`. To compile for a different system, set the `COMPILER_DEFAULTS_PATH` environment variable to a path or to a colon-separated list of paths designating where the compiler is to look for the `compiler.defaults` file.

The target `compiler.defaults` file must contain a `-DEFAULT:` *option* specifier that specifies the default ABI, ISA, and processor in the following format:

```
-DEFAULT:[abi=n32|64] [:isa=mipsn] [:proc=r4000|r5000|r8000|r10000] [:opt=0|1|2|3] [:arith=1|2|3]
```

Note that command line settings override any settings in the system-supplied `compiler.defaults` file or in the `compiler.defaults` file that you create.

2.57 -TENV:...

Specifies the target environment option group. These options control the target environment assumed and/or produced by the compiler.

The following sections describe the -TENV:... options.

2.57.1 -TENV:align_aggregate=*bytes*

Controls alignment of allocated aggregates (that is, arrays and derived types). The value specified for *bytes* specifies that any aggregate object at least that large is to be given at least that alignment. By default, or if *bytes* is not specified, aggregates are aligned to the integer register size, which, for example, is 8 bytes for 64-bit programs and 4 bytes for 32-bit programs.

If align_aggregate=0 is specified, the value specifies that the minimum alignment consistent with the ABI is to be used. Otherwise, the value specified must be 1, 2, 4, 8, or 16.

2.57.2 -TENV:check_div=*n*

Inserts checks for divide by zero operations and overflow conditions on integer divide operations. Enter 0, 1, 2, or 3 for *n*. The default is check_div=0.

check_div=0 inhibits checking. check_div=1 checks for division by zero. check_div=2 checks for overflow. check_div=3 checks for both division by zero and overflow.

2.57.3 -TENV:large_GOT=*setting*

Generates code to accommodate a larger Global Offset Table (GOT) than is standard. Enter ON or OFF for *setting*. The default is large_GOT=OFF.

The standard GOT is 64K bytes. For more information on controlling the GOT, see the -TENV:small_GOT option.

2.57.4 -TENV:small_GOT=*setting*

Assumes that the GOT for shared code is smaller than 64K bytes, that is, assume small offsets for references to it. Enter ON or OFF for *setting*. The default is small_GOT=ON.

For more information on controlling the GOT, see the `-TENV:large_GOT` option.

2.57.5 `-TENV:trapuv=`*setting*

Forces all uninitialized stack, automatic, and dynamically allocated variables to be initialized with `0xFFFA5A5A`. If this value is used as a floating-point variable, it is treated as a floating-point NaN and causes a floating-point trap. If it is used as a pointer, an address or segmentation violation may occur. Enter `ON` or `OFF` for *setting*. The default is `OFF`.

You can obtain the functionality of a `-TENV:trapuv=ON` specification by specifying `-trapuv`. For more information on the `-trapuv` option, see Section 2.58, page 46.

2.57.6 `-TENV:X=`*n*

Specifies the level of enabled exceptions that will be assumed for purposes of performing speculative code motion The default is `X=2` when `-O3` is in effect. The default is `X=1` when other `-O` optimization levels are in effect. Enter 0, 1, 2, 3, or 4 for *n*. The default is `X=1`.

Generally, an instruction is not speculated (moved above a branch by the optimizer) unless any exceptions it might cause are disabled by this option. `X=0` inhibits speculative code motion.

`X=1` specifies that safe speculative code motion be performed and disables all underflow and inexact exceptions according to ANSI/IEEE 754-1985, the IEEE Standard for Binary Floating-point Arithmetic.

`X=2` disables all exceptions described in ANSI/IEEE 754-1985, the IEEE Standard for Binary Floating-point Arithmetic, except divide by zero.

`X=3` disables all exceptions described in ANSI/IEEE 754-1985, the IEEE Standard for Binary Floating-point Arithmetic, including divide by zero.

`X=4` disables or ignores memory exceptions.

At levels higher than the `X=1` default level, various hardware exceptions, which are normally useful for debugging, or which are trapped and repaired by the hardware, may be disabled or ignored. This can hide obscure bugs. The program should not explicitly manipulate the IEEE floating-point trap-enable flags in the hardware if this option is used.

2.58 -trapuv

Initializes all uninitialized stack, automatic and dynamically allocated variables to 0xFFFA5A5A. When this value is used as a floating-point variable, it is treated as a floating-point NaN and it causes a floating-point trap. When it is used as a pointer, an address or segmentation violation is likely to occur.

You can obtain the functionality of a -trapuv specification by specifying -TENV:trapuv=ON.

2.59 -U*var*

Undefines a variable for the source preprocessor. See the -D*var* option for information on defining variables.

2.60 -version

Writes compiler release version information to stdout. No input file needs to be specified when this option is used.

2.61 -w[*arg*]

Specifies messages. This option can take one of the following forms:

Option	Action
-w	Suppresses warning messages.
-w2	Shows warning messages. Default.

2.62 -woff*num*

Specifies message numbers to suppress. Examples:

- Specifying -woff2026 suppresses message number 2026.
- Specifying -woff2026-2352 suppresses messages 2026 through 2352.
- Specifying -woff2026-2352,2400-2500 suppresses messages 2026 through 2353 and messages 2400-2500.

In the message level indicator, the message numbers appear after the dash. This option applies to warning messages only.

2.63 `-x`*dirlist*

Disables specified directives or specified classes of directives. If specifying a multiword directive, either enclose the directive name in quotation marks or remove the spaces between the words in the directive's name.

For *dirlist*, enter one of the following:

dirlist	Directives disabled
`mipspro`	All directives.
directive	One or more directives. If specifying more than one, separate them with commas, as follows: `-x DOACROSS,"ASSERT NOARGUMENTALIASING"`.

2.64 `-xgot`

Uses a larger, nondefault Global Symbol Table (GOT). Specify this option if you receive a GOT overflow message. When this option if specified, the resulting executable is somewhat larger and slower. Specifying `-xgot` has the same effect as specifying `-TENV:large_GOT`.

A better solution for GOT overflow problems is to compile the object files without the `-xgot` option and to load using `ld -multigot` option. For more information, see the `ld`(1) man page.

For more information about the GOT, see the `-avoid_gp_overflow` option, the `gp_overflow`(5) man page, and the *What should I do about a GOT overflow?* question in the FAQ section of the `dso`(5) man page.

2.65 `--`

Signifies the end of options. After this symbol, you can specify the files to be processed.

2.66 *file.suffix*[90][*file.suffix*[90]...]

File or files to be processed, where suffix is either an uppercase `F` or a lowercase `f` for source files. Files ending in `.i`, `.o`, and `.s` are also accepted. The Fortran source files are compiled, and an executable object file is produced.

The default name of the executable object file is `a.out`. For example, the following command line produces `a.out`:

```
% f90 myprog.f
```

By default, several files are created during processing. The MIPSpro 7 Fortran 90 compiler adds a suffix to the file portion of the file name and places the files it creates into your working directory. The following is a file summary:

File	Content
file.B	Intermediate file written by the front end of the compiler.
file.i	File generated by the source preprocessor.
file.L	Listing file
file.s	Assembly language file

Directives [3]

A *directive* is a line inserted into Fortran source code that specifies actions to be performed by the compiler. Directive lines are not Fortran 90 statements.

Many MIPSpro 7 Fortran 90 compiler features are implemented as either command line options or directives. The features implemented as command line options are set at compile time and applied to all files in the compilation. The features implemented through directives are set within your Fortran 90 source code, and they apply to portions of your source code.

This chapter introduces the MIPSpro 7 Fortran 90 directive set and describes the general directives. The following other chapters also describe directives:

- Chapter 4, page 65, Multiprocessing Directives.
- Chapter 5, page 105, Origin Series Directives. This chapter describes the directives that are available to you if you are running the MIPSpro 7 Fortran 90 compiler on an Origin 2000, Origin 200, or Cray Origin 2000 system.
- Chapter 6, page 131, CF90 Directives. This chapter describes the Cray Research CF90 compiler directives that the MIPSpro 7 Fortran 90 compiler supports.

The sections in this chapter are as follows:

- Using directives
- Loop nest optimization (LNO) directives
- Argument aliasing directives
- Symbol storage directives
- Inlining and IPA directives

3.1 Using directives

All directives are of the following form:

```
prefix directive
```

prefix Each directive begins with a prefix. The prefix needed for each directive is shown in the directive's description. The following directive prefixes are used by the MIPSpro 7 Fortran 90 compiler:

- `!*$*`, `C*$*`
- `!$PAR`, `C$PAR`
- `!$`, `C$`
- `!DIR$`, `CDIR$`
- `!MIC$`, `CMIC$`

The prefix used also depends on which Fortran 90 source form you are using, as follows:

- If you are using fixed source form, begin a directive line with the characters `C`*prefix* or `!`*prefix*. The `!` or `C` character should appear in column 1. Beginning the directive with a `!` or `C` character ensures that compilers other than the MIPSpro 7 Fortran 90 compiler will treat compiler directive lines as comment lines.
- If you are using free source form, begin a directive line with the characters `!`*prefix*, followed by a space, and then one or more directives. The `!`*prefix* need not start in column 1, but it must be the first text on a line.

Because both fixed source form and free source form accept directives that start with the exclamation point (`!`), that is the initial character used in all directive syntax descriptions in this manual.

directive This is the specific directive's syntax. The syntax usually consists of the directive name. Some directives accept arguments. A directive's arguments, if any, are shown in the description for the directive itself.

The following sections describe the general format for directives and explain how directives are continued across source code lines.

3.1.1 Directives and command line options

Some compiler features can be activated on the command line or through compiler directives. The difference is that a command line setting applies to all

files in the compilation, but a directive applies to only a program unit or to another specific part of a source file.

Generally, and by default, directives override command line options. There are exceptions to this rule, however. The exceptions, if any, are noted in the introductory text to each directive group.

3.1.2 Directive range

The range of a particular directive depends on the directive itself, as follows:

- If a directive appears within a program unit, it applies only to that program unit.
- If a directive appears outside a program unit, for example, at the top of a file, it applies to the entire file.

The descriptions for the individual directives indicate the range of the directive.

3.1.3 Directive continuation and other considerations

It is sometimes necessary to continue a directive across one or more source code lines. The continuation character used and its placement within the directive line depends on the type of directive you are using. The introductory text for each directive group indicates the continuation character that is appropriate for that group. For all directives in this chapter, the prefix for a directive line that is a continuation line is `!*$*&`.

Do not use source preprocessor (`#`) directives within multiline compiler directives.

3.2 LNO directives

The loop nest optimization (LNO) directives control loop nest optimizations. By default, directives override command line options. To reverse this, and have command line options override the LNO directives, specify `-LNO:ignore_pragmas`.

To continue a directive, the continuation line must begin with `!*$*&`.

The following directives control loop nest optimizations:

- `AGGRESSIVEINNERLOOPFISSION`

- BLOCKABLE
- BLOCKINGSIZE, NOBLOCKING
- FISSION, FISSIONABLE, NOFISSION
- FUSE, FUSABLE, NOFUSION
- INTERCHANGE, NOINTERCHANGE
- PREFETCH
- PREFETCH_MANUAL
- PREFETCH_REF
- PREFETCH_REF_DISABLE
- UNROLL

The following sections describe the LNO directives.

3.2.1 AGGRESSIVEINNERLOOPFISSION

The AGGRESSIVEINNERLOOPFISSION directive specifies that the following loop should be split into as many loops as possible. In a loop nest, this directive must precede an inner loop.

The format of this directive is as follows:

```
!*$* AGGRESSIVEINNERLOOPFISSION
```

3.2.2 BLOCKABLE

The BLOCKABLE directive specifies that it is legal to cache block the subsequent loops. For more information on controlling cache blocking, see Section 2.35.2.1, page 21, and Section 2.35.2.2, page 21.

The format of this directive is as follows:

```
!*$* BLOCKABLE (do_variable, do_variable [, do_variable]...)
```

do_variable Specify the *do_variable* names of two or more loops. The loops identified by the *do_variable* names must be adjacent and nested within each other, although they need not be perfectly nested.

This directive informs the compiler that these loops may legally be involved in a blocking situation with each other, even if the compiler would consider such a transformation illegal. The loops must also be interchangeable and unrollable. This directive does not instruct the compiler on which of these transformations to apply.

3.2.3 BLOCKINGSIZE, NOBLOCKING

The BLOCKINGSIZE and NOBLOCKING directives assert that the loop following the directive either is (or is not) involved in a cache blocking for the primary or secondary cache.

The formats of these directives are as follows:

```
!*$* BLOCKINGSIZE(n1[,n2])

!*$* NOBLOCKING
```

n1,*n2* An integer number that indicates the block size. If the loop is involved in a blocking, it will have a block size of *n1* for the primary cache and *n2* for the secondary cache. The compiler attempts to include this loop within such a block, but it cannot guarantee this.

If *n1* or *n2* are 0, the loop is not blocked, but the entire loop is inside the block.

Example:

```
      SUBROUTINE AMAT(X,Y,Z,N,M,MM)
      REAL(KIND=8) X(100,100), Y(100,100), Z(100,100)
      DO K = 1, N
!*$* BLOCKING SIZE (20)
         DO J = 1, M
!*$* BLOCKING SIZE (20)
            DO I = 1, MM
               Z(I,K) = Z(I,K) + X(I,J)*Y(J,K)
            END DO
         END DO
```

```
END DO
END
```

For the preceding code, the compiler makes 20 X 20 blocks when blocking, but it could block the loop nest such that loop K is not included in the tile. If it did not, add a BLOCKINGSIZE(0) directive just before loop K to specify that the compiler should generate a loop such as the following:

```
SUBROUTINE AMAT(X,Y,Z,N,M,MM)
REAL(KIND=8) X(100,100), Y(100,100), Z(100,100)
DO JJ = 1, M, 20
   DO II = 1, MM, 20
      DO K = 1, N
         DO J = JJ, MIN(M, JJ+19)
            DO I = II, MIN(MM, II+19)
               Z(I,K) = Z(I,K) + X(I,J)*Y(J,K)
            END DO
         END DO
      END DO
   END DO
END DO
END
```

Note that an INTERCHANGE directive can be applied to the same loop nest as a BLOCKINGSIZE directive. The BLOCKINGSIZE directive applies to the loop it directly precedes; it moves with that loop when an interchange is applied.

The NOBLOCKING directive prevents the compiler from involving the subsequent loop in a cache blocking situation.

3.2.4 FISSION, FISSIONABLE, NOFISSION

The fission control directives specify whether the compiler should perform loop fission on the loops that immediately follow these directives.

The formats of these directives are as follows:

```
!*$* FISSION[(level)]

!*$* FISSIONABLE

!*$* NOFISSION
```

level Specify an integer number that indicates the number of loop levels that should undergo loop fission.

The `FISSION` directive specifies that loop fission should be attempted. The compiler performs a validity test on the subsequent loops unless you have also specified a `FISSIONABLE` directive. The `NOFISSION` directive specifies that the following loop should not undergo fission, but its inner loops, if any, may undergo fission.

These directives do not cause statements to be reordered.

3.2.5 FUSE, FUSABLE, NOFUSION

The fusion control directives specify whether the compiler should perform loop fusion on the loops that immediately follow these directives.

The formats of these directives are as follows:

```
!*$* FUSE[(n,[level])]

!*$* FUSABLE

!*$* NOFUSION
```

n Specify an integer number that indicates the number of subsequent loops that should undergo loop fusion. The default is 2.

level Specify an integer that indicates how deeply the loops should be fused.

The level of loop fusion is determined by the maximal perfectly nested loop levels of the fused loops, although partial fusion is allowed.

Loop iterations may be peeled as needed during loop fusion. The limit of this peeling is 5, or the number specified by the `-LNO:fusion_peeling_limit` command line option.

The `FUSE` directive specifies that loop fusion should be attempted. The compiler performs a validity test on the subsequent loops unless you have also specified a `FUSABLE` directive. When the `FUSABLE` directive is specified, the fusion is done for loops with identical iteration counts. The `NOFUSION` directive specifies that the following loop should not be fused with any other loop.

Example. Consider the following code:

```
DO I = 1,N
  DO J = 1,N
    S1
  END DO
END DO
DO I = 1,N
  DO J = 1,N
    S2
  END DO
END DO
```

Fusing the loops with a *level* of 1 results in the following loop nest:

```
DO I = 1,N
  DO J = 1,N
    S1
  END DO
  DO J = 1,N
    S2
  END DO
END DO
```

Fusing the loops with a *level* of 2 results in the following loop nest:

```
DO I = 1,N
  DO J = 1,N
    S1
    S2
  END DO
END DO
```

3.2.6 **INTERCHANGE, NOINTERCHANGE**

The loop interchange control directives specify whether or not the order of the following two or more loops should be interchanged. These directives apply to the loops that they immediately precede.

The formats of these directives are as follows:

```
!*$* INTERCHANGE (do_variable1, do_variable2 [, do_variable3]...)

!*$* NOINTERCHANGE
```

do_variable Specifies two or more *do_variable* names. The *do_variable* names can be specified in any order, and the compiler reorders the loops. The loops must be perfectly nested. If the loops are not perfectly nested, you may receive unexpected results.

The compiler reorders the loops such that the loop with *do_variable1* is outermost, then loop *do_variable2*, then loop *do_variable3*.

The `NOINTERCHANGE` directive inhibits loop interchange on the loop that immediately follows the directive.

3.2.7 PREFETCH

The `PREFETCH` directive controls the MIPS IV prefetch instruction. Using this directive can increase performance in program units that are likely to encounter cache misses during execution. This directive applies only to the program unit in which it appears.

When the directive is specified, the compiler estimates the memory references that will be cache misses, inserts prefetches for the misses, and schedules the prefetches ahead of their corresponding references. You can specify different levels of prefetching aggressiveness for the primary and secondary cache.

The format of this directive is as follows:

```
!*$* PREFETCH (primary_cache [,secondary_cache])
```

primary_cache, *secondary_cache* For each of these, specify 0, 1, or 2. The number specified indicates the level of prefetching requested for the primary and secondary cache levels, respectively.

A 0 disables all prefetching. 1 requests conservative prefetching. 2 requests aggressive prefetching. By default, *primary_cache* and *secondary_cache* are both set to 1 when the `-r10000` command line option is in effect, and they are set to 0 for all other processor settings.

This directive is honored only if the `-mips4` and `-r10000` command line options are in effect.

3.2.8 PREFETCH_MANUAL

The PREFETCH_MANUAL directive specifies whether the PREFETCH_REF and the PREFETCH_REF_DISABLE directives, which perform manual prefetches, should be respected or ignored within a subprogram. This directive applies only to the program unit in which it appears.

The format of this directive is as follows:

```
!*$* PREFETCH_MANUAL (n)
```

n — Specify either 0 or 1 for *n*. 0 indicates that the compiler should ignore all prefetch directive. 1 indicates that all prefetch directives should be honored. By default, all prefetch directives are honored.

This directive is honored only if the -mips4 and -r10000 command line options are in effect.

3.2.9 PREFETCH_REF

The PREFETCH_REF directive requests prefetching for a specific memory reference. This directive applies only to the loop nest that includes references to *array*, and the directive must immediately precede this loop nest.

When this directive is specified, all references to *array* in the subsequent loop nest are ignored by the automatic prefetcher (if enabled).

The format of this directive is as follows:

```
!*$* PREFETCH_REF=array [,stride=stride[,stride]] [,level=level[,level]]
   [,kind=rw] [,size=size]
```

array — For *array*, specify identification information for the array. For example: A(I,J).

stride — Specifies prefetching for every *stride* iterations of the loop. The default is 1.

level — Specifies the level in the memory hierarchy to prefetch. Specify 1 or 2. The default is 2. 1 specifies a prefetch from secondary cache to primary cache. 2 specifies a prefetch from memory to primary cache.

rw — Specify rd or wr. The default is wr.

size Specifies the size, in Kbytes, of *array*. Must be a constant.

If *size* is specified, the automatic prefetcher (if enabled) reduces the effective cache size by that amount in its calculations. The compiler tries to issue one prefetch per *stride* iterations, but this cannot be guaranteed.

This directive generates a single prefetch instruction to a specified memory reference. It searches for array references that match the supplied reference in the current loop nest and takes the following actions:

- If the reference is found, the reference is scheduled relative to the prefetch node, based on the miss latency for the specified level of the cache.
- If no such reference is found, the prefetch is generated at the start of the loop body.

This directive is honored only if the `-mips4` and `-r10000` command line options are in effect.

3.2.10 PREFETCH_REF_DISABLE

The `PREFETCH_REF_DISABLE` directive disables prefetching for all references to an array. This directive applies to all array references within the program unit.

If the automatic prefetcher is enabled, it ignores the specified array. The size is used for volume analysis.

The format of this directive is as follows:

```
!*$* PREFETCH_REF_DISABLE=array [, size=size]
```

array For *array*, specify identification information for the array. For example: `A(I,J)`.

size Specifies the size, in Kbytes, of *array*. Must be a constant.

This directive is honored only if the `-mips4` and `-r10000` command line options are in effect.

3.2.11 UNROLL

The `UNROLL` directive specifies loop unrolling. This directive applies to the loop that immediately follows the directive.

The format of this directive is as follows:

```
!*$* UNROLL (n)
```

n Specifies the number of copies of the loop body to be generated, as follows:

- When this directive precedes an inner loop, the compiler generates *n* — 1 copies of the loop body. This is standard loop unrolling.
- When this directive precedes an outer loop, the compiler performs an ***unroll and jam*** operation on the loop.

The value of *n* must be at least 1 in order for unrolling to occur. If *n* = 1, no unrolling is performed.

Even with this directive specified, unrolling is not performed if the compiler determines that unrolling would be unsafe. To specify that the compiler unroll the loop regardless of its analysis, you must also specify a `BLOCKABLE` directive. For information on the `BLOCKABLE` directive, see Section 3.2.2, page 52.

3.3 Argument aliasing directives

The `ASSERT ARGUMENTALIASING` and `ASSERT NOARGUMENTALIASING` directives allow the compiler to make assumptions about procedure dummy arguments when performing optimizations.

It is possible to call a procedure and specify the same variable or array element in two or more positions of the argument list. Within the procedure, two or more dummy argument names, which appear to refer to different memory locations, actually refer to the same location. This practice violates the Fortran standard. You can use the `ASSERT ARGUMENTALIASING` directive to allow the compiler to be more conservative.

By default, `ASSERT NOARGUMENTALIASING` is in effect.

The formats for these directives are as follows:

```
!*$* ASSERT ARGUMENTALIASING

!*$* ASSERT NOARGUMENTALIASING
```

If these directives appear outside of a program unit, they are applied to all program units in the source file. If they appear in a program unit, they are applied to that program unit only. If one of these directives is encountered, it remains in effect until reset by the opposing directive.

3.4 Symbol storage directives

The `ALIGN_SYMBOL` and `FILL_SYMBOL` directives control the way symbols are stored.

The `ALIGN_SYMBOL` directive aligns the start of *symbol* at a specified alignment boundary.

The `FILL_SYMBOL` directive pads *symbol* with additional storage so that the symbol is assured not to overlap with any other data item within the storage of the specified size. The additional padding required is divided between each end of the specified variable. For example, a `FILL_SYMBOL(X,L1CACHELINE)` directive guarantees that `X` does not suffer from false sharing for the primary cache line.

The formats for these directives are as follows:

```
!*$* ALIGN_SYMBOL (symbol [, storage])

!*$* FILL_SYMBOL (symbol [, storage])
```

symbol Specify the name of a symbol. *symbol* can be a common block name, common block variable, or automatic variable. *symbol* cannot be a component of a derived type or an array element.

storage Specifies the storage size. Specify one of the following values for *storage*:

storage	Action
`L1CACHELINE`	Specifies the machine-specific first-level cache line size, typically 32 bytes.
`L2CACHELINE`	Specifies the machine-specific secondary cache line size, typically 128 bytes.

`PAGE`	Specifies a machine-specific page. Typically 16 Kbytes.
power-of-two	An integer value that is a power of 2.

For common block variables, these directives are required at each declaration of the common block. Because the directives modify the allocated storage and its alignment for the named *symbol*, inconsistent directives can lead to undefined results.

The `ALIGN_SYMBOL` directive has no effect on local variables of fixed-size symbols, such as simple scalars or arrays of known size. The directive continues to be effective for stack-allocated arrays of dynamically determined size.

You cannot specify an `ALIGN_SYMBOL` directive and a `FILL_SYMBOL` directive for the same *symbol*.

Example:

```
! X IS A COMMON BLOCK VARIABLE
!
    INTEGER(KIND=4) X
!*$* ALIGN_SYMBOL (X, 32)

!   X WILL START AT A 32-BYTE BOUNDARY.
!   WARNING: THE LAYOUT OF THE COMMON BLOCK WILL BE AFFECTED

!*$* ALIGN_SYMBOL (X, 2)
!   ERROR: CANNOT REQUEST AN ALIGNMENT LOWER THAN THE NATURAL
!   ALIGNMENT OF THE SYMBOL.

    REAL(KIND=8) Y
!   Y IS A COMMON BLOCK OR LOCAL VARIABLE
!*$* FILL_SYMBOL (Y, L2CACHELINE)


!   ALLOCATE EXTRA STORAGE BOTH BEFORE AND AFTER Y SO THAT
!   Y IS WITHIN AN L2CACHELINE (128 BYTES) ALL BY ITSELF.
!   THIS CAN BE USEFUL TO AVOID FALSE-SHARING BETWEEN MULTIPLE
!   PROCESSORS FOR THE CACHELINE CONTAINING Y.
```

3.5 Inlining and IPA directives

The following are the inlining and interprocedural analysis (IPA) directives:

- `INLINE`, `NOINLINE`
- `IPA`, `NOIPA`

Note: Neither inlining nor IPA are enabled by default. By default, the directives in this section, if present in your source code, are ignored. To enable the directives and turn on inlining and IPA, specify an `-INLINING:` option or an `-IPA:` option on your `f90`(1) command line. For more information on the command line interaction with these features, see `f90`(1) or `ipa`(1).

Inlining is the process of replacing a procedure reference with a copy of the procedure's code. This eliminates procedure call overhead and exposes the relationships between the procedure code, the return value, and the surrounding code. The `INLINE` and `NOINLINE` directives allow you to specify procedures that should be inlined.

Interprocedural analysis (IPA) is a MIPSpro compiler feature that includes inlining, common block array padding, constant propagation, dead procedure elimination, dead variable elimination, and global name optimizations. For detailed information on the IPA feature, see `ipa`(5). The `IPA` and `NOIPA` directives allow you to control IPA.

The formats of these directives are as follows:

```
!*$* INLINE [(name [,name] ...)] location

!*$* NOINLINE [(name [,name] ...)] location
```

```
!*$* IPA [name [,name] ...] location

!*$* NOIPA [name [,name] ...] location
```

name For the inlining directives, each *name* specification represents one or more routines to be inlined. If no routines are named, all routines in the program are inlined.

For the IPA directives, each *name* specification represents one or more routines to undergo IPA. If no routines are named, all routines in the program undergo IPA.

location Specify one of the following for *location*:

location	Action
`HERE`	Specifies that routines named on the subsequent source code line should be inlined or should undergo IPA. This is the default *location*.
`ROUTINE`	Specifies that the named function should be inlined or should undergo IPA everywhere it appears within the current routine.
`GLOBAL`	Specifies that the named function should be inlined or should undergo IPA throughout the source file.

Example. Consider the following code fragment:

```
      DO I = 1,N
!*$* INLINE (BETA) HERE
         CALL BETA(I,1)
      ENDDO
      CALL BETA(N,2)
```

Using the specifier `ROUTINE` rather than `HERE` in this example would inline both calls to `BETA`. Note that `-INLINE:=ON` must be specified on the `f90`(1) command line when this code is compiled in order for the inlining directive to be honored.

Multiprocessing Directives [4]

The MIPSpro 7 Fortran 90 multiprocessing directives let you optimize your code by helping you to split your program into concurrently executing pieces. This chapter describes techniques for analyzing your code and preparing it for execution on multiple CPUs.

This chapter describes two sets of directives to use for multiprocessing. The first set consists of the loop-level multiprocessing directives. The second set consists of directives based on the work of the Parallel Computing Forum (PCF). The PCF directives allow you to specify multiprocessing based on the model of a parallel region. The following sections describe the multiprocessing directives and how to use them.

The `-mp` option must be specified on the `f90`(1) command line in order for the compiler to honor the directives in this chapter. For more information on multiprocessing, see `mp`(3F) and `sync`(3F).

4.1 Using directives

Certain multiprocessing features are available to you either through the command line or through directives. For command line options and directives that accept either `ON` or `OFF` as arguments, the compiler turns the feature `OFF` when conflicting settings are present. If a feature accepts a numeric setting as an argument, the compiler compares the command line setting and the directive setting and uses the minimum setting.

Some command line options act like global directives. Other command line options override directives. Many directives have corresponding command line options. If you specify conflicting settings in the command line and a directive, the compiler chooses the most restrictive setting.

The following sections contain general information that applies to both the loop-level and the PCF directives.

4.1.1 Directive range

Directives placed on the first line of an input file are called *global directives.* The compiler interprets them as if they appeared at the top of each program unit in the file.

Directives appearing anywhere else in the file apply only until the end of the current program unit. The compiler resets the value of the directive to the global value at the start of the next program unit.

4.1.2 Directive continuation

To continue the loop-level multiprocessing directives onto another line, use `!$&` as the first characters in the continued line(s). For example:

```
!$DOACROSS share(ALPHA, BETA, GAMMA, DELTA,
!$&  EPSILON, OMEGA), LASTLOCAL(I, J, K, L, M, N),
!$&  LOCAL(XXX1, XXX2, XXX3, XXX4, XXX5, XXX6, XXX7,
!$&  XXX8, XXX9)
```

To continue the PCF directives onto another line, begin the continued line with the characters `!$PAR&`.

4.2 Loop-level multiprocessing directives: DOACROSS, CHUNK, MP_SCHEDTYPE, and !$

It is possible for the compiler to execute different iterations of a DO loop on multiple processors. For example, suppose a DO loop consisting of 200 iterations will run on a machine with four processors using the simplest scheduling method. The first 50 iterations run on one processor, the next 50 on another, and so on.

A multiprocessing code adjusts itself at run time to the number of processors actually available to it on the machine. By default, the multiprocessing code does not use more than 8 processors. If you want to use more processors, set the MP_SET_NUMTHREADS environment variable to a different value. If the 200-iteration loop was moved to a machine with only two processors, it would be divided into two blocks of 100 iterations each, without any need to recompile or reload. In fact, multiprocessing code can be run on single-processor machines. The loop is divided into one block of 200 iterations. This allows code to be developed on a single-processor system and later run on a multiprocessor.

The processes that participate in the parallel execution of a task are arranged in a master/slave organization. The original process is the master. It creates zero or more slaves to assist. When a parallel DO loop is encountered, the master contacts the slaves for help. When the loop is complete, the slaves wait for the master, and the master resumes normal execution. The master process and each of the slave processes are called a *thread of execution* or simply a *thread*. By default, the number of threads is set to the number of processors on the

machine or is set to 8, whichever is smaller. You can override the default and explicitly control the number of threads of execution used by a parallel job.

For multiprocessing to work correctly, the iterations of the loop must not depend on each other; each iteration must stand alone and produce the same answer regardless of when any other iteration of the loop is executed. Not all DO loops have this property, and loops without it cannot be correctly executed in parallel. However, many of the loops encountered in practice fit this model. Further, many loops that cannot be run in parallel in their original form can be rewritten to run wholly or partially in parallel. For information about determining data dependencies in loops, see Section 4.3, page 77.

The loop-level multiprocessing directives are as follows:

- DOACROSS
- CHUNK
- MP_SCHEDTYPE

The following sections describe the loop-level multiprocessing directives.

4.2.1 DOACROSS directive

The basis for the loop-level multiprocessing directives is the DOACROSS directive. This directive indicates to the compiler that it should run iterations of the subsequent DO loop in parallel. This directive must appear directly before the loop that is to be operated on, and it remains in effect for that loop only.

The format of this directive is as follows:

```
!$DOACROSS [ clause [, clause ] ...]
```

clause This directive accepts one or more of the following *clause*s:

- AFFINITY
- BLOCKED
- CHUNK
- IF
- LASTLOCAL
- LOCAL

- MP_SCHEDTYPE
- NEST
- PRIVATE
- REDUCTION
- SHARED

The sections that follow describe the DOACROSS directive clauses.

Appendix B, page 199, contains information on debugging when DOACROSS directives are used.

Note: The Fortran compiler does not support direct nesting of DOACROSS loops.

For example, the following is illegal and generates a compilation error:

```
!$DOACROSS LOCAL(I)
        DO I = 1, N
!$DOACROSS LOCAL(J)
           DO J = 1, N
              A(I,J) = B(I,J)
           END DO
        END DO
```

However, to simplify separate compilation, a different form of nesting is allowed. A routine that uses !$DOACROSS can be called from within a multiprocessed region. This can be useful if a single routine is called from several different places: sometimes from within a multiprocessed region, sometimes not. Nesting does not increase the parallelism. When the first !$DOACROSS loop is encountered, that loop is run in parallel. While in the parallel loop, if a call is made to a routine that itself has a !$DOACROSS, the subsequent loop is executed serially.

4.2.1.1 AFFINITY clause

Affinity scheduling allows you to map parallel loop iterations onto underlying threads. This clause is used most often on Origin series systems.

For more information on using this DOACROSS clause, see Section 5.2.2.1, page 112.

4.2.1.2 BLOCKED clause

The BLOCKED clause has the following format:

```
BLOCKED (int_expr)
```

int_expr Specify an integer expression.

4.2.1.3 CHUNK clause

The CHUNK clause affects work scheduling among the participating tasks in a loop. It breaks the work up into pieces specified by *int_expr*. This clause is valid only when the MP_SCHEDTYPE=DYNAMIC or MP_SCHEDTYPE=INTERLEAVE clauses have also been specified.

This clause has the following format:

```
CHUNK = int_expr
```

int_expr Specify an integer expression that represents the size of the chunk (that is, the number of iterations per chunk).

The CHUNK directive also affects the division of work. For more information on the CHUNK directive, see Section 4.2.2, page 73.

If CHUNK is specified, and MP_SCHEDTYPE is not, MP_SCHEDTYPE defaults to DYNAMIC. For more information on how this clause interacts with the MP_SCHEDTYPE clause, see Section 4.2.1.6, page 71.

4.2.1.4 IF clause

The IF clause determines whether the loop is actually executed in parallel. This clause has the following format:

```
IF (logical_expr)
```

logical_expr Specify a logical expression. If *logical_expr* evaluates to TRUE, the loop is executed in parallel. If *logical_expr* evaluates to FALSE, the loop is executed serially.

The expression tests the number of times the loop will execute to verify whether or not there is enough work in the loop to justify the overhead of parallel execution. Currently, the break-even point is approximately 4000 CPU clocks of work, which normally translates to almost 1000 floating point operations.

4.2.1.5 LASTLOCAL, LOCAL, and SHARED clauses

The LASTLOCAL, LOCAL, and SHARED clauses specify lists of variables used within parallel loops. A variable can appear in only one of these lists. The effect of these clauses is as follows:

- The LASTLOCAL clause specifies variables that are local to each process. Unlike with the LOCAL clause, the compiler saves only the value of the logically last iteration of the loop when it exits. The name LASTLOCAL is preferred over LAST LOCAL.
- The LOCAL clause specifies variables that are local to each process. If a variable is declared as LOCAL, each iteration of the loop is given its own uninitialized copy of the variable. You can declare a variable as LOCAL if its value does not depend on any other iteration of the loop and if its value is used only within a single iteration. In effect, the LOCAL variable is just temporary; a new copy can be created in each loop iteration without changing the final answer. The name LOCAL is preferred over PRIVATE.
- The SHARED clause specifies variables that are shared across all processes. If a variable is declared as SHARED, all iterations of the loop use the same copy of the variable. You can declare a variable as SHARED if it is only read (not written) within the loop or if it is an array in which each iteration of the loop uses a different element of the array. The name SHARED is preferred over SHARE.

By default, the DO variable is LASTLOCAL and all other variables are SHARED.

These clauses have the following formats:

```
LASTLOCAL var [ , var ... ]

LOCAL var [ , var ... ]

SHARED var [ , var ... ]
```

var Specify the name of a variable. If any *var* is an array, it is listed without any subscripts.

Common blocks, allocatable arrays, and Fortran 90 pointers cannot appear as *var* arguments in a LOCAL list.

LOCAL is a little faster than LASTLOCAL, so if you do not need the final value, it is good practice to put the DO index variable into the LOCAL list, although this is not required.

4.2.1.6 MP_SCHEDTYPE clause

The MP_SCHEDTYPE clause affects the way the compiler schedules work among the participating tasks in a loop.

This clause has the following format:

```
MP_SCHEDTYPE = mode
```

mode Specify one of the following for *mode*:

- DYNAMIC. Specifying MP_SCHEDTYPE=DYNAMIC breaks the iterations into pieces the size of which is specified with the CHUNK clause. As each process finishes a piece, it enters a critical section to grab the next available piece. This gives good load balancing at the price of higher overhead. The CHUNK clause is valid with this *mode*.
- GSS. Specifying MP_SCHEDTYPE=GSS results in a variation of the guided self-scheduling algorithm. The piece size is varied depending on the number of iterations remaining. By parceling out relatively large pieces to start with and relatively small pieces toward the end, the system can achieve good load balancing while reducing the number of entries into the critical section. Specifying GUIDED for *mode* performs the same function as specifying GSS, but GSS is preferred.
- INTERLEAVE. Specifying MP_SCHEDTYPE=INTERLEAVE breaks the iterations into pieces of the size specified by the CHUNK clause, and execution of those pieces is interleaved among the processes. For example, if there are four processes and CHUNK=2, the first process executes iterations 1–2, 9–10, 17–18, …; the second process executes iterations 3–4, 11–12, 19–20,…; and so on. Although this is more complex than the

simple method, it is still a fixed schedule with only a single scheduling decision. The CHUNK clause is valid with this *mode*. Specifying INTERLEAVED for *mode* performs the same function as specifying INTERLEAVE, but INTERLEAVE is preferred.

- RUNTIME. Specifying MP_SCHEDTYPE=RUNTIME directs the scheduling routine to examine environment variables to select a *mode*. For the list of valid environment variables, see pe_environ(5).
- SIMPLE. Specifying MP_SCHEDTYPE=SIMPLE divides the iterations among processes by dividing them into contiguous pieces and assigning one piece to each process. Specifying STATIC for *mode* performs the same function as specifying SIMPLE, but SIMPLE is preferred.

The MP_SCHEDTYPE clause interacts with the CHUNK clause as follows:

- If both the MP_SCHEDTYPE and CHUNK clauses are omitted, SIMPLE scheduling is assumed.
- If MP_SCHEDTYPE=INTERLEAVE or MP_SCHEDTYPE=DYNAMIC and the CHUNK clause is omitted, CHUNK=1 is assumed.
- If MP_SCHEDTYPE is set to one of the other values, CHUNK is ignored.
- If the MP_SCHEDTYPE clause is omitted, but CHUNK is set, MP_SCHEDTYPE=DYNAMIC is assumed.

4.2.1.7 NEST clause

The NEST clause allows you to exploit nested concurrency. This DOACROSS clause is used most often on Origin series systems. For more information on this clause, see Section 5.2.2, page 112.

4.2.1.8 REDUCTION clause

The REDUCTION clause specifies variables involved in a reduction operation. In a *reduction operation*, the compiler keeps local copies of the variables and combines them when it exits the loop.

This clause has the following format:

```
REDUCTION var [ , var ]...
```

var Specify one or more variable names for *var*. Each *var* must be a scalar individual variable, not an array. A *var* can be an array element (for example `REDUCTION(A(I,J))`).

One element of an array can be used in a reduction operation while other elements of the array are used in other ways. To allow for this, if an element of an array appears in the `REDUCTION` list, the entire array can also appear in the `SHARED` list.

The four types of reductions supported are `sum`(+), `product`(*), `min`(), and `max`(). Note that `min` and `max` reductions must use the `MIN`(3I) and `MAX`(3I) intrinsic functions to be recognized correctly.

The compiler confirms that the reduction expression is legal by making some simple checks. The compiler does not, however, check all statements in the DO loop for illegal reductions. You must ensure that the reduction variable is used correctly in a reduction operation.

Example:

```
!$DOACROSS LOCAL(I), REDUCTION(A(1))
      DO I = 2,N
         A(1) = A(1) + A(I)
      END DO
```

4.2.2 CHUNK directive

The `CHUNK` directive breaks work up into pieces. Like the `MP_SCHEDTYPE` directive, the `CHUNK` directive acts as an implicit clause, in this case a `CHUNK` clause, for all `DOACROSS` directives in the scope. The `CHUNK` directive is in effect from the place it occurs in the source until another corresponding directive is encountered or the end of the procedure is reached.

The format of this directive is as follows:

```
CHUNK=int_expr
```

int_expr Specify an integer expression that represents the size of the chunk (that is, the number of iterations per chunk).

The `CHUNK` clause to the `DOACROSS` directive also divides work. For more information, see Section 4.2.1.3, page 69.

4.2.3 MP_SCHEDTYPE directive

The MP_SCHEDTYPE directive affects the way the compiler schedules work among the participating tasks in a loop. Like the CHUNK directive, the MP_SCHEDTYPE directive acts as an implicit clause, in this case an MP_SCHEDTYPE clause, for all DOACROSS directives in the scope. The MP_SCHEDTYPE directive is in effect from the place it occurs in the source until another corresponding directive is encountered or the end of the procedure is reached.

The MP_SCHEDTYPE directive specifies the scheduling type to be used for subsequent !$DOACROSS directives that are specified without an explicit scheduling type.

The format of this directive is as follows:

```
!$MP_SCHEDTYPE mode
```

mode This directive accepts a *mode* argument as described in Section 4.2.1.6, page 71.

The MP_SCHEDTYPE clause to the DOACROSS directive also divides work. For more information, see Section 4.2.1.6, page 71.

4.2.4 !$ directive

The !$ directive, which is really only a prefix, precedes code that should be honored only when multiprocessing is enabled. These directive lines are considered comment lines except when multiprocessing. A line beginning with !$ is treated as a conditionally compiled Fortran statement.

The format of this directive is as follows:

```
!$ statement
```

statement For *statement*, specify a standard Fortran statement. This feature can be used to insert debugging statements or other arbitrary code.

The following code demonstrates the use of the !$ directive:

```
!$    PRINT 10
!$ 10 FORMAT('BEGIN MULTIPROCESSED LOOP')
```

```
!$DOACROSS LOCAL(I), SHARED(A,B)
        DO I = 1, 100
           CALL COMPUTE(A, B, I)
        END DO
```

4.2.5 DOACROSS directive examples

Simple DOACROSS directive. Consider the following code fragment:

```
     DO 10 I = 1, 100
        A(I) = B(I)
10   CONTINUE
```

By inserting a directive, it can be multiprocessed:

```
!$DOACROSS LOCAL(I), SHARED(A, B)
     DO 10 I = 1, 100
        A(I) = B(I)
10   CONTINUE
```

Here, the defaults are sufficient provided that A and B are mentioned in a nonparallel region or in another SHARED list. The following code will then work:

```
!$DOACROSS
     DO 10 I = 1, 100
        A(I) = B(I)
10   CONTINUE
```

A DOACROSS directive with a LOCAL clause. Consider the following code fragment:

```
     DO 10 I = 1, N
        X = SQRT(A(I))
        B(I) = X*C(I) + X*D(I)
10   CONTINUE
```

The following shows this fragment rewritten for multiprocessing using explicit clauses:

```
!$DOACROSS LOCAL(I, X), SHARED(A, B, C, D, N)
     DO 10 I = 1, N
        X = SQRT(A(I))
        B(I) = X*C(I) + X*D(I)
10   CONTINUE
```

The following shows the fragment rewritten for multiprocessing using the default settings:

```
!$DOACROSS LOCAL(X)
     DO 10 I = 1, N
        X = SQRT(A(I))
        B(I) = X*C(I) + X*D(I)
10   CONTINUE
```

A DOACROSS directive with a LASTLOCAL clause. Consider the following code fragment:

```
     DO 10 I = M, K, N
        X = D(I)**2
        Y = X + X
        DO 20 J = I, MAX
           A(I,J) = A(I,J) + B(I,J) * C(I,J) * X + Y
20   CONTINUE
10   CONTINUE
     PRINT*, I, X
```

In this example, the final values of I and X are needed after the loop completes. A correct directive is shown in the following:

```
!$DOACROSS LOCAL(Y,J), LASTLOCAL(I,X),
!$& SHARED(M,K,N,ITOP,A,B,C,D)
     DO 10 I = M, K, N
         X = D(I)**2
         Y = X + X
         DO 20 J = I, ITOP
             A(I,J) = A(I,J) + B(I,J) * C(I,J) *X + Y
20       CONTINUE
10   CONTINUE
     PRINT*, I, X
```

You can also use the defaults:

```
!$DOACROSS LOCAL(Y,J), LASTLOCAL(X)
     DO 10 I = M, K, N
         X = D(I)**2
         Y = X + X
         DO 20 J = I, MAX
             A(I,J) = A(I,J) + B(I,J) * C(I,J) *X + Y
20       CONTINUE
10   CONTINUE
```

```
PRINT*, I, X
```

In the preceding code example, I is a loop index variable for the DOACROSS loop, so it is LASTLOCAL by default. Even though J is a loop index variable, it is not the loop index of the loop being multiprocessed and has no special status. If it is not declared, it is assigned the default value of SHARED, which produces an incorrect answer.

4.3 Analyzing data dependencies for multiprocessing

The essential condition required to parallelize a loop correctly is that each iteration of the loop must be independent of all other iterations. If a loop meets this condition, then the order in which the iterations of the loop execute is not important. They can be executed backward or at the same time, and the answer is still the same. This property is captured by the notion of ***data independence***.

For a loop to be data-independent, no iterations of the loop can write a value into a memory location that is read or written by any other iteration of that loop. It is all right if the same iteration reads and/or writes a memory location repeatedly as long as no others do; it is all right if many iterations read the same location, as long as none of them write to it.

In a Fortran program, memory locations are represented by variable names. So, to determine if a particular loop can be run in parallel, examine the way variables are used in the loop. Because data dependence occurs only when memory locations are modified, pay particular attention to variables that appear on the left-hand side of assignment statements. If a variable is not modified or if it is passed to a function or subroutine, there is no data dependence associated with it.

The Fortran compiler supports four kinds of variable usage within a parallel loop: SHARED, LOCAL, LASTLOCAL, and REDUCTION. If a variable is declared as SHARED, all iterations of the loop use the same copy. If a variable is declared as LOCAL, each iteration is given its own uninitialized copy. A variable is declared SHARED if it is only read (not written) within the loop or if it is an array where each iteration of the loop uses a different element of the array. A variable can be LOCAL if its value does not depend on any other iteration and if its value is used only within a single iteration. The LOCAL variable is essentially temporary; a new copy can be created in each loop iteration without changing the final answer. As a special case, if only the last value of a variable computed on the last iteration is used outside the loop (but would otherwise qualify as a LOCAL variable), the loop can be multiprocessed by declaring the variable to be LASTLOCAL.

It is often difficult to analyze loops for data dependence information. Each use of each variable must be examined to determine if it fulfills the criteria for LOCAL, LASTLOCAL, SHARED, or REDUCTION. If all of the uses conform, the loop can be parallelized. If not, the loop cannot be parallelized as written, but can possibly be rewritten into an equivalent parallel form.

An alternative to manually analyzing variable usage is to use the MIPSpro Automatic Parallelization Option. This optional software package is a Fortran preprocessor that analyzes loops for data dependence. If the MIPSpro Automatic Parallelization Option software determines that a loop is data-independent, it automatically inserts the required compiler directives. If it cannot determine if the loop is independent, it produces a listing file detailing where the problems lie.

4.3.1 Dependency analysis examples

Example 1: Simple independence. In this example, each iteration writes to a different location in A, and none of the variables appearing on the right-hand side are ever written to; they are only read from. This loop can be correctly run in parallel. All the variables are SHARED except for I, which is either LOCAL or LASTLOCAL, depending on whether the last value of I is used later in the code.

```
      DO 10 I = 1,N
10    A(I) = X + B(I)*C(I)
```

Example 2: Data dependence. The following code fragment contains A(I) on the left-hand side and A(I-1) on the right. This means that one iteration of the loop writes to a location in A and the next iteration reads from that same location. Because different iterations of the loop read and write the same memory location, this loop cannot be run in parallel.

```
       DO 20 I = 2,N
20     A(I) = B(I) - A(I-1)
```

Example 3: Stride not 1. This example is similar to the previous example. The difference is that the stride of the DO loop is now 2 rather than 1. A(I) now references every other element of A, and A(I-1) references exactly those elements of A that are not referenced by A(I). None of the data locations on the right-hand side is ever the same as any of the data locations written to on the left-hand side. The data are disjoint, so there is no dependence. The loop can be run in parallel. Arrays A and B can be declared SHARED, while variable I should be declared LOCAL or LASTLOCAL.

```
      DO 20 I = 2,N,2
20    A(I) = B(I) - A(I-1)
```

Example 4: Local variable. In the following loop, each iteration of the loop reads and writes the variable X. However, no loop iteration ever needs the value of X from any other iteration. X is used as a temporary variable; its value does not survive from one iteration to the next.

This loop can be parallelized by declaring X to be a LOCAL variable within the loop. Note that B(I) is both read and written by the loop. This is not a problem because each iteration has a different value for I, so each iteration uses a different B(I). The same B(I) is allowed to be read and written as long as it is done by the same iteration of the loop. The loop can be run in parallel. Arrays A and B can be declared SHARED, while variable I should be declared LOCAL or LASTLOCAL.

```
DO I = 1, N
     X = A(I)*A(I) + B(I)
     B(I) = X + B(I)*X
END DO
```

Example 5: Function call. The value of X in any iteration of the following loop is independent of the value of X in any other iteration, so X can be made a LOCAL variable. The loop can be run in parallel. Arrays A, B, C, and D can be declared SHARED, while variable I should be declared LOCAL or LASTLOCAL.

```
     DO 10 I = 1, N
        X = SQRT(A(I))
        B(I) = X*C(I) + X*D(I)
10 CONTINUE
```

This loop invokes an intrinsic function, SQRT. It is possible to use functions and/or subroutines (intrinsic or user defined) within a parallel loop. However, verify that the parallel invocations of the routine do not interfere with one another. In particular, SQRT returns a value that depends only on its input argument, does not modify global data, and does not use static storage (it has *no side effects*).

The Fortran 90 intrinsic functions have no side effects. The intrinsic functions can be used safely within a parallel loop. The intrinsic subroutines, however, can have side effects. Most Fortran library functions cannot be included in a parallel loop. In particular, rand is not safe for multiprocessing. For user-written routines, it is the your responsibility to ensure that the routines can be correctly multiprocessed.

Caution: Do not use the -static option on the f90(1) command line when compiling routines called within a parallel loop.

Example 6. Rewritable data dependence. Here, the value of INDX survives the loop iteration and is carried into the next iteration. This loop cannot be parallelized as it is written. Making INDX a LOCAL variable does not work; you need the value of INDX computed in the previous iteration. It is possible to rewrite this loop to make it parallel. See Section 4.3.2, page 81, for an example.

```
INDX = 0
DO I = 1, N
     INDX = INDX + I
     A(I) = B(I) + C(INDX)
END DO
```

Example 7: Exit branch. The following loop contains an exit branch; that is, under certain conditions the flow of control suddenly exits the loop. The compiler cannot parallelize loops containing exit branches.

```
       DO I = 1, N
          IF (A(I) .LT. EPSILON) GOTO 320
          A(I) = A(I) * B(I)
       END DO
   320 CONTINUE
```

Example 8: Complicated independence. Initially, it appears that the following loop cannot be run in parallel because it uses both W(I) and W(I-K). However, because the value of I varies between K+1 and 2*K, then I-K goes from 1 to K. This means that the W(I-K) term varies from W(1) to W(K), while the W(I) term varies from W(K+1) to W(2*K). Therefore, W(I-K) in any iteration of the loop is never the same memory location as W(I) in any other iterations. Because there is no data overlap, there are no data dependencies. This loop can be run in parallel. Elements W, B, and K can be declared SHARED, while variable I should be declared LOCAL or LASTLOCAL.

```
DO I = K+1, 2*K
     W(I) = W(I) + B(I,K) * W(I-K)
END DO
```

The preceding code illustrates a general rule: the more complex the expression used to index an array, the harder it is to analyze. If the arrays in a loop are indexed only by the loop index variable, the analysis is usually straightforward.

Example 9: Inconsequential data dependence. The data dependence in the following loop is present because it is possible that at some point that I will be the same as INDEX, so there will be a data location that is being read and written by different iterations of the loop. In this special case, you can simply ignore it. You know that when I and INDEX are equal, the value written into A(I) is exactly the same as the value that is already there. The fact that some iterations of the loop read the value before it is written and some after it is written is not important because they all get the same value. Therefore, this loop can be parallelized. Array A can be declared SHARED, but variable I should be declared LOCAL or LASTLOCAL.

```
INDEX = SELECT(N)
DO I = 1, N
     A(I) = A(INDEX)
END DO
```

Example 10: Local array. In the following code fragment, each iteration of the loop uses the same locations in the D array. However, closer inspection reveals that the entire D array is being used as a temporary. This can be multiprocessed by declaring D to be LOCAL. The Fortran compiler allows arrays (even multidimensional arrays) to be LOCAL variables with one restriction: the size of the array must be known at compile time. The dimension bounds must be constants; the LOCAL array cannot have been declared using a variable or the asterisk syntax.

```
DO I = 1, N
     D(1) = A(I,1) - A(J,1)
     D(2) = A(I,2) - A(J,2)
     D(3) = A(I,3) - A(J,3)
     TOTAL_DISTANCE(I,J) = SQRT(D(1)**2 + D(2)**2 + D(3)**2)
END DO
```

The preceding loop can be parallelized. Arrays TOTAL_DISTANCE and A can be declared SHARED, and array D and variable I can be declared LOCAL or LASTLOCAL.

4.3.2 Rewriting data dependencies

Many loops that have data dependencies can be rewritten so that some or all of the loop can be run in parallel. You must first locate the statement(s) in the loop that cannot be made parallel and try to find another way to express it that does not depend on any other iteration of the loop. If this fails, try to pull the

statements out of the loop and into a separate loop, allowing the remainder of the original loop to be run in parallel.

After you identify data dependencies, you can use various techniques to rewrite the code to break the dependence. Sometimes the dependencies in a loop cannot be broken, and you must either accept the serial execution rate or try to find a new parallel method of solving the problem. The following examples show how to deal with commonly occurring situations. These are by no means exhaustive but cover many situations that happen in practice.

Example 1: Loop-carried value. The following code segment is the same as the rewritable data dependence example in the previous section. INDX has its value carried from iteration to iteration. However, you can compute the appropriate value for INDX without making reference to any previous value.

```
INDX = 0
DO I = 1, N
   INDX = INDX + I
   A(I) = B(I) + C(INDX)
END DO
```

For example, consider the following code:

```
!$DOACROSS LOCAL (I, INDX)
    DO I = 1, N
       INDX = (I*(I+1))/2
       A(I) = B(I) + C(INDX)
    END DO
```

In this loop, the value of INDX is computed without using any values computed on any other iteration. INDX can correctly be made a LOCAL variable, and the loop can now be multiprocessed.

Example 2: Indirect indexing. Consider the following code:

```
     DO 100 I = 1, N
        IX = INDEXX(I)
        IY = INDEXY(I)
        XFORCE(I) = XFORCE(I) + NEWXFORCE(IX)
        YFORCE(I) = YFORCE(I) + NEWYFORCE(IY)
        IXX = IXOFFSET(IX)
        IYY = IYOFFSET(IY)
        TOTAL(IXX, IYY) = TOTAL(IXX, IYY) + EPSILON
100 CONTINUE
```

It is the final statement that causes problems. The indexes `IXX` and `IYY` are computed in a complex way and depend on the values from the `IXOFFSET` and `IYOFFSET` arrays. It is not known if `TOTAL(IXX,IYY)` in one iteration of the loop will always be different from `TOTAL(IXX,IYY)` in every other iteration of the loop.

You can pull the statement out into its own separate loop by expanding `IXX` and `IYY` into arrays to hold intermediate values, as follows:

```
!$DOACROSS LOCAL(IX, IY, I)
      DO I  = 1, N
         IX = INDEXX(I)
         IY = INDEXY(I)
         XFORCE(I) = XFORCE(I) + NEWXFORCE(IX)
         YFORCE(I) = YFORCE(I) + NEWYFORCE(IY)
         IXX(I) = IXOFFSET(IX)
         IYY(I) = IYOFFSET(IY)
      END DO
      DO 100 I = 1, N
         TOTAL(IXX(I),IYY(I)) = TOTAL(IXX(I), IYY(I)) + EPSILON
100 CONTINUE
```

Here, `IXX` and `IYY` have been turned into arrays to hold all the values computed by the first loop. The first loop (containing most of the work) can now be run in parallel. Only the second loop must still be run serially. This is true if `IXOFFSET` or `IYOFFSET` are permutation vectors.

If you were certain that the value for `IXX` was always different in every iteration of the loop, then the original loop could be run in parallel. It could also be run in parallel if `IYY` was always different. If `IXX` (or `IYY`) is always different in every iteration, then `TOTAL(IXX,IYY)` is never the same location in any iteration of the loop, and so there is no data conflict.

This sort of knowledge is program-specific and should always be used with great care. It may be true for a particular data set, but to run the original code in parallel as it stands, you need to be sure it will always be true for all possible input data sets.

Example 3: Recurrence. The following example shows a *recurrence*, which exists when a value computed in one iteration is immediately used by another iteration. There is no good way of running this loop in parallel. If this type of construct appears in a critical loop, try pulling the statement(s) out of the loop as in the previous example. Sometimes another loop encloses the recurrence; in that case, try to parallelize the outer loop.

```
DO I = 1,N
     X(I) = X(I-1) + Y(I)
END DO
```

Example 4: Sum reduction. The following example shows an operation known as a *reduction*. Reductions occur when an array of values is combined and reduced into a single value.

```
SUM  = 0.0
DO I = 1,N
     SUM = SUM + A(I)
END DO
```

This example is a sum reduction because the combining operation is addition. Here, the value of `SUM` is carried from one loop iteration to the next, so this loop cannot be multiprocessed. However, because this loop simply sums the elements of `A(I)`, you can rewrite the loop to accumulate multiple, independent subtotals and do much of the work in parallel, as follows:

```
      NUM_THREADS = MP_NUMTHREADS()
!
!  IPIECE_SIZE = N/NUM_THREADS ROUNDED UP
!
      IPIECE_SIZE = (N + (NUM_THREADS-1)) / NUM_THREADS
      DO K = 1, NUM_THREADS
         PARTIAL_SUM(K) = 0.0
!
!  THE FIRST THREAD DOES 1 THROUGH IPIECE_SIZE, THE
!  SECOND DOES IPIECE_SIZE + 1 THROUGH 2*IPIECE_SIZE,
!  ETC. IF N IS NOT EVENLY DIVISIBLE BY NUM_THREADS,
!  THE LAST PIECE NEEDS TO TAKE THIS INTO ACCOUNT,
!  HENCE THE "MIN" EXPRESSION.
!
         DO I = K*IPIECE_SIZE - IPIECE_SIZE + 1, MIN(K*IPIECE_SIZE,N)
            PARTIAL_SUM(K) = PARTIAL_SUM(K) + A(I)
         END DO
      END DO
!
!  NOW ADD UP THE PARTIAL SUMS
      SUM = 0.0
      DO I = 1, NUM_THREADS
         SUM = SUM + PARTIAL_SUM(I)
      END DO
```

The outer loop K can be run in parallel. In this method, the array pieces for the partial sums are contiguous, resulting in good cache utilization and performance.

Because this is an important and common transformation, automatic support is provided by the REDUCTION clause:

```
     SUM = 0.0
!$DOACROSS LOCAL (I), REDUCTION (SUM)
     DO 10 I = 1, N
        SUM = SUM + A(I)
10 CONTINUE
```

The previous code has essentially the same meaning as the much longer and more confusing code above. Adding an extra dimension to an array to permit parallel computation and then combining the partial results is an important technique for trying to break data dependencies. This technique is often useful.

Reduction transformations such as this do not produce the same results as the original code. Because computer arithmetic has limited precision, when you sum the values together in a different order, as was done here, the round-off errors accumulate slightly differently. It is probable that the final answer will be slightly different from the original loop. Both answers are equally correct. The difference is usually irrelevant, but sometimes it can be significant. If the difference is significant, neither answer is really trustworthy.

This example is a `sum` reduction because the operator is plus (+). The Fortran compiler supports the following types of reduction operations:

- sum: `p = p+a(i)`
- product: `p = p*a(i)`
- min: `m = MIN(m,a(i))`
- max: `m = MAX(m,a(i))`

For example,

```
!$DOACROSS LOCAL(I),REDUCTION(BG_SUM,BG_PROD,BG_MIN,BG_MAX)
          DO I = 1,N
             BG_SUM  = BG_SUM + A(I)
             BG_PROD = BG_PROD * A(I)
             BG_MIN  = MIN(BG_MIN, A(I))
             BG_MAX  = MAX(BG_MAX, A(I))
          END DO
```

The following is another example of a reduction transformation:

```
DO I = 1, N
   TOTAL = 0.0
   DO J = 1, M
      TOTAL = TOTAL + A(J)
   END DO
   B(I) = C(I) * TOTAL
END DO
```

Initially, it might look as if the inner loop should be parallelized with a REDUCTION clause. However, consider the outer I loop. Although TOTAL cannot be made a LOCAL variable in the inner loop, it fulfills the criteria for a LOCAL variable in the outer loop: the value of TOTAL in each iteration of the outer loop does not depend on the value of TOTAL in any other iteration of the outer loop. Thus, you do not have to rewrite the loop; you can parallelize this reduction on the outer I loop, making TOTAL and J local variables.

4.4 Work quantum

A certain amount of overhead is associated with multiprocessing a loop. If the work occurring in the loop is small, the loop can actually run slower by multiprocessing than by single processing. To avoid this, make the amount of work inside the multiprocessed region as large as possible, as is shown in the following examples.

Example 1: Loop interchange. Consider the following code:

```
DO K = 1, N
     DO I = 1, N
        DO J = 1, N
           A(I,J) = A(I,J) + B(I,K) * C(K,J)
        END DO
     END DO
END DO
```

For the preceding code fragment, you can parallelize the J loop or the I loop. You cannot parallelize the K loop because different iterations of the K loop read and write the same values of A(I,J). Try to parallelize the outermost DO loop if possible, because it encloses the most work. In this example, that is the I loop. For this example, use the technique called *loop interchange.* Although the parallelizable loops are not the outermost ones, you can reorder the loops to make one of them outermost.

Thus, loop interchange would produce the following code fragment:

```
!$DOACROSS LOCAL(I, J, K)
        DO I = 1, N
           DO K = 1, N
              DO J = 1, N
                 A(I,J) = A(I,J) + B(I,K) * C(K,J)
              END DO
           END DO
        END DO
```

Now the parallelizable loop encloses more work and shows better performance. In practice, relatively few loops can be reordered in this way. However, it does occasionally happen that several loops in a nest of loops are candidates for parallelization. In such a case, it is usually best to parallelize the outermost one.

Occasionally, the only loop available to be parallelized has a fairly small amount of work. It may be worthwhile to force certain loops to run without parallelism or to select between a parallel version and a serial version, on the basis of the length of the loop.

Example 2: Conditional parallelism. Consider the following code:

```
     J = (N/4) * 4
     DO I = J+1, N
        A(I) = A(I) + X*B(I)
     END DO
     DO I = 1, J, 4
        A(I) = A(I) + X*B(I)
        A(I+1) = A(I+1) + X*B(I+1)
        A(I+2) = A(I+2) + X*B(I+2)
        A(I+3) = A(I+3) + X*B(I+3)
     END DO
```

Loop unrolling of order four is used here to improve speed. For the first loop, the number of iterations is always fewer than four, so this loop does not do enough work to justify running it in parallel. The second loop is worthwhile to parallelize if `N` is big enough. To overcome the parallel loop overhead, `N` needs to be around 500.

An optimized version would use the `IF` clause on the `DOACROSS` directive:

```
     J = (N/4) * 4
     DO I = J+1, N
        A(I) = A(I) + X*B(I)
```

```
     END DO
!$DOACROSS IF (J .GE. 500), LOCAL(I)
        DO I = 1, J, 4
           A(I) = A(I) + X*B(I)
           A(I+1) = A(I+1) + X*B(I+1)
           A(I+2) = A(I+2) + X*B(I+2)
           A(I+3) = A(I+3) + X*B(I+3)
        END DO
     ENDIF
```

4.5 Cache effects and optimization

It is best to try to write loops that take the cache into account, with or without parallelism. The technique for attaining the best cache performance is quite simple: make the loop step through the array in the same way that the array is laid out in memory. For Fortran, this means stepping through the array without any gaps and with the leftmost subscript varying the fastest. This does not depend on multiprocessing, nor is it required in order for multiprocessing to work correctly. However, multiprocessing can affect how the cache is used.

4.5.1 Performing a matrix multiply

Consider the following code segment:

```
DO I = 1, N
     DO K = 1, N
        DO J = 1, N
           A(I,J) = A(I,J) + B(I,K) * C(K,J)
        END DO
     END DO
END DO
```

To get the best cache performance, the `I` loop should be innermost. At the same time, to get the best multiprocessing performance, the outermost loop should be parallelized.

For this example, you can interchange the `I` and `J` loops, and get the best of both optimizations:

```
!$DOACROSS LOCAL(I, J, K)
        DO J = 1, N
           DO K = 1, N
              DO I = 1, N
```

```
          A(I,J) = A(I,J) + B(I,K) * C(K,J)
       END DO
    END DO
 END DO
```

4.5.2 Optimization costs

Sometimes you must choose between the possible optimizations and their costs. Look at the following code segment:

```
DO J = 1, N
     DO I = 1, M
        A(I) = A(I) + B(J)*C(I,J)
     END DO
END DO
```

This loop can be parallelized on `I` but not on `J`. You could interchange the loops to put `I` on the outside, thus getting a bigger work quantum.

```
!$DOACROSS LOCAL(I,J)
     DO I = 1, M
        DO J = 1, N
           A(I) = A(I) + B(J)*C(I,J)
        END DO
     END DO
```

However, putting `J` on the inside means that you will step through the `C` array in the wrong direction; the leftmost subscript should be the one that varies the fastest. It is possible to parallelize the `I` loop where it stands:

```
     DO J = 1, N
!$DOACROSS LOCAL(I)
        DO I = 1, M
           A(I) = A(I) + B(J)*C(I,J)
        END DO
     END DO
```

However, `M` needs to be large for the work quantum to show any improvement. In this example, `A(I)` is used to do a sum reduction, and it is possible to use reduction techniques to rewrite this in a parallel form. However, that involves converting array `A` from a one-dimensional array to a two-dimensional array to hold the partial sums; this is analogous to the way the scalar summation variable was converted into an array of partial sums.

If A is large, however, the conversion can take too much memory. It can also take extra time to initialize the expanded array and increase the memory bandwidth requirements.

```
      NUM = MP_NUMTHREADS()
      IPIECE = (N + (NUM-1)) / NUM
!$DOACROSS LOCAL(K,J,I)
      DO K = 1, NUM
         DO J = K*IPIECE - IPIECE + 1, MIN(N, K*IPIECE)
            DO I = 1, M
               PARTIAL_A(I,K) = PARTIAL_A(I,K) + B(J)*C(I,J)
            END DO
         END DO
      END DO
!$DOACROSS LOCAL (I,K)
      DO I = 1, M
         DO K = 1, NUM
            A(I) = A(I) + PARTIAL_A(I,K)
         END DO
      END DO
```

You must analyze the various possible optimizations to find the combination that is right for the particular job.

4.5.3 Load balancing

When the Fortran compiler divides a loop into pieces, by default it uses the simple method of separating the iterations into contiguous blocks of equal size for each process. It can happen that some iterations take significantly longer to complete than other iterations. At the end of a parallel region, the program waits for all processes to complete their tasks. If the work is not divided evenly, time is wasted waiting for the slowest process to finish.

Consider the following code:

```
DO I = 1, N
     DO J = 1, I
        A(J, I) = A(J, I) + B(J)*C(I)
     END DO
END DO
```

The previous code segment can be parallelized on the I loop. Because the inner loop goes from 1 to I, the first block of iterations of the outer loop will end long before the last block of iterations of the outer loop.

In this example, this is easy to see and predictable, so you can change the program:

```
      NUM_THREADS = MP_NUMTHREADS()
!$DOACROSS LOCAL(I, J, K)
      DO K = 1, NUM_THREADS
         DO I = K, N, NUM_THREADS
            DO J = 1, I
               A(J, I) = A(J, I) + B(J)*C(I)
            END DO
         END DO
      END DO
```

In this rewritten version, instead of breaking up the I loop into contiguous blocks, break it into interleaved blocks. Thus, each execution thread receives some small values of I and some large values of I, giving a better balance of work between the threads. Interleaving usually, but not always, cures a load balancing problem.

You can use the MP_SCHEDTYPE clause to automatically perform this desirable transformation, as in this example:

```
!$DOACROSS LOCAL(I,J), MP_SCHEDTYPE=INTERLEAVE
         DO 20 I = 1, N
            DO 10 J = 1, I
               A (J,I) = A(J,I) + B(J)*C(J)
   10    CONTINUE
   20    CONTINUE
```

The previous code has the same meaning as the rewritten form above.

Interleaving can cause poor cache performance because the array is no longer stepped through at stride 1. You can improve performance somewhat by adding a CHUNK=*int_expr* clause. Usually 4 or 8 is a good value for *int_expr*. Each small chunk will have stride 1 to improve cache performance, while the chunks are interleaved to improve load balancing.

The way that iterations are assigned to processes is known as *scheduling*. Interleaving is one possible schedule. Both interleaving and the simple scheduling methods are examples of *fixed* schedules; the iterations are assigned to processes by a single decision made when the loop is entered. For more complex loops, it may be desirable to use DYNAMIC or GSS schedules.

Comparing the output from SpeedShop allows you to see how well the load is being balanced so you can compare the different methods of dividing the load. For more information on SpeedShop, see `ssrun`(1).

Even when the load is perfectly balanced, iterations may still take varying amounts of time to finish because of random factors. One process may take a page fault, another may be interrupted to let a different program run, and so on. Because of these unpredictable events, the time spent waiting for all processes to complete can be several hundred cycles, even with near perfect balance.

4.5.4 Local common blocks

The `-Xlocal` option to the `ld`(1) command allows named common blocks to be local to a process. Each process in the parallel job gets its own private copy of the common block. This can be helpful in converting certain types of Fortran programs into a parallel form.

The common block must be a named common block (blank common cannot be made local), and it must not be initialized by `DATA` statements.

To create a local common block, use the special loader option `-Xlocal` followed by a list of common block names. The external name of a common block known to the loader has a trailing underscore and is not surrounded by slashes. For example, the following command makes the common block `/foo/` a local common block in the resulting `a.out` file. You can specify multiple `-Xlocal` options if necessary.

```
% f90 -mp a.o -Xlocal,foo_
```

You can use the `!$COPYIN` directive to copy values from the master thread's version of the common block into the slave thread's version. This directive has the following format:

```
!$COPYIN item [, item] ...
```

item Specify one or more members of a local common block. Each *item* can be a variable, an array, an individual element of an array, or the entire common block.

Note: The `!$COPYIN` directive cannot be executed from inside a parallel region.

The following example propagates the values for `x` and `y`, all the values in the common block `foo`, and the `i`th element of array `a`:

```
!$COPYIN X,Y, /FOO/, A(I)
```

This directive is translated into executable code, so in this example `I` is evaluated at the time this statement is executed.

4.6 PCF directives

In addition to the simple loop-level parallelism offered by the `DOACROSS` directive, the compiler supports a set of directives that allows you to specify a more general model of parallelism. This model is based on the work done by the Parallel Computing Forum (PCF), which itself formed the basis for the proposed ANSI-X3H5 standard.

The main concept in this model is the *parallel region*, which can be any arbitrary section of code (not just a DO loop). Within the parallel region, there are special *work-sharing constructs* that can be used to divide the work among separate processes or threads. All master and slave threads synchronize at the bottom of a work-sharing construct. None of the threads continue past the end of a construct until they all have completed execution within that construct.

The parallel region can also contain a *critical section* construct, where exactly one process executes at a time. Within a critical section, only one thread executes at a time, and threads do not synchronize at the bottom of a critical section.

The master thread executes the user program until it reaches a parallel region. It then spawns one or more slave threads that begin executing code at the beginning of a parallel region. Each thread executes all the code in the region until a work sharing construct is encountered. Each thread then executes some portion of the work sharing construct, and then resumes executing the parallel region code. At the end of the parallel region, all the threads synchronize, and the master thread continues execution of the user program.

For information on interthread communication with library routines, see Appendix A, page 179.

The compiler recognizes the PCF directives when multiprocessing is enabled with either the `-mp` or the `-pfa` option to the `f90`(1) command. The PCF directives are as follows:

- `BARRIER`
- `CRITICALSECTION`, `ENDCRITICALSECTION`

- PARALLEL, ENDPARALLEL
- PARALLELDO
- PDO, ENDPDO
- PSECTION[S], SECTION, and ENDPSECTION[S]
- SINGLEPROCESS, ENDSINGLEPROCESS

The following sections describe the syntax of the PCF directives.

> **Note:** Generated code from the PCF directives is sometimes slower than the generated code from the special case parallelism offered by the DOACROSS directive. PCF directive code is slower because of the extra synchronization required. When a DOACROSS loop executes, there is a synchronization point at entry and another at exit. When a parallel region executes, there is a synchronization point at entry to the region, another at each entry to a work-sharing construct, another at each exit from a work-sharing construct, and one at exit from the region. Thus, several separate DOACROSS loops typically execute faster than a single parallel region with several PDO directives. Limit your use of the parallel region construct to those few cases that actually need it.

4.6.1 BARRIER directive

The BARRIER directive ensures that each process waits until all processes reach the barrier before proceeding.

This directive has the following format:

```
!$PAR BARRIER
```

4.6.2 CRITICALSECTION and ENDCRITICALSECTION directives

The CRITICALSECTION and ENDCRITICALSECTION directives ensure that the enclosed block of code is executed by only one process (thread) at a time. Another process attempting to gain entry to the critical section must wait until the previous process has exited. Threads do not synchronize at the bottom of a critical section.

The critical section construct can appear anywhere in a program, including inside and outside a parallel region and within a DOACROSS loop.

These directives have the following format:

```
!$PAR CRITICALSECTION [ (lock_variable) ]

!$PAR ENDCRITICALSECTION
```

lock_variable Specify an integer variable that is initialized to zero. The parentheses are required. If you do not specify *lock_variable*, the compiler automatically supplies a global lock. Multiple critical section constructs inside the same parallel region are considered to be independent of each other unless they use the same explicit *lock_variable*.

4.6.3 PARALLEL and ENDPARALLEL directives

The PARALLEL and ENDPARALLEL directives enclose a parallel region that includes work-sharing constructs and critical sections. It signifies the boundary within which slave threads execute. A user program can contain any number of parallel regions.

These directives have the following format:

```
!$PAR PARALLEL [clause [,clause]...]

!$PAR ENDPARALLEL
```

clause Specify one of the following clauses:

- IF (*logical_expression*)
- LOCAL *var*[, *var*] ...
- SHARED *var*[, *var*] ...

The IF, LOCAL, and SHARED clauses have the same meaning as for the DOACROSS directive. Also as with the DOACROSS directive, the keyword LOCAL is preferred to PRIVATE and the keyword SHARED is preferred to SHARE. For more information on these clauses and their syntax, see Section 4.2.1, page 67.

The preferred form of the directive has no commas between the clauses.

In the following code, all threads enter the parallel region and call routine FOO:

```
          SUBROUTINE EX1(INDEX)
          INTEGER I
!$PAR PARALLEL LOCAL(I)
          I = MP_MY_THREADNUM()
          CALL FOO(I)
!$PAR END PARALLEL
          END
```

4.6.4 PARALLELDO directive

The PARALLELDO directive indicates that the iterations of the subsequent DO loop should be executed by different processes. This directive produces the same effect as the DOACROSS directive, and it is conceptually the same as a parallel region containing exactly one PDO construct and no other code. Each thread inside the enclosing parallel region executes separate iterations of the loop within the parallel DO construct. This directive must not appear within a parallel region.

This directive has the following format:

```
!$PAR PARALLELDO [clause [,clause ] ...]
```

clause — For *clause*, enter one or more of the DOACROSS clauses described in Section 4.2.1, page 67.

4.6.5 PDO and ENDPDO directives

The PDO and ENDPDO directives surround a loop and indicate that the iterations of the enclosed loop should be executed by different processes. These directives must be enclosed within a parallel region delimited by PARALLEL and ENDPARALLEL directives.

Within a parallel region, each thread inside the region executes a separate iteration of a loop within a PDO construct.

These directives have the following format:

```
!$PAR PDO [clause [, clause]...]
```

```
[!$PAR ENDPDO [NOWAIT]]
```

clause Specify one of the following clauses:

- AFFINITY
- CHUNK=*int_expr*
- LASTLOCAL *var*
- LOCAL *var* [, *var*] ...
- MP_SCHEDTYPE=*mode*
- (ORDERED). Specifying the (ORDERED) clause is equivalent to specifying MP_SCHEDTYPE=DYNAMIC and CHUNK=1. The parentheses are required.

Each clause has the same meaning as for the DOACROSS directive. Also as with the DOACROSS directive, the keyword LASTLOCAL is preferred to LAST LOCAL and the keyword LOCAL is preferred to PRIVATE.

The (ORDERED) clause is not a supported DOACROSS clause.

For more information on the AFFINITY clause and its syntax, see Section 5.2.2.1, page 112. For more information on the other clauses and their syntax, see Section 4.2.1, page 67.

It is legal to declare a data item as LOCAL in a PDO directive even if it was declared as SHARED in the enclosing parallel region.

The ENDPDO directive is optional. If specified, this directive must appear immediately after the end of the DO loop. The optional NOWAIT clause specifies that each process should proceed directly to the code immediately following the directive. If you do not specify NOWAIT, the processes wait until all have reached the directive before proceeding.

The code in the following example is equivalent to a DOACROSS loop. In fact, the compiler recognizes this as a special case and generates the same (more efficient) code as for a DOACROSS directive.

```
          SUBROUTINE EX2(A,N)
          REAL A(N)
!$PAR PARALLEL LOCAL(I) SHARED(A)
!$PAR PDO
          DO I = 1, N
            A(I) = A(I) + 1.0
          END DO
```

```
!$PAR END PARALLEL
      END
```

4.6.6 PSECTION[S], SECTION, and ENDPSECTION[S] directives

The PSECTION[S] and ENDPSECTION[S] directives delimit a parallel section construct and distribute code blocks to processes. These directives have an effect that is similar to the Fortran 90 SELECT construct. Each block of code is parceled out in turn to a separate thread.

The SECTION directive indicates a starting line for an individual section within a parallel section.

These directives must be enclosed within a parallel region delimited by PARALLEL and ENDPARALLEL directives.

These directives have the following format:

```
!$PAR PSECTION[S] [LOCAL var[, var] ...]

[!$PAR SECTION]

!$PAR ENDPSECTION[S] [NOWAIT]
```

var Specify a variable name for *var*. The LOCAL keyword has the same meaning as it does on the DOACROSS directive. The LOCAL keyword is preferred to PRIVATE. For more information on LOCAL, see Section 4.2.1, page 67.

It is legal to declare a data item as LOCAL in a parallel sections construct even if it was declared as SHARED in the enclosing parallel region.

The optional NOWAIT clause specifies that each process should proceed directly to the code immediately following the directive. If you do not specify NOWAIT, the processes wait until all have reached the ENDPSECTION directive before proceeding.

Parallel sections can contain critical section constructs, but they cannot contain any of the following types of constructs:

- A DO loop that is preceded by a PDO directive
- A DO loop that is preceded by a PARALLELDO or a DOACROSS directive

- Code delimited by SINGLEPROCESS and ENDSINGLEPROCESS directives

Each code block is executed in parallel (depending on the number of processes available). The code blocks are assigned to threads one at a time, in the order specified. Each code block is executed by only one thread.

For example, consider the following code:

```
          SUBROUTINE EX3(A,N1,B,N2,C,N3)
          REAL A(N1), B(N2), C(N3)
!$PAR PARALLEL LOCAL(I) SHARED(A,B,C)
!$PAR PSECTIONS
!$PAR SECTION
          DO I = 1, N1
            A(I) = 0.0
          END DO
!$PAR SECTION
          DO I = 1, N2
            B(I) = 0.5
          END DO
!$PAR SECTION
          CALL NORMALIZE(C,N3)
          DO I = 1, N3
            C(I) = C(I) + 1.0
          END DO
!$PAR END PSECTION
!$PAR END PARALLEL
          END
```

The first thread to enter the parallel section construct executes the first block, the second thread executes the second block, and so on. This example has only three sections, so if more than three threads are in the parallel region, the fourth and higher threads wait at the !$PAR ENDPSECTION directive until all threads are finished. If the parallel region is being executed by only two threads, whichever thread finishes its block first continues and executes the remaining block.

This example uses DO loops, but a parallel section can be any arbitrary block of code. Parallel constructs have significant overhead. Make sure the amount of work performed is enough to outweigh the extra overhead.

The sections within a parallel section construct are assigned to threads one at a time, from the top down. There is no other implied ordering to the operations within the sections. In particular, a later section cannot depend on the results of an earlier section, unless some form of explicit synchronization is used. If there

is such explicit synchronization, you must be sure that the lexical ordering of the blocks is a legal order of execution.

4.6.7 SINGLEPROCESS and ENDSINGLEPROCESS directives

The SINGLEPROCESS and ENDSINGLEPROCESS directives enclose a block of code that should be executed by only one process. These directives must be enclosed within a parallel region delimited by PARALLEL and ENDPARALLEL directives.

These directives have the following format:

```
!$PAR SINGLEPROCESS  [LOCAL var [, var] ...]

!$PAR ENDSINGLEPROCESS [NOWAIT]
```

var Specify a variable name for *var*. The LOCAL keyword has the same meaning as it does on the DOACROSS directive. The LOCAL keyword is preferred to PRIVATE. For more information on LOCAL, see Section 4.2.1, page 67.

It is legal to declare a data item as LOCAL in a single process construct even if it was declared as SHARED in the enclosing parallel region.

The optional NOWAIT clause specifies that each process should proceed directly to the code immediately following the directive. If you do not specify NOWAIT, the processes waits until all have reached the ENDSINGLEPROCESS directive before proceeding.

This construct is semantically equivalent to a parallel section construct with only one section. The single process construct provides a more descriptive syntax.

The first thread to reach a single process section executes the code in that block. All other threads wait at the end of the block until the code has been executed.

Notice the use of the repetition of the IF test in the first parallel loop:

```
             IF (A(I,J) .GT. CUR_MAX) THEN
!$PAR CRITICAL SECTION
               IF (A(I,J) .GT. CUR_MAX) THEN
```

This practice is called *test&test&set.* It is a multiprocessing optimization. The following straightforward code segment is incorrect:

```
          DO I = 1, N
            IF (A(I,J) .GT. CUR_MAX) THEN
!$PAR CRITICAL SECTION
                  INDEX_X = I
                  INDEX_Y = J
                  CUR_MAX = A(I,J)
!$PAR END CRITICAL SECTION
            ENDIF
          ENDDO
```

Because many threads execute the loop in parallel, there is no guarantee that once inside the critical section, CUR_MAX still has the same value it did in the IF test outside the critical section (some other thread may have updated it). In particular, CUR_MAX may now have a value that is larger than A(I,J). Therefore, the critical section must be locked before testing the value of CUR_MAX. Changing the previous code into the following code works correctly, but suffers from a serious performance penalty: the critical section lock must be acquired and released (an expensive operation) for each element of the array:

```
            DO I = 1, N
!$PAR CRITICAL SECTION
                IF (A(I,J) .GT. CUR_MAX) THEN
                  INDEX_X = I
                  INDEX_Y = J
                  CUR_MAX = A(I,J)
                ENDIF
!$PAR END CRITICAL SECTION
         ENDDO
```

Because the values are rarely updated, this process involves a lot of wasted effort. It is almost certainly slower than just executing the loop serially.

Combining the two methods, as in the original example, produces code that is both fast and correct. If the IF test outside of the critical section fails, you can be certain that the values will not be updated, and can proceed. You can expect that the outside IF test will account for the majority of cases. If the outer IF test passes, then the values might be updated, but you cannot always be certain. To ensure correctness, you must perform the test again after acquiring the critical section lock.

You can prefix one of the two identical IF tests with !$ to reduce overhead in the non-multiprocessed case.

Lastly, note the difference between the single process and critical section constructs. If several processes arrive at a critical section construct, they execute the code one at a time. However, they will all execute the code. If several processes arrive at a single process construct, only one process executes the code. The other processes bypass the code and wait at the end of the construct for the chosen process to finish.

4.6.8 Restrictions on the PCF directives

The three work-sharing constructs, PDO, PSECTION, and SINGLEPROCESS, must be executed by all the threads executing in the parallel region or by none of the threads. The following is illegal:

```
        .
        .
        .
!$PAR PARALLEL
          IF (MP_MY_THREADNUM() .GT. 5) THEN
!$PAR SINGLE PROCESS
              MANY_PROCESSES = .TRUE.
!$PAR END SINGLE PROCESS
          ENDIF
           .
           .
           .
           .
```

The preceding code cannot run successfully when more than 6 processors are used. One or more processes will be stuck at the !$PAR ENDSINGLEPROCESS directive waiting for all the threads to arrive. Because some of the threads never took the appropriate branch, they will never encounter the construct. However, the following kind of simple looping is supported:

```
       ...
!$PAR PARALLEL LOCAL(I,J) SHARED(A)
          DO I= 1,N
!$PAR PDO
            DO J = 2,N
       ...
```

The distinction here is that all of the threads encounter the work-sharing construct. They all complete it, and they all loop around and encounter it again.

This restriction does not apply to the critical section construct, which operates on one thread at a time without regard to any other threads.

Parallel regions cannot be nested inside of other parallel regions, nor can work-sharing constructs be nested. However, as an aid to writing library code, you can call an external routine that contains a parallel region even from within a parallel region. In this case, only the first region is actually run in parallel. Therefore, you can create a parallelized routine without accounting for whether it will be called from within an already parallelized routine.

Parallel Programming on Origin series systems [5]

This chapter describes the support provided for parallel programs on Origin series systems.

For information on environment variables that can control run-time features, see the `pe_environ`(5) man page.

The multiprocessing features described in this chapter require support from the MP run-time library. IRIX operating system versions 6.3 and later are automatically shipped with this library. If you need to access these features on a machine running a different IRIX version, contact your sales representative.

5.1 Performance tuning on Origin series systems

Origin series systems provide cache-coherent, shared memory in the hardware. Memory is physically distributed across processors. Processors can read data only from the primary cache. If the required data is not present in the primary cache, a *cache miss* is said to have occured. Therefore, references to locations in the remote memory of another processor take substantially longer to complete than references to locations in local memory. Cache misses adversely affect program performance.

Figure 2 shows a simplified version of the Origin series memory hierarchy.

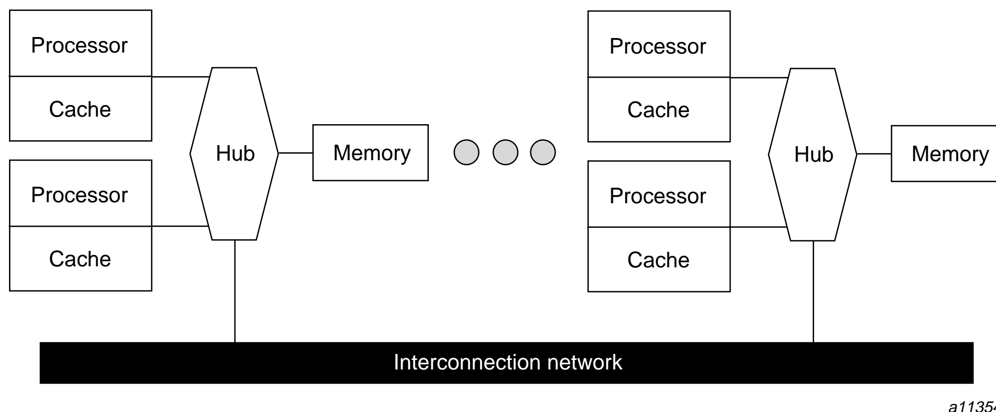


Figure 2. Origin series memory hierarchy

5.1.1 Improving program performance

To obtain good performance in parallel programs it is important to schedule computation and to distribute the data across the underlying processors and memory modules, ensuring that most cache misses are satisfied from local rather than from remote memory. The primary goal of programming support is to enable user control over data placement and user control over computation scheduling.

Cache behavior is the largest single factor affecting performance, and programs with infrequent cache misses usually have little need for explicit data placement. These programs write data to memory and reuse it as many times as possible before overwriting it. Depending on your system's processor, you may also be able to use `perfex`(1) to find information on your program's cache misses. For more information on `perfex`, see `perfex`(1).

In programs with many cache misses, if the misses correspond to true data communication between processors, data placement is unlikely to help. In these cases, it may be necessary to redesign your program to reduce interprocessor communication. When redesigning your program to reduce interprocessor communication, keep the following in mind:

- Make sure the data needed by a processor is at least local to the processor's memory.
- Make sure that each processor is working independently and not relying on the changing data of other processors.

- Minimize cache misses.

If the misses are to data that is referenced primarily by a single processor, then data placement may be able to convert remote references to local references, thereby reducing the latency of the miss. The possible methods for data placement are ***automatic page migration*** or ***explicit data distribution***, either regular or reshaped, described in detail in Section 5.3.1, page 119, and Section 5.3.2, page 120. The differences between these methods are shown in Figure 3. Some criteria for choosing between these methods are discussed in Section 5.1.2, page 109.

Automatic page migration requires no user intervention and is based on the run-time cache miss behavior of the program. It can, therefore, adjust to dynamic changes in the reference patterns. However, page migration is very conservative, and the compiler may be slow to react to changes in the reference patterns. It is also limited to performing page-level data allocation.

Regular data distribution (performing only page-level placement of the array) is also limited to page-level allocation, but is useful when the page migration heuristics are slow and the desired distribution is known to the programmer.

Finally, reshaped data distribution changes the layout of the array. This overcomes the page-level allocation constraints, but it is useful only if a data structure has the same (static) distribution for the duration of the program. Given these differences, it may be necessary to use each of these methods for different data structures in the same program.

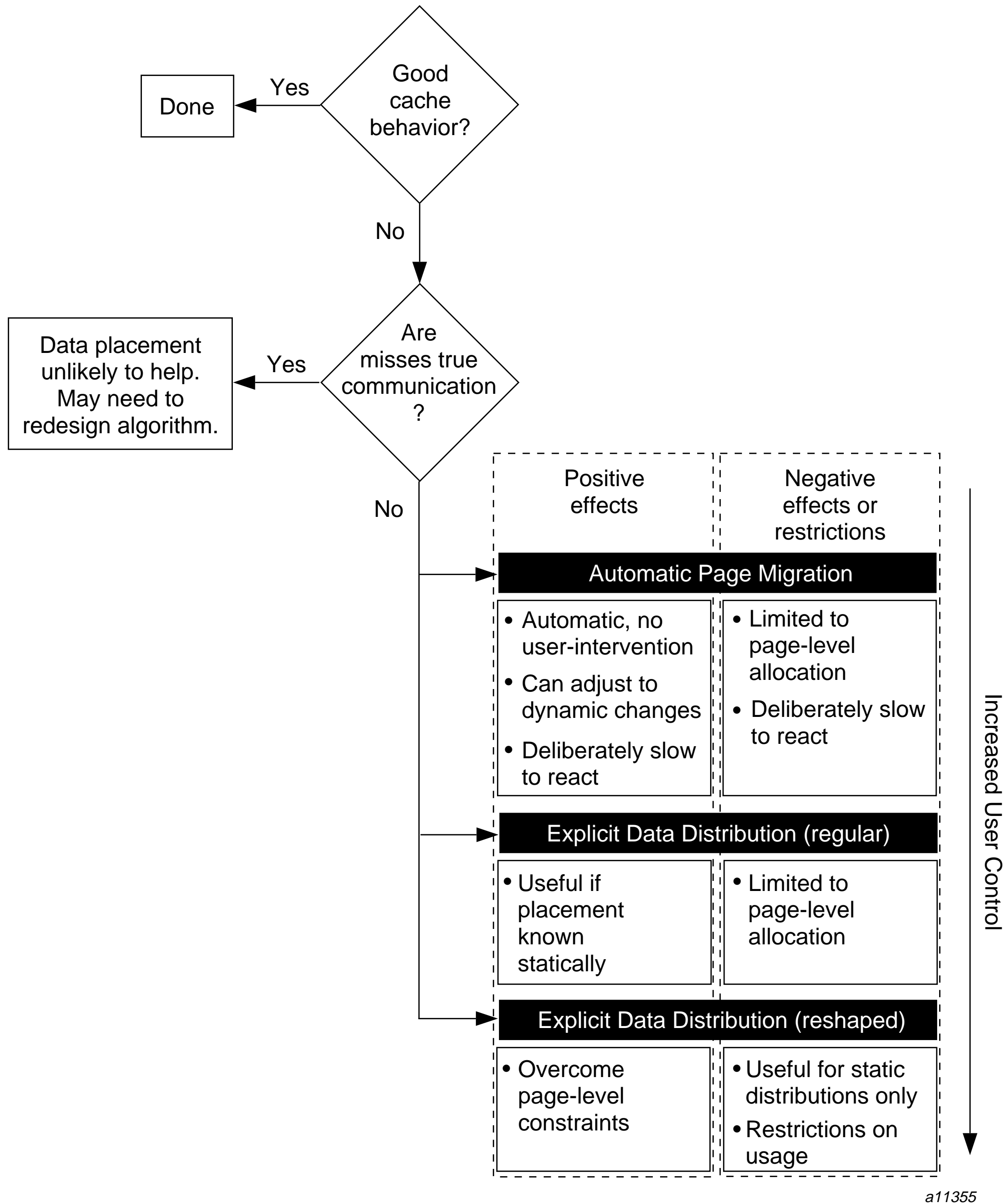


Figure 3. Cache behavior and solutions

5.1.2 Choosing a tuning method

For a given data structure in the program, you can choose between the automatic page migration method or the data distribution method. Your choice will be based on the following criteria:

- If the program repeatedly references the data structure and benefits from reuse in the cache, data placement is not needed.
- If the program incurs a large number of cache misses on the data structure, then you should identify the desired distribution in the array dimensions (such as `BLOCK` or `CYCLIC`) based on the desired parallelism in the program.

  The following example suggests a `A(BLOCK, *)` distribution:

  ```
  !$DOACROSS
  DO I = 2, N
      DO J = 2, N
        A(I,J) = 3*I + 4*J + A(I, J-1)
      END DO
  END DO
  ```

  However, the following example suggests a `A(*, BLOCK)` distribution:

  ```
  DO I = 2, N
  !$DOACROSS
      DO J = 2, N
        A(I,J) = 3*I + 4*J + A(I-1, J)
      END DO
  END DO
  ```

After identifying the desired distribution, you can select either *regular* or *reshaped* distribution based on the size of an individual processor's portion of the distributed array. Regular distribution is useful only if each processor's portion is substantially larger than the page size in the underlying system (16 Kbytes on the Origin series systems). Otherwise, regular distribution is probably not useful, and you should use the `DISTRIBUTE_RESHAPE` directive, where the compiler changes the layout of the array to overcome page-level constraints.

For example, consider the following code:

```
REAL(KIND=8) A(M, N)
!$DISTRIBUTE A(BLOCK, *)
```

In this example, the size of each processor's portion is approximately m/P elements ($8 \times (m/P)$ bytes), where P is the number of processors. If m is

1,000,000 then each processor's portion is likely to exceed a page and regular distribution is sufficient. However, if *m* is 10,000 then DISTRIBUTE_RESHAPE is required to obtain the desired distribution.

In contrast, consider the following distribution:

```
!$DISTRIBUTE A(*, BLOCK)
```

In this example, the size of each processor's portion is approximately $(m \times n)/P$ elements ($8 \times (m \times n)/P$ bytes). Therefore, if *n* is 100, for example, regular distribution may be sufficient even if *m* is only 10,000.

Distributing the outer dimensions of an array increases the size of an individual processor's portion (favoring regular distribution), but distributing the inner dimensions is more likely to require reshaped distribution.

The IRIX operating system on Origin series systems follows a default "first-touch" page-allocation policy. This means that each page is allocated from the local memory of the processor that incurs a page-fault on that page. Therefore, in programs where the array is initialized and is consequently first referenced in parallel, even a regular distribution directive may not be necessary, because the underlying pages are allocated from the desired memory location automatically due to the first-touch policy.

5.2 Directives for performance tuning

The MIPSpro 7 Fortran 90 compiler supports directives for performance tuning on Origin series systems. You must be licensed for the MIPSpro Automatic Parallelization Option in order for these directives to be honored. In addition, the -MP, -mp, or -pfa options must be in effect.

The directives supported are as follows:

- DISTRIBUTE
- DISTRIBUTE_RESHAPE
- DOACROSS
- DYNAMIC
- PAGE_PLACE
- REDISTRIBUTE

The following sections describe the syntax of these directives.

5.2.1 DISTRIBUTE, DISTRIBUTE_RESHAPE, and REDISTRIBUTE

The DISTRIBUTE directive determines the data distribution for an array. The DISTRIBUTE_RESHAPE directive dynamically redistributes an array. The REDISTRIBUTE directive performs data distribution with reshaping.

The format of this directive is as follows:

```
!$DISTRIBUTE array (dist1,dist2) [ONTO (target1, target2 [, targetN]
   ...)]

!$DISTRIBUTE_RESHAPE array (dist1,dist2) [ONTO (target1, target2 [,
   targetN] ...)]

!$REDISTRIBUTE array (dist1,dist2) [ONTO (target1, target2 [, targetN]
   ...)]
```

array Specify the name of an array.

dist Specify the type of distribution for the named array. The number of *dist* arguments specified must be equal to the number of array dimensions. *dist* can be one of the following:

- BLOCK. Indicates that BLOCK distribution should be used.
- CYCLIC [*expr*]. If *expr* is not specified, a chunk size of 1 is assumed. For performance reasons, do not specify an *expr* that is 3 or evaluates to 3; this may be incompatible when passing a reshaped array as a parameter to another routine.
- An asterisk (*). Indicates that the axis is not distributed.

target Specify the target processor topology. This argument to the ONTO clause specifies how to partition the processors across the distributed dimensions. There must be one *target* argument specified for each BLOCK and CYCLIC distribution specified.

The data distribution directives and DOACROSS NEST clause have an optional ONTO clause. The ONTO clause allows you to specify the processor topology when two (or more) dimensions of processors are required.

The following example array is distributed in two dimensions, so you can use the ONTO clause to specify how to partition the processors across the distributed dimensions:

```
! ASSIGN PROCESSOR IN THE RATIO 1:2 TO THE TWO DIMENSION
REAL(KIND=8) A(100, 200)
!$DISTRIBUTE A (BLOCK, BLOCK) ONTO (1, 2)
```

You can supply a DISTRIBUTE directive on a formal parameter, thereby specifying the distribution on the incoming actual parameter. If different calls to the subroutine have parameters with different distributions, you can omit the DISTRIBUTE directive on the formal parameter. Data affinity loops in that subroutine are automatically implemented through a run-time lookup of the distribution. This is allowed only for regular data distribution. For reshaped array parameters, the distribution must be fully specified on the formal parameter.

For more information on using the data distribution directives, see Section 5.3, page 118.

5.2.2 DOACROSS

The DOACROSS directive indicates to the compiler that it should run iterations of the subsequent DO loop in parallel. This directive must appear directly before the loop that is to be operated on, and it remains in effect for that loop only. The format of this directive is as follows:

```
!$DOACROSS [ clause [, clause ] ...]
```

clause This directive accepts one or more standard *clause* arguments. The AFFINITY clause allows you to specify either data affinity scheduling or thread affinity scheduling. The NEST clause allows you to specify nested DOACROSS statements.

For information on all the standard accepted *clause* arguments, see Section 4.2.1, page 67.

For information on the AFFINITY clause, see Section 5.2.2.1, page 112. For information on the NEST clause, see Section 5.2.2.2, page 114.

5.2.2.1 AFFINITY clause

Affinity scheduling controls the mapping of iterations of a parallel loop for execution onto the underlying threads. The DOACROSS directive with the

AFFINITY clause must immediately precede the loop to which it applies, and it is in effect only for that loop.

You can specify affinity scheduling with an additional clause to a DOACROSS directive. An AFFINITY clause, if supplied, overrides an MP_SCHEDTYPE clause.

There are two type of affinity scheduling: data affinity and thread affinity.

The AFFINITY clause to the DOACROSS directive has the following format:

```
!$DOACROSS AFFINITY (int_expr, expr)

!$DOACROSS AFFINITY(do_variable) = DATA(array_element)

!$DOACROSS AFFINITY(do_variable) = THREAD(expr)
```

int_expr Specify an integer expression.

do_variable Specify the DO loop identifier.

array_element Enter an array element.

expr Specify a thread number. *do_variable* is executed on the thread number specified, modulo the number of threads.

Because the threads may need to evaluate *expr* in each iteration of the loop, the variables used in the *expr* (other than the *do_variable*) must be declared SHARED and must not be modified during the execution of the loop. Violating these rules can lead to incorrect results. For information on declaring shared variables, see Section 4.2.1.5, page 70.

If the *expr* does not depend on the DO variable, all iterations execute on the same thread and do not benefit from parallel execution.

The default MP_SCHEDTYPE for parallel loops is SIMPLE. However, when -O3 is in effect, loops that reference reshaped arrays default to data affinity scheduling for the most frequently accessed reshaped array in the loop (chosen by the compiler). To obtain SIMPLE scheduling even at -O3, you can explicitly specify the MP_SCHEDTYPE clause on the DOACROSS directive.

Data affinity for loops with nonunit stride can sometimes result in nonlinear affinity expressions. In such situations the compiler issues a warning, ignores the affinity clause, and defaults to `SIMPLE` scheduling.

Example 1. The following code shows an example of data affinity:

```
!$DISTRIBUTE A(BLOCK)
!$DOACROSS AFFINITY(I) = DATA(A(A*I+B))
DO I = 1, N
     A(A*I+B) = 0
END DO
```

The multiplier for `A` and the constant term `B` must both be literal constants, with `A` greater than zero.

This example distributes the iterations of the parallel loop to match the data distribution specified for array `A`, such that iteration `I` is executed on the processor that owns element `A(A*I+B)` based on the distribution for `A`. The iterations are scheduled based on the specified distribution, and are not affected by the actual underlying data distribution, which may, for example, differ at page boundaries.

Example 2. In case of a multidimensional array, affinity is provided for the dimension that contains the loop index variable. The loop index variable cannot appear in more than one dimension in an `AFFINITY` directive. In the following example, the loop is scheduled based on the block distribution of the first dimension:

```
!$DISTRIBUTE A (BLOCK, CYCLIC(1))
!$DOACROSS AFFINITY(I) = DATA(A(I+3, J))
DO I = 1, N
    DO J = 1, N
      A(I+3, J) = A(I+3,J-1)
    END DO
END DO
```

Example 3. The following directive executes iteration `I` on the thread number given by the user-supplied expression (modulo the number of threads):

```
!$DOACROSS AFFINITY (I) = THREAD( expr)
```

5.2.2.2 `NEST` clause

The `NEST` clause on the `DOACROSS` directive allows you to exploit nested concurrency in a limited manner. Although true nested parallelism is not

supported, you can exploit parallelism across iterations of a perfectly nested loop nest.

The `NEST` clause to the `DOACROSS` directive has the following format:

```
!$DOACROSS NEST ( do_variable , do_variable [, do_variable ] ...)
   [ONTO (target1, target2 [, targetn] ...)]
```

index Specify a *do_variable* name that identifies a subsequent loop. At least two *do_variable* names must be specified. The loops identified must be perfectly nested.

target Specify the target processor topology. The `ONTO` clause allows you to specify the processor topology when two (or more) dimensions of processors are required. This argument specifies how to partition the processors across the distributed dimensions. *target* can be either an integer expression or an asterisk (`*`).

Example 1. In a nested `DOACROSS` with two or more nested loops, you can use the `ONTO` clause to specify the partitioning of processors across the multiple parallel loops, as follows:

```
! USE 2 PROCESSORS IN THE OUTER LOOP,
! AND THE REMAINING IN THE INNER LOOP
!$DOACROSS NEST(I, J) ONTO(2, *)
       DO I = 1, N
           DO J = 1, M
               A(J,I) = ...
           END DO
       END DO
```

Example 2. The following directive specifies that the entire set of iterations across both loops can be executed concurrently:

```
!$DOACROSS NEST(I, J)
       DO I = 1, N
            DO J = 1, M
               A(I,J) = 0
            END DO
       END DO
```

It is restricted, however, in that loops `I` and `J` must be perfectly nested. No code is allowed between either the `DO I ...` and `DO J ...` statements or

between the END DO statements. You can also supply the nest clause with a PDO directive. For more information on the PDO directive, see Section 4.6.5, page 96.

You can combine a nested DOACROSS with an AFFINITY clause or with an MP_SCHEDTYPE clause specified as SIMPLE or INTERLEAVED (DYNAMIC and GSS are not currently supported). The default is SIMPLE scheduling, except when accessing reshaped arrays. For more information on the AFFINITY clause, see Section 5.2.2.1, page 112. For more information on the MP_SCHEDTYPE clause see Section 4.2.1.6, page 71.

The following code uses an AFFINITY clause:

```
!$DOACROSS NEST(I, J) AFFINITY(I,J) = DATA(A(I,J))
       DO I = 2, N-1
             DO J = 2, M-1
                 A(I,J) = A(I,J) + I*J
             END DO
       END DO
```

5.2.3 DYNAMIC

The DYNAMIC directive informs the compiler that a particular array can be dynamically redistributed. This directive is required for arrays in procedures that contain DOACROSS loops with data affinity for arrays in the loops.

By default, the compiler assumes that a distributed array is not dynamically redistributed, and it directly schedules a parallel loop for the specified data affinity. In contrast, a redistributed array can have multiple possible distributions, and data affinity for a redistributed array must be implemented in the run-time system based on the particular distribution.

However, the compiler does not know if an array is redistributed because the array may be redistributed in another procedure or in another file. Therefore, you must explicitly specify the DYNAMIC declaration for redistributed arrays. The DYNAMIC directive implements data affinity for that array at run time rather than at compile time. If you know an array has a specified distribution throughout the duration of a procedure, you do not have to supply the DYNAMIC directive. The result is more efficient compile time affinity scheduling. This directive is required only in those procedures that contain a DOACROSS loop with data affinity for that array. This tells the compiler that the array can be dynamically redistributed. Data affinity for such arrays is implemented through a run-time lookup.

The format of this directive is as follows:

```
!$DYNAMIC (array)
```

array Specify the name of an array.

The run-time lookup incurs some extra overhead compared to a direct compile-time implementation. Because the compiler assumes that a distributed array is not redistributed at run time, the distribution is known at compile time, and data affinity for the array can be implemented directly by the compiler. In contrast, because a redistributed array can have multiple possible distributions at run time, data affinity for a redistributed array is implemented in the run-time system based on the distribution at run time, incurring extra run-time overhead.

You can avoid this overhead when a procedure contains data affinity for a redistributed array and the distribution of the array for the entire duration of that procedure is known. In this situation, you can supply the `DISTRIBUTE` directive with the particular distribution and omit the `DYNAMIC` directive.

Because reshaped arrays cannot be dynamically redistributed, this is an issue only for regular data distribution.

5.2.4 PAGE_PLACE

The `PAGE_PLACE` directive allows you to explicitly place irregular data structures in the physical memory of a particular processor.

The format of this directive is as follows:

```
!$PAGE_PLACE (object, size, threadnum)
```

object Specify the name of the object.

size Specify the size of *object*, in bytes.

threadnum Specify the processor number upon which *object* is to be placed.

This directive causes all the pages spanned by the virtual address range (*address* to *address+size*) to be allocated from the local memory of processor number *threadnum*. It is an executable statement; therefore, you can use it to place either statically or dynamically allocated data.

An example of this directive is as follows:

```
REAL(KIND=8) A(100)
!$PAGE_PLACE (A, 800, 3)
```

5.3 Using the data distribution directives

The data distribution directives, DISTRIBUTE, REDISTRIBUTE, and DISTRIBUTE_RESHAPE, allow you to specify distributions for array data structures. For irregular data structures, the directives can explicitly place data directly on a specific processor.

The DISTRIBUTE, DYNAMIC, and DISTRIBUTE_RESHAPE directives are declarations that must be specified in the declaration part of the program, along with the array declaration. The REDISTRIBUTE directive is an executable statement and can appear in any executable portion of the program.

You can specify a data distribution directive for any local, global, or common block array. Each dimension of a multidimensional array can be independently distributed. The possible distribution types for an array dimension are BLOCK, CYCLIC[(*expr*)], and *, as follows:

- A BLOCK distribution is one that partitions the elements of the dimension of size *N* into *P* blocks (one per processor), with each block of size *B* = ceiling (*N*/*P*).

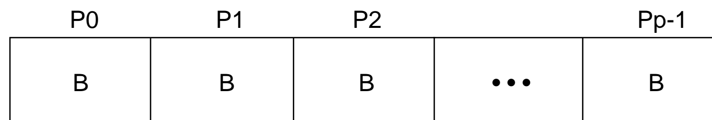


Figure 4. Block distribution

- A CYCLIC distribution can include an *expr* to indicate the chunk size. A chunk size that is either greater than 1 or is determined at run time is sometimes also called BLOCK-CYCLIC.

- The * distribution indicates that the array is not distributed.

A BLOCK distribution

A CYCLIC[(*expr*)] distribution partitions the elements of the dimension into pieces of size *expr* each and distributes them sequentially across the processors:

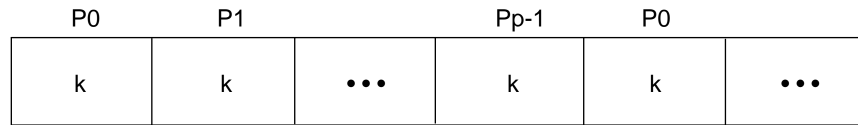


Figure 5. Cyclic distribution

A distributed array is distributed across all of the processors being used in that particular execution of the program, as determined by the MP_SET_NUMTHREADS environment variable. If a distributed array is distributed in more than one dimension, then by default the processors are apportioned as equally as possible across each distributed dimension. For example, if an array has two distributed dimensions, then an execution with 16 processors assigns 4 processors to each dimension (4 x 4=16), whereas an execution with 8 processors assigns 4 processors to the first dimension and 2 processors to the second dimension. You can override this default and explicitly control the number of processors in each dimension using the ONTO clause with a data distribution directive.

5.3.1 Regular data distribution

The regular data distribution directives try to achieve the desired distribution solely by influencing the mapping of virtual addresses to physical pages without affecting the layout of the data structure. Because the granularity of data allocation is a physical page (at least 16 Kbytes), the achieved distribution is limited by the underlying page granularity. However, the advantages are that regular data distribution directives can be added to an existing program without any restrictions, and can be used for affinity scheduling.

For example, the following directive dynamically redistributes array A:

```
!$REDISTRIBUTE A (BLOCK, CYCLIC(K))
```

The REDISTRIBUTE directive is an executable statement that changes the distribution permanently (or until another REDISTRIBUTE statement). It also affects subsequent affinity scheduling.

The DYNAMIC directive specifies that the named array is redistributed in the program, and is useful in controlling affinity scheduling for dynamically redistributed arrays.

5.3.2 Data distribution with reshaping

Similar to regular data distribution, the RESHAPE directive specifies the desired distribution of an array. In addition, however, the RESHAPE directive declares that the program makes no assumptions about the storage layout of that array. The compiler performs aggressive optimizations for reshaped arrays that violate standard Fortran layout assumptions, but it guarantees the desired data distribution for that array.

As shown in the following example, the RESHAPE directive accepts the same distributions as the regular data distribution directive:

```
!$DISTRIBUTE_RESHAPE (BLOCK, CYCLIC(1))
```

5.3.2.1 Restrictions on Reshaped Arrays

Because the DISTRIBUTE_RESHAPE directive specifies that the program does not depend on the storage layout of the reshaped array, restrictions on the arrays that can be reshaped include the following:

- The distribution of a reshaped array cannot be changed dynamically (that is, there is no REDISTRIBUTE_RESHAPE directive).
- Initialized data cannot be reshaped.
- Arrays that are explicitly allocated through the alloca(3C) or MALLOC(3F) routines and accessed through Cray pointers cannot be reshaped.
- An array that is equivalenced to another array cannot be reshaped.
- I/O for a reshaped array cannot be mixed with namelist I/O or a function call in the same I/O statement.
- A common block containing a reshaped array cannot be loaded with the -Xlocal option on ld(1).

Caution: This user error is not detected by the compiler or loader.

There are two possible outcomes if a reshaped array is passed as an actual parameter to a subroutine:

- The array is passed in its entirety; that is, CALL FUNC(A) passes the entire array A, whereas CALL FUNC(A(I,J)) passes a portion of A. The compiler automatically clones a copy of the called subroutine and compiles it for the incoming distribution. The actual and formal parameters must match in the number of dimensions, and the size of each dimension.

You can restrict a subroutine to accept a particular reshaped distribution on a parameter by specifying a `DISTRIBUTE_RESHAPE` directive on the formal parameter within the subroutine. All calls to this subroutine with a mismatched distribution will lead to compile time or load time.

- A portion of the array can be passed as a parameter, but the callee must access only a single processor's portion. If the callee exceeds a single processor's portion, the results are undefined. You can use intrinsics to access details about the array distribution.

5.3.2.2 Error detection for reshaped arrays

Most errors in accessing reshaped arrays are detected either at compile time or at load time. These errors include:

- Inconsistencies in reshaped arrays across common blocks (including across files).
- Using the `EQUIVALENCE` statement to declare a reshaped array as equivalent to another array.
- Inconsistencies in reshaped distributions on actual and dummy arguments.
- Other errors such as disallowed I/O statements involving reshaped arrays, reshaping initialized data, or reshaping dynamically allocated data.

Errors such as matching the declared size of an array dimension typically can be caught only at run time. You can use the `-MP:CHECK_RESHAPE=ON` option on the `f90`(1) command to perform these tests at run time. These run-time checks are not generated by default because they incur overhead, but they are useful during program development.

The types of run-time checks performed can detect the following:

- Inconsistencies in array bounds declarations on each actual and dummy argument
- Inconsistencies in declared bounds of a dummy argument that corresponds to a portion of a reshaped actual argument

5.3.2.3 Implementation of reshaped arrays

The compiler transforms a reshaped array into a pointer to a *processor array*. The processor array has one element per processor, with the element pointing to the portion of the array local to the corresponding processor.

Figure 6 shows the effect of a DISTRIBUTE_RESHAPE directive with a BLOCK distribution on a one-dimensional array. *N* is the size of the array dimension, *P* is the number of processors, and *B* is the block-size on each processor, CEILING=(*N*/*P*).

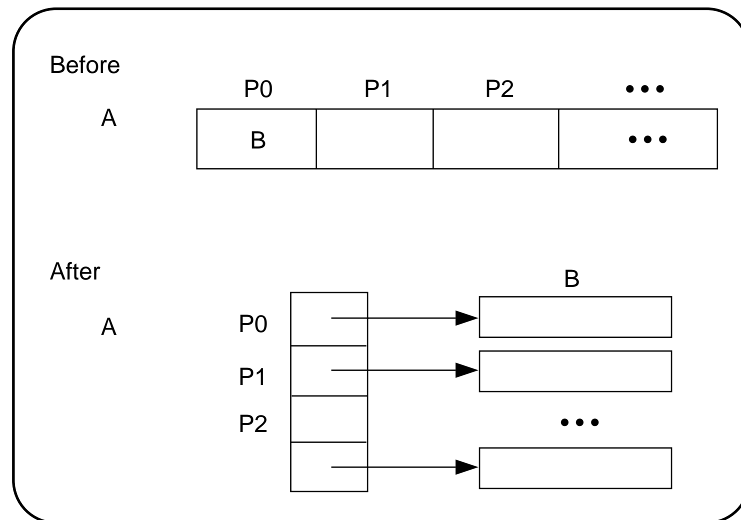


Figure 6. Implementation of BLOCK distribution

With this implementation, an array reference A(I) is transformed into a two-dimensional reference A[I/B][I%B] (in C syntax with C dimension order), where *B* is the size of each block, and given by CEILING(*N*/*P*). Thus A[I/B] points to a processor's local portion of the array, and A[I/B][I%B] refers to a specific element within the local processor's portion.

A CYCLIC distribution with a chunk size of 1 is implemented as shown in Figure 7.

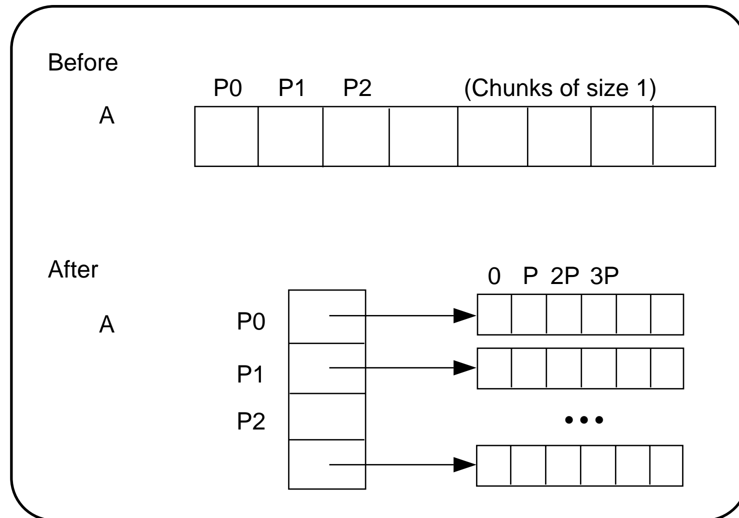


Figure 7. Implementation of CYCLIC(1) distribution

An array reference, A(I), is transformed to A[I%P][I/P], where *P* is the number of threads in that distributed dimension.

Finally, a CYCLIC distribution with a chunk size that is either a constant greater than 1 or a run-time value (also called BLOCK-CYCLIC) is implemented as Figure 8 shows.

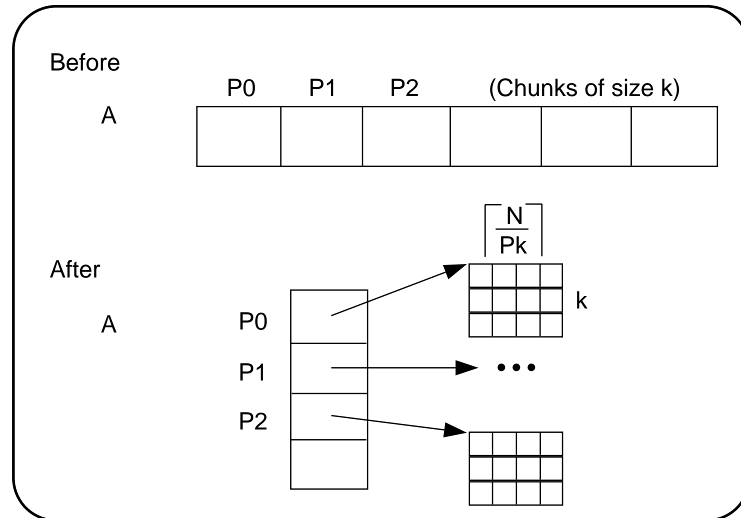


Figure 8. Implementation of BLOCK-CYCLIC Distribution

An array reference, `A(I)`, is transformed to the three-dimensional reference `A[(I/K)%P][I/(PK)][I%K]`, where *P* is the total number of threads in that dimension, and *K* is the chunk size.

The compiler tries to optimize these divide/modulo operations out of inner loops through aggressive loop transformations such as blocking and peeling.

5.3.3 Regular versus reshaped data distribution

Regular distributions have an advantage in that they do not impose any restrictions on the distributed arrays and can be freely applied in existing codes. Furthermore, they work well for distributions where page granularity is not a problem. For example, consider a `BLOCK` distribution of the columns of a two-dimensional Fortran array of size `A(R,C)` (column-major layout) and distribution `(*, BLOCK)`. If the size of each processor's portion, `CEILING=(`*C*`/`*P*`)(`*R*`)(`*element_size*`)` is significantly greater than the page size (16KB on Origin2000 systems), then regular data distribution should be effective in placing the data in the desired fashion.

However, regular data distribution is limited by page-granularity considerations. For instance, consider a `(BLOCK,BLOCK)` distribution of a two-dimensional array in which the size of a column is much smaller than a page. Each physical

page is likely to contain data belonging to multiple processors, making the data distribution quite ineffective. However, data distribution may still be useful from the standpoint of affinity scheduling considerations.

Reshaped data distribution addresses the problems of regular distributions by changing the layout of the array in memory to guarantee the desired distribution. However, because the array no longer conforms to standard Fortran storage layout, there are restrictions on the usage of reshaped arrays.

Given both types of data distribution, you can choose between the two based on the characteristics of the particular array in an application.

5.4 Examples

The following sections provide several examples of data distribution and affinity scheduling.

5.4.1 Distributing columns of a matrix

Example 1. This example distributes the columns of a matrix sequentially. Such a distribution places data effectively only if the size of an individual column exceeds that of a page.

```
REAL(KIND=8) A(N, N)
! DISTRIBUTE COLUMNS IN CYCLIC FASHION
!$DISTRIBUTE A (*, CYCLIC(1))

! PERFORM GAUSSIAN ELIMINATION ACROSS COLUMNS
! THE AFFINITY CLAUSE DISTRIBUTES THE LOOP ITERATIONS BASED
! ON THE COLUMN DISTRIBUTION OF A
DO I = 1, N
!$DOACROSS AFFINITY(J) = DATA(A(I,J))
     DO J = I+1, N
!        ... REDUCE COLUMN J BY COLUMN I ...
     END DO
END DO
```

If the columns are smaller than a page, it may be beneficial to reshape the array. This is easily specified by using a DISTRIBUTE_RESHAPE directive in place of the DISTRIBUTE directive.

In addition to overcoming size constraints as shown in the preceding example, the DISTRIBUTE_RESHAPE directive is useful when the desired distribution is contrary to the layout of the array.

Example 2. If you want to distribute the rows of a two-dimensional matrix, this example uses the DISTRIBUTE_RESHAPE directiveto overcome the storage layout constraints to provide the desired distribution.

```
REAL(KIND=8) A(N, N)
! DISTRIBUTE ROWS IN BLOCK FASHION
!$DISTRIBUTE_RESHAPE A (BLOCK, *)
REAL(KIND=8) SUM(N)
!$DISTRIBUTE SUM(BLOCK)

! PERFORM SUM-REDUCTION ON THE ELEMENTS OF EACH ROW
!$DOACROSS LOCAL(J) AFFINITY(I) = DATA(A(I,J))
DO I = 1,N
     DO J = 1,N
        SUM(I) = SUM(I) + A(I,J)
     ENDDO
ENDDO
```

5.4.2 Using data distribution and data affinity scheduling

The following example demonstrates regular data distribution and data affinity. This example, run on a 4-processor Origin2000 server, uses simple block scheduling. Processor 0 calculates the values of the first 25,000 elements of A, processor 1 calculates the second 25,000 values of A, and so on. Arrays B and C are initialized using one processor. Therefore, all of the memory pages are touched by the master processor (processor 0) and are placed in processor 0's local memory.

Using data distribution changes the placement of memory pages for arrays A, B, and C to match the data reference pattern.

Without data distribution:

```
     REAL(KIND=8) A(1000000), B(1000000)
     REAL(KIND=8) C(1000000)
     INTEGER I

!$PAR PARALLEL SHARED(A, B, C) LOCAL(I)
!$PAR PDO
      DO I = 1, 100000
```

```
         A(I) = B(I) + C(I)
      END DO
!$PAR END PARALLEL
```

With data distribution:

```
     REAL(KIND=8) A(1000000), B(1000000)
     REAL(KIND=8) C(1000000)
     INTEGER I
!$DISTRIBUTE A(BLOCK), B(BLOCK), C(BLOCK)

!$PAR PARALLEL SHARED(A, B, C) LOCAL(I)
!$PAR PDO AFFINITY(I) = DATA(A(I))
      DO I = 1, 100000
         A(I) = B(I) + C(I)
      END DO
!$PAR END PARALLEL
```

5.4.3 Argument passing

Example 1. The following code shows how a distributed array can be passed as an argumentr to a subroutine that has a matching declaration for the dummy argument:

```
REAL(KIND=8) A(M, N)
!$DISTRIBUTE_RESHAPE A (BLOCK, *)
CALL FOO(A, M, N)
END

SUBROUTINE FOO(A, P, Q)
REAL(KIND=8) A(P, Q)
!$DISTRIBUTE_RESHAPE A (BLOCK, *)
!$DOACROSS AFFINITY(I) = DATA(A(I, J))
    DO I = 1, P
    END DO
END
```

Because the array is reshaped, it is required that the `DISTRIBUTE_RESHAPE` directive in the caller and the callee match exactly. Furthermore, all calls to subroutine `FOO` must pass in an array with the exact same distribution.

If the array was only distributed (not reshaped) in the preceding example, then subroutine `FOO` could be called from different places with different incoming distributions. In that case, you could omit the distribution directive on the

dummy argument, thereby ensuring that any data affinity within the loop is based on the distribution (at run time) of the incoming actual argument, as shown in this example:

```
REAL(KIND=8) A(M, N), B(P, Q)
REAL(KIND=8) A (BLOCK, *)
REAL(KIND=8) B (CYCLIC(1), *)
CALL FOO(A, M, N)
CALL FOO(B, P, Q)
---------------------------------------------------------
SUBROUTINE FOO(X, S, T)
REAL(KIND=8) X(S, T)

!$DOACROSS AFFINITY(I) = DATA(X(I+2, J))
DO I =
   ...
END DO
```

5.4.4 Redistributed arrays

Example 1. The following example shows how an array is redistributed at run time:

```
SUBROUTINE BAR(X, N)
REAL(KIND=8) X(N, N)
...
!$REDISTRIBUTE X (*, CYCLIC(expr))
...
END
---------------------------------------------------------
SUBROUTINE FOO
REAL(KIND=8) LOCALARRAY(1000, 1000)
!$DISTRIBUTE LOCALARRAY (*, BLOCK)
! THE CALL TO SUBROUTINE BAR MAY REDISTRIBUTE LOCALARRAY
!$DYNAMIC LOCALARRAY
...
CALL BAR(LOCALARRAY, 100)
! THE DISTRIBUTION FOR THE FOLLOWING DOACROSS
! IS NOT KNOWN STATICALLY
!$DOACROSS AFFINITY(I) = DATA(A(I, J))
END
```

Example 2. The following example illustrates a situation in which the DYNAMIC directive can be optimized away. The main routine contains local array A that is both distributed and dynamically redistributed. This array is passed as an argument to FOO before being redistributed and to FOO after being (possibly) redistributed. The incoming distribution for FOO is statically known; you can specify a DISTRIBUTE directive on the dummy argument, thereby obtaining more efficient static scheduling for the DOACROSS directive with data affinity. The subroutine BAR, however, can be called with multiple distributions, requiring run-time scheduling of the DOACROSS loop.

```
PROGRAM MAIN
!$DISTRIBUTE A (BLOCK, *)
!$DYNAMIC A
CALL FOO(A)
IF (X .NE. 17) THEN
     !$REDISTRIBUTE A (CYCLIC(X), *)
END IF
CALL BAR(A)
END

SUBROUTINE FOO (A)
! Incoming distribution is known to the user
!$DISTRIBUTE A(BLOCK, *)
!$DOACROSS AFFINITY(I) = DATA(A(I, J))
     ...
END

SUBROUTINE BAR(A)
! Incoming distribution is not known statically
!$DYNAMIC A
!$DOACROSS AFFINITY(I) = DATA(A(I, J))
     ...
END
```

5.4.5 Irregular distributions and thread affinity

This example consists of a large array that is conceptually partitioned into unequal portions, one for each processor. This array is indexed through index array IDX, which stores the starting index value and the size of each processor's portion.

```
REAL(KIND=8) A(N)
! IDX ---> INDEX ARRAY CONTAINING START INDEX INTO A (IDX(P, 0))
```

```
! AND SIZE (IDX(P, 1)) FOR EACH PROCESSOR
REAL(KIND=4) IDX (P, 2)
!$PAGE_PLACE (A(IDX(0, 0)), IDX(0, 1)*8, 0)
!$PAGE_PLACE (A(IDX(1, 0)), IDX(1, 1)*8, 1)
!$PAGE_PLACE (A(IDX(2, 0)), IDX(2, 1)*8, 2)
...
!$DOACROSS AFFINITY(I) = THREAD(I)
DO I = 0, P-1
!   ... PROCESS ELEMENTS ON PROCESSOR I
!     ... A(IDX(I, 0)) TO A(IDX(I,0)+IDX(I,1))
END DO
```

CF90 Directives [6]

The MIPSpro 7 Fortran 90 compiler, running on IRIX systems, recognizes some of the directives that are supported by the Cray Research CF90 compiler on UNICOS and UNICOS/mk systems. This chapter describes these directives.

Table 1 categorizes the directives according to purpose. It also indicates the pages that contain the main descriptions of the individual directives.

Table 1. Directives

Purpose and Name	Description
Local use of compiler features:	
`BOUNDS`, `NOBOUNDS`	Section 6.3.1, page 133
`FREE`, `FIXED`	Section 6.3.2, page 135
Autotasking:	
`DOALL`	Section 6.4.1, page 136
`DOPARALLEL`, `ENDDO`	Section 6.4.2, page 138
`GUARD`, `ENDGUARD`	Section 6.4.3, page 139
`PARALLEL`, `ENDPARALLEL`	Section 6.4.4, page 140
Miscellaneous:	
`ID`	Section 6.5.1, page 142
`IVDEP`	Section 6.5.2, page 144
`NAME`	Section 6.5.3, page 144

Note: The implementation of the following directives is deferred: `DOALL`, `DOPARALLEL`, `ENDDO`, `GUARD`, `ENDGUARD`, `PARALLEL`, and `ENDPARALLEL`.

6.1 Using directives

The following sections describe how to use the CF90 directives and the effects they have on programs.

6.1.1 Directive continuation

In the following example, an asterisk (*) appears in column 6 to indicate that the second line is a continuation of the preceding line:

```
!DIR$ NA
!DIR$*ME
```

The FIXED and FREE directives must appear alone on a directive line and cannot be continued.

If you want to specify more than one directive on a line, separate each directive with a comma. Some directives require that you specify one or more arguments; when specifying a directive of this type, no other directive can appear on the line.

Spaces can precede, follow, or be embedded within a directive, regardless of source form.

Do not use source preprocessor (#) directives within multiline compiler directives (CDIR$ or !DIR$).

6.1.2 Directive range and placement

The range and placement of directives is as follows:

- The FIXED and FREE directives can appear anywhere in your source code. All other directives must appear within a program unit.
- The BOUNDS and NOBOUNDS directives take effect at the point at which they appear in the source code.
- The ID directive does not apply to any particular range of code. It adds information to the *file*.o generated from the input program.
- The ENDDO directive must appear after the loop body of a DOPARALLEL loop, if it appears. The corresponding DOPARALLEL directive must be present.
- The following directives apply only to the next loop encountered lexically:
  - DOALL
  - DOPARALLEL
  - IVDEP

- The NAME directive does not apply to particular ranges of code. It is a declarative directive that alters the status of entities in ways that affect compilation.
- The following Autotasking directives must appear as pairs within a program unit:
  - GUARD, ENDGUARD
  - PARALLEL, ENDPARALLEL

6.1.3 Interaction of directives with the -x command line option

The -x option on the f90(1) accepts one or more directives as arguments. When your input is compiled, the compiler ignores directives named as arguments to the -x option. If you specify -x mipspro, all directives are ignored. If you specify -x *dirname*, a particular directive is ignored. For more information on this command line option, see Section 2.63, page 47.

6.2 Optimization directives

The following sections describe the directives used to control vectorization and tasking, which are as follows:

6.3 Local use of compiler features

Certain directives provide local control over specific compiler features. They are as follows:

- BOUNDS and NOBOUNDS
- FREE and FIXED

The following sections describe these directives.

6.3.1 Check array bounds: BOUNDS and NOBOUNDS

Array bounds checking provides a check of most array references at both compile time and run time to ensure that each subscript is within the array's declared size.

The -C option on the f90(1) command line controls bounds checking for a whole compilation. The BOUNDS and NOBOUNDS directives toggle the feature on and off within a program unit. Either directive can specify particular arrays or can apply to all arrays. The formats of these directives are as follows:

```
!DIR$ BOUNDS [ array [, array ] ... ]

!DIR$ NOBOUNDS [ array [, array ] ... ]
```

array The name of an array. The name cannot be a subobject of a derived type. When no array name is specified, the directive applies to all arrays.

BOUNDS remains in effect for a given array until the appearance of a NOBOUNDS directive that applies to that array, or until the end of the program unit. Bounds checking can be enabled and disabled many times in a single program unit.

Note: To be effective, these directives must follow the declarations for all affected arrays. It is suggested that they be placed at the end of a program unit's specification statements unless they are meant to control particular ranges of code.

The bounds checking feature detects any reference to an array element whose subscript exceeds the array's declared size. For example:

```
      REAL A(10)
!  DETECTED AT COMPILE TIME:
      A(11) = X
!  DETECTED AT RUN TIME IF IFUN(M) EXCEEDS 10:
      A(IFUN(M)) = W
```

The compiler generates a message when it detects an out-of-bounds subscript. If the compiler cannot detect the out-of-bounds subscript (for example, if the subscript includes a function reference), a message is issued for out-of-bound subscripts when your program runs.

Bounds checking increases program run time. If an array's last dimension declarator is *, checking is not performed on the last dimension's upper bound. Arrays in formatted WRITE and READ statements are not checked.

If bounds checking detects an out-of-bounds array reference, a message is issued and the program halts.

6.3.2 Specify source form: FREE and FIXED

The FREE and FIXED directives specify whether the source code in the program unit is written in free source form or fixed source form. The FREE and FIXED directives override the -fixedform and -freeform options, if specified, on the command line. The formats of these directives are as follows:

```
!DIR$ FREE

!DIR$ FIXED
```

These directives apply to the source file in which they appear, and they allow you to switch source forms within a source file.

You can change source form within an INCLUDE file. After the INCLUDE file has been processed, the source form reverts back to the source form that was being used prior to processing of the INCLUDE file.

Note: The source preprocessor does not recognize the FREE and FIXED directives. These directives must not be specified in a file that is submitted to the source preprocessor.

6.4 Autotasking directives (deferred implementation)

If your system includes multiple central processing units (CPUs), your program may be able to make use of *multitasking*, or running simultaneously on more than one CPU. This technology speeds up program execution by decreasing elapsed time. You can determine the number of CPUs on your system by entering the hinv(1) command.

The compiler automatically recognizes many parallel coding constructs, and it compiles them for multitasking without requiring additional user input; this capability is called *Autotasking*.

Autotasking directives let you specify the level of parallelism desired. You can start and end parallel processing at any number of suitable points within a subprogram. These directives are useful when the compiler fails to recognize parallelism that you know exists. This can occur, for example, when you have subroutine calls that can be executed in parallel.

This section provides an overview of the Autotasking directives recognized by the compiler.

Caution: The ability to use Autotasking directives in a subprogram that host associates a variable can result in undefined behavior. This applies only to Autotasking directives; it does not apply to parallelism detected by the compiler.

A branch out of a parallel region is not permitted and can produce incorrect results.

Autotasking directives control the way the compiler multitasks your program. You can insert tasking directive lines directly into your source code. The compiler supports the following Autotasking directives:

- DOALL
- DOPARALLEL, ENDDO
- GUARD, ENDGUARD
- PARALLEL, ENDPARALLEL

The following sections describe the Autotasking directives.

6.4.1 Mark parallel loop: DOALL (deferred implementation)

The !MIC$ DOALL directive indicates that the DO loop beginning on the next line may be executed in parallel by multiple processors. No directive is needed to end a DOALL loop, (that is, the DOALL initiates a parallel region that contains only a DO loop with independent iterations). The loop index variable for a DOALL must be specified as a PRIVATE variable.

When the compiler generates code for a !MIC$ DOALL, all the variables and arrays in the region must be defined in a SHARED or PRIVATE parameter.

The format of this directive is as follows:

```
!MIC$ DOALL parameter [ [,] parameter ] ...
```

parameter Table 2 describes parameters for the DOALL directive. More than one parameter can appear

on the directive, but they must be separated by commas or blanks.

Table 2. Autotasking directive *parameter*

parameter	Description
IF(*expr*)	Performs a run-time test to choose between uniprocessing and multiprocessing. When not specified, multiprocessing is chosen if the loop is not in a routine that was called from within a parallel region. The logical expression (*expr*) determines (at run time) whether multiprocessing will occur. When *expr* is true, multiprocessing is enabled.
MAXCPUS(*n*)	Specifies the maximum number of CPUs that the parallel region can use effectively. Does not ensure that *n* processors will be assigned. This is the optimal maximum. The *n* argument must be of type integer. Argument *n* can be a constant, a variable, or an expression. Both of the following are valid specifications: MAXCPUS (2) MAXCPUS (NUM)
PRIVATE(*var* [,*var*] ...)	Specifies that the variables listed will have *private* scope; that is, each task (original or helper) will have its own private copy of these variables. The PRIVATE clause identifies those variables that are not shared between parallel processes. A variable cannot be declared both PRIVATE and SHARED. The loop control variable of the DOALL loop cannot be specified as SHARED; it must be specified as PRIVATE. Variables cannot be subobjects (that is, array elements or components of derived types).
SAVELAST	Specifies that the values of private variables, from the final iteration of a DOALL directive, will continue in the original task after execution of the iterations of the DOALL. By default, private variables are not guaranteed to retain the last iteration values. SAVELAST can be used only with DOALL, and if the full iteration set is not completed (for example, if the loop is exited early), the values of private variables are indeterminate.
SHARED(*var* [,*var*] ...)	Specifies that the variables listed will have *shared* scope; that is, they are accessible to both the original task and all helper tasks. The SHARED clause identifies those variables that are shared between parallel processes. A variable cannot be declared both PRIVATE and SHARED. The loop control variable of the DOALL loop cannot be specified as SHARED; it must be specified as PRIVATE. Variables cannot be subobjects (that is, array elements or components of derived types).

6.4.2 Mark parallel loop: DOPARALLEL and ENDDO (deferred implementation)

The !MIC$ DOPARALLEL directive indicates that the DO loop beginning on the next line may be executed in parallel by multiple processors. No directive is needed to end a DOPARALLEL loop.

The !MIC$ ENDDO directive extends a control structure beyond the DO loop. Without a !MIC$ ENDDO directive, all CPUs synchronize immediately after the loop, so that no processors can continue executing until all of the iterations are done. A !MIC$ ENDDO directive moves this point of synchronization from the end of the loop to the line of the !MIC$ ENDDO directive.

This lets the compiler use parallelism in loops containing some forms of reduction computations. These directives can be used only within a parallel region bounded by the PARALLEL and ENDPARALLEL directives.

Every variable in a parallel region must be declared as PRIVATE or SHARED.

The formats for these directives are as follows:

```
!MIC$ DOPARALLEL [parameter]

!MIC$ ENDDO
```

The *parameters* are described in Table 2, page 137. Only one *parameter* can be used for a given DO loop.

In the following example, a parallel region is defined by PARALLEL and ENDPARALLEL. A reduction computation is implemented by a DOPARALLEL/ENDDO pair, which ensures that all contributions to SUM and BIG are included, and GUARD/ENDGUARD, which protects the updating of shared variables SUM and BIG.

```
      SUM = 0.0
      BIG = -1.0
!MIC$ PARALLEL PRIVATE(XSUM,XBIG,I)
!MIC$*         SHARED(SUM,BIG,AA,BB,CC)
      XSUM = 0.0
      XBIG = -1.0
!MIC$ DOPARALLEL
      DO I = 1, 2000
         :
         XSUM = XSUM + (AA(I)*(BB(I)-CC(AA(I))))
         XBIG = MAX(ABS(AA(I)*BB(I)), XBIG)
```

```
          :
      END DO
!MIC$ GUARD
      SUM = SUM + XSUM
      BIG = MAX(XBIG,BIG)
!MIC$ ENDGUARD
!MIC$ ENDDO
!MIC$ ENDPARALLEL
```

6.4.3 Critical region: GUARD and ENDGUARD (deferred implementation)

The `!MIC$ GUARD` and `!MIC$ ENDGUARD` directives delimit a critical region, providing the necessary synchronization to protect or guard the code inside the critical region. A *critical region* is a code block that is to be executed by only one processor at a time, although all processors that enter a parallel region will execute it.

The formats for these directives are as follows:

```
!MIC$ GUARD [n]

!MIC$ ENDGUARD [n]
```

n Mutual exclusion flag; two regions with the same flag cannot be active concurrently. *n* must be of type integer and can be a variable or an expression, from which the low-order 6 bits are used. For example, `GUARD 1` and `GUARD 2` can be active concurrently, but two `GUARD 7` directives cannot.

For optimal performance, no *n* should be specified. Otherwise, *n* should be an integer constant; a general expression can be used for the unusual case that the critical region number must be passed to a lower-level routine. When *n* is not provided, the critical region blocks only other instances of itself, but no other critical regions. Critical regions may appear anywhere in a program. That is, they are not limited to parallel regions.

Numbered `GUARD` directives are not supported. They are implemented as unnamed `GUARD` directives. This can lead to deadlock if the user has nested `GUARD` directives.

6.4.4 Mark parallel region: PARALLEL and ENDPARALLEL (deferred implementation)

The !MIC$ PARALLEL and !MIC$ ENDPARALLEL directives mark, respectively, the beginning and end of a parallel region. Parallel regions are combinations of redundant code blocks and partitioned code blocks. The formats for these directives are as follows:

```
!MIC$ PARALLEL [parameter [ [,] parameter ] ...]

!MIC$ ENDPARALLEL
```

The *parameter*s are described in Table 2, page 137.

The PARALLEL directive indicates where multiple processors enter execution. The portion of code that all processors execute until reaching a DOPARALLEL directive is called a ***redundant code block***. Because the iterations of the DO loop within a DOPARALLEL directive are distributed across available processors, this portion of code is called the ***partitioned code block***. The scope of a variable in a parallel region is either shared or private. Shared variables are used by all processors; private variables are unique to a processor.

When the compiler generates code for a !MIC$ PARALLEL directive, all the variables and arrays in the region must be defined in a SHARED or PRIVATE parameter.

6.4.5 Examples (deferred implementation)

The following examples show shared and private variables and arrays.

6.4.5.1 Read-only variables

The following examples show read-only variables:

```
!MIC$ DOALL PRIVATE(I) SHARED(N1,N2,A)
      DO I = N1, N2
      ...= A
      END DO
```

A is a shared variable because it is a read-only variable. All processors share the same location for A.

```
!MIC$ DOALL SHARED(N1,N2,M1,M2,V) PRIVATE(I,J)
      DO 10 I = N1, N2
```

```
      DO 10 J = M1, M2
         ... = V(J)
      END DO
```

V is shared because it is a read-only array. N1, N2, M1, and M2 are also shared because they are read-only variables. I and J are written and then read, so they are private variables.

6.4.5.2 Array indexed by loop index

The following example shows an array indexed by the loop index:

```
!MIC$ DOALL SHARED(N1,N2,V,U,J) PRIVATE(I,T)
      DO I = N1, N2
        T = V(I)
        U(I,J) = T
      END DO
```

U and V are shared arrays because they are indexed by the loop index. All processors share the same location for V and U. T is written and then read, so it is a private variable. J is shared because it is a read-only variable.

6.4.5.3 Read-then-write variables

The following example shows read-then-write variables:

```
      SUM = 0.0
!MIC$ DOALL SHARED(N1,N2,V,SUM) PRIVATE(I,T)
      DO I = N1, N2
        T = V(I)
!MIC$ GUARD
        SUM = SUM + T
!MIC$ ENDGUARD
      END DO
```

SUM is a shared variable because it is read before it is written. Special care is needed in writing into a shared variable that is not indexed by the loop control variable.

6.4.5.4 Write-then-read variables and arrays

The following example shows write-then-read variables and arrays:

```
!MIC$ DOALL SHARED(N1,N2,M1,M2) PRIVATE(I,J,V)
      DO 10 I = N1, N2
      DO 10 J = M1, M2
      V(J) = ...
      ... = V(J)
      END DO
```

`V` is written to and then read. It must be a private array.

6.5 Miscellaneous directives

The following directives allow you to use several different compiler features:

- `ID`
- `IVDEP`
- `NAME`

6.5.1 Create identification string: `ID`

The `ID` directive inserts a character string into the *file*`.o` produced for a Fortran source file. The format of this directive is as follows:

```
!DIR$ ID "character_string"
```

character_ string — The character string to be inserted into *file*`.o`. The syntax box shows quotation marks as the *character_string* delimiter, but you can use either apostrophes (`' '`) or quotation marks (`" "`).

The *character_string* can be obtained from *file*`.o` in one of the following ways:

- Method 1 — Using the `what`(1) command. To use the `what`(1) command to retrieve the character string, begin the character string with the characters `@(#)`. For example, assume that `id.f` contains the following source code:

  ```
  !DIR$ ID  "File: id.f   Date: 1 July 1997"
        PRINT *, 'hello'
        END
  ```

  The next step is to use file `id.o` as the argument to the `what`(1) command, as follows:

```
% what id.o
% id.o:
%   file.f 03 February 1997
```

Note that `what`(1) does not include the special sentinel characters in the output.

In the following example, *character_string* does not begin with the characters `@(#)`. The output shows that `what`(1) does not recognize the string.

Input file `id2.o` contains the following:

```
!DIR$ ID  'file.f 03 February 1997'
      PRINT *, 'Hello, world'
      END
```

The `what`(1) command generates the following output:

```
% what id2.o
% id2.o:
```

- Method 2 — Using `strings`(1) or `od`(1). The following example shows how to obtain output using the `strings`(1) command.

  Input file `id.f` contains the following:

```
!DIR$ ID  "File: id.f   Date: 1 July 1997"

         PRINT *, 'hello'
         END
```

  The `strings`(1) command generates the following output:

```
% f90 -c id.o
% strings id.o
File: id.f  Date: 1 July 1997
% od -c id.o
```

... portion of dump deleted

```
0002300  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0002320   F   i   l   e   :       i   d   .   f               D   a   t
0002340   e   :       1       J   u   l   y       1   9   9   7 001  \0
0002360  \0  \0  \0  \0 024 003 240 031  \0  \0 203 031  \0  \0 205 005
```

... portion of dump deleted

6.5.2 Ignore dependencies: IVDEP

When the IVDEP directive appears before a loop, the compiler ignores vector dependencies, including explicit dependencies, in any attempts to vectorize the loop. IVDEP applies to the first DO loop or DO WHILE loop that follows the directive. The directive applies to only the first loop that appears after the directive within the same program unit. Whether or not IVDEP is used, conditions other than vector dependencies can inhibit vectorization. The format of this directive is as follows:

```
!DIR$ IVDEP
```

If a loop with an IVDEP directive is enclosed within another loop with an IVDEP directive, the IVDEP directive on the outer loop is ignored.

When the compiler vectorizes a loop, it may reorder the statements in the source code to remove vector dependencies. When IVDEP is specified, the statements in the loop are assumed to contain no dependencies as written, and the compiler does not reorder loop statements.

Note: The description for this directive describes how this directive works when the -OPT:cray_ivdep=ON command line option is in effect. When -OPT:cray_ivdep=OFF is in effect, you may notice different behavior. For more information on controlling this directive, see Section 2.47.3, page 31.

6.5.3 External name mapping directive: NAME

The NAME directive allows you to specify a case-sensitive external name, or a name that contains characters outside of the Fortran character set, in a Fortran program. This directive must appear inside a program unit. The case-sensitive external name is specified on the NAME directive, in the following format:

```
!DIR$ NAME (fortran_name="external_name"
   [, fortran_name="external_name" ] ... )
```

fortran_name The name used for the object throughout the Fortran program.

external_name The external form of the name.

Rules for Fortran naming do not apply to the *external_name* string; any character sequence is valid. You can use this directive, for example, when writing calls to C routines.

Example:

```
      PROGRAM MAIN
!DIR$ NAME (FOO="XyZ")
      CALL FOO            ! XyZ IS REALLY BEING CALLED
      END PROGRAM
```

Source Preprocessing [7]

Source preprocessing can help you port a program from one platform to another by allowing you to specify source text that is platform-specific.

For a source file to be preprocessed automatically, it must have an uppercase extension, either `.F` (for a file in fixed source form) or `.F90` (for a file in free source form). To specify preprocessing of source files with other extensions, including lowercase ones, use the `-cpp`, `-E`, or `-ftpp` options described in Chapter 2, page 5.

7.1 General rules

You can alter the source code through source preprocessing directives. These directives are fully explained in Section 7.2, page 148. The directives must be used according to the following rules:

- Do not use source preprocessor (#) directives within multiline compiler directives.
- You cannot include a source file that contains an `#if` directive without a balancing `#endif` directive within the same file.

  The `#if` directive includes the `#ifdef` and `#ifndef` directives.
- If a directive is too long for one source line, the backslash character (\) is used to continue the directive on successive lines. Successive lines of the directive can begin in any column (up to the column limit of 132).

  The backslash character (\) can appear in any location within a directive in which whitespace can occur. A backslash character (\) in a comment is treated as a comment character. It is not recognized as signaling continuation.
- Every directive begins with the pound character (#), and the pound character (#) must be in column 1.
- Blank and tab (HT) characters can appear between the pound character (#) and the directive keyword.
- You cannot write form feed (FF) or vertical tab (VT) characters to separate tokens on a directive line. That is, if a source preprocessing line spans lines, it must be continued by using a backslash character (\).

- Blanks are significant, so the use of spaces within a source preprocessing directive is independent of the source form of the file. The fields of a source preprocessing directive must be separated by blank or tab (HT) characters.
- Because source preprocessing directives are independent of source form, a directive can be up to 132 columns on a single source line.

  Any directive text that extends past column 132 is ignored. The directive text is truncated, which is likely to produce parsing errors or unexpected results. If a directive is too long to fit on a single line, you can continue the line by using the backslash character (\). It cannot be continued using standard Fortran 90 continuation methods.
- Any user-specified identifier that is used in a directive must follow Fortran 90 rules for identifier formation. The exception to this rule is that the first character in the name can be an underscore character (_).
- Source preprocessing identifier names are case sensitive.
- Numeric literal constants must be integer literal constants or real literal constants, as defined for Fortran 90.
- Comments written in the style of the C language, beginning with `/*` and ending with `*/`, can appear anywhere within a source preprocessing directive in which blanks or tabs can appear. The comment, however, must begin and end on a single source line.
- The blanks shown in the syntax descriptions of the source preprocessing directives are significant. The tab character (HT) can be used in place of a blank. Multiple blanks can appear wherever a single blank appears in a syntax description.

7.2 Directives

The following sections describe the source preprocessing directives.

7.2.1 `#include` directive

The `#include` directive directs the system to use the content of a file or directory. Just as with the `INCLUDE` line processing defined by the Fortran 90 standard, an `#include` directive effectively replaces that directive line with the content of *filename*. This directive has the following formats:

```
#include "filename"

#include <filename>
```

filename A file or directory to be used.

In the first form, if *filename* does not begin with a slash (/) character, the system searches for the named file, first in the directory of the file containing the #include directive, then in the sequence of directories specified by the -I option(s) on the f90(1) command line, and then the standard (default) sequence. If *filename* begins with a slash (/) character, it is used as is and is assumed to be the full path to the file.

The second form directs the search to begin in the sequence of directories specified by the -I option(s) on the f90(1) command line and then search the standard (default) sequence.

The Fortran 90 standard prohibits recursion in INCLUDE files, so recursion is also prohibited in the #include form.

The #include directives can be nested.

When the compiler is invoked to do only source preprocessing, not compilation, text will be included by #include directives but not by Fortran 90 INCLUDE lines. For information on the source preprocessing command line options, see Section 7.4, page 155.

7.2.2 #define directive

The #define directive lets you declare a source preprocessing variable and associate a token string with the variable. It also allows you to define a function-like macro. This directive has the following formats:

```
#define identifier value

#define identifier(dummy_arg_list) value
```

The first format defines an object-like macro (also called a *source preprocessing variable*), and the second defines a function-like macro. In the second format, the left parenthesis that begins the *dummy_arg_list* must immediately follow the identifier, with no intervening white space.

identifier — Specifies the name of the variable or macro being defined.

dummy_arg_list — Specifies a list of dummy argument identifiers.

value — Specifies the *value* as a sequence of tokens. The *value* can be continued onto more than one line using backslash (\) characters.

If a preprocessor *identifier* appears in a subsequent `#define` directive without being the subject of an intervening `#undef` directive, and the *value* in the second `#define` directive is different from the value in the first `#define` directive, then the preprocessor issues a warning message about the redefinition. For more information on the `#undef` directive, see Section 7.2.3, page 151.

When an object-like macro's identifier is encountered as a token in the source file, it is replaced with the value specified in the macro's definition. This is referred to as an *invocation* of the macro. By default, tokens are not processed in Fortran source code. They are recognized only when used in other source preprocessing directives.

The invocation of a function-like macro is more complicated. It consists of the macro's identifier, immediately followed by a left parenthesis with no intervening white space, then a list of actual arguments separated by commas, and finally a terminating right parenthesis. There must be the same number of actual arguments in the invocation as there are dummy arguments in the `#define` directive. Each actual argument must be balanced in terms of any internal parentheses. The invocation is replaced with the value given in the macro's definition, with each occurrence of any dummy argument in the definition replaced with the corresponding actual argument in the invocation.

The following two examples must be compiled with `-macro_expand` specified on the `f90`(1) command line.

- The following program prints `Hello, world.` when compiled and run:

```
      PROGRAM P
#define GREETING 'Hello, world.'
      PRINT *, GREETING
      END PROGRAM P
```

- The following program prints `Hello, Hello, world.` when compiled and run:

```
      PROGRAM P
#define GREETING(str1, str2) str1, str1, str2
```

```
PRINT *, GREETING('Hello, ', 'world.')
END PROGRAM P
```

7.2.3 #undef directive

The #undef directive sets the definition state of *identifier* to an undefined value. If *identifier* is not currently defined, the #undef directive has no effect. This directive has the following format:

```
#undef identifier
```

identifier Specifies the name of the source preprocessing variable or macro being undefined.

7.2.4 # (null) directive

The null directive simply consists of the pound character (#) in column 1 with no significant characters following it. That is, the remainder of the line is typically blank or is a source preprocessing comment. This directive is generally used for spacing out other directive lines.

7.2.5 Conditional directives

Conditional directives cause lines of code to either be produced by the source preprocessor or to be skipped. The conditional directives within a source file form *if-groups*. An if-group begins with an #if, #ifdef, or #ifndef directive, followed by lines of source code that you may or may not want skipped. Several similarities exist between the Fortran 90 IF construct and if-groups:

- The #elif directive corresponds to the ELSE IF statement.
- The #else directive corresponds to the ELSE statement.
- Just as an IF construct must be terminated with an END IF statement, an if-group must be terminated with an #endif directive.
- Just as with an IF construct, any of the blocks of source statements in an if-group can be empty.

  For example, you can write the following directives:

  ```
  #if MIN_VALUE == 1
  #else
    ...
  ```

```
#endif
```

Determining which group of source lines (if any) to compile in an if-group is essentially the same as the Fortran 90 determination of which block of an `IF` construct should be executed.

7.2.5.1 `#if` directive

The `#if` directive has the following format:

`#if` *expression*

expression An expression. The values in *expression* must be integer literal constants or previously defined preprocessor variables. The expression is an integer constant expression as defined by the C language standard. All the operators in the expression are C operators, not Fortran 90 operators. The *expression* is evaluated according to C language rules, not Fortran 90 expression evaluation rules.

Note that unlike the Fortran 90 `IF` construct and `IF` statement logical expressions, *expression* in an `#if` directive need not be enclosed in parentheses.

The `#if` expression can also contain the unary `defined` operator, which can be used in either of the following formats:

`defined` *identifier*

`defined(`*identifier*`)`

When the `defined` subexpression is evaluated, the value is 1 if *identifier* is currently defined, and 0 if it is not.

All currently defined source preprocessing variables in *expression*, except those that are operands of `defined` unary operators, are replaced with their values. During this evaluation, all source preprocessing variables that are undefined evaluate to 0.

Note that the following two directive forms are **not** equivalent:

- `#if X`

- `#if defined(X)`

In the first case, the condition is true if `X` has a nonzero value. In the second case, the condition is true only if `X` has been defined (has been given a value that could be 0).

7.2.5.2 `#ifdef` directive

The `#ifdef` directive is used to determine if *identifier* is predefined by the source preprocessor, has been named in a `#define` directive, or has been named in a `f90 -D` command line option. For more information on the `-D` option, see Section 7.4, page 155. This directive has the following format:

```
#ifdef identifier
```

The `#ifdef` directive is equivalent to either of the following two directives:

- `#if defined` *identifier*
- `#if defined(`*identifier*`)`

7.2.5.3 `#ifndef` directive

The `#ifndef` directive tests for the presence of an *identifier* that is not defined. This directive has the following format:

```
#ifndef identifier
```

This directive is equivalent to either of the following two directives:

- `#if ! defined` *identifier*
- `#if ! defined(`*identifier*`)`

7.2.5.4 `#elif` directive

The `#elif` directive serves the same purpose in an if-group as does the `ELSE IF` statement of a Fortran 90 `IF` construct. This directive has the following format:

```
#elif expression
```

expression The expression follows all the rules of the integer constant expression in an `#if` directive.

7.2.5.5 `#else` directive

The `#else` directive serves the same purpose in an if-group as does the `ELSE` statement of a Fortran 90 `IF` construct. This directive has the following format:

```
#else
```

7.2.5.6 `#endif` directive

The `#endif` directive serves the same purpose in an if-group as does the `END IF` statement of a Fortran 90 `IF` construct. This directive has the following format:

```
#endif
```

7.3 Predefined macros

The MIPSpro 7 Fortran 90 source preprocessor supports a number of predefined macros. They are divided into groups as follows:

- Macros that are based on the host machine
- Macros that are based on IRIX system targets

The following predefined macros are based on the host system (the system upon which the compilation is being done):

Macro	Notes
`unix`, `__unix`	Always defined. (The leading characters in the second form consist of 2 consecutive underscores.)

The following predefined macros are based on an IRIX system target:

Macro	Notes
D_LANGUAGE_FORTRAN90, DLANGUAGE_FORTRAN90	
host_mips, __host_mips	(The leading characters in the second form consist of 2 consecutive underscores.)
LANGUAGE_FORTRAN, _LANGUAGE_FORTRAN	
MIPSEB, _MIPSEB	
mips	
sgi, __sgi	(The leading characters in the second form consist of 2 consecutive underscores.)
_SGI_SOURCE	
_SVR4_SOURCE	
_SYSTYPE_SVR4	

7.4 Command line options

Several `f90`(1) command line options affect source preprocessing. For more information on these options, see Chapter 2, page 5. They are as follows:

- The `-cpp` option
- The `-D` *identifier*[=*value*] [, *identifier*[=*value*]] ... option
- The `-E` option
- The `-F` option
- The `-ftpp` option
- The `-U` *identifier* [, *identifier*] ... option

The `-D` *identifier*[=*value*] [, *identifier*[=*value*]] ..., `-F`, and `-U` *identifier* [, *identifier*] ... options are ignored unless the Fortran input source file is specified as either *file*`.F` or *file*`.F90`.

Interlanguage Calling [8]

You may want to call external procedures written in C, C++, or some other language, or you may need to call a Fortran 90 procedure from one of those languages. This chapter focuses on the interface between Fortran 90 and C/C++.

If your application has source programs written in different languages, you should compile each file separately, with the appropriate compiler, and then load them in a separate step. You can create object files suitable for loading by specifying the `-c` option on the `f90`(1) command, which disables the load step and writes the binary file to ***file***`.o`.

In the following example, the C/C++ compiler and the MIPSpro 7 Fortran 90 compilers produce object files that can be loaded. These files are named `main.o` and `rest.o`:

```
%   cc -c main.c
%   f90 -c rest.f
```

This chapter provides more details on compiling and loading application programs that are written in Fortran 90, C, and C++.

8.1 External and public names

When your Fortran 90 program defines the body of a procedure, the compiler places the name of the procedure, as a character string, in the object file it generates. This is a ***public name***, which is accessible to other object files.

When your Fortran 90 program uses a procedure, the compiler places the name of the procedure in the generated object file. This is an ***external name***, which is used by the object file but not defined in it. Names of common blocks and names of data and procedures declared within object files are also external names. You can use the `nm`(1) utility to display the public and external names defined in a file. For more information on this utility, see the *MIPS Compiling and Performance Tuning Guide*.

It is up to the IRIX loader, `ld`(1), to resolve each reference to an external name by finding that same name as a public name in some other module. This is the main job of the loader.

8.1.1 How Fortran 90 handles external and public names

The Fortran 90 compiler converts input source text (other than the contents of character literals) to uppercase as the first step of compilation. As a result, it may change the case of the names of procedures and named common blocks while it translates the source file. The names recorded in the object file, these names are changed in the following two ways from the way you may have written them:

- They are converted to all uppercase letters.
- They are normally extended with a final underscore (_) character.

Procedure names and common block names are translated, too.

The following declarations produce the identifiers `matrix_`, `mixedcase_`, and `cblk_` in the object file:

```
SUBROUTINE MATRIX
external function MixedCase()
COMMON /CBLK/a,b,c
```

These changes cause no problems when loading program units compiled by Fortran 90 or FORTRAN 77. The same convention is used for both the public and external names, so the names match.

> **Note:** Some IRIX-based FORTRAN 77 compilers support the `-U` command line option, which prevents the compiler from forcing all uppercase input to lowercase. As a byproduct, it becomes possible to put mixed-case public names in the object file. This option is not supported by the MIPSpro 7 Fortran 90 compiler.
>
> In addition, some IRIX-based FORTRAN 77 compilers take the use of the `$` character as the final letter of a procedure name as a signal to suppress the underscore in the public name. The `$` is not permitted to appear in a name if the program is to be compiled by the MIPSpro 7 Fortran 90 compiler. There is no way to suppress the final underscore in an external name.

8.1.2 Calling a Fortran 90 subprogram from C

To call a Fortran 90 subprogram from a C procedure, spell the name the way the Fortran 90 compiler spells it, using all uppercase letters and a trailing underscore.

For example, consider the following Fortran 90 declaration:

```
SUBROUTINE HYPOT()
```

This must be declared in a C function as follows (note the use of uppercase with a trailing underscore):

```
extern int HYPOT_()
```

8.1.3 Calling a C function from Fortran 90

The public name of a C function can be in uppercase or mixed case, and they have terminal underscores only when you write them that way. To call a C function from a Fortran 90 program, ensure that the C function's name is spelled the way the Fortran 90 compiler expects it to be. When you control the name of the C function, the simplest solution is to give it a name that consists of uppercase letters with a terminal underscore. For example, the following C function:

```
int FROMFORT_() {...}
```

could be declared in a Fortran 90 program as follows:

```
external FROMFORT
```

When you do not control the name of a C function, you must supply a function name that Fortran 90 can call. The only solution is to write a C function that takes the same arguments, but that has a name composed of uppercase letters and ending in an underscore. This C function can then call the function whose name contains mixed-case letters. You can write such a wrapper function manually, or you can use the `mkf2c`(1) utility to do it automatically. For more information on using `mkf2c`(1), see Section 8.6, page 172.

8.2 Correspondence of Fortran 90 and C data types

When you exchange data between Fortran 90 and C, either as arguments, as function results, or as members of common blocks, you must make sure that the two languages agree on the size, alignment, and subscript of each data object.

8.2.1 Corresponding scalar types

The correspondence between Fortran 90 and C scalar data types is shown in Table 3. This table assumes that the default command line options that affect precision are in effect. Use of compiler options such as `-i2` or `-r8` affects storage sizes for integer, logical, real, and double precision data types.

Table 3. Corresponding Fortran 90 and C Data Types

Fortran Data Type Declaration	C Data Type
INTEGER(KIND=1), LOGICAL(KIND=1)	signed char
CHARACTER	unsigned char
INTEGER(KIND=2), LOGICAL(KIND=2)	short
INTEGER, INTEGER(KIND=4), LOGICAL, LOGICAL(KIND=4)	int
INTEGER(KIND=8), LOGICAL(KIND=8)	long long
REAL, REAL(KIND=4)	float
DOUBLE PRECISION, REAL(KIND=8)	double
REAL(KIND=16)	long double
COMPLEX, COMPLEX(KIND=4)	typedef struct{float real, imag; } cpxk4;
DOUBLE COMPLEX, COMPLEX(KIND=8)	typedef struct{double real, imag;} cpxk8;
COMPLEX(KIND=16)	typedef struct{long double re, im;} cpxk16;
CHARACTER(***n***)	typedef char fstr_***n***[***n***]

For type character, Fortran 90 declarations with a length designator, such as `CHARACTER(LEN=N) :: X`, are equivalent to a C declaration of `unsigned char X[N]`.

To set a NULL character in a Fortran string, use `CHAR(0)`. Examples:

```
character*4 aaa
aaa(1:3) = 'abc'
aaa(4:4) = CHAR(0)
```

8.2.2 Corresponding character types

The Fortran 90 `CHARACTER` data type declaration corresponds to the C type `unsigned char`. However, the two languages differ in the treatment of strings of characters.

A Fortran 90 variable can be declared as `CHARACTER(`*n*`)`, where *n*>1, contains exactly *n* characters at all times. When a shorter character expression is assigned to it, it is padded on the right with spaces to reach *n* characters.

A C vector of characters is normally sized 1 greater than the longest string assigned to it. It may contain fewer meaningful characters than its size allows, and the end of meaningful data is marked by a null byte. There is no null byte at the end of a Fortran 90 string (except by chance memory alignment).

There is no terminal null byte, so most of the string library functions familiar to C programmers (`strcpy()`(3C), `strcat()`(3C), `strcmp()`(3C), and so on) cannot be used with Fortran 90 string values. The `strncpy()`(3C), `strncmp()`(3C), `bcopy()`(3C), and `bcmp()`(3C) functions can be used because they depend on a count rather than a delimiter.

8.2.3 Corresponding array elements

Fortran 90 and C use different arrangements for the elements of an array in memory. Fortran 90 uses column-major order (when iterating sequentially through memory, the leftmost subscript varies fastest), whereas C uses row-major order (the rightmost subscript varies fastest to generate sequential storage locations). In addition, Fortran 90 array indexes are by default origin-1, and can be declared as any origin, while C indexes are always origin-0.

To use a Fortran 90 array in C, perform the following steps:

1. Reverse the order of dimension limits when declaring the array.
2. Reverse the sequence of subscript variables in a subscript expression.
3. Adjust the subscripts to origin-0 (usually, decrement by 1).

The correspondence between Fortran 90 and C subscript values is depicted in Figure 9. You derive the C subscripts for a given element by decrementing the Fortran 90 subscripts and using them in reverse order; for example, Fortran 90 (99,9) corresponds to C [8][98].

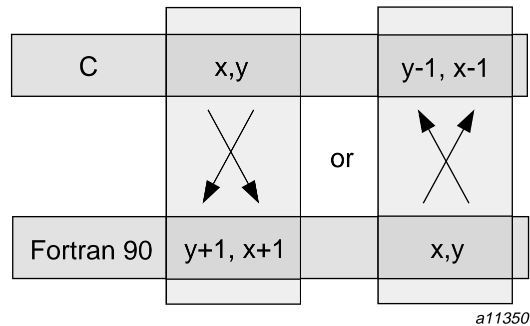


Figure 9. Correspondence between C and Fortran 90 subscripts

Note: A Fortran 90 array can be declared with some other lower bound than the default of 1. If the Fortran 90 subscript is origin-0, no adjustment is needed. If the Fortran 90 lower bound is greater than 1, the C subscript is adjusted by that amount.

8.2.4 Unsupported array arguments

Fortran 90 supports assumed-shape arrays, deferred-shape arrays, and array sections. You cannot pass any of these types of array to a non-Fortran 90 procedure because Fortran 90 represents such arrays in memory using a descriptor containing indirect pointers and other data. The format of this descriptor is not part of the published programming interface to MIPSpro 7 Fortran 90, as it is subject to change.

If you attempt to pass an assumed-shape array, deferred-shape array, or an array section to a non-Fortran 90 function, the function does not receive the address of array elements in memory as it would when an array is passed. Instead it receives the address of a descriptor with undocumented contents, resulting in unpredictable behavior.

8.3 How Fortran 90 passes arguments

When calling non-Fortran 90 functions, you must know how arguments are passed. When calling Fortran 90 subprograms from other languages, you must cause the other language to pass arguments correctly.

Note: You should be aware that all compilers for a given version of IRIX use identical conventions for passing arguments. These conventions are documented at the machine instruction level in the *MIPSpro Assembly Language Programmer's Guide*, which also describes the differences in the conventions used in different releases.

An argument passed to a subprogram, regardless of its data type, is passed as the address of the actual in memory. This rule is extended for two special cases:

- The length of each `CHARACTER(`*n*`)` declaration is passed as an implicit additional `INTEGER(KIND=4)` value, following the explicit arguments.
- When a function returns type `CHARACTER(`*n*`)` the address of the space to receive the result is passed as the first argument to the function, and the length of the result space is passed as the second implicit argument, preceding all explicit arguments.

Example 1. Consider the following code:

```
COMPLEX(KIND=8) :: CP8
CHARACTER*(16) :: CSTR1, CSTR2
EXTERNAL CPXASC
CALL CPXASC(CSTR1,CSTR2,CP8)
```

The code generated from the subroutine call in this example passes the following arguments:

- The address of `CSTR1`
- The address of `CSTR2`
- The address of `CP8`
- The length of `CSTR1`, an integer value of 16
- The length of `CSTR2`, an integer value of 16

Example 2. Consider the following code:

```
CHARACTER*(8) :: SYMBL,PICKSYM
CHARACTER*(100) :: SENTENCE
INTEGER NSYM
SYMBL = PICKSYM(SENTENCE,NSYM)
```

The code generated from the function call in the preceding example passes the following arguments:

- The address of `SYMBL`, the result variable.

- The length of SYMBL, an integer value of 8
- The address of SENTENCE, the first explicit argument
- The address of NSYM, the second explicit argument
- The length of SENTENCE, an integer value of 100

8.4 Calling Fortran 90 from C

There are two types of callable Fortran 90 subprograms: subroutines and functions. In C terminology, both types of subprograms are external functions. The difference is the use of the function return value from each.

8.4.1 Calling a Fortran 90 subroutine from C

From the standpoint of a C module, a Fortran 90 subroutine is an external function returning int. The integer return value is normally ignored by a C caller (it is the alternate return statement number, if any).

Example 1. The following example shows a simple Fortran 90 subroutine that adds arrays of complex numbers:

```
SUBROUTINE ADDC32(Z, A, B, N)
COMPLEX(KIND=16) Z(1),A(1),B(1)
INTEGER :: N, I
DO 10 I = 1, N
      Z(I) = A(I) + B(I)
END DO
RETURN
END SUBROUTINE ADDC32
```

The Fortran 90 subroutine could be called from C using the following code fragment:

```
typedef struct{long double real, imag;} cpx32;
extern void
   ADDC32_(cpx32 *,cpx32 *,cpx32 *,int *);
cpx32 z[MAXARRAY], a[MAXARRAY], b[MAXARRAY];
...
      int n = MAXARRAY;
      addc32_(&z, &a, &b, &n);
```

The preceding code fragments show how the Fortran 90 subroutine is named in the C code using uppercase letters and a terminal underscore. This is the way the Fortran 90 compiler spells the public name in the object file.

Example 2. The following subroutine takes assumed-length character arguments:

```
SUBROUTINE PRT(BEF, VAL, AFT)
CHARACTER*(*) :: BEF, AFT
REAL :: VAL
PRINT *, BEF, VAL, AFT
RETURN
END SUBROUTINE PRT
```

The following C code prepares CHARACTER(16) values and passes them to the Fortran 90 subroutine:

```
typedef char fstr_16[16];
extern int
     prt_(fstr_16 *, float *, fstr_16 *,
                  int, int);
main()
{
      float val = 2.1828e0;
      fstr_16 bef,aft;
      strncpy(bef,"Before..........",sizeof(bef));
      strncpy(aft,"..........After",sizeof(aft));
      (void)PRT_(bef, &val, aft, sizeof(bef), sizeof(aft));
}
```

Note that the subroutine call requires five actual arguments: the addresses of the three explicit arguments and the lengths of the two string arguments. In the C code, the string length arguments are generated using sizeof(), which returns the memory size of the typedef fstr_16.

When the Fortran 90 code does not require a specific string length, the C code that calls it can pass an ordinary C character vector, as shown in the following code fragment:

```
extern int
prt_(char *, float *, char *, int, int);
main()
{
          float val = 2.1828e0;
          char *bef = "Start:";
```

```
          char *aft = ":End";
          (void)PRT_(bef, &val, aft, strlen(bef), strlen(aft));
}
```

In this example, the string length implicit argument values are calculated dynamically using `strlen()`.

8.4.2 Calling a Fortran 90 function from C

A Fortran 90 function that returns a scalar value as its result corresponds exactly to the C concept of a function with an explicit return value. When a Fortran 90 function returns any type shown in Table 3, page 160, other than `CHARACTER(`*n*`)`, where (*n*>1), you can call the function from C and handle its return value exactly as if it were a C function returning that data type.

Example 1. The following function accepts and returns `COMPLEX(KIND=8)` values.

```
FUNCTION FSUB8(INP)
COMPLEX(KIND=8) :: INP
FSUB8 = INP
END FUNCTION FSUB8
```

Although a complex value is declared as a structure in C, it can be used as the return type of a function. The following C code shows how the preceding Fortran 90 function is declared and called:

```
typedef  struct{ double real, imag; } cpx8;
extern cpx8 FSUB8_(cpx8 *);
main()
{
      cpx8 inp = { -3.333, -5.555 };
      cpx8 oup = { 0.0, 0.0 };
      printf("testing fsub8...");
      oup = FSUB8_( &inp );
      if ( inp.real == oup.real && inp.imag == oup.imag )
          printf("Ok\n");
      else
          printf("Nope\n");
}
```

The arguments to a function, like the arguments to a subroutine, are passed as pointers, but the value returned is a value, not a pointer to a value.

Example 2. The following function has a `CHARACTER(16)` return value.

```
FUNCTION FS16(J, K, S)
   CHARACTER*(16) :: FS16, S
   INTEGER J, K
   FS16 = S(J:K)
RETURN
END FUNCTION FS16
```

When a Fortran 90 function returns `CHARACTER(`*n*` )`, where *n*>1, value, the returned value is not the explicit result of the function. Instead, you must pass the address and length of the result area as the first two arguments of the function, preceding the explicit arguments. This is demonstrated in the following C code:

```
typedef char fstr_16[16];
extern void
fs16_ (fstr_16 *, int, int *, int *, fstr_16 *, int);
main()
{
      char work[64];
      fstr_16 inp, oup;
      int j = 7;
      int k = 11;
      strncpy(inp,"0123456789abcdef", sizeof(inp));
      fs16_ ( oup, sizeof(oup), &j, &k, inp, sizeof(inp) );
      strncpy(work, oup, sizeof(oup));
      work[sizeof(oup)] = '\0';
      printf("FS16 returns <%s>\n", work);
}
```

In this example, the address and length of the function result are the first two arguments of the function. Because type `fstr_16` is an array, its name, `oup`, evaluates to the address of its first element. The next three arguments are the addresses of the three named arguments. The final argument is the length of the string argument.

8.5 Calling C from Fortran 90

You can call units of C code from Fortran 90 as if they were written in Fortran 90, provided that the C modules follow the Fortran 90 conventions for passing arguments. For more information on this, see Section 8.3, page 162.

When the C function expects arguments passed using other conventions, you normally need to build a wrapper for the C function using the mkf2c(1) command.

8.5.1 Calls to C functions

The following C function is written to use the Fortran 90 conventions for its name (uppercase with final underscore) and for argument passing:

```
/*
|| C functions to export the facilities of strtoll()
|| to Fortran 90 programs.  Effective Fortran declaration:
||
|| FUNCTION ISCAN(S,J)
|| INTEGER(KIND=8) :: ISCAN
|| CHARACTER*(*) S
|| INTEGER J
||
|| String S(J:) is scanned for the next signed long value
|| as specified by strtoll(3c) for a "base" argument of 0
|| (meaning that octal and hex literals are accepted).
||
|| The converted long long is the function value, and J is
|| updated to the nonspace character following the last
|| converted character, or to 1+LEN(S).
||
|| Note: if this routine is called when S(J:J) is neither
|| whitespace nor the initial of a valid numeric literal,
|| it returns 0 and does not advance J.
*/
#include <ctype.h> /* for isspace() */
long long iscan_(char *ps, int *pj, int ls)
{
   int  scanPos, scanLen;
   long long ret = 0;
   char wrk[1024];
   char *endpt;
   /* when J>LEN(S), do nothing, return 0 */
   if (ls >= *pj)
   {
      /* convert J to origin-0, permit J=0 */
      scanPos = (0 < *pj)? *pj-1 : 0 ;
```

```
        /* calculate effective length of S(J:) */
        scanLen = ls - scanPos;

        /* copy S(J:) and append a null for strtoll() */
        strncpy(wrk,(ps+scanPos),scanLen);
        wrk[scanLen] = '\0';

        /* scan for the integer */
        ret = strtoll(wrk, &endpt, 0);

        /*
        || Advance over any whitespace following the number.
        || Trailing spaces are common at the end of Fortran
        || fixed-length char vars.
        */
        while(isspace(*endpt)) { ++endpt; }
        *pj = (endpt - wrk)+scanPos+1;
    }
    return ret;
}
```

The following Fortran 90 code fragment demonstrates a call to the preceding C function:

```
EXTERNAL ISCAN
INTEGER(KIND=8) ISCAN
INTEGER(KIND=8) RET
INTEGER J,K
CHARACTER*(50) INP
INP = '1  -99   3141592  0xfff  033 '
J = 0
DO WHILE (J .LT. LEN(INP))
      K = J
      RET = ISCAN(INP,J)
      PRINT *, K,': ',RET,' -->',J
END DO
END
```

8.5.2 Using Fortran 90 common blocks in C code

A C function can refer to the contents of a common block defined in a Fortran 90 program. The name of the block as given in the COMMON statement is altered as described in Section 8.1.1, page 158. (The name is converted to

uppercase and extended with an underscore). The name of the blank common is _BLNK__, with one leading underscore and two trailing ones.

To refer to the contents of a common block, take these steps:

1. Declare a C structure with fields that have the appropriate data types to match the successive elements of the Fortran 90 common block. For information on corresponding data types, see Table 3, page 160.

2. Declare the common block name as an external structure of that type.

The following example employs this method:

```
      INTEGER STKTOP, STKLEN, STACK(100)
      COMMON /WITHC/ STKTOP, STKLEN, STACK

struct fstack {
      int stktop, stklen;
      int stack[100];
}
extern fstack WITHC_;
int peektop_()
{
      if (withc_.stktop) /* stack not empty */
          return WITHC_.stack[WITHC_.stktop-1];
      else...
}
```

The restrictions on this capability are as follows:

- You cannot map a common block that contains Fortran 90 pointer-based variables.
- If the common block contains a variable of Fortran 90 derived type (a structure), ensure that the derived type is declared with the SEQUENCE attribute. Otherwise, its fields may not appear in the expected sequence in memory.
- When -O3 is in effect, the compiler may split up common blocks.

8.5.3 Using Fortran 90 arrays in C code

As described in Section 8.2.3, page 161, a C program must take special steps to access arrays created in Fortran 90. The following examples illustrate this.

Example 1. The following Fortran 90 code fragment declares a matrix in a common block and then calls a C subroutine to modify the array:

```
INTEGER IMAT(10,100), R, C
COMMON /WITHC/ IMAT
R = 74
C = 6
CALL CSUB(C, R, 746)
PRINT *, IMAT(6,74)
END
```

Example 2. The following C function stores its third argument in the common array using the subscripts passed in the first two arguments. In the C function, the order of the dimensions of the array are reversed, so the subscript values are reversed to match, and decremented by 1 to provide 0-origin indexing:

```
extern struct { int imat[100][10]; } WITHC_;
int csub_(int *, int *, int *)
{
      WITHC_.imat[*pr-1][*pc-1] = *pval;
      return 0; /* All Fortran subroutines return int */
}
```

8.5.4 Calls to C using %LOC and %VAL

You can use the nonstandard intrinsic functions %VAL and %LOC to pass arguments in ways other than the standard Fortran 90 conventions described in Section 8.3, page 162.

8.5.4.1 Using %VAL

The %VAL function is used in an argument list to cause an argument to be passed by value rather than by reference. Suppose that you need to call a C function having the following prototype:

```
int TAKESINT_ (int, char *, int)
```

The first argument to this function is an integer value, not the address of an integer value in memory. You could call this function from Fortran 90 code similar to the following:

```
CHARACTER(80) SENTENCE
INTEGER(4) J
CALL TAKESINT(%VAL(J), SENTENCE)
```

The use of %VAL(*j*) causes the contents of *j* to be passed, rather than the address of *j*.

8.5.4.2 Using %LOC

The %LOC function returns the address of its argument. It can be used with %VAL to prevent passing the length of a character value as a hidden argument. In other words, the argument %VAL(LOC(*char_var*)) passes only the address of *char_var*. It does not pass the implicit length argument.

8.6 Making C wrappers with mkf2c(1)

The mkf2c(1) utility provides an alternate interface for C routines called by Fortran 90.

The mkf2c(1) utility reads a file of C function prototype declarations and generates an assembly language module. This module contains one callable entry point for each C function. The entry point, or wrapper, accepts arguments in the Fortran 90 calling convention and passes the same values to the C function using the C conventions.

The following shows a simple case of using a function as input to mkf2c(1):

```
simplefunc (int a, double df)
{ /* function body ignored */ }
```

For this function, using mkf2c(1), with no options, generates a wrapper function named simple_. This is truncated to 6 characters, made lowercase, and appended with an underscore. The wrapper function expects two arguments: one must be a default integer and one must be a REAL(KIND=8) value. These must be passed according to Fortran 90 conventions; that is, by reference. The code of the wrapper loads the values of the arguments into registers using C conventions for passing arguments by value, and calls simplefunc().

8.6.1 mkf2c(1) argument assumptions

The mkf2c(1) utility processes only the C source, not the Fortran 90 source, so it treats the Fortran 90 arguments based on the data types specified in the C function header. These treatments are summarized in Table 4, page 173.

Table 4. How mkf2c(1) treats function arguments

Data type in C prototype	Treatment by generated wrapper code
unsigned char	Load CHARACTER from memory to register, no sign extension.
char	Load CHARACTER from memory to register; sign extension only when -signed is specified.
unsigned short, unsigned int	Load INTEGER(KIND=2) or INTEGER(KIND=4) from memory to register, no sign extension.
short	Load INTEGER(KIND=2) from memory to register with sign extension.
int, long	Load INTEGER(KIND=4) from memory to register with sign extension.
long long	Load INTEGER(KIND=8) from memory to register with sign extension.
float	Load REAL(KIND=4) from memory to register, extending to double unless -f is specified.
double	Load REAL(KIND=8) from memory to register.
long double	Load REAL(KIND=16) from memory to register.
char *name*[], *name*[*n*]	Pass address of CHARACTER(*n*) and pass length as integer argument as Fortran does.
char *	Copy CHARACTER(*n*) value into allocated space, append null byte, pass address of copy.

8.6.2 mkf2c(1) character string treatment

In Table 4, page 173, notice the different treatments for an argument declared as a character array and one declared as a character address, even though these two declarations are semantically the same in C.

When the C function expects a character address, mkf2c(1) generates the code to dynamically allocate memory and to copy the Fortran 90 character value, for its specified length, to memory. This creates a null-terminated string and has the following other outcomes:

- The address passed to C points to allocated memory.
- The length of the value is not passed as an implicit argument.

- There is a terminating null byte in the value.
- Changes in the string are not reflected back to Fortran 90.

A character array is passed by mkf2c(1) as a Fortran 90 CHARACTER*(*n*) value. This has the following outcomes:

- The address prepared by Fortran 90 is passed to the C function.
- The length of the value is passed as an implicit argument (see Section 8.3, page 162).
- The character array contains no terminating null byte.
- Changes in the array by the C function are visible to Fortran 90.

The C function cannot declare the extra string-length argument because if it declares the argument, mkf2c processes it as an explicit argument. The two ways to access the string length are as follows:

- If the Fortran 90 program always passes character values of the same size, the length argument can simply be ignored.
- If its value is needed, the varargs macro can be used to retrieve it.

Suppose the C function prototype is specified as follows:

```
void func1 (char carr1[], int i, char *str, char carr2[]);
```

In this case, mkf2c(1) passes a total of six arguments to C. The fifth argument is the length of the Fortran 90 value corresponding to carr1. The sixth is the length of carr2. The C function can use the varargs macros to retrieve these hidden arguments. mkf2c(1) ignores the varargs macro va_alist that appears at the end of the argument name list.

When func1 is changed to use varargs, the C source file is as follows:

```
#include "varargs.h"
void
func1 (char carr1[], int i, char *str, char carr2[], va_alist);
{}
```

The C routine would retrieve the lengths of carr1 and carr2, placing them in the local variables carr1_len and carr2_len, using code like the following fragment:

```
va_list ap;
int carr1_len, carr2_len;
```

```
va_start(ap);
carr1_len = va_arg (ap, int)
carr2_len = va_arg (ap, int)
```

8.6.3 mkf2c(1) restrictions

When it does not recognize the data type specified in the C function, mkf2c(1) issues a warning message and generates code to simply pass the pointer set up by Fortran 90. It does this for the following cases:

- Any nonstandard data type name, for example a data type that might be declared using typedef or a data type defined as a macro
- Any structure argument
- Any argument with multiple indirection (two or more asterisks, for example *char***)

The mkf2c(1) utility does not support structure-valued arguments, so it does not support passing COMPLEX(KIND=*n*) values or derived types. Nor does mkf2c(1) have any means of passing assumed-shape or deferred-shape arrays.

8.6.4 Using mkf2c(1) and extcentry(1)

The mkf2c(1) utility accepts only a limited subset of the C grammar. This subset includes common C syntax for function entry point, C-style comments, and function bodies. However, it does not include constructs such as typedefs, external function declarations, or C preprocessor directives. You may receive unexpected results from mkf2c(1) if these types of constructs are provided as input.

To ensure that only the constructs understood by mkf2c(1) are included in wrapper input, you need to place special comments around each function for which Fortran 90-to-C wrappers are to be generated.

After the special comments, /* CENTRY */ and /* ENDCENTRY */, are placed around the code, you can use the extcentry(1) utility to generate the input file for mkf2c(1).

The following example uses extcentry(1). It shows C file foo.c containing function foo, which is to be made Fortran 90 callable:

```
typedef unsigned short grunt[4];
struct {
     long 1,11;
```

```
     char *str;
} bar;
main ()
{
     int kappa =7;
     foo (kappa,bar.str);
}
/* CENTRY */
foo (integer, cstring)
int integer;
char *cstring;
{
     if (integer==1) printf("%s",cstring);
} /* ENDCENTRY */
```

The special comments `/* CENTRY */` and `/* ENDCENTRY */` surround the section that is to be made Fortran 90 callable. To generate the assembly language wrapper, `foowrp.s`, from the file `foo.c`, use the following set of commands:

```
% extcentry foo.c foowrp.fc
% mkf2c foowrp.fc foowrp.s
```

8.7 Makefile considerations

The `make`(1) utility contains default rules to help automate wrapper generation. The following example of a makefile illustrates the use of these rules.

In the following sample makefile, an executable object file is created from files `main.f`, a Fortran 90 main program, and `callc.c`:

```
test:  main.o callc.o
     f90 -o test main.o callc.o
callc.o: callc.fc
clean:
     rm -f *.o test *.fc
```

In the preceding makefile, `main.f` calls a C routine in `callc.c`. The extension `.fc` has been adopted for Fortran 90-to-call-C wrapper source files. The wrappers created from `callc.fc` are assembled and combined with the binary created from `callc.c`. Also, the dependency of `callc.o` on `callc.fc` causes `callc.fc` to be recreated from `callc.c` whenever the C source file changes. You must put special comments for `extcentry`(1) in the C source code as required.

Note: Options to mkf2c(1) can be specified when make(1) is invoked by setting the make(1) variable F2CFLAGS.

Do not create a .fc file for the modules that need to have wrappers created. These files are both created and removed by make(1) in response to the file.o:file.fc dependency.

The makefile generates the wrappers and Fortran 90 objects. You can add modules to the executable object file in one of the following ways:

- If the file is a native C file with routines that are not to be called from Fortran 90 using a wrapper interface, or if it is a native Fortran 90 file, add the .o suffix to the final make(1) target and dependencies.
- If the file is a C file containing routines to be called from Fortran 90 using a wrapper interface, the comments for extcentry(1) must be placed in the C source code, and the .o file placed in the target list. In addition, the dependency of the .o file on the .fc file must be placed in the makefile. This dependency is illustrated in the example makefile above, where callc.o depends on callc.fc.

8.8 Calling assembly language from Fortran 90

You can write modules in MIPS assembly language, following the guidelines in the *MIPSpro Assembly Language Programmer's Guide*. Procedures in these modules can be called from Fortran 90. There is only one special consideration.

Operating in assembly language, you can change the operating mode and the rounding mode of the CPU. When running Fortran 90 programs that contain quad precision operations, you must run the compiler in round-to-nearest mode. This mode is in effect by default, so you usually do not need to set it. You usually need to set this mode when writing programs that call your own assembly routines. For more information, see swapRM(3C).

Library Usage [A]

The MIPSpro 7 Fortran 90 compiler can use many intrinsic procedures and library routines to aid in quicker compiling time. This chapter discusses many of the commonly used procedures and routines that can be used with the compiler.

A.1 The `assign` command

The `assign`(1) command can be used to alter the details of a Fortran file connection, such as device residency, alternative file names, or file space allocations. The `assign` options are associated with file names, file name patterns, or unit numbers. When associated with file names or file name patterns, the options are applied whenever a matching file name is opened from a Fortran program. When associated with a unit number, the options are applied whenever that unit becomes connected.

For complete details about the `assign` command, see the `assign`(1) man page or the Cray publication, the *Application Programmer's I/O Guide*, publication SG–2168.

A.1.1 Options to the `assign` command

The `assign` command has the following syntax:

```
assign [-a actualfile] [-b bs] [-f fortstd] [-s ft] [-t] [-y setting] [-B
    setting] [-D fildes] [-F spec [,specs]] [-I] [-O] [-R] [-S setting]
    [-T setting] [-U setting] [-V] [-W setting] [-Y setting] assign_object
```

The `assign -R` and `assign -V` commands cannot be used with any other options.

The following list describes the `assign` command options.

`-a` *actualfile*	Specifies the actual file or the `FILE=` specifier.
`-b` *bs*	Specifies ibrary buffer size in 4096–byte blocks.
`-f` *fortstd*	Specifies the Fortran standard.

Use `irixf90` for *fortstd* to be compatible with the MIPSpro 7 Fortran 90 compiler.

Use `irixf77`for *fortstd* to be compatible with Silicon Graphic's FORTRAN 77 compiling system which runs on IRIX systems.

Use `77` for *fortstd* to be compatible with the FORTRAN 77 standard and Cray Research's CF77 compiling system.

Use `90` for *fortstd* to be compatible with the Fortran 90 standard and Cray Research's CF90 compiling system.

`-s` *ft*
Specifies the ile type. Enter `text`, `cos`, `blocked`, `unblocked`, `u`, `sbin`, or `bin` for *ft*.

`-t`
Specifies temporary file use.

`-y` *setting*
Produces repeat counts in list-directed output. *setting* can be either `on` or `off`. The default setting on UNICOS and UNICOS/mk systems is `on`. The default setting on IRIX systems is `off`.

`-B` *setting*
Activates or suppresses the passing of the `O_DIRECT` flag to the `open`(2) system call. Enter either `on` or `off` for *setting*.

`-D` *fildes*
Specifies a connection to a standard file. Enter `stdin`, `stdout`, or `stderr` for *fildes*.

`-F` *spec* [*, specs*]
Specifies use of Flexible file I/O (FFIO). See the `assign`(1) man page for complete details about allowed values for *spec* and for details about hardware platform support. See the `INTRO_FFIO`(3F) man page for details about the individual FFIO layers.

See Section A.1.2 for a list of supported FFIO layers.

`-I`
Specifies an incremental assign. All attributes are added to the attributes already assigned to the current *assign_object*.

`-O`	Specifies a replacement assign (default control option). All currently existing `assign` attributes for the current *assign_object* are replaced.
`-R`	Specifies removal of all `assign` attributes for *assign_object*.
`-T` *setting*	Activates or suppresses truncation after write for sequential Fortran files. Enter either `on` or `off` for *setting*.
`-U` *setting*	Produces a UNICOS form of list-directed output. This is a global setting which sets the value for the `-y`, `-S`, and `-W` options. Enter either `on` or `off` for *setting*. The default setting on UNICOS and UNICOS/mk systems is `on`. The default setting on IRIX systems is `off`.
`-V`	Views attributes for *assign_object*.
`-W` *setting*	Produces compressed width in list-directed output. Enter either `on` or `off` for *setting*. The default setting on UNICOS and UNICOS/mk systems is `on`. The default setting on IRIX systems is `off`.
`-Y` *setting*	Skips unmatched namelist groups in a namelist input record. Enter either `on` or `off` for *setting*. The default setting on UNICOS and UNICOS/mk systems is `on`. The default setting on IRIX systems is `off`.
assign_object	Specifies either a file name or a unit number for *assign_object*. The `assign` command associates the attributes with the file or unit specified. These attributes are used during the processing of Fortran `OPEN` statements or during implicit file opens.

Use one of the following formats for *assign_object*:

- `f:`*file_name* (for example, `f:file1`)
- `g:`*io_type*; *io_type* can be `su`, `sf`, `du`, `df`, `ff`, or `aq` (for example, `g:ff`)
- `p:`*pattern* (for example, `p:file%`)
- `u:`*unit_number* (for example, `u:9`)

- *file_name* (for example, `myfile`)

A.1.2 Supported FFIO layers

The Flexible File I/O (FFIO) system lets the user specify a comma-separated list of layers through which I/O data will be passed. This is done by providing a value for the *spec* argument to the `-F` option on the `assign` command. The FFIO layers act as “filters” that are used to manipulate the data file as it is being read or written. The layers include performance options and the capability to read and write files in different vendors’ blocking formats.

The following layers are available for IRIX systems:

Layer	Definition
`bufa`	Library-managed asynchronous buffering
`cache`	`cache` layer
`cachea`	`cachea` layer
`cos`	COS blocking
`f77`	UNIX record blocking
`fd`	File descriptor
`null`	The `null` layer
`syscall`	System call I/O
`system`	Generic system layer
`text`	Newline separated record formats
`user` and `site`	User-supplied or site-supplied layer

A.1.3 FFIO and asynchronous I/O

The FFIO library on IRIX systems calls `aio_sgi_init` library call the first time the library issues an asynchronous I/O call. It passes the following parameters to `aio_sgi_init`:

```
aio_numusers=MAX(64,sysconf(_SC_NPROC_CONF))
aio_threads=5
aio_locks=3
```

If a program is using multiple threads and asynchronous I/O, it is important that the value in `aio_numusers` be at least as large as the number of sprocs or

pthreads that the application contains. See the aio_sgi_init man page on your IRIX system for more details.

Users can change these values by setting environment variables to the desired value or users can supersede the FFIO library's call to aio_sgi_init by calling it themselves, before the first I/O statement in their programs. See the Cray Research publication, the *Application Programmer's I/O Guide*, publication SG–2168 or the INTRO_FFIO(3F) man page for more details.

The following FFIO layers may issue asynchronous I/O calls on IRIX systems:

- cos: see the description of cos on the INTRO_FFIO(3F) man page for a description of the circumstances when the cos layer uses asynchronous I/O.
- cachea and bufa: users should assume that these layers may issue asynchronous I/O calls.
- system or syscall: these layers may issue asynchronous I/O calls if called from a BUFFER IN or BUFFER OUT Fortran statement, or if called from one of the listed layers.

A.2 Intrinsic procedures

Intrinsic procedures are predefined by the computer programming language. They are invoked in the same way that other procedures are invoked. The Fortran 90 standard defines intrinsic procedures, and the MIPSpro 7 Fortran 90 compiler includes other intrinsics as extensions to the standard.

For details about the available intrinsic procedures, see the Cray Research publications, the *Intrinsic Procedures Reference Manual*, publication SR–2138 and *Fortran Language Reference Manual, Volume 2*, publication SR–3903.

A.3 Library routines

A library is a collection of subprograms, usually grouped around a specific subject, such as input and output (I/O). You can call library routines explicitly in your program, or they can be called by the compiler.

The following list describes the library routines that are available with the MIPSpro 7 Fortran 90 compiler. See the individual man pages for more details.

- **FFIO routines:** C routines used with the FFIO layers

- fffcntl(3C)
- ffopen(3C)
- ffpos(3C)
- ffread(3C)
- ffseek(3C)

- **Interface routines:** job control routines that control program terminations or execute a shell command
  - ABORT(3F)
  - EXIT(3F)
  - ISHELL(3F)
- **I/O routines:** routines to control input and output
  - ASNCTL(3F)
  - ASNQFILE(3F)
  - ASSIGN(3F)
  - FLUSH(3F)
  - NUMBLKS(3F)
  - RNL(3F)
  - RNLECHO(3F)
  - RNLSKIP(3F)
  - RNLTYPE(3F)
  - WNL(3F)
  - WNLLINE(3F)
  - WNLLONG(3F)
- **Programming aids:** routines for time and dates, packing and unpacking, and character argument counters
  - SECOND(3F)
  - SECONDR(3F)

- SYSCLOCK(3F)
- TIMEF(3F)

- **POSIX routines:** routines to access constructs not directly accessible with the Fortran standard
  - IPXFARGC(3F)
  - PXFACCESS(3F)
  - PXFCHDIR(3F)
  - PXFCHMOD(3F)
  - PXFCHOWN(3F)
  - PXFCHROOT(3F)
  - PXFCLEARENV(3F)
  - PXFCONST(3F)
  - PXFCREAT(3F)
  - PXFCTERMID(3F)
  - PXFDIRECTORY(3F)
  - PXFESTRGET(3F)
  - PXFEXECV(3F)
  - PXFFCNTL(3F)
  - PXFFILENO(3F)
  - PXFFORK(3F)
  - PXFGETARG(3F)
  - PXFGETARG(3F)
  - PXFGETEGID(3F)
  - PXFGETENV(3F)
  - PXFGETEUID(3F)
  - PXFGETGID(3F)

- PXFGETGRGID(3F)
- PXFGETGRNAM(3F)
- PXFGETGROUPS(3F)
- PXFGETLOGIN(3F)
- PXFGETPGRP(3F)
- PXFGETPID(3F)
- PXFGETPPID(3F)
- PXFGETPWNAM(3F)
- PXFGETPWUID(3F)
- PXFGETUID(3F)
- PXFINTGET(3F)
- PXFINTSET(3F)
- PXFISATTY(3F)
- PXFISBLK(3F)
- PXFISCHR(3F)
- PXFISDIR(3F)
- PXFISFIFO(3F)
- PXFISREG(3F)
- PXFLINK(3F)
- PXFLOCALTIME(3F)
- PXFOPEN(3F)
- PXFRENAME(3F)
- PXFRMDIR(3F)
- PXFSETENV(3F)
- PXFSETGID(3F)
- PXFSETPGID(3F)

- PXFSETSID(3F)
- PXFSETUID(3F)
- PXFSTAT(3F)
- PXFSTRGET(3F)
- PXFSTRUCTCOPY(3F)
- PXFSTRSET(3F)
- PXFSTRUCTCREATE(3F)
- PXFSTRUCTFREE(3F)
- PXFSYSCONF(3F)
- PXFTIME(3F)
- PXFTIMES(3F)
- PXFUCOMPARE(3F)
- PXFUMASK(3F)
- PXFUNAME(3F)
- PXFUNLINK(3F)
- PXFUTIME(3F)
- PXFWAIT(3F)

• **Multiprocessing routines**: Fortran multiprocessing routines

- mp_block
- mp_blocktime
- mp_create
- mp_destroy
- mp_my_threadnum
- mp_numthreads
- mp_set_numthreads
- mp_setup

- `mp_unblock`
- `mp_setlock`
- `mp_suggested_numthreads`
- `mp_unsetlock`
- `mp_barrier`
- `mp_in_doacross_loop`
- `mp_set_slave_stacksize`

A.4 Library functions

The Fortran library routines provide an interface from Fortran programs to the IRIX system functions. System functions are facilities that are provided by the IRIX system kernel directly, as opposed to functions that are supplied by library code loaded with your program.

Table 5 summarizes the routines in the Fortran run-time library that can be used with the compiler. See the individual man pages for details about each routine.

Table 5. Summary of System Interface Library Routines

Function	Purpose
`abort`	abnormal termination
`access`	determine accessibility of a file
`acct`	enable/disable process accounting
`alarm`	execute a subroutine after a specified time
`barrier`	perform barrier operations
`blockproc`	block processes
`brk`	change data segment space allocation
`chdir`	change default directory
`chmod`	change mode of a file
`chown`	change owner

Function	Purpose
chroot	change root directory for a command
close	close a file descriptor
creat	create or rewrite a file
ctime	return system time
dtime	return elapsed execution time
dup	duplicate an open file descriptor
etime	return elapsed execution time
exit	terminate process with status
fcntl	file control
fdate	return date and time in an ASCII string
fgetc	get a character from a logical unit
fork	create a copy of this process
fputc	write a character to a Fortran logical unit
free_barrier	free barrier
fseek	reposition a file on a logical unit
fseek64	reposition a file on a logical unit for 64-bit architecture
fstat	get file status
ftell	reposition a file on a logical unit
ftell64	reposition a file on a logical unit for 64-bit architecture
gerror	get system error messages
getarg	return command line arguments
getc	get a character from a logical unit
getcwd	get pathname of current working directory
getdents	read directory entries
getegid	get effective group ID
gethostid	get unique identifier of current host
getenv	get value of environment variables

Function	Purpose
geteuid	get effective user ID
getgid	get user or group ID of the caller
gethostname	get current host ID
getlog	get user's login name
getpgrp	get process group ID
getpid	get process ID
getppid	get parent process ID
getsockopt	get options on sockets
getuid	get user or group ID of caller
gmtime	return system time
iargc	return command line arguments
idate	return date or time in numerical form
ierrno	get system error messages
ioctl	control device
isatty	determine if unit is associated with tty
itime	return date or time in numerical form
kill	send a signal to a process
link	make a link to an existing file
loc	return the address of an object
lseek	move read/write file pointer
lseek64	move read/write file pointer for 64-bit architecture
lstat	get file status
ltime	return system time
m_fork	create parallel processes
m_get_myid	get task ID
m_get_numprocs	get number of subtasks
m_kill_procs	kill process
m_lock	set global lock

Function	Purpose
m_next	return value of counter
m_park_procs	suspend child processes
m_rele_procs	resume child processes
m_set_procs	set number of subtasks
m_sync	synchronize all threads
m_unlock	unset a global lock
mkdir	make a directory
mknod	make a directory/file
mount	mount a filesystem
new_barrier	initialize a barrier structure
nice	lower priority of a process
open	open a file
oserror	get/set system error
pause	suspend process until signal
perror	get system error messages
pipe	create an interprocess channel
plock	lock process, test, or data in memory
prctl	control processes
profil	execution-time profile
ptrace	process trace
putc	write a character to a Fortran logical unit
putenv	set environment variable
qsort	quick sort
read	read from a file descriptor
readlink	read value of symbolic link
rename	change the name of a file
rmdir	remove a directory
sbrk	change data segment space allocation

Function	Purpose
schedctl	call to scheduler control
send	send a message to a socket
setblockproccnt	set semaphore count
setgid	set group ID
sethostid	set current host ID
setoserror	set system error
setpgrp	set process group ID
setsockopt	set options on sockets
setuid	set user ID
sginap	put process to sleep
sginap64	put process to sleep in 64-bit environment
shmat	attach shared memory
shmdt	detach shared memory
sighold	raise priority and hold signal
sigignore	ignore signal
signal	change the action for a signal
sigpause	suspend until receive signal
sigrelse	release signal and lower priority
sigset	specify system signal handling
sleep	suspend execution for an interval
socket	create an endpoint for communication TCP
sproc	create a new share group process
stat	get file status
stime	set time
symlink	make symbolic link
sync	update superblock
sysmp	control multiprocessing
sysmp64	control multiprocessing in 64-bit environment

Function	Purpose
system	issue a shell command
taskblock	block tasks
taskcreate	create a new task
taskctl	control task
taskdestroy	kill task
tasksetblockcnt	set task semaphore count
taskunblock	unblock task
time	return system time (must be declared EXTERNAL)
ttynam	find name of terminal port
uadmin	administrative control
ulimit	get and set user limits
ulimit64	get and set user limits in 64-bit architecture
umask	get and set file creation mask
umount	dismount a file system
unblockproc	unblock processes
unlink	remove a directory entry
uscalloc	shared memory allocator
uscalloc64	shared memory allocator in 64-bit environment
uscas	compare and swap operator
usclosepollsema	detach file descriptor from a pollable semaphore
usconfig	semaphore and lock configuration operations
uscpsema	acquire a semaphore
uscsetlock	unconditionally set lock
usctlsema	semaphore control operations
usdumplock	dump lock information
usdumpsema	dump semaphore information
usfree	user shared memory allocation
usfreelock	free a lock

Function	Purpose
`usfreepollsema`	free a pollable semaphore
`usfreesema`	free a semaphore
`usgetinfo`	exchange information through an arena
`usinit`	semaphore and lock initialize routine
`usinitlock`	initialize a lock
`usinitsema`	initialize a semaphore
`usmalloc`	allocate shared memory
`usmalloc64`	allocate shared memory in 64-bit environment
`usmallopt`	control allocation algorithm
`usnewlock`	allocate and initialize a lock
`usnewpollsema`	allocate and initialize a pollable semaphore
`usnewsema`	allocate and initialize a semaphore
`usopenpollsema`	attach a file descriptor to a pollable semaphore
`uspsema`	acquire a semaphore
`usputinfo`	exchange information through an arena
`usrealloc`	user share memory allocation
`usrealloc64`	user share memory allocation in 64-bit environment
`ussetlock`	set lock
`ustestlock`	test lock
`ustestsema`	return value of semaphore
`usunsetlock`	unset lock
`usvsema`	free a resource to a semaphore
`uswsetlock`	set lock
`wait`	wait for a process to terminate
`write`	write to a file

A.5 Compatibility with sproc

The parallelism used in Fortran is implemented using the sproc(2) system call. It is recommended that programs not attempt to use both !$DOACROSS loops and sproc calls. It is possible, but there are several restrictions:

- Any threads you create may not execute $DOACROSS loops; only the original thread is allowed to do this.
- The calls to routines like mp_block and mp_destroy apply only to the threads created by mp_create or to those automatically created when the Fortran job starts; they have no effect on any user-defined threads.
- Calls to routines such as m_get_numprocs do not apply to the threads created by the Fortran routines. However, the Fortran threads are ordinary subprocesses; using the kill routine with the arguments 0 and sig (for example, kill(0,sig)) to signal all members of the process group might kill threads used to execute !$DOACROSS. If you choose to intercept the SIGCLD signal, you must be prepared to receive this signal when the threads used for the !$DOACROSS loops exit; this occurs when mp_destroy is called or at program termination.
- The m_fork call is implemented using sproc, so it is not legal to run m_fork on a family of processes that each subsequently executes !$DOACROSS loops. Only the original thread can execute !$DOACROSS loops.

A.6 Communicating between threads

The routines described in this section allow you to perform explicit communication between threads within their multiprocessed Fortran program. These communication mechanisms are similar to message-passing, one-sided-communication, or shmem, and may be desirable for reasons of performance or style.

The operations allow a thread to fetch from (get) or send to (put) data belonging to other threads. Therefore these operations can be performed only on data that has been declared to be -Xlocal (that is, each thread has its own private copy of that data; see the ld(1) reference page for details on Xlocal), the equivalent of the Cray TASKCOMMON directive. A **get** operation requires that source point to Xlocal data, while a **put** operation requires that target point to Xlocal data.

These routines are similar to the original shmem routines (see the shmem reference page on your IRIX systems system), but are prefixed by mp_:

- mp_shmem_get32
- mp_shmem_put32
- mp_shmem_iget32
- mp_shmem_iput32
- mp_shmem_get64
- mp_shmem_put64
- mp_shmem_iget64
- mp_shmem_iput64

For the preceding routines:

- Both source and target are pointers to 32-bit quantities for the 32-bit versions, and to 64-bit quantities for the 64-bit versions of these calls.
- length specifies the number of elements to be copied, in units of 32 or 64-bit elements, as appropriate.
- The source_thread and target_thread specify the thread-number of the remote PE.
- A **get** copies from the remote PE, and **put** copies to the remote PE.
- target_inc and source_inc are specified for the strided iget/iput operations. They specify the increment (in units of 32 or 64 bit elements) along each of source and target when performing the data transfer. The number of elements copied during a strided put or get operation is still determined by length.

Call these routines only after the threads have been created (typically, the first DOACROSS/PARALLEL region). Performing these operations while the program is still serial leads to a run-time error because each thread's copy has not yet been created.

In the following example, compiling with -Wl, -Xlocal,mycommon_ ensures that each thread has a private copy of x and y.

```
integer x
real*8 y(100)
```

```
common /mycommon/ x, y
```

The following example copies the value of `x` on thread 3 into the private copy of `x` for the current thread.

```
call mp_shmem_get32 (x, x, 1, 3)
```

The next example copies the value of `localvar` into the thread-5 copy of `x`.

```
call mp_shmem_put32 (x, localvar, 1, 5)
```

The following example below fetches values from the thread-7 copy of array `y` into `localarray`.

```
call mp_shmem_get64 (localarray, y, 100, 7)
```

The next example copies the value of every other element of `localarray` into the thread-9 copy of `y`.

```
call mp_shmem_iput64 (y, localarray, 2, 2, 50, 9)
```

Debugging [B]

This appendix describes some aspects of debugging Fortran 90 source code. The recommended debugger for use with the MIPSpro 7 Fortran 90 compiler is dbx(1). The dbx(1) debugger includes the following features to support the Fortran 90 language: allocatable arrays, pointer-based variables, nonstandard stride arrays, modules, and derived types. For more information on this debugger, see the dbx(1) man page.

B.1 Compiling and running parallel Fortran

After you have written a program for parallel processing, it is best to debug your program in a single-processor environment. After your program executes successfully on a single processor, you can compile it for multiprocessing by using the -mp option to the -f90 command. This option causes the Fortran compiler to generate multiprocessing code for the files being compiled. At load time, you can specify both object files produced with the -mp option and object files produced without it. If any or all of the files are compiled with -mp, the executable must be loaded with -mp so that the correct libraries are used.

B.1.1 Using the `-static` option

When multiprocessing is used, it creates a demand for stack use to allow multiple threads of execution to execute the same code simultaneously.

If the parallel loop calls an external routine, that external routine cannot be compiled with -static. You can mix static and multiprocessed object files in the same executable; however, a static routine cannot be called from within a parallel loop.

Example 1: Multiprocessor executable. The following command line compiles and loads the Fortran program foo.f:

```
% f90 -mp foo.f
```

Example 2: Multiprocessor and optimizer. In the following example, the Fortran routines in the file snark.f are compiled with multiprocessing code generation (the -mp option) enabled. The optimizer is also used (the -O2 option):

```
% f90 -c -mp -O2 snark.f
```

A standard snark.o binary is produced, which must be loaded:

```
% f90 -mp -o boojum snark.o bellman.o
```

In this example, the -mp option signals the loader to use the Fortran multiprocessing library. The bellman.o file did not have to be compiled with the -mp option.

After loading, the resulting executable can be run like any executable file. Creating multiple execution threads, running and synchronizing them, and task termination are all handled automatically.

When an executable is loaded with -mp, the Fortran initialization routines determine how many parallel threads of execution to create. This determination occurs each time the task starts; the number of threads is not compiled into the code. The default is to use either 4 or the number of processors that are on the machine, whichever is less. The number of processors is determined by the value returned by the sysmp(MP_NAPROCS) system call; see the sysmp(2) man page for more information.

You can override the default by setting the MP_SET_NUMTHREADS environment variable. If it is set, any Fortran tasks use the specified number of execution threads regardless of the number of processors physically present on the machine. The value for MP_SET_NUMTHREADS can be from 1 to 64.

B.2 Profiling a parallel Fortran program

After converting a program from use on one processor to one that can be multiprocessed, you should examine execution profiles to judge the effectiveness of the transformation. Good profiles of the program are crucial to help you focus on the loops that use the most time. You can use SpeedShop to obtain these profiles. For more information on SpeedShop, see the *SpeedShop User's Guide* or the ssrun(1) man page.

If your job uses multiple threads, you can use SpeedShop creates multiple profile data files, one profile file for each thread. Use the prof(1) standard profile analyzer to examine this output. You can also use timex(1); this command indicates if the parallelized versions performed better overall than the serial version.

The profile of a Fortran parallel job is different from a standard profile. To produce a parallel program, the compiler pulls the parallel DO loops out into separate subroutines, one routine for each loop. Each of these loops is shown as a separate procedure in the profile. You can compare the amount of time spent

in each loop by the various threads to determine how well the workload is balanced.

You can use par(1) to trace the activity of a single process, a related group of processes, or the system as a whole. The par(1) utility is a process activity reporter. For more information on par(1), see the par(1) man page.

In addition to the loops, the profile returned by the prof(1) command shows the special routines that actually do the multiprocessing. The __mp_parallel_do routine is the synchronizer and controller. Slave threads wait for work in the routine __mp_slave_wait_for_work; the less time they wait, the more time they work. This gives a rough estimate of the extent of parallelism in a program. For more information on these routines, see the mp(3F) man page.

B.3 Debugging parallel Fortran

Debugging a multiprocessed program can be more difficult than debugging a single-processor program. Therefore you should do as much debugging as possible on the single-processor version. Try to isolate the problem and, if possible, try to reduce the problem to a single DOACROSS loop.

To determine if a loop can be multiprocessed, change the order of the iterations on the parallel DO loop on a single-processor version. If the loop can be multiprocessed, the iterations can execute in any order and produce the same answer. If the loop cannot be multiprocessed, changing the order usually causes the single-processor version to fail. You can use single-process debugging techniques to determine the problem.

Example. Erroneous !$DOACROSS. In this example, two references to A have the indexes in reverse order. If the indexes were in the same order (if both were A(I,J) or both were A(J,I)), the loop could be multiprocessed. As written, there is a data dependency, so the DOACROSS is an error.

```
!$DOACROSS LOCAL(I,J)
        DO I = 1, N
           DO J = 1, N
              A(I,J) = A(J,I) + X*B(I)
           END DO
        END DO
```

Because a (correct) multiprocessed loop can execute its iterations in any order, you could rewrite this as:

```
!$DOACROSS LOCAL(I,J)
        DO I = N, 1, -1
           DO J = 1, N
              A(I,J) = A(J,I) + X*B(I)
           END DO
        END DO
```

This loop no longer gives the same answer as the original even when compiled without the -mp option. This reduces the problem to a normal debugging problem.

B.3.1 Other debugging tips for multiprocessed loops

If a multiprocessed loop produces the wrong answer, use the following checklist to determine the cause:

Item to investigate	Reasons
LOCAL variables	Check the LOCAL variables when the code runs correctly as a single process but fails when multiprocessed. Check any scalar variables that appear in the left-hand side of an assignment statement in the loop to be sure they are all declared as LOCAL. Be sure to include the DO variable of any loop nested inside the parallel loop.
LASTLOCAL	A problem occurs when you need the final value of a variable but the variable is declared LOCAL rather than LASTLOCAL. If the use of the final value happens several hundred lines farther down, or if the variable is in a common block and the final value is used in a completely separate routine, a variable can look as if it is LOCAL when in fact it should be LASTLOCAL. To combat this problem, simply declare all the LOCAL variables LASTLOCAL when debugging a loop.
EQUIVALENCE	Check for EQUIVALENCE problems. Two variables of different names may in fact refer to the same storage location if they are associated through an EQUIVALENCE.

	EQUIVALENCE statements affect storage of local variables and can cause data dependencies when parallelizing code. EQUIVALENCE statements with local variables cause the storage location to be initialized to zero and saved between calls to the subroutine.
Uninitialized variables	Some programs assume uninitialized variables are set to 0. This works with the -static option on the f90(1) command, but without it, uninitialized values assume the value that remains on the stack. When compiling with the -mp option on the f90(1) command, the program executes differently and the stack contents are different. You should suspect this type of problem when a program is compiled with -mp and is run on a single processor and produces a different result when it is compiled without -mp. To discover this type of problem, compile suspected routines with the -static option. If an uninitialized variable is the problem, you should initialize the variable rather than compile the program with the -static option.
Ranges on arrays	Perform array bounds checking analysis by compiling with the -C option on the f90(1) command. If arrays are indexed out of bounds, a memory location may be referenced in unexpected ways. This is particularly true of adjacent arrays in a common block.

Errors in choosing which arrays are SHARED can be detected only when running on multiple processors. When stepping through the code in the debugger, the program executes correctly.

The most likely candidates for this error are arrays with complicated subscripts. If the array subscripts are simply the variables of a DO loop, the analysis is probably correct. If the subscripts are more involved, examine those subscripts first.

If you suspect this type of error, print out all the values of all the subscripts on each iteration through the loop. Then use the uniq(1) command to look for duplicates. If duplicates are found, there is a data dependency.

Differences [C]

This appendix describes the differences between the MIPSpro 7 Fortran 90 compiler and the Cray Research CF90 compiler.

C.1 Model differences

The model differences are as follows:

- The model for the CF90 `REAL(KIND=16)` data type on CRAY T90 systems that support IEEE floating-point arithmetic is different from the model for the MIPSpro 7 Fortran 90 compiler. This means that the results of math functions, arithmetic calculations, I/O, and other library routines are different for this particular data type.
- The internal size of `INTEGER(KIND=1)`, `INTEGER(KIND=2)`, `LOGICAL(KIND=1)`, and `LOGICAL(KIND=2)` on the MIPSpro 7 Fortran 90 compiler is actually one and two bytes, respectively. The CF90 compiler treats these kind type parameters as `INTEGER(KIND=4)` and `LOGICAL(KIND=4)`.
- The default sizes of the MIPSpro 7 Fortran 90 integer, real, and logical data types are 32 bits. This differs from the CF90 default of 64 bits. The default data type sizes for the MIPSpro 7 Fortran 90 compiler may be incorrect for routines such as `IRTC`(3I) and SHMEM.
- The MIPSpro 7 Fortran 90 compiler does not support Cray character pointers.
- Pointer arithmetic is in default numeric storage units when using the CF90 compiler. Pointer arithmetic is in bytes when using the MIPSpro 7 Fortran 90 compiler.

For more information on the model, see the `model`(3I) man page.

C.2 Fortran 90 statement differences

The Fortran 90 statement differences are as follows:

- When using the MIPSpro 7 Fortran 90 compiler, the execution of the `STOP` statement does not cause the word `STOP` to be written to `stdout` unless there is an argument to the `STOP` statement. The CF90 compiler always writes `STOP` to `stdout`.

- When using the MIPSpro 7 Fortran 90 compiler, the initialization of entities in a common block in a `DATA` statement mcan only be done in one program unit. That is, if a common block contains two variables initialized in a `DATA` statement, those `DATA` statements must be in one program unit. The load indicates the presence of multiple initializations, and only one initialization is done.

  With the CF90 compiler, each variable can be initialized in `DATA` statements in separate program units.

C.3 Functions and procedures

The CF90 typeless functions (such as `MASK`(3I), `SHIFTL`(3I), `SHIFTR`(3I), `SHIFT`(3I), `CVM`(3I), and so on) are typed as integer functions by the MIPSpro 7 Fortran 90 compiler. Conversion occur in expressions involving a mixture of floating point and integer functions. When called by the CF90 compiler, these functions are typeless and no conversion occurs when there is a mixture of floating point and these typeless functions.

C.4 Modules

When using the MIPSpro 7 Fortran 90 compiler, the compilation of Fortran 90 modules creates a *file*`.mod` for each module in the source file. When using the CF90 compiler, compiling modules creates one *file*`.o` that contains all the Fortran 90 modules in the source file.

C.5 I/O library

The I/O library differences are as follows:

- Direct access formatted output files cannot be read as sequential formatted files by MIPSpro 7 Fortran 90 programs unless an `assign`(1) command with `-s unblocked` or `-F cachea` is supplied for the particular file.
- The set of I/O library errors begins at 4000 for MIPSpro 7 Fortran 90 programs. The error numbers begin at 1000 for CF90 programs.
- The `FILENV` environment variable must be set for MIPSpro 7 Fortran 90 programs when using the `assign`(1) command. For CF90 uers, this environment variable need not be set.

C.6 Library functions and procedures

The library function and intrinsic procedure differences are as follows:

- The CRI_IEEE_DEFINITIONS module is available for the MIPSpro 7 Fortran 90 compiler, but the preferred name is FTN_IEEE_DEFINITIONS for the IEEE module and the interface to the IEEE procedures.
- The MAXVAL(3I) intrinsic procedure returns negative infinity for a zero-sized input array on when called from a MIPSpro 7 Fortran 90 program and returns -HUGE(3I) when called from a CF90 program. A request for interpretation has been submitted to the Fortran standards committee.
- The MINVAL(3I) intrinsic procedure returns positive infinity for a zero-sized input array on when called from a MIPSpro 7 Fortran 90 program and returns +HUGE(3I) when called from a CF90 program. A request for interpretation has been submitted to the Fortran standards committee.

C.7 Math library

The math library differences are as follows:

- The math routines from the MIPSpro 7 Fortran 90 compiler are referenced from the compiler. The results of the scalar math routines may differ from the results of the vector math routines. The results of the math routines from the MIPSpro 7 Fortran 90 compiler may differ from the results returned by the math routines for the CF90 compiler.
- Signalling of errors during references to the MIPSpro 7 Fortran 90 compiler math routines is not turned off. For the CF90 compiler, the math routines turn off signaling of errors and detect input data errors through source code checks.

Index

!$ directive, 74
(null) directive, 151
– option, 47
-32 option, 6
-64 option, 6

A

ABI, 6
Affinity clause, 112
Affinity scheduling, 112
  data affinity, 113
  examples, 125
  thread affinity, 113
AGGRESSIVEINNERLOOPFISSION directive, 52
AINT, 33
ALIGN_SYMBOL directive, 61
-alignn option, 8
AMOD, 33
ANINT, 33
-ansi option, 9
Application Binary Interface (ABI)
  (See ABI), 6
ar, 4
Archive library
  definition, 3
Archiving tool
  definition, 3
Argument aliasing directives
  (See Directives), 60
Array slices, 162
Arrays
  assumed-shape, 162
  character, 174
  deferred-shape, 162
  elements
    Fortran 90 and C correspondence, 161
  example, 7
  Fortran 90 arrays in C code, 170
  processor, 121
  reshaped, 120
  slices, 162
  unsupported array arguments, 162
Assembly language
  calling from Fortran 90, 177
ASSERT ARGUMENTALIASING directive, 60
ASSERT NOARGUMENTALIASING directive, 60
assign, 179
Assumed-shape arrays, 162
Asynchronous I/O, 182
Autocloning
  enable/disable, 29
Automatic page migration, 107
Autotasking
  restrictions, 136
Autotasking directives
  overview, 135, 136
-avoid_gp_overflow option, 9

B

BARRIER directive, 94
BLOCK distribution, 122
BLOCK-CYCLIC distribution, 123
BLOCKABLE directive, 52
BLOCKINGSIZE directive, 53
Blocks
  common, 92
BOUNDS directive, 132, 133

C

-C option, 9
-c option, 9
C/C++, 157
  calling C from Fortran 90, 167

calling Fortran 90, 164
calling Fortran 90 functions, 166
calling Fortran 90 subroutines, 164
external functions, 164
Fortran 90 and C correspondence, 159
Fortran 90 arrays in C code, 170
Fortran 90 blocks in C code, 169
makefile considerations, 176
making wrappers, 172
mkf2c argument assumptions, 172
mkf2c character string treatment, 173
mkf2c restrictions, 175
normal calls to C functions, 168
using %LOC, 171
using %VAL, 171
Cache
and optimization, 88
memory management, 23
performance, 88
prefetch options, 25
TLB, 24
transformation options, 20
CDIR$, 131, 132
Character address, 173
Character array, 174
Character types
Fortran 90 and C correspondence, 160
-check_bounds option, 9
CHUNK directive, 73
-chunk=integer option, 10
CIF, 3
cifconv, 3
Clauses
affinity, 112
NEST, 114
CMIC!, 132
CMIC$, 131
Code scheduler
specifying, 41
-coln option, 10
Common blocks
Fortran 90 in C code, 169
reorganizing, 35
common blocks, 92
Communication
between threads, 195
Compiler
invoking, 1
Compiler features, 49
Compiler information file (CIF)
(See CIF), 3
COMPILER_DEFAULTS_PATH, 42, 43
Conditional compilation
directives
(See Directives), 148
overview, 147
Conditional directives
(See Directives), 151
Consistency checks, 28
Constructs
critical section, 93, 101
parallel sections, 98, 100
PDO, 96
single process, 100
work-sharing, 93
Continuation character, 51
!$COPYIN directive, 92
cord, 10
-cord option, 10
Correspondence
between Fortran 90 and C data types, 159
cpp, 10
-cpp option, 10
CPU targeting
(See also Cross compiling), 42
Critical section, 93
CRITICALSECTION directive, 94
Cross compiling
definition, 42
CYCLIC distribution, 122

D

-D option, 11
Data dependence
examples, 78

rewriting, 81
Data dependencies, 77
multiprocessing errors, 203
Data distribution
*, 118
BLOCK, 118
CYCLIC, 118
DISTRIBUTE directive, 111
DISTRIBUTE_RESHAPE, 111
examples, 125
REDISTRIBUTE, 111
regular, 109, 119
regular vs. reshaped, 124
RESHAPE directive, 120
reshaped, 109
restriction on reshaped arrays, 120
with reshaping, 120
Data placement
automatic page migration, 107
regular data distribution, 107
Data types
Fortran 90 and C correspondence, 159
Debugging
generating information, 13
parallel Fortran, 201
tips for multiprocessed loops, 202
-default64 option, 11
Deferred-shape arrays, 162
#define, 11
#define directive, 149
Dependency analysis
examples, 78
!DIR$, 131, 132
Directive
definition, 49
Directives
!$, 74
(null), 151
AGGRESSIVEINNERLOOPFISSION, 52
ALIGN_SYMBOL, 61
example, 62
and command line options, 50, 65
ASSERT ARGUMENTALIASING, 60
ASSERT NOARGUMENTALIASING, 60
BARRIER, 94
BLOCKABLE, 52
BLOCKINGSIZE, 53
CHUNK, 73
conditional, 151
continuation, 51, 66
continuing, 132
!$COPYIN, 92
CRITICALSECTION, 94
data distribution, 29
#define, 11, 149
DISTRIBUTE, 111
DISTRIBUTE_RESHAPE, 109, 111, 122
DOACROSS, 67, 112
DSM, 29
DYNAMIC, 116
#elif, 151, 153
#else, 151, 154
ENDCRITICALSECTION, 94
#endif, 151, 154
ENDPARALLEL, 95
ENDPDO, 96
ENDPSECTION, 98
ENDSINGLEPROCESS, 100
FILL_SYMBOL, 61
example, 62
FISSION, 54
FISSIONABLE, 54
fixed source form, 50
for Autotasking, 135
for local use of compiler features, 133
for optimization, 133
free source form, 50
FUSABLE, 55
FUSE, 55
global
definition, 65
#if, 152
#ifdef, 153
#ifndef, 153
#include, 148
INLINE, 63
Inlining and interprocedural analysis (IPA), 63

interaction with -x dirname option, 133
INTERCHANGE, 56
IPA, 63
IVDEP, 31
LNO, 51
MP_SCHEDTYPE, 74
multiprocessing, 65
NOBLOCKING, 53
NOFISSION, 54
NOFUSION, 55
NOINLINE directive, 63
NOINTERCHANGE, 56
NOIPA, 63
overview, 131
PAGE_PLACE, 117
PARALLEL, 95
PARALLELDO, 96
PCF, 93
restrictions, 102
PDO, 96
performance tuning, 110
PREFETCH, 57
PREFETCH_MANUAL, 58
PREFETCH_REF, 58
PREFETCH_REF_DISABLE, 59
PSECTION, 98
range, 51
range and placement, 132
REDISTRIBUTE, 111
regular data distribution, 119
RESHAPE, 120
SECTION, 98
SINGLEPROCESS, 100
source preprocessor, 51
symbol storage, 61
syntax, 49
#undef, 151
UNROLL, 59
using, 49
DISTRIBUTE directive, 111
DISTRIBUTE_RESHAPE directive, 109, 111, 122
-dn option, 11
DO loop, 66
DO PARALLEL directive, 138
DOACROSS, 201
example, 75
DOACROSS directive, 67, 112
!$DOACROSS
sproc compatibility, 195
!$DOACROSS loop, 68
DOALL directive, 132, 136
DOPARALLEL directive, 132
DYNAMIC directive, 116
DYNAMIC schedules, 92
Dynamic shared libraries, 15

E

-E option, 12
#elif directive, 151, 153
#else directive, 151, 154
ENDCRITICALSECTION directive, 94
ENDDO directive, 132, 138
ENDGUARD directive, 133, 139
#endif directive, 151, 154
ENDPARALLEL directive, 95, 133, 140
ENDPDO directive, 96
ENDPSECTION directive, 98
ENDSINGLEPROCESS directive, 100
Environment variables
affecting compilation, 5
COMPILER_DEFAULTS_PATH, 42, 43
MP_SET_NUMTHREADS, 66, 119, 200
Error detection, 3
Examples
arrays, 7
loading Fortran 90 object files, 26
setting stack size, 8
specifying libraries, 27
extcentry, 177
using with mkf2c, 175
-extend_source option, 12
External name, 157

F

f90 command
  example, 1
  MIPSpro Automatic Parallelization Option, 5
  options, 17
    –, 47
    -32, 6
    -64, 6
    -alignn, 8
    -ansi, 9
    -avoid_gp_overflow, 9
    -C, 9
    -c, 9
    -check_bounds, 9
    -chunk=integer, 10
    -coln, 10
    -cord, 10
    -cpp, 10
    -D, 11
    -default64, 11
    -dn, 11
    -E, 12
    -extend_source, 12
    -fbfile.cfb, 12
    -feedbackfile, 12
    -fixedform, 12
    -freeform, 12
    -ftpp, 13
    -gdebug_lvl, 13
    -help, 13
    -Idir, 13
    -ignore_suffix, 14
    -in, 14
    -INLINE:…, 14
    -INLINING, 63
    -IPA, 63
    -IPA:…, 14
    -keep, 15
    -KPIC, 15
    -Ldirectory, 15
    -LIST:..., 15
    -listing, 27
    -llibrary, 26
    -LNO:…, 16
    -macro_expand, 28
    -MDupdate, 27
    -mipsn, 6, 28
    -mp, 199
    -MP:, 28
    -nocpp, 30
    -noextend_source, 30
    -nostdinc, 30
    -o, 38
    -Olevel, 39
    -OPT:…, 31
    -P, 40
    -pfa, 40
    -rprocessor, 41
    -rreal_spec, 40
    -S, 41
    -static, 41, 80
    -TARG:..., 42
    -TENV:..., 44
    -trapuv, 46
    -Uvar, 46
    -warg, 46
    -woffnum, 46
    -xgot, 47
  syntax, 5
  using multiple options, 5
-fbfile.cfb option, 12
Feedback files
  naming, 12
  specifying, 12
-feedbackfile option, 12
FFIO
  and asynchronous I/O, 182
  routines
    (See Library routines), 183
  supported layers for IRIX, 182
file.suffix90, 48
file.suffix90 option, 48
FILL_SYMBOL directive, 61
FISSION directive, 54
FISSIONABLE directive, 54
FIXED directive, 132, 135

Fixed source form, 50
-fixedform option, 12
Flexible File I/O (FFIO)
    (See FFIO), 182
Floating-point mode, 42
FORTRAN 77 compiler
    $ character difference, 158
    -U option, 158
Fortran 90
    and C data types, 159
    arrays in C code, 170
    calling assembly language, 177
    calling C, 167
    calling from C, 164
    calling function from C, 166
    calling subroutines from C, 164
    common blocks in C code, 169
    compiling, 199
    functions, 164
    makefile considerations, 176
    making C wrappers, 172
    mkf2c argument assumptions, 172
    mkf2c character string treatment, 173
    mkf2c restrictions, 175
    naming C functions, 159
    naming subprogram from C, 158
    normal calls to C functions, 168
    passing subprogram arguments, 162
    subroutines, 164
    using %LOC, 171
    using %VAL, 171
FREE directive, 132, 135
Free source form, 50
-freeform option, 12
ftnchop, 4
ftnlint, 3
ftnlist, 3
ftnmgen, 4
ftnsplit, 4
ftpp, 12, 13
-ftpp option, 13
Functions
    calling Fortran 90 from C, 166
    normal calls to C functions, 168
FUSABLE directive, 55
FUSE directive, 55

G

-gdebug_lvl option, 13
getwd, 27
Global directives, 65
Global Symbol Table (GOT)
    (See GOT), 9
GOT, 9
    accommodating larger, 44
    overflow message, 47
GSS schedules, 92
GUARD directive, 133, 139

H

-help option, 13
hinv, 6, 39

I

I/O routines
    (See Library routines), 184
ID directive, 132, 142
-Idir option, 13
IEEE Floating-point Arithmetic
    level of conformance, 34
IF parameter, 137
#if directive, 152
#ifdef directive, 153
#ifndef directive, 153
-ignore_suffix option, 14
-in option, 14
#include directive, 148
#include files
    searching for, 13
INLINE directive, 63
-INLINE:... option, 14

Inlining
  definition, 63
  intrafile subprogram inlining, 14
  standalone inliner, 14
Inlining and interprocedural analysis (IPA) directives
  (See Directives), 63
-INLINING option, 63
Instruction Set Architecture (ISA)
  (See ISA), 6
INTERCHANGE directive, 56
Interface routines
  (See Library routines), 184
Interlanguage calling, 157
Interleaving
  cache performance, 91
  load balancing, 91
Interprocedural analysis (IPA)
  definition, 63
  ipa, 63
Interprocedural analyzer (IPA)
  (See IPA), 14
Interthread communication, 93
Intrinsic procedures, 3, 183
  AINT, 33
  AMOD, 33
  ANINT, 33
  libfortran, 3
  libm, 3
  NINT, 33
  turning into a call, 35
IPA, 14
  directives, 63
ipa, 63
IPA directive, 63
-IPA option, 63
-IPA:... option, 14
IRIX loader
  ld, 3
Irregular data structures, 117
ISA
  specifying, 28
IVDEP directive, 132, 144

K

-keep option, 15
KIND specification
  values, 11
Kind specification
  real values, 40
-KPIC option, 15

L

Language interface
  C/C++, 157
LASTLOCAL variable, 77
ld, 3, 92, 157
-Ldirectory option, 15
libfortran, 3
libm, 3
Libraries, 3
  changing search algorithm, 15
  FFIO, 182
  loaded by default, 26
  searching lib.library.a, 26
Library options, 179
Library routines, 188, 183
  communication between threads, 195
  FFIO
    fffcntl, 183
    ffopen, 183
    ffpos, 183
    ffread, 183
    ffseek, 183
  I/O
    ASNCTL, 184
    ASNQFILE, 184
    ASSIGN, 184
    FLUSH, 184
    NUMBLKS, 184
    RNL, 184
    RNLECHO, 184
    RNLSKIP, 184
    RNLTYPE, 184

WNL, 184
WNLLINE, 184
WNLLONG, 184
Interface
ABORT, 184
EXIT, 184
ISHELL, 184
POSIX
IPXFARGC, 185
PXFACCESS, 185
PXFCHDIR, 185
PXFCHMOD, 185
PXFCHOWN, 185
PXFCHROOT, 185
PXFCLEARENV, 185
PXFCONST, 185
PXFCREAT, 185
PXFCTERMID, 185
PXFDIRECTORY, 185
PXFESTRGET, 185
PXFEXECV, 185
PXFFCNTL, 185
PXFFILENO, 185
PXFFORK, 185
PXFGETARG, 185
PXFGETEGID, 185
PXFGETENV, 185
PXFGETEUID, 185
PXFGETGID, 185
PXFGETGRGID, 185
PXFGETGRNAM, 185
PXFGETGROUPS, 185
PXFGETLOGIN, 185
PXFGETPGRP, 185
PXFGETPID, 185
PXFGETPPID, 185
PXFGETPWNAM, 185
PXFGETPWUID, 185
PXFGETUID, 185
PXFINTGET, 185
PXFINTSET, 185
PXFISATTY, 185
PXFISBLK, 185
PXFISCHR, 185
PXFISDIR, 185
PXFISFIFO, 185
PXFISREG, 185
PXFLINK, 185
PXFLOCALTIME, 185
PXFOPEN, 185
PXFRENAME, 185
PXFRMDIR, 185
PXFSETENV, 185
PXFSETGID, 185
PXFSETPGID, 185
PXFSETSID, 185
*PXFSETUID, 185
PXFSTAT, 185
PXFSTRGET, 185
PXFSTRSET, 185
PXFSTRUCTCOPY, 185
PXFSTRUCTCREATE, 185
PXFSTRUCTFREE, 185
PXFSYSCONF, 185
PXFTIME, 185
PXFTIMES, 185
PXFUCOMPARE, 185
PXFUMASK, 185
PXFUNAME, 185
PXFUNLINK, 185
PXFUTIME, 185
PXFWAIT, 185
programming aids
SECOND, 184
SECONDR, 184
SYSCLOCK, 184
TIMEF, 184
Lines
restricting Fortran source code lines, 30
specifying length, 12
specifying width, 10
lint
(See ftnlint), 3
-LIST:... option
arguments, 15
Lister
ftnlist, 3

using f90 command, 3
Listing file
writing to, 15
writing to assembly listing file, 15
-listing option, 27
-llibrary option, 26
LNO
directives
(See Directives), 51
-LNO option, 16
-LNO option arguments, 16
Load balancing, 90, 92
Loader
ld, 3
Loading compiler, 3
%LOC intrinsic function, 172
Local common blocks, 92
LOCAL variable, 77
Loop nest optimization, 51
Loop nest optimizer (LNO)
(See LNO), 16
Loop unrolling
UNROLL directive, 59
Loops
unrolled, 38

M

Macro expansion, 28
-macro_expand option, 28
Macros
based on host system, 154
based on IRIX system, 154
predefined, 154
D_LANGUAGE_FORTRAN90, 155
DLANGUAGE_FORTRAN90, 155
host_mips, 155
__host_mips, 155
LANGUAGE_FORTRAN, 155
_LANGUAGE_FORTRAN, 155
mips, 155
MIPSEB, 155
_MIPSEB, 155
sgi, 155
__sgi, 155
_SGI_SOURCE, 155
_SVR4_SOURCE, 155
_SYSTYPE_SVR4, 155
unix, 154
__unix, 154
make, 176
Makefile
considerations, 176
man, 4
Master/slave
Common block, 92
Master/slave organization, 66
Matrix multiply, 88
MAXCPUS parameter, 137
-MDupdate option, 27
Messages
generation of, 9
specifying, 46
!MIC$, 131, 132
-mipsn option, 6, 28
MIPSpro 7 Fortran 90 compiler
definition, 3
invoking, 5
MIPSpro assembly language
calling from Fortran 90, 177
MIPSpro Automatic Parallelization Option, 5
mkf2c, 172
argument assumptions, 172
character string treatment, 173
restrictions, 175
using with extcentry, 175
-mp option, 199
-MP: option
arguments, 28
MP_SCHEDTYPE directive, 74
MP_SET_NUMTHREADS, 119, 200
MP_SET_NUMTHREADS environment variable, 66
Multiprocessing
analyzing data dependencies, 77
debugging program, 201

directives
(See Directives), 65
DO loop, 66
!$DOACROSS, 68
loop-level, 66
master/slave orgzniation, 66
Origin series, 105
parallel Fortran, 199
specifying options, 28
thread of execution, 66
work quantum, 86
multiprocessing routines, 187
Multitasking, 135

N

NAME directive, 133, 144
NEST clause, 114
Nested parallelism, 114
NINT, 33
nm, 157
NOBLOCKING directive, 53
NOBOUNDS directive, 132, 133
-nocpp option, 30
-noextend_source option, 30
NOFISSION directive, 54
NOFUSION directive, 55
NOINLINE directive, 63
NOINTERCHANGE directive, 56
NOIPA directive, 63
-nostdinc option, 30

O

-o option, 38
Object file tools
definition, 4
-Olevel option, 39
Online documentation utilities, 4
-OPT:... option, 31
Optimization
controlling, 31
costs, 89
specifying level, 39
Optimization directives, 133
option, 17
Options
help, 13
Origin series
improving program performance, 106
memory hierarchy, 106
parallel programming, 105
performance tuning, 105

P

-P option, 40
PAGE_PLACE directive, 117
Parallel Computing Forum (PCF)
(See PCF), 93
PARALLEL directive, 95, 133, 140
Parallel processing
analyzing source code, 40
Parallel programming
Origin series, 105
Parallel region
definition, 93
PARALLELDO directive, 96
Parallelism
cache performance, 88
conditional, 87
general model based on PCF, 93
implementation, 195
nested, 114
profiling, 200
sproc, 195
Passing arguments, 163
PCF
directives, 93
PDO directive, 96
pe_environ, 5
Performance tuning, 105
Performance tuning directives
(See Directives), 110

-pfa option, 40
pixie, 10
pmake command, 27
Position-independent code (PIC)
(See PIC), 15
POSIX routines
(See Library routines), 185
Power Fortran, 78
Predefined macros
for conditional compilation, 154
PREFETCH directive, 57
PREFETCH_MANUAL directive, 58
PREFETCH_REF directive, 58
PREFETCH_REF_DISABLE directive, 59
Preprocessing, 147
f90 command line options, 155
Power Fortran, 78
Preprocessor
using f90 command, 3
PRIVATE parameter, 137
Procedure rearranging, 10
Processor array, 121
prof, 10, 200
Profiling
a parallel Fortran program, 200
Fortran parallel vs. standard, 200
__mp_parallel_do synchronizer and controller, 201
__mp_slave_wait_for_work routine, 201
prof standard profile analyzer, 200
timex profile analyzer, 200
Programming aids
(See Library routines), 184
Proprocessing
source, 10
PSECTION directive, 98
Public name, 157

R

Reciprocal operations
specifying faster, 36
REDISTRIBUTE directive, 111
Redistribution
DYNAMIC directive, 116
Reduction operation
definition, 72
REDUCTION variable, 77
Regular data distribution directives
(See Directives), 119
Relational operators, 34
unsigned, 34
RESHAPE directive, 120
Reshaped arrays
error detection, 121
implementation of, 121
restrictions, 120
Restrictions
on PCF directives, 102
on reshaped arrays, 120
-rprocessor option, 41
-rreal_spec option, 40

S

-S option, 41
SAVELAST parameter, 137
Scalar types
Fortran 90 and C correspondence, 159
Scheduling, 29
affinity, 68, 112
DYNAMIC schedules, 91
fixed schedules, 91
GSS schedules, 91
interleaving, 91
work, 69
SECTION directive, 98
Semantics, 31
sh, 7
SHARED parameter, 137
SHARED variable, 77
shmem
thread communication, 195
SINGLEPROCESS directive, 100
smake command, 27

Source preprocessing, 40, 147
Source preprocessor, 13
  cpp, 10
  disabling, 30
  ftpp, 12
Speculative code motion, 45
SpeedShop, 92
sproc
  compatibility with !$DOACROSS, 195
Square root
  calculation, 33
Static analyzer
  ftnlint utility, 3
-static option, 41, 199
  Caution, 80
Subroutines
  calling Fortran 90 from C, 164
Subscripts
  Fortran 90 and C correspondence, 162
Symbol storage directives, 61
System defaults
  predefined, 6

T

-TARG:... option
  arguments, 42
Target environment
  controlling alignment, 44
  specifying, 44
TASKCOMMON directive
  thread communication, 195
Tasking directives, 135
-TENV:... option, 44
Thread communication, 195
  examples, 196
timex, 200
-trapuv option, 46
Tuning, 105
  choosing a method, 109

U

#undef directive, 151
UNROLL directive, 59
-Uvar option, 46

V

%VAL intrinsic function, 171
Variables
  allocating local, 41
Vector dependencies
  ignoring, 35
VSEARCH directive, 132

W

-warg option, 46
-woffnum option, 46
Work quantum, 86
Work-sharing constructs, 93

X

-x dirname option, 133
-xgot option, 47
Xlocal
  thread communication, 195