GUIDE™ Reference Manual

Version 3.5

Document #9607001
Kuck & Associates, Inc.

GUIDE™ Reference Manual
Version 3.5

Revised September, 1997

Kuck & Associates, Inc.
1906 Fox Drive
Champaign, IL 61820-7345
USA

Phone: (217) 356-2288
FAX: 217-356-5199
Internet: kai@kai.com
WWW: http://www.kai.com/kpts/guide/

Printed in the United States of America.

Table of Contents

CHAPTER 1 1 *Introduction*

1 About Guide
2 Using this Reference Manual
2 *Reference Manual Contents*
3 *Reference Manual Conventions*
3 Guide On-line
4 Technical Support
4 Comments

CHAPTER 2 5 *Using Guide*

5 Parallel Processing Model
5 *Overview*
7 *Increasing Efficiency*
9 *Data Sharing*
9 Using Guide to Develop Parallel Programs

10 *Analyze*
10 *Restructure*
10 *Tune*
11 Orphaned Directives
13 *A Few Rules About the "Orphaned" Directives*

CHAPTER 3 15 *OpenMP Directives*

15 Control Directives
15 *PARALLEL/END PARALLEL*
16 *PDO/END PDO*
17 *PSECTIONS/END PSECTIONS*
17 *PSINGLE/END PSINGLE*
18 *PARALLEL DO/END PARALLEL DO*
19 *PARALLEL SECTIONS/END PARALLEL SECTIONS*
20 *IF(if_expr)*
20 *DEFAULT(SHARED|PRIVATE|NONE)*
SHARED(shared_vars)
PRIVATE(private_vars)
21 *FIRSTPRIVATE (firstprivate_vars)*
21 *LASTPRIVATE (lastprivate_vars)*
21 *REDUCTION(operator:reduction_vars)*
REDUCTION(intrinsic:reduction_vars)
22 *COPYIN (copyin_vars)*
22 Common Privatization
23 *INSTANCE PARALLEL*
23 *THREAD PRIVATE*
23 *Declaring Private Commons*
24 *Allocating Private Commons*
25 Synchronization Directives
25 *CRITICAL/END CRITICAL*
25 *ORDERED/END ORDERED*
26 *MASTER/END MASTER*
26 *ATOMIC*

26 *FLUSH[(string)]*
27 *BARRIER*
27 Scheduling Options
28 *Scheduling Options Using Directives*
29 *Scheduling Options Using Environment Variables*
29 Environment Variables
29 *KMP_LIBRARY=<string>*
30 *KMP_STACKSIZE=<integer>[,<character>]*
30 *KMP_STATSFILE=<file>*
30 *LD_LIBRARY_PATH=<path>*
30 *OMP_DYNAMIC=<boolean>*
31 *OMP_NUM_THREADS=<integer>*
31 *OMP_SCHEDULE=<string>[,<integer>]*

CHAPTER 4 33 *The Guide Drivers*

33 About Guidef77 and Guidef90
34 Using the Drivers
35 Driver Options
35 *Displaying the Driver Usage Message*
35 *Displaying All Command Lines*
35 *Suppressing Guidef Warnings*
36 Driver-Specific Options
36 *WG,guide_option_1[[[,guide_option_2],guide_option_3],...]*
36 *WGcompiler=<path>*
36 *WGcpp=<path>*
36 *WGf77=<path>*
36 *WGf90=<file>*
36 *WGfortran=<path>*
36 *WGftn=<path>*
37 *WGkeep*
37 *WGkeepcpp*
37 *WGld=<file>*

37 *WGlibpath=<path>*
37 *WGlink=<path>*
37 *WGnocpp*
37 *WGnokeep*
38 *WGnoprocess*
38 *WGnorc*
38 *WGonly*
38 *WGpath=<path>*
38 *WGprefix=<string>*
38 *WGsrcdir*
39 *WGversion*
39 Guide Options
39 Guide Options Functional Categories
39 *General Optimization*
39 *Input-Output*
39 *Listing*
39 *Advanced Optimization*
40 *FORTRAN Dialect*
40 *Hardware*
40 *Directive Recognition*
40 Guide Options Table
43 Guide Options Alphabetic Listing
43 *alignmax=<integer>*
43 *assume=<string> or a=<string>; noassume or nas*
44 *blank_padding or bp; noblank_padding or nbp*
44 *case or case; nocase or ncase*
44 *chunk=<integer> or chk=<integer>*
44 *cmp[=<file>]*
45 *concurrentize, conc; noconcurrentize, noconc*
45 *datasave or ds; nodatasave or nds*
45 *directives=p or dr=p; nodirectives or ndr*
45 *dlines or dl; nodlines or ndl*
46 *free; nofree*

46 *heaplimit=<integer> or heap=<integer>*
47 *ignoreoptions or ig; noignoreoptions or nig*
47 *include=<directory> or inc=<directory>*
47 *input=<file> or i=<file>*
47 *integer=<integer> or int=<integer>*
48 *lines=<integer> or ln=<integer>*
48 *list[=<file>] or l=[<file>]; nolist or nl*
48 *listoptions=<string> or lo=<string>*
48 *logical=<integer> or log=<integer>*
49 *minconcurrent=<integer> or mc=<integer>*
49 *onetrip or 1; noonetrip or n1*
50 *optimize=<integer> or o=<integer>*
50 *real=<integer> or rl=<integer>*
50 *recursion or rc; norecursion or nrc*
50 *roundoff=<string> or r=<string>*
51 *save=<string> or sv=<string>*
52 *scalaropt=<integer> or so=<integer>*
52 *scan=<integer> or scan=<integer>*
52 *scheduling=<character> or schd=<character>*
53 *suppress=<string> or su=<string>*
53 *syntax=<string> or sy=<string>*
53 *type or ty; notype or nty*
54 *c*$*options Line*

CHAPTER 5 55 *Libraries*

55 Selecting a Library
55 *Serial*
56 *Turnaround*
56 *Throughput*
56 The Guide_stats Library
57 Linking the Libraries
57 External Routines

57 *mppbeg(), mppend()*
58 *omp_get_max_threads()*
58 *omp_get_num_procs()*
58 *omp_get_num_threads()*
59 *omp_get_thread_num()*

CHAPTER 6 61 *GuideView*

61 Introduction
61 Using GuideView
62 GuideView Options
62 *mhz=<integer>*
63 *ovh=<file>*
63 *WJ,[java_option]*
63 JAVA Options
63 *ms<integer>[{k,m}]*
63 *mx<integer>[{k,m}]*

CHAPTER 7 65 *The Guide_stats Report*

66 Settings
66 Parallel Region
67 Synchronized Code
67 Synchronization
67 *Locks*
67 *Barriers*
68 *Join Barriers*
68 Average Statistics
68 Performance Projections
69 Event Counts
69 *Program Start/Finish*
70 *Internal Checks*
70 *Fork*

70 *Private Commons with INSTANCE PARALLEL or THREAD PRIVATE*
71 *Dynamic Scheduling*
71 *Synchronization Events*
72 *Internal Synchronization Events*
72 *Routine Events*
72 *Library Calls*
72 *Internal Events*

CHAPTER 8 73 *Directive Translation*

73 KAP/Pro Parallel Directive to OpenMP Directive Translator
74 Cray Directive to OpenMP Directive Translator
76 *Cray TASKCOMMON as opposed to OpenMP THREAD PRIVATE*
77 SGI Directive to KAP/Pro Parallel Directive Translator
79 KAP Directive to OpenMP Directive Translator

APPENDIX A 81 *Examples*

82 PDO: A Simple Difference Operator
83 PDO: Two Difference Operators
84 PDO: Reduce Fork/Join Overhead
85 PSECTIONS: Two Difference Operators
86 PSINGLE: Updating a Shared Scalar
87 PSECTIONS: Updating a Shared Scalar
88 PDO: Updating a Shared Scalar
89 PARALLEL DO: A Simple Difference Operator
90 PARALLEL SECTIONS: Two Difference Operators
91 Simple Reduction
92 TASKCOMMON: Private Common
93 THREAD PRIVATE: Private Common and Master Thread
94 INSTANCE PARALLEL: As a Private Common
95 INSTANCE PARALLEL: As a Shared and then a Private Common
96 Avoiding External Routines: Reduction

98 Avoiding External Routines: Temporary Storage
100 FIRSTPRIVATE: Copying in Initialization Values
101 THREAD PRIVATE: Copying in Initialization Values
102 INSTANCE PARALLEL: Copying in Initialization Values

APPENDIX B 103 *Timing Guide Constructs*

104 Typical Overhead

CHAPTER 1

Introduction

About Guide

The KAP/Pro Toolset is a system of tools and application accelerators for developers of large scale, parallel scientific-engineering software.

The KAP/Pro Toolset is intended for users who understand their application programs and understand parallel processing. The Guide component of the toolset implements OpenMP directives on all popular shared memory parallel (SMP) systems that support threads. The KAP/Pro Toolset uses the de facto industry standard OpenMP directives to express parallelism. The OpenMP directive set is compatible with the older directives from PCF, X3H5, SGI and Cray.

Throughout this manual, the term “OpenMP directives” is used to refer to the KAP/Pro Toolset implementation of the OpenMP de facto standard, unless stated otherwise.

The KAP/Pro Toolset includes a utility to translate directives from the older KAP/Pro Toolset parallel processing directives which were based on X3H5 (`c$par` directives) to the new OpenMP directives (`c$omp`).

The input to Guide is a FORTRAN program with OpenMP directives. The output of Guide is a FORTRAN program with the directive parallelism implemented using threads and the Guide support libraries. This output can be compiled using a FORTRAN compiler.

Please note that this version of Guide does not support `MODULES` and `INTERNAL PROCEDURES`.

Using this Reference Manual

Reference Manual Contents

Chapter 2, “Using Guide,” beginning on page 5, contains the OpenMP parallel processing model, an overview for using Guide, and an example to illustrate how to insert Guide directives .

Chapter 3, “OpenMP Directives,” beginning on page 15, contains definitions for all OpenMP directives. OpenMP directives control the level of parallelism within your code. This chapter also defines the Guide environment variables.

Chapter 4, “The Guide Drivers,” beginning on page 33, describes the Guide drivers, and it contains descriptions of all Guide command line options. These options allow you to alter Guide’s default behaviors.

.Chapter 5, “Libraries,” beginning on page 55, explains the differences between the libraries, and how to use them.

Chapter 6, “GuideView,” beginning on page 61, describes the GuideView graphical viewer.

Chapter 7, “The Guide_stats Report,” beginning on page 65, describes the Guide statistics report that is generated when you link with the *guide_stats* library.

Chapter 8, “Directive Translation,” beginning on page 73, describes the cray2omp utility that translates Cray cmic$ directives to OpenMP directives.

Appendix A, “Examples,” beginning on page 81, contains code examples with OpenMP directives.

Appendix B, “Timing Guide Constructs,” beginning on page 103, shows the expense associated with using OpenMP directives.

Reference Manual Conventions

To distinguish filenames, commands, variable names, and code examples from the remainder of the text, these terms are printed in `courier` typeface. Command line options are printed in **bold** typeface.

With Guide’s *command line options* and *directives*, one can control a program’s parallelization by providing information to Guide. Some of these command line options and directives require arguments. In their descriptions, **<integer>** indicates an integer number, **<path>** indicates a directory, **<name>** indicates an argument name, **<file>** indicates a filename, **<character>** indicates a single character, and **<string>** indicates a string of characters. For example, **-lines=<integer>** in this user’s guide indicates that an integer needs to be provided in order to change the **-lines** option from the default value to a new value (such as **-lines=0**).

Optional items are denoted with square brackets:

-[no]dlines

The **no** is optional. If **-dlines** is used, *dlines* is turned on. To turn *dlines* off, use **-nodlines**.

To differentiate user input and code examples from descriptive text, they are presented:

```
In Courier typeface, indented where possible.
```

For brevity, throughout this manual, we use Guidef to represent Guidef77 and Guidef90, the Guide drivers for the system FORTRAN 77 and FORTRAN 90 compilers.

Guide On-line

Visit the Guide Home Page at *http://www.kai.com/kpts/guide/* for the latest information on Guide.

Technical Support

KAI spends considerable effort to produce high-quality software; however, if Guide produces a fatal error or incorrect results, please send a copy of the source code, a list of the switches and options used, and as much output and error information as possible to Kuck & Associates (KAI), **guide@kai.com**.

Comments

If there is a way for Guide to provide more meaningful results, messages, or features that would improve usability, let us know. Our goal is to make Guide easy to use as you improve your productivity and the execution speed of your applications. Please send your comments to **guide@kai.com**.

CHAPTER 2

Using Guide

Parallel Processing Model

This section defines general parallel processing terms and explains how different constructs affect parallel code. The user interested in exact semantics is encouraged to consult the OpenMP standard document or the *ANSI X3H5 Parallel Extensions for FORTRAN* document number X3H5/93-SD1-Revision M or contact KAI at *http://www.kai.com/kpts/guide/* or email KAI at **guide@kai.com** for more information.

Overview

After placing OpenMP parallel processing directives in an application, and after the application is processed with Guide and compiled, it can be executed in parallel. As the parallel program begins execution, a single thread begins. This thread is called the base or master thread. The master thread will continue serial processing until it encounters a parallel region. Parallel regions are delineated by the `PARALLEL/END PARALLEL` directive pair.

When the master thread enters a parallel region, a team, or group of threads, is formed. Starting from the beginning of the parallel region, code is replicated (executed by all team members) until a worksharing construct is encountered. The `PDO`, `PSECTIONS`, or `PSINGLE` constructs are defined as worksharing constructs

because they distribute the enclosed work among the members of the current team. encountered. A worksharing construct is only distributed if it occurs dynamically inside of a parallel region. If the worksharing construct occurs lexically inside of the parallel region then it is always executed by distributing the work among the team members. If the worksharing construct is not lexically enclosed by a parallel region (i.e. it is orphaned), then the worksharing construct will be distributed among the team members of the closest enclosing parallel region if one exists, otherwise it will be executed serially.

The PDO/END PDO directive pair controls parallel execution for a DO loop. PSECTIONS/END PSECTIONS directive pair controls parallel execution for arbitrary blocks of sequential code, one section per thread. The PSINGLE/END PSINGLE directive pair defines a section of code where exactly one thread is allowed to execute the code.

Synchronization constructs are CRITICAL/END CRITICAL, ORDERED/END ORDERED, MASTER/END MASTER, ATOMIC, FLUSH and BARRIER. Synchronization can be specified within a parallel region or a worksharing construct with the CRITICAL/END CRITICAL directive pair. Only one thread at a time is allowed to execute the code within this directive pair. Within a PDO or PSECTIONS construct, synchronization can be specified with an ORDERED/END ORDERED directive pair. This directive pair is used in conjunction with a PDO or PSECTIONS construct with the ORDERED clause to impose an order on the execution of a section of code. The MASTER/END MASTER directive pair is another synchronization directive pair that can be used to force execution by the master thread. Another way to specify synchronization is with a BARRIER directive. A BARRIER directive can be used to force all team members to gather at a particular point in code. Each team member that executes a BARRIER waits at the BARRIER until all of the team members have arrived. BARRIERs cannot occur (cause deadlock) within worksharing or synchronization constructs.

When a thread reaches the end of a worksharing construct (an END PDO, END PSECTIONS, or END PSINGLE directive), it must wait until all team members within that construct have completed their work. When all of the work defined by the worksharing construct is completed, the team exits the worksharing construct and continues executing the code that follows the worksharing construct.

At the end of the parallel region, the master thread waits until all the team members have arrived. Then the team is logically disbanded (but may be reused in the next parallel region) and the master thread continues sequentially until it encounters the next parallel region.

Increasing Efficiency

Scheduling options can be selected for worksharing constructs to increase efficiency. Scheduling options specify the way processes are assigned iterations for a loop. These options control the chunk size and load balancing. A `NOWAIT` option can be used to increase efficiency. The `NOWAIT` option allows processes that finish their work to continue executing code. These processes do not wait at the end of the worksharing construct.

Enabling the option **-optimize** can also help increase efficiency. For example, using **-optimize=1** will eliminate unnecessary `BARRIER`s. The default setting for this option is **-optimize=1**.

For your convenience, the following example has been adapted from the *ANSI X3H5 Parallel Extensions for FORTRAN* document.

Figure 2-1 "Pseudo Code of the Parallel Processing Model"

```
program main        ! Begin Serial Execution
                    !
 ...                ! Only the master thread executes
                    !
parallel            ! Begin a Parallel Construct,
                    ! form a team
                    !
    ...             ! This is Replicated Code where each team
    ...             ! member executes the same code
                    !
    psections       ! Begin a Worksharing Construct
                    !
    section         ! One unit of work
       ...          !
    section         ! Another unit of work
       ...          !
    end psections   ! Wait until both units of work complete
                    !
    ...             ! More Replicated Code
                    !
    pdo             ! Begin a Worksharing Construct,
                    ! each iteration is a unit of work
                    !
    ...             ! Work is distributed among the team
                    !
    end pdo nowait  ! End of Worksharing Construct,
                    ! nowait is specified
                    !
    ...             ! More Replicated Code
                    !
    barrier         ! Wait for all team members to arrive
                    !
    ...             ! More Replicated Code
                    !
end parallel        ! End of Parallel Construct, disband team
                    ! continue with serial execution
                    !
 ...                ! Possibly more Parallel Constructs
                    !
end                 ! End serial execution
```

Data Sharing

Data sharing is specified at the start of a parallel region or worksharing constructs using the SHARED and PRIVATE clauses. All variables in the SHARED clause are shared among the members of a team. It is the programmer's responsibility to synchronize access to these variables. All variables in the PRIVATE clause are private to each team member. For the entire parallel region, assuming *t* team members, we have *t+1* copies of all the variables in the PRIVATE clause; one global copy that is active outside parallel regions, and a private copy for each team member. Initialization of PRIVATE variables at the start of a parallel region is the programmer's responsibility, unless the FIRSTPRIVATE clause is specified in which case the PRIVATE copy is initialized from the global copy at the start of the construct at which the FIRSTPRIVATE clause is specified. In general, updating the global copy of a PRIVATE variable at the end of a parallel region is the programmer's responsibility. However the LASTPRIVATE clause of a PDO directive enables updating the global copy from the team member that executed the last iteration of the PDO.

In addition to the SHARED and PRIVATE clauses, entire COMMON blocks can be privatized using the INSTANCE PARALLEL directive along with NEW or COPY NEW directives. For INSTANCE PARALLEL, if a NEW or COPY NEW appears, then there are *t+1* copies of the COMMON block when there are *t* team members. This follows the same model as for PRIVATE variables. If a NEW or COPY NEW is not encountered for an INSTANCE PARALLEL COMMON block, only one copy of the COMMON block exists.

Another method for privatizing COMMON blocks is by using a THREAD PRIVATE directive. For compatibility with Cray TASKCOMMON directives, THREAD PRIVATE blocks always have *t* copies for *t* team members. The master thread uses the global copy as its private copy for the duration of each parallel region.

Using Guide to Develop Parallel Programs

To help users who are familiar with parallel programming, this section contains a high-level overview for using Guide to develop a parallel application. This manual is not intended to be a comprehensive treatment of parallel processing. For more information about parallel processing, consult your favorite parallel processing text.

Analyze

- Profile the program to find out where it spends most of its time. This is the part of the program that needs to be parallelized.
- In this part of the program there are usually nested loops. Locate a loop that has very few cross-iteration dependences. Work through the call tree to do this.

Restructure

- If the loop is parallel, introduce a Guide `PARALLEL DO` directive around this loop.
- In the routine with the `PARALLEL DO`, list the variables that are present in the loop on the `SHARED()`, `PRIVATE()`, `LASTPRIVATE()`, or `FIRSTPRIVATE()` clauses.
- List the `DO` index of the parallel loop as `PRIVATE()`.
- `COMMON` block elements must not be placed on the `PRIVATE()` list if their global scope is to be preserved. The common privatization directives can be used to privatize to a thread the `COMMON` containing those variables with global scope.
- Attempt to remove cross-iteration dependencies by rewriting the algorithm.
- Synchronize the remaining cross-iteration dependences by placing `CRITICAL` directives around the uses and assignments to variables involved in the dependences.
- Any I/O in the `PARALLEL DO` should be synchronized.
- Identify more parallel loops and restructure them.
- If possible merge adjacent `PARALLEL DO`s into a single parallel region with multiple `PDO`s to reduce execution overhead.

Tune

- Guide supports the tuning process by including the *guide_stats* library. The tuning process should include minimizing the sequential code in `CRITICAL` sections and load balancing by using various scheduling options.

For users with parallel FORTRAN 77 programs that already have embedded Cray autotasking directives, a tool is included with Guide to help automate the

job of translating them to KAP/Pro parallel directives. See “Directive Translation” on page 73.

Orphaned Directives

The KAP/Pro Toolset 3.0 introduces an extension to the parallel programming model that dramatically increases the expressiveness of the parallel directives. In addition to being able to specify all of the parallel directives inline in one subroutine, you are now able to partition the directives among many different subroutines so that you are not constrained in your programming style in order to use parallelism.

The example:

```
c$omp parallel private(i) shared(n)
c$omp pdo
        do i = 1, n
           call work(i)
        end do
c$omp end parallel
```

is a common programming idiom for using the PDO worksharing construct to concurrentize the execution of the loop. If we had two such loops we might write:

```
c$omp parallel private(i,j) shared(n)
c$omp pdo
        do i = 1, n
           call some_work(i)
        end do
c$omp pdo
        do j = 1, n
           call more_work(j)
        end do
c$omp end parallel
```

However, programs are sometimes naturally structured by placing each of the major computational sections into it's own subroutine. For example:

```
subroutine phase1
   do i = 1, n
      call some_work(i)
   end do
end

subroutine phase2
   do j = 1, n
      call more_work(j)
   end do
end
```

In KAP/Pro Toolset 3.0, you can parallelize this code in a natural manner, and still maintain all of the benefits of specifying parallelism using KAP/Pro Toolset parallel directives.

```
        ...
c$omp parallel
        call phase1
        call phase2
c$omp end parallel
        ...

        subroutine phase1
c$omp pdo
           do i = 1, n
              call some_work(i)
           end do
        end

        subroutine phase2
c$omp pdo
           do j = 1, n
              call more_work(j)
           end do
        end
```

Notice in this example, the directives specifying the parallelism are divided into three separate subroutines.

A Few Rules About the "Orphaned" Directives

1. An orphaned worksharing construct (PDO/PSECTION/PSINGLE) that is executed outside of a dynamic parallel region will be executed sequentially. In the following example the first call to PHASE0 is executed serially, and the second call is partitioned among the processors on the machine.

```
        ...
        call phase0(10)
c$omp parallel
        call phase0(10)
c$omp end parallel
        ...

        subroutine phase0(n)
c$omp pdo
        do i = 1, n
           call other_work(i)
        end do
        end
```

2. Any collective operation (worksharing construct or barrier) executed inside of a worksharing construct is illegal. For example:

```
    ...
c$omp parallel
c$omp pdo
        do i = 1, n
           call bar
        end do
c$omp end parallel
    ...
        subroutine bar
c$omp barrier
        end
```

3. It is illegal to execute a collective operation (worksharing or barrier) from inside of a synchronization region (CRITICAL/ORDERED).

```
c$omp parallel
c$omp critical
        call test
c$omp end critical
c$omp end parallel
        ...

        subroutine test
c$omp pdo
        do i = 1, n
           call work(i)
        end do
        end
```

4. All structured directives must occur in the same block of the program. For example (PDO/END PDO, CRITICAL/END CRITICAL, PARALLEL/END PARALLEL, ORDERED/END ORDERED, etc.).

5. Private scoping of a variable at a worksharing construct can be specified at the worksharing construct. Shared scoping must be specified at the parallel region. Please consult the OpenMP specification for complete details.

```
        subroutine test
        common /cmndat/ i
c$omp pdo
        do i = 1, n
            call work(i)
        end do
        end
```

CHAPTER 3

OpenMP Directives

Control Directives

Guide uses OpenMP directives to support a single level of parallelism. Each directive begins with `C$OMP`. When a parallel processing directive is continued on subsequent lines, each additional line begins with `C$OMP&`. Several directives must be paired (directive and `END` directive). The same type of directive may not be nested, e.g. a `C$OMP PDO` directive may not appear within the scope of another `PDO` directive within the same subroutine. Please note that items enclosed in square brackets (`[]`) are optional. The syntax of the OpenMP directives accepted by Guide is presented below.

PARALLEL/END PARALLEL

The `C$OMP PARALLEL` and `C$OMP END PARALLEL` directives define the scope of a parallel region.

```
C$OMP PARALLEL
C$OMP& [IF (if_expr)]
C$OMP& [DEFAULT(SHARED|PRIVATE|NONE)]
C$OMP& [SHARED(shared_vars)]
C$OMP& [PRIVATE(private_vars)]
C$OMP& [FIRSTPRIVATE(firstprivate_vars)]
C$OMP& [REDUCTION(operator:reduction_vars)]
C$OMP& [REDUCTION(intrinsic:reduction_vars)]
C$OMP& [COPYIN(copyin_vars)]
C$OMP END PARALLEL
```

The various clauses are described below.

When a parallel region is encountered in the dynamic scope of another parallel region, the parallel regions are each executed using a team of one thread.

PDO/END PDO

The C$OMP PDO directive states that the next statement is an iterative DO loop which will be executed using multiple threads. If the PDO/END PDO directive is encountered in the execution of the program while a parallel region is not active, then the directives do not cause work to be distributed and the entire loop is executed on the thread that encounters this construct.

```
C$OMP PDO
C$OMP& [Scheduling_Options]
C$OMP& [PRIVATE(private_vars)]
C$OMP& [FIRSTPRIVATE(firstprivate_vars)]
C$OMP& [LASTPRIVATE(lastprivate_vars)]
C$OMP& [REDUCTION(operator:reduction_vars)]
C$OMP& [REDUCTION(intrinsic:reduction_vars)]
C$OMP& [ORDERED]
[C$OMP END PDO [NOWAIT]]
```

Using the C$OMP END PDO directive is optional. Without the NOWAIT clause, the C$OMP PDO directive will hold all threads that reach the end of the PDO loop until all iterations of that DO loop have been completed. Therefore, the C$OMP END PDO directive without the NOWAIT clause has no effect, and the end of the C$OMP PDO directive is marked by the end of the DO loop. Specifying the C$OMP END PDO NOWAIT directive allows early finishing threads to execute code that appears after the C$OMP END PDO NOWAIT directive. If the C$OMP END PDO directive is used, no statements or directives may appear between the last statement of the DO loop and the C$OMP END PDO directive.

PSECTIONS/END PSECTIONS

The C$OMP PSECTIONS and C$OMP END PSECTIONS directive pair delineates sections of code that will be executed on different threads. Each parallel section between the C$OMP PSECTIONS and C$OMP END PSECTIONS directives must be preceded by the C$OMP SECTION directive. If the PSECTIONS/ END PSECTIONS directive is encountered in the execution of the program while a parallel region is not active then the directives do not cause work to be distributed and all the psections are executed on the thread that encounters this construct.

```
C$OMP PSECTIONS
C$OMP& [PRIVATE(private_vars)]
C$OMP& [FIRSTPRIVATE(firstprivate_vars)]
C$OMP& [LASTPRIVATE(lastprivate_vars)]
C$OMP& [REDUCTION(operator:reduction_vars)]
C$OMP& [REDUCTION(intrinsic:reduction_vars)]
C$OMP& [ORDERED]
[C$OMP SECTION]
      [code A]
C$OMP SECTION
      [code B]
C$OMP END PSECTIONS [NOWAIT]
```

This example executes code A and code B in parallel on two threads.

PSINGLE/END PSINGLE

The C$OMP PSINGLE and C$OMP END PSINGLE directives define a section of code where exactly one thread is allowed to execute the code.

```
C$OMP PSINGLE
C$OMP& [PRIVATE(private_vars)]
C$OMP& [FIRSTPRIVATE(firstprivate_vars)]
C$OMP& [LASTPRIVATE(lastprivate_vars)]
C$OMP& [REDUCTION(operator:reduction_vars)]
C$OMP& [REDUCTION(intrinsic:reduction_vars)]
C$OMP END PSINGLE [NOWAIT]
```

The first arriving thread is allowed to execute the C$OMP PSINGLE directive. Other threads must wait until the master thread has finished the section of code, then they continue executing with the statement after the C$OMP END PSINGLE directive. If the NOWAIT clause is present, then the other threads do not wait until the master thread has executed the section of code and immediately skip the section of code.

PARALLEL DO/END PARALLEL DO

The C$OMP PARALLEL DO and C$OMP END PARALLEL DO directives are a short form syntax for a parallel region with a single PDO. The parallel loop is enclosed by the PARALLEL DO/END PARALLEL DO pair. The PARALELL DO/END PARALLEL DO directive is used in place of the PARALLEL/END PARALLEL and PDO/ END PDO directive pairs. If these directives are encountered while a parallel region is already active, then these directives are executed by a team of one thread and the entire loop is executed by each thread that encounters it.

```
C$OMP PARALLEL DO
C$OMP& [IF (if_expr)]
C$OMP& [Scheduling_Options]
C$OMP& [DEFAULT(SHARED|PRIVATE|NONE)
C$OMP& [SHARED(shared_vars)
C$OMP& [PRIVATE(private_vars)]
C$OMP& [FIRSTPRIVATE(firstprivate_vars)]
C$OMP& [LASTPRIVATE(lastprivate_vars)]
C$OMP& [COPYIN(copyin_vars)]
C$OMP& [ORDERED]
[C$OMP END PARALLEL DO]
```

The above example is equivalent to the following:

```
C$OMP PARALLEL
C$OMP& [IF (if_expr)]
C$OMP& [DEFAULT(SHARED|PRIVATE|NONE)
C$OMP& [SHARED(shared_vars)]
C$OMP& [PRIVATE(private_vars)]
C$OMP& [COPYIN(copyin_vars)]
C$OMP PDO
C$OMP& [Scheduling_Options]
C$OMP& [FIRSTPRIVATE(firstprivate_vars)]
C$OMP& [LASTPRIVATE(lastprivate_vars)]
C$OMP& [REDUCTION(operator:reduction_vars)]
C$OMP& [REDUCTION(intrinsic:reduction_vars)]
C$OMP& [ORDERED]
C$OMP END PDO NOWAIT
C$OMP END PARALLEL
```

PARALLEL SECTIONS/END PARALLEL SECTIONS

The C$OMP PARALLEL SECTIONS/C$OMP END PARALLEL SECTIONS directives are a short form for a parallel region with a single psection. The PSECTION is enclosed by the PARALLEL SECTIONS/END PARALLEL SECTIONS directive pair. The PARALLEL SECTIONS/END PARALLEL SECTIONS directive pair is used in place of the PARALLEL/END PARALLEL and PSECTIONS/END PSECTIONS directive pairs. The psection between the C$OMP PARALLEL SECTIONS and C$OMP END PARALLEL SECTIONS directives must be preceded by the C$OMP SECTION directive. If the C$OMP PARALLEL SECTIONS and C$OMP END PARALLEL SECTIONS directives are encountered in the execution of the program while a parallel region is already active, then these directives are executed by a team of one thread and the entire construct is executed by each thread that encounters it.

```
C$OMP PARALLEL SECTIONS
C$OMP& [IF (if_expr)]
C$OMP& [DEFAULT(SHARED|PRIVATE|NONE)
C$OMP& [SHARED(shared_vars)
C$OMP& [PRIVATE(private_vars)]
C$OMP& [FIRSTPRIVATE(firstprivate_vars)]
C$OMP& [LASTPRIVATE(lastprivate_vars)]
C$OMP& [ORDERED]
C$OMP& [REDUCTION(operator:reduction_vars)]
C$OMP& [REDUCTION(intrinsic:reduction_vars)]
C$OMP& [COPYIN(copyin_vars)]
[C$OMP SECTION]
      [code A]
C$OMP SECTION
      [code B]
C$OMP END PARALLEL SECTIONS
```

The above example is equivalent to the following:

```
C$OMP PARALLEL
C$OMP& [IF (if_expr)]
C$OMP& [DEFAULT(SHARED|PRIVATE|NONE)
C$OMP& [SHARED(shared_vars)]
C$OMP& [PRIVATE(private_vars)]
C$OMP& [FIRSTPRIVATE(firstprivate_vars)]
C$OMP& [REDUCTION(operator:reduction_vars)]
C$OMP& [REDUCTION(intrinsic:reduction_vars)]
C$OMP& [COPYIN(copyin_vars)]
C$OMP PSECTIONS
C$OMP& [LASTPRIVATE(lastprivate_vars)]
C$OMP& [ORDERED]
[C$OMP SECTION]
      [code A]
C$OMP SECTION
      [code B]
C$OMP END PSECTIONS NOWAIT
C$OMP END PARALLEL
```

IF(if_expr)

When the logical `IF (if_expr)` expression exists, the `IF` clause is evaluated. If the logical expression evaluates to `.FALSE.`, then all of the code in the parallel region is executed by a team of one thread. If the logical expression evaluates to `.TRUE.`, then the coden the parallel region may be executed by a team of multiple threads. When the `IF` clause is not present, it is assumed to be `.TRUE.`

DEFAULT(SHARED|PRIVATE|NONE)
SHARED(shared_vars)
PRIVATE(private_vars)

The `SHARED ( )` and `PRIVATE ( )` lists in the parallel region state the explicit forms of data sharing among the threads that execute the parallel code. When distinct threads should reference the same variable or array, place the variable in the `SHARED` list. When distinct threads can reference distinct instances of variables or arrays, place the variable in the `PRIVATE` list.

The `PRIVATE` clause is allowed on `PARALLEL`, `PDO` and `PSECTIONS` directives. The `DEFAULT` and `SHARED` clauses are only allowed on `PARALLEL` directives.

When a variable that is referenced in the lexical extent of a parallel region is not listed on any list, its default sharing classification is determined based upon the DEFAULT clause. DEFAULT(SHARED) causes unlisted variables to be SHARED, DEFAULT(PRIVATE) causes unlisted variables to be PRIVATE, and DEFAULT(NONE) causes unlisted variables to generate an error. The only exceptions to the DEFAULT rules are loop control variables (loop indices) and f90 statement scoped entities, which are PRIVATE unless explicitly overridden. The default is DEFAULT(SHARED).

FIRSTPRIVATE (firstprivate_vars)

A variable or array in a FIRSTPRIVATE() list is copied from the variable or array of the same name in the enclosing context by each team member before execution of the construct.

The FIRSTPRIVATE clause is allowed on PARALLEL, PDO, PSECTIONS and PSINGLE directives.

LASTPRIVATE (lastprivate_vars)

A variable or array in a LASTPRIVATE() list is copied back into the variable or array of the same name in the enclosing context before the execution terminates for the team member that executes the final iteration of the index set for a PDO, the last lexical SECTION of a PSECTIONS construct, or the code enclosed by a PSINGLE, as appropriate. If the loop is executed and the LASTPRIVATE variable is not written in the final iteration of the index set for a PDO or the last lexical SECTION in a PSECTIONS construct, then the value of the shared variable is undefined.

The LASTPRIVATE clause is allowed on PDO, PSECTIONS, and PSINGLE directives.

REDUCTION(operator:reduction_vars)
REDUCTION(intrinsic:reduction_vars)

A variable or array element in the REDUCTION(reduction_vars) list is treated as a reduction by creating a PRIVATE temporary for that variable and computing into the original variable after the end of the construct using a CRITICAL section. The allowed operators are +, -, *, .AND., .OR., .EQV., and .NEQV. The allowed intrinsics are MAX, MIN, IAND, IOR, and IEOR.

The REDUCTION clause is allowed on PARALLEL, PDO, PSECTIONS, and PSINGLE directives.

```
c$omp parallel do
c$omp& shared (a,n)
c$omp& private (i)
c$omp& reduction (max:maxa)
        do i = 1, n
            maxa = max ( maxa, a(i) )
        enddo
c$omp end parallel do
```

The above example is equivalent to the following:

```
c$omp parallel
c$omp& shared (a,n,maxa,maxa_orig)
c$omp& private (i,maxa_local)
       maxa_local = minimum_valu_for_type_of_maxa
c$omp pdo
       do i = 1, n
           maxa_local = max ( maxa_local, a(i) )
       enddo
c$omp end pdo nowait
c$omp critical
      maxa = max (maxa, maxa_local)
c$omp end critical
c$omp end parallel
```

COPYIN (copyin_vars)

The COPYIN( ) clause applies only to THREAD PRIVATE common blocks and their members. This clause provides a mechanism to copy the master thread's values of the listed variables to the other members of the team at the start of a parallel region.

The COPYIN directive is only allowed on PARALLEL directives.

Common Privatization

Directives Globally addressable storage that is private to each thread in a computation is useful as a place to store information needed to coordinate between different subroutines executed by one thread of a parallel region.

This notion is supported by using the following two types of private commons:

6. INSTANCE PARALLEL as defined by ANSI X3H5
7. THREAD PRIVATE as a migration feature for Cray's TASKCOMMON

INSTANCE PARALLEL

INSTANCE PARALLEL private COMMON blocks separate the declaration point from the allocation point. The allocation is specified via an explicit NEW or COPY NEW directive. The absence of an allocation directive causes all the threads to reference the same storage.

THREAD PRIVATE

THREAD PRIVATE causes an implicit NEW in every parallel region. COPY NEW of an THREAD PRIVATE COMMON block is provided as a migration feature for SGI's COPYIN. Because NEW is implicit, an THREAD PRIVATE COMMON block is always private.

Declaring Private Commons

Private COMMON blocks are declared by the INSTANCE PARALLEL or THREAD PRIVATE directives. The INSTANCE PARALLEL directive creates a copy of each specified COMMON block for each thread, only when a NEW is encountered. The THREAD PRIVATE directive assigns each specified COMMON block to the master thread and creates a copy of the COMMON block for each additional thread. The syntax for the two directives is as follows:

```
C$OMP INSTANCE PARALLEL [/CMN1/ [,/CMN2/] ]
C$OMP THREAD PRIVATE [/CMN1/ [,/CMN2/] ]
```

These directives are placed in the declaration section. If a COMMON block appears in an INSTANCE PARALLEL or THREAD PRIVATE directive somewhere, the COMMON block must appear, respectively, in an INSTANCE PARALLEL or THREAD PRIVATE directive everywhere. A variable in a private COMMON block (INSTANCE PARALLEL or THREAD PRIVATE) cannot appear in a DATA statement.

Allocating Private Commons

The NEW and COPY NEW directives allocate the named INSTANCE PARALLEL COMMON blocks inside parallel regions. For THREAD PRIVATE COMMON blocks, this allocation is implicit to allow compatibility with Cray's TASKCOMMON. These directives must occur either in the declaration section for subroutines which are called in the parallel region, or immediately following the PARALLEL statement when COMMON is used in the subroutine containing the parallel region.

When a thread inside a parallel region encounters a NEW directive, the named private COMMON blocks are allocated and initialized if they have not already been allocated and initialized. If the thread has previously allocated and initialized the COMMON block for a different parallel region, that space is simply reused.

Whether a COMMON block is private to each thread for a given parallel region depends upon whether a NEW or COPY NEW directive for that COMMON block has been seen either inside the parallel region or in any of the routines called from that parallel region. If no NEW directive has been seen, the private COMMON block acts as a regular shared COMMON block.

The syntax for using the NEW directive is as follows:

```
C$OMP NEW [/CMN1/ [,/CMN2/] ]
```

The COPY NEW directive is similar to the NEW directive. COPY NEW allocates and initializes private COMMON blocks if they have not already been allocated and initialized. However, COPY NEW causes the values in the original COMMON to be copied into the private COMMON in each parallel region entered. When NEW is specified, the COMMON is initialized only for the first parallel region encountered.

When COPY NEW is used with the THREAD PRIVATE directive, it must occur in the same subroutine immediately after the PARALLEL statement.

The syntax for using the COPY NEW directive is as follows:

```
C$OMP COPY NEW [/CMN1/ [,/CMN2/] ]
```

NOTE: Behavior is undefined when COMMON blocks are allocated and initialized with both a NEW and a COPY NEW within a single parallel region. If any

thread executes a NEW or COPY NEW for a COMMON block, every thread must execute a NEW or COPY NEW, respectively, for the same COMMON block.

Synchronization Directives

CRITICAL/END CRITICAL

The C$OMP CRITICAL and C$OMP END CRITICAL directives define the scope of a critical section. Only one thread at a time is allowed inside the critical section. The name has global scope. Two CRITICAL directives with the same name are automatically mutually exclusively. All unnamed CRITICAL sections are assumed to map to the same name.

```
C$OMP CRITICAL [(name)]
C$OMP END CRITICAL [(name)]
```

ORDERED/END ORDERED

The C$OMP ORDERED and C$OMP END ORDERED directives define the scope of an ordered section. Only one thread at a time is allowed inside an ordered section of the same name.

```
C$OMP ORDERED [(name)]
C$OMP END ORDERED [(name)]
```

The optional variable can be used to name an ordered section. Ordered sections are allowed either within or outside of parallel regions, but when they occur lexically outside of a parallel region, they must be unnamed.

In addition, the ordered section must be dynamically enclosed in a PDO loop with the ORDERED scheduling modifier. It is an error to use this directive pair when not in the dynamic scope of a PDO with an ORDERED modifier.

The semantics of an ordered section are defined in terms of the loop's iteration space. The threads executing the iteration space are granted permission to enter the ordered section in the same order as the iterations are executed in the sequential version of the loop.

Each ordered section with a given name must only be entered once or not at all during the execution of a PDO iteration.

A deadlock situation will occur if these rules are not observed.

MASTER/END MASTER

The section of code enclosed in a C$OMP MASTER/C$OMP END MASTER pair is executed by the master thread of the team. Other threads of the team skip the enclosed section of code and continue execution. Note that there is no implied BARRIER on entry to or exit from the master section.

ATOMIC

This directive ensures atomic update of a location in memory that may otherwise be exposed to the possibility of multiple, simultaneous, writing threads. This directive only applies to the immediately following statement which must have one of the following forms:

```
x = x <op> <expr>
x = <expr> <op> x
x = <intrinsic> (x, <expr>)
x = <intrinsic> (<expr>, x)
```

where

x	is an intrinsic typed variable
<expr>	is a scalar expression that does not reference x
<intrinsic>	is one of MAX,MIN,IAND,IOR,IEOR
<op>	is one of +, -, *, /, .AND., .OR., .EQV., .NEQV.

Correct use of this directive requires that if an object is updated using this directive, than all references to that object must use this directive.

FLUSH[(string)]

This directive causes thread visible variables to be written back to memory and is provided for users who wish to write their own synchronization directly through shared memory. The optional string is a list which may be used to specify varaibles that need to be flushed. If the list is absent, all variables are flushed to memory.

BARRIER

BARRIER directives are used to gather all team members to a particular point in the code. BARRIERs force team members to wait at that point in the code until all of the team members encounter that BARRIER. BARRIER directives are not allowed inside of worksharing constructs.

```
C$OMP BARRIER
```

Scheduling Options

Scheduling options are used to specify the iteration dispatch mechanism for each parallel loop. Scheduling options can be specified in the following three ways:

1. Command Line Options
2. Directives
3. Environment Variables

Command line options and directives are used to specify the scheduling mechanism when the source file is being processed by Guide. For loops that are processed with the RUNTIME scheduling mechanism, described below, scheduling can be changed at run time with environment variables.

Loop scheduling is dependent on the scheduling mechanism and the chunk parameter. The table below describes what each scheduling option does. Assume that the loop has the following attributes: *l* iterations, *p* threads are being used to execute the loop, and *n* is an integer greater than 0 specifying the chunk size.

Table 3-1 Scheduling Options

Scheduling Designator	Chunk	Meaning
s	n	Static scheduling with a chunk size of *n*. *n* iterations are dispatched statically to each thread (repeat until *l* iterations have been dispatched). If *n* is missing, this is the same as static even scheduling.
e	ignored	Static even scheduling. The chunk size has no effect here. *l/p* iterations are dispatched statically to each thread. The same as static with a chunk size of *l/p*.
i	ignored	Static interleaved scheduling. The chunk size has no effect here. Thread *i* is statically dispatched iterations *i*, *i+p*, *i+2p*, ...
d	n	Dynamic scheduling with a chunk size of *n*. *n* iterations are dispatched dynamically to each thread.
g	n	Guided scheduling with a minimum chunk size of *n*. An exponentially decreasing number of iterations are dispatched dynamically to each thread. At least *n* iterations are dispatched every time except the last.
t	n	Trapezoidal scheduling with minimum chunk size of *n*. A linearly decreasing number of iterations are dispatched dynamically to each thread. At least *n* iterations are dispatched every time except the last.
r	ignored	Runtime scheduling is specified when the scheduling is to be determined via the `OMP_SCHEDULE` environment variable.

Scheduling Options Using Directives

The list below shows the syntax for specifying scheduling options with the `PDO` and `PARALLEL DO` directives.

```
SCHEDULE(STATIC      [,<chunk_size>] )
SCHEDULE(DYNAMIC     [,<chunk_size>] )
SCHEDULE(GUIDED      [,<chunk_size>] )
SCHEDULE(TRAPEZOIDAL [,<chunk_size>] )
SCHEDULE(INTERLEAVED)
SCHEDULE(RUNTIME)
```

If <chunk_size> is not specified, it is assumed to be 1 for DYNAMIC, GUIDED and TRAPEZOIDAL, and assumed to be missing for STATIC. See Table 3-1 on page 28 for a complete description of the scheduling options. The ORDERED scheduling modifier is necessary for PDO and PARALLEL DO loops which contain ordered sections.

Scheduling Options Using Environment Variables

Scheduling options may also be specified at run time when the run time scheduling type has been specified at Guide processing time with the OMP_SCHEDULE environment variable. The syntax for the OMP_SCHEDULE environment variable is as follows:

```
OMP_SCHEDULE = <string>[,<integer>]
```

Where <string> is one of STATIC, INTERLEAVED, DYNAMIC, GUIDED, or TRAPEZOIDAL, and the optional <integer> parameter is a chunk size for the dispatch method. For a complete description of the scheduling options, see Table 3-1.

Environment Variables

Guide uses some environment variables which may need to be set before running Guide.

KMP_LIBRARY=<string>

This variable selects the Guide library. The three available options are:

ser	serial
tur	turnaround
thr	throughput

See Chapter 5, “Libraries,” beginning on page 55 for more information about the Guide libraries.

KMP_STACKSIZE=<integer>[,<character>]

This variable specifies the number of bytes, kilobytes, or megabytes that will be allocated for each parallel thread to use as its private stack. Use the optional suffix **k**, **b**, or **m** to specify bytes, kilobytes, or megabytes. The default is **1m** or one megabyte. This number may be too small if many local variables are used in the parallel regions, or the parallel region calls subroutines that have many local variables.

KMP_STATSFILE=<file>

When this variable is used, in conjunction with the *guide_stats* library, the statistics report is written to the specified file. The default file name for the statistics report is `statsfile`.

Three metacharacter sequences are defined that can be included in the file name and expanded at runtime to provide unique context sensitive information as part of the file name. These three metacharacter sequences are:

`%H`: This expands into the hostname of the machine running the parallel program.

`%I`: This expands into a unique numeric identifier for this execution of the program. It is the process identifier of the program.

`%P`: Is the value of the `OMP_PARALLEL` environment variable which determines the number of threads that are created by the parallel program.

LD_LIBRARY_PATH=<path>

This variable is used to specify an alternate path for the Guide run-time libraries. This variable may need to be set when you run your application if you compile with shared objects or use dynamic linking.

OMP_DYNAMIC=<boolean>

The `OMP_DYNAMIC` environment variable enables dynamic adjustment of the number of threads between parallel regions. A value of `TRUE` for `<boolean>` enables dynamic adjustment, whereas a value of `FALSE` disables any change in the number of threads. If dynamic adjustment is enabled, the number of threads may be adjusted only at the beginning of each parallel region. No threads are created or destroyed during the execution of the parallel region.

OMP_NUM_THREADS=<integer>

The OMP_NUM_THREADS environment variable is used to dynamically control the number of threads. The <integer> is a positive number. Performance of parallel programs usually degrades when the requested parallelism is larger than the number of physical processors.

OMP_SCHEDULE=<string>[,<integer>]

The OMP_SCHEDULE environment variable controls the schedule type and chunk size for PDO constructs with a SCHEDULE(RUNTIME) clause or those with no clause if the command line scheduling designator is set to r. The schedule type is given by <string>, which is one of STATIC, INTERLEAVED, DYNAMIC, GUIDED, or TRAPEZOIDAL, and the optional chunk size is given by <integer> for those scheduling types which allow a chunk size. See Table 3-1, “Scheduling Options,” on page 28.

CHAPTER 4

The Guide Drivers

About Guidef77 and Guidef90

The Guide drivers, Guidef77 and Guidef90, replace the system FORTRAN 77 and FORTRAN 90 compilers on the command line and integrate Guide instrumentation and the compile/link step into one command line. In scripts and Makefiles, replacing the standard compiler (typically `f77` or `f90`) with `guidef77` or `guidef90`, respectively, will execute the necessary C preprocessor, Guide, and compiler commands automatically.

In addition to all of the command line options accepted by the Fortran compiler, the Guide drivers accept prefixed forms of all Guide options as well as driver-specific options. An absence of command line arguments causes the drivers to emit a usage message.

For brevity, throughout this manual, we will use Guidef to represent Guidef77 and Guidef90, the Guide drivers for the system FORTRAN 77 and FORTRAN 90 compilers.

Using the Drivers

To run Guide, use the following command line:

```
guidef -WG,<option>[,<option>,…] filename <compiler_options>
```

where *filename* is the input file to Guide. The **-WG** driver option specifies additional Guide arguments. For example, to change the default scheduling designator and the chunk size from the command line, use **-WG,-scheduling=d,-chunk=4**. Multiple options must be separated with a comma.

If a list of FORTRAN source files is specified on the Guidef command line without the **-c** compiler option, and if Guide fails to process any of the files, then the driver will compile (but not link) all successfully processed files.

Instrumented source files (Guide output files) are removed by default after successful Guide instrumentation and compilation. There are, however, four instances where output files are not removed:

- When Guide fails to process a FORTRAN source file, the output files from each failing source file are *not* removed, while the output files from successfully processed files *are* removed.
- If the compile/link step fails for any of the source files Guide successfully instruments, none of the output files are removed.
- If you specify **-WGkeep**, none of the output files are removed.
- If the compiler debug flag (e.g. **-g**) is specified on the command line, none of the output files are removed (**-WGkeep** is implied). **-WGnokeep** will cause output files to be deleted even in the presence of **-g** or **-WGkeep**.

Guide output files consist of the name of the original source file with the prefix `G_` added to the beginning of the filename. The compiler is given the names of these output files and creates object files with a `G_` prefix. The driver then removes this prefix from the object files. For example, if Guide processes an input file named `foo.f`, it would create an output file named `G_foo.f`. Guidef would then pass this name on to the compiler. If an object file is created by the compilation process, then it would be named `G_foo.o` by the compiler, and Guidef would then rename this object file `foo.o`.

FORTRAN files with capitalized suffixes (e.g. `filename.F`) are first passed through the C preprocessor before Guide is invoked. The C preprocessor will create files with a `cppG_` suffix (e.g. `cppG_filename.F`). As mentioned above, Guide will create an output file whose name is based on the original source file name.

Driver Options

The Guidef driver recognizes the FORTRAN compiler options and several driver-specific options. If Guidef fails to recognize a command line option, it is ignored and passed directly to the compiler.

Default driver-specific options are located in a file named `.guidefrc` in either the current directory or your home directory. All driver-specific options listed in this chapter can be placed in the `.guidefrc` file. These options must be separated by white space or new lines. All instances of **<file>** in these options must contain the full path to the new executable, which should include the filename of the executable.

In the following descriptions, **<integer>** indicates an integer number, **<path>** indicates a directory, **<name>** indicates an argument name, **<file>** indicates a filename, **<character>** indicates a single character, and **<string>** indicates a string of characters.

Displaying the Driver Usage Message

The **-h** option displays the usage message for the driver. This flag will cause Guidef to abort execution.

Displaying All Command Lines

The **-v** option causes the driver to display all command lines executed. This flag is passed on to the compiler.

Suppressing Guidef Warnings

Use the **-w** option to suppress mild Guidef warnings. This flag is passed on to the compiler.

Driver-Specific Options

The following driver-specific options are *not* passed on to the FORTRAN compiler.

WG,guide_option_1[[[,guide_option_2],guide_option_3],...]

This flag prefixes any specified Guide option(s). Multiple Guide options may be appended using commas as delimiters.

For instance, to pass the **-o=1** and **-real=8** options to Guide, the appropriate Guidef driver option would be **-G,-o=1,-real=8**.

WGcompiler=<path>

The **-WGcompiler** option allows you to specify an alternate **<path>** for the FORTRAN compiler executable. This option can also be specified with the **-WGftn**, **-WGfortran**, **-WGf77**, and **-WGf90** arguments.

WGcpp=<path>

The **-WGcpp** option allows you to specify an alternate path for the C preprocessor executable.

WGf77=<path>

See the definition of the **-WGcompiler** option above.

WGf90=<file>

See the definition of the **-WGcompiler** option above.

WGfortran=<path>

See the definition of the **-WGcompiler** option above.

WGftn=<path>

See the definition of the **-WGcompiler** option above.

WGkeep

If **-WGkeep** is stated, output files generated by Guide and temporary files created by the C preprocessor will not be unlinked after compilation. By default, these files are automatically removed after a successful compilation.

WGkeepcpp

If **-WGkeepcpp** is stated, output files generated by the preprocessor will not be removed after a successful compilation.

WGld=<file>

The **-WGld** option allows you to specify an alternate path for the linker executable.

WGlibpath=<path>

This option specifies an alternate **<path>** in which to search for the Guide libraries. For dynamic/shared compilation, be sure to add **<path>** to your `LD_LIBRARY_PATH` environment variable before running an executable created by Guide.

WGlink=<path>

The **-WGlink** option allows you to specify an alternate path for the linker executable.

WGnocpp

-WGnocpp prevents Guidef from calling the C preprocessor for FORTRAN source files with the `.F` suffix. By default, the driver will automatically call the C preprocessor for all files with the `.F` suffix.

WGnokeep

Use **-WGnokeep** to force output and temporary C preprocessor files to be removed. The presence of this flag overrides any instance of **-WGkeep** on the command line, including the **-WGkeep** implied from **-g** and **-WGonly**.

WGnoprocess

Guide will not process any of the specified source files if **-WGnoprocess** is present on the command line. This flag can be used to compile source code that has already been processed by Guide.

WGnorc

This flag will turn off any driver-specific options that were found in your `$HOME/.guidefrc` file. Since this option will also cancel any driver-specific options that precede it, **-WGnorc** should be the first driver-specific option to appear on the command line.

WGonly

When **-WGonly** is used, Guide will process the source code in all specified source files, but neither the compiler nor linker will be executed. Like **-WGkeep**, this option retains output files and temporary files generated by Guide and the C preprocessor.

WGpath=<path>

-WGpath specifies an alternate path to the Guide executable.

WGprefix=<string>

The **-WGprefix** option changes the prefix string added to the Guide and preprocessor output files. For instance, if you specify the following:

```
guidef -WGprefix=qqq -WGcpp -WGkeep file1.F
```

the results are `cppqqqfile1.f` and `qqqfile1.f` instead of `G_file1.f` and `cppG_file1.f`.

WGsrcdir

-WGsrcdir specifies that the preprocessor and Guide output files should be in the same directory as the source file.

WGversion

The Guidef driver displays its internal version number and other information to `stderr` when **-WGversion** is used. Using this option will abort execution.

Guide Options

The **-WG** driver option specifies additional Guide arguments. To state a Guide option, the long (full) name, short name, or any portion of the long name, starting from the beginning, which uniquely identifies the option may be used. Multiple options must be separated by a comma. For example, to change the scheduling designator and the chunk size, use **-WG,-scheduling=d,-chunk=4**.

Guide Options Functional Categories

Table 4-1 lists the Guide options. These options are grouped into the following functional categories:

General Optimization

These options control large classes of optimizations.

Input-Output

These options affect the input file selection and output file naming, placement, and characteristics.

Listing

These options control listing information that is provided about the transformations and optimizations performed.

Advanced Optimization

These options customize and fine-tune the optimizer for maximum performance.

FORTRAN Dialect

These options help customize for particular dialects of FORTRAN.

Hardware

These options inform Guide about your target architecture. The default settings have been chosen to take advantage of the architecture of the target machine. In most cases, you will not need to change the default settings.

Directive Recognition

These options enable or disable recognition and processing of directives that can be placed in the code.

Guide Options Table

In Table 4-1, Guide options are listed alphabetically within each functional category. The default settings are also listed. Guide options that require an argument list the default argument. For more information on Guide options, see the section “Guide Options Alphabetic Listing” on page 43.

Table 4-1 Guide Options

Long Name	Short Name	Default Setting
General Optimization:		
optimize=<integer>	o=<integer>	1
roundoff=<integer>	r=<integer>	0
scalaropt=<integer>	so=<integer>	0
Input-Output:		
cmp[=<file>]	cmp[=<file>]	G_<file>
input=<file>	i=<file>	<file>
[no]list=<file>	[n]l=<file>	none
Listing:		
lines=<integer>	ln=<integer>	55
listoptions=<string>	lo=<string>	k
suppress=<string>	su=<string>	no suppress
Advanced Optimization:		
[no]assume	[n]as=<string>	cel
[no]concurrentize	[no]conc	noconcurrentize
minconcurrent=<integer>	mc=<integer>	1000

Table 4-1 Guide Options (Continued)

Long Name	Short Name	Default Setting
FORTRAN Dialect:		
alignmax=<integer>	alignmax=<integer>	16
[no]blank_padding	[n]bp	blank_padding
[no]case	[n]case	nocase
[no]datasave	[n]ds	datasave
[no]dlines	[n]dl	nodlines
[no]free	[no]free	nofree
include=<path>	inc=<path>	no include
integer=<integer>	int=<integer>	4
logical=<integer>	log=<integer>	4
[no]onetrip	[n]1	noonetrip
real=<integer>	rl=<integer>	4
[no]recursion	[n]rc	norecursion
save=<string>	sv=<string>	manual
scan=<integer>	scan=<integer>	72
syntax=<string>	sy=<string>	no syntax
[no]type	[n]ty	notype
Directive Recognition:		
[no]directives=<string>	[n]dr=<string>	p
[no]ignoreoptions	[n]ig	noignoreoptions
[no]openmpcc_lines	[no]ompcc_lines	ompcc_lines
default=<string>	default=<string>	shared
Hardware:		
heaplimit=<integer>	heap=<integer>	500
Scheduling:		
chunk=<integer>	chk=<integer>	1
scheduling=<character>	schd=<character>	e

The **-listoptions=k** command line option can be used to determine what your default settings are.

Guide Options Alphabetic Listing

This section lists the Guide options that can be specified by using the **-WG** driver option. To make these options easy to find, they are listed alphabetically rather than by functional category. The headings in the following sections list the full and short names for each option.

alignmax=<integer>

This option selects the maximum size datatype that will be naturally aligned. The integer argument represents the boundary size in bytes. The default is **-alignmax=16**.

assume=<string> or a=<string>
noassume or nas

The **-assume** option instructs Guide to make certain global assumptions about the program being processed. The **-assume** option switch values are the following:

- **a** Different subroutine or function parameters may refer to the same object.
- **b** Array subscripts may go outside the declared bounds.
- **c** Constants used in subroutine or function calls will be placed in temporary variables.
- **e** `EQUIVALENCE` statements may cause different names to refer to the same memory location.
- **l** Unless Guide can prove they are unneeded, Guide must insert code to assign to variables in transformed loops the values they would have had after the original serial loop.

The default value is **-assume=cel**. To disable all the above assumptions, specify **-noassume** on the command line.

blank_padding or bp
noblank_padding or nbp

The **-blank_padding** option instructs Guide to pad the input line with blanks. This option is on by default.

case or case
nocase or ncase

The **-case** option instructs Guide to distinguish between upper and lowercase in identifier names. The default **-nocase** instructs Guide to be case-insensitive in variable names.

When Guide inserts or modifies lines in a program, it usually creates the new code in capital letters. The **-case** option requires Guide to preserve the original case of variables in the new code. Making Guide case-sensitive can be important. If, for example, there is a variable named **n** and a variable named **N** in the original source code, Guide will change the **n** to a **N** when it optimizes the code unless **-case** is specified, causing a conflict between two different variables which now have the same name.

chunk=<integer> or chk=<integer>

This option specifies a parameter for parallel loop scheduling, and is to be used in conjunction with the **-scheduling** option. Together, the **-scheduling** and the **-chunk** options establish default scheduling for all the parallel loops for this Guide run. Individual loops can override this default scheduling mechanism using the scheduling options on the `PARALLEL DO` or `PDO` directive. The default chunk size is 1. See “Scheduling Options” on page 27 for descriptions of the **-chunk** options.

cmp[=<file>]

The **-cmp=<file>** option instructs Guide to place the optimized FORTRAN program in a specified transformed program file (the *compile* file). The default name of the FORTRAN output file is derived from the input filename by Guide adding `G_` to the beginning of the filename and changing the extension to `.f`. If **-cmp=<file>** is specified, the FORTRAN output file is written to the specified file. If **-cmp** is specified, then the output is written to standard output.

concurrentize, conc
noconcurrentize, noconc

Guide uses the **-concurrentize** switch to enable parallelizaion of loops derived from array syntax only. The **-concurrentize** option also implies **-scalaropt=1**.

datasave or ds
nodatasave or nds

The **-datasave** option instructs Guide to treat local variables in a subroutine or function which appear in DATA statements as if they were also in SAVE statements. That is, their values will be retained between invocations of the subroutine or function. This is the practice of many commercial FORTRAN compilers, and **-datasave** is on by default. This choice affects certain optimizations performed by Guide.

The negative option, **-nodatasave**, complies with the FORTRAN standard. See also the description of the **-save** command line option.

directives=p or dr=p
nodirectives or ndr

The **-directives=p** or **-dr=p** option enables parallel programming directives. This option is on by default. To disable parallel programming directives, use **-nodirectives** or **-ndr**.

dlines or dl
nodlines or ndl

The **-dlines** option instructs Guide to treat a D in column 1 as a space character. The rest of that line will then be parsed as a normal FORTRAN statement. By default, Guide treats these lines as comments. This option is useful for the inclusion or exclusion of debugging lines.

In the following example, the first (default) case shows that the D line is ignored:

```
      do 10 i = 1,n
         a (i) = b (i)
d        write (*,*) a (i)
   10 continue
```

becomes

```
      do 10 i=1,n
         a(i) = b(i)
10    continue
```

But when **-dlines** is specified, Guide sees a `WRITE` statement:

```
      do 10 i=1,n
         a(i) = b(i)
         write (*, *) a(i)
10    continue
```

free
nofree

The **-free** option removes the standard column restrictions for FORTRAN source code. Continuation lines are indicated with an "&" as the last character of the continued line and as the first character of the continuation line.

The **-free** option is off by default, and the usual FORTRAN 77 conventions apply.

heaplimit=<integer> or heap=<integer>

Guide may require large amounts of memory in order to process your source code. The **-heaplimit** option specifies the maximum size in megabytes that the Guide heap can grow. If this limit is breached, Guide will stop processing the source code and try to exit gracefully with an "out of memory" error message. The default size is 500 megabytes.

If the **-heaplimit** setting is greater than the amount of available memory, Guide may run out of memory before it reaches the **-heaplimit**.

Guide relies upon the operating system to tell it that the OS has run out of memory before that problem occurs. Some operating systems kill Guide without first telling Guide that there is insufficient memory. In that case, Guide may stop processing the code and exit in an undefined manner. Using **-heaplimit** makes a graceful exit more likely.

ignoreoptions or ig
noignoreoptions or nig

The **-ignoreoptions** option directs Guide to ignore any **c*$*options** or ***$*options** line that may appear at the top of the input file. Normally, Guide reads the **c*$*options** or ***$*options** instruction for further command line options, as explained in the description of the **c*$*options** line below.

Setting **-noignoreoptions** directs Guide to acknowledge the **c*$*options** line in the source program. That is, Guide will accept the command line options given on the **c*$*options** line.

include=<directory> or inc=<directory>

By default, Guide looks only in the current directory to locate files specified in `INCLUDE` statements. The **-include** option allows an alternate directory to be specified for locating those files. An `INCLUDE` file whose name does not begin with a slash (/) is sought first in the directory containing the file being processed, then in the directory named in the **-include** option.

input=<file> or i=<file>

When running Guide in stand-alone mode on UNIX systems, simply enter the source filename on the command line. This option is available for special circumstances and for compatibility with other operating systems.

On UNIX systems, if the **-input** option is specified without a filename, Guide will read its source from standard input and write the transformed code to standard output. In this case, no listing file will be generated unless a filename is explicitly provided with the **-list** option.

integer=<integer> or int=<integer>

This option specifies a size in bytes, `N`, for the default size of `INTEGER` variables. When `N=2` or `4`, take `INTEGER*N` as the default `INTEGER` type. When **-integer=0**, Guide uses the ordinary default length for `INTEGER` variables. The default is **-integer=4**.

lines=<integer> or ln=<integer>

The **-lines** option enables Guide's listing to be paginated for printing in different formats. The number of lines per page on the listing may be changed using the **-lines** option. The setting **-lines=0** instructs Guide to paginate only at subroutine boundaries. The default setting is **-lines=55**.

list[=<file>] or l=[<file>]
nolist or nl

The **-list** option informs Guide where to place the listing file. Guide derives the default name of the listing file from the input filename by adding `G_` to the beginning of the filename and changing the extension to `.lst`. If only this option is stated, then the listing file is written to the specified file. To disable generation of the listing file, enter **-nolist** on the command line. The default is **-nolist**.

listoptions=<string> or lo=<string>

The **-listoptions** option tells Guide what optional information to include in the listing, transformed code, and error files.

Any of the following information can be selected:

Value	Prints
k	Guide options used, printed at the end of each program unit
o	Original source program annotated listing
t	Transformed program annotated listing

To disable all of the above switches and produce no listing file, enter **-nolist** on the command line. The default value is **-listoptions=k**.

logical=<integer> or log=<integer>

This option specifies a size in bytes, `N`, for the default size of `LOGICAL` variables. When `N=1`, `2`, or `4`, take `LOGICAL*N` as the default `LOGICAL` type. The value assigned to **-logical** should be equal to the value assigned to **-integer**. The default is **-logical=4**.

minconcurrent=<integer> or mc=<integer>

Executing a loop in parallel incurs overhead which varies with different systems. If a loop has little work, parallel execution may be slower than serial execution because of the overhead. However, beyond a certain level, performance gain may be obtained through parallel execution. This level is passed to Guide with the **-minconcurrent** option.

The range of values for this option is all numbers greater than or equal to 0. The higher the **-minconcurrent** value, the larger the loop body must be (have more iterations, more statements, or both) to run concurrently.

At compilation time, Guide estimates the amount of computation inside a loop by multiplying the loop iteration count by the sum of the nonindex operands/results and the nonassignment operators and compares this value with the **-minconcurrent** value. If the estimated amount of work is greater than the **-minconcurrent** value, Guide generates concurrent code for the loop. Otherwise, it leaves the loop serial. If the `DO` loop bounds are known at compilation time, the exact iteration count can be computed. However, if the `DO` loop bounds are unknown, Guide generates an `IF` expression in the directive. This is interpreted by the compiler as a request to generate two loops, one concurrentized and one left serial, and an `IF-THEN-ELSE` to make a run time check to decide whether or not to execute the loop in parallel. (This case is called a two-version loop.)

To disable the generation of two-version loops throughout the program, use the command line option **-minconcurrent=0**.

The **-minconcurrent** option only applies to parallel loops created by Guide from array syntax. The **-minconcurrent** option implies the **-concurrentize** switch.

onetrip or 1
noonetrip or n1

The **-onetrip** option allows *one-trip* `DO` loops to be specified. Many pre-FORTRAN 77 compilers implemented `DO` loops which would always have at least one iteration, even if the initial value of the loop control variable was higher than the final value. This option informs Guide that the program being processed contains loops which need the *one-trip* feature. This option is off by default.

optimize=<integer> or o=<integer>

The **-optimize** option sets the base optimization and analysis level.

The meaning of the different optimization levels is as follows:

`0` Guide performs no optimizations on parallel directives.

`1` Guide optimizes parallel directives.

The default is **-optimize=1**.

real=<integer> or rl=<integer>

This option specifies a size in bytes, for the default size of `REAL` variables. When the **-real** option is present, Guide uses `REAL*<integer>` as the default `REAL` type.

The default value is **-real=4**.

NOTE: This option merely informs Guide about the default `REAL` size; it does NOT ask Guide to convert from `REAL*4` to `REAL*8`.

recursion or rc
norecursion or nrc

The **-recursion** option informs Guide that subroutines and functions in the source program may be called recursively (that is, a subroutine or function calls itself, or it calls another routine which calls it). Recursion affects storage allocation decisions and the interpretation of the **-save** option. This option is off by default.

The **-recursion** option must be in force in each recursive routine that Guide processes or unsafe transformations could result.

roundoff=<string> or r=<string>

The **-roundoff** option specifies the amount of change from serial roundoff error that is tolerable in the program. If an arithmetic reduction is accumulated in a different order in the processed program than it was in the original program, then the roundoff error is accumulated differently, and the final result may differ from that of the original program. In most cases, the difference is insignificant.

However, if the source program is numerically unstable or if it requires extreme precision, certain restructuring transformations performed by Guide must be disabled in order to obtain exactly the same results as those obtained in the original program.

The **-roundoff** option has the values 0 or 1.

The **-/roundoff** levels are defined as follows:

0 Guide allows no roundoff-changing transformations. When **-roundoff=0**, the transformed code is in strict conformance to the FORTRAN standard. When **-roundoff>0**, the standards are relaxed. This is the default.

1 Guide enables expression simplification and code floating.

save=<string> or sv=<string>

The **-save** option instructs Guide on how to handle the storage class of local scalar variables. In particular, Guide can be instructed to perform *live variable analysis* to help Guide decide whether to save the value of a local scalar variable between invocations of a function or a routine by generating a SAVE statement. Guide can also be instructed to treat the default storage class of all local scalar variables as either AUTOMATIC or STATIC. In any case, Guide will not delete or ignore a hand coded SAVE statement.

There are four possible settings for the **-save** option:

Specifying **-save=all** (**-save=a**) tells Guide not to perform live variable analysis. However, all variables local to a function or a routine and COMMON blocks will be treated as if they are saved. The **-save=all** option is not affected by the **-[no]recursion** option.

The default **-save=manual** (**-save=m**) tells Guide not to perform live variable analysis. Guide assumes that the necessary SAVE statements have been inserted into the code and it performs no corresponding analysis of its own. Hand coded SAVE statements are assumed to be correct and sufficient. The **-save=manual** setting is not affected by the **-recursion** option.

Specifying **-save=manual_adjust** (**-save=ma**) instructs Guide to perform live variable analysis. The effect of **-save=manual_adjust** depends on the **-[no]recursion** setting:

With **-norecursion**, SAVE statements will be added for variables which are used before being defined on at least one path from one entry point to the routine.

With **-recursion**, SAVE statements will be added for variables which are used before being defined on all paths from all entry points to the routine.

Specifying **-save=all_adjust** (**-save=aa**) instructs Guide to perform live variable analysis. The effect of **-save=all_adjust** depends on the **-[no]recursion** setting:

With **-norecursion**, treat all local variables as saved, except those which are defined before use in all paths from all entry points and which are not in hand coded SAVE statements.

With **-recursion**, this is the same as **-save=all**.

Saving local variables may be required for correct execution, but can restrict Guide optimizations. Accordingly, **-save=ma** should be used with caution.

scalaropt=<integer> or so=<integer>

The **-scalaropt** option sets the level of scalar transformations performed. The allowed values and their meanings are:

0 No scalar optimizations are performed. This is the default.

1 Forward substitution and backward elimination are performed.

scan=<integer> or scan=<integer>

The **-scan** option allows the length of the FORTRAN input lines to be set. Guide will ignore (by treating as a comment) characters on columns beyond the value of the **-scan** option. The value must be one of 72, 120, or 132. The default is **-scan=72**.

scheduling=<character> or schd=<character>

The **-scheduling** option tells the compiler what kind of scheduling to use for loop iterations on a multiprocessor machine. This option is used in conjunction with the **-chunk** option. See “Scheduling Options” on page 27 for a description of the **-scheduling** options.

suppress=<string> or su=<string>

The **-suppress** option disables the printing of individual classes of Guide messages. These message classes range from syntax warning and error messages to messages about the optimizations performed. The allowed values of the **-suppress** option are as follows:

Value	Disables
d	Data Dependence messages
e	Syntax Error messages
i	Informational messages
n	Not Optimized messages
q	Questions
s	Standardized messages
w	Syntax Warning messages

The default instructs Guide to report all message types listed above.

syntax=<string> or sy=<string>

The **-syntax** option directs Guide to check for compliance with certain syntactic rules. If you are familiar with a different implementation of FORTRAN, then using a dialect switch can prevent a construct from being translated differently than expected. The default is to accept a superset FORTRAN 77 and FORTRAN 90 except for `MODULES` and `INTERNAL PROCEDURES`.

With **-syntax=a**, Guide checks for strict compliance with the ANSI FORTRAN 77/90 standard. Warning and error messages are issued for syntax which does not conform to the standard.

Note: With **-syntax=a**, syntax errors are issued for array references without subscripts.

With **-syntax=v**, Guide accepts the extensions and interpretations of Digital or DEC FORTRAN 77/90.

type or ty
notype or nty

The **-type** option instructs Guide to issue error messages for variables not explicitly typed. The **-notype** default suppresses this checking.

c*$*options Line

When a source file should always be run with the same command line options, the first line of the file may be used to specify them. The format of this line is as follows:

```
c*$*options option[=value] [option[=value]]...
```

The **c*$*options** (or ***$*options**) must appear in columns 1-11 (or 1-10) with a character space between this command and the options that follow.

Only the first line may be used for **c*$*options**. Short or long option names may be used on this line.

Options of the form **-option=<name>** (e.g., **-cmp** or **-inline**) cannot be specified on the **c*$*options** line of the source file. These options may be specified on the command line only.

If conflicting options are specified on the command line and on the **c*$*options** line, the **c*$*options** line takes precedence. If additional options are specified on the **c*$*options** line, these are used in addition to those specified on the command line.

If the command line option **-ignoreoptions** is set, any **c*$*options** line in the source file is treated as a comment.

CHAPTER 5

Libraries

Selecting a Library

Guide supplies two libraries, an end-user library and a development library. The end-user library is *guide*. It should be used for normal or performance-critical runs on applications that have already been tuned. The development library is *guide_stats*. It provides performance information about the code, but it slightly degrades performance. It should be used to tune the performance of applications. Both the *guide* and *guide_stats* libraries contain the serial, turnaround, and throughput libraries described below (these libraries are selected by using the `KMP_LIBRARY` environment variable, see “KMP_LIBRARY=<string>” on page 29).

Serial

The serial library forces parallel applications to be run on a single processor.

Turnaround

In a dedicated (batch or single user) parallel environment where all of the processors for a program are exclusively allocated to the program for its entire run, it is most important to effectively utilize all of the processors all of the time. The turnaround library is designed to keep all of the processors active and involved in the parallel computation to minimize the execution time of a single job.

NOTE: Avoid over-allocating system resources. This occurs if either too many processors have been specified, or if too few processors are available at run time. If system resources are over-allocated, this library will cause poor performance. The throughput library should be used if this occurs.

Throughput

In a multi-user environment where the load on the parallel machine is not constant or where the job stream is not predictable, it may be better to design and tune for throughput. This minimizes the total time to run multiple jobs simultaneously, or to mix sequential and parallel jobs on the same machine.

The throughput library is designed to let the program be aware of it's environment (i.e. the system load) and to adjust the resources used to produce efficient execution in a dynamic environment.

The Guide_stats Library

The *guide_stats* library is designed to provide the user with detailed statistics about a program's execution. These statistics help the user to "see inside" the program to analyze performance bottlenecks and to make parallel performance predictions. With this information, it is possible to modify the program (or the execution environment) to make more efficient use of the parallel machine.

When a program is compiled with Guidef, linked with the *guide_stats* library, and executed, statistics are output to the file specified with the `KMP_STATSFILE` environment variable (the default file name *guide_stats* is used if this environment variable is not specified). In addition, running with the *guide_stats* library enables additional runtime checks that may aid in program debugging. When using the *guide_stats* library, make sure that the main program, and any subroutines that cause program termination, are compiled by using Guidef.

The resulting statistics are most easily viewed and analyzed by using GuideView, discussed in Chapter 6, “GuideView,” beginning on page 61.

Linking the Libraries

Guide uses the *guide* library by default. To use the *guide_stats* library, use the **-WGstats** command line option to Guidef. For example, the following command line can be used to compile a source file with the *guide_stats* library:

```
guidef -WGstats source.f
```

External Routines

The following library routines can be used for low-level debugging to verify that the library code and the user’s application are functioning as intended.

The use of these routines is discouraged; using them requires that the application be linked with the *guide* library, even when the code is being executed sequentially. In addition, using these routines makes validating the program with Assure more difficult.

In most cases, directives can be used in place of these routines. For example, thread-private storage should be implemented by using the `PRIVATE()` clause of the `PARALLEL` directive or the `TASKCOMMON` directive, rather than by explicit expansion and indexing. Appendix A, “Examples,” beginning on page 81, contains examples of coding styles that avoid the use of these routines.

mppbeg(), mppend()

These routines are not necessary if a program is written entirely in FORTRAN and compiled by using Guidef. In a mixed language environment, however, the program’s entry or exit points might be written in a different language. In this case, the user must ensure that `mppbeg()` is called at the beginning of the main program (if the main program is not compiled by using Guidef) and that `mppend()` is called at all points that cause program termination (if those routines are not compiled by using Guidef).

Calling these routines from another language requires knowledge of the cross-language calling standards on your platform. The following convention is typically used with the C language. An underscore is appended to names that are declared in FORTRAN subroutines. Thus, a main program in C that can be used with Guide FORTRAN code might look like:

```
void
main( int argc, char *argv[] )
{
    extern mppbeg_(), mppend_();
    mppbeg_();
    work();
    mppend_();
    exit(0);
}
```

The calls to `mppbeg()` and `mppend()` must occur when the program is executing sequentially, not when a parallel region is active.

omp_get_max_threads()

This routine returns the maximum number of threads that are available on the parallel machine. The returned value is a positive integer, and is equal to the value of the `OMP_NUM_THREADS` environment variable, if set.

omp_get_num_procs()

This routine returns the maximum number of processors that are available on the parallel machine. The returned value is a positive integer.

omp_get_num_threads()

This routine returns the number of threads that are being used in the current parallel region. The returned value is a positive integer.

NOTE: The number of threads used may change from one parallel region to the next. When designing parallel programs it is best to not introduce assumptions that the number of threads is constant across different instances of parallel regions. The number of threads may increase of decrease between parallel regions, but will never exceed the `OMP_NUM_THREADS` environment variable value.

omp_get_thread_num()

This routine returns the thread id of the calling thread. The returned value is an integer between zero and `omp_get_num_threads()-1`.

CHAPTER 6

GuideView

Introduction

GuideView is a graphical tool that presents a window into the performance details of a program's parallel execution. Performance anomalies can be understood at a glance with the intuitive, color coded display of parallel performance bottlenecks.

GuideView graphically illustrates what each processor is doing at various levels of detail by using a hierarchical summary. Statistical data are collapsed into relevant summaries which focus on the actions to be taken.

Using GuideView

GuideView uses as input a statistics file that was output when a Guide instrumented program was run. An optional file with library overhead information and an optional configuration file can also be provided. The syntax for invoking GuideView is as follows:

```
guideview [<guideview_options>] <file>
```

The *guideview_options* represents optional GuideView options, and the *file* argument is the name of the statistics file created by a Guide run that used the *guide_stats* library (see Chapter 6).

The GuideView browser looks for a configuration file named `.guideviewrc` or `guide.ini` when it starts up. It first looks in the current directory, then in your home directory, and then in each directory in turn that appears in your `$CLASSPATH` environment variable setting. Using this file you can configure several options that will control fonts, colors, window sizes, window locations, line numbering, tab expansion in source, and other features of the GUI.

An example of an initialization file is provided with your Guide installation. This example file contains comments that explain the meaning and usage of the supported options. If Guide is installed in directory `X` on your machine, the example of an initialization file that explains the options available in it will be in `X/guide35/class/example.guideviewrc`.

The default location for this example initialization file is in the directory `/usr/local/KAI/guide/class`. If the default location is different from the installed location, then a symbolic link will be created from the default location to the installed location if the default location is writable at install time. The easiest way to use this file is to copy it and then edit the copy as needed, uncommenting lines you want and/or setting the options to values you prefer or need.

Detailed information about GuideView's operation can be found under the help menu of the GUI.

GuideView Options

mhz=<integer>

The **-mhz=<integer>** option denotes the MHz rate for the machine used for calculating statistics.

ovh=<file>

The **-ovh=<file>** specifies an overheads file for the input statistics file. There are small overheads that exist in the GuideView library. These overheads can be measured in terms of the number of cycles for each library call or event. You can override the default values to get more accurate overhead values for your machine by using the **-ovh=<file>** option to create a file that contains machine-specific values.

An example overheads file is provided with your Guide installation. This example file contains comments that explain the meaning and usage of the supported options. If Guide was installed in directory `X` on your machine, this example file resides in `X/guide/class/guide.ovh`.

WJ,[java_option]

The GuideView GUI is implemented in JAVA. The **-WJ** flag prefixes any specified JAVA option. The JAVA options are passed to the JAVA interpreter.

Any valid JAVA interpreter option may be used; however, the options listed below may be particularly beneficial when used with GuideView to enhance the performance of the GUI:

JAVA Options

The **-WJ** flag must prefix any specified JAVA option. For example, to pass the **-ms5m** option to the JAVA interpreter, use **-WJ,-ms5m**.

ms<integer>[{k,m}]

The **-ms** option specifies how much memory is allocated for the heap when the interpreter starts up. The initial memory is specified either in bytes, kilobytes (with the suffix `k`), or megabytes (with the suffix `m`). For example, to specify one megabyte, use **-ms1m**.

mx<integer>[{k,m}]

The **-mx** option specifies the maximum heap size the interpreter will use for dynamically allocated objects. The maximum heap size is specified either in bytes, kilobytes (with the suffix `k`), or megabytes (with the suffix `m`). For example, to specify two megabytes, use **-mx2m**.

CHAPTER 7

The Guide_stats Report

The *guide_stats* library can be used to gain insight into how a parallel program is behaving — or misbehaving. The *guide_stats* library contains monitoring software that collects statistics during the execution of a Guide instrumented program. When a program is compiled with Guidef, linked with the *guide_stats* library, and executed, a statistics report file is produced in a file named by the `KMP_STATSFILE` environment variable. See Chapter 5, “Libraries,” beginning on page 55 for more details on using the *guide_stats* library.

This chapter documents the format and content of the statistics report produced by the *guide_stats* library. The easiest way to view and interpret these statistics is by using GuideView, described in Chapter 6, “GuideView,” beginning on page 61.

The use of the *guide_stats* library does introduce some overhead into the execution of a parallel program, so the *guide_stats* library should not be used for production runs that require peak performance. The statistics report, however, reflects the proportion of time that would be spent in each part of a parallel program if that program had been run without using the *guide_stats* library.

In the following description, output from the library will be shown in courier font. If an entire line of the statistics report contains zeros, that line is not displayed. Also, the following examples show whole program statistics, but the actual statistics report will contain information for each individual parallel region. The whole program statistics are totals over all of the parallel regions.

The statistics report provides timings in real (wall clock) time, which includes both system time (for system calls) and user time (for computation and Guide library calls).

It is important to consider the MAX, MIN range of values for each statistic. If there is a large difference between MAX and MIN, it may indicate a load imbalance due to poor scheduling. If static scheduling is being used, consider reducing the chunk size or employing dynamic scheduling.

Settings

```
KMP_STATSFILE  = stats
KMP_PARALLEL   = 4
KMP_STACKSIZE  = 1048576
KMP_SCHEDULING = <default>
KMP_CHUNK      = <default>
KMP_SPINLOCKS  = <default>
```

The statistics report starts with a description of the environment variables set during the run. See “Environment Variables” on page 29 for a description of these variables.

Parallel Region

```
whole program time spent in parallel region (seconds):
              MAX    AVG    MIN    #00    #01    #02    #03
 real time   12.3   11.8   11.5   12.3   11.5   11.7   11.7
```

This section lists the time spent in parallel regions for each of the processors this run was made with. (Processors are number consecutively starting at #00.)

Synchronized Code

```
whole program time spent in synchronized code (seconds):
              MAX    AVG    MIN    #00    #01    #02    #03
 real time    0.0    0.0    0.0    0.0    0.0    0.0    0.0
```

Time is accumulated in synchronized code after a thread has acquired a lock and before it releases the lock. Attempt to reduce the size of CRITICAL sections, ORDERED sections, and PSINGLE regions as much as possible. Even if the time spent in these regions is small, reducing their size may provide insight into how to eliminate them entirely. Eliminating synchronized code can save time spent waiting at locks as well. Keep in mind that using dynamic scheduling causes implicit lock acquisition, and therefore accumulates time in synchronized code.

Synchronization

There are three classes of synchronization overhead reported: locks, barriers, and join barriers.

Locks

```
whole program time spent in waiting at locks (seconds):
              MAX    AVG    MIN    #00    #01    #02    #03
 real time    0.5    0.4    0.3    0.3    0.5    0.3    0.3
```

Time is accumulated in locks whenever a lock is acquired, either explicitly, with the use of CRITICAL section, ORDERED section, or PSINGLE directives, or implicitly, with the use of the Guide library (for example, implicit locking occurs when using dynamic scheduling). A large proportion of parallel region time spent waiting for locks indicates contention for lock resources. If many CRITICAL sections, ORDERED sections, or PSINGLE regions are used, consider naming each region separately, or try to merge regions if the time spent in the associated synchronized code is small.

Barriers

```
whole program time spent in waiting at barriers (seconds):
              MAX    AVG    MIN    #00    #01    #02    #03
 real time    1.2    1.0    0.6    0.6    1.2    1.2    1.1
```

Time is accumulated here when waiting at barriers for the BARRIER directives.

Join Barriers

```
whole program time spent in waiting at join barriers (seconds):
              MAX    AVG    MIN    #00    #01    #02    #03
 real time    0.8    0.3    0.0    0.8    0.0    0.0    0.4
```

Time is accumulated here when waiting at implicit barriers. These implicit barriers occur at the end of parallel regions and at other places in the Guide library.

Average Statistics

```
whole program average statistics (seconds):
            Total Parallel     Lock  Barrier     Sync  Joining
 real time   39.5     11.8      0.4      1.0      0.0      0.3

whole program estimated parallelism:
 63.0 to 64.6 percent parallel
```

Whole program statistics estimate the amount of parallelism obtained during a given parallel execution. This estimate allows the success of your parallelization efforts to be measured. The estimate is computed by dividing the `Parallel` time, which is the average of the time spent in parallel regions across all processors, by the `Total` time, with correction factors for `Lock`, `Barrier`, and `Sync` time. Two estimates are given; the low parallelism estimate assumes high lock overhead, and the high parallelism estimate assumes low lock overhead.

Performance Projections

```
whole program speedup projections (estimated):
                 #01  #02  #03  #04  #05  #06  #07  #08
 63.0% parallel  1.0  1.5  1.7  1.9  2.0  2.1  2.2  2.2
 64.6% parallel  1.0  1.5  1.8  1.9  2.1  2.2  2.2  2.3
```

The `%Parallel` time is used to estimate the speedup that would be obtained if more processors were available.

Amdahl's Law is used:

$$\text{Speedup} \leq \frac{\text{Total Time}}{\text{Serial Time} + (\text{Parallel Time} / \text{Processors})}$$

This number can be optimistic because parallel overhead may not scale linearly with the number of processors used. If these speedup numbers are considered insufficient, despite this built-in optimism, explore finding more parallelism in your program.

Event Counts

The event counts section lists the total number of parallel processing events that occurred on each processor. This information can be useful for debugging the program. If more or fewer events of a certain type occur than expected, it may indicate a bug in the program. If too much time is being spent in parallel overhead functions such as waiting at barriers, it may be either that too many of these operations are being used or that the operations themselves are taking too long due to system load. Compare the time spent in the function to the number of events in those types of functions to the estimate the time spent per event to help decide which is the case.

Program Start/Finish

```
whole program event counts (zero event counts not shown):
```

	TOTAL	MAX	AVG	MIN	#00	#01	#02	#03
Initialize	1	1	0	0	1	0	0	0
Terminate	1	1	0	0	1	0	0	0

These events mark the beginning and end of the program as a whole for statistics initialization and reporting. They should occur only once and should be executed by the first processor.

Internal Checks

	TOTAL	MAX	AVG	MIN	#00	#01	#02	#03
Check Active	36	36	9	0	36	0	0	0
Check Task	94382	39994	23595	16509	39994	19776	18103	16509
Check Stack	0	0	0	0	0	0	0	0

These are internal checks on program status. `Check Active` tests whether a thread is running in parallel. `Check Task` requests the thread's unique identifier. `Check Stack` determines whether there is sufficient stack space to continue execution.

Fork

	TOTAL	MAX	AVG	MIN	#00	#01	#02	#03
Fork	36	36	9	0	36	0	0	0
Fork Join	36	36	9	0	36	0	0	0

Fork counts the entries to parallel regions. These should be executed by thread `#00`.

Private Commons with INSTANCE PARALLEL or THREAD PRIVATE

	TOTAL	MAX	AVG	MIN	#00	#01	#02	#03
InstPar qAlloc	0	0	0	0	0	0	0	0
InstPar qCopy	0	0	0	0	0	0	0	0
InstPar qSetup	0	0	0	0	0	0	0	0

These are counts of internal events when `INSTANCE PARALLEL` is used for declaring `COMMON` blocks. `InstPar qAlloc` is a count of `COMMON` block allocations. `InstPar qCopy` is a count of times that `COMMON` blocks are copied and allocated. `InstPar qSetup` is a count of times that `COMMON` blocks are set up and declared.

	TOTAL	MAX	AVG	MIN	#00	#01	#02	#03
InstTsk qAlloc	0	0	0	0	0	0	0	0
InstTsk qCopy	0	0	0	0	0	0	0	0

These are counts of internal events when THREAD PRIVATE is used for declaring COMMON blocks. InstTsk qAlloc is a count of COMMON block allocations. InstTsk qCopy is a count of times that COMMON blocks are copied and allocated.

	TOTAL	MAX	AVG	MIN	#00	#01	#02	#03
InstTsk tAlloc	160878	44659	40219	37285	38061	44659	40873	37285
InstPar tCopy	0	0	0	0	0	0	0	0
InstPar tSetup	0	0	0	0	0	0	0	0
InstTsk tAlloc	0	0	0	0	0	0	0	0
InstTsk tCopy	0	0	0	0	0	0	0	0

These are counts of internal events very similar to those above. These are slightly more efficient and Guide will use them when possible.

Dynamic Scheduling

	TOTAL	MAX	AVG	MIN	#00	#01	#02	#03
Loop Setup	136	34	34	34	34	34	34	34
Loop Dispatch	4514	1214	1128	1048	1102	1214	1150	1048

These two events indicate the number of dynamic scheduling operations that have been performed. Loop Setup counts the entries to a loop. Loop dispatch counts the number of times additional iterations are scheduled.

Synchronization Events

	TOTAL	MAX	AVG	MIN	#00	#01	#02	#03
Barrier	272	68	68	68	68	68	68	68
Enter Critical	4530	1218	1132	1052	1106	1218	1154	1052
Exit Critical	4530	1218	1132	1052	1106	1218	1154	1052
Begin Single	0	0	0	0	0	0	0	0
End Single	0	0	0	0	0	0	0	0

These events indicate synchronization events that have occurred. Barrier indicates BARRIERs, Enter Critical and Exit Critical indicate CRITICAL sections, and Begin Single and End Single indicate PSINGLEs.

Internal Synchronization Events

	TOTAL	MAX	AVG	MIN	#00	#01	#02	#03
Variable Sync	0	0	0	0	0	0	0	0

These are synchronization events internal to the Guide runtime library.

Routine Events

	TOTAL	MAX	AVG	MIN	#00	#01	#02	#03
Task ID	94382	39994	23595	16509	39994	19776	18103	16509
Max CPUs	1	1	0	0	1	0	0	0
Num Threads	0	0	0	0	0	0	0	0
Max Threads	0	0	0	0	0	0	0	0

This is a count of the number of calls to external routines. The `Task ID` section refers to the `omp_get_thread_num()` routine. `Max CPUs` refers to the `omp_get_num_procs()` routine. `Num Threads` refers to the `omp_get_num_threads()` routine, and `Max Threads` refers to the `omp_get_max_threads()` routine.

Library Calls

	TOTAL	MAX	AVG	MIN	#00	#01	#02	#03
Region Name	6	6	1	0	6	0	0	0

This is a count of the number of calls to the library to register the source line number and file corresponding to a particular parallel region. This is used by the statistics library to give context sensitive information for the timers and events.

Internal Events

	TOTAL	MAX	AVG	MIN	#00	#01	#02	#03
Yield	0	0	0	0	0	0	0	0
Sleep	0	0	0	0	0	0	0	0

These are internal events. High numbers for `Yield` or `Sleep` indicate that the program is not working as efficiently as it could. This may be due to load imbalance, lock contention, or over-allocation of system resources. Try to obtain more `Yields` than `Sleeps` on an efficiently running system by reducing the number of processors requested with the `OMP_NUM_THREADS` environment variable or by reducing or eliminating critical sections. A null call is basically an empty function.

CHAPTER 8

Directive Translation

KAP/Pro Parallel Directive to OpenMP Directive Translator

Programs which have been parallelized with KAP/Pro Toolset directives can be used as the basis for a port to the new OpenMP version of Guide. The `kpts2omp.pl` program will help translate KAP/Pro Parallel directives into OpenMP directives that Guide accepts.

The `kpts2omp.pl` program accepts as an argument the name of a file with KAP/Pro Toolset directives. The translated file is written to `stdout` with OpenMP directives added. The `stdout` can be redirected to capture the translated file. Any directives or constructs that cannot be handled automatically cause diagnostics to be added inline in the translated output. The `stderr` output contains a synopsis of the diagnostics.

The `kpts2omp.pl` translation is a totally automatic process because all of the functionality provided by KAP/Pro Toolset directives is available in the KAP/Pro Toolset implementation of OpenMP directives.

Table 8-1, “kpts2omp.pl Translator Options,” below lists the options that are available when running `kpts2omp.pl`.

Table 8-1 kpts2omp.pl Translator Options

Option	Description
`-[hH?]`	print usage info
`-i`	`ifdef` mode, generates ‘`#ifdef _OPENMP/#endif`’ around directives
`-I`	disables `ifdef` mode (default setting)
`-o`	original directives included in output
`-O`	original directives not included in output (default setting)
`-t <num>`	number of spaces for continuation directives (`0 <= num <= 8`, `default = 4`)
`-v`	verbose mode, give messages about likely errors (default setting)
`-V`	disables verbose messages

NOTE: `Perl` must be installed on the system to use `kpts2omp.pl`.

Cray Directive to OpenMP Directive Translator

Programs which have been parallelized with Cray directives can be used as the basis for a port to Guide. The `cray2omp.pl` program will help translate Cray Autotasking directives into OpenMP directives that Guide accepts. It is assumed that the Cray program with Autotasking directives has been ported to work on the target machine and compiler in serial mode.

The `cray2omp.pl` program accepts as an argument the name of a file with Cray Autotasking directives. The translated file is written to `stdout` with OpenMP directives added. The `stdout` can be redirected to capture the translated file. Any directives or constructs that cannot be handled automatically cause diagnostics to be added inline in the translated output. The `stderr` output contains a synopsis of the diagnostics.

The `cray2omp.pl` translation is not a totally automatic process because of some semantic differences between the two directive sets. However, this translation performs a majority of the work required for migration, and most programs

will not require manual intervention. If manual intervention is required, searching for "cray2omp" in the output will lead to places where cray2omp.pl had trouble performing translations automatically.

Table 8-2, "cray2omp.pl Translator Options," below lists the options that are available when running cray2omp.pl.

Table 8-2 cray2omp.pl Translator Options

Option	Description
-[hH?]	print usage info
-i	ifdef mode, generates '#ifdef _OPENMP/#endif' around directives
-I	disables ifdef mode (default setting)
-o	original directives included in output (default setting)
-O	original directives not included in output
-t <num>	number of spaces for continuation directives (0 <= num <= 8, default = 4)
-v	verbose mode, give messages about likely errors (default setting)
-V	disables verbose messages

Table 8-3, "Cray to OpenMP Translations," below lists the cray2omp.pl translations that are performed. Many of the directives in the table have optional clauses that are translated by cray2omp.pl when possible. A diagnostic is produced when there is not an equivalent OpenMP directive.

Table 8-3 Cray to OpenMP Translations

Cray	OpenMP
cmic$ taskcommon tcb	c$omp threadprivate (/tcb/)
cdir$ taskcommon tcb	c$omp threadprivate (/tcb/)
cdir$ ivdep	*$* assert no recurrence
cdir$ no recurrence	*$* assert no recurrence
cmic$ guard	c$omp critical
cmic$ end guard	c$omp end critical
cmic$ parallel	c$omp parallel

Table 8-3 Cray to OpenMP Translations (Continued)

Cray	OpenMP
`First cmic$ case`	`c$omp psections` `c$omp section`
`Subsequent cmic$ case`	`c$omp section`
`cmic$ endcase`	`c$omp end psections`
`cmic$ do parallel`	`c$omp pdo`
`cmic$ enddo`	`c$omp barrier`
`cmic$ doall`	`c$omp parallel do`
`single`	`schedule(dynamic)`
`guided`	`schedule(guided,64)`
`vector`	`schedule(guided,64)`
`guided(n)`	`schedule(guided,n)`
`chunksize(n)`	`schedule(dynamic,n)`

The following directives are not directly translatable into OpenMP syntax:

```
cmic$ process
cmic$ also process
cmic$ end process
cmic$ stop all process
cmic$ do global
cmic$ continue
cmic$ getcpus
cmic$ numcpus
cmic$ relcpus
cmic$ soft exit
cmic$ micro
```

NOTE: `Perl` must be installed on the system to use `cray2omp.pl`.

Cray TASKCOMMON as opposed to OpenMP THREAD PRIVATE

The tools provided with Guide perform a semi-automatic translation of Cray FORTRAN prallel directives into OpenMP directives. However, some hand editing of the resulting program may be necessary.

Cray `taskcommon` also has a semantic feature that is not supported in the OpenMP syntax. Individual elements of a `taskcommon` can be placed in the private list of a `parallel do`.

In the following example, the scalar elements of `taskcommon /tcb1/`, which are `x` and `y`, are on the private list but the large array `z` is not. With OpenMP, one could use the `copyin` clause to achieve this effect. Since all the elements of `taskcommon /tcb2/` are on the private list, the entire `/tcb2/` can be placed on the `copyin` clause.

For example, this Cray version

```
cmic$ taskcommon tcb1, tcb2
    common /tcb1/ x,y,z(10000)
    common /tcb2/ a,b,c

    x = 1
    y = 2
cmic$ do parallel private(i,x,y,a,b,c) shared(n)
    do i = 1, n
       ...
    enddo
```

should be translated into:

```
c$omp thread private tcb1, tcb2
    common /tcb1/ x,y,z(10000)
    common /tcb2/ a,b,c

    x = 1
    y = 2
c$omp parallel do private(i) shared(n)
copyin(x,y,/tcb2/)
    do i = 1, n
       ...
    enddo
```

SGI Directive to KAP/Pro Parallel Directive Translator

Programs which have been prallelized with SGI `c$` directives can be used as the basis for a port to Guide. The `sgi2par.pl` program will help translate SGI directives into KAP/Pro parallel processing directives. These KAP/Pro directives may then be translated automatically into OpenMP directives that Guide accepts using the `kpts2omp.pl` program described earlier on page 73.

The `sgi2par.pl` program accepts as an argument the name of a file with SGI directives. The translated file is written to `stdout` with KAP/Pro parallel process-

ing directives added. The `stdout` can be redirected to capture the translated file. Any directives or consutructs that cannot be handled automatically cause diagnostics to be added inline in the translated output. The `stderr` output contains the synopsis of the diagnostics.

Most of the common SGI directives are handled automatically by this program. Whenever manual intervention is required, searching for "`sgi2par.pl`" in the output will lead to places where `sgi2par.pl` had trouble performing translations.

Table 8-4, "SGI to KAP/Pro Translations," below lists the SGI directives and their translations that are performed. Many of the directives in the table have optional clauses that are translated by `sgi2par.pl` when possible. A diagnostic is produced when there is not an equivalent KAP/Pro parallel processing directive.

None of the SGI scheduling keywords are automatically translated my `sgi2par.pl`. `Sgi2par.pl` produces a diagnostic to assist in manually inserting scheduling keywords into the program.

Table 8-4 SGI to KAP/Pro Translations

SGI directive or clause or library routine	KAP/Pro Translation
`c$doacross`	`c$par parallel do`
`c$ call mp_barrier`	`c$par barrier`
`c$ call mp_setlock`	`c$par critical section`
`c$ call mp_unsetlock`	`c$par end critical section`
`mp_my_threadnum`	Not translated automatically, but can be translated using mpptid
`mp_numthreads`	Not translated automatically, but can be translated using mppnth
`c$copyin`	Not translated automatically, but can be translated manually
`c$ mp_schedtype clause`	Not translated automatically, but can be translated manually
`c$mp_schedtype directive`	No translation, have to propagate scheduling type to rest of file manually

NOTE: `Perl` must be installed on your system to use `sgi2par.pl`.

KAP Directive to OpenMP Directive Translator

Programs which contain the older PCF directives of the form *KAP* can be used as the basis for a port to OpenMP. The `kap2omp.pl` program will help translate KAP directives into OpenMP directives.

The `kap2omp.pl` program accepts the name of a file with KAP directives. The translated file is written to `stdout` with OpenMP directives added. The `stdout` can be redirected to capture the translated file. Any directives or constructs that cannot be handled automatically cause diagnostics to be added inline in the translated output. The `stderr` output contains the synopsis of the diagnostics. All `cray2omp.pl` translator options given in Table 8-2, “cray2omp.pl Translator Options,” on page 75, are also available for the `kap2omp.pl` program.

NOTE: `Perl` must be installed on the system to use `kap2omp.pl`.

APPENDIX A Examples

For your convenience, the following examples have been adapted from the ANSI *X3H5 Parallel Extensions for FORTRAN* document.

PDO: A Simple Difference Operator

This example shows a simple parallel loop where the amount of work in each iteration is different. We used dynamic scheduling to get good load balancing. The `end pdo` has a `nowait` because there is an implicit `barrier` at the `end parallel`. Alternately, using the option **-optimize=1** would have also eliminated the `barrier`.

```
        subroutine pdo_1 (a,b,n)
        real a(n,n), b(n,n)

c$omp parallel
c$omp&   shared(a,b,n)
c$omp&   private(i,j)
c$omp pdo schedule(dynamic,1)
        do i = 2, n
            do j = 1, i
               b(j,i) = ( a(j,i) + a(j,i-1) ) / 2
            enddo
        enddo
c$omp end pdo nowait
c$omp end parallel
        end
```

PDO: Two Difference Operators

Shows two parallel regions fused to reduce fork/join overhead. The first `end pdo` has a `nowait` because all the data used in the second `pdo` is different than all the data used in the first `pdo`.

```
        subroutine pdo_2 (a,b,c,d,m,n)
        real a(n,n), b(n,n), c(m,m), d(m,m)

c$omp parallel
c$omp&   shared(a,b,c,d,m,n)
c$omp&   private(i,j)
c$omp pdo schedule(dynamic,1)
        do i = 2, n
            do j = 1, i
                b(j,i) = ( a(j,i) + a(j,i-1) ) / 2
            enddo
        enddo
c$omp end pdo nowait
c$omp pdo schedule(dynamic,1)
        do i = 2, m
            do j = 1, i
                d(j,i) = ( c(j,i) + c(j,i-1) ) / 2
            enddo
        enddo
c$omp end pdo nowait
c$omp end parallel
        end
```

PDO: Reduce Fork/Join Overhead

Routines `pdo_3a` and `pdo_3b` perform numerically equivalent computations, but because the `parallel` directive in routine `pdo_3b` is outside the `do j` loop, routine `pdo_3b` probably forms teams less often, and thus reduces overhead.

```
        subroutine pdo_3a (a,b,m,n)
        real a(n,m), b(n,m)

        do j = 2, m
c$omp parallel
c$omp&   shared(a,b,n,j)
c$omp&   private(i)
c$omp pdo
            do i = 1, n
                a(i,j) = b(i,j) / a(i,j-1)
            enddo
c$omp end pdo nowait
c$omp end parallel
        enddo
        end


        subroutine pdo_3b (a,b,m,n)
        real a(n,m), b(n,m)

c$omp parallel
c$omp&   shared(a,b,m,n)
c$omp&   private(i,j)
        do j = 2, m
c$omp pdo
            do i = 1, n
                a(i,j) = b(i,j) / a(i,j-1)
            enddo
c$omp end pdo nowait
        enddo
c$omp end parallel
        end
```

PSECTIONS: Two Difference Operators

Identical to “PDO: Two Difference Operators” on page 83 but uses `psections` instead of `pdo`. Here the speedup is limited to 2 because there are only 2 units of work whereas in “PDO: Two Difference Operators” on page 83 there are *n-1* + *m-1* units of work .

```
        subroutine psections_1 (a,b,c,d,m,n)
        real a(n,n), b(n,n), c(m,m), d(m,m)

c$omp parallel
c$omp&   shared(a,b,c,d,m,n)
c$omp&   private(i,j)
c$omp psections
c$omp psection
        do i = 2, n
            do j = 1, i
                b(j,i) = ( a(j,i) + a(j,i-1) ) / 2
            enddo
        enddo
c$omp psection
        do i = 2, m
            do j = 1, i
                d(j,i) = ( c(j,i) + c(j,i-1) ) / 2
            enddo
        enddo
c$omp end psections nowait
c$omp end parallel
        end
```

PSINGLE: Updating a Shared Scalar

This example demonstrates how to use a `psingle` construct to update an element of the shared array `a`. The optional `end pdo nowait` after the first `pdo` is omitted because we need to wait at the end of the `pdo` before proceeding into the `psingle`.

```
              subroutine sp_1a (a,b,n)
              real a(n), b(n)

      c$omp parallel
      c$omp&   shared(a,b,n)
      c$omp&   private(i)
      c$omp pdo
              do i = 1, n
                  a(i) = 1.0 / a(i)
              enddo
      c$omp psingle
              a(1) = min( a(1), 1.0 )
      c$omp end psingle
      c$omp pdo
              do i = 1, n
                  b(i) = b(i) / a(i)
              enddo
      c$omp end pdo nowait
      c$omp end parallel
              end
```

PSECTIONS: Updating a Shared Scalar

Identical to “PSINGLE: Updating a Shared Scalar” on page 86 but using different directives.

```
              subroutine psection_sp_1 (a,b,n)
              real a(n), b(n)

      c$omp parallel
      c$omp&   shared(a,b,n)
      c$omp&   private(i)
      c$omp pdo
              do i = 1, n
                  a(i) = 1.0 / a(i)
              enddo
      c$omp psections
              a(1) = min( a(1), 1.0 )
      c$omp end psections
      c$omp pdo
              do i = 1, n
                  b(i) = b(i) / a(i)
              enddo
      c$omp end pdo nowait
      c$omp end parallel
              end
```

PDO: Updating a Shared Scalar

Identical to “PSINGLE: Updating a Shared Scalar” on page 86 but using different directives.

```
        subroutine pdo_sp_1 (a,b,n)
        real a(n), b(n)

c$omp parallel
c$omp&   shared(a,b,n)
c$omp&   private(i)
c$omp pdo
        do i = 1, n
            a(i) = 1.0 / a(i)
        enddo
c$omp end pdo
c$omp pdo
        do i = 1, 1
            a(1) = min( a(1), 1.0 )
        enddo
c$omp end pdo
c$omp pdo
        do i = 1, n
            b(i) = b(i) / a(i)
        enddo
c$omp end pdo nowait
c$omp end parallel
        end
```

PARALLEL DO: A Simple Difference Operator

Identical to “PDO: A Simple Difference Operator” on page 82 but using different directives.

```
        subroutine paralleldo_1 (a,b,n)
        real a(n,n), b(n,n)

c$omp parallel do
c$omp&   shared(a,b,n)
c$omp&   private(i,j)
c$omp&   schedule(dynamic,1)
        do i = 2, n
            do j = 1, i
                b(j,i) = ( a(j,i) + a(j,i-1) ) / 2
            enddo
        enddo
        end
```

PARALLEL SECTIONS: Two Difference Operators

Identical to “PSECTIONS: Two Difference Operators” on page 85 but using different directives. The maximum performance improvement is limited to the number of sections run in parallel, so this example has a maximum parallelism of 2.

```
        subroutine psections_2 (a,b,c,d,m,n)
        real a(n,n), b(n,n), c(m,m), d(m,m)

c$omp parallel sections
c$omp&   shared(a,b,c,d,m,n)
c$omp&   private(i,j)
c$omp psection
        do i = 2, n
            do j = 1, i
               b(j,i) = ( a(j,i) + a(j,i-1) ) / 2
            enddo
        enddo
c$omp psection
        do i = 2, m
            do j = 1, i
               d(j,i) = ( c(j,i) + c(j,i-1) ) / 2
            enddo
        enddo
c$omp end parallel sections
        end
```

Simple Reduction

This demonstrates how to perform a reduction using partial sums while avoiding synchronization in the loop body.

```
        subroutine reduction_1 (a,m,n,sum)
        real a(m,n)

c$omp parallel
c$omp&   shared(a,m,n,sum)
c$omp&   private(i,j,local_sum)
        local_sum = 0.0
c$omp pdo
        do i = 1, n
            do j = 1, m
                local_sum = local_sum + a(j,i)
            enddo
        enddo
c$omp end pdo nowait
c$omp critical
        sum = sum + local_sum
c$omp end critical
c$omp end parallel
        end
```

The above reduction could also use the `REDUCTION ()` clause as follows:

```
        subroutine reduction_2 (a,m,n,sum)
        real a(m,n)

c$omp parallel do
c$omp&   shared(a,m,n)
c$omp&   private(i,j)
c$omp&   reduction(+:sum)
        do i = 1, n
            do j = 1, m
                local_sum = local_sum + a(j,i)
            enddo
        enddo
        end
```

TASKCOMMON: Private Common

This example demonstrates the use of `taskcommon` privatizable `common` blocks.

```
              subroutine tc_1 (n)
              common /shared/ a
              real a(100,100)
              common /private/ work
              real work(10000)
      c$omp thread private (/private/)  ! this privatizes the
                                        ! common /private/
      c$omp parallel
      c$omp&   shared(n)
      c$omp&   private(i)
      c$omp pdo
              do i = 1, n
                  call construct_data() ! fills in array work()
                  call use_data()       ! uses array work()
              enddo
      c$omp end pdo nowait
      c$omp end parallel
              end
```

THREAD PRIVATE: Private Common and Master Thread

In this example, the value 2 is printed since the master thread's copy of a variable in a `thread private` privatizable `common` block is accessed within a `master` section or in serial code sections. If a `psingle` was used in place of the `master` section, some single thread, but not necessarily the master thread, would set `j` to 2 and the printed result would be indeterminate.

```
        subroutine tc_2
        common /blk/ j
c$omp thread private (/blk/)

        j = 1
c$omp parallel
c$omp master
        j = 2
c$omp end master
c$omp end parallel

        print *, j
        end
```

INSTANCE PARALLEL: As a Private Common

This demonstrates the use of `instance parallel` privatizable common blocks.

```
              subroutine ip_1 (n)
              common /shared/ a
              real a(100,100)
              common /private/ work
              real work(10000)
      c$omp instance parallel (/private/)

      c$omp parallel
      c$omp&   shared(n)
      c$omp&   private(i)
      c$omp new (/private/)            ! this privatizes the
      c$omp pdo                        ! common /private/
              do i = 1, n
                  call construct_data()! fills in array work()
                  call use_data()      ! uses array work()
              enddo
      c$omp end pdo nowait
      c$omp end parallel
              end
```

INSTANCE PARALLEL: As a Shared and then a Private Common

This demonstrates the use of an `instance parallel common` block first as a shared `common` block and then as a private `common` block. This would not be possible with `taskcommon` blocks since `taskcommon` blocks are always private.

```
        subroutine ip_2 (n,m)
        common /shared/ a,b
        real a(100,100), b(100,100)
        common /private/ work
        real work(10000)
c$omp instance parallel (/private/)

c$omp parallel                      ! common /private/ is
c$omp&   shared(a,b,n)              ! shared here since
c$omp&   private(i)                 ! no new appears
c$omp pdo
        do i = 1, n
            work(i) = b(i,i) / 4.0
        enddo
c$omp end pdo nowait
c$omp end parallel

        do i = 1, n
            do j = 1, m
                a(j,i) = work(i) * ( a(j-1,i) + a(j+1,i)
     x                   + a(j,i-1) + a(j,i+1) )
            enddo
        enddo

c$omp parallel
c$omp&   shared(m)
c$omp&   private(i)
c$omp new (/private/)               ! this privatizes the
c$omp pdo                           ! common /private/
        do i = 1, m
            call construct_data()   ! fills in array work()
            call use_data()         ! uses array work()
        enddo
c$omp end pdo nowait
c$omp end parallel
        end
```

Avoiding External Routines: Reduction

This example demonstrates two coding styles for reductions, one using the external routines `omp_get_max_threads()` and `omp_get_thread_num()` and the other using only OpenMP directives.

```
                      subroutine reduction_3a (n)
                      real gx( 0:7 )   ! assume 8 processors

                      do i = 0, omp_get_max_threads()-1
                          gx(i) = 0
                      enddo

              c$omp parallel
              c$omp&   shared(a)
              c$omp&   private(i,lx)
                      lx = 0
              c$omp pdo
                      do i = 1, n
                          lx = lx + a(i)
                      enddo
              c$omp end pdo nowait
                      gx( omp_get_thread_num() ) = lx
              c$omp end parallel

                      x = 0
                      do i = 0, omp_get_max_threads()-1
                          x = x + gx(i)
                      enddo

                      print *, x
                      end
```

As is shown below, this example could have been written without the external routines.

```
        subroutine reduction_3b (n)

        x = 0
c$omp parallel
c$omp&   shared(a,x)
c$omp&   private(i,lx)
        lx = 0
c$omp pdo
        do i = 1, n
            lx = lx + a(i)
        enddo
c$omp end pdo nowait
c$omp critical
        x = x + lx
c$omp end critical
c$omp end parallel

        print *, x
        end
```

This example could have also been written more simply using the `reduction()` clause as follows:

```
        subroutine reduction_3c (n)

        x = 0
c$omp parallel
c$omp&   shared(a)
c$omp&   private(i)
c$omp pdo reduction(+:x)
        do i = 1, n
            x = x + a(i)
        enddo
c$omp end pdo nowait
c$omp end parallel

        print *, x
        end
```

Avoiding External Routines: Temporary Storage

This example demonstrates three coding styles for temporary storage, one using the external routine and `omp_get_thread_num()` and the other two using only directives.

```
          subroutine local_1a (n)
          dimension a(100)
          common /cmn/ t( 100, 0:7 )  ! assume 8 processors
         max.
   c$omp parallel do
   c$omp&   shared(a,t)
   c$omp&   private(i)
          do i = 1, n
              do j = 1, n
                  t(j, omp_get_thread_num()) = a(i) ** 2
              enddo
              call work( t(1,omp_get_thread_num()) )
          enddo
          end
```

If `t` is not global, then the above could be accomplished by putting `t` in the `private` clause:

```
          subroutine local_1b (n)
          dimension t(100)

   c$omp parallel do
   c$omp&   shared(a)
   c$omp&   private(i,t)
          do i = 1, n
              do j = 1, n
                  t(j) = a(i) ** 2
              enddo
              call work( t )
          enddo
          end
```

If `t` is global, then the `instance` `parallel` and `new` directives can be used instead.

```
        subroutine local_1c (n)
        dimension t(100)
        common /cmn/ t
c$omp instance parallel (/cmn/)

c$omp parallel do
c$omp&   shared(a)
c$omp&   private(i)
c$omp new (/cmn/)
        do i = 1, n
            do j = 1, n
                t(j) = a(i) ** 2
            enddo
            call work   ! access t from common /cmn/
        enddo
        end
```

FIRSTPRIVATE: Copying in Initialization Values

Not all of the values of a and b are initialized in the loop before they are used (the rest of the values are produced by `init_a` and `init_b`). Using `firstprivate` for a and b causes the initialization values produced by `init_a` and `init_b` to be copied into private copies of a and b for use in the loops.

```
        subroutine dsq3_b (c,n)
        integer n
        real a(100), b(100), c(n,n), x, y
        call init_a( a, n )
        call init_b( b, n )
c$omp parallel do shared(c,n) private(i,j,x,y)
firstprivate(a,b)
        do i = 1, n
            do j = 1, i
                a(j) = calc_a(i)
                b(j) = calc_b(i)
            enddo
            do j = 1, n
                x = a(i) - b(i)
                y = b(i) + a(i)
                c(j,i) = x * y
            enddo
        enddo
c$omp end parallel do
        print *, x, y
        end
```

THREAD PRIVATE: Copying in Initialization Values

Similar to “FIRSTPRIVATE: Copying in Initialization Values” on page 100 except using `thread private` common blocks. For `thread private`, `copyin` is used instead of `firstprivate` to copy initialization values from the shared (master) copy of `/blk/` to the private copies.

```
        subroutine dsq3_b_tc (c,n)
        integer n
        real a(100), b(100), c(n,n), x, y
        common /blk/ a,b
c$omp thread private (/blk/)

        call init_a( a, n )
        call init_b( b, n )
c$omp parallel do shared(c,n) private(i,j,x,y)
copyin(a,b)
        do i = 1, n
            do j = 1, i
                a(j) = calc_a(i)
                b(j) = calc_b(i)
            enddo
            do j = 1, n
                x = a(i) - b(i)
                y = b(i) + a(i)
                c(j,i) = x * y
            enddo
        enddo
c$omp end parallel do
        print *, x, y
        end
```

INSTANCE PARALLEL: Copying in Initialization Values

Similar to “FIRSTPRIVATE: Copying in Initialization Values” on page 100 except using `instance parallel` privatizable common blocks. For `instance parallel`, `copy new` is used instead of `firstprivate` to privatize the common block and to copy initialization values from the shared (master) copy of `/blk/` to the private copies.

```
        subroutine dsq3_b_ip (c,n)
        integer n
        real a(100), b(100), c(n,n), x, y
        common /blk/ a,b
c$omp instance parallel (/blk/)

        call init_a( a, n )
        call init_b( b, n )
c$omp parallel do shared(c,n) private(i,j,x,y)
c$omp copy new (/blk/)
        do i = 1, n
            do j = 1, i
                a(j) = calc_a(i)
                b(j) = calc_b(i)
            enddo
            do j = 1, n
                x = a(i) - b(i)
                y = b(i) + a(i)
                c(j,i) = x * y
            enddo
        enddo
c$omp end parallel do
        print *, x, y
        end
```

APPENDIX B

Timing Guide Constructs

The table contained in this appendix demonstrates the amount of time expended for OpenMP directives in comparison to a null call for a typical RISC processor based SMP. A null call is a call to an empty function.

```
SUBROUTINE NULL
RETURN
END
```

In the table below, it took about 10 cycles to call the null function. A `BARRIER` construct (`!$OMP BARRIER`) is about 10 times slower for 1 processor, and about 70 times slower for 2 processors.

Typical Overhead

Guide Construct	1 processor		2 processor		3 processor		4 processor	
	X null call	cycles	X null call	cycles	X null call	cycles	X null call	cycles
function call	1	10	1	10	1	10	1	10
barrier	10	100	70	700	90	900	100	1000
psingle	20	200	90	900	110	1100	130	1300
critical section	30	300	70	700	150	1500	210	2100
parallel region	50	500	190	1900	220	2200	280	2800

This information can be used to draw the following general conclusions:

- A BARRIER statement is 30 to 50 percent less expensive than a PARALLEL region.
- BARRIERs and PSINGLEs have roughly the same overhead.
- After 2 processors, all the costs follow a linear pattern as you add processors.

Index

A

advanced optimization 39, 41
  command line options 39, 41
alignmax 42, 43
all
  save option 51
all_adjust
  save option 52
as 41, 43
assume 41, 43
ATOMIC 26

B

BARRIER 27
barrier 7
  reported overhead 67
blank_padding 42, 44
bold typeface 3
bp 42, 44

C

c*$*options 47, 54
case 42, 44
chk 42, 44
chunk 28, 42, 44
cmp 41, 44
command line options 41, 50
  1 42, 49
  advanced optimization 39, 41
  alignmax 42, 43
  alphabetic listing 43–54
  as 41, 43
  assume 41, 43
  blank_padding 42, 44
  bp 42, 44
  case 42, 44
  chk 42, 44
  chunk 42, 44
  cmp 41, 44
  conc 41, 45
  concurrentize 41, 45
  datasave 42, 45
  directives 42, 45
  dl 42, 45

dlines 42, 45
dr 42, 45
ds 42, 45
free 42
heap 42, 46
heaplimit 42, 46
i 41, 47
ig 42, 47
ignoreoptions 42, 47
inc 42, 47
include 42, 47
input 41, 47
int 42, 47
integer 42, 47
l 41, 48
lines 41, 48
list 41, 48
listoptions 41, 48
ln 41, 48
lo 41, 48
log 42, 48
logical 42, 48
mc 41
minconcurrent 41
o 41, 50
onetrip 42, 49
optimize 41, 50
rc 42, 50
real 42, 50
recursion 42, 50
rl 42, 50
roundoff 41, 50
save 42, 51
scalaropt 41, 52
scan 42, 52
schd 42, 52
scheduling 42, 52
so 41, 52
specifying 47, 54
su 41, 53
suppress 41, 53
sv 42, 51
sy 42, 53
syntax 42, 53
ty 42, 53
type 42, 53
common blocks
allocating private 24
declaring private 23
privatizing 10, 23
common privatization 22
allocating private commons 24
declaring private commons 23
INSTANCE PARALLEL 23
common privatization directives
THREAD PRIVATE 23
conc 41, 45
concurrentize 41, 45
control directives 15–22
COPYIN 22
END PARALLEL 15
END PARALLEL DO 18
END PARALLEL SECTIONS 19
END PDO 16
END PSECTIONS 17
END PSINGLE 17
FIRSTPRIVATE 21
LASTPRIVATE 21
PARALLEL 15
PARALLEL DO 18
PARALLEL SECTIONS 19
PDO 16
PSECTIONS 17
PSINGLE 17
REDUCTION 21
COPYIN 22
courier font 3
CRITICAL 25

D

datasave 42, 45
debugging code 45
DEC
FORTRAN extensions 53
Digital
FORTRAN extensions 53
directives 42, 45
ATOMIC 26
BARRIER 27
control 15–22

COPYIN 22
CRITICAL 25
END CRITICAL 25
END MASTER 26
END ORDERED 25
END PARALLEL 15
END PARALLEL DO 18
END PARALLEL SECTIONS 19
END PDO 16
END PSECTIONS 17
END PSINGLE 17
FIRSTPRIVATE 21
FLUSH 26
INSTANCE PARALLEL 23
LASTPRIVATE 21
MASTER 26
ORDERED 25
PARALLEL 15
PARALLEL DO 18
PARALLEL SECTIONS 19
PDO 16
PSECTIONS 17
PSINGLE 17
recognition 45
REDUCTION 21
synchronization 25
THREAD PRIVATE 23
dl 42, 45
dlines 42, 45
dr 42, 45
driver options
h 35
v 35
w 35
WG 36
WGcompiler 36
WGcpp 36
WGf77 36
WGf90 36
WGfortran 36
WGftn 36
WGkeep 37
WGkeepcpp 37
WGld 37
WGlibpath 37
WGlink 37
WGnocpp 37
WGnokeep 37
WGnoprocess 38
WGnorc 38
WGonly 38
WGpath 38
WGprefix 38
WGsrcdir 38
WGversion 39
ds 42, 45

E

eliminating 7
END CRITICAL 25
END MASTER 26
END ORDERED 25
END PARALLEL 15
END PARALLEL DO 18
END PARALLEL SECTIONS 19
END PDO 16
END PSECTIONS 17
END PSINGLE 17
environment variables 29, 30, 31
kmp_library 29
kmp_scheduling 29
kmp_stacksize 30
kmp_statsfile 30
ld_library_path 30
omp_dynamic 30
omp_num_threads 31
omp_schedule 31
scheculing options 29
error messages 53
suppressing 53
external routines 57
mppbeg() 57
mppend() 57
omp_get_max_threads() 58
omp_get_num_procs() 58
omp_get_num_threads() 58
omp_get_thread_num() 59

F

FIRSTPRIVATE 21

FLUSH 26
FORTRAN
  dialects 40, 53
free 42

G
guidefrc 35

H
heap 42, 46
heaplimit 42, 46

I
i 41, 47
ig 42, 47
ignoreoptions 42, 47
inc 42, 47
include 42, 47
input 41, 47
INSTANCE PARALLEL 23
int 42, 47
integer 42, 47

K
kmp_library 29
kmp_scheduling 29
kmp_stacksize 30
kmp_statsfile 30

L
l 41, 48
LASTPRIVATE 21
ld_library_path 30
libraries 55, 57
  linking 57
  selecting 55
lines 41, 48
linking
  libraries 57
list 41, 48
listoptions 41, 48
ln 41, 48
lo 41, 48
log 42, 48
logical 42, 48

M
manual
  save option 51
manual_adjust
  save option 51
MASTER 26
mc 41
messages
  suppressing 53
minconcurrent 41
mppbeg() 57
mppend() 57

O
o 41, 50
omp_dynamic 30
omp_get_max_threads() 58
omp_get_num_procs() 58
omp_get_num_threads() 58
omp_get_thread_num() 59
omp_num_threads 31
omp_schedule 31
onetrip 42, 49
optimize 41, 50
options 47, 54
ORDERED 25

P
PARALLEL 15
PARALLEL DO 18
PARALLEL SECTIONS 19
PDO 16
private commons
  allocating 24
  declaring 23
privatization
  directives 10, 23
PSECTIONS 17
PSINGLE 17

R
r 41, 50
rc 42, 50
real 42, 50
recursion 42, 50

REDUCTION 21
roundoff 41, 50

S
save 42, 51
  all 51
  all_adjust 52
  manual 51
  manual_adjust 51
scalaropt 41, 52
scan 42, 52
schd 42, 52
scheduling 42, 52
scheduling options 27
  chunk size 28
  environment variables 29
so 41, 52
su 41, 53
suppress 41, 53
sv 42, 51
sy 42, 53
synchronization directives 25, 26
  ATOMIC 26
  BARRIER 27
  CRITICAL 25
  FLUSH 26
  MASTER 26
  ORDERED 25
syntax 42, 53

T
THREAD PRIVATE 23
ty 42, 53
type 42, 53

W
warnings
  suppressing 53
WG 36
WGcompiler 36
WGcpp 36
WGf77 36
WGf90 36
WGfortran 36
WGftn 36
WGkeep 37
WGkeepcpp 37
WGld 37
WGlibpath 37
WGlink 37
WGnocpp 37
WGnokeep 37
WGnoprocess 38
WGnorc 38
WGonly 38
WGpath 38
WGprefix 38
WGsrcdir 38
WGversion 39