GUIDE™ Reference Manual (Fortran Edition)

Version 3.6

Document #9607001
Kuck & Associates, Inc.

GUIDE™ Reference Manual
Version 3.6

Revised January, 1999

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

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 the OpenMP API 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. This 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 specification, unless stated otherwise.

The KAP/Pro Toolset includes utilities to translate directives from older parallel processing 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 is then compiled using your existing FORTRAN compiler.

Guide requires the native Fortran compiler. GuideView requires a Java™ interpreter, which can be obtained from Sun or Microsoft via the world wide web. Perl is required for the directive translation scripts described in Chapter 8. Links to these packages are available on the KAI web site at *http://www.kai.com/kpts/helpers*.

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 OpenMP directives.

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

Chapter 4, “The Guide Drivers,” beginning on page 39, 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 63, explains the differences among Guide’s several run–time libraries.

Chapter 6, “GuideView,” beginning on page 73, describes the GuideView graphical performance viewer.

Chapter 7, “PerView,” beginning on page 77, describes the PerView application manager and monitor.

Chapter 8, “Directive Translation,” beginning on page 87, describes the included utilities that translate older directives to OpenMP directives.

Appendix A, “Examples,” beginning on page 95, contains code examples with OpenMP directives.

Appendix B, “Timing Guide Constructs,” beginning on page 117, 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*, you 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, **<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 strives 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. For exact semantics, please consult the OpenMP Fortran API standard document available at *http://www.openmp.org/* 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. When the parallel program begins execution, a single thread exists. This thread is called the base or master thread. The master thread will continue serial processing until it encounters a parallel region. Several OpenMP directives apply to sections, or blocks, of source code. These OpenMP directives come in pairs and have the following form:

```
c$omp <directive>
      <structured block of code>
c$omp end <directive>
```

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 `do`, `sections`, and `single` constructs are defined as worksharing constructs because they distribute the enclosed work among the members of the current team. 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 dynamically enclosing parallel region if one exists. Otherwise, it will be executed serially.

The `do` directive specifies parallel execution of a `do` loop. The `sections` directive specifies parallel execution for arbitrary blocks of sequential code, one section per thread. The `single` directive defines a section of code where exactly one thread is allowed to execute the code.

Synchronization constructs are `critical`, `ordered`, `master`, `atomic`, `flush`, and `barrier`. Synchronization can be specified within a parallel region or a worksharing construct with the `critical` directive. Only one thread at a time is allowed to execute the code within a `critical` section. Within a `do` or `sections` construct, synchronization can be specified with an `ordered` directive. This directive is used in conjunction with a `do` or `sections` construct with the `ordered` clause to impose an order on the execution of a section of code. The `master` directive is another synchronization directive 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. `barrier`s cannot occur within worksharing or synchronization constructs due to the potential for deadlock.

When a thread reaches the end of a worksharing construct, it may 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 threads wait 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 the `do` worksharing construct to increase efficiency. Scheduling options specify the way processes are assigned iterations for a loop. 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 **-WG,-optimize** can also help increase efficiency. For example, using **-WG,-optimize=1** will perform optimizations, such as eliminating unnecessary `barrier`s. The default setting for this option is **-WG,-optimize=1**.

Figure 2-1 “Pseudo Code of the Parallel Processing Model”

```
program main          ! Begin Serial Execution
                      !
 ...                  ! Only the master thread executes
                      !
!$omp parallel        ! Begin a Parallel Construct,
                      ! form a team
                      !
    ...               ! This is Replicated Code where each team
    ...               ! member executes the same code
                      !
    !$omp sections    ! Begin a Worksharing Construct
                      !
    !$omp section     ! One unit of work
       ...            !
    !$omp section     ! Another unit of work
       ...            !
    end sections      ! Wait until both units of work complete
                      !
    ...               ! More Replicated Code
                      !
    !$omp do          ! Begin a Worksharing Construct,
                      ! each iteration is a unit of work
                      !
    ...               ! Work is distributed among the team
                      !
    end do nowait     ! End of Worksharing Construct,
                      ! nowait is specified
                      !
    ...               ! More Replicated Code
                      !
    !$omp 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 construct by 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 respon-

sibility 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 also the programmer's responsibility, unless the `firstprivate` clause is specified. In this 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 `do` directive enables updating the global copy from the team member that executed the last iteration of the `do`.

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 `threadprivate` directive. For compatibility with Cray `TASKCOMMON` directives, `threadprivate` 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 those familiar with parallel programming, this section contains a high-level overview of 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 a parallel processing text.

Prepare

- Before inserting any OpenMP parallel directives, verify that your code is safe for parallel execution by placing local variables on the stack. Normally, an "-automatic" compiler option or similar option achieves this. If your application doesn't pass tests with stack allocation of local data, you normally need to iden-

tify subroutines that need some variables saved across invocations. These variables should be placed in a SAVE statement. Make a note of these variables, as SAVE statements make them shared data, and access to shared data must generally be synchronized among threads.

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 `parallel do` directive around this loop.
- 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` region should be synchronized.
- Identify more parallel loops and restructure them.
- If possible, merge adjacent `parallel do`s into a single parallel region with multiple `do`s to reduce execution overhead.

Tune

- Guide supports the tuning process via the *guide_stats* library and GuideView. The tuning process should include minimizing the sequential code in `critical` sections and load balancing by using the scheduling options listed in “Scheduling Options” on page 28.

For parallel FORTRAN 77 programs with older parallel directives, a tool is included with Guide to help automate the job of translating them to OpenMP parallel directives. See “Directive Translation” on page 87.

Orphaned Directives

OpenMP contains a new feature, called orphaning, that dramatically increases the expressiveness of parallel directives. While earlier models required all of the directives related to a parallel region to occur lexically within a single program unit, OpenMP relaxes this restriction. Now, directives such as `do`, `critical`, `barrier`, `sections`, `single`, and `master` can be “orphaned”. That is, they can occur by themselves in a program unit, dynamically “binding” to the enclosing `parallel` region at run time.

Orphaned directives allow parallelism to be inserted into existing code with a minimum of code restructuring. Orphaning can also improve performance by allowing a single `parallel` region to bind with multiple `do` directives located within called subroutines. The example:

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

is a common programming idiom for using the `do` 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 do
        do i = 1, n
           call some_work(i)
        end do
c$omp do
        do j = 1, n
           call more_work(j)
        end do
c$omp end parallel
```

However, programs are sometimes naturally structured by placing each of the major computational sections into its own program unit. 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
```

With OpenMP, you can parallelize this code in a more natural manner than was possible with previous directive sets.

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

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

        subroutine phase2
c$omp do
           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 program units.

A Few Rules about "Orphaned" Directives

1. An orphaned worksharing construct (`do/section/single`) that is dynamically executed outside of a 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 do
        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 do
        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 within 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 do
        do i = 1, n
           call work(i)
        end do
        end
```

4. The opening and closing directives of a directive pair must occur in a single block of the program.

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

CHAPTER 3

OpenMP Directives

Guide uses OpenMP directives to support a single level of parallelism. Each directive begins with `c$omp` or `!$omp`. When a directive is continued on subsequent lines, each additional line begins with `c$omp&` or `!$omp&`. Several directives must be paired (directive and `end` directive). Please note that items enclosed in square brackets (`[]`) are optional. The syntax of the OpenMP directives accepted by Guide is presented below.

Many of the directives in this chapter include a reference to a `<structured-block>` in their description. A structured block has a single entry point and a single exit point. No sequence of statements is a structured block if there is a branch into or out of that sequence. For example, `goto` statements and labeled statements may not be included in structured blocks unless both the `goto` and its corresponding labeled statement are both contained within the sequence of statements which comprise the structured block.

Parallel Directive

parallel

The `parallel` directive defines a parallel region.

```
c$omp parallel [ <clause> [[,] <clause> ] ... ] <new-line>
  <structured-block>
c$omp end parallel <new-line>
```

where `<clause>` is one of the following:

```
if (<scalar-expression>)
default (shared | private | none)
shared (<list>)
private (<list>)
firstprivate (<list>)
reduction (<operator> : <list>)
reduction (<intrinsic> : <list>)
copyin (<list>)
```

When the logical `if` clause exists, the `<scalar-expression>` is evaluated at run time. 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 code in the parallel region may be executed by a team of multiple threads. When the `if` clause is not present, it is treated as if `if (.true.)` were present.

When a parallel region is encountered in the dynamic scope of another parallel region, the inner parallel region is executed using a team of one thread. The remaining clauses are described in “Data Scope Attribute Clauses” on page 24.

Worksharing Directives

do

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

```
c$omp do [ <clause> [[,] <clause> ] ... ] <new-line>
  <do-loop>
[ c$omp end do [ nowait ] <new-line> ]
```

where <clause> is one of the following:

```
schedule (<type>[, <chunk-size>)
private (<list>)
firstprivate (<list>)
lastprivate (<list>)
reduction (<operator> : <list>)
reduction (<intrinsic> : <list>)
ordered
```

Using the end do directive is optional. Without the nowait clause, all threads that reach the end of the loop will wait until all iterations have been completed. Therefore, the end do directive without the nowait clause has no effect, and the end of the do directive is marked by the end of the do loop. Specifying the end do nowait directive allows early finishing threads to execute code that follows the loop. If the end do directive is used, no statements or directives may appear between the last statement of the do loop and the end do directive.

The schedule clause is described in more detail in “Scheduling Options” on page 28. The ordered clause is described on page 21.

sections

The sections directive delineates sections of code that can be executed on different threads. Each parallel section except the first must be preceded by the section directive. If the sections 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 sections are executed on the thread that encounters this construct.

```
c$omp sections [ <clause> [[,] <clause> ] ... ] <new-line>
[ c$omp section <new-line> ]
  <structured-block>
[ c$omp section <new-line>
  <structured-block>
.
.
. ]
c$omp end sections [ nowait ] <new-line>
```

where <clause> is one of the following:

```
private (<list>)
firstprivate (<list>)
lastprivate (<list>)
reduction (<operator> : <list>)
reduction (<intrinsic> : <list>)
ordered
```

The `ordered` clause is a KAP/Pro Toolset extension and is described on page 21.

single

The `single` directive defines a section of code where exactly one thread is allowed to execute the code.

```
c$omp single [ <clause> [[,] <clause> ] ... ] <new-line>
  <structured-block>
c$omp end single [ nowait ] <new-line>
```

where `<clause>` is one of the following:

```
private (<list>)
firstprivate (<list>)
lastprivate (<list>)
reduction (<operator> : <list>)
reduction (<intrinsic> : <list>)
```

The first arriving thread is allowed to execute the `<structured-block>` of code following the `single` directive. Other threads wait until this thread has finished the section of code, then they continue executing with the statement after the `single` block. If the `nowait` clause is present, then the other threads do not wait, but instead immediately skip the `<structured-block>`.

The `lastprivate` and `reduction` clauses are KAP/Pro Toolset extensions.

Combined Parallel and Worksharing Directives

parallel do

The `parallel do` directive is a short form syntax for a parallel region enclosing a single `do`. The `parallel do` directive is used in place of the `parallel` and `do` directives. If this directive is encountered while a parallel region is already active, then this directive is executed by a team of one thread and the entire loop is executed by each thread that encounters it.

```
c$omp parallel do [ <clause> [[,] <clause> ] ... ] <new-line>
  <do-loop>
[ c$omp end parallel do <new-line> ]
```

where `<clause>` is one of the following:

```
if (<scalar-expression>)
default (shared | private | none)
schedule (<type>[, <chunk-size>)
shared (<list>)
private (<list>)
firstprivate (<list>)
lastprivate (<list>)
reduction (<operator> : <list>)
reduction (<intrinsic> : <list>)
copyin (<list>)
ordered
```

The `parallel do` construct above is equivalent to the following nested `parallel` and `do` constructs:

```
c$omp parallel [ <par-clause> [[,] <par-clause> ] ... ] <new-line>
c$omp do [ <do-clause> [[,] <do-clause> ] ... ] <new-line>
  <do-loop>
c$omp end do nowait <new-line>
c$omp end parallel <new-line>
```

where `<par-clause>` is one of the following:

```
if (<scalar-expression>)
default (shared | private | none)
shared (<list>)
private (<list>)
copyin (<list>)
```

and `<do-clause>` is one of the following:

```
schedule (<type>[, <chunk-size>)
firstprivate (<list>)
lastprivate (<list>)
reduction (<operator> : <list>)
reduction (<intrinsic> : <list>)
ordered
```

parallel sections

The `parallel sections` directive is a short form for a parallel region containing a single `sections` directive. If the `parallel sections` directive is encountered in the execution of the program while a parallel region is already active, then this directive is executed by a team of one thread and the entire construct is executed by each thread that encounters it.

```
c$omp parallel sections [ <clause> [[,] <clause> ] ... ] <new-
line>
[ c$omp section <new-line> ]
  <structured-block>
[ c$omp section <new-line>
  <structured-block>
.
.
. ]
c$omp end parallel sections <new-line>
```

where `<clause>` is one of the following:

```
if (<scalar-expression>)
default (shared | private | none)
shared (<list>)
private (<list>)
firstprivate (<list>)
lastprivate (<list>)
reduction (<operator> : <list>)
reduction (<intrinsic> : <list>)
copyin (<list>)
ordered
```

The `parallel sections` construct above is equivalent to the following nested `parallel` and `sections` constructs:

```
c$omp parallel [ <par-clause> [[,] <par-clause> ] ... ] <new-line>
c$omp sections [ <sec-clause> [[,] <sec-clause> ] ... ] <new-line>
[ c$omp section <new-line> ]
  <structured-block>
[ c$omp section <new-line>
  <structured-block>
.
.
. ]
c$omp end sections nowait <new-line>
c$omp end parallel <new-line>
```

where <par-clause> is one of the following:

```
if (<scalar-expression>)
default (shared | private | none)
shared (<list>)
private (<list>)
copyin (<list>)
```

and <sec-clause> is one of the following:

```
firstprivate (<list>)
lastprivate (<list>)
reduction (<operator> : <list>)
reduction (<intrinsic> : <list>)
ordered
```

Synchronization Directives

critical

The `critical` directive defines the scope of a critical section. Only one thread at a time is allowed inside the critical section.

```
c$omp critical [ (<name>) ] <new-line>
  <structured-block>
c$omp end critical [ (<name>) ] <new-line>
```

The name has global scope. Two `critical` directives with the same name are automatically mutually exclusive. All unnamed `critical` sections are assumed to map to the same name.

ordered

The `ordered` directive defines the scope of an ordered section. Only one thread at a time is allowed inside an ordered section of a given name.

```
c$omp ordered [ (<name>) ] <new-line>
  <structured-block>
c$omp end ordered [ (<name>) ] <new-line>
```

An optional name can be given to an ordered section. Named ordered sections are a KAP/Pro Toolset extension to OpenMP. Ordered sections are allowed either lexically within or outside of parallel regions, but when they occur lexically outside of a parallel region, they must be unnamed.

The ordered section must be dynamically enclosed in a `do` or `sections` construct with the `ordered` clause. It is an error to use this directive when not within the dynamic scope of one of the above constructs with an `ordered` clause.

The semantics of an ordered section are defined in terms of the sequential order of execution for the construct. The threads are granted permission to enter the ordered section in the same order as the `do` iterations, `sections` would be executed in the sequential version of the code.

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

Only one ordered section with a given name may be encountered during the execution of each `do` iteration or `section`.

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

master

The section of code following a `master` directive is executed by the master thread of the team.

```
c$omp master <new-line>
  <structured-block>
c$omp end master <new-line>
```

Other threads of the team skip the following 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.

```
c$omp atomic <new-line>
  <assignment-statement>
```

where `<assignment-statement>` must have one of the following forms:

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

and where:

x is a scalar variable of intrinsic type.
<expr> is a scalar expression that does not reference x.
<intrinsic> is one of max, min, iand, ior, or ieor.
<op> is one of +, -, *, /, .and., .or., .eqv., or .neqv.

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

flush

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.

```
c$omp flush [ (<list>) ] <new-line>
```

The optional list may be used to specify variables 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.

```
c$omp barrier <new-line>
```

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 or other synchronization constructs.

Data Scope Attribute Clauses

default (shared | private | none)
shared (<list>)
private (<list>)

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 should reference distinct instances of variables or arrays, place the variable in the `private` list.

The `private` clause is allowed on `parallel`, `do`, `sections`, and `single` directives. The `default` and `shared` clauses are only allowed on `parallel` directives.

When a variable is not present in 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, but referenced, 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 (<list>)

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`, `do`, `sections` and `single` directives.

lastprivate (<list>)

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 `do`, the last lexical `section` of a `sections` construct, or the code enclosed

by a `single`, as appropriate. If the loop is executed and the `lastprivate` variable is not written in the final iteration of the index set for a `do`, or the last lexical `section` in a `sections` construct, then the value of the shared variable is undefined.

The `lastprivate` clause is allowed on `do`, `sections`, and `single` directives. The use of the `lastprivate` clause on a `single` is a KAP/Pro Toolset extension.

reduction (<operator>:<list>)
reduction (<intrinsic>:<list>)

A variable or array element in the `reduction` list is treated as a reduction by creating a `private` temporary for that variable and updating 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`, `do`, `sections`, and `single` directives. The use of the `reduction` clause on a `single` is a KAP/Pro Toolset extension.

```
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 do
       do i = 1, n
           maxa_local = max ( maxa_local, a(i) )
       enddo
c$omp end do nowait
c$omp critical
       maxa = max (maxa, maxa_local)
c$omp end critical
c$omp end parallel
```

copyin (<list>)

The `copyin()` clause applies only to `threadprivate` 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 in a parallel region.

This notion is supported by the following two types of private `COMMON` directives:

1. `c$omp threadprivate`, and
2. `c$omp instance parallel` (a KAP/Pro Toolset extension defined by ANSI X3H5).

threadprivate

The `c$omp threadprivate` directive creates thread–private copies of one or more `COMMON` blocks for use within parallel regions. This directive can also be used as a migration feature for Cray's `taskcommon`. The `copyin` clause on parallel directives, or the `c$omp copy new` directive, can be used as a migration feature for SGI's `copyin` directive. A `threadprivate COMMON` block is always private in each parallel region of each routine where the `threadprivate` directive appears.

instance parallel

The `c$omp instance parallel` directive is a KAP/Pro Toolset extension that implements ANSI X3H5 semantics for private `COMMON` blocks. These semantics allow private `COMMON` blocks to be privatized selectively in one or more parallel regions in a routine. An instance parallel private `COMMON` block is declared via a `c$omp instance parallel` directive, but it is not private in a

particular parallel region unless a c$omp new or c$omp copy new directive appears at that parallel region. The absence of one of these directives means that all threads access the same storage in a particular parallel region.

Declaring Private Commons

Private COMMON blocks are declared by the c$omp threadprivate or c$omp instance parallel directives. The syntax for the two directives is as follows:

```
c$omp threadprivate ([/cmn1/ [,/cmn2/] ])
c$omp instance parallel ([/cmn1/ [,/cmn2/] ])
```

These directives are placed in the declaration section of a routine. If a COMMON block appears in a c$omp threadprivate or c$omp instance parallel directive in one routine, the COMMON block must appear in that same directive in all routines where the COMMON block is used.

The c$omp threadprivate directive assigns each specified COMMON block to the master thread and creates an uninitialized copy of the COMMON block for each additional thread. The copyin clause can be used to initialize a threadprivate COMMON block from the master's copy.

The c$omp instance parallel directive creates a copy of each specified COMMON block for each thread, but only for parallel regions where a c$omp new or c$omp copy new directive is encountered. The c$omp copy new directive can be used to initialize an instance parallel COMMON block from the sequential copy.

Allocating Private Commons

Thread–private copies for threadprivate COMMON blocks are always allocated, implicitly, at each parallel region. In instance parallel COMMON blocks, however, private allocation occurs only for parallel regions where c$omp new or c$omp copy new directives appear. This section describes how c$omp new and c$omp copy new are used with c$omp instance parallel. These directives are KAP/Pro Toolset extensions to OpenMP.

The c$omp new or c$omp copy new directives associated with instance parallel COMMON blocks must appear in the declaration section of a routine, or immediately following the parallel directive when an instance parallel COMMON block is used in a routine containing a parallel region.

When a thread inside a parallel region encounters a c$omp new directive, the named private COMMON blocks are allocated and initialized if they have not already been allocated and initialized. If the thread had previously allocated and initialized the COMMON block in a different parallel region, this space is simply reused.

Whether a COMMON block is private to each thread for a given parallel region depends on whether a c$omp new or c$omp 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 c$omp new directive is as follows:

```
c$omp new ([/cmn1/ [,/cmn2/] ])
```

The c$omp copy new directive is similar to the c$omp new directive, except that c$omp copy new copies the values in the original COMMON block into the thread-private copies at the beginning of each applicable parallel region. In contrast, the c$omp new directive only initializes the thread–private copies for the first parallel region encountered.

The syntax for using the c$omp copy new directive is as follows:

```
c$omp copy new ([/cmn1/ [,/cmn2/] ])
```

The behavior is undefined when COMMON blocks are allocated and initialized with both a c$omp new and a c$omp copy new directive within a single parallel region. If any thread executes a c$omp new or c$omp copy new directive for a COMMON block, every thread must execute that directive for that COMMON block.

Scheduling Options

Scheduling options are used to specify the iteration dispatch mechanism for each parallel loop do construct. They 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 default scheduling mechanism when the source file is being processed by . 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 each scheduling option. Assume the following: the loop has *l* iterations, *p* threads execute the loop, and *n* is a positive integer specifying the chunk size.

Table 3-1 Scheduling Options

Scheduling Designator	Chunk	Meaning

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*. Static even scheduling is the default method of loop scheduling.
		To specify static even scheduling from the command line use:
		`-WG,-scheduling=e`
		or
		`-WG,-scheduling=e -WG,-chunk=<integer>`
		[same as **-WG,-scheduling=e**, the chunk size has no effect]
		To specify static even scheduling with the `SCHEDULE` directive use:
		`schedule (static)`
		There is no `even` argument for the `schedule` directive. To perform even scheduling using the `schedule` directive, use the `static` argument without specifying a chunk size.
		To specify static even scheduling with the `OMP_SCHEDULE` environment variable use:
		`OMP_SCHEDULE = static`
		There is not an `even` argument for the `OMP_SCHEDULE` environment variable. To perform even scheduling using the `OMP_SCHEDULE` environment variable, use the `static` argument without specifying a chunk size.

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.
		To specify static scheduling from the command line use:
		`-WG,-scheduling=s -WG,-chunk=<integer>`
		or
		`-WG,-scheduling=s` [specifies even scheduling when **-WG,-chunk** is not stated]
		To specify static scheduling with the `schedule` directive use:
		`schedule (static[,<integer>])`
		To specify static scheduling with the `OMP_SCHEDULE` environment variable use:
		`OMP_SCHEDULE = static[,<integer>]`

i	ignored	Static interleaved scheduling. The chunk size has no effect here. Thread *i* is statically dispatched iterations *i*, *i+p*, *i+2p*, …
		To specify static interleaved scheduling from the command line use:
		`-WG,-scheduling=i`
		or
		`-WG,-scheduling=i -WG,-chunk=<integer>` [same as **-WG,-scheduling=i**, the chunk size has no effect]
		To specify static interleaved scheduling with the `schedule` directive use:
		`schedule (interleaved)`
		To specify static interleaved scheduling with the `omp_schedule` environment variable use:
		`OMP_SCHEDULE = interleaved`
		Interleaved scheduling is a KAP/Pro Toolset extension to OpenMP.

Scheduling Designator	Chunk	Meaning

d	n	Dynamic scheduling with a chunk size of *n*. *n* iterations are dispatched dynamically to each thread.
		To specify dynamic scheduling from the command line use:
		`-WG,-scheduling=d -WG,-chunk=<integer>`
		To specify dynamic scheduling with the `schedule` directive use:
		`schedule (dynamic[,<integer>])`
		To specify dynamic scheduling with the `OMP_SCHEDULE` environment variable use:
		`OMP_SCHEDULE = dynamic[,<integer>]`
		If no chunk size is specified, a size of 1 will be used.

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.
		To specify guided scheduling from the command line use:
		`-WG,-scheduling=g -WG,-chunk=<integer>`
		To specify guided scheduling with the `schedule` directive use:
		`schedule (guided[,<integer>])`
		To specify guided scheduling with the `OMP_SCHEDULE` environment variable use:
		`OMP_SCHEDULE = guided[,<integer>]`
		If no chunk size is specified, a size of 1 will be used.

Scheduling Designator	Chunk	Meaning

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.
		To specify trapezoidal scheduling from the command line use:
		`-WG,-scheduling=t -WG,-chunk=<integer>`
		To specify trapezoidal scheduling with the `schedule` directive use:
		`schedule (trapezoidal,<integer>)`
		To specify trapezoidal scheduling with the `OMP_SCHEDULE` environment variable use:
		`OMP_SCHEDULE = trapezoidal,<integer>`
		If no chunk size is specified, a size of 1 will be used.
		Trapezoidal scheduling is a KAP/Pro Toolset extension to OpenMP.

Scheduling Designator	Chunk	Meaning

r	ignored	Runtime scheduling specifies the scheduling will be determined via the `OMP_SCHEDULE` environment variable at run time.
		
		To specify scheduling at runtime, use the following from the command line:
		
		`-WG,-scheduling=r`
		
		To specify runtime scheduling with the `schedule` directive use:
		
		`schedule (runtime)`
		
		To specify runtime scheduling with the `OMP_SCHEDULE` environment variable use:
		
		`OMP_SCHEDULE = <string>[,<integer>]`
		
		Where `<string>` is one of `static`, `interleaved`, `dynamic`, `guided`, or `trapezoidal` and the optional `<integer>` parameter is the chunk size for the dispatch method.
		
		If the `OMP_SCHEDULE` environment variable is not set, then Guide assumes a default value of "`dynamic,1`".

Scheduling Options Using Directives

The list below shows the syntax for specifying scheduling options with the `do` and `parallel do` directives.

```
schedule (static      [,<integer>] )
schedule (dynamic     [,<integer>] )
schedule (guided      [,<integer>] )
schedule (trapezoidal [,<integer>] )
schedule (interleaved)
schedule (runtime)
```

Where the `<integer>` parameter is a chunk size for the dispatch method. If `<integer>` 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 30 for a complete description of the scheduling options.

The default is `schedule (static)`.

Scheduling Options Using Environment Variables

The `OMP_SCHEDULE` environment variable sets, at run time, scheduling options for loops containing a `schedule(runtime)` clause. The syntax for this 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.

Environment Variables

Some environment variables may need to be set before running Guide generated programs.

KMP_BLOCKTIME=<integer>[<character>]

This variable specifies the number of milliseconds that the libraries should wait after completing the execution of a parallel region before putting threads to sleep. Use the optional suffix `s`, `m`, `h`, or `d` to specify seconds, minutes, hours, or days. The default is `1s` or one second. This default may be too large if threads will be used to execute other threaded code between parallel regions. The default may be too small if threads are reserved solely for the use by the Guide library.

KMP_LIBRARY=<string>

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

- serial
- turnaround
- throughput

See Chapter 5, “Libraries,” beginning on page 63 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 **b**, **k**, or **m** to specify bytes, kilobytes, or megabytes. The default is **1m** or one megabyte. This default value may be too small if many private 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 `guide_stats`.

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

1. This expands into the hostname of the machine running the parallel program.
2. This expands into a unique numeric identifier for this execution of the program. It is the process identifier of the program.
3. This is replaced with the value of the `OMP_NUM_THREADS` environment variable which determines the number of threads that are created by the parallel program.

OMP_DYNAMIC=<boolean>

The `OMP_DYNAMIC` environment variable enables or disables dynamic adjustment of the number of threads between parallel regions. Enabling dynamic threads allows the Guide library to adjust the number of threads in response to system load. Such an adjustment can improve the turnaround time for all jobs on a loaded system. 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.

The default value is `FALSE`.

OMP_NUM_THREADS=<integer>

The OMP_NUM_THREADS environment variable is used to specify the number of threads. The <integer> is a positive number. Performance of parallel programs usually degrades when the number of threads exceeds the number of physical processors.

The special value ALL is also allowed. A value of ALL specifies that one thread will be created per processor on the machine. This is the default.

OMP_SCHEDULE=<string>[,<integer>]

The OMP_SCHEDULE environment variable controls the schedule type and chunk size for do constructs with a schedule(runtime) clause or those with no schedule 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 “Scheduling Options” on page 28.

OMP_NESTED=<boolean>

The OMP_NESTED environment variable controls whether nested parallelism is enabled at run time. Nested parallelism is currently unimplemented, so this variable has no effect. Allowed values are TRUE and FALSE, and the default value is FALSE.

LD_LIBRARY_PATH=<path>

This variable is used to specify an alternate path for the run time libraries. You may need to set this variable to the directory where the guide libraries were installed when you run your application if you compile with shared objects or use dynamic linking.

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`guidef77` or `guidef90`, . 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. **-debug**) 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 **-debug** 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, but creates object files without the `G_` prefix.

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 instructs the driver to print a usage message and exit.

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 **-onetrip** options to Guide, the appropriate `guidef` driver option would be **-WG,-o=1,-onetrip**.

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

The **-WGcpp** option forces the C preprocessor to be run on all source files.

WGcpp=<path>

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

If no preprocessor path is specified, the **-WGcpp=<path>** option forces the preprocessor to be run on all input to the compiler. Specifying a preprocessor path does *not* force preprocessing. In order to force all compiler input to be processed by another preprocessor, use the following options:

```
-WGcpp=/bin/cpp2 -WGcpp
```

WGf77=<path>

This is an alternate form of the **-WGcompiler** option above.

WGf90=<file>

This is an alternate form of the **-WGcompiler** option above.

WGfortran=<path>

This is an alternate form of the **-WGcompiler** option above.

WGftn=<path>

This is an alternate form 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 removed 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.

WGlibpath=<path>

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

WGnocpp

-WGnocpp prevents `guidef` from calling the C preprocessor for FORTRAN source files with the `.F` suffix. Normally, the driver will invoke the C preprocessor in situations where the native compiler would do the same. On most UNIX platforms, the C preprocessor is invoked for files with a capital `F` in the extension (`.F`, `.F77`, `.F90`).

WGnokeep

Use **-WGnokeep** to force output and temporary C preprocessor files to be removed. The presence of this flag overrides any previous 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.

WGnowork

When **-WGnowork** is specified, the driver shows, but does not execute, the commands it would normally run.

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 the default `cppG_file1.f` and `G_file1.f`.

WGsrcdir

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

WGstatic_library

Normally, the driver links applications against shared versions of the Guide libraries. To specify static linking instead, supply the **-WGstatic_library** flag.

WGuser

The **-WGuser** driver option allows a string to be invoked as a command on each source file specified on the driver command line. This command is invoked after the C-preprocessor (`cpp`) but before Guide source-file processing. The syntax for using the **-WGuser** driver option is as follows:

```
-WGuser=<cmd>[%_<options>][%_%i][%_<options>][%_%o][%_<options>]
```

Where `<cmd>` is the name of the command to be executed.

Spaces are not allowed in the command string. If a space is required then the token `%_` is used to generate a space.

The `%i` and `%o` arguments are tokens for the filenames passed to the command. The `%i` refers to the input file and `%o` refers to output file. The order of these tokens and the options in the command string corresponds to the order of the input/output files and options for the command to be executed. Both filename tokens are optional and are case-insensitive. If `%i` is omitted, then the input file is piped as `stdin`. If `%o` is omitted, then the `stdout` output is redirected to a disk file. The output file name is created by prepending the text `usrG_` to the name of the file being processed by Guide. For example, if the file being processed is named `x.f`, the output file would be named `usrA_x.f`.

The return status of the command is checked, and success is assumed to be zero. Failed files will not be processed further.

Table 4-1 gives the command that the driver will execute when `x.f` is the file being processed.

Table 4-1 WGuser Examples

Switch	Command Executed
`-WGuser=cat`	`cat < x.f > usrG_x.f`
`-WGuser=cmd1%_%i`	`cmd1 x.f > usrG_x.f`
`-WGuser=cmd2%_%o`	`cmd2 usrG_x.f < x.f`
`-WGuser=cmd3%_%o%_%i`	`cmd3 usrG_x.f x.f`
`-WGuser=cmd4%_-G`	`cmd4 -G < x.f > usrG_x.f`

WGuser2

This option is identical to **-WGuser**, except the command is run after Guide source file processing but before the compiler.

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, that 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**.

Table 4-2 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

Guide can generate a listing file that contains information about the transformations and optimizations it performs. The options in this category control the information Guide includes in its listing file. When the **-listoptions** argument includes “`k`”, Guide will itemize the values of all options in the listing file.

Advanced Optimization

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

FORTRAN Dialect

These options specify the dialect of FORTRAN in use.

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 present in the code.

Guide Options Table

In Table 4-2, 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 50.

Table 4-2 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>	[no]list=<file>	nolist
Listing:		
lines=<integer>	ln=<integer>	55
listoptions=<string>	lo=<string>	ko
suppress=<string>	su=<string>	nosuppress
Advanced Optimization:		
[no]assume	[n]as=<string>	cel
[no]concurrentize	[no]conc	noconcurrentize
minconcurrent=<integer>	mc=<integer>	1000

Table 4-2 Guide Options (Continued)

Long Name	Short Name	Default Setting
FORTRAN Dialect:		
alignmax=<integer>	alignmax=<integer>	platform dependent
[no]blank_padding	[n]bp	platform dependent
[no]case	[n]case	nocase
[no]datasave	[n]ds	datasave
[no]dlines	[n]dl	nodlines
include=<path>	inc=<path>	noinclude
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>	nosyntax
[no]type	[n]ty	notype
Directive Recognition:		
[no]directives=<string>	[n]dr=<string>	p
[no]ignoreoptions	[n]ig	noignoreoptions
[no]openmpcc_lines	[no]openmpcc_lines	openmpcc_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. The default listing file name is `G_<file>.lst`.

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 is an expert option that you would normally not use. The **-alignmax** option tells Guide the size of the largest data type the native compiler will pad in a common block or VAX structure in order to achieve natural alignment. The default value is platform-specific, and the driver provides an appropriate value based on the command line switches passed to the native compiler.

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 The '`l`' value applies only to parallel loops generated automatically from array syntax by Guide, when **-concurrent** is specified. When '`l`" is specified, Guide ensures the shared copy of each private variable is updated after a parallel loop, using the value assigned in the loop's final iteration.

This behavior is analogous to using the `lastprivate()` instead of `private()` for all private variables. If '`l`' is omitted, Guide will assume private variables do not need their final values stored in the shared copy.

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 input lines with trailing blanks. The default value of this option varies by platform and is chosen to match the behavior of the native compiler.

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 ignore case 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 by using scheduling options on the `parallel do` or `do` directive. The default chunk size is 1. See “Scheduling Options” on page 28 for descriptions of the FORTRAN **chunk** options.

cmp[=<file>]

The **-cmp=<file>** option instructs Guide to place the optimized FORTRAN program in a specified file. The default name of this file is derived from the input filename by adding `G_` to the beginning of the input filename. If **-cmp=<file>** is specified, the FORTRAN output file is written to the specified file. If **-cmp** is specified with no argument, then the output is written to standard output.

concurrentize, conc
noconcurrentize, noconc

Guide uses the **-concurrentize** switch to enable parallelization of loops derived from array syntax only. This option can be used to generate parallel loops from FORTRAN 90 array syntax. Guide will only run a loop in parallel if it determines there is sufficient work to benefit from parallelism. You can adjust Guide's idea of sufficient work via the **-minconcurrent** option. 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
```

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. This is the default.

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. Multiple **-include** options may be used to specify multiple include directories. Normally, you should use your native compiler's include option, often **-I**, instead.

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>]
nolist

The **-list** option informs Guide where to place the listing file. When no filename is specified, 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 `.out`. If a filename is specified, then the listing file is written to that 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 produce no listing file, enter **-nolist** on the command line. The default value is **-listoptions=ko**.

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. 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 that 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 that 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 that are defined before use in all paths from all entry points and that 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 Guide 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 28 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, **nosyntax**, instructs Guide to accept a superset FORTRAN 77 and FORTRAN 90.

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.

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 those options. The format of this line is:

```
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 three libraries, a development library, a management and monitoring library, and a production library. The production library is called the *guide* library. 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. The management and monitoring library is called the *guide_perview* library. It can be used to interactively and remotely monitor and manage the parallel performance of a running application program. This library degrades application performance slightly also. All three libraries contain the serial, turnaround, gang, and throughput modes described below. These modes are selected by using the `KMP_LIBRARY` environment variable at run-time; see “KMP_LIBRARY=<string>” on page 35.

Serial

The serial mode forces parallel applications to 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 mode is designed to keep all of the processors active and involved in the parallel computation to minimize the execution time of a single job. In this mode, the worker threads actively wait for more parallel work, without yielding to other threads.

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

Gang

This mode is identical to the turnaround mode, except gang scheduling is enabled on systems that support it.

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. In this mode, the worker threads will yield to other threads while waiting for more parallel work.

The throughput mode is designed to make the program aware of its environment (i.e. the system load) and to adjust its resource usage to produce efficient execution in a dynamic environment.

This is the default.

The Guide_stats Library

The *guide_stats* library is designed to provide you with detailed statistics about a program's execution. These statistics help you 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 program units that cause program termination are compiled with Guidef.

This library may minimally degrade application performance compared to the *guide* library by an amount proportional to the frequency that the OpenMP directives are encountered.

The resulting statistics are most easily viewed and analyzed by using GuideView, discussed in Chapter 6, "GuideView," beginning on page 73.

The Guide_perview Library

The *guide_perview* library is part of the interactive parallel performance monitoring and management tool called *PerView*. Using *PerView*, application users can remotely monitor parallel performance and application progress, modify the number of threads, switch between dynamic and static thread count, and pause or abort parallel applications. When using the *guide_perview* library, make sure that the main program and any program units that cause program termination are compiled with `guidef`.

In the current version of Guide, the *guide_perview* library also provides all the functionality of the *guide_stats* library. Future versions are not guaranteed to support the *guide_stats* library functionality. The *guide_perview* library currently enables additional runtime checks that may aid in program debugging. Currently, this library may minimally degrade application performance compared to the *guide*

library by an amount proportional to the frequency that the OpenMP directives are encountered.

See “PerView,” beginning on page 77 for more information about the use of the *guide_perview* library.

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
```

To use the *guide_perview* library, use the **-WGperview** command line option to `guidef`. To switch between the *guide*, *guide_stats*, and *guide_perview* libraries, only relinking is necessary. Recompilation is not needed.

External Routines

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

The use of these routines is discouraged; using them requires that the application be linked with one of the Guide libraries, even when the code is executed sequentially. In addition, using these routines makes validating the program with Assure more difficult or impossible.

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 `threadprivate` directive, rather than by explicit expansion and indexing with `omp_get_thread_num()`. Appendix A, “Examples,” beginning on page 95, contains examples of coding styles that avoid the use of these routines.

mppbeg()
mppend()

These routines are not necessary if the main program unit and all exit points are compiled using Guidef or Guidec, Guidef's C language counterpart. If this isn't the case, you must ensure that `mppbeg()` is called at the beginning of the main program and that `mppend()` is called at all points that cause program termination.

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

kmp_get_blocktime

This routine returns the integer value of time, in milliseconds, that the Guide libraries wait after completing the execution of a parallel region before putting threads to sleep. This value can be changed via the `kmp_set_blocktime` routine or the `KMP_BLOCKTIME` environment variable. See the description of the `KMP_BLOCKTIME` environment variable on page 35 for more information.

kmp_get_library

This routine returns an integer value that designates the version of the Guide run time library being used. This value can be used as the parameter to subsequent calls to `kmp_set_library`. The library setting can also be changed via the `kmp_set_library_xxx` calls or the `KMP_LIBRARY` environment variable.

kmp_get_stacksize (<integer>)

This routine returns the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can be changed via the `kmp_set_stacksize` routine, prior to the first parallel region or via the `KMP_STACKSIZE` environment variable. See the description of the `KMP_STACKSIZE` environment variable on page 36 for more information.

kmp_set_blocktime (<integer>)

This routine sets the number of milliseconds that the Guide libraries wait after completing the execution of a parallel region before putting threads to sleep. This value can also be changed via the `KMP_BLOCKTIME` environment variable. See the description of `KMP_BLOCKTIME` on page 35 for more information.

In order for `kmp_set_blocktime` to have an effect, it must be called before the beginning of the first (dynamically executed) parallel region in the program.

kmp_set_library (<integer>)

This routine selects the Guide run time library. The parameter value corresponds to a value previously returned by a call to `kmp_get_library`. To determine the values of this parameter that correspond to particular libraries, call the `kmp_set_library_xxx` routines and then call the `kmp_get_library` routine to obtain the parameter values. The library setting can also be changed via the `KMP_LIBRARY` environment variable.

kmp_set_library_serial

This routine selects the Guide serial run time library. The library setting can also be changed via the `kmp_set_library` call or the `KMP_LIBRARY` environment variable.

kmp_set_library_throughput

This routine selects the Guide throughput run time library. The library setting can also be changed via the `kmp_set_library` call or the `KMP_LIBRARY` environment variable.

kmp_set_library_turnaround

This routine selects the Guide turnaround run time library. The library setting can also be changed via the `kmp_set_library` call or the `KMP_LIBRARY` environment variable.

kmp_set_stacksize (integer)

This routine sets the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can also be changed via the `KMP_STACKSIZE` environment variable. See the description of `KMP_STACKSIZE` on page 36 for more information.

In order for `KMP_SET_STACKSIZE` to have an effect, it must be called before the beginning of the first (dynamically executed) parallel region in the program.

OMP_DESTROY_LOCK(<var>)

This routine ensures that the lock pointed to by the parameter `<var>` is uninitialized. No thread may own the lock when this routine is called. The `<var>` parameter must be a lock variable that was initialized by the `OMP_INIT_LOCK()` routine.

OMP_GET_MAX_THREADS()

This routine returns the maximum number of threads that are available for parallel execution. 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 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 or decrease between parallel regions, but will never exceed the OMP_NUM_THREADS environment variable value.

When called outside a parallel region, this function returns 1.

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.

When called from a serial region or a serialized parallel region, this function returns 0.

OMP_INIT_LOCK(<var>)

This routine initializes a lock associated with the lock variable <var> for use by subsequent calls. The lock variable must of integer type and of KIND large enough to hold an address. The initial state is unlocked.

The lock variable, <var>, must only be accessed through the OpenMP library lock routines.

OMP_SET_LOCK(<var>)

This routine forces the executing thread to wait until the specified lock is available. If the lock is not available, the thread is blocked from further execution until the thread is granted ownership of the lock. <var> must be a lock variable that was initialized by the OMP_INIT_LOCK() routine.

Signal Handling

In order for interrupts and runtime errors to be handled correctly during parallel execution, the Guide libraries normally install their own handlers for interrupt signals such as SIGHUP, SIGINT, SIGQUIT, and SIGTERM and for runtime error signals such as SIGSEGV, SIGBUS, SIGILL, SIGABRT, SIGFPE, and SIGSYS.

The Guide libraries normally install their handlers at the beginning of the first (dynamically executed) parallel region in the program. These handlers remain active until the end of program execution, throughout the parallel and remaining serial portions of the program.

The Guide libraries provide a mechanism for allowing user–installed signal handlers. If the program installs a handler for a signal before the beginning of the first parallel region, the libraries will not install their handlers for that signal.

OMP_TEST_LOCK(<var>)

This routine tries to obtain ownership of the lock, but does not block execution of the calling thread if the lock is not available. The routine returns `.TRUE.` if the lock was successfully obtained; otherwise, it returns `.FALSE.` The `<var>` must be a lock variable that was initialized by the `OMP_INIT_LOCK()` routine.

OMP_UNSET_LOCK(<var>)

This routine releases the executing thread from ownership of the lock. The behavior is undefined if the executing thread is not the owner of the lock. `<var>` must be a lock variable that was initialized by the `OMP_INIT_LOCK()` routine.

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 the statistics file that is output when a Guide instrumented program is run. See "Libraries," beginning on page 63 to learn how to build an instrumented executable. The syntax for invoking GuideView is as follows:

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

The *file* arguments are the names of the statistics files created by Guide runs that used the *guide_stats* library (see Chapter 5). Optional GuideView arguments are the topic of the next section.

The GuideView browser looks for a configuration file named `GVproperties.txt` 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 control fonts, colors, window sizes, window locations, line numbering, tab expansion in source, and other features of the GUI.

An example 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 `<install_dir>` on your machine, the example initialization file will be in

`<install_dir>/class/example.GVproperties`.

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 its Help menu.

GuideView Options

mhz=<integer>

The **-mhz=<integer>** option denotes the processor rate in MHz 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 `<install_dir>` on your machine, this example file resides in `<install_dir>/class/guide.ovh`.

jpath=<file>

The **-jpath=<file>** option specifies the path to an alternate Java interpreter. This can be used to override the Java virtual machine selected at installation or to provide a path to the Java virtual machine if none was selected during installation.

WJ,[java_option]

The GuideView GUI is implemented in Java. The **-WJ** flag prefixes any Java option that should be passed to the Java interpreter.

Any valid Java interpreter option may be used; however, the options listed in the next section may be particularly beneficial when used with GuideView to enhance the performance of the GUI.

Java Options

The **-WJ** flag must prefix Java options. 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**.

nojit

The **-nojit** option disables the Java just-in-time compiler. This Java feature can sometimes lead to incorrect Java behavior. Use **-WJ**,**-nojit** to disable the just-in-time compiler if you experience problems with the AssureView GUI.

CHAPTER 7

PerView

Introduction

PerView is an interactive parallel performance monitoring and management tool. With PerView, users of your application can remotely monitor parallel performance and application progress, modify the number of threads, switch between dynamic and static thread count, and pause or abort parallel applications.

Enabling the PerView Server

PerView makes its capabilities available through the use of a web server, embedded in the parallel application. By default, Guide does not include the PerView server in your application. Its functionality is only included when specifically requested.

Including the PerView server in your application is as simple as relinking your application with the *guide_perview* library, introduced in “Libraries,” beginning on page 63. To embed the PerView server in your application, add the **-WGperview** flag when linking with the Guidef driver. For example, to build a PerView-enabled application, issue the following commands:

```
guidef77 -c main.f
guidef77 -WGperview main.o
```

Security

The PerView server provides an access control mechanism, which limits unauthorized access to your parallel application at run-time. Access control is specified via the KMP_HTTP_ACCESS environment variable, the value of which behaves like a password. This variable can take on any string value, but the string should contain no white space. The value of KMP_HTTP_ACCESS is read once upon application execution, and the PerView server requires any connecting PerView client know this value.

If KMP_HTTP_ACCESS is not specified, the server disables access control, and clients can communicate without a password. This is the default.

Running with PerView

Using PerView is a two-step process. First, a PerView enabled parallel application is run, which listens for PerView client requests. During the execution of the parallel application, one or more PerView clients can connect to the server, to remotely monitor the application.

The server and client applications can be run on the same or different hosts.

Starting the Server

The server starts when the application begins running if the environment variable KMP_HTTP_PORT is set. If this variable is unset when the application starts, the server becomes inactive for the duration of the run. Normally, the PerView server serves documents from and below a top-level directory. This top-level directory is specified via the KMP_HTTP_HOME environment variable.

The following paragraphs detail the environment variables used by the PerView server.

KMP_HTTP_PORT=<port>

This variable specifies the network port on which the server will listen. It should be a positive integer larger than 1024.

If KMP_HTTP_PORT has value 0 or is unspecified, the PerView server is disabled. This is the default.

KMP_HTTP_HOME=<path>

In addition to its built-in documents, the PerView server can serve documents out of a "public_html" directory. This variable specifies the top-level directory that contains the public_html directory. The default value is the current directory, ".", so files in ./public_html will be available through the server. If you specify a valid directory path, the PerView server will instead serve files from <path>/public_html.

Documents located in and below the public_html directory are accessible via a standard Web browser, such as Netscape or Internet Explorer, via the URL "http://<host>:<port>/". If a password is specified, the URL is instead "http://<host>:<port>/cgi-pwd/<password>/".

To disable this feature, set KMP_HTTP_HOME=/dev/null or any non-existent directory.

KMP_HTTP_ACCESS=<password>

Using this variable, you can limit access to a running parallel application to those who know the password given in <password>. The password is an arbitrary string containing no white space characters.

Starting the Client

The PerView client, or simply PerView, communicates with the server in the application via a network connection, specified by two values: a host name and a port number. The correct password must also be used if the KMP_HTTP_ACCESS variable was set before running the application.

To start the PerView client, type:

```
perview <host> <port>
```

or

```
perview <host> <port> <password>
```

The following example illustrates the use of PerView on two machines, named “server” and “desktop”. The application runs on server but is monitored from desktop:

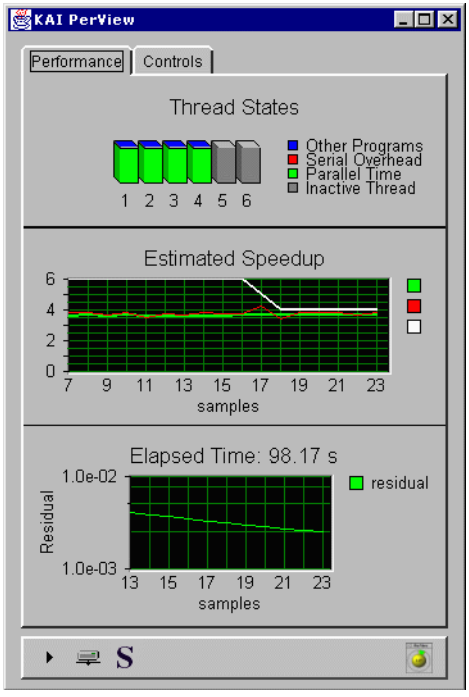
```
server % guidef77 –o mondo mondo.f -WGperview
server % setenv KMP_HTTP_PORT 8000
server % setenv KMP_HTTP_ACCESS secret
server % ./mondo

desktop % perview server 8000 secret
```

Multiple clients can simultaneously communicate with each PerView server, to allow monitoring from more than one location.

Using PerView

Once PerView has started and has connected to the server, it presents its main screen, shown in Figure 7-1. The PerView interface consists of two “views” of displays and controls, selectable by the tabs labeled Performance and Controls.

Figure 7-1



Performance

The Performance view consists of three panels, displaying thread states, projected speedup, and progress. The thread states panel shows the state of each OpenMP thread present in the application, by displaying one stacked bar graph per thread. The height of the bar represents 100% of each thread's time. The bar is divided into time spent doing productive work (green), time lost to parallel overheads and serial waiting time (red), and time lost due to excess load on the machine (blue). Inactive threads are shown in gray.

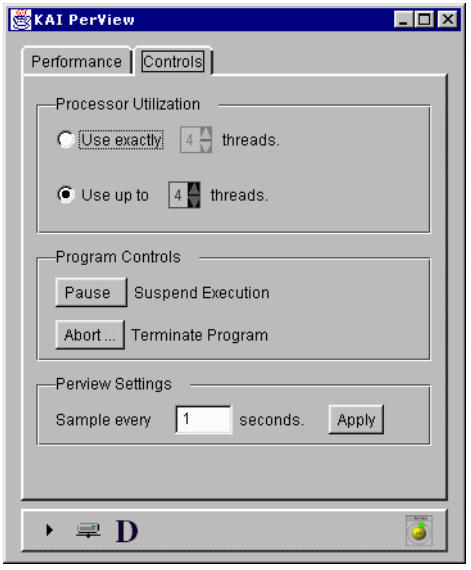
PerView uses this thread state data to estimate the parallel speedup of the application. This instantaneous speedup estimate is plotted, along with its time-averaged value and the thread count, in the center panel. PerView contacts the server at regular intervals to obtain new data. Each data set is one sample, and the speedup graph is plotted in terms of these samples.

The bottom panel displays the progress of the application. By default, only the elapsed time since the beginning of the application run is shown here. With the application's cooperation, however, PerView can display a percent completed graph, a string representing progress, or a convergence graph. See "Progress Data" on page 84 for details.

Controls

Using the Controls panel, shown in Figure 7-2 you can modify the parallel behavior of the application, to respond to changing conditions on the machine where it is running.

Figure 7-2



You might reduce the number of threads being used by an application, for example, to make room for another application to start. To adjust the number of threads, click on the up and down arrows in the Processor Utilization group to set the desired number of threads. To allow an application to monitor and automatically adjust its own thread count, select Use up to N threads in the top panel.

To temporarily suspend the application, click on Pause in the Program Controls group. The button text changes to Resume once the application has been paused. When the Resume button is pressed, the application resumes processing.

The Abort… button can be used to prematurely terminate the application.

The Perview Settings group contains a sampling interval control. This specifies how frequently PerView contacts the server for new data. To change the sampling interval type to a new, positive integer, then press Apply.

Status Bar

The bottom of the PerView window contains a status bar, shown in Figure 7-3. The icons in the status bar summarize the state of the application and PerView's connection to it.

Figure 7-3

The application status icon uses familiar symbols to represent whether the application is running (▸), paused (⏸), or complete (■).

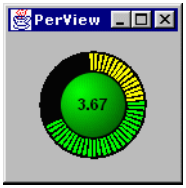
The connection icon indicates whether PerView is connected to the application. When the connection is broken, due to application completion, network failure, or application failure, the icon is obscured by a large, red X.

The dynamic threads icon indicates with an "S" or "D", respectively, whether the application's thread count is static (fixed) or dynamic (variable).

Minimal Monitor

The rightmost icon on the status bar is the "minimize" button. Clicking this button replaces the PerView screen with a minimal view, shown in Figure 7-4, suitable for general performance monitoring.

Figure 7-4



This view consists of a colored button, surrounded by a “marching” segments performance display. The colored button shows the current value of the estimated speedup in its center. The button is green, yellow, or red, depending on the value of the estimated speedup, relative to the number of threads in use.

The marching display consists of colored rays, emanating from the button and representing the time history of the button’s color. Using this display, you can get recent performance information at a glance. An all green display is ideal. Occasional yellow or red rays are normal, but a display dominated by yellow or red usually requires attention. Green indicates good projected speedup, yellow represents marginal performance, and red indicates parallel performance problems.

Click on the colored button to return to the detailed view and, if necessary, adjust the processor utilization.

Progress Data

By default, PerView displays the elapsed time in the bottom panel of the Performance view. This area, however, is provided for you to communicate more detailed information about your application’s progress to the user. Using a simple API, you can enable a progress meter, showing percent complete, an X-Y graph, showing the evolution of a convergence variable or other data, or simply display a string, representing the current phase of the computation.

Progress Bar

The progress bar is automatically displayed in PerView when you provide progress information to the PerView server via the `kweb_set_meter` library routine. The interface to this routine is:

```
call kweb_set_meter(meter_name, icurrent, istart, iend)
```

`Meter_name` is a string value used to label this meter. It is unused at this time.

`icurrent`, `istart`, and `iend` are integer values, representing the current, beginning, and ending values of a computation, such as a time-stepping loop.

The progress bar computes percent complete as `(icurrent-istart)/(iend-istart)`.

The PerView client computes a percentage complete from these values and displays it in a progress meter.

Progress Graph

The progress graph is automatically displayed in PerView when you provide progress information to the PerView server via the `kweb_set_residual` library routine. The interface to this routine is:

```
call kweb_set_residual(meter_name, current, ymin, ymax)
```

`Meter_name` is a string value used to label this meter. It is unused at this time.

`current` is a double precision value representing the data to be plotted as a function of time.

`ymin` and `ymax` are double precision values representing initial minimum and maximum `Y` coordinate limits for the graph.

Progress String

The progress string is automatically displayed in PerView when you provide progress information to the PerView server via the `kweb_set_string` library routine. The interface to this routine is:

```
call kweb_set_string(meter_name, current_phase)
```

`meter_name` is a string value used to label this meter. It is unused at this time.

`current_phase` is a string value used to describe the current state of the application. It could be used, for example, to present the major phases of a computation, such as problem setup, solution, and I/O.

Extending PerView

Both the PerView server and client are extensible, to allow application-specific data and displays. Please contact us at *kpts@kai.com* for more information.

CHAPTER 8

Directive Translation

Many programs written with older directive-based parallel programming models can be easily moved to equivalent OpenMP implementations. While the translation is often simple, it can also be tedious. Guide includes a number of translator scripts, designed to automate much of the work involved in updating codes to OpenMP.

All of the translation scripts require Perl to operate. Perl is generally available on Unix systems, but is less frequently installed on Windows NT systems. Links to UNIX Perl source and Windows NT binaries are available from *http://www.kai.com/kpts/helpers*.

Most Unix systems can run the Perl-based translators as if they were executable files:

```
sgi2omp.pl program.f > program_omp.f
```

On NT systems, however, you may need to call the translator scripts directly from Perl, for example:

```
perl c:\KAI\guide36\bin\sgi2omp.pl program.f > program_omp.f
```

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$omp parallel`
`First cmic$ case`	`c$omp sections` `c$omp section`
`Subsequent cmic$ case`	`c$omp section`
`cmic$ endcase`	`c$omp end sections`
`cmic$ do parallel`	`c$omp do`
`cmic$ 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)`

Table 8-3 Cray to OpenMP Translations (Continued)

Cray	OpenMP

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`. Please see *http://www.kai.com/kpts/helpers* for links to Perl source and binaries.

Cray TASKCOMMON as opposed to OpenMP THREADPRIVATE

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.

With Cray `taskcommon`, the individual elements of a `taskcommon` can be placed in the private list of a `parallel do`. This is not supported in OpenMP.

SGI Directive to OpenMP Directive Translator

Programs which have been parallelized with SGI `c$` directives can be used as the basis for a port to Guide. The `sgi2omp.pl` program will help translate SGI directives into OpenMP directives.

The `sgi2omp.pl` program accepts as an argument the name of a file with SGI 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 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 "`sgi2omp.pl`" in the output will lead to places where `sgi2omp.pl` had trouble performing translations.

Table 8-4, "SGI to OpenMP 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 `sgi2omp.pl` when possible. A diagnostic is produced when there is not an equivalent OpenMP directive.

None of the SGI scheduling keywords are automatically translated by `sgi2omp.pl`. `Sgi2omp.pl` produces a diagnostic to assist in manually inserting scheduling keywords into the program.

Table 8-4 SGI to OpenMP Translations

SGI directive or clause or library routine	KAP/Pro Translation
`c$doacross`	`c$omp parallel do`
`c$ call mp_barrier`	`c$omp barrier`
`c$ call mp_setlock`	`c$omp critical`
`c$ call mp_unsetlock`	`c$omp end critical`
`mp_my_threadnum`	Not translated automatically, but can be translated using `omp_get_thread_num()`
`mp_numthreads`	Not translated automatically, but can be translated using `omp_get_num_threads()` and `omp_get_max_threads()`
`c$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 `sgi2omp.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 89, are also available for the `kap2omp.pl` program.

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

APPENDIX A

Examples

The following example programs illustrate the use of OpenMP directives.

do: 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 do` has a `nowait` because there is an implicit `barrier` at the end of the parallel region. Alternately, using the option **-optimize=1** would have also eliminated the `barrier`.

```
        subroutine do_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 do 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$omp end do nowait
c$omp end parallel
        end
```

do: Two Difference Operators

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

```
        subroutine do_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 do 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$omp end do nowait
c$omp do 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$omp end do nowait
c$omp end parallel
        end
```

do: Reduce Fork/Join Overhead

Routines `do_3a` and `do_3b` perform numerically equivalent computations, but because the `parallel` directive in routine `do_3b` is outside the `do j` loop, routine `do_3b` probably forms teams less often, and thus reduces overhead.

```
        subroutine do_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 do
            do i = 1, n
                a(i,j) = b(i,j) / a(i,j-1)
            enddo
c$omp end do nowait
c$omp end parallel
        enddo
        end


        subroutine do_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 do
            do i = 1, n
                a(i,j) = b(i,j) / a(i,j-1)
            enddo
c$omp end do nowait
        enddo
c$omp end parallel
        end
```

sections: Two Difference Operators

Identical to “do: Two Difference Operators” on page 97 but uses `sections` instead of `do`. Here the speedup is limited to 2 because there are only 2 units of work whereas in “do: Two Difference Operators” on page 97 there are *n-1* + *m-1* units of work.

```
        subroutine sections_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 sections
c$omp section
        do i = 2, n
            do j = 1, i
                b(j,i) = ( a(j,i) + a(j,i-1) ) / 2
            enddo
        enddo
c$omp section
        do i = 2, m
            do j = 1, i
                d(j,i) = ( c(j,i) + c(j,i-1) ) / 2
            enddo
        enddo
c$omp end sections nowait
c$omp end parallel
        end
```

single: Updating a Shared Scalar

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

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

sections: Updating a Shared Scalar

Identical to “single: Updating a Shared Scalar” on page 100 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 do
              do i = 1, n
                  a(i) = 1.0 / a(i)
              enddo
      c$omp sections
              a(1) = min( a(1), 1.0 )
      c$omp end sections
      c$omp do
              do i = 1, n
                  b(i) = b(i) / a(i)
              enddo
      c$omp end do nowait
      c$omp end parallel
              end
```

do: Updating a Shared Scalar

Identical to “single: Updating a Shared Scalar” on page 100 but using different directives.

```
        subroutine do_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 do
        do i = 1, n
            a(i) = 1.0 / a(i)
        enddo
c$omp end do
c$omp do
        do i = 1, 1
            a(1) = min( a(1), 1.0 )
        enddo
c$omp end do
c$omp do
        do i = 1, n
            b(i) = b(i) / a(i)
        enddo
c$omp end do nowait
c$omp end parallel
        end
```

parallel do: A Simple Difference Operator

Identical to “do: A Simple Difference Operator” on page 96 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 “sections: Two Difference Operators” on page 99 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 sections_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 section
              do i = 2, n
                  do j = 1, i
                     b(j,i) = ( a(j,i) + a(j,i-1) ) / 2
                  enddo
              enddo
      c$omp section
              do i = 2, m
                  do j = 1, i
                     d(j,i) = ( c(j,i) + c(j,i-1) ) / 2
                  enddo
              enddo
      c$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 do
        do i = 1, n
            do j = 1, m
                local_sum = local_sum + a(j,i)
            enddo
        enddo
c$omp end do nowait
c$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
                sum = sum + a(j,i)
            enddo
        enddo
        end
```

threadprivate: Private Common

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

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

threadprivate: Private Common and Master Thread

In this example, the value 2 is printed since the master thread's copy of a variable in a `threadprivate` privatizable `common` block is accessed within a `master` section or in serial code sections. If a `single` 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 threadprivate (/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$omp do                         ! common /private/
              do i = 1, n
                  call construct_data()! fills in array work()
                  call use_data()      ! uses array work()
              enddo
      c$omp end do 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 `threadprivate` blocks since `threadprivate` 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 do
        do i = 1, n
            work(i) = b(i,i) / 4.0
        enddo
c$omp end do nowait
c$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$omp do                            ! common /private/
        do i = 1, m
            call construct_data() ! fills in array work()
            call use_data()       ! uses array work()
        enddo
c$omp end do 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 do
                      do i = 1, n
                          lx = lx + a(i)
                      enddo
              c$omp end do nowait
                      gx( omp_get_thread_num() ) = lx
              c$omp end parallel

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

                      print *, x
                      end
```

As shown below, this example can be 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 do
        do i = 1, n
            lx = lx + a(i)
        enddo
c$omp end do nowait
c$omp critical
        x = x + lx
c$omp end critical
c$omp end parallel

        print *, x
        end
```

This example can also be 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 do reduction(+:x)
        do i = 1, n
            x = x + a(i)
        enddo
c$omp end do nowait
c$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 can 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 `threadprivate` directive can be used instead.

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

c$omp parallel do
c$omp&   shared(a)
c$omp&   private(i)
        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
```

threadprivate: Copying in Initialization Values

Similar to “firstprivate: Copying in Initialization Values” on page 114 except using `threadprivate` common blocks. For `threadprivate`, `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 threadprivate (/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 114 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 micro-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 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
single	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.
- `barrier`s and `single`s have roughly the same overhead.
- After 2 processors, all the costs follow a nearly linear pattern as you add processors.

Index