KAP/Pro™ Toolset Reference Manual

Version 4.0

KAP/Pro™ Reference Manual
Version 4.0

Revised May 23, 2001

KAI Software, A Division of Intel Americas, Inc.
1906 Fox Drive
Champaign, IL 61820-7345
USA
Phone: (217) 356-2288
FAX: (217) 356-5199
Email: kappro-support@kai.com
URL: http://www.kai.com/parallel/kappro

---

*. Third party brand and product names are trademarks or registered trademarks of their respective companies.

Printed in the United States of America.

Table of Contents

CHAPTER 1 Introduction

About KAP/Pro

The KAP/Pro Toolset is a system of tools and application accelerators for developers of 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* Application Programming Interface (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 Guide component of the toolset compiles parallel programs annotated with OpenMP directives and can provide detailed performance statistics which are useful in performance tuning parallel programs. The input to Guide is either a Fortran, C, or C++ program annotated with OpenMP directives, which take the form of comments in Fortran programs and pragmas in C/C++ programs. The output of Guide is either a Fortran or C program, with the parallelism implemented using

threads and the Guide support libraries. This output is then compiled using your existing Fortran or C compiler. Parallel performance data is displayed using the GuideView GUI, which can be used to find regions where additional tuning could yield better performance.

The Assure component of the toolset validates the correctness of parallel programs annotated with OpenMP directives and identifies programming errors that occurred when parallelizing a sequential application. The inputs to Assure are an OpenMP parallel program that is assumed to run correctly in sequential mode and a data set for that program. Assure uses the semantics of a program's sequential execution to find differences that could occur in that program's parallel execution. For each data set that is used when an Assure-processed program is run, errors are identified when the parallel program is inconsistent with the corresponding sequential program. Assure displays its results using AssureView, a graphical user interface (GUI). AssureView pinpoints any errors that Assure finds down to the exact location in your source code.

The KAP/Pro Toolset also includes utilities to translate directives from older parallel processing directives to the new OpenMP directives.

Requirements

KAP/Pro requires a Fortran compiler and/or a C compiler, depending on the programming language(s) of the original source. Users running Unix or Linux will need the native C and/or Fortran compiler; C++ support is supplied by KCC. Users running Microsoft Windows have more options: **either** the Intel C++ and/or Fortran compiler **or** Microsoft Visual C++ and/or Compaq Visual Fortran. Both 32-bit and 64-bit multithreaded executables can be built on Windows systems.

GuideView, AssureView, and PerView require 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 Appendix E. Links to these packages are available on the KAI web site at *http://www.kai.com/parallel/kappro/helpers/*.

Newer versions of operating systems and compilers are constantly being announced; please see our web site at *http://www.kai.com/parallel/kappro/platforms/* for information on which versions are currently supported.

Installing KAP/Pro

To install the KAP/Pro Toolset on a machine running Windows simply run **guide*.exe** (the exact file name depends on your hardware platform and operating system) by double-clicking on its icon and then answering a few questions. Unix or Linux users can run this program by typing its name at a command prompt and pressing `<enter>`.

The KAP/Pro Toolset is licensed software and requires a license file from KAI in addition to the installer package(s). The installer will install any necessary license manager components, and will prompt you for the location of your license file. If you intend to use the AssureView, GuideView, or PerView GUI's, you will need to supply the location of your Java™ interpreter to the installer. The installation process for the Assure package is similar; use the file **assure*.exe**. Please contact us at **kappro-support@kai.com** for assistance if you have difficulty with the installation.

Using this Reference Manual

Reference Manual Contents

Chapter 2, "Parallel Processing and OpenMP," beginning on page 7, contains an overview of the OpenMP parallel processing model, and examples illustrating how to insert OpenMP directives.

Chapter 3, "Using Guide," beginning on page 17, contains an overview of Guide's functionality and examples illustrating how to build multithreaded programs using Guide.

Chapter 4, "Libraries and External Routines," beginning on page 21, explains the differences between Guide's several run–time libraries.

Chapter 5, "Using Assure," beginning on page 31, contains an overview of Assure's functionality and examples illustrating how to correct common parallel programming errors.

Chapter 6, “The KAP/Pro Drivers,” beginning on page 57, describes the Assure and Guide drivers, as well as descriptions of all Assure and Guide command line options. These options allow you to alter Assure or Guide’s default behaviors.

Chapter 7, “GuideView,” beginning on page 81, describes the GuideView graphical performance viewer.

Chapter 8, “AssureView,” beginning on page 91, describes how to use AssureView, which graphically displays Assure output.

Chapter 9, “PerView,” beginning on page 101, describes the PerView application manager and monitor.

Appendix A, “OpenMP Directives,” beginning on page 111, contains definitions for all OpenMP directives. OpenMP directives specify the parallelism within your code. This chapter also defines the Guide environment variables.

Appendix B, “C/C++ Examples,” beginning on page 155, contains code examples demonstrating the usage of OpenMP pragmas in C/C++.

Appendix C, “Fortran Examples,” beginning on page 175, contains Fortran code examples demonstrating the usage of OpenMP directives in Fortran.

Appendix D, “Additional KAP/Pro Options,” beginning on page 199, contains Fortran code examples demonstrating the usage of OpenMP directives in Fortran.

Appendix E, “Fortran Directive Translation,” beginning on page 217, describes the included utilities that translate older directives to 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 within text are printed in **bold** typeface.

With KAP/Pro’s *command line options* and *directives*, you can control a program’s parallelization by providing information to either Guide or Assure. 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, possibly with path included, **<character>** indi-

cates a single character, and **<string>** indicates a string of characters. For example, **-lines=<integer>** in this user's guide indicates that an integer must be provided in order to change the **-lines** option from the default value to a new value (such as **-lines=0**). As another example, **-WGdefault=<string>** in this user's guide indicates that a string must be provided in order to change the **-WGdefault** option from the default value to a new value (such as **-WGdefault=private**).

Optional items are denoted with square brackets:

-[no]dlines

In the above example, 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.
```

The KAP/Pro Toolset is available on a variety of platforms. Supported Windows platforms include Windows NT and Windows 2000; these will all be referred to in this manual as Windows. In general, anything specified for Unix users in this manual is also applicable to Linux users. Any counterexamples will be specifically indicated. For brevity, throughout this manual we use **Assuref** to represent any of the various Assure Fortran drivers (**Assuref77**, **Assuref90**, **Assureifl**, etc...) and **Assurec** to represent the corresponding C/C++ drivers. Similarly **Guidef** and **Guidec** represent the Guide drivers. When more generality is required, **Assure** is used to represent any Assure driver and **Guide** is used to represent any Guide driver.

KAP/Pro On-line

Visit the KAP/Pro Home Page at *http://www.kai.com/parallel/kappro* for the latest information on the KAP/Pro Toolset.

Technical Support

KAI strives to produce high-quality software. However, if any component of KAP/Pro 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 (see “Displaying all Command Lines” on page 63) to KAI Software, a division of Intel Americas, Inc. at **kappro-support@kai.com**.

Comments

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

CHAPTER 2

Parallel Processing and OpenMP

Parallel Processing Model

This chapter defines general parallel processing terms and explains how different OpenMP constructs affect parallel code. Further OpenMP details are available in the appendix called "OpenMP Directives," beginning on page 111. For exact semantics, please consult the OpenMP C/C++ or Fortran API standard document available at *http://www.openmp.org*, or contact KAI Software, a division of Intel Americas, Inc. at *http://www.kai.com/parallel/kappro* or **kappro-support@kai.com**.

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.

In C/C++, these OpenMP directives come in the form of pragmas and have the following form:

```
C/C++ syntax:
#pragma omp <directive>
      <structured block of code>
```

Here a structured block can be a single statement or several statements delineated by a "{" "}" pair. See the OpenMP C/C++ documentation for other rules on structured blocks.

In Fortran, these OpenMP directives have one of the following forms:

```
Fortran syntax:
!$omp <directive>
      <structured block of code>
!$omp end <directive>
```

or

```
!$omp <directive>
      <statement>
```

When the master thread enters a parallel region, a team 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 OpenMP `sections`, `single`, C/C++ `for`, and Fortran `do` constructs are defined as worksharing constructs because they distribute the enclosed work among the threads 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 C/C++ `for` and Fortran `do` directives specify parallel execution of loop iterations. The `sections` directive specifies parallel execution for arbitrary blocks of sequential code. Each `section` will be assigned to a unique thread within the team. The `single` directive defines a section of code where exactly one thread is allowed to execute the code; threads not chosen to execute this section ignore the code.

The OpenMP synchronization directives are `critical`, `ordered`, `master`, `atomic`, `flush`, and `barrier`. Within a parallel region or a worksharing construct only one thread at a time is allowed to execute the code within a `critical` section. The `ordered` directive is used in conjunction with a C/C++ `for`, Fortran `do`, or `sections` construct to impose a serial order on the execution of a section of code. The `master` directive is used to force execution by the master thread. A `barrier` directive forces 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. A `barrier` cannot be used within worksharing or other 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 finished, the team exits the worksharing construct and continues executing the code that follows.

At the end of the parallel region, threads wait until all team members have arrived. The team is logically disbanded (but may be reused in the next parallel region), and the master thread continues serial execution until it encounters the next parallel region.

A sample program using some of the more common OpenMP directives is shown in Figure 2-1 on page 11 Figure 2-2 on page 12. These examples also indicate the difference between serial regions and parallel regions.

Increasing Efficiency

Scheduling options can be selected for a C/C++ `for` or a Fortran `do` worksharing construct to increase efficiency. Scheduling options specify the way threads are assigned iterations of a loop. The scheduling options include `static`, `dynamic`, `guided`, and `runtime`.

Also attached to `for` and `do` worksharing constructs, a `nowait` option can be used to increase efficiency. The `nowait` option allows threads that finish their work to continue executing code, regardless of whether the other threads in the team have finished. These threads do not wait at the end of the worksharing or workqueuing construct, but proceed immediately to the code following.

Enabling certain Guide compile options can also help increase efficiency. These options are covered in Chapter 6.

Figure 2-1 Pseudo C/C++ Code of the Parallel Processing Model

```
main() {                  // Begin serial execution
 ...                      // Only the master thread executes
#pragma omp parallel      // Begin a Parallel Construct,form
 {                        // a team. This is Replicated Code
  ...                     // (each team member executes
  ...                     // the same code)
                          //
  #pragma omp sections    // Begin a Worksharing Construct
   {                      //
    #pragma omp section   // One unit of work
      {...}               //
    #pragma omp section   // Another unit of work
      {...}               //
   }                      // Wait until both units of work
                          // complete
  ...                     // More Replicated Code
                          //
  #pragma omp for nowait  // Begin a Worksharing Construct;
    for(...) {            // each iteration is unit of work
                          //
      ...                 // Work is distributed among the
                          // team members
                          //
    }                     // End of Worksharing Construct;
                          // nowait was specified, so
                          // threads proceed
                          //
  #pragma omp critical    // Begin a Critical Section
   {                      //
     ...                  // Replicated Code, but only one
                          // thread can execute it at a
   }                      // given time
  ...                     // More Replicated Code
                          //
  #pragma omp barrier     // Wait for all team members to
                          // arrive
  ...                     // More Replicated Code
                          //
}                         // End of Parallel Construct;
                          // disband team and continue
                          // serial execution
                          //
...                       // Possibly more Parallel
                            // Constructs
                            //
}                           // End serial execution
```

Figure 2-2 Pseudo Fortran 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
       ...                !
    !$omp end sections    ! Wait until both units of work
                          ! complete
                          !
    ...                   ! More Replicated Code
                          !
    !$omp do              ! Begin a Worksharing Construct,
      do                  ! each iteration is a unit of work
                          !
    ...                   ! Work is distributed among the
team
      end do              !
    !$omp end do nowait   ! End of Worksharing Construct,
                          ! nowait is specified
                          !
    ...                   ! More Replicated Code
                          !
    !$omp barrier         ! Wait for all team members to
                          ! arrive
                          !
    ...                   ! More Replicated Code
                          !
!$omp end parallel        ! End of Parallel Construct,
                          ! disband team and 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 responsibility to synchronize access to these variables. All variables in the `private` clause are private to each team member. For the entire parallel region, assuming *t* team members, we have *t*+1 copies of all the variables in the `private` clause: one global copy that is active outside parallel regions and a private copy for each team member. Initialization of `private` variables at the start of a parallel region is the programmer's responsibility, unless the `firstprivate` clause is specified. In 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 C/C++ `for` or Fortran `do` directive enables updating the global copy from the team member that executed the serially last iteration of the loop.

In addition to `shared` and `private` variables, individual variables and entire `COMMON` blocks in Fortran can be privatized using the `threadprivate` directive. For compatibility with Cray `taskcommon` directives, `threadprivate common` blocks always create *t* copies, one for each of the *t* team members. The master thread uses the global copy as its `private` copy for the duration of each parallel region.

In addition to the `shared` and `private` clauses, file-scope and namespace-scope variables in C++ can be made private to a thread using the `threadprivate` directive. Threadprivate variables always have a copy created for each team member. The master thread uses the global copy as its private copy for the duration of each parallel region.

Local static variables in C can also be made threadprivate. This is a KAP/Pro Toolset extension to OpenMP.

Orphaned Directives

OpenMP contains a feature called orphaning which dramatically increases the expressiveness of parallel directives. By orphaning we mean that directives related to a parallel region are not required to occur lexically within a single program unit. Directives such as `critical`, `barrier`, `sections`, `single`, `master`, and C/C++ `for` or Fortran `do`, 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 C/C++ `for` or Fortran `do` directives located within called subroutines. Consider the following code segment:

C/C++ syntax:

```
void main() {
    #pragma omp parallel
      {
        phase1();
        phase2();
      }
}

void phase1(void) {
    ...
    #pragma omp for private(i) shared(n)
    for(i=0; i < n; i++) {
        some_work(i);
    }
}

void phase2(void) {
    ...
    #pragma omp for private(j) shared(n)
    for(j=0; j < n; j++) {
        more_work(j);
    }
}
```

Fortran syntax:

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

        subroutine phase1
!$omp do private(i) shared(n)
           do i = 1, n
              call some_work(i)
           end do
!$omp end do
        end

        subroutine phase2
!$omp do private(j) shared(n)
           do j = 1, n
              call more_work(j)
           end do
!$omp end do
        end
```

Notice in this example, the directives specifying the parallelism are divided across three separate program units.

A Few Rules about Orphaned Directives

1. An orphaned worksharing construct (`section, single`, C/C++ `for` or Fortran `do`) or C/C++ workqueuing construct (`taskq`) that is dynamically executed outside of a parallel region will be executed by a team consisting of one thread; *i.e.* serially. In the following example the first call to `phase0` is executed serially, and the second call is partitioned among the threads created for the parallel region.

2. Any collective operation (worksharing construct, workqueuing construct, or barrier) executed inside of a worksharing construct is illegal.

3. It is illegal to execute a collective operation (worksharing, workqueuing, or barrier) from within a synchronization region (`critical/ordered`).

4. The opening and closing directives of a Fortran 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 standard documentation for complete details.

CHAPTER 3

Using Guide

Introduction

Guide accepts a C/C++ or Fortran program containing OpenMP directives as input and produces a multithreaded version of the code which is then passed to an underlying C or Fortran compiler. To run multithreaded, the user simply assigns the number of threads desired to the OpenMP environment variable OMP_NUM_THREADS before execution.

Guide also provides functionality for performance analysis of parallel programs; by liking with a statistics library one can obtain detailed information about which portions of the code require the largest amount of time to execute and where parallel performance bottlenecks are located. This information can be displayed graphically by using the GuideView GUI, a powerful tool which shows performance information that is not available with traditional profilers.

The components of Guide include:

- The Guide compile driver, named as listed in the following table:

	Windows	**Unix**	**Linux**
C	guideicl, guideecl, *or* guidec[64]	guidec	guideicc, guideecc, *or* guidec
C++	*not available*	guidec++	guideicc, guideecc, *or* guidec++
FORTRAN 77	guideifl, guideefl, *or* guidef[64]	guidef77	guideifc *or* guideefc
Fortran 90	guideifl, guideefl, *or* guidef[64]	guidef90	guideifc *or* guideefc

- GuideView, a tool for viewing parallel performance statistics

On Windows, use **guideicl** for building 32-bit applications using the Intel C++ compiler, **guideecl** for building 64-bit applications using the Intel C++ compiler, and **guidec** for building C applications using the Microsoft Visual Studio C++ compiler. C++ is not currently supported on Windows.

Similarly, use **guideifl** on Windows for building 32-bit applications using the Intel Fortran compiler, **guideefl** for building 64-bit applications using the Intel Fortran compiler, and **guidef** for building 32-bit or 64-bit applications using the appropriate Compaq Visual Fortran compiler.

Special drivers called **guidec64** and **guidef64** are supplied for Windows users who install the KAP/Pro Toolset on a 32-bit machine and wish to build 64-bit applications with the appropriate Microsoft Visual Studio C++ compiler and Compaq Visual Fortran compiler, respectively. The **guidec** and **guidef** drivers produce executables compatible with the machine architecture on which KAP/Pro is installed.

On Linux, use **guideicc** for building 32-bit applications using the Intel C++ compiler, **guideecc** for building 64-bit applications using the Intel C++ compiler, and **guidec** or **guidec++** for building C or C++ applications, respectively, using the gcc or g++ compiler. Use **guideifc** for building 32-bit applications

using the Intel Fortran compiler, **guideefc** for building 64-bit applications using the Intel Fortran compiler.

Using Guide to Develop Parallel Programs

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

Prepare

- Before inserting any OpenMP parallel directives, verify that your code is safe for parallel execution by placing local variables on the stack. This is the default behavior of many compilers. Normally, a `-automatic` flag or similar compiler option achieves this. If your application is unable to execute correctly with stack allocation of local data, this generally indicates that your code has subroutines that need some variables saved across invocations. These variables should be made `STATIC` when using C/C++, or be placed in a `SAVE` statement in Fortran. By default these variables become shared across threads, and you may need to add synchronization code to ensure proper access by threads.

Analyze

- Profile the program to find out where it spends most of its time. This is the part of the program that would benefit most from parallelization efforts. This stage can be accomplished using a standard profiler, such as `prof`, or by using the profiling features of GuideView.
- In this part of the program there are usually nested loops. Locate a loop that has very few cross-iteration dependences.

Restructure

- If a chosen loop is able to execute iterations in parallel, introduce a `parallel for` or `parallel do` directive around this loop.
- Attempt to remove any 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.
- List the variables that are present in the loop within appropriate `shared()`, `private()`, `lastprivate()`, or `firstprivate()` clauses.
- List the `for` (C/C++) or `do` (Fortran) index of the parallel loop as `private()`. This step is optional in Fortran.
- In C/C++: File-scope variables must not be placed on the `private()` list if their file-scope visibility is to be preserved. Instead, use the `threadprivate` directive to make a variable private to a thread while preserving its file-scope visibility.
- In Fortran: `COMMON` block elements must not be placed on the `private()` list if their global scope is to be preserved. The `threadprivate` directive can be used to privatize to a thread the `COMMON` block containing those variables with global scope.
- Any I/O in the `parallel` region should be synchronized.
- Identify more parallel loops and restructure them.
- If possible, merge adjacent C/C++ `parallel for` or Fortran `parallel do` sections into a single parallel region with multiple `for` or `do` directives 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 139.

For parallel Fortran programs containing older parallel directives, a tool is included with Guide to help automate the job of translating them to OpenMP parallel directives. See “Fortran Directive Translation” on page 217.

CHAPTER 4

Libraries and External Routines

Selecting a Library

The standard edition of Guide supplies three libraries: a development library, a management and monitoring library, and a production library. These libraries allow the user to run an application under different execution modes. 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 slightly degrades performance and 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. This library degrades application performance slightly also.

To switch between the *guide*, *guide_stats*, and *guide_perview* libraries, only relinking of the object files is necessary; recompilation of the source code is not needed.

Guide allows the user the ability to run an application under different execution modes that can be specified at run time. All three libraries support the serial, turn-

around, and throughput modes described below. These modes are selected by using the KMP_LIBRARY environment variable at run time; see “KMP_IGNORE_MPPBEG <integer>” on page 75.

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 active all of the processors involved in the parallel computation in order 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 instead if this occurs.

Throughput

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

After a certain period of wall-clock time has elapsed, during which the worker threads do not find parallel work, the threads will stop seeking work. Instead, they will wait for the master thread to notify them of the availability of new parallel work. This time is set by the KMP_BLOCKTIME environment variable and the kmp_set_blocktime() library call, described in the sections “OpenMP Environment Variables” on page 148 and “External Routines” on page 25,

respectively. The default time before "blocking" is one second. It is recommended that KMP_BLOCKTIME be set to a small value if your application contains "hand-threaded" regions, to avoid contention for compute resources.

The guide_stats *Library*

Guide links in the *guide* library by default. To use the *guide_stats* library, include the **-WGstats** option to the Guide driver within the command line. For example, the following command line can be used on Windows to compile a Fortran source file with the *guide_stats* library:

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

The *guide_stats* library is designed to provide 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. The resulting statistics are most easily viewed and analyzed using GuideView, discussed in Chapter 7, "GuideView," beginning on page 81.

When a program is compiled with Guide, 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.gvs 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 have been compiled with Guide.

The *guide_stats* library gathers performance statistics for each parallel region executed during a run. The *guide_stats* library also gathers statistics for each of these serial portions of code around and between the OpenMP parallel regions. The parallel regions are designated R1, R2, etc., following the order in which they are first executed. Similarly, the sequential code blocks are designated S1, S2, etc., following the order in which they are first executed.

If a parallel region contains one or more barriers, including implicit barriers after worksharing constructs, the parallel region statistics are further subdivided. For example, if the parallel region R3 contains two barriers with the second barrier

being at the end of the parallel region, we have separate performance statistics for regions `R3` (extending from the beginning of the parallel region to the first barrier) and `R3B1` (extending from the first barrier to the end of the parallel region).

By default, each region is given a name, based on the name of the file that contains the beginning of the region. However, it is possible to manually specify a name for each parallel region by means of an external routine, `kmp_set_parallel_name()`. This routine takes a character string as an argument, and should be called before the start of the parallel region to be named. All parallel regions following a call to this routine get the specified name until another call is made to `kmp_set_parallel_name()`. Default names can be restored by supplying an empty string as the argument.

The *guide_stats* library may minimally degrade application performance compared to the *guide* library. The amount of application slowdown is proportional to the frequency with which OpenMP directives are encountered.

The guide_perview *Library*

To link with the *guide_perview* library, use the **-WGperview** command line option to the Guide driver. The *guide_perview* library is part of the interactive parallel performance monitoring and management tool called *PerView.* Using *PerView*, one 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 have been compiled with Guide.

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 enables additional runtime checks that may aid in program debugging. This library may minimally degrade application performance compared to the *guide* library by an amount proportional to the frequency with which the OpenMP directives are encountered.

See “PerView,” beginning on page 101 for more information about the use of the *guide_perview* library and the PerView tool.

External Routines

This section details library routines that can be used for low-level debugging to verify that the library code and application are functioning as intended. These routines are not part of the OpenMP standard and their use 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, "C/C++ Examples," beginning on page 155, and Appendix C, "Fortran Examples," beginning on page 175, contain examples of coding styles that avoid the use of these routines. A run-time procedure call takes precedence over an environment variable setting.

To use these functions in a C/C++ program, include the OpenMP header file

```
#include <omp.h>
```

in your source. In the descriptions of the routines that follow, the C/C++ prototype is given first, followed by the Fortran calling convention.

In some cases, the same effect can be achieved by either setting an environment variable or by using one of these routines. In these cases, the external routine overrides any environment variable settings.

int kmp_get_blocktime(void), integer function 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 set via the `kmp_set_blocktime()` routine or the `KMP_BLOCKTIME` environment variable. See the description of the `KMP_BLOCKTIME` environment variable on page 75 for more information.

int kmp_get_library(void), integer function kmp_get_library()

This routine returns an integer value that designates the version of the Guide runtime library being used. This value can be used as the parameter to subsequent calls to `kmp_set_library()`. The library setting can also be set via the `kmp_set_library_<mode>()` set of routines or the `KMP_LIBRARY` environment variable.

int kmp_get_stacksize(void), integer function kmp_get_stacksize()

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 76 for more information.

void kmp_set_blocktime(int), subroutine 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 75 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.

void kmp_set_library(int), subroutine kmp_set_library(<integer>)

This routine selects the Guide run time library. The parameter value corresponds to the version number 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_<mode>` routines and then call the `kmp_get_library()` routine to obtain the parameter values. The library setting can also be set via the `KMP_LIBRARY` environment variable.

void kmp_set_library_serial(void), subroutine

kmp_set_library_serial()

This routine selects the Guide serial runtime library. The library setting can also be set via the `kmp_set_library()` call or the `KMP_LIBRARY` environment variable.

void kmp_set_library_throughput(void), subroutine kmp_set_library_throughput()

This routine selects the Guide throughput runtime library. The library setting can also be set via the `kmp_set_library()` call or the `KMP_LIBRARY` environment variable.

void kmp_set_library_turnaround(void), subroutine kmp_set_library_turnaround()

This routine selects the Guide turnaround runtime library. The library setting can also be set via the `kmp_set_library()` call or the `KMP_LIBRARY` environment variable.

void kmp_set_stacksize(int), subroutine 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 set via the `KMP_STACKSIZE` environment variable (see page 76 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.

void kmp_set_parallel_name(char *), subroutine kmp_set_parallel_name(<string>)

This routine associates the character string argument to subsequent parallel regions. The name remains in effect until the next call to the routine. To restore default naming of parallel regions, supply an empty string as the argument.

This routine should be called before the start of the parallel region to be named. The associated name will appear in the statistics file output by the *guide_stats* library and in the GuideView performance viewer.

void mppbeg(void), subroutine mppbeg()
void mppend(void), subroutine mppend()

These routines are not necessary if the main program unit and all exit points are compiled using Guide. 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 lead to program termination.

Calling these routines from another programming language requires knowledge of the cross-language calling standards on your platform. Typically an underscore is appended to names that are declared in Fortran subroutines when called from C code. 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_();
    Fortran_work();
    mppend_();
    exit(0);
}
```

A main program written in Fortran might look like:

```
program main
call mppbeg()
call C_work
call mppend()
end
```

In other programming languages, you may need to append an underscore to the routine names to successfully link: e.g., `mppbeg_()` and `mppend_()`.

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

Signal Handling (Unix only)

In order for interrupts and runtime errors to be handled correctly during parallel execution on Unix, the Guide libraries normally install their own handlers for interrupt signals (e.g. `SIGHUP`, `SIGINT`, `SIGQUIT`, and `SIGTERM`) and for runtime error signals (e.g. `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 all 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.

CHAPTER 5

Using Assure

Introduction

Assure is designed to validate the correctness of an OpenMP parallel program and to identify programming errors that occur when parallelizing a serial application. Assure uses the semantics of a program's sequential execution to find errors that could occur in that program's parallel execution. For each data set that is used when executing an Assure-processed program, errors are identified when the parallel program is inconsistent with the corresponding serial program.

Programs validated by Assure must be assumed to be sequentially correct. Serial programs should be compiled and tested with all local variables in each routine allocated on the stack (i.e., as *automatic* variables), and no uninitialized accesses, out-of-bounds memory references, etc. should be present in the sequential code.

How to Verify an Application

The components of Assure include:

- The Assure compile driver, named as listed in the following table.

	Windows NT	Unix	Linux
C	assureicl, assureecl, *or* assurec[64]	assurec	assureicc, assureecc *or* assurec
C++	*not available*	assurec++	assureicc, assureecc, *or* assurec++
FORTRAN 77	assureifl, assureefl, *or* assuref[64]	assuref77	assureifc *or* assureefc
Fortran 90	assureifl, assureefl, *or* assuref[64]	assuref90	assureifc *or* assureefc

- AssureView, a tool for viewing the results of Assure

On Windows, use **assureicl** for debugging 32-bit applications using the Intel C++ compiler, **assureecl** for debugging 64-bit applications using the Intel C++ compiler, and **assurec** for debugging C applications using the Microsoft Visual Studio C++ compiler. C++ is not currently supported on Windows.

Similarly, use **assureifl** on Windows for debugging 32-bit applications using the Intel Fortran compiler, **assureefl** for debugging 64-bit applications using the Intel Fortran compiler, and **assuref** for debugging 32-bit or 64-bit applications using the appropriate Compaq Visual Fortran compiler.

Special drivers called **assurec64** and **asssuref64** are supplied for Windows users who install the KAP/Pro Toolset on a 32-bit machine and wish to debug 64-bit applications with the appropriate Microsoft Visual Studio C++ compiler and Compaq Visual Fortran compiler, respectively. The **assurec** and **assuref** drivers produce executables compatible with the machine architecture on which KAP/Pro is installed.

On Linux, use **assureicc** for debugging 32-bit applications using the Intel C++ compiler, **assureecc** for debugging 64-bit applications using the Intel C++ compiler, and **assurec** or **assurec++** for debugging C or C++ applications, respectively, using the gcc or g++ compiler. Use **assureifc** for debugging 32-bit applications using the Intel Fortran compiler, **assureefc** for debugging 64-bit applications using the Intel Fortran compiler.

The steps involved in using Assure to verify an application program are depicted in Figure 5-1, “Assure Process,” on page 34:

1. Compile the program using Assure.

   Assure takes as input a correct, sequential program that has been annotated with OpenMP parallel directives. This step produces a *project file*, a file with a `.prj` suffix, used by AssureView to document the structure of the code. An Assure link step produces a program that, when executed, generates data for AssureView.

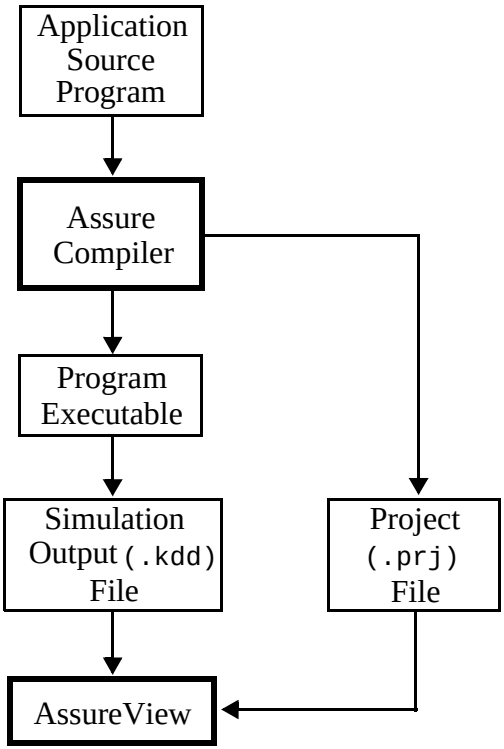
2. Run the compiled program.

   This step produces simulation output in a file with a `.kdd` suffix, which contains encoded results of the execution. Execution with Assure can require significantly more time and memory. Whenever possible, use input datasets of minimal size that exercise as much of the code as possible.

3. View the results using AssureView.

   AssureView takes as its inputs the project file produced in step 1 and the simulation output file produced in step 2 and displays the results via a GUI. Results may also be displayed as text.

Figure 5-1 Assure Process



When the application program, here called `pgm`, is contained in a single source file, the following sequence of commands can be used on Unix to run Assure and AssureView:

```
assurec -WGpname=prog -o pgm pgm.c
pgm
assureview prog.prj
```

The `-WGpname` flag is used to select the name of the project file (in this example, `prog.prj`). On Windows the following sequence of commands can be used:

```
assurec -WGpname=prog -exe:pgm.exe pgm.c
pgm.exe
assureview prog.prj
```

When the application program consists of multiple source files, the following sequence of commands could be used on Unix for compiling and linking:

```
assurec -WGpname=prog -c f1.c
assurec -WGpname=prog -c f2.c
assurec -WGpname=prog -o pgm f1.o f2.o
```

The following sequence of commands could be used on Windows:

```
assurec -WGpname=prog -c f1.c
assurec -WGpname=prog -c f2.c
assurec -WGpname=prog -exe:pgm.exe f1.obj f2.obj
```

Running the executable and viewing the Assure results is the same as in the single source file example. When using makefiles, it may be sufficient to change your compile and link commands in each makefile to

```
assurec -WGpname=prog
```

For projects with multiple build directories, the project file should be specified as an absolute path to ensure that the same project file is used in each step of the build process. For example on Unix:

```
assurec -WGpname=/projects/prog/prog.prj f.c
```

or on Windows:

```
assurec -WGpname=c:\projects\prog\prog.prj f.c
```

The results of Assure can be viewed by using the full name of the project file on the command line:

```
assureview /projects/prog/prog.prj
```

In addition, the `make clean` rule in makefiles should be modified to remove any files with `.prj` or `.kdd` suffixes.

During compilation, in order to serialize access to project files (*e.g.*, when running makefiles in parallel) an explicit lockfile is associated with each file. If a project name is not specified, the default name is `assure.prj` and the corresponding lockfile would be named `.assure.prj.lck`. Each lockfile is placed in the same directory as its corresponding project file. Periodic messages may appear if an Assure or AssureView step is waiting for the release of a lockfile. These messages can be ignored in a properly executed parallel make. Parallel makes may fail in NFS mounted directories due to problems with file locking.

An Example

The following is a correct sequential program fragment to multiply matrices `a` and `b`. This program has been parallelized with OpenMP directives and can be validated by Assure.

C/C++ syntax:

```
#pragma omp parallel for \
                shared(a,b,c,s,n,m) \
                private(i,j,k)
for (j = 0; j < n; j++) {
    for (i = 0; i < m; i++) {
L10:    s = 0.0;
        for (k = 0; k < m; k++) {
L20:        s += a[k][i] * b[j][k]
        }
L30:    c[j][i] = s;
    }
}
```

Fortran syntax:

```
!$omp parallel do
!$omp&   shared(a,b,c,n,m,s)
!$omp&   private(i,j,k)
      do j = 1, n
          do i = 1, m
10              s = 0.0
                do k = 1, m
20                  s = s + a(i,k) * b(k,j)
                enddo
30              c(i,j) = s
          enddo
      enddo
!$omp end parallel do
```

When Assure is run on a complete program containing this program fragment, AssureView will report storage conflicts errors (see “Storage Conflicts” on page 37) on the statements labeled `10`, `20`, and `30`. The programmer forgot to make `s` a private variable, which would cause the program to generate incorrect results when run in parallel. The correct parallel program fragment is given below.

C/C++ syntax:

```
#pragma omp parallel for \
                shared(a,b,c,n,m) \
                private(i,j,k,s)
for (j = 0; j < n; j++) {
    for (i = 1; i < m; i++) {
L10:    s = 0.0;
        for (k = 0; k < m; k++) {
L20:        s += a[k][i] * b[j][k];
        }
L30:    c[j][i] = s;
    }
}
```

Fortran syntax:

```
!$omp parallel do
!$omp&   shared(a,b,c,n,m)
!$omp&   private(i,j,k,s)
      do j = 1, n
          do i = 1, m
10             s = 0.0
               do k = 1, m
20                 s = s + a(i,k) * b(k,j)
               enddo
30             c(i,j) = s
          enddo
      enddo
!$omp end parallel do
```

When Assure is run on a complete program containing this program fragment, no storage conflict errors are reported, so this portion of the program will run correctly in parallel.

Storage Conflicts

In order to produce correct results, a correctly-parallelized program must preserve the constraints on the order of variable references imposed by the original sequential program execution (these constraints are also known as *data dependence* constraints). *Storage conflicts* are violations of these variable-reference-order constraints which cause a parallel program to yield incorrect or indeterminate results when compared to the original serial program.

Three different types of storage conflicts occur in parallel programs, each of which is identified by Assure:

1. Write → Read storage conflicts

   These conflicts denote violations of *flow-dependence* (or *true dependence*) constraints. Such a constraint is introduced in a serial program when one statement updates a variable that may be read by a subsequent statement.

   For example, consider the following sequence of statements:

   ```
   s1: a = b + c
   s2: d = a + e
   ```

   A flow-dependence constraint on the variable `a` exists between `s1` and `s2` since `s1` must execute before `s2` in order for `s2` to use the correct value for `a` (i.e., the value produced by `s1`).

2. Read → Write storage conflicts

   These conflicts denote violations of *anti-dependence* constraints. Such a constraint is introduced in a serial program when one statement reads a variable that may be written by a subsequent statement.

   For example, consider the following sequence of statements:

   ```
   s1: a = b + c
   s2: b = d + e
   ```

   An anti-dependence constraint on the variable `b` exists between `s1` and `s2` since `s1` must execute before `s2` in order for `s1` to use the correct value for `b` (i.e., the value produced by some statement before `s1`, not the value produced by `s2`).

3. Write → Write storage conflicts

   These conflicts denote violations of *output-dependence* constraints. Such a constraint is introduced in a serial program when one statement updates a variable that may be written by a subsequent statement.

   For example, consider the following sequence of statements:

   ```
   s1: a = b + c
   s2: a = d + e
   ```

   An output-dependence constraint on the variable `a` exists between `s1` and `s2` since `s1` must execute before `s2` in order for subsequent statements to use the correct value for `a` (i.e., the value produced by `s2`, not the value produced by `s1`).

Assure reports storage conflicts by specifying the variable(s) and statement(s) involved in a sequential-program constraint that may be violated in the corresponding parallel program. A *source* and a *sink* variable reference are specified; the constraint being violated is that the *source* reference must always occur before the *sink* reference.

Storage conflicts occur when two variable references can be executed by two different threads in an indeterminate order. Common causes of storage conflicts include:

1. A variable was `shared` between threads when it should have been `private` to each thread.
2. A variable was `shared` but its accesses were not synchronized (e.g., by enclosing references to the variable in `critical` sections).
3. The algorithm used by the program cannot be directly executed in parallel by the simple change of a variable's classification to `shared` or `private`. Usually, in this case, the algorithm used in the computation must be changed. In the next section we give examples to illustrate some of the more advanced transformations needed to correctly execute a serial program in parallel.

Correcting Errors

The following examples are designed to illustrate common parallel programming errors, how Assure treats each one, and how they may be corrected by using OpenMP directives to restructure the parallel code.

Example: Parallelizing Reduction Loops

Consider the following sequential program, which sums the numbers one through ten and prints the answer (55).

C/C++ syntax:

```
#include <stdio.h>
main ()
{
    int i, k = 0;
    for (i = 1; i <= 10; i++) k += i;
    printf("%d\n", k);
}
```

Fortran syntax:

```
      program sum10a
      k = 0
      do i = 1, 10
         k = k + i
      end do
      print *, k
      end
```

In this program, the variable k is reused (between loop iterations) to act as an accumulator. This reuse causes a storage conflict that must be resolved in order to execute the loop in parallel. Suppose that the sequential program is parallelized as follows. Assure identifies a Write → Write storage conflict in this program, on the variable k inside the for or do loop.

C/C++ syntax:

```
#include <stdio.h>
main ()
{
    int i, k = 0;
    #pragma omp parallel for shared(k) private(i)
    for (i = 1; i <= 10; i++) k += i;
    printf("%d\n", k);
}
```

Fortran syntax:

```
      program sum10b
      k = 0
!$omp parallel do shared(k) private(i)
      do i = 1, 10
          k = k + i
      end do
!$omp end parallel do
      print *, k
      end
```

Since multiple threads are executing the update of the variable k without proper synchronization, threads could easily overwrite the results of other threads, thus yielding the wrong final answer.

One common method of parallelizing reduction (accumulation) loops is to perform partial reductions on each processor and then perform the final reduction

into the output variable. This can also be written as a sequential algorithm as follows:

C/C++ syntax:

```
#include <stdio.h>
main ()
{
    int i, k = 0, kl = 0;
    for (i = 1; i <= 10; i++)
        kl += i;
    k += kl;
    printf("%d\n", k);
}
```

Fortran syntax:

```
      program sum10c
      k = 0
      kl = 0
      do i = 1, 10
         kl = kl + i
      end do
      k = k + kl
      print *, k
      end
```

This new sequential algorithm is potentially less efficient than the previous example; however, by introducing a new variable, we are allowed more freedom in parallelizing the code since we have removed constraints on its parallel execution. The new sequential code can be parallelized as follows.

C/C++ syntax:

```
#include <stdio.h>
main ()
{
    int i, k = 0, kl;
    #pragma omp parallel shared(k) private(i,kl)
    {
        kl = 0;
        #pragma omp for
        for (i = 1; i <= 10; i++)
            kl += i;
        k += kl;
    }
    printf("%d\n", k);
}
```

Fortran syntax:

```
      program sum10d
      k = 0
!$omp parallel shared(k) private(i,kl)
      kl = 0
!$omp do
      do i = 1, 10
          kl = kl + i
      enddo
!$omp end do
      k = k + kl
!$omp end parallel
      print *, k
      end
```

Unfortunately, Assure will identify storage conflicts in this parallel program as well. By introducing a new private variable, `kl`, we removed the original storage conflict in the parallel loop. However, there is still a storage conflict in the final reduction, `k += kl or k = k + kl`. This final reduction needs to be synchronized (serialized) to produce a correct parallel algorithm as follows:

C/C++ syntax:

```
#include <stdio.h>
main ()
{
    int i, k = 0, kl;
    #pragma omp parallel shared(k) private(i,kl)
    {
        kl = 0;
        #pragma omp for
        for (i = 1; i <= 10; i++)
            kl += i;
        #pragma omp critical
        k += kl;
    }
    printf("%d\n", k);
}
```

Fortran syntax:

```
      program sum10e
      k = 0
!$omp parallel shared(k) private(i,kl)
      kl = 0
!$omp do
      do i = 1, 10
          kl = kl + i
      enddo
!$omp end do
!$omp critical
      k = k + kl
!$omp end critical
!$omp end parallel
      print *, k
      end
```

Assure identifies no errors in this version of the parallel code.

Alternatively, this same code segment could be written using the OpenMP `reduction` directive as follows:

C/C++ syntax:

```
#include <stdio.h>
main ()
{
    int i, k = 0;
    #pragma omp parallel for reduction(+:k) \
      private(i)
        for (i = 1; i <= 10; i++)
            k += i;
    printf("%d\n", k);
}
```

Fortran syntax:

```
      program sum10f
      k = 0
!$omp parallel do reduction(+:k) private(i)
      do i = 1, 10
          k = k + i
      enddo
!$omp end parallel do
      print *, k
      end
```

Each parallel version of this summation algorithm has a corresponding sequential program that correctly computes the desired result. In each case, if the parallel program is run on one processor it will be equivalent to the corresponding sequential program, as if the OpenMP directives were ignored. This pairing of the sequential and parallel semantics of a program allows Assure to determine when the parallel program is incorrect when compared to the specification provided by the sequential program.

Example: Privatizing to Resolve Storage Conflicts

As shown in the previous example, the act of parallelization (converting a serial algorithm into a parallel algorithm) is often an incremental process. This process proceeds from the assumption that the computation to be performed is logically concurrent but that a particular implementation of the algorithm introduces dependences that can be removed through the use of OpenMP directives or code restructuring.

Two of the most common techniques for resolving storage conflicts are privatization (storage localization, replication) and synchronization (serialization). Consider the following example:

C/C++ syntax:

```
void dsq( float a[], float b[], float c[], int n )
{
    float x, y;
    int i;
    for (i = 0; i < n; i++) {
        x = a[i] - b[i];
        y = b[i] + a[i];
        c[i] = x * y;
    }
}
```

Fortran syntax:

```
      subroutine dsq( a, b, c, n )
      integer n
      real a(n), b(n), c(n), x, y
      do i = 1, n
        x = a(i) - b(i)
        y = b(i) + a(i)
        c(i) = x * y
      end do
      end subroutine
```

The above serial routine implements a logically concurrent algorithm: applying a function to each element of a vector. A first attempt at parallelization of this program might yield the following:

C/C++ syntax:

```
void dsq_a(float a[], float b[], float c[], int n)
{
    float x, y;
    int i;
    #pragma omp parallel for shared(a,b,c,n,x,y) \
                    private(i)
    for (i = 0; i < n; i++) {
        x = a[i] - b[i];
        y = b[i] + a[i];
        c[i] = x * y;
    }
}
```

Fortran syntax:

```
      subroutine dsq_a (a,b,c,n)
      integer n
      real a(n), b(n), c(n), x, y
!$omp parallel do shared(a,b,c,n,x,y) private(i)
      do i = 1, n
          x = a(i) - b(i)
          y = b(i) + a(i)
          c(i) = x * y
      enddo
!$omp end parallel do
      end
```

Assure reports Write → Write storage conflicts on the variables `x` and `y` in this parallel program. This means that, on two different iterations of the parallel loop, these variables were updated (potentially by different threads). These storage conflicts can be resolved through privatization or synchronization. To correctly synchronize this code, a `critical` section should be added surrounding the definitions and uses of the offending variables:

C/C++ syntax:

```
void dsq_b(float a[], float b[], float c[], int n)
{
    float x, y;
    int i;
    #pragma omp parallel for shared(a,b,c,n,x,y) \
                    private(i)
    for (i = 0; i < n; i++) {
        #pragma omp critical
        {
            x = a[i] - b[i];
            y = b[i] + a[i];
            c[i] = x * y;
        }
    }
}
```

Fortran syntax:

```
      subroutine dsq_b (a,b,c,n)
      integer n
      real a(n), b(n), c(n), x, y
!$omp parallel do shared(a,b,c,n,x,y) private(i)
      do i = 1, n
!$omp critical
          x = a(i) - b(i)
          y = b(i) + a(i)
          c(i) = x * y
!$omp end critical
      enddo
!$omp end parallel do
      end
```

Assure reports no storage conflicts in this parallel program. However, this is not an efficient method of resolving the previous conflicts. The addition of the `critical` section serializes the execution of the loop entirely, thereby prohibiting any performance improvement through parallelism. If the only way to resolve storage conflicts is through synchronization, then it is likely that the original algorithm was not inherently concurrent and that this algorithm should be run sequentially.

The storage conflicts on `x` and `y` could also be resolved through privatization.

C/C++ syntax:

```
void dsq_c(float a[], float b[], float c[], int n)
{
    float x, y;
    int i;
    #pragma omp parallel for shared(a,b,c,n) \
                    private(i,x,y)
    for (i = 0; i < n; i++) {
        x = a[i] - b[i];
        y = b[i] + a[i];
        c[i] = x * y;
    }
}
```

Fortran syntax:

```
      subroutine dsq_c (a,b,c,n)
      integer n
      real a(n), b(n), c(n), x, y
!$omp parallel do shared(a,b,c,n) private(i,x,y)
      do i = 1, n
          x = a(i) - b(i)
          y = b(i) + a(i)
          c(i) = x * y
      enddo
!$omp end parallel do
      end
```

In this parallel program, `x` and `y` have been declared private to each thread that executes the parallel region. Privatization removes the storage conflicts by giving each thread executing the parallel loop its own local copy of the variables `x` and `y`. This is the preferred way to resolve these types of storage conflicts.

Parallelism is inhibited by synchronization and is enabled by privatization. To enhance the performance of parallel programs, privatization should be utilized instead of synchronization whenever possible. Some runtime operations (e.g., I/O routines) may not be safe to execute in parallel; in these cases, synchronizing these operations allows the rest of a parallel region to be executed in parallel. If the percentage of time spent in a synchronization region is small, when compared to the time spent executing in parallel, it can be beneficial to add synchronization to a parallel region.

Static allocation of local variables in Fortran and variables defined within a new scope in C/C++ into the stack is an easy way to specify variables as private to each thread. This is typically specified with the `-automatic` option on most compilers if this is not the default. Since each thread executing a parallel region has its own stack, this ensures that whenever multiple threads concurrently call a routine, variables local to that routine are not shared between threads.

As we have seen, OpenMP directives permit variables to be made private in a particular parallel construct. The `private()` clause indicates, for the duration of a parallel region, that each thread executing the region will have a unique, local instance of each listed variable.

Example: Using private variables outside of parallel regions

Another class of errors occurs in the interfaces between parallel regions and sequential code. Consider the following sequential routine:

C/C++ syntax:

```
void dsq2(float a[], float b[], float c[], int n)
{
    float x, y;
    int i;
    for (i = 0; i < n; i++) {
        x = a[i] - b[i];
        y = b[i] + a[i];
        c[i] = x * y;
    }
    printf("%f %f\n", x, y);
}
```

Fortran syntax:

```
      subroutine dsq2( a, b, c, n )
      integer n
      real a(n), b(n), c(n), x, y
      do i = 1, n
           x = a(i) - b(i)
           y = b(i) + a(i)
           c(i) = x * y
      end do
      print *, x, y
      end
```

This program is identical to the previous example except that the final values of the variables x and y are propagated out of the loop to be printed. This sequential code could be parallelized as in the previous example.

C/C++ syntax:

```
void dsq2_a(float a[],float b[],float c[],int n)
{
    float x, y;
    int i;
    #pragma omp parallel for shared(a,b,c,n) \
                    private(i,x,y)
    for (i = 0; i < n; i++) {
        x = a[i] - b[i];
        y = b[i] + a[i];
        c[i] = x * y;
    }
    printf("%f %f\n", x, y);
}
```

Fortran syntax:

```
      subroutine dsq2_a (a,b,c,n)
      integer n
      real a(n), b(n), c(n), x, y
!$omp parallel do shared(a,b,c,n) private(i,x,y)
      do i = 1, n
          x = a(i) - b(i)
          y = b(i) + a(i)
          c(i) = x * y
      enddo
!$omp end parallel do
      print *, x, y
      end
```

Here, Assure identifies that the `private` variables `x` and `y` have their values used outside the parallel region. Since `x` and `y` are private to each thread executing the region, the values of these variables outside the region are undefined. The `lastprivate()` clause can be used to copy the values of `x` and `y` from the last serial iteration of the parallel loop back into the sequential code.

C/C++ syntax:

```
void dsq2_b(float a[],float b[],float c[],int n)
{
    float x, y;
    int i;
    #pragma omp parallel for shared(a,b,c,n) \
                    private(i) lastprivate(x,y)
    for (i = 0; i < n; i++) {
        x = a[i] - b[i];
        y = b[i] + a[i];
        c[i] = x * y;
    }
    printf("%f %f\n", x, y);
}
```

Fortran syntax:

```
      subroutine dsq2_b (a,b,c,n)
      integer n
      real a(n), b(n), c(n), x, y
!$omp parallel do shared(a,b,c,n) private(i)
!$omp& lastprivate(x,y)
      do i = 1, n
          x = a(i) - b(i)
          y = b(i) + a(i)
          c(i) = x * y
      enddo
!$omp end parallel do
      print *, x, y
      end
```

Assure identifies no errors in this parallel program. The `lastprivate()` clause specifies, during the execution of the parallel loop, that each thread is to have its own instance of the variables `x` and `y`, but that the values assigned on the last serial iteration of the loop are to be copied to the global `x` and `y` after the loop completes.

Example: Using firstprivate()

Another interface problem occurs in the transition between sequential code and parallel regions. Consider the following parallel routine:

C/C++ syntax:

```
void dsq3( float *c[], int n )
{
    float a[100], b[100], x, y;
    int i, j;
/*
Initialize all elements of arrays a and b
in the serial region
*/
    init_a( a, n );
    init_b( b, n );
    #pragma omp parallel for shared(a,b,c,n) \
                    private(i,j) lastprivate(x,y)
    for (i = 0; i < n; i++) {
        for (j = 0; j <= i; j++) {
/*
Re-assign some elements of arrays a and b
in the parallel region. In serial execution,
this step simply overwrites some of the values
set in routines init_a and init_b. In parallel
execution, this results in partially uninitialized
arrays a and b
*/
            a[j] = calc_a[i];
            b[j] = calc_b[i];
        }
        for (j = 0; j < n; j++) {
            x = a[j] - b[j];
            y = b[j] + a[j];
            c[i][j] = x * y;
        }
    }
    printf("%f %f\n", x, y);
}
```

Fortran syntax:

```
      subroutine dsq3 (c,n)
      integer n
      real a(100), b(100), c(n,n), x, y
c Initialize all elements of arrays a and b
c in the serial region
      call init_a( a, n )
      call init_b( b, n )
!$omp parallel do shared(a,b,c,n) private(i,j)
!$omp& lastprivate(x,y)
c Re-assign some elements of arrays a and b
c in the parallel region. In serial execution,
c this step simply overwrites some of the values
c set in routines init_a and init_b. In parallel
c execution, this results in partially
c uninitialized arrays a and 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(j) - b(j)
               y = b(j) + a(j)
               c(j,i) = x * y
          enddo
      enddo
!$omp end parallel do
      print *, x, y
      end
```

In this example, the arrays `a` and `b` are being used as temporary vectors for the calculation of the matrix `c`. However, not all of the values of `a` and `b` are initialized in the first j-loop before they are used in the second j-loop (an initial set of values for all relevant array elements are passed in to the loop from `init_a` and `init_b`). Assure reports Write → Write storage conflicts on `a` and `b` that can be removed by privatizing these variables to the parallel loop.

C/C++ syntax:

```
void dsq3_a( float *c[], int n )
{
    float a[100], b[100], x, y;
    int i, j;
    init_a( a, n );
    init_b( b, n );
    #pragma omp parallel for shared(c,n) \
       private(i,j,a,b) lastprivate(x,y)
    for (i = 0; i < n; i++) {
        for (j = 0; j <= i; j++) {
            a[j] = calc_a[i];
            b[j] = calc_b[i];
        }
        for (j = 0; j < n; j++) {
            x = a[j] - b[j];
            y = b[j] + a[j];
            c[i][j] = x * y;
        }
    }
    printf("%f %f\n", x, y);
}
```

Fortran syntax:

```
      subroutine dsq3_a (c,n)
      integer n
      real a(100), b(100), c(n,n), x, y
      call init_a( a, n )
      call init_b( b, n )
!$omp parallel do shared(c,n) private(i,j,a,b)
!$omp& lastprivate(x,y)
      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(j) - b(j)
               y = b(j) + a(j)
               c(j,i) = x * y
          enddo
      enddo
!$omp end parallel do
      print *, x, y
      end
```

This parallel program now has a different problem: each processor has its own private copy of a and b, but a and b are not fully-initialized on each processor because values of copies of `private` variables are initially undefined. Assure reports this error by identifying the uninitialized references to a and b inside the parallel loop. This type of error can be resolved through the use of the `firstprivate()` clause.

C/C++ syntax:

```
void dsq3_b( float *c[], int n )
{
    float a[100], b[100], x, y;
    int i, j;
    init_a( a, n );
    init_b( b, n );
    #pragma omp parallel for shared(c,n) \
    private (i,j) lastprivate(x,y) firstprivate(a,b)
    for (i = 0; i < n; i++) {
        for (j = 0; j <= i; j++) {
            a[j] = calc_a[i];
            b[j] = calc_b[i];
        }
        for (j = 0; j < n; j++) {
            x = a[j] - b[j];
            y = b[j] + a[j];
            c[i][j] = x * y;
        }
    }
    printf("%f %f\n", x, y);
}
```

Fortran syntax:

```
      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 )
!$omp parallel do shared(c,n) private(i,j)
!$omp& lastprivate(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(j) - b(j)
               y = b(j) + a(j)
               c(j,i) = x * y
          enddo
      enddo
!$omp end parallel do
      print *, x, y
      end
```

This parallel code is correct since the `firstprivate()` clause instructs each processor to begin with a private copy of the data in `a` and `b` initialized with values from the sequential code before the parallel loop. Each processor then (partially) overwrites its private copy of `a` and `b` with additional initialization data and proceeds with its computation.

CHAPTER 6

The KAP/Pro Drivers

About the KAP/Pro drivers

This chapter describes the functionality of the KAP/Pro drivers. There are many different driver names, depending on the source language and computational environment, but they all behave in essentially the same manner. The material in this chapter is organized as follows:

- Overview of the C/C++ Guide and Assure drivers
- Overview of the Fortran Guide and Assure drivers
- List of KAP/Pro driver options
- Environment variables that influence driver behavior

Overview of the C/C++ Guide and Assure drivers

The KAP/Pro C/C++ drivers, referred to here collectively as **guidec** and **assurec**, are designed to replace native compiler drivers such as `cc` or `icl`. The actual driver name depends on the operating system and native compiler; complete lists of Guide and Assure driver names are given in the tables on page 18 and page 32, respectively.

Both the Guide and Assure instrumentation and the compile/link steps are combined into one command line which may be invoked manually, from a script, or from a Makefile. The necessary C preprocessor, KAP/Pro, and compiler commands are executed automatically. The standard `link` command on Windows, which automatically adds the appropriate KAP/Pro library to the link step, can be replaced with **guidec** or **assurec** followed by the `-link` flag or alternatively by using the special drivers **guidel** or **assurel**.

Guidec and **assurec** are based on KAI C++, a high-performance, ISO standard-compliant C and C++ compiler. This reference manual documents only the places where KAP/Pro's default behavior differs from or extends upon KAI C++. Documentation for KAI C++ is located in the `KCC_docs` subdirectory within the Guide or Assure installation directory.

The default language of **guidec** and **assurec** is ANSI C, whereas the default language of KAI C++ is C++. To enable C++ in KAP/Pro on Unix systems, use the **`--c++`** command line switch or the **guidec++** or **assurec++** driver (C++ is currently not supported on Windows systems). To improve performance, **guidec** and **assurec** disable C++ exceptions by default. Exceptions can be enabled via the **`--exceptions`** command line switch.

In addition to all of the command line options accepted by the C/C++ compiler, the **assurec** and **guidec** drivers accept prefixed forms of all KAP/Pro options as well as driver-specific options. An absence of command line arguments causes the driver to emit a usage message.

Using the C/C++ drivers

To compile a C or C++ program with Guide or Assure, use one of the following command lines on Unix systems:

C program

```
guidec [<Guide options>] [<KAI C++ options>] <filenames>
assurec [<Assure options>] [<KAI C++ options>] <filenames>
```

C++ program

```
guidec++ [<Guide options>] [<KAI C++ options>] <filenames>
assurec++ [<Assure options>] [<KAI C++ options>] <filenames>
```

or one of the following command lines on Linux systems:

C or C++ program

```
guideicc [<Guide options>] [<icc options>] <filenames>
guideecc [<Guide options>] [<ecc options>] <filenames>

assureicc [<Assure options>] [<icc options>] <filenames>
assureecc [<Assure options>] [<ecc options>] <filenames>
```

or one of the following command lines on Windows systems:

C or C++ program

```
guideicl [<Guide options>] [<icl options>] <filenames>
guideecl [<Guide options>] [<ecl options>] <filenames>
guidec [<Guide options>] [<standard options>] <filenames>

assureicl [<Assure options>] [<icl options>] <filenames>
assureecl [<Assure options>] [<icl options>] <filenames>
assurec [<Assure options>] [<standard options>] <filenames>
```

where `<filenames>` is one or more input files to Guide or Assure.

Guide and Assure produce intermediate source files which are then passed to the underlying compiler. These files are removed by default after successful instrumentation and compilation. See “-WG[no]keep” on page 67 and “-WG[no]keeperr” on page 68 for more information.

The output filename from **guidec** or **assurec** is derived from the input filename by removing the file extension and adding the extension `.int.c`. The object file created by the driver does not have this suffix. For example, Guide or Assure would generate a file called `foo.int.c` from a file called `foo.c`, but the object file would be called `foo.o`.

Overview of the Fortran Guide and Assure drivers

The KAP/Pro Fortran drivers, referred to collectively as **guidef** and **assuref**, are designed to replace native compiler drivers such as `f90` or `ifl`. The actual driver name depends on the operating system and native compiler; complete lists of Guide and Assure driver names are given in the tables on page 18 and page 32, respectively.

On Unix systems, the Assure or Guide Fortran drivers replace the system FORTRAN 77 and Fortran 90 compilers on the command line and integrate Assure or Guide instrumentation and the compile/link step into one command line. In scripts and Makefiles, replacing the standard compiler (typically **f77** or **f90**) with the appropriate Assure or Guide driver will execute the necessary C preprocessor, Assure or Guide, and compiler commands automatically.

On Windows systems, the Assure or Guide Fortran drivers replace the Intel Fortran compiler or the Compaq Visual Fortran compiler and integrate Assure or Guide instrumentation and the compile/link step into one command line. Additionally, `assurel` and `guidel` replace the standard `link` to automatically add the Assure or Guide library to the link step.

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

Using the Fortran drivers

To compile a Fortran program with Guide or Assure, use one of the following command lines on Unix systems:

FORTRAN 77 program

```
guidef77 [<Guide options>] [<f77 options>] <filenames>
assuref77 [<Assure options>] [<f77 options>] <filenames>
```

Fortran 90 program

```
guidef90 [<Guide options>] [<f90 options>] <filenames>
assuref90 [<Assure options>] [<f90 options>] <filenames>
```

or one of the following command lines on Linux systems:

Fortran program

```
guideifc [<Guide options>] [<ifc options>] <filenames>
guideefc [<Guide options>] [<efc options>] <filenames>

assureifc [<Assure options>] [<ifc options>] <filenames>
assureefc [<Assure options>] [<efc options>] <filenames>
```

or one of the following command lines on Windows systems:

Fortran program

```
guideifl [<Guide options>] [<ifl options>] <filenames>
guideefl [<Guide options>] [<efl options>] <filenames>
guidef [<Guide options>] [<standard options>] <filenames>

assureifl [<Assure options>] [<ifl options>] <filenames>
assureefl [<Assure options>] [<ifl options>] <filenames>
assuref [<Assure options>] [<standard options>] <filenames>
```

where `<filenames>` is one or more input files to Guide or Assure.

Guide and Assure produce intermediate source files which are then passed to the underlying compiler. These files are removed by default after successful instrumentation and compilation. See “-WG[no]keep” on page 67 and “-WG[no]keeperr” on page 68 for more information.

The output filename from **guidef** or **assuref** is derived from the input filename by adding the prefix `G_` or `A_`, respectively. For example, Guide would generate a file called `G_foo.f` from a file called `foo.f`, and Assure would generate a file called `A_foo.f`.

Fortran files with capitalized suffixes (e.g. `filename.F`) are first passed through the C preprocessor before Assure or Guide is invoked. The C preprocessor will create files with a `cppA_` or `cppG_` prefix (e.g. `cppA_filename.F`). As mentioned above, **assuref** and **guidef** will create output files whose name is based on the original source file name.

KAP/Pro driver options

Default processing by Assure and Guide is to compile all source files listed on the command line and link them to produce an executable. This behavior can be modified by command line options and/or environment variables, as described in the remainder of this chapter. Use the **-WGhelp** flag on the Guide or Assure command line for a list of KAP/Pro options.

Most of these options only directly affect the behavior of Guide and Assure and are not passed to the underlying compiler; exceptions to this rule, such as **-c**, are identified in the individual descriptions. If Guide or Assure fails to recognize a command line option, it simply ignores it and passes it to the compiler.

The **guidec** and **assurec** drivers recognize all the KAI C++ compiler options (on Unix), Intel C++ compiler options (on Windows), and many of the Microsoft Visual C++ compiler options (on Windows). The **assuref** and **guidef** drivers recognize all the Intel Fortran compiler options (on Window) and most of the native Fortran compiler options (on Unix). In addition, all KAP/Pro drivers recognize several OpenMP-related options.

The following conventions apply to the KAP/Pro drivers:

- Anything in the form **-WG*xxx*** is a KAP/Pro driver option. The **-WG** distinguishes it from standard compiler options.
- Anything in the form **-WG,-*xxx*** is an option that affects the internal workings of KAP/Pro. Normally, these options are not needed by end users.
- Anything undecorated, such as **-O3**, is an underlying compiler option. Some of these options, such as the **-v**, **-c**, and **-wnowarn** flags described below, influence both KAP/Pro and the underlying compiler.

Some driver options are listed with **[no]** as part of the name; this means that both positive and negative settings, such as **-WGkeep** and **-WGnokeep**, are accepted. Some driver options are specific to a particular KAP/Pro component or computational platform; these are listed in individual sections after the alphabetical list.

Displaying all Command Lines

The **-v** option causes the driver to display all command lines executed. This flag is passed on to the compiler. Use -WGnorpath (Unix only) if your compiler does not recognize the **-v** option.

Disabling automatic linking of object files

If the **-c** compiler option is included, the drivers will only compile the source files. If Assure or Guide is unable to correctly process one or more source files, all other source files within the command will be compiled (but not linked) regardless of whether or not the **-c** option is present.

Suppressing warnings (Fortran only)

Use the **-wnowarn** option to suppress mild **assuref** and **guidef** warnings. This flag is passed on to the compiler.

Additional KAP/Pro driver options

Guide and Assure accept several advanced options that can be specified by the **-WG,...** driver option. These options have the following syntax:

```
-WG,assure_option_1[[[,assure_option_2],assure_option_3
            ],...]
-WG,guide_option_1[[[,guide_option_2],guide_option_3],.
            ..]
```

A list of these additional options, which are normally not needed by end users, is given in appendix D, “Additional KAP/Pro Options,” beginning on page 199

Alphabetical listing of Driver Options

In the following descriptions, **<integer>** indicates an integer number, **<path>** indicates a directory name, **<file>** indicates a file name, possibly including a full path, **<character>** indicates a single character, and **<string>** indicates a string of characters. All other bracketed symbols are strings whose values are option-dependent; legal values for these symbols will be listed in the option description. Every driver option should be preceded by the “**-**” character; Windows users can also use the “/” character.

-WGcatch=<class> (Unix C/C++ only)

This option instructs Guide or Assure to intercept certain exceptions which violate the OpenMP API and abort with an error message at run-time. Legal values for `<class>` are `all`, `safe`, and `none`. The default is **none**.

The OpenMP standard requires that exceptions thrown within an active parallel construct must cause execution to resume within the dynamic extent of the same OpenMP construct in which the throw occurs. In addition, under the same conditions, the exception must be thrown and caught by the same thread. Setting this switch to "`all`" or "`safe`" will cause exceptions that violate these rules to be intercepted. When this occurs, the program will exit with an error message.

The "`-WGcatch=all`" setting causes the program to intercept and report exceptions which violate the OpenMP API for all OpenMP constructs. This option has the largest run–time overhead. Use this option to help determine whether your application has OpenMP-compliant exception handling.

The "`-WGcatch=safe`" setting causes the program to intercept and report exceptions which violate the OpenMP API for only the C/C++ `parallel`, `parallel for`, `parallel sections`, `parallel for`, `taskq`, and `task` constructs. This option has medium run–time overhead.

The "`-WGcatch=none`" setting causes the program to ignore exceptions which violate the OpenMP API. A program which violates the OpenMP exception rules may exhibit unpredictable behavior with this setting. This option has no run-time overhead and is the default. Use this setting if you are confident your application has OpenMP-compliant exception handling.

-WGcheck=<string> (Assure only)

Controls the overall speed at which Assure checks a program for errors. The **fast** setting typically reports the fewest errors while the **slow** setting reports *all* errors. The **medium** setting typically finds most errors, but runs faster than the slow setting. The default is **slow.**

-WGcompiler=<path>
-WGcc=<path> (C/C++ only)
-WGftn=<path> (Fortran only)
-WGfortran=<path> (Fortran only)
-WGf77=<path> (Fortran only)
-WGf90=<path> (Fortran only)

This option is used to specify an alternative path to the native compiler. The default is determined when the KAP/Pro Toolset is installed.

-WG[no]cpp

This option forces the C preprocessor to be run on all source files. Normally the driver will invoke the C preprocessor in situations where the native compiler would do the same. On most Unix platforms, for example, the C preprocessor is invoked for Fortran source files with a capital `F` in the extension (`.F`, `.F77`, `.F90`).

-WGcpp=<file>

This option allows you to specify an alternate path for the C preprocessor executable.

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
```

-WGcritname=<pattern>

This option applies to mixed language programs to allow matching of named and unnamed `critical` and `ordered` directives in C/C++ to their Fortran counterparts. Valid values are `lower`, `upper`, `_lower`, `_upper`, `lower_`, `upper_`, `_lower_`, and `_upper_`. The default value is chosen to match the default behavior of the native Fortran compiler.

Guide creates a global lock object for every named and unnamed critical and ordered section in the source code. An unnamed section and a named section with name `Foo` would be translated, respectively, as follows:

<pattern>	Symbol for unnamed section	Symbol for named section "Foo"
lower	mpp_uc_none	mpp_nc_Foo
upper	MPP_UC_NONE	MPP_NC_Foo
_lower	_mpp_uc_none	_mpp_nc_Foo
_upper	_MPP_UC_NONE	_MPP_NC_Foo
lower_	mpp_uc_none_	mpp_nc_Foo_
upper_	MPP_UC_NONE_	MPP_NC_Foo_
lower	_mpp_uc_none_	_mpp_nc_Foo_
upper	_MPP_UC_NONE_	_MPP_NC_Foo_

-WG[no]debug (Fortran only)

When **-WGdebug** is specified, `#line` directives are generated in the intermediate source file (typically beginning with the prefix `G_` when using Guide or `A_` when using Assure) so that debuggers can relate back to the actual input source file and not the intermediate files. The default is **-WGdebug** on most platforms, except on platforms where the underlying Fortran compiler is unable to process `#line` directives.

-WGdefault=<class>

This option specifies the default classification of unlisted variables in OpenMP `parallel` directives. Its effect is as if `default(<class>)` were placed on every parallel directive that doesn't have an explicit `default(…)` clause. Allowed values of `<class>` are `shared` and `none`. When not in strict OpenMP mode, the value `private` is also allowed. The default value is **shared**.

-WGdefault_library

By default, Guide and Assure link using static libraries on Windows systems and shared libraries on Unix systems. This option instructs the driver to use the default linking conventions when linking the Guide or Assure libraries into the generated executable. See also -WGstatic_library and -WGdynamic_library.

-WGdynamic_library

By default, Guide and Assure link using static libraries on Windows systems and shared libraries on Unix systems. This option instructs the driver to dynamically link the Guide or Assure libraries into the generated executable. See also -WGstatic_library and -WGdefault_library.

-WGfullpath

Use this option to make all filenames and directories fully qualified. This option can sometimes improve the functionality of debugging tools; see also -WG[no]debug on page 66.

-WGhelp

This option directs the driver to print a usage message and exit (Windows users can also type -**?**, -**h**, or -**help**).

-WGimplylang (Windows C only)

If this option is specified, KAP/Pro will assume that source files ending in ".cpp" or ".cxx" are C++, thus allowing mixed C and C++ compilation in a single command line.

-WGincpath

This option specifies an alternate path in which to search for the Assure or Guide include files.

-WG[no]keep

Instrumented source files (Assure or Guide output files) and temporary C preprocessor files are removed by default after successful Assure or Guide instrumentation and compilation. There are several instances where output files are not removed:

- When Assure or Guide fails to process a 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 Assure or Guide successfully instruments, none of the output files are removed.

- If you specify **-WGkeep**, none of the output files are removed.

The presence of the **-WGnokeep** flag overrides any previous instance of **-WGkeep** on the command line, including the **-WGkeep** implied from **-WGonly** and **-g** (Unix) or **-Zi** (Windows).

-WGkeepcpp

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

-WG[no]keeperr

If **-WGnokeeperr** is stated, then Guide and Assure will remove intermediate files even if there are compile errors. The default is **-WGkeeperr**.

-WG[no]keepobjects

If **-WGkeepobjects** is stated, then object files will not be removed after linking. The default is **-WGkeepobjects** except when compiling a single source file on Unix systems.

-WGlibpath=<path>

This option specifies an alternate path in which to search for the Assure or Guide libraries at link time.

-WGlink=<file>
-WGld=<file>

This option allows you to specify an alternate filename for the linker executable.

-WGlocation=<string> (Assure only)

Controls the level of accuracy with which Assure pinpoints errors in a program. The **exact** setting will determine the exact location of errors while the **approx** setting will approximate the location of errors. The default is **exact**.

-WGnoimply=<kwd>[,<kwd>...] (not C/C++ Unix)

Each keyword describes a type of switch that the driver should avoid implying. These driver switch implications are usually made to make sure particular switches or libraries are passed to the backend compiler to ensure correct KAP/Pro operation in typical/default cases.

Use these keywords to have the driver avoid passing certain switches to the back-end compiler and/or linker:

- **auto**: switches like **-automati**c specifying that local variables should be allocated from the stack
- **threads**: switches that make compilation thread-safe
- **align**: switches that set up the correct variable alignment/padding in the back-end compiler
- **io**: switches that make runtime I/O libraries thread-safe
- **rpath**: switches like **-rpath** that set the path to the KAP/Pro runtime libraries when doing shared linking (applicable Unix platforms only)

Use these keywords to have the driver avoid putting certain libraries on the link line:

- **extra_lib**: any extra libraries that the driver might have to add for correct linking on a per-platform basis (e.g., **-lnsl** on Solaris)
- **kpts_lib**: the main KAP/Pro runtime library
- **threads_lib**: the system threading library used by kpts_lib
- **all_lib**: same as **extra_lib**, **kpts_lib**, **threads_lib**

-WGnorc

This flag will turn off driver-specific options that were found in any initialization file in your home directory (e.g. `$HOME/.assurefrc` or `$HOME/.guidefrc` on Unix or `$HOME\.assureini` or `$HOME\.guideini` on Windows). 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 to allow all succeeding options to be used.

-WGnorpath (Unix only)

Normally, Guide or Assure encodes the location of shared libraries into an executable. This option instructs the driver to omit the path to shared libraries. Often, when this option is used, the `LD_LIBRARY_PATH` variable must be set at run-time to locate the Guide or Assure libraries.

-WGnowork

This option tells the driver to only print the commands it would normally execute.

-WGonly

When **-WGonly** is used, Assure or Guide will process the source code in all listed source files, but neither the compiler nor linker will be executed. This option implies the **-WGkeep** option.

-WG[no]openmp (Guide only)

Setting **-WGnoopenmp** specifies that Guide is being run for profiling purposes only, and that OpenMP directives are to be ignored. This flag can be used along with **-WGprof** to enable profiling but not OpenMP.

-WGopt=<integer>

This option sets the optimization level for OpenMP directives. Valid values are the integers 0 through 3.

Level 0 optimization disables all directive optimizations.

Level 1 optimization attempts to remove unnecessary `barrier` directives from the code.

Level 2 includes level 1 optimizations and is reserved for future use.

Level 3 includes level 1 and 2 optimizations and adds parallel region merging.

The default value is **3**.

-WGpath=<path>

This option is used to specify an alternate path to the Guide or Assure executable. The default is determined when the KAP/Pro Toolset is installed.

-WG[no]perview (Guide only)

This option specifies whether the Guide driver uses the application management and monitoring version of the Guide run–time library. See Chapter 9, “PerView,” beginning on page 101 for a complete description of this library. The default is **-WGnoperview**.

-WGprefix=<string>

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

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

the results are `cppqqqfile1.f` and `qqqfile1.f` instead of the default `cppA_file1.f` and `A_file1.f`.

-WG[no]process

This option specifies whether Guide or Assure processes OpenMP directives into parallel code. The **-WGnoprocess** flag can be used to compile source code that has already been processed by Assure or Guide or to bypass processing of files that cannot be handled. Note that Assure may report false errors if files not processed by Assure are linked with successfully processed files. See also the description of -WG[no]openmp. The default is **-WGprocess**.

-WG[no]prof

Specifying **-WGprof** activates standard performance profiling for Vampir and GuideView; this flag implies **-WGstats**. Specifying **-WGnoprof** disables profiling. The default behavior is **-WGnoprof**.

-WGprof_leafprune=<integer>

Sets the minimum size of procedures to retain in the Vampir or GuideView profile. Guide will not instrument the entry and exit of leaf subroutines that have fewer than

`integer` lines. This scheme may significantly reduce the tracefile size and the overhead due to subroutine instrumentation.

-WGproject_name=<file> (Assure only)
-WGpname=<file> (Assure only)
-WGprj=<file> (Assure only)

Any of these equivalent options specifies a name for the Assure project file, the program database that Assure uses to record static information about the application. This information is then merged with the data gathered at runtime to be displayed by AssureView. A project file is required for applications with more than one source file. If an application's source files are spread over multiple directories, then an absolute path to the project file is required, for example:

```
-WGpname=/home/me/apps/myproject.prj
```

If the specified project file name does not end in the suffix `.prj`, it is added automatically. By default, Assure will create a project file named **assure.prj** if this option is not specified. In this mode Assure requires all source files to be contained in one directory.

-WGsched=<type>[,<integer>]

This option specifies the default scheduling type and optional chunk size for C/C++ `for` and Fortran `do` directives. Its effect is as if `schedule(<type>[,<integer>])` were placed on every C/C++ `parallel for`, `for` and Fortran `parallel do`, `do` that doesn't have an explicit `schedule(…)` clause. Allowed values of `<type>` are `static`, `dynamic`, `guided`, and `runtime`. Valid values of the optional `<integer>` chunk size are positive integers. The default value is **static**, with no chunk size. For `dynamic` and `guided`, the default chunk size is **1**. See "Scheduling Options," beginning on page 139 for many more details.

-WGsrcdir

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

-WGstatic_library

By default, Guide and Assure link using static libraries on Windows systems and shared libraries on Unix systems. This option instructs the driver to statically link the Guide or Assure libraries into the generated executable. See also -WGdynamic_library and -WGdefault_library.

-WG[no]stats (Guide only)

This option specifies whether the Guide driver uses the statistics version of the Guide run-time library. See Chapter 7, "GuideView," beginning on page 81 for more information on this option. The default is **-WGnostats**.

-WG[no]strict

This option specifies whether Guide or Assure in strict mode, in which it flags non-standard usage of OpenMP directives as errors. KAP/Pro Toolset's OpenMP extensions include:

- `psingle`, `psections`, and `pfor` are accepted as synonyms for the C/C++ `single`, `sections`, and `for` directives, respectively.
- The `ordered` clause is allowed on the `sections` directive and `ordered` directives are allowed within `section` blocks.
- The `lastprivate` and `reduction` clauses are allowed on `single` directive.
- A `default(private)` clause is allowed on the `parallel` directive.
- The C/C++ `taskq` model of unstructured parallelism is enabled.
- The curly braces may be omitted for the C/C++ `sections` directive if it contains only a single `section`.
- The `threadprivate` directive can be used with local static variables in C.

The default is **-WGnostrict**.

-WGuser=<string>

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 Assure or 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 correspond 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 usrA_ or usrG_ to the name of the file being processed by Assure or Guide, respectively. For example, if the file being processed is named x.f, the output file would be named usrA_x.f. This file is then passed on to Assure or Guide, respectively.

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

The table below gives the command that the driver will execute when x.f is the file being processed.

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

-WGversion

When this option is present, Guide or Assure displays its version number to stderr. A source file must be supplied on the command line for version information to be printed.

Environment Variables for Guide

The following environment variables affect the run time behavior of Guide-processed executables. All of the standard OpenMP environment variables are also recognized; see “OpenMP Directives,” beginning on page 111 for details.

KMP_BLOCKTIME=<integer>[<character>]

This variable specifies the number of milliseconds that the Guide libraries should wait after completing the execution of a parallel region before putting threads to sleep. Use the optional character suffix `s`, `m`, `h`, or `d` to specify seconds, minutes, hours, or days. The default value of **1000** (one second) is used if `KMP_BLOCKTIME` is not specified. 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_IGNORE_MPPBEG <integer>

If this environment variable is set to 1, then all explicit calls to `MPPBEG` and `__kmpc_begin()` are ignored.

KMP_IGNORE_MPPEND <integer>

If this environment variable is set to 1, then all explicit calls to MPPEND and __kmpc_end() are ignored.

KMP_INTERVAL <integer>[{s,m,h,d}]

By default a program that has been instrumented with Guide stats library will write its results to the stats file (`.gvs` file) when the program terminates. Guide has a control that allows performance information to be written periodically. The environment variable `KMP_INTERVAL` indicates the time interval that the program waits before updating a partial stats file. For example, if `KDD_INTERVAL` is set to `5m`, then every 5 minutes the program will update the stats file with all the results found during the last time interval. If no new results were found, the file will be left unchanged. Any open GuideView windows will not be updated; it is necessary to restart GuideView to see any new results.

Valid suffixes for the time interval, an integer number, are `s` (seconds), `m` (minutes), `h` (hours), and `d` (days). The default suffix is **m**.

KMP_LIBRARY=<string>

This variable selects the Guide run time library. The three available options for `<string>` are:

- `serial`
- `turnaround`
- `throughput`

The default value of **throughput** is used if `KMP_LIBRARY` is not specified. See Chapter 4, "Libraries and External Routines," beginning on page 21 for more information about the Guide libraries.

KMP_STACKOFFSET=<integer>[<character>]

If no suffix is specified, the value of `integer` is interpreted as given in bytes. On 32-bit Microsoft Windows platforms, setting `KMP_STACKOFFSET` causes each worker thread's stack to be padded with `omp_get_thread_num*KMP_STACKOFFSET` bytes, relative to the initial stack base address. Use the optional suffix `b`, `k`, or `m` to specify bytes, kilobytes, or megabytes. For Pentium 4 processors and earlier, the default value of `integer` is **0**.

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 of **1m** (one megabyte) is used if `KMP_STACKSIZE` is not set. This default value may be too small if many private variables are used in the parallel regions, or if the parallel region calls subroutines that have many local variables.

Windows users should be aware that executables contain stack size information that can be modified with the "editbin" command. For example, to change the stack size of executable `program.exe` to 16 megabytes, type:

```
editbin /STACKSIZE:16000000 program.exe
```

from a command prompt.

The default value for `<character>` is **b**; in this case `KMP_STACKSIZE` will be set to `<integer>` or 8192, whichever is larger.

KMP_STATSCOLS <integer>

Specifies how many columns are in the Guide stats file which is produced (**guide.gvs** by default) when running an executable that has been compiled with **-WGstats**. The default value is **80**.

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 file is **guide.gvs**.

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:

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

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

`%P`: 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.

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 are installed when you run your application if you compile with shared objects or use dynamic linking.

Environment Variables for Assure

The following environment variables affect the run-time behavior of Assure–processed executables. All of the standard OpenMP environment variables, with the exception of OMP_NUM_THREADS, are also recognized; see “OpenMP Directives,” beginning on page 111 for details.

KDD_OUTPUT <file>

The KDD_OUTPUT environment variable is used to specify where the output of the simulation is stored. The .kdd extension is automatically appended to the end of the filename if it is not specified. If not specified, the base name of this filename is the same as the base of the .prj filename. Both the project file (.prj) and output file (.kdd) must be specified to the AssureView viewer when their base names do not match.

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

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

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

%P: This expands into the value of the OMP_NUM_THREADS environment variable.

KDD_INTERVAL <integer>[{s,m,h,d}]
KDD_DELAY <integer>[{s,m,h,d}]

By default a program that has been instrumented with Assure will write its results to the output file (.kdd file) every fifteen minutes.

For some programs this may be too often or too seldom. Assure has two controls that help solve this problem. The environment variable KDD_INTERVAL indicates the time interval that the program waits before updating a partial results file. For example, if KDD_INTERVAL is set to 5m, then every five minutes the program will update the results file with all the results found during the last time interval. If no new results were found, the file will be left unchanged. Any open AssureView windows will not be updated; it is necessary to restart AssureView to see any new results.

Valid suffixes for the time interval, an integer number, are s (seconds), m (minutes), h (hours), and d (days). If no suffix is specified, the unit of time is assumed to be minutes.

The second control, KDD_DELAY, deals with the problem of long running programs by letting the program run without error checking for a specified period of time. After the period has elapsed, the program starts checking for errors on entry to the next parallel region. This variable is also specified as a time duration. For example, if KDD_DELAY is set to 30m, then for the first 30 minutes of the program's execution, no errors will be checked, and no errors recorded. After the 30 minutes has elapsed, Assure will turn on error checking on entry to the next parallel region. Once error checking is enabled, the KDD_INTERVAL variable is used to determine how often updates to the results file are to be done.

If you only want to calculate the amount of stack memory used, set KDD_DELAY to a large number so that the program finishes before the time elapses.

KDD_MALLOC

The KDD_MALLOC environment variable is used to control how storage allocated via malloc() calls inside parallel regions but outside worksharing constructs is treated by the Assure simulator.

To make such storage shared, set KDD_MALLOC to one of the following values:

- shared
- 1
- true

To make such storage private, set KDD_MALLOC to one of the following values:

- private
- 0
- false

The default is private for such storage. Any storage allocated in the serial part of the program or inside a worksharing construct is always considered shared.

Preprocessor Macros

Several preprocessor macros are defined which may be useful when different instructions should be executed depending on the guild environment. As an example, the following lines could be inserted into your source code if some instructions are to be executed only when compiling with Guide:

```
#ifdef _GUIDE

... executed only when compiling with Guide

#endif
```

_OPENMP

This has the form `YYYYMM` where `YYYY` is the year and `MM` is the month of the OpenMP Fortran specification supported.

_GUIDE

This is defined only when compiling with Guide.

_ASSURE

This is defined only when compiling with Assure.

CHAPTER 7

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 that indicate where attention should be focused, i.e. regions of the code where improving local performance would have the greatest impact on overall performance.

Using GuideView

GuideView uses as input the statistics file that is output when a Guide instrumented program is run. See "Libraries and External Routines," beginning on page 21 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 4). Optional GuideView arguments are the topic of a subsequent section.

The GuideView browser looks for a configuration file named `GVproperties.txt` when it starts up. The directory search order is first 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. Under Windows, a home directory can be specified via the `HOME` environment variable.

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. The example initialization file will be in

```
/class/example.GVproperties/
```

under the Guide installation directory. 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 in its extensive online help system, under the **Help** menu.

Using Named Parallel Regions

By default, parallel regions are identified only by the file that contains the region. It is also possible to associate a specific name with one or more parallel regions. Such regions are known as "named parallel regions," or simply "named regions." To name a parallel region, call the external routine `kmp_set_parallel_name`. This routine takes a character string name for the region as an argument.

Once enabled, all following parallel regions are assigned the most recently supplied name, until named regions are disabled by a call to `kmp_set_parallel_name` with an empty string. The *guide_stats* library gathers performance statistics separately for each named parallel region.

A simple use of this feature is to name a parallel region of interest so that its performance statistics can be readily located in the GuideView display. The following program illustrates this. This approach can be extended to multiple parallel regions, by using the same or different names. Even when multiple parallel regions have the same name, however, their performance statistics are shown separately by GuideView.

C syntax:

```
#include <omp.h>
main() {
  /* The following parallel region is named
     "REGION1". */
  kmp_set_parallel_name( "REGION1" );
  #pragma omp parallel
    work( iiter );


  /* The following parallel region is named
     "REGION2". */
  kmp_set_parallel_name( "REGION2" );
  #pragma omp parallel
    work( jiter );

        ...

  /* Naming is disabled for this and subsequent
     regions. */
  kmp_set_parallel_name( "" );
  #pragma omp parallel
    work( kiter );

}


void work( int niter ) {

  int i;

  #pragma omp for private(i)
    for( i = 0; i < niter; i++ ) {

        ...

    }
}
```

Fortran syntax:

```
      program use_region_1
      external kmp_set_parallel_name

        ...

! The following parallel region is named "REGION1"
      call kmp_set_parallel_name('REGION1')
      !$omp parallel
      call work( iiter )
      !$omp end parallel

        ...

! The following parallel region is named "REGION2".
      call kmp_set_parallel_name('REGION2')
      !$omp parallel
      call work( jiter )
      !$omp end parallel

        ...

      ! Naming is disabled for subsequent regions.
      call kmp_set_parallel_name( '' )
      !$omp parallel
      call work( kiter )
      !$omp end parallel

        ...

      end

      subroutine work( niter )

      !$omp do
      do i = 1, niter

        ...

      end do
      !$omp end do

      return
      end
```

Named regions can also be used to split the performance statistics of a parallel region for different data sets. In the following example, the parallel region of interest is assigned a name based upon the size of the data set. During a run, the parallel region is executed multiple times, each time with a different data set that activates different names for the parallel region. Performance statistics are gathered separately for each range of data sizes, and the statistics are associated with the appropriate names in the *guide_stats* report and GuideView display. The separate sets of statistics allow analysis of the parallel region as a function of the data set size.

C syntax:

```
#include <omp.h>
main() {

  ...

  for(i = 0; i < nsizes; i++) {
    int iter = isizes[i];

    if ( iter <= n1 )
          kmp_set_parallel_name("FIRST BIN");
    else if ( iter <= n2 )
          kmp_set_parallel_name("SECOND BIN");
    else
          kmp_set_parallel_name( "" );

    #pragma omp parallel
      work(iter);
  }
}
```

Fortran syntax:

```
      program use_region_2
      external kmp_set_parallel_name

        ..

      do i=1,nsizes

        iter = isize(i)

        if ( iter .le. n1 ) then
          call kmp_set_parallel_name('FIRST BIN')
        else if ( iter .le. n2 ) then
          call kmp_set_parallel_name('SECOND BIN')
        else
          call kmp_set_parallel_name( '' )
        end if

        !$omp parallel
        call work( iter )
        !$omp end parallel

      end do

      end
```

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. This example file resides in

`/class/guide.ovh/`

under the Guide installation directory.

-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
-Djava.compiler=none

The **-nojit** or **-Djava.compiler=none** option disables the Java just-in-time compiler. This Java feature can sometimes lead to incorrect Java behavior. Use **-WJ**,**-nojit** or **-WJ**,**-Djava.compiler=none** to disable the just-in-time compiler if you experience problems with either the GuideView or AssureView GUI.

Measuring OpenMP Overhead

The following table demonstrates the amount of time expended for OpenMP directives compared to a null call for a typical microprocessor based SMP. A null call is a call to an empty function.

```
C/C++ syntax:
void null(){};


Fortran syntax:
subroutine null
return
end
```

As shown in the table below, it took about ten cycles to call the null function. A `barrier` construct is about ten times slower for one processor, and about 70 times slower for two processors.

The program to produce this information is included in your Guide distribution. Please run it to calibrate your particular environment. You can use this information to determine the relative costs of various Guide constructs.

Typical Overhead

	1 processor		2 processor		3 processor		4 processor	
Guide Construct	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 two processors, all the costs follow a nearly linear pattern as you add processors.

CHAPTER 8

AssureView

Introduction

AssureView displays the results of Assure instrumentation by using the project file information produced by Assure and the simulation output produced by running the Assure-compiled program. The results can be viewed via the AssureView Graphical User Interface (GUI) or as text output.

The AssureView output describes all the errors identified by Assure and pinpoints their exact locations in the source code. The AssureView GUI allows you to browse the errors associated with each parallel construct and to view the corresponding offending locations in the source code.

Documentation for the features and usage of the GUI is available within the GUI itself, under the **Help** menu on the menu bar.

Using AssureView

AssureView takes as its primary arguments a project file (`.prj` suffix) and a simulation output file (`.kdd` suffix) from Assure. By default, AssureView output is displayed via the GUI. If the **-txt** option is used, text output is produced on the standard output instead. When the GUI is used, AssureView also produces an *AssureView GUI Input* file (`.agi` suffix) that may be used subsequently with the AssureView GUI in place of the project and simulation output files.

Several aspects of the AssureView browser, such as fonts, window size, window location, line numbering in source, etc... can be configured by using an initialization file. This file is named `.assureviewrc` on Unix systems or `assure.ini` on Windows systems. AssureView looks for the configuration file, in order, in the current directory, in your home directory, and in the directories listed in the `CLASSPATH` environment variable.

This capability of reading initialization files is included primarily for backwards compatibility; most, if not all, of these options can be controlled by the **Preferences** menu in the GUI (see "How to Use the GUI" on page 94).

An example of the configuration file is provided with the Assure installation. If Assure is installed in directory `<install-dir>` on your machine, the example file that explains the options available will be in

```
<install-dir>/class/example.assureviewrc.
```

The default location for this example configuration file is in the directory:

```
/usr/local/KAI/assure40/class/example.assureviewrc
```

on Unix and

```
C:\kai\assure40\class\example.assureviewrc
```

on Windows.

If the install location on Unix is different from the default location, then a symbolic link will be created from the default location to the installed location, providing that the default location is writable at install time. The easiest way to use this file is to copy it to a new file, and then edit it as needed. To change settings, uncomment the desired lines and set the options to preferred values.

The following examples show the most common ways of invoking AssureView:

```
assureview
```

When AssureView is run with no arguments, it uses the default project name, `assure.prj`, and the default run file, `assure.kdd`, in the current directory. The results are displayed using the AssureView GUI. This produces an `assure.agi` file that can be used with a subsequent "`assureview -agi=assure`" command.

```
assureview -txt
```

Run AssureView when Assure was run on a single-file program and no **-WGpname=** was specified to Assure. Output the results as text to the standard output.

```
assureview myprogram
```

Run AssureView when Assure was run on a multi-file program with **-WGpname=myprogram** specified to Assure. Use the AssureView GUI to display the results. This produces a `myprogram.agi` file that can be used with a subsequent "`assureview -agi=myprogram`" command.

```
assureview <path_to_project_file>/myprogram.prj
```

Run AssureView when the project file is located in a different directory than the directory in which the program was run. AssureView will read the run data from the file `myprogram.kdd`, located in the current directory. This also produces a `myprogram.agi` file.

```
assureview <path1>/myprogram.prj
<path2>/myprogram.kdd
```

Run AssureView with a specific project file and specific run data file.

AssureView GUI Elements

The AssureView GUI displays the following types of information in its various windows:

- A main error list that summarizes and displays errors found in a program by Assure.
- Graphs that display error counts by location in a program.

- Source code display windows (accessible by selecting a particular error in the error list) that display the source location(s) associated with a selected error.
- A whole-program dynamic call tree display (accessible from the **Windows** menu).
- Windows that display the dynamic call sequences (call stack) made to arrive at particular source code locations in source code display windows.
- Windows that allow searching for strings in the error list and in source code.

How to Use the GUI

Start the GUI by invoking AssureView with options other than **-txt** or **-nogui**. If any errors were found by Assure, the main error list is displayed to summarize these errors and their locations. Lines in the error list are marked with red octagons for serious errors, orange diamonds for less serious cautions, yellow triangles for warning conditions, and green check marks for areas where no errors occurred. Clicking on one of these errors causes the source code location(s) associated with that error to be shown in source code display window(s) with the same red, orange, and yellow markings on the offending lines. Errors are grouped in the error list according to the parallel construct in which they occurred.

The colored graphs at the bottom of the window display the number of errors, cautions, and warnings for constructs that were run. Constructs that were not run are also identified in blue. Clicking on a graph will highlight the list of errors associated with that construct. A separate panel shows graphs for program wide problems, such as insufficient stack space.

From a source code display window, the dynamic subroutine call sequence that occurred to arrive at the displayed point in the source code can be seen by pressing the “Show Stack” button. Clicking on one of the calls in this display will cause the location of that call to be displayed in the source code display window.

The “CallTree” option in the **Windows** menu causes the whole-program dynamic call tree to be displayed. At a given level of this display, a subroutine’s name can be seen; below this name, a list of all the subroutines that were called from this calling subroutine will be displayed, each preceded by the line number in the calling subroutine. An icon on each line gives the depth (number of sub-

routines) in the call tree below that line. For instance, an “8” icon on a line for a subroutine indicates eight more levels of subroutines in the call tree below that subroutine; a “>” icon indicates that there are more than nine levels. Clicking on a line in this display will cause the location of that subroutine or call site to be displayed in a source code display window. The call tree also displays the locations of all parallel constructs encountered during the run. Individual constructs can be shown or hidden via toggle buttons located at the bottom of the window.

The **Search** menu and the “Go Search” button bring up windows that allow searches of the error list and source code display windows to be performed.

The **Print** menu and the “Printer” button on the toolbar allow you to print the main error list or the call tree information to a printer or to a file. Individual call stack displays can also be printed.

The **Preferences** menu controls many aspects of AssureView. You can specify terse or verbose messages, whether to number source lines, how searching works, appearance (look-and-feel), fonts, colors, search directories for source code files, and other preferences. If you choose to save these preferences, a file with suffix “`.opt`” is created in the current directory. Copy this file to your home directory if you want these saved preferences to be used every time you use AssureView.

An option exists to operate the GUI in a low-memory mode (which typically runs more slowly) when examining data from particularly large programs. These options are further explained within the online **Help** menu. While AssureView still supports an initialization file, the **Preferences** menu offers a broader set of options.

When working with AssureView, you may want to ignore or hide certain classes of error messages. The **Preferences** menu option “Hiding Errors” allows you to hide errors based upon their priority, their type, or upon rules you create. Also available on the toolbar is an “Eye” (Hide Error) button. Clicking this button automatically creates a new rule to hide the currently selected error message.

Low Priority Errors occur when the semantics of the parallel program and serial program differ, but Assure has determined that the difference likely is not a programming error. Such errors can occur, for example, in parallel reductions. AssureView flags these messages as “Low Priority”, and hides them by default.

Custom rules consist of one or more “clauses” logically ANDed with each other. If an error message satisfies all the clauses in a rule, then that message will be hidden. Each “clause” compares an object to a string via a comparison function.

Objects include:

- error message text
- source or sink routine name
- source or sink file name
- line numbers

Comparison functions include:

- is
- is not
- starts with
- ends with
- does not start with
- does not end with
- contains
- does not contain
- equals
- does not equal
- is greater than
- is less than

Some examples of rules you can create are:

- Don't show an error if it is in file `notMyFile.f`
- Only show errors that are in file `currentTask.c`
- Don't show errors that refer to (contain) variable `notMyProblem`
- Don't show errors of a particular type (for example, a message that contains the string "inconsistent size")
- Don't show errors from lines 200 through 350 of file `worksOk.c` of type "READ->WRITE".

Rules can be deactivated and reactivated via a checkmark in the "Active" column of the rules display.

The **Reorder** menu allows you to sort the errors within each program construct. Errors can be sorted by error message text, symbol name, or subroutine name. The **Options** menu lets you control several aspects of the GUI operation and appearance. Please see the **Help** menu for a detailed explanation of these options.

AssureView Options

The command line options listed below are recognized by AssureView. Each option should be preceded by the "**-**" character (Windows users can also use the "/" character).

-? or -h

Display a summary of AssureView command line options and invocation methods.

-agi=<file>

The **-agi** option specifies the name of the AssureView text file, which was produced by a previous AssureView GUI invocation, to be used as input to AssureView in place of project and simulation output files.

-[no]gui

The **-nogui** option is used to process a `.prj` file and a `.kdd` file to create an `.agi` file but without viewing the `.agi` results with the GUI. The results in the `.agi` file can then be viewed later with AssureView (the `.prj` and `.kdd` files are no longer needed; use the AssureView **-agi=** option to invoke the GUI). The **-gui** option specifies that results should be displayed by using the AssureView GUI. The default is **-gui**.

-prefix=<remove>:<add>

The paths to the source files processed by Assure are known to AssureView and are displayed in the output. In some circumstances, such as when running Assure and AssureView on different machines, or when using networked filesystems, it may be necessary to modify this path information in order to allow AssureView to reach the source files. The **-prefix** option stipulates that the **<remove>** string, if specified, is to be deleted from the head of the path names displayed in the AssureView output,

and then that the **<add>** string, if specified, is to be prepended to the (resulting) path names. This mechanism provides a way to remove, add, or edit path information. Either **<remove>** or **<add>** can be omitted.

-project_name=<file>
-prj=<file>

This option specifies the name of the project file to be used as input to AssureView (see "Using AssureView" on page 92). If no such option is specified, the first file specified on the command line is used as the project file (a `.prj` extension is appended if the filename does not already have this extension). If no project file is specified, the default project filename `assure.prj` is used.

-run_data=<file>
-kdd=<file>

This option specifies the name of the simulation output file to be used as input to AssureView. If no such option is specified, the second file specified on the command line is used as the simulation output file (a `.kdd` extension is appended if the filename does not already have this extension). If no simulation output file is specified, a default filename based on the project filename is used.

-[no]suppress

Certain messages are normally not displayed by AssureView because they typically do not cause errors during parallel execution; the **-nosuppress** setting causes these messages to be displayed. The messages fall into several categories:

- Properly synchronized, unordered variable references that would have caused storage conflicts had they not been synchronized. While these references are not errors, not employing `ORDERED` synchronization might cause the results of parallel runs to differ from those of serial runs because of varying roundoff behavior.
- Properly synchronized, unordered I/O operations inside of parallel constructs. While these references are not errors, not employing `ORDERED` synchronization might cause the I/O behavior of parallel runs to differ from that of serial runs.

- Variable references for local reductions that would otherwise cause errors. In most cases, these messages are due to reductions that have been coded by using the `REDUCTION` clause of a C/C++ `PARALLEL FOR` or Fortran `PARALLEL DO` or by using local reduction variables and correctly synchronized updates of a global result variable.

The default is **-suppress**.

-txt

The **-txt** option specifies that results should be displayed as text on the standard output.

-WJ,[java_option]

The AssureView GUI is implemented in JAVA. The **-WJ** flag prefixes any specified JAVA option. The JAVA options are passed to the JAVA interpreter. Any valid JAVA interpreter option may be used. However, the options listed below may be particularly beneficial when used with AssureView to enhance the performance of the GUI.

JAVA Options

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

-ms<integer>[{k,m}]

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

-mx<integer>[{k,m}]

The **-mx** option specifies the maximum heap size the interpreter will use for dynamically allocated objects. The maximum heap size is specified either in bytes, kilobytes (with the suffix `k`), or megabytes (with the suffix `m`). For example, to specify twenty megabytes, use **-mx20m**.

You should use this option to increase the heap size if you receive “Out of Memory” messages when running AssureView.

-nojit
-Djava.compiler=none

The **-nojit** or **-Djava.compiler=none** option disables the Java just-in-time compiler. This Java feature can sometimes lead to incorrect Java behavior. Use **-WJ**,**-nojit** or **-WJ**,**-Djava.compiler=none** to disable the just-in-time compiler if you experience problems with the either the Assure View or GuideView GUI.

CHAPTER 9

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 and External Routines," beginning on page 21. To embed the PerView server in your application, add the **-WGperview** flag when linking with the Guide driver. For example, to build a PerView-enabled Fortran application on Windows issue the following commands (`guidef` can be replaced by `guidec` for a C application):

```
guidef -compile_only main.for
guidef -WGperview main.obj
```

You may need to add other libraries when linking manually, since PerView requires network functions often not included in the standard C library. To see the libraries required on your system, inspect the output of the following command:

```
guidef -compile_only main.for
link main.obj guide_perview.lib kweb.lib wsock32.lib
```

PerView Environment Variables

Several environment variables influence the behavior of PerView; they are listed below:

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>/`”. Use the following URL instead if a password is specified using `KMP_HTTP_ACCESS`:
“`http://<host>:<port>/cgi-pwd/<password>/`”. You may need to use the full machine name for `<host>`.

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.

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.

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

Unix syntax:

```
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
```

Windows syntax:

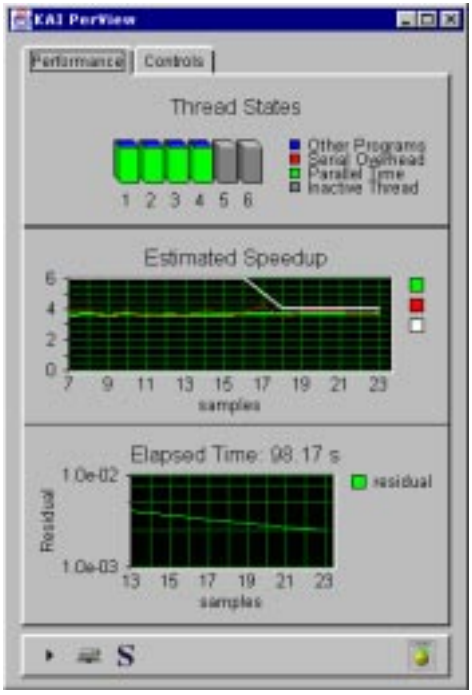
```
server C: guidef /exe:mondo.exe mondo.f /WGperview
server C: set KMP_HTTP_PORT=8000
server C: set KMP_HTTP_ACCESS=secret
server C: mondo.exe

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 9-1. The PerView interface consists of two “views” of displays and controls, selectable by the tabs labeled Performance and Controls.

Figure 9-1



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 108 for details.

Controls

Using the Controls panel shown in Figure 9-2 you can modify the parallel behavior of the application to respond to changing conditions on the machine where it is running.

Figure 9-2

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 9-3. The icons in the status bar summarize the state of the application and PerView's connection to it.

Figure 9-3

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, whether 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 9-4, suitable for general performance monitoring.

Figure 9-4

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 `kwebc_set_meter` (C/C++) or `kweb_set_meter` (Fortran) library routine. The interface to this routine is:

```
   C/C++ syntax:
   void kwebc_set_meter(char* meter_name, int icurrent,
int istart, int iend);

   Fortran syntax:
   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 `kwebc_set_residual` (C/C++) or `kweb_set_residual` (Fortran) library routine. The interface to this routine is:

```
   C/C++ syntax:
   void kwebc_set_residual(char* meter_name, int
current, int ymin, int ymax);

   Fortran syntax:
   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 `kwebc_set_string` (C/C++) or `kweb_set_string` (Fortran) library routine. The interface to this routine is:

```
   C/C++ syntax:
   void kwebc_set_string(char* meter_name, char*
current_phase);

   Fortran syntax:
   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 *kappro-support@kai.com* for more information.

APPENDIX A

OpenMP Directives

Introduction

The KAP/Pro Toolset uses OpenMP directives to support a single level of parallelism. Each directive begins with `*$omp`, `c$omp`, or `!$omp` in Fortran and `#pragma omp` in C/C++. The Fortran directives are not case-sensitive. The `!$omp` sentinel can be used in either free or fixed Fortran source, whereas the other sentinels are only allowed in fixed source mode. For the sake of clarity we will use the `!$omp` form in examples and when describing the syntax. When a Fortran directive is continued on subsequent lines, each additional line begins with `!$omp&`; continuation in C/C++ is accomplished by using the standard backslash at the end of a line. Comments may be appended to the end of Fortran directive lines by using a "`!`" character; otherwise OpenMP directives and clauses can not be interleaved with comments or executable code. Several Fortran directives must be paired (*directive* and *end directive*); in some cases the *end directive* statement is optional. In this manual, items that are optional are enclosed in square brackets (`[]`).

The syntax of the OpenMP directives accepted by the KAP/Pro Toolset is presented below. These directives are a superset of the OpenMP C/C++ Specification version 1.0 and OpenMP Fortran specification version 2.0. More information is available at the OpenMP website "http://www.openmp.org".

Many of the directives in this chapter include a reference to a <structured-block> in their description. A structured block is a sequence of statements that has a single entry point and a single exit point. No sequence is a structured block if there is a jump into or out of that sequence (including a call to longjmp() or a use of throw; however a call to exit is permitted). As another example, Fortran 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. A compound statement is a structured block in C/C++ if its execution always begins at the opening curly brace and always ends at the closing curly brace. An expression statement, selection statement, or iteration statement is a structured block if the corresponding statement obtained by enclosing it in curly braces would be a structured block. For example, jump statements and labeled statements are not structured blocks.

Parallel Directive

parallel

The parallel directive defines a parallel region.

C/C++ syntax:

```
#pragma omp parallel [ <clause> [ <clause> ] ... ]
  <structured-block>
```

where <clause> is one of the following:

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

Fortran syntax:

```
!$!$omp parallel [ <clause> [[,] <clause> ] ... ]
  <structured-block>
!$omp end parallel
```

where <clause> is one of the following:

```
if (<scalar-logical-expression>)
```

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

When the logical `if` clause exists, the `<scalar-logical-expression>` is evaluated at run time. If the logical expression evaluates to false (`0` in C/C++, `.false.` in Fortran) then all of the code in the parallel region is executed by a team of one thread. If the logical expression evaluates to true (`non-zero` in C/C++, `.true.` in Fortran) then the code in the parallel region may be executed by a team of multiple threads.

When the `num_threads` clause exists, the `<scalar-integer-expression>` is evaluated at run time, and a team of the specified number of threads is created to execute the code in the parallel region.

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 133.

Work within a parallel region is divided up among the threads by means of worksharing directives.

Worksharing Directives

for (C/C++) and **do** (Fortran)

The C/C++ `for` pragma and Fortran `do` directive state that the next statement is an iterative loop which will be executed using multiple threads. If the 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 by the thread that encounters this construct.

C/C++ syntax:

```
#pragma omp for [ <clause> [ <clause> ] ... ]
  <for-loop>
```

where <clause> is one of the following:

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

and the <for-loop> header is restricted to have the following form:

```
for (<var> = <lb>; <var> <logic-op> <ub>; <incr-expr>)
```

where <var> is a signed integer variable that must not be modified in the body of the for loop;

<logic-op> is one of <, <=, >, or >= ; and

<incr-expr> is one of the following:

```
++<var>
<var>++
--<var>
<var>--
<var> += <incr>
<var> -= <incr>
<var> = <var> + <incr>
<var> = <incr> + <var>
<var> = <var> - <incr>
```

<lb>, <ub>, and <incr> are loop invariant integer expressions for lower bound, upper bound, and loop increment, respectively. Any side effects from these expressions may produce indeterminate results.

Fortran syntax:

```
!$omp do [ <clause> [[,] <clause> ] ... ]
  <do-loop>
[ !$omp end do [ nowait ] ]
```

where <clause> is one of the following:

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

A few words are in order regarding the `end do` directive in Fortran. The `end do` 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 within the parallel region 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 139. The `ordered` clause is described on page 129.

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 in Fortran or enclosed by the `section` pragma in C/C++. 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 enclosed `section` structured blocks are executed sequentially on the thread that encounters this construct.

C/C++ syntax:

```
#pragma omp sections [ <clause> [ <clause> ] ... ]
{
[ #pragma omp section ]
    <structured-block>
[ #pragma omp section
    <structured-block>
  .
  .
  . ]
}
```

or,

```
#pragma omp sections [ <clause> [ <clause> ] ... ]
  <structured-block>
```

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

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

```
nowait
```

Fortran syntax:

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

where <clause> is one of the following:

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

The ordered clause on OpenMP sections is a KAP/Pro Toolset extension and is described on page 129.

single

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

C/C++ syntax:

```
#pragma omp single [ <clause> [ <clause> ] ... ]
  <structured-block>
```

where <clause> is one of the following:

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

Fortran syntax:

```
!$omp single [ <clause> [[,] <clause> ] ... ]
  <structured-block>
!$omp end single [ <end-single-modifier> ]
```

where <clause> is one of the following:

```
private (<list>)
```

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

and <end-single-modifier> is one of the following (but not both):

```
nowait
copyprivate (<list>) [[,] copyprivate(<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 all continue executing with the statement after the single block. If the nowait clause is present, threads not executing the <structured-block> do not wait, but instead immediately begin execution of the statement following the construct.

The copyprivate clause copies listed private values computed within the single construct to all other threads. It is an alternative to using a shared variable for the value, or pointer association, and is useful when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level). The names of any common blocks appearing in <list> must have their names enclosed in slashes. Any variables appearing <list> in must not appear in a PRIVATE, FIRSTPRIVATE or LASTPRIVATE clause for the SINGLE construct.

The lastprivate and reduction clauses on OpenMP single are KAP/Pro Toolset extensions.

Workqueuing Pragmas in C/C++

While the OpenMP worksharing constructs (for, sections, single) are useful for single loops and statically defined parallel sections, they cannot easily handle the more general cases of recursive and list structured data and complicated control structures. The KAP/Pro Toolset addresses this limitation by introducing the concept of *workqueuing*. The workqueuing constructs are only available to C/C++ codes.

Workqueuing is a new construct type that supplements the existing OpenMP construct types (parallel, worksharing, and synchronization). Workqueuing constructs

are similar to worksharing constructs but are distinguished by the following features:

- Workqueuing constructs may be nested inside one other. (But they may not be nested inside worksharing constructs and vice versa.)
- Re-privatization of variables is allowed at workqueuing constructs. That is, variables made private at the dynamically enclosing `parallel` pragma can also be made private to a `taskq` and/or `task`.

The `taskq` and `task` pragmas are very similar to the `sections` and `section` pragmas but offer more flexibility:

- A `task` pragma may be placed anywhere lexically inside a `taskq` region. The `task` pragma cannot be orphaned
- The number of active `tasks` will be determined at runtime depending on the placement of pragmas inside a `taskq` region. For example, a `task` can occur inside a loop contained in a `taskq` region
- `taskq` pragmas can be recursively nested, allowing for parallelism in multi-dimensional loops, across linked lists, over tree-based data, *etc*.

The Taskq Model in C/C++

taskq

The workqueuing model centers on the concept of a task queue (`taskq`). A `taskq` contains `task`s that can be executed concurrently. A `taskq` can also contain another `taskq` to enable multi-level parallelism.

```
#pragma omp taskq [ <clause> [ <clause> ] ... ]
  <structured-block>
```

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

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

When a team of OpenMP threads encounters a `taskq` pragma, the behavior is as if a single thread first creates an empty queue and then executes the structured

block that follows. When the controlling thread encounters a `task` pragma inside the `taskq` block, the work in the `task` block is enqueued. Any available thread within the `taskq` pool can dequeue and execute tasks from the queue.

A `taskq` pragma is legal when a team of threads is executing redundant code in a `parallel` construct or a single thread is executing a `task` or `taskq` construct. In either case, the code in a `taskq` construct is always executed in single-threaded fashion. The enqueued tasks are themselves executed concurrently among available threads.

No worker thread may exit a `taskq` construct until the thread executing the `taskq` construct exits. Likewise, the thread executing the `taskq` construct cannot exit until all enqueued tasks are complete. When the `nowait` clause is present on a `taskq` construct, however, a thread may exit the `taskq` construct, once all the enclosed tasks, including those recursively queued, have been dequeued.

When a thread is already inside a `taskq` or `task` construct and encounters a `taskq` pragma, it forms another queue and executes the `taskq` construct to insert work in the new queue.

Tasks may contain `ordered` sections, provided the enclosing `taskq` contains an `ordered` clause. The ordered sections of code are executed in the same order the tasks were enqueued.

It should be noted that transfer of execution of the `taskq` block between threads is allowed. Thus, it is recommended that the use of data indexed by `omp_get_thread_num()` should be avoided.

task

```
#pragma omp task [ <clause> [ <clause> ] ... ]
  <structured-block>
```

where `<clause>` may be:

```
private (<list>)
```

A `task` pragma must be lexically enclosed within the structured block following a `taskq` pragma. The `task` pragma is said to bind to the lexically enclosing `taskq`.

When a thread encounters a `task` pragma, the work in the block following the `task` pragma is enqueued on the queue associated with the binding `taskq`. Any thread, including that which enqueued the work, can dequeue and execute this work.

Data Privatization in Workqueues

Like OpenMP worksharing constructs, `taskq` and `task` constructs can classify variables as private. An important distinction, however, is that such variables become private to the task queue and task, respectively, rather than to a thread.

Variables are privatized at a `taskq` via the `private()`, `firstprivate()`, and `lastprivate()` clauses. Variables classified as `private` are uninitialized upon entry to the `taskq` block. Variables classified as `firstprivate` are initialized from the same-named variable in the enclosing context. The values of `lastprivate` variables are copied from the final values in the last enqueued `task` to the same-named variables in the enclosing context.

When a task is enqueued, it receives a "snapshot" of the current state of all variables private to the `taskq`. In addition, variables can be privatized at the `task` itself. Private variables of this type provide uninitialized private storage to each `task`.

The following example illustrates use of the data privatization rules (the `ordered` clause enforces correct order for the `printf` output):

C/C++ syntax:

```
#include <omp.h>

main() {
  int me, i, temp, out, three=3, four=4, five=5;
  #pragma omp parallel private(me)
  {
    me = omp_get_thread_num();
    #pragma omp taskq private(i,four) firstprivate(five) \
       lastprivate(out) ordered
    {
      printf("1: me=%d\n", me);
      for(i = 0; i < 3; i++) {
        #pragma omp task private(temp)
        {
          temp = i*2;
          out = temp*2;
          #pragma omp ordered
          printf("2: me=%d i=%d three=%d four=%d five=%d\n",\
                 me, i, three, four, five);
        }
      }
    }
    #pragma omp single
    printf("3: out=%d temp=%d\n", out, temp);
  }
}
```

The output of this program is:

```
1: me=0
2: me=2 i=0 three=3 four=0 five=5
2: me=1 i=1 three=3 four=0 five=5
2: me=3 i=2 three=3 four=0 five=5
3: out=8 temp=536877680
```

Line "`1:`" is executed by only one thread, in this case thread zero. The output of this is indeterminate, since any thread can execute the `taskq`. Lines "`2:`" show the correct values of `me`, since data made private at a parallel pragma remains private to each thread. The variable `i` has the same value as when the `task` was enqueued, because it is private to the `taskq`. The variable `three` is correct, because shared variables remain visible to tasks. The value of `four` is undefined but uniform across tasks, since it is private to the `taskq` but was not initialized in the `taskq` region. The value of `five` is correct, since it was privatized with a `firstprivate` clause. In line "`3:`", the value of `out` is obtained from the last task enqueued, in which `i==2`. The value of `temp` is undefined, since it was assigned only inside the tasks, where it was private.

Examples

Appendix B, “C/C++ Examples” on page 155 includes `taskq` examples. These may serve to clarify the workqueuing model and illustrate its possible uses.

workshare (Fortran only)

The `workshare` directive divides the work of executing the enclosed code into separate units of work, and causes the threads of the team to share the work of executing the enclosed code. The units of work may be assigned to threads in any manner as long as each unit of work gets executed exactly once.

```
   Fortran syntax:
!$omp workshare
  <structured-block>
!$omp end workshare [nowait]
```

The primary use of a `workshare` construct is to parallelize Fortran90 array expressions, including transformational array intrinsic functions that compute scalar values from arrays. Evaluation of each array element of the array expression is a unit of work. Individual assignments, including `atomic` and `critical` constructs, are also units of work. Please see the latest OpenMP Fortran Specification at http://www.openmp.org for specific details on how a unit of work is defined.

Without the `nowait` clause, all threads that reach the end of the `workshare` directive will wait until all units of work have been completed.

Combined Parallel Worksharing and Workqueuing Directives

parallel for (C/C++) and **parallel do** (Fortran)

The C/C++ `parallel for` and Fortran `parallel do` directives are a short form syntax for a parallel region enclosing a single `for` or `do`. The combined directive is used in place of the two lines taken by a `parallel` directive followed immediately by the worksharing directive. If this directive is encountered

while a parallel region is already active the directive is executed by a team of one thread and the entire loop is executed by each thread of the enclosing parallel region that encounters it.

C/C++ syntax:

```
#pragma omp parallel for [ <clause> [ <clause> ] ... ]
  <for-loop>
```

where <clause> is one of the following:

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

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

```
#pragma omp parallel [ <par-clause> \
  [ <par-clause> ] ... ]
{
  #pragma omp for nowait [ <for-clause> \
    [ <for-clause> ] ... ]
    <for-loop>
}
```

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

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

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

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

Fortran syntax:

```
!$omp parallel do [ <clause> [[,] <clause> ] ... ]
```

```
  <do-loop>
[ !$omp end parallel do ]
```

where <clause> is one of the following:

```
if (<scalar-logical-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
num_threads(<scalar-integer-expression>)
```

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

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

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

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

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 the directive is executed by a team of one thread and the entire construct is executed by each thread from the enclosing parallel region that encounters it.

C/C++ syntax:

```
#pragma omp parallel sections [ <clause> \
  [ <clause> ] ... ]
{
[ #pragma omp section ]
    <structured-block>
[ #pragma omp section
    <structured-block>
  .
  .
  . ]
}
```

or,

```
#pragma omp parallel sections [ <clause> \
  [ <clause> ] ... ]
  <structured-block>
```

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

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

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

```
#pragma omp parallel [ <par-clause> [ \
  <par-clause> ] ... ]
{
  #pragma omp sections nowait [ <sec-clause> \
    [ <sec-clause> ] ... ]
    {
[     #pragma omp section ]
        <structured-block>
  [   #pragma omp section
```

```
        <structured-block>
    .
    . ]
  }
}
```

or,

```
#pragma omp parallel [ <par-clause> \
  [ <par-clause> ] ... ]
{
  #pragma omp sections nowait [ <sec-clause> \
  [ <sec-clause> ] ... ]
    <structured-block>
}
```

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

```
if (<scalar-logical-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>)
ordered
```

Fortran syntax:

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

where <clause> is one of the following:

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

```
num_threads(<scalar-integer-expression>)
```

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

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

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

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

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

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

parallel taskq (C/C++ only)

C/C++ syntax:

```
#pragma omp parallel taskq [ <clause> \
  [ <clause> ] ... ]
  <structured-block>
```

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

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

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

```
#pragma omp parallel [ <par-clause> \
  [ <par-clause> ] ... ]
{
  #pragma omp taskq nowait [ <taskq-clause> \
    [ <taskq-clause> ] ... ]
      <structured-block>
}
```

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

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

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

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

parallel workshare (Fortran only)

Fortran syntax:

```
!$omp parallel workshare [ <clause> [[,] <clause>] ... ]
  <structured-block>
!$omp end parallel workshare
```

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

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

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

```
!$omp parallel [ <par-clause> [[,] <par-clause> ] ... ]
!$omp workshare
  <structured-block>
```

```
!$omp end workshare nowait
!$omp end parallel
```

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

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

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/C++ syntax:

```
#pragma omp critical [ (<name>) ]
  <structured-block>
```

Fortran syntax:

```
!$omp critical [ (<name>) ]
  <structured-block>
!$omp end critical [ (<name>) ]
```

The name has global scope. Two or more `critical` directives with the same name are automatically mutually exclusive. That is, while a thread is executing within critical section *foo*, no other thread will be allowed to enter any other critical section named *foo* no matter where it may reside within the code. 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/C++ syntax:

```
#pragma omp ordered
```

```
  <structured-block>
```

Fortran syntax:

```
!$omp ordered [ (<name>) ]
  <structured-block>
!$omp end ordered [ (<name>) ]
```

An optional name can be given to an ordered section in Fortran, but not in C/C++. Named ordered sections in Fortran 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 `sections`, (KAP/Pro Toolset extension) C/C++ `for`, or Fortran `do` or `taskq` construct. The enclosing construct must have the `ordered` clause attached to its definition. It is an error to use this directive within a construct without 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, one thread at a time, to enter the ordered section. The ordered sections are executed in the same order as the `for` or `do` iterations, `sections`, or `task`s would be executed in the sequential version of the code.

In general, an iteration of a loop with a `for` or `do` directive must not execute the same ordered section more than once and must not execute more than one ordered section. With the KAP/Pro Toolset extension of named ordered sections in Fortran, the above restriction applies to ordered sections of the same name. That is, parallel `do` loop iterations may execute more than one ordered section if the ordered sections do not have the same name. This constraint applies to named ordered sections within the scope of `sections` directives. As with named critical sections, all unnamed ordered sections are assumed to share the same name.

In other words, each ordered section with a given name must only be entered once or not at all during the execution of each `for` or `do` iteration, `section`, or `task`. Only one ordered section with a given name may be encountered during the execution of each `for` or `do` iteration, or `section`, or `task`.

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/C++ syntax:

```
#pragma omp master
  <structured-block>
```

Fortran syntax:

```
!$omp master
  <structured-block>
!$omp end master
```

Other threads of the team skip the section of code and continue execution. 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/C++ syntax:

```
#pragma omp atomic
  <expression-statement>
```

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

```
x <binary-op> = <expr>;
x++;
++x;
x--;
--x;
```

and where

`x` is an *lvalue* expression with scalar type and without side effects.

`<expr>` is a scalar expression without side effects that does not reference `x`.

`<binary-op>` is one of `+`, `-`, `*`, `/`, `&`, `^`, `|`, `<<`, or `>>`.

Fortran syntax:

```
!$omp atomic
  <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 future atomic updates to that object must have a compatible type.

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/C++ syntax:

```
#pragma omp flush [ (<list>) ]
```

Fortran syntax:

```
!$omp flush [ (<list>) ]
```

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. A `flush` is implied on the `parallel`, `end parallel`, `barrier`, `critical`, and `end critical` OpenMP constructs.

barrier

The `barrier` directive gathers all team members to a particular point in the code.

C/C++ syntax:

```
#pragma omp barrier
```

Fortran syntax:

```
!$omp barrier
```

Barriers force threads within a team to wait at that point in the code until all of the team members encounter that barrier. The `barrier` directive is not allowed inside of worksharing constructs, workqueuing 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 threads that execute the parallel code. When distinct threads should reference the same variable, place the variable in the `shared` list. When distinct threads should reference distinct instances of a variable, place the variable in the `private` list.

The `private` clause is allowed on `parallel`, `sections`, `single`, Fortran `do`, C/C++ `for, taskq,` and `task` directives. Reprivatizion of variables is now allowed in Fortran. That is, variables made private or shared at a parallel region may be declared `private` for an enclosed worksharing construct. The `default` and `shared` clauses are only allowed on `parallel` directives. Variables on the `private` list are uninitialized upon entering a parallel region; see also the following description of `firstprivate` .

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. Scalar variables, pointers, and arrays (including deferred shape and assumed size arrays) can be privatized.

The only exceptions to the `default` rules in Fortran are loop control variables (loop indices) and Fortran 90 statement-scoped entities, which are `private` unless explicitly overridden. The only exceptions to the `default()` rules in

C/C++ are loop control variables (loop indices) within `for` pragmas, threadprivate variables, and const-qualified variables. The first two are private, and the latter is shared, unless explicitly overridden. The default for both C/C++ and Fortran is `default(shared)`.

Note that `default(private)` in C/C++ is a KAP/Pro Toolset extension to OpenMP.

firstprivate (<list>)

A variable included in `<list>` has the semantics of a `private` data scope. Before execution of the affiliated construct, the value from the variable of the same name in the enclosing context is copied into the private counterpart of each thread.

The `firstprivate` clause is allowed on `parallel`, `sections`, `single`, C/C++ `for`, `taskq`, and Fortran `do` directives.

lastprivate (<list>)

A variable included in `<list>` has the semantics of a `private` data scope. Upon completion of the affiliated construct, the value of the variable in the enclosing context is assigned the value of the corresponding private copy held by the thread that executes the last dynamically encountered `task` of a `taskq` construct in C/C++, the final iteration of the index set for a C/C++ `for` or Fortran `do` loop, the last lexical `section` of a `sections` construct, or the code enclosed by a `single`, as appropriate. If the `lastprivate` variable is not updated within the sequentially final iteration, `section`, `task`, or `single` code, the value of the original variable following the completed construct will be undefined.

The `lastprivate` clause is allowed on `sections`, `single`, C/C++ `for`, `taskq` and Fortran `do` directives. The use of the `lastprivate` clause on a `single` or `taskq` is a KAP/Pro Toolset extension.

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

A variable, array element, or array in the `reduction` list creates a `private` temporary for each thread. Deferred shape and assumed size arrays are not allowed on the `reduction` clause. Upon completion of the affiliated con-

struct, the value of the original variable is updated by combining the values held in the temporary variables with the given associative operator or intrinsic function. The allowed C/C++ operators are +, -, *, &, ^, |, &&, and ||. The allowed Fortran operators are +, -, *, .AND., .OR., .EQV., and .NEQV. The allowed Fortran intrinsics are MAX, MIN, IAND, IOR, and IEOR.

The reduction clause is allowed on parallel, sections, single, C/C++ for, taskq and Fortran do directives. The use of the reduction clause on a single or taskq is a KAP/Pro Toolset extension.

C/C++ syntax:

```
#pragma omp parallel for shared(a,t,n) \
    private(i) reduction(+:sum) \
    reduction(&&:truth)

    for(i=0; i < n; i++) {
        sum += a[i];
        truth = truth && t[i];
    }
```

The above C/C++ example is equivalent to the following:

C/C++ syntax:

```
#pragma omp parallel shared(a,t,n) private(i)
{
    int sum_local = 0;
    int truth_local = 1;

    #pragma omp for nowait
    for(i=0; i < n; i++) {
        sum_local += a[i];
        truth_local = truth_local && t[i];
    }

    #pragma omp critical
    {
        sum += sum_local;
        truth = truth && truth_local;
    }
}
```

Fortran syntax:

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

The above Fortran example is equivalent to the following:

Fortran syntax:

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

copyin (<list>)

The `copyin()` clause applies only to `threadprivate` variables in C/C++ and to `threadprivate` variables, `COMMON` blocks and use-associated variables in Fortran. The `<list>` can contain individual variables or entire `threadprivate` `COMMON` blocks; names of common blocks must be surrounded by slashes. 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 and combined parallel worksharing and workqueuing directives.

The `threadprivate` clause is described in the next two sections.

Privatization of Fortran Variables, Common Blocks and Use-Associated Variables

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 the same thread in a parallel region. This notion is supported by the `!$omp threadprivate` directive.

threadprivate

The `!$omp threadprivate` directive in Fortran creates thread-private copies of one or more variables or `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 can be used as a migration feature for SGI's `copyin` directive. A `threadprivate` variable or `COMMON` block is always private in each parallel region of each routine where the `threadprivate` directive appears.

Declaring Private Variables or Commons

Private variables or `COMMON` blocks in Fortran are declared by the `threadprivate` directive. The syntax for the directive is as follows:

Fortran syntax:

```
!$omp threadprivate (<list>)
```

This directive is placed in the declaration section of a routine. If a variable or `COMMON` block appears in a `threadprivate` directive in one routine, it must appear in that same directive in all routines where the variable or `COMMON` block is used. Names of `COMMON` blocks in `<list>` must be surrounded by slashes.

The `threadprivate` directive assigns each specified variable or `COMMON` block to the master thread and creates an uninitialized private copy for each additional thread. Updated values to `threadprivate` variables will not be seen by other threads. The `copyin` clause can be used to initialize a `threadprivate` `COMMON` block from the master copy. Thread–private copies for `threadprivate` `COMMON` blocks in Fortran are always allocated, implicitly, at each parallel region.

If a common block or a variable that is declared in the scope of a module appears in a `threadprivate` directive, it implicitly has the `SAVE` attribute. A variable that appears in a `threadprivate` directive and is not declared in the scope of a module must have the `SAVE` attribute.

Privatization of Global Variables in C/C++

OpenMP provides privatization of file-scope and namespace-scope variables in C/C++ via the `threadprivate` pragma. Threadprivate variables become private

to each thread but retain their file-scope or namespace-scope visibility within each thread.

The syntax of the `threadprivate` pragma is:

C/C++ syntax:

```
#pragma omp threadprivate(<list>)
```

where *list* is a comma-separated list of one or more file-scope or namespace-scope variables. The `threadprivate` pragma must follow the declaration of the listed variables and appear in the same scope. The following example is legal:

C/C++ syntax

```
extern int x;
#pragma omp threadprivate(x)

namespace foo {
    int me;
    #pragma omp threadprivate( me )
};

main() {
}
```

while the following is illegal, since the variable `x` is neither file-scope nor namespace-scope:

C/C++ syntax:

```
main() {
    extern int x;
    #pragma omp threadprivate(x)
}
```

As an extension to OpenMP, KAP/Pro allows the use of the `threadprivate` pragma with local static variables in C. The following, for example, is legal:

C/C++ syntax:

```
main() {
    int x;
    {
        static int y;
        #pragma omp threadprivate(y)
    }
}
```

Initializing Threadprivate Variables

When a team consists of *t* threads, there are exactly *t* copies of each `threadprivate` variable. The master thread uses the global copy of each variable as its private copy. Each `threadprivate` variable is initialized once before its first use. If an explicit initializer is present, then each thread's copy is suitably initialized. If no explicit initializer is present, then each thread's copy is zero-initialized.

`threadprivate` variables can also be initialized upon entry to a parallel region via the `copyin` clause on the `parallel` pragma. When this clause is present, each thread's copy of each listed `threadprivate` variable is copied, as if by assignment, from the master's copy upon each entry to the parallel region. The `copyin` is executed each time the associated parallel region executes.

Persistence of Threadprivate Variables

After the first parallel region executes, the data in the `threadprivate` variables are guaranteed to persist only if the dynamic threads mechanism is disabled. Dynamic threading is disabled by default, but can be enabled via the `OMP_DYNAMIC` environment variable and the `omp_set_dynamic()` library call.

Scheduling Options

Scheduling options are used to specify the iteration dispatch mechanism for each parallel loop (C/C++ `for` or Fortran `do`) construct. They can be specified in the following three ways

1. OpenMP Directives
2. Environment Variables
3. Command Line Switches

If a parallel loop has a `schedule` clause (except for `runtime`), then the directive specifies the loop scheduling. If a parallel loop has a `schedule (runtime)` clause, described below, then the environment variable OMP_SCHEDULE specifies the loop scheduling at run time. Guide command line switches are used to specify the default scheduling mechanism for parallel loops with no `schedule` clause. In the absence of OpenMP directives, environment variables, and command line switches the default loop scheduling mechanism is `static`.

Scheduling Options Using OpenMP Directives

The list below shows the syntax for specifying scheduling options with the C/C++ `for`, `parallel for` and Fortran `do`, `parallel do` directives.

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

The list below shows the syntax for specifying scheduling options that are only available with the Fortran `do` and `parallel do` directives.

```
schedule (trapezoidal [,<integer>])
schedule (interleaved)
```

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 9-5 on page 142 for a complete description of these 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`, `dynamic`, or `guided`, (`trapezoidal`, `interleaved` in Fortran only) and the optional `<integer>` parameter is a chunk size for the dispatch method.

See Table 9-5 on page 142 for a complete description of these scheduling options.

Scheduling Options using Command Line Switches

The following command line switches affect the scheduling of loops without an explicit `schedule` clause. These options are available in both C/C++ and Fortran.

```
-WGsched=dynamic or -WGsched=d
-WGsched=guided or -WGsched=g
-WGsched=runtime or -WGsched=r
-WGsched=static or -WGsched=s
```

The KAP/Pro Toolset also allows several command line scheduling mechanisms that are extensions to the OpenMP standard; the following are available in Fortran only:

```
-WGsched=even or -WGsched=e
-WGsched=interleaved or -WGsched=i
-WGsched=trapezoidal or -WGsched=t
```

Assure accepts all the scheduling methods present in OpenMP directives that Guide supports. However, the only scheduling method which currently affects the operation of Assure is the `ordered` clause. For this reason, command line options and environment variables for scheduling are not supported in Assure.

These scheduling options are fully described in Table 9-5 on page 142.

Scheduling Options Table

The various scheduling options are summarized in the following table. Assume the following: the loop has *l* iterations, *p* threads execute the loop, and *n* is a positive integer specifying the chunk size.

Table 9-5 Scheduling Options

Scheduling Type	Chunk	Meaning

even, e *(Fortran only)*	ignored	Even scheduling. *l/p* iterations are dispatched statically to each thread. The chunk size has no effect here. Even scheduling is the default method of loop scheduling.
		
		To specify even scheduling from the Fortran command line use:
		`-WGsched=even`
		or
		`-WGsched=even,<integer>` [same as -WGsched=even; the chunk size has no effect]
		Alternatively, the word `even` may be abbreviated:
		`-WGsched=e`
		or
		`-WGsched=e,<integer>` [same as -WGsched=e; the chunk size has no effect]
		To specify static even scheduling with the `SCHEDULE` clause use:
		`schedule (static)`
		There is no `even` argument for the `schedule` clause. 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 no `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 Type	Chunk	Meaning

static, s *(C/C++, Fortran)*	n	Static scheduling with a chunk size of *n*. *n* iterations are dispatched statically to each thread (repeat until *l* iterations have been dispatched in a round robin fashion among all threads). If *n* is missing, this is the same as static even scheduling. In C/C++, *l/p* iterations are dispatched statically to each thread so that each thread gets only a single chunk of the iteration space.
		
		To specify static scheduling from the command line use:
		`-WGsched=static,<integer>`
		or
		`-WGsched=static` [specifies even scheduling when **chunk** is not stated]
		
		Alternatively, the word `static` may be abbreviated as follows:
		`-WGsched=s,<integer>`
		or
		`-WGsched=s` [specifies even scheduling when **chunk** is not stated]
		
		To specify static scheduling with the `schedule` clause use:
		`schedule (static[,<integer>])`
		
		To specify static scheduling with the `OMP_SCHEDULE` environment variable use:
		`OMP_SCHEDULE = static[,<integer>]`

Scheduling Type	Chunk	Meaning

inter-leaved, i *(Fortran only)*	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 Fortran command line use:
		 `-WGsched=interleaved`
		 or
		 `-WGsched=interleaved,<integer>` [same as -WGsched=interleaved; the chunk size has no effect]
		 Alternatively the word `interleaved` may be abbreviated as follows:
		 `-WGsched=i`
		 or
		 `-WGsched=i,<integer>` [same as -WGsched=i; the chunk size has no effect]
		 To specify static interleaved scheduling with the `schedule` clause 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 Type	Chunk	Meaning

dynamic, d *(C/C++, Fortran)*	n	Dynamic scheduling with a chunk size of *n*. The first *n*p* iterations are statically assigned to threads in chunks of *n* iterations. Once a thread finishes the assigned set of iterations, a new set of *n* iterations is scheduled. This dynamic scheduling continues until all iterations have been consigned.
		To specify dynamic scheduling from the command line use:
		`-WGsched=dynamic[,<integer>]`
		If no chunk size is specified, a size of 1 will be used.
		Alternatively the word d`ynamic` may be abbreviated as follows:
		`-WGsched=d[,<integer>]`
		If no chunk size is specified, a size of 1 will be used.
		To specify dynamic scheduling with the `schedule` clause 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.

Scheduling Type	Chunk	Meaning

guided, g *(C/C++, Fortran)*	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:
		`-WGsched=guided[,<integer>]`
		If no chunk size is specified, a size of 1 will be used.
		Alternatively, the word `guided` may be abbreviated as follows:
		`-WGsched=g[,<integer>]`
		If no chunk size is specified, a size of 1 will be used.
		To specify guided scheduling with the `schedule` clause 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 Type	Chunk	Meaning

trapezoidal, t *(Fortran only)*	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 Fortran command line use:
		`-WGsched=trapezoidal[,<integer>]`
		If no chunk size is specified, a size of 1 will be used.
		
		Alternatively the word `trapezoidal` may be abbreviated as follows:
		`-WGsched=t[,<integer>]`
		If no chunk size is specified, a size of 1 will be used.
		
		Trapezoidal scheduling is a KAP/Pro Toolset extension to OpenMP.
		
		To specify trapezoidal scheduling with the `schedule` clause 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 Type	Chunk	Meaning

runtime, r *(C/C++, Fortran)*	ignored	Runtime scheduling specifies the scheduling that will be determined via the `OMP_SCHEDULE` environment variable at run time.
		
		To specify scheduling at runtime, use the following from the command line:
		`-WGsched=runtime`
		Alternatively, the word `runtime` may be abbreviated as follows:
		`-WGsched=r`
		To specify runtime scheduling with the `schedule` clause, use:
		`schedule(runtime)`
		To specify runtime scheduling with the `OMP_SCHEDULE` environment variable use:
		`OMP_SCHEDULE = <string>[,<integer>]`
		where `<string>` is one of `static`, `dynamic`, or `guided`, (`trapezoidal, interleaved` in Fortran only) and the optional `<integer>` parameter is the chunk size for the dispatch method.
		
		If the `OMP_SCHEDULE` environment variable is not set, then the default is assumed to be "`dynamic,1`".

OpenMP Environment Variables

Some environment variables may need to be set before running Guide-generated programs. A list and description of each OpenMP environment variable, along with acceptable option values is presented in this section. Additional KAP/Pro environment variables are described in sections "Environment Variables for Guide" and "Environment Variables for Assure" in Chapter 6, "The KAP/Pro Drivers," beginning on page 57

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. The default value of **ALL** specifies that one thread will be created per processor on the machine.

OMP_SCHEDULE=<string>[,<integer>]

The OMP_SCHEDULE environment variable controls the schedule type and chunk size for C/C++ for and Fortran do constructs with a schedule(runtime) clause or those with no schedule clause if the command line scheduling designator is set to r (runtime). The schedule type is given by <string>, which is one of static, dynamic, or guided, (trapezoidal, interleaved in Fortran only) and the optional chunk size is given by <integer> for those scheduling types which allow a chunk size. See “Scheduling Options” on page 139.

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. This environment variable does not affect nested parallelism implemented via nested taskq pragmas within a single parallel pragma in C/C++. Allowed values are TRUE and FALSE, and the default value is **FALSE**.

OpenMP Routines

This section describes the syntax of several OpenMP routines which can be used to manipulate locks, determine the number of threads and/or processors, and determine elapsed wallclock time.

void omp_destroy_lock(omp_lock_t *lock), subroutine omp_destroy_lock(<var>)

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

int omp_get_max_threads(void), integer function 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.

int omp_get_num_procs(void), integer function 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.

int omp_get_num_threads(void), integer function 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. When called outside a parallel region, this function returns `1`.

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 maximum number of threads specified via the

OMP_NUM_THREADS environment variable or the OMP_SET_NUM_THREADS() API call.

int omp_get_thread_num(void), integer function omp_get_thread_num()

This routine returns the 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.

double omp_get_wtime(void), double precision function omp_get_wtime()

This function returns a double precision value equal to the elapsed wallclock time in seconds relative to an arbitrary reference time. The reference time is guaranteed not to change during program execution.

double omp_get_wtick(void), double precision function omp_get_wtick()

This function returns a double precision value equal to the number of seconds between successive clock ticks.

void omp_init_lock(omp_lock_t *lock), subroutine omp_init_lock(<var>)

void omp_init_nest_lock(omp_nest_lock_t *lock), subroutine omp_init_nest_lock(<var>)

These routines initialize a lock associated with the lock variable *lock (using C/C++) or <var> (using Fortran) for use by subsequent calls. The initial state is unlocked. For a nestable lock, the initial nesting count is zero. The lock variable must only be accessed through the OpenMP library lock routines. When using C/C++, the lock parameter must be a pointer to type omp_lock_t or omp_init_nest_lock_t (as defined in the header file omp.h). When using

Fortran, <var> must of integer type and of KIND large enough to hold an address. Two special KIND types, OMP_LOCK_KIND, and OMP_NEST_LOCK_KIND are defined for simple and nested lock variables, respectively.

int omp_in_parallel(void), logical function omp_in_parallel()

The C/C++ function returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel, otherwise it returns zero. The Fortran function returns .TRUE. if it is called within the dynamic extent of a parallel region executing in parallel, otherwise .FALSE. is returned.

void omp_set_lock(omp_lock_t *lock), subroutine omp_set_lock(<var>)

void omp_set_nest_lock(omp_nest_lock_t *lock), subroutine omp_set_nest_lock(<var>)

These routines force 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. The lock parameter *lock (using C/C++) or <var> (using Fortran) must first be initialized by the appropriate OMP_INIT_LOCK() or OMP_INIT_NEST_LOCK() routine.

For a nestable lock, the nesting count is incremented, and the calling thread is granted, or retains, ownership of the lock.

int omp_test_lock(omp_lock_t *lock), logical function omp_test_lock(<var>)

int omp_test_nest_lock(omp_nest_lock_t *lock), logical function omp_test_nest_lock(<var>)

These routines try to obtain ownership of the lock, but do not block execution of the calling thread if the lock is not available. The C/C++ routines return a non-zero value if the lock was successfully obtained, otherwise they return zero. The Fortran routines return .TRUE. if the lock was successfully obtained, otherwise .FALSE. is returned. The lock parameter *lock (using C/C++) or <var> (using Fortran) must be an initialized lock variable.

void omp_unset_lock(omp_lock_t *lock), subroutine omp_unset_lock(<var>)

void omp_unset_nest_lock(omp_nest_lock_t *lock), subroutine omp_unset_nest_lock(<var>)

These routines release the executing thread from ownership of the lock. The behavior is undefined if the executing thread is not the owner of the lock. The lock parameter `*lock` (using C/C++) or `<var>` (using Fortran) must be an initialized lock variable.

void omp_set_num_threads(int), subroutine omp_set_num_threads(<integer>)

This function sets the number of threads to use for subsequent parallel regions. The value of the argument must be positive. Its effect depends upon whether dynamic adjustment of threads is enabled. If dynamic adjustment is disabled, the value is used as the number of threads for all subsequent parallel regions prior to the next call to this function; otherwise, the value is the maximum number of threads that will be used. This function can only be called from serial regions of the code.

void omp_set_dynamic(int), subroutine omp_set_dynamic(<logical>)

This function enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. It has effect only when called from serial regions. If the argument is not `0` (using C/C++) or `.TRUE.` (using Fortran) the number of threads that are used for executing subsequent parallel regions may be adjusted automatically by the run-time environment to best utilize system resources. As a consequence, the number of threads specified by the `OMP_NUM_THREADS` environment variable or `omp_set_num_threads()` function is the maximum thread count. The number of threads always remains fixed over the duration of each parallel region. If the argument is `0` (using C/C++) or `.FALSE.` (using Fortran) dynamic adjustment is disabled.

int omp_get_dynamic(void), logical function omp_get_dynamic()

The C/C++ function returns non-zero if dynamic thread adjustment is enabled and returns `0` otherwise. The Fortran function returns `.TRUE.` if dynamic thread adjustment is enabled and returns `.FALSE.` otherwise.

void omp_set_nested(int), subroutine omp_set_nested(<logical>)

This function enables or disables nested parallelism. Nested parallelism is not implemented in the KAP/Pro Toolset, so this function has no effect. To exploit nested parallelism when using C/C++, please see “Workqueuing Pragmas in C/C++” on page 117.

int omp_get_nested(void), logical function omp_get_nested()

The C/C++ function always returns `0` and the Fortran function always returns `.FALSE.` in the current version of KAP/Pro Toolset.

APPENDIX B C/C++ Examples

Examples of OpenMP usage in C/C++

The following examples show usage of OpenMP examples in C/C++.

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

```
void for_1 (float a[], float b[], int n)
{
    int i, j;

    #pragma omp parallel shared(a,b,n) \
     private(i,j)
    {
        #pragma omp for schedule(dynamic,1) nowait
        for(i = 1; i < n; i++) {
            for(j = 0; j <= i; j++)
               b[j + n*i] = \
               (a[j + n*i] + a[j + n*(i-1)])/2.0;
        }
    }
}
```

B.2 for: Two Difference Operators

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

```
void for_2 (float a[], float b[], float c[], \
float d[], int n, int m)
{
    int i, j;

    #pragma omp parallel shared(a,b,c,d,n,m) \
     private(i,j)
    {
        #pragma omp for schedule(dynamic,1) nowait
        for(i = 1; i < n; i++) {
            for(j = 0; j <= i; j++)
               b[j + n*i] = \
               ( a[j + n*i] + a[j + n*(i-1)] )/2.0;
        }

        #pragma omp for schedule(dynamic,1) nowait
        for(i = 1; i < m; i++) {
            for(j = 0; j <= i; j++)
               d[j + m*i] = \
               ( c[j + m*i] + c[j + m*(i-1)] )/2.0;
        }
    }
}
```

B.3 for: Reduce Fork/Join Overhead

Routines `for_3a` and `for_3b` perform numerically equivalent computations, but because the `parallel` pragma in routine `for_3b` is outside the loop, routine `for_3b` probably forms teams less often, and thus reduces overhead.

```
void for_3a (float a[], float b[], int n, int m)
{
    int i, j;

    for(j = 0; j < m; j++) {
        #pragma omp parallel shared(a,b,n,j) \
         private(i)
        {
            #pragma omp for nowait
            for(i = 0; i < n; i++)
                a[i + n*j] = \
                b[i + n* ] / a[i + n*(j-1)];
        }
    }
}

void for_3b (float a[], float b[], int n, int m)
{
    int i, j;

    #pragma omp parallel shared(a,b,n) \
     private(i,j)
    {
        for(j = 0; j < m; j++) {
            #pragma omp for nowait
            for(i = 0; i < n; i++)
                a[i + n*j] = \
                b[i + n*j] / a[i + n*(j-1)];
        }
    }
}
```

B.4 sections: Two Difference Operators

Identical to "for: Two Difference Operators" on page 157 but uses `sections` instead of `for`. Here the speedup is limited to 2 because there are only 2 units of work whereas in "for: Two Difference Operators" on page 157 there are *n-1 + m-1* units of work.

```
void sections_1 (float a[], float b[], \
     float c[], float d[], int n, int m)
{
    int i, j;

    #pragma omp parallel shared(a,b,c,d,n,m) \
     private(i,j)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for(i = 1; i < n; i++) {
                for(j = 0; j <= i; j++)
                   b[j + n*i] = \
                   (a[j + n*i]+a[j + n*(i-1)])/2.0;
            }

            #pragma omp section
            for(i = 1; i < m; i++) {
                for(j = 0; j <= i; j++)
                   d[j + m*i] = \
                   (c[j + m*i]+c[j + m*(i-1)])/2.0;
            }
        }
    }
}
```

B.5 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`.

```
void single_sp_1a (float a[], float b[], int n)
{
    int i;

    #pragma omp parallel shared(a,b,n) private(i)
    {
        #pragma omp for
        for(i = 0; i < n; i++)
            a[i] = 1.0 / a[i] ;

        #pragma omp single nowait
        a[0] = min( a[0], 1.0 ) ;

        #pragma omp for nowait
        for(i = 0; i < n; i++)
            b[i] = b[i] / b[i] ;
    }
}
```

B.6 sections: Updating a Shared Scalar

Identical to “single: Updating a Shared Scalar” on page 160 but using different pragmas.

```
void sections_sp_1 (float a[], float b[], int n)
{
    int i;

    #pragma omp parallel shared(a,b,n) private(i)
    {
        #pragma omp for
        for(i = 0; i < n; i++)
            a[i] = 1.0 / a[i] ;

        #pragma omp sections nowait
        a[0] = min( a[0], 1.0 ) ;

        #pragma omp for nowait
        for(i = 0; i < n; i++)
            b[i] = b[i] / b[i] ;
    }
}
```

B.7 for: Updating a Shared Scalar

Identical to “single: Updating a Shared Scalar” on page 160 but using different pragmas.

```
void for_sp_1 (float a[], float b[], int n)
{
    int i;

    #pragma omp parallel shared(a,b,n) private(i)
    {
        #pragma omp for
        for(i = 0; i < n; i++)
            a[i] = 1.0 / a[i] ;

        #pragma omp for nowait
        for(i = 0; i < 1; i++)
            a[i] = min( a[i], 1.0 ) ;

        #pragma omp for nowait
        for(i = 0; i < n; i++)
            b[i] = b[i] / b[i] ;
    }
}
```

B.8 parallel for: A Simple Difference Operator

Identical to “for: A Simple Difference Operator” on page 156 but using different pragmas.

```
void parallelfor_1 (float a[], float b[], int n)
{
    int i, j;

    #pragma omp parallel for shared(a,b,n) \
            private(i,j) schedule(dynamic,1)
    for(i = 1; i < n; i++) {
        for(j = 0; j <= i; j++)
           b[j + n*i] = \
           ( a[j + n*i] + a[j + n*(i-1)] ) / 2.0;
    }
}
```

B.9 parallel sections: Two Difference Operators

Identical to “sections: Two Difference Operators” on page 159 but using different pragmas.

```
void sections_2 (float a[], float b[], \
     float c[], float d[], int n, int m)
{
    int i, j;

    #pragma omp parallel sections \
     shared(a,b,c,d,n,m) private(i,j)
    {
        #pragma omp section
        for(i = 1; i < n; i++) {
            for(j = 0; j <= i; j++)
               b[j + n*i] = \
               (a[j + n*i] + a[j + n*(i-1)])/2.0;
        }

        #pragma omp section
        for(i = 1; i < m; i++) {
            for(j = 0; j <= i; j++)
               d[j + m*i] = \
               (c[j + m*i] + c[j + m*(i-1)])/2.0;
        }
    }
}
```

B.10 Simple Reduction

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

```
void reduction_1 (float a[], int m, int n, \
     float sum)
{
    int i, j;
    float local_sum;

    #pragma omp parallel shared(a,m,n,sum) \
                private(i,j,local_sum)
    {
        local_sum = 0.0;
        #pragma omp for nowait
        for(i = 0; i < n; i++) {
            for(j = 0; j < m; j++)
                local_sum = local_sum + a[j + i*m];
        }
        #pragma omp critical
        sum = sum + local_sum;
    }
}
```

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

```
void reduction_2 (float a[], int m, int n, \
     float sum)
{
    int i, j;

    #pragma omp parallel for shared(a,m,n) \
     private(i,j) reduction(+:sum)
    for(i = 0; i < n; i++) {
        for(j = 0; j < m; j++)
            sum = sum + a[j + i*m];
    }
}
```

B.11 threadprivate: Private File-Scope Variable

This example demonstrates the use of `threadprivate` file-scope variables.

```
float work[10000];
#pragma omp threadprivate(work)

extern void construct_data() ;
extern void use_data() ;

void tc_1(int n)
{
   int i;

   #pragma omp parallel shared(n) private(i)
   {
      #pragma omp for
      for(i = 0; i < n; i++) {
        construct_data(); /* fills array work() */
        use_data();       /* uses array work() */
      }
   }
}
```

B.12 threadprivate: Private File-Scope Variable and Master Thread

In this example, the value 2 is printed since the master thread's copy of `threadprivate` variable 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.

```
#include <stdio.h>

int j;
#pragma omp threadprivate(j)

int main()
{
    j = 1;

    #pragma omp parallel copyin(j)
    {
        #pragma omp master
        j = 2;
    }

    printf("j = %d\n", j);
}
```

B.13 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 pragmas.

```
#include <stdio.h>
#include <omp.h>

void reduction_3a (int n, float a[])
{
    int i;
    float gx[8], lx, x;  /* assume 8 processors */

    x = 0.0 ;
    for(i = 0; i < omp_get_max_threads(); i++)
        gx[i] = 0.0;

    #pragma omp parallel shared(a,n,g) \
     private(i,lx)
    {
        lx = 0.0;
        #pragma omp for nowait
        for(i = 0; i < n; i++)
            lx = lx + a[i];

        gx[ omp_get_thread_num() ] = lx;
    }

    for(i = 0; i < omp_get_max_threads(); i++)
        x = x + gx[i];

    printf("x = %f\n", x);
}
```

As shown below, this example can be written without the external routines.

```
#include <stdio.h>
void reduction_3b (int n, float a[])
{
    int i;
    float lx, x;

    x = 0.0;
```

```
    #pragma omp parallel shared(a,n) private(i,lx)
    {
        lx = 0.0;
        #pragma omp for nowait
        for(i = 0; i < n; i++)
            lx = lx + a[i];

        #pragma omp critical
        x = x + lx;
    }

    printf("x = %f\n", x);
}
```

This example can also be written more simply using the `reduction()` clause as follows:

```
#include <stdio.h>
void reduction_3c (int n, float a[])
{
    int i;
    float x;

    x = 0.0 ;

    #pragma omp parallel for shared(a,n) \
     private(i) reduction(+:x)
    for(i = 0; i < n; i++)
        x = x + a[i];

    printf("x = %f\n", x) ;
}
```

B.14 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 pragmas.

```
#include <omp.h>

void local_1a (int n, float a[])
{
 int i, j;
 extern float t[8][100]; /* assume 8 procs max. */
    #pragma omp parallel for shared(a,t,n) \
     private(i,j)
    for(i = 0; i < n; i++) {
       for(j = 0; j < n; j++)
         t[omp_get_thread_num()][j] = a[i] * a[i];
        work( &(t[omp_get_thread_num()][0]) );
    }
}
```

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

```
void local_1b (int n, float a[])
{
    int i, j;
    float t[100];

    #pragma omp parallel for shared(a,n) \
     private(i,t,j)
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++)
            t[j] = a[i] * a[i];
        work( t );
    }
}
```

If `t` is global, then the `threadprivate` pragma can be used instead.

```
float t[100];
#pragma omp threadprivate(t)

void local_1c (int n, float a[])
```

```
{
    int i, j;

    #pragma omp parallel for shared(a,n) \
     private(i,j)
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++)
            t[j] = a[i] * a[i];
        work( t );
    }
}
```

B.15 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.

```
#include <stdio.h>

void dsq3_b (float c[], int n)
{
    int i, j;
    float a[100], b[100], x, y;
    init_a( a, n );
    init_b( b, n );
    #pragma omp parallel for shared(c,n) \
     private(i,j,x,y) firstprivate(a,b)
    for(i = 0; i < n; i++) {
        for(j = 0; j <= i; j++) {
            a[j] = calc_a(i);
            b[j] = calc_b(i);
        }
        for(j = 0; j < n; j++) {
            x = a[i] - b[i];
            y = b[i] + a[i];
            c[j + n*i] = x*y;
        }
    }
    printf(“x, y = %f, %f\n”, x, y );
}
```

B.16 threadprivate: Copying in Initialization Values

Similar to “firstprivate: Copying in Initialization Values” on page 172 except using `threadprivate` variables. For `threadprivate`, `copyin` is used instead of `firstprivate` to copy initialization values from the shared (master) copies of `a` and `b` to the private copies.

```
float a[100], b[100];
#pragma omp threadprivate(a,b)

void dsq3_b_tc (float c[], int n) {
    int i, j;
    float x, y;

    init_a( a, n );
    init_b( b, n );

    #pragma omp parallel for shared(c,n) \
     private(i,j,x,y) copyin(a,b)
    for(i = 0; i < n; i++) {
        for(j = 0; j <= i; j++) {
            a[j] = calc_a(i);
            b[j] = calc_b(i);
        }
        for(j = 0; j < n; j++) {
            x = a[i] - b[i];
            y = b[i] + a[i];
            c[i+n*j] = x*y;
        }
    }
    printf(“x, y = %f, %f\n”, x, y);
}
```

B.17 taskq: Parallelizing across Loop Nests

The OpenMP `for` pragma is limited in that it can only parallelize on a single `for` loop at a time. Using `taskq`, nested loops can be parallelized. Each iteration of the loop is independent and is enqueued as a task.

```
void multiple_doalls( int m, int n, int* sum_p ) {
    int i, j;
    int sum = 0;
    #pragma omp parallel taskq shared(n,m) \
     private(i) lastprivate(j) reduction(+:sum)
    for( i = 0; i < n; ++i ) {
        for( j = 0; j < m; ++j ) {
            partial_sum( &sum );
            #pragma omp task
            do_work( i, j, &sum );
        }
    }
    *sum_p += sum;
    foo( &j );
}
```

APPENDIX C

Fortran Examples

Examples of OpenMP usage in Fortran

The following example programs illustrate the use of OpenMP directives in Fortran.

C.1 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)

!$omp parallel
!$omp&   shared(a,b,n)
!$omp&   private(i,j)
!$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
!$omp end do nowait
!$omp end parallel
         end
```

C.2 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)

   !$omp parallel
   !$omp&   shared(a,b,c,d,m,n)
   !$omp&   private(i,j)
   !$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
   !$omp end do nowait
   !$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
   !$omp end do nowait
   !$omp end parallel
          end
```

C.3 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
!$omp parallel
!$omp&   shared(a,b,n,j)
!$omp&   private(i)
!$omp do
            do i = 1, n
                a(i,j) = b(i,j) / a(i,j-1)
            enddo
!$omp end do nowait
!$omp end parallel
        enddo
        end


        subroutine do_3b (a,b,m,n)
        real a(n,m), b(n,m)

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

C.4 sections: Two Difference Operators

Identical to “do: Two Difference Operators” on page 177 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 177 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)

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

C.5 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)

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

C.6 sections: Updating a Shared Scalar

Identical to “single: Updating a Shared Scalar” on page 180 but using different directives.

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

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

C.7 do: Updating a Shared Scalar

Identical to “single: Updating a Shared Scalar” on page 180 but using different directives.

```
          subroutine do_sp_1 (a,b,n)
          real a(n), b(n)

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

C.8 parallel do: A Simple Difference Operator

Identical to “do: A Simple Difference Operator” on page 176 but using different directives.

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

  !$omp parallel do
  !$omp&   shared(a,b,n)
  !$omp&   private(i,j)
  !$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
```

C.9 parallel sections: Two Difference Operators

Identical to “sections: Two Difference Operators” on page 179 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)

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

C.10 barrier: Testing then Modifying a Shared Object

Using a `barrier` after the first `end do` instead of synchronizing on array `b` everywhere. The `end do` after the first loop is optional but specified in this case.

```
          subroutine barrier_1 (a,b,c,d,m,n)
             real a(n,n), b(n,n), c(m,m), d(m,m)
             real div

!$omp  parallel
!$omp&    shared(a,b,c,d,m,n)
!$omp&    local(i,j,div)
!$omp  do always dynamic trapezoidal
         do i = 1, n-1
            do j = 1, i
               b(j,i) = a(j,i) + a(j,i+1)
            enddo
         enddo
!$omp  end do
         if ( b(1,1) .gt. 100.0 ) then
             div = 2.0
         else
             div = 4.0
         endif
!$omp  barrier
!$omp  do always dynamic guided
         do i = 1, n
            do j = 1, i
               b(j,i) = b(j,i) / div
            enddo
         enddo
!$omp  end do nowait
!$omp  end parallel
        end
```

C.11 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)

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

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

C.12 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)
!$omp threadprivate (/private/) ! this privatizes
                                ! common /private/
!$omp parallel
!$omp&   shared(n)
!$omp&   private(i)
!$omp do
       do i = 1, n
         call construct_data() ! fills array work()
         call use_data()       ! uses array work()
       enddo
!$omp end do nowait
!$omp end parallel
        end
```

C.13 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
!$omp threadprivate (/blk/)

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

          print *, j
          end
```

C.14 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

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

```
        enddo
!$omp end do nowait
!$omp critical
        x = x + lx
!$omp end critical
!$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
!$omp parallel
!$omp&   shared(a)
!$omp&   private(i)
!$omp do reduction(+:x)
        do i = 1, n
            x = x + a(i)
        enddo
!$omp end do nowait
!$omp end parallel

        print *, x
        end
```

C.15 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 procs max
  !$omp parallel do
  !$omp&   shared(a,t)
  !$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)

  !$omp parallel do
  !$omp&   shared(a)
  !$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
  !$omp threadprivate (/cmn/)

  !$omp parallel do
```

```
!$omp&   shared(a)
!$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
```

C.16 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 )
!$omp parallel do shared(c,n) private(i,j,x,y)
!$omp& 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
!$omp end parallel do
          print *, x, y
          end
```

C.17 threadprivate: Copying in Initialization Values

Similar to “firstprivate: Copying in Initialization Values” on page 193 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
!$omp threadprivate (/blk/)

          call init_a( a, n )
          call init_b( b, n )
!$omp parallel do shared(c,n) private(i,j,x,y)
!$omp& 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
!$omp end parallel do
          print *, x, y
          end
```

C.18 Manual loop collapsing

Consider the following FORTRAN 77 code segment which performs the addition of two arrays:

```
...
      integer i, j, k, n, m, l
      parameter (n=1000, m=100, l=10)
      real a(n,m,l), b(n,m,l), c(n,m,l)
...
      do k = 1, l
      do j = 1, m
      do i = 1, n
               a(i,j,k) = b(i,j,k) + c(i,j,k)
      end do
      end do
      end do
...
```

Since most current implementations of OpenMP do not support multiple levels of nested parallelism, only the outermost loop is parallelizable without loop restructuring:

```
...
      integer i, j, k, n, m, l
      parameter (n=1000, m=100, l=10)
      real a(n,m,l), b(n,m,l), c(n,m,l)
...
c Only the outermost loop is parallelized.
!$omp parallel do private(i,j,k) shared(a,b,c,n,m,l)
      do k = 1, l
         do j = 1, m
         do i = 1, n
               a(i,j,k) = b(i,j,k) + c(i,j,k)
         end do
         end do
      end do
!$omp end parallel do
...
```

This is not necessarily the most efficient parallelization possible, because even though the `k` iterations are distributed among threads, the inner `i` and `j` loops still execute in serial. Parallel performance and scalability are likely to be poor if the value of `k` is small compared to the number of processors and/or the values of `i` and

`j`. One possible parallelization strategy is to manually collapse two or more of the loops as follows:

```
...
      integer i, j, k, n, m, l, kj
      parameter (n=1000, m=100, l=10)
      real a(n,n,n), b(n,n,n), c(n,n,n)
...
c Now all of the “k” and “j” iterations are
c distributed among threads; the inner loop is
c still executed in serial.
!$omp parallel do private(i,j,k,kj)
!$omp&            shared(a,b,c,n,m,l)
      do kj = 0, l*m - 1
      k = 1 + kj/m
      j = 1 + mod(kj, m)
         do i = 1, n
               a(i,j,k) = b(i,j,k) + c(i,j,k)
         end do
      end do
!$omp end parallel do
...
```

Instead of distributing `l` iterations among threads, now `l*m` iterations are distributed. However, calculating the loop indices involves an additional computational expense. GuideView can be used to determine whether this parallelization strategy is appropriate for any particular loop (performance and scalability will likely depend on the actual values of the loop indices and the amount of work performed within the loop). See the example called “workshare” on page 197 for another possible parallelization strategy.

C.19 workshare

Consider the following FORTRAN 77 code segment which performs the addition of two arrays:

```
...
      integer i, j, k, n, m, l
      parameter (n=1000, m=100, k=10)
      real a(n,m,l), b(n,m,l), c(n,m,l)
...
      do k = 1, l
      do j = 1, m
      do i = 1, n
               a(i,j,k) = b(i,j,k) + c(i,j,k)
      end do
      end do
      end do
...
```

This computation can be expressed more succintly using Fortran 90 array syntax as follows:

```
...
      integer i, j, k, n, m, l
      parameter (n=1000, m=100, l=10)
      real a(n,m,l), b(n,m,l), c(n,m,l)
...
      a = b + c
...
```

Satisfactory parallelization of this computation is difficult without the `workshare` directive. Since most current implementations of OpenMP do not support multiple levels of nested parallelism, only the outermost loop in the FORTRAN 77 example is parallelizable without code restructuring (see the example called “Manual loop collapsing” on page 195):

```
...
      integer i, j, k, n, m, l
      parameter (n=1000, m=100, l=10)
      real a(n,m,l), b(n,m,l), c(n,m,l)
...
c Only the outermost loop is parallelized.
!$omp parallel do private(i,j,k)
!$omp&            shared(a,b,c,n,m,l)
      do k = 1, l
```

```
            do j = 1, m
            do i = 1, n
                  a(i,j,k) = b(i,j,k) + c(i,j,k)
            end do
            end do
         end do
!$omp end parallel do
...
```

The Fortran 90 example cannot be parallelized at all without the `workshare` directive; placing the array assignment inside of a parallel region merely causes each thread to duplicate the computation:

```
...
      integer i, j, k, n, m, l
      parameter (n=1000, m=100, l=10)
      real a(n,m,l), b(n,m,l), c(n,m,l)
...
c Placing the array assignment inside a parallel
c region produces NO performance gain; all threads
c perform the entire computation
!$omp parallel
      a = b + c
!$omp end parallel
...
```

Finally, here is how this computation can be efficiently parallelized using the `workshare` directive:

```
...
      integer i, j, k, n, m, l
      parameter (n=1000, m=100, l=10)
      real a(n,m,l), b(n,m,l), c(n,m,l)
...
c Placing the array assignment inside a workshare
c directive causes the work to be divided among
c threads
!$omp parallel workshare
      a = b + c
!$omp end parallel workshare
...
```

Evaluation of each element of the array expression is a unit of work, and these units of work are assigned to threads in an such a way that each unit is executed exactly once.

APPENDIX D

Additional KAP/Pro Options

Additional KAP/Pro Options: Alphabetic Listing

This section lists the additional Assure and Guide options that can be specified 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 (short names are given in parentheses). Note that each of these options is preceded by a '-' character. A summary of these options in the form of a table is given in the following section, "Additional KAP/Pro Options: Table" on page 211

c*$*options line (*Fortran only*)

When a Fortran 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**) 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 (see page 204), any **c*$*options** line in the source file is treated as a comment.

-alignmax=<integer>

This is an expert option that you would normally not use. The **-alignmax** option tells Assure or 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> (-a=<string>)
-noassume (-nas)

The **-assume** option instructs Assure or 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.
- **f** The '`l`' value applies only to parallel loops generated automatically from array syntax by Assure or Guide, when **-concurrent** is specified. When '`l`' is specified, Assure or 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, Assure or 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 (-bp)
(-noblank_padding) (-nbp)

The **-blank_padding** option instructs Assure or 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
-nocase (-ncase)

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

When Assure or Guide inserts or modifies lines in a program, it usually creates the new code in capital letters. The **-case** option requires Assure or Guide to preserve the original case of variables in the new code. Making Assure or 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, Assure or Guide will change the **n** to **N**, causing a conflict between two different variables which now have the same name. Using -**WG,-case** would preserve the case-sensitive variable names and avoid any contention.

-chunk=<integer> (-chk=<integer>) (*Guide only*)

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 parallel loops within the source files being compiled. Individual loops can override this default scheduling mechanism by using explicit scheduling options on the `parallel do` or `do` directive. The default chunk size is 1. See "Scheduling Options" on page 139 for descriptions of the -**chunk** options.

-cmp[=<file>]

The **-cmp** option instructs Assure or 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 A_ (Assure) or G_ (Guide) 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) (*Guide only*)
-noconcurrentize (-noconc) (*Guide only*)

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 available 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 (-ds) (*Fortran only*)
-nodatasave (-nds) (*Fortran only*)

The **-datasave** option instructs Assure or 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, 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 Assure or Guide.

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

-directives=p (-dr=p)
-nodirectives (-ndr)

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

-dlines (-dl) (*Fortran only*)
-nodlines (-ndl) (*Fortran only*)

The **-dlines** option instructs Assure or Guide to treat a `D` or `d` in column 1 of Fortran source as a space character. The rest of that line will then be parsed as a normal Fortran statement. By default, Assure or 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:

Fortran syntax:

```
                  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, Assure or Guide sees a `WRITE` statement:

Fortran syntax:

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

-heaplimit=<integer> (-heap=<integer>)

Assure and Guide may require large amounts of memory in order to process the source code. The **-heaplimit** option specifies the maximum size in megabytes that Assure or Guide can use for a heap. If this limit is breached, Assure or Guide will stop processing the source code and try to exit gracefully with an “out of memory” error message. The default size is system-dependent.

If `integer` is greater than the amount of available memory, Assure or Guide may run out of memory before it reaches the heaplimit. Assure and Guide rely upon the operating system to tell it that the OS has run out of memory before that problem occurs. Some operating systems kill Assure and Guide without first saying that there is insufficient memory. In that case, Assure or Guide may stop processing the

code and exit in an undefined manner. Using **-heaplimit** makes a graceful exit more likely.

-ignoreoptions (-ig) (*Fortran only*)
-noignoreoptions (-nig) (*Fortran only*)

The **-ignoreoptions** option directs Assure or Guide to ignore any **c*$*options** or ***$*options** line that may appear at the top of a Fortran input file. Normally, Assure and Guide read the **c*$*options** or ***$*options** instruction for further command line options, as explained in the description of the **c*$*options** line on page 199.

Setting **-noignoreoptions** directs Assure or Guide to acknowledge and accept the **c*$*options** line in the source program. This is the default.

-include=<path> (-inc=<path>)

By default, Assure and Guide look 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. It is recommended that you use the native compiler's include option, often **-I**, instead.

-input=<file> (-i=<file>)

When running Assure or Guide in stand-alone mode, 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, Assure or 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> (-int=<integer>)

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

-lines=<integer> (-ln=<integer>)

The **-lines** option enables the listing from Assure or Guide 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 Assure or Guide to paginate only at subroutine boundaries. The default setting is **-lines=55**.

-list[=<file>]
-nolist

The **-list** option informs Assure or Guide where to place the listing file. When no filename is specified, Assure and Guide derive the default name of the listing file from the input filename by adding `A_` (Assure) or `G_` (Guide) 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> (-lo=<string>)

The **-listoptions** option tells Assure or Guide what optional information to include in the listing, transformed code, and error files, if such files are to be generated.

Any of the following information can be selected:

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

The **-listoptions=k** command line option can be used to determine what your default settings are. The default listing file name is derived from the input filename by adding `A_` (Assure) or `G_` (Guide) to the beginning of the filename and changing the extension to `.out`.

To produce no listing file, enter **-nolist** on the command line. The default value is **-listoptions=ko**.

-logical=<integer> (-log=<integer>)

This option specifies a size in bytes, `integer`, for the default size of `LOGICAL` variables. When `integer=1`, `2`, or `4`, take `LOGICAL*`<**integer**> 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> (-mc=<integer>) (*Guide only*)

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

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 argument value must be a positive integer or 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 non-index operands/results and the non-assignment 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**. This setting might affect performance if branch prediction is an issue.

-onetrip (-1)
-noonetrip (-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 Assure or Guide that the program being processed contains loops which need the *one-trip* feature. This option is off by default.

-optimize=<integer> (-o=<integer>)

The **-optimize** option sets the base optimization and analysis level. The allowed optimization levels and their meanings are:

0 Assure and Guide perform no optimizations on parallel directives.

1 Assure and Guide optimize parallel directives.

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

-real=<integer> (-rl=<integer>)

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

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

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

-recursion (-rc) (*Fortran only*)
-norecursion (-nrc) (*Fortran only*)

The **-recursion** option informs Assure and 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 Assure or Guide processes or unsafe transformations could result.

-roundoff=<string> (-r=<string>) (*Guide only*)

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 allowed **-roundoff** levels and their meanings are:

`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. When **-roundoff>0**, the standards are relaxed.

`1` Guide enables expression simplification and code floating.

-save=<string> (-sv=<string>)

The **-save** option instructs Assure or Guide on how to handle the storage class of local scalar variables. In particular, Assure and Guide can be instructed to perform *live variable analysis* to help decide whether to save the value of a local scalar variable between invocations of a function or a routine by generating a `SAVE` statement. Assure and Guide can also be instructed to treat the default storage class of all local scalar variables as either `AUTOMATIC` or `STATIC`. In any case, neither Assure nor Guide will delete or ignore a hand coded `SAVE` statement.

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

Specifying **-save=all** (**-save=a**) tells Assure or 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 Assure or Guide not to perform live variable analysis. Assure and Guide assume that the necessary `SAVE` statements have been inserted into the code, and will perform 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 **-[no]recursion** option.

Specifying **-save=manual_adjust** (**-save=ma**) instructs Assure or 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 Assure or 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 Assure and Guide optimizations. Accordingly, **-save=ma** should be used with caution.

-scalaropt=<integer> (-so=<integer>) (*Guide only*)

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> (-scan=<integer>)

The **-scan** option allows the length of Fortran input lines to be set. Assure and 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> (-schd=<character>) (*Guide only*)

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 139 for a description of the **-scheduling** options.

-suppress=<string> (-su=<string>)

The **-suppress** option disables the printing of individual classes of Assure or 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

Any number of these options can be combined in a single string, for example **-WG,-suppress=eq**. The default instructs Assure and Guide to report all message types listed above.

-syntax=<string> (-sy=<string>)

The **-syntax** option directs Assure and 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.

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

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

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

The default, **nosyntax**, instructs Assure and Guide to accept a superset FORTRAN 77 and Fortran 90.

-tablesize=<integer> (-ts=<integer>)

The tablesize value is compared to the mathematical product of the number of statements and the number of variables referenced in a given program unit. When the tablesize value is less than this product, a "program-too-large" message will be issued stating the required tablesize.

For Unix-based platforms, please note that you should also carefully review your process resource limits with the `limit` command before adjusting the tablesize command-line switch. Use the commands

```
unlimit
```

or e.g.

```
limit stacksize 32768
```

to increase all, or specific resource limits.

-type (-ty)
-notype (-nty)

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

Additional KAP/Pro Options: Table

The **-WG** driver option specifies additional arguments for Assure or Guide. To state an Assure or 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 default size of `INTEGER`, `LOGICAL`, and `REAL` variables, use **-WG,-integer=8,-logical=8,-real=8**. As another example, to change the scheduling designator and the chunk size, use **-WG,scheduling=d,chunk=4**.

Table D-1 lists the additional KAP/Pro options, 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

Assure and Guide can generate listing files that contain information about the transformations and optimizations it performs. The options in this category control the information Assure or Guide includes in its listing file.

Advanced Optimization

These options customize and fine-tune Guide or Assure for maximum performance.

Fortran Dialect

These options specify the dialect of Fortran in use.

Limits

These options inform Assure or Guide about hardware or software limitations imposed by 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.

Scheduling

These options inform Assure or Guide about scheduling options for parallel work-sharing loops.

.

Table D-1: Additional "-WG,..." KAP/Pro Options

Long Name	Short Name	Default Setting
General Optimization:		
-optimize=<integer>	-o=<integer>	1
-roundoff=<integer> (*Guide only*)	-r=<integer>	0
-scalaropt=<integer> (*Guide only*)	-so=<integer>	0
Input-Output:		
-cmp[=<file>]	-cmp[=<file>]	A_<file> (*Assure*) or G_<file> (*Guide*)
-input=<file>	-i=<file>	<file>
-[no]list=<file>	-[no]list=<file>	nolist
-project_name=<file> (*Assure only*)	-pname=<file>	assure.prj
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 (*Guide only*)	-[no]conc	noconcurrentize
-minconcurrent=<integer> (*Guide only*)	-mc=<integer>	1000

Table D-1: Additional "-WG,..." KAP/Pro Options

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
Limits:		
-heaplimit=<integer>	-heap=<integer>	system-specific
-tablesize=<integer>	-ts=<integer>	24000000
Scheduling:		
-chunk=<integer>	-chk=<integer>	1
-scheduling=<character>	-schd=<character>	e

APPENDIX E

Fortran Directive Translation

Many Fortran 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 systems. Links to UNIX Perl source and Windows binaries are available from *http://www.kai.com/parallel/kappro/helpers*.

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

```
sgi2omp.pl pgm.f > pgm_omp.f
```

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

```
perl c:\KAI\guide40\bin\sgi2omp.pl pgm.f > pgm_omp.f
```

KAP/Pro Parallel Directive to OpenMP Directive Translator

Fortran 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 Assure and Guide. The `kpts2omp.pl` program will help translate KAP/Pro Parallel directives into OpenMP directives that Assure and Guide accept.

The `kpts2omp.pl` program accepts as an argument the name of a Fortran 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 9-6, “kpts2omp.pl Translator Options,” below lists the options that are available when running `kpts2omp.pl`.

Table 9-6 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

Fortran programs which have been parallelized with Cray directives can be used as the basis for a port to Assure or Guide. The `cray2omp.pl` program will help translate Cray Autotasking directives into OpenMP directives that Assure and Guide accept. 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 Fortran 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 9-7, "cray2omp.pl Translator Options," below lists the options that are available when running `cray2omp.pl`.

Table 9-7 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 9-8, “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 9-8 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 9-8 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 pro-`
`cess`
`cmic$ do global`
`cmic$ continue`
`cmic$ getcpus`
`cmic$ numcpus`
`cmic$ relcpus`
`cmic$ soft exit`
`cmic$ micro`	

Cray TASKCOMMON as opposed to OpenMP THREADPRIVATE

The tools provided with Assure and Guide perform a semi-automatic translation of Cray Fortran parallel 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.

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

For example, this Cray version

Fortran syntax:

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

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

```
            ...
        enddo
```

should be translated into:

Fortran syntax:

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

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

SGI Directive to OpenMP Directive Translator

Fortran programs which have been parallelized with SGI `c$` directives can be used as the basis for a port to Assure and 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 Fortran 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 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.

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 9-9, "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 9-9 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

Fortran 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 Fortran 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 9-7, “cray2omp.pl Translator Options,” on page 219, are also available for the `kap2omp.pl` program.

Index