

*ASSURE<sup>TM</sup> Reference Manual*  
*(C/C++ Edition)*

Version 3.9

*ASSURE*<sup>™</sup> Reference Manual  
Version 3.9

Revised March 27, 2000

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

Phone: (217) 356-2288  
FAX: 217-356-5199  
Email: kai@kai.com

URL: <http://www.kai.com/parallel/kapro/assure/>

The information in this document is subject to change without notice. No part of this document may be reproduced, copied or distributed in any form or by any means, electronic or mechanical, for any purpose, without the express written consent of Kuck & Associates, Inc.

© Copyright 1983-2000 by Kuck & Associates, Inc. All rights reserved.

KAI, KAP/Pro Toolset, Assure, and Guide are trademarks of Kuck & Associates, Inc.  
Cray is a registered trademark of Cray Research, Inc.  
DEC and Digital are trademarks of Digital Equipment Corp.  
Java is a trademark of Sun Microsystems, Inc.  
UNIX is a registered Trademark in the USA and other countries, licensed exclusively through X/Open Company Limited.  
All other brand and product names are trademarks or registered trademarks of their respective companies.

GOVERNMENT RESTRICTED RIGHTS. Use, duplication, or disclosure by the U.S. government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c) (1) and (2) of the Commercial Computer Software-Restricted Rights clause at 48 CFR 52.227-19, as applicable.

Printed in the United States of America.

---

# *Table of Contents*

---

<b>CHAPTER 1</b>	1	<i>Introduction</i>
	1	About Assure
	2	Using this Reference Manual
	2	<i>Reference Manual Contents</i>
	2	<i>Reference Manual Conventions</i>
	3	Assure On-line
	3	Technical Support
	3	Comments
<b>CHAPTER 2</b>	5	<i>Using Assure</i>
	5	Introduction
	6	Requirements
	6	How to Verify an Application
	8	An Example
	9	Storage Conflicts

	11	Correcting Errors
	11	<i>Example: Parallelizing Reduction Loops</i>
	14	<i>Example: Privatizing to Resolve Storage Conflicts</i>
	17	<i>Example: Using lastprivate()</i>
	19	<i>Example: Using firstprivate()</i>
	21	Parallel Processing Model
	21	<i>Overview</i>
	25	<i>Data Sharing</i>
<b>CHAPTER 3</b>	27	<b><i>OpenMP Pragmas</i></b>
	28	Parallel Pragma
	28	<i>parallel</i>
	28	Worksharing Pragmas
	28	<i>for</i>
	29	<i>sections</i>
	30	<i>single</i>
	31	Workqueuing Pragmas
	32	<i>The Taskq Model</i>
	32	<i>taskq</i>
	33	<i>task</i>
	33	<i>Data Privatization</i>
	35	<i>Examples</i>
	35	Combined Parallel and Worksharing/Workqueuing Pragmas
	35	<i>parallel for</i>
	36	<i>parallel sections</i>
	38	<i>parallel taskq</i>
	39	Synchronization Pragmas
	39	<i>critical</i>
	39	<i>ordered</i>
	40	<i>master</i>
	40	<i>atomic</i>
	40	<i>flush</i>
	41	<i>barrier</i>

	41	Data Scope Attribute Clauses
	41	<i>default (shared   private   none); shared (&lt;list&gt;); private (&lt;list&gt;)</i>
	42	<i>firstprivate (&lt;list&gt;)</i>
	42	<i>lastprivate (&lt;list&gt;)</i>
	42	<i>reduction (&lt;operator&gt;:&lt;list&gt;)</i>
	43	<i>copyin (&lt;list&gt;)</i>
	43	Privatization of Global Variables
	44	<i>Initializing Threadprivate Variables</i>
	45	<i>Persistence of Threadprivate Variables</i>
	45	Scheduling Options
	48	<i>Scheduling Options Using Pragmas</i>
	48	<i>Scheduling Options Using Environment Variables</i>
<b>CHAPTER 4</b>	49	<i>The Assure Driver</i>
	49	About Assurec
	50	Using the Driver
	50	Driver Options
	51	Driver-specific Options
	51	<i>WAhelp</i>
	51	<i>WAversion</i>
	51	<i>WApject_name=&lt;file&gt;; WApname=&lt;file&gt;; WAprj=&lt;file&gt;</i>
	51	<i>WAstrict</i>
	51	<i>WAnostrict</i>
	52	<i>WAdefault=&lt;class&gt;</i>
	52	<i>WAsched=&lt;type&gt;[,&lt;chunk&gt;]</i>
	52	<i>WAOpt=&lt;integer&gt;</i>
	53	<i>WApocess</i>
	53	<i>WAnopocess</i>
	53	<i>WAonly</i>
	53	<i>WKeep</i>
	53	<i>WAnokeep</i>
	53	<i>WAnowork</i>
	53	<i>WAcritname=&lt;pattern&gt;</i>

54	<i>WAsstatic_library</i>
54	<i>WApath=&lt;path&gt;</i>
54	<i>WAcompiler=&lt;path&gt;</i>
54	<i>WAcc=&lt;path&gt;</i>
54	<i>WAlibpath=&lt;directory&gt;</i>
54	<i>WAcatch=&lt;class&gt;</i>
55	<i>WAnorpath</i>
55	Environment Variables
55	<i>KDD_OUTPUT &lt;filename&gt;</i>
56	<i>KDD_INTERVAL &lt;integer&gt;[<i>{s,m,h,d}</i>]; <i>KDD_DELAY &lt;integer&gt;[<i>{s,m,h,d}</i>]</i></i>
57	<i>KDD_MALLOC</i>

**CHAPTER 5**

59	<i>AssureView</i>
59	Introduction
59	Using AssureView
61	AssureView GUI Elements
62	How to Use the GUI
65	AssureView Options
65	<i>? or h</i>
65	<i>agi=&lt;file&gt;</i>
65	<i>gui</i>
65	<i>nogui</i>
65	<i>prefix=&lt;remove&gt;:&lt;add&gt;</i>
66	<i>project_name=&lt;file&gt; or prj=&lt;file&gt;</i>
66	<i>run_data=&lt;file&gt; or kdd=&lt;file&gt;</i>
66	<i>suppress; nosuppress</i>
67	<i>txt</i>
67	<i>WJ,[java_option]</i>
67	JAVA Options
67	<i>ms&lt;integer&gt;[<i>{k,m}</i>]</i>
67	<i>mx&lt;integer&gt;[<i>{k,m}</i>]</i>
68	<i>nojit; Djava.compiler=none</i>

## CHAPTER 1

*Introduction*

---

*About Assure*

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

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

Throughout this manual, the term “OpenMP pragmas” is used to refer to the KAP/Pro Toolset implementation of the OpenMP specification, unless stated otherwise.

The Assure component of the toolset validates the correctness of parallel programs annotated with OpenMP pragmas 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.

---

## *Using this Reference Manual*

### Reference Manual Contents

Chapter 2, “Using Assure,” beginning on page 5, contains an overview for using Assure, examples to illustrate how to correct common parallel programming errors, and the OpenMP parallel processing model.

Chapter 3, “OpenMP Pragmas,” beginning on page 27, contains definitions for all OpenMP pragmas. OpenMP pragmas specify the parallelism within your code.

Chapter 4, “The Assure Driver,” beginning on page 49, describes the Assure drivers, and it contains descriptions of all Assure command line options. These options allow you to alter Assure’s default behaviors.

Chapter 5, “AssureView,” beginning on page 59, describes how to use AssureView, which graphically displays Assure output.

### Reference Manual Conventions

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

With Assure’s *command line options* and *pragmas*, you can control a program’s parallelization by providing information to Assure. Some of these command line options and pragmas require arguments. In their descriptions, **<integer>** indicates an integer number, **<path>** indicates a directory, **<file>** indicates a filename, **<character>** indicates a single character, and **<string>** indicates a string of characters. For example, **-WAdefault=<string>** in this user’s guide indicates

that a string needs to be provided in order to change the **-WAdefault** option from the default value to a new value (such as **-WAdefault=private**).

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

In Courier typeface, indented where possible.

---

### *Assure On-line*

Visit the Assure Home Page at <http://www.kai.com/parallel/kapro/assure/> for the latest information on Assure.

---

### *Technical Support*

KAI strives to produce high-quality software; however, if Assure produces a fatal error or incorrect results, please send a copy of the source code, a list of the switches and options used, and as much output and error information as possible to Kuck & Associates (KAI), [assure@kai.com](mailto:assure@kai.com).

---

### *Comments*

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



## CHAPTER 2

*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 sequential 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 an Assure-processed program is run, errors are identified when the parallel program is inconsistent with the corresponding sequential program.

Since Assure works by identifying differences between a parallel program and its corresponding sequential version, programs validated by Assure must be assumed to be sequentially correct. No uninitialized accesses, out-of-bounds memory references, etc. should be present in the sequential code.

---

## Requirements

Assure 3.9 requires the native C compiler. AssureView requires a Java™ interpreter, which can be obtained from Sun via the worldwide web. Links to these packages are available on the KAI web site at <http://www.kai.com/parallel/kappro/helpers/>.

---

## How to Verify an Application

The components of Assure include:

- Assurec, the C language compile driver for the purposes of Assure
- Assurec++, the C++ compile driver for the purposes of Assure (note that we will use Assurec to refer to either the Assurec or Assurec++ driver)
- AssureView, a tool for viewing the results of Assure

The steps involved in using Assure to verify an application program are depicted in Figure 2-1, “Assure Process,” on page 7:

1. Compile the program using Assurec.

Assurec takes as input a correct, sequential program that has been parallelized using OpenMP parallel pragmas. This step produces a *project file*, a file with a `.prj` suffix, which is used by AssureView to interpret and display the results. An Assurec link step produces a program that, when executed, generates the Assure results.

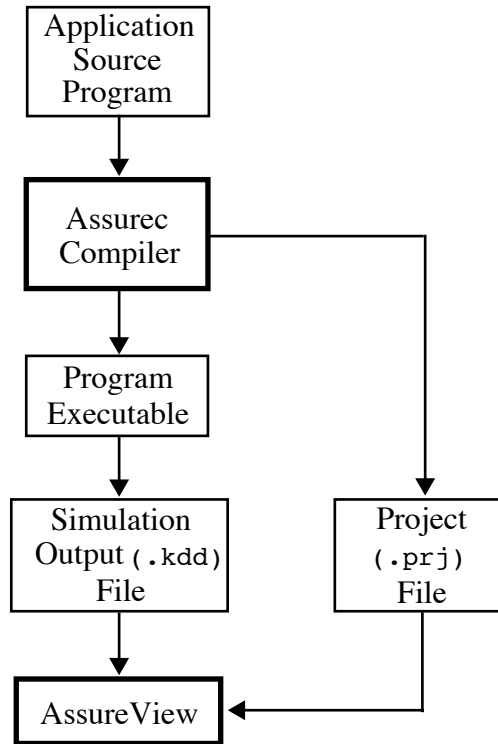
2. Run the compiled program.

This step produces simulation output in a file with a `.kdd` suffix, which is interpreted by AssureView to display the results.

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 or as text.

Figure 2-1 Assure Process



When the application program, here called `pgm`, is contained in a single source file, the following sequence of commands can be used:

```
assurec -o pgm pgm.c
pgm
assureview
```

When the application program consists of multiple source files, the following sequence of commands could be used (the compile and link steps are separated for clarity):

```
assurec -WApname=prog -c file1.c ... fileN.c
assurec -WApname=prog -o pgm file*.o
pgm
assureview prog
```

When using makefiles, it may be sufficient to change `cc` or `CC` and `ld` in each makefile to `Assurecc -WApname=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:

```
assurecc -WApname=/projects/prog/prog.prj
```

Then, the results of Assure can be viewed by using the command `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.

In order to serialize access to project files (e.g., when running makefiles in parallel) an explicit lockfile is associated with each project file. In the preceding examples, if a project name is not specified, the project file is named `assure.prj` and the corresponding lockfile is named `.assure.prj.lck`. If the project file is named `prog.prj`, then the lockfile is named `.prog.prj.lck`. Each lockfile is placed in the same directory as its corresponding project file. Periodic messages may appear on `stderr` if an Assurecc 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 and can be validated by Assure.

```
#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;
    }
}
```

When Assure is run on a complete program containing this program fragment, AssureView will report storage conflicts errors (see “Storage Conflicts” on page 9) on the statements labeled L10, L20, and L30. 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.

```
#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;  
    }  
}
```

When Assure is run on a complete program containing this program fragment, no storage conflict errors are reported, so 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 (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 sequential program.

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

1. Write  $\rightarrow$  Read storage conflicts

These conflicts denote violations of *flow-dependence* (or *true dependence*) constraints. Such a constraint is introduced in a sequential program when one statement writes 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 sequential 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 sequential program when one statement writes 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 should always occur before the *sink* reference.

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

1. A variable was `shared` between processors when it should have been `private` on each processor.
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 can not be directly executed in parallel by the simple change of a variables classification as `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 sequential program in parallel.

---

## *Correcting Errors*

The following examples are designed to illustrate common parallel programming errors and how they may be corrected by using OpenMP pragmas to restructure the parallel code.

### **Example: Parallelizing Reduction Loops**

Consider the following sequential program, which sums the numbers 1 through 10 and prints the answer (55).

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

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:

```
#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);
}
```

Assure identifies a Write → Write storage conflict in this program, on the variable `k` inside the `for` loop.

The standard way 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:

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

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.

```
#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);
}
```

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`. This final reduction needs to be synchronized (serialized) to produce a correct parallel algorithm as follows:

```
#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);
}
```

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

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 pragmas 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 sequential 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 pragmas 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:

```
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;
    }
}
```

The above sequential program 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:

```
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;
    }
}
```

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 reused (potentially by different processors). 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:

```
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;
        }
    }
}
```

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 loop should be run sequentially.

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

```
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;
    }
}
```

In this parallel program, `x` and `y` have been declared private to each processor that executes the parallel region. Privatization removes the storage conflicts by giving each processor 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.

The use of stack based allocation of routine local variables and variables defined within a new scope in C and C++ is an easy way to specify that a very important class of variables is private to each processor. Since each thread executing a parallel region has its own stack, this ensures that whenever multiple processors concurrently call a routine, variables local to that routine are not shared between processors.

OpenMP pragmas permit variables to be privatized in a particular parallel construct. The `private()` clause indicates, for the duration of a parallel region, that each processor executing the region will have a unique, local instance of each `private` variable that will not conflict with the instances of these variables accessed by other processors.

### Example: Using `lastprivate()`

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

```
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);
}
```

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.

```
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);
}
```

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 processor 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 iteration of the parallel loop back into the sequential code.

```

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);
}

```

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

The use of `lastprivate()` is functionally equivalent to executing the last iteration of the loop sequentially.

```

void dsq2_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-1; i++) {
        x = a[i] - b[i];
        y = b[i] + a[i];
        c[i] = x * y;
    }
    if (n >= 0) {
        x = a[n-1] - b[n-1];
        y = b[n-1] + a[n-1];
        c[n-1] = x * y;
    }
    printf("%f %f\n", x, y);
}

```

This is also a correct parallel program that produces the same result; however, the use of `lastprivate()` is preferred since it decreases the size of the routine and requires only one copy of the code in the loop body.

### Example: Using `firstprivate()`

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

```
void dsq3( 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(a,b,c,n) \
        private(i,j) 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[i] - b[i];
            y = b[i] + a[i];
            c[i][j] = x * y;
        }
    }
    printf("%f %f\n", x, y);
}
```

In this example, the variables `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 loop before they are used (the rest of the values 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.

```
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[i] - b[i];
            y = b[i] + a[i];
            c[i][j] = x * y;
        }
    }
    printf("%f %f\n", x, y);
}
```

This parallel program 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. 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.

```
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[i] - b[i];
            y = b[i] + a[i];
            c[i][j] = x * y;
        }
    }
    printf("%f %f\n", x, y);
}
```

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` (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.

---

## *Parallel Processing Model*

This section defines general parallel processing terms and explains how different constructs affect parallel code. For exact semantics, please consult the OpenMP C/C++ API standard document available at <http://www.openmp.org/> or contact KAI at <http://www.kai.com/parallel/kapro/assure/> or email KAI at [assure@kai.com](mailto:assure@kai.com) for more information.

### **Overview**

After placing OpenMP parallel processing pragmas 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 pragmas apply to sections, or

---

blocks, of source code. A structured block can be a single statement or several statements delineated by a “{” “}” pair. See the OpenMP C/C++ API for other rules on structured blocks.

When the master thread enters a parallel region, a team, or group of threads, is formed. Starting from the beginning of the parallel region, code is replicated (executed by all team members) until a worksharing construct is encountered. The `for`, `sections`, and `single` constructs are defined as worksharing constructs because they distribute the enclosed work among the members of the current team. A worksharing construct is only distributed if it occurs dynamically inside of a parallel region. If the worksharing construct occurs lexically inside of the parallel region then it is always executed by distributing the work among the team members. If the worksharing construct is not lexically enclosed by a parallel region (i.e. it is orphaned), then the worksharing construct will be distributed among the team members of the closest dynamically enclosing parallel region if one exists. Otherwise, it will be executed serially.

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

Synchronization constructs are `critical`, `ordered`, `master`, `atomic`, `flush`, and `barrier`. Synchronization can be specified within a parallel region or a worksharing construct with the `critical` pragma. Only one thread at a time is allowed to execute the code within a `critical` section. Within a `for` or `sections` construct, synchronization can be specified with an `ordered` pragma. This pragma is used in conjunction with a `for` or `sections` construct with the `ordered` clause to impose an order on the execution of a section of code. The `master` pragma is another synchronization pragma that can be used to force execution by the master thread. Another way to specify synchronization is with a `barrier` pragma. A `barrier` pragma can be used to force all team members to gather at a particular point in code. Each team member that executes a `barrier` waits at the `barrier` until all of the team members have arrived. `barriers` cannot occur within worksharing or synchronization constructs due to the potential for deadlock.

When a thread reaches the end of a worksharing construct, it may wait until all team members within that construct have completed their work. When all of the work defined by the worksharing construct is completed, the team exits the

worksharing construct and continues executing the code that follows the worksharing construct.

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

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

```

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
  {...}          //
}                // Wait until both units of work complete
                //
...              // More Replicated Code
                //
omp for nowait   // Begin a Worksharing Construct;
for(...) {      // each iteration is a unit of work
                //
...              // Work is distributed among the
                // team members
                //
}                // End of Worksharing Construct; nowait
                // was specified, so no barrier
                //
omp critical     // Begin a Critical Section
{                //
...              // Replicated Code, but only one thread
}                // can execute it at a given time
                //
...              // More Replicated Code
                //
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

```

## 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 also the programmer's responsibility, unless the `firstprivate` clause is specified. In this case, the `private` copy is initialized from the global copy at the start of the construct at which the `firstprivate` clause is specified. In general, updating the global copy of a `private` variable at the end of a parallel region is the programmer's responsibility. However, the `lastprivate` clause of a `for` pragma enables updating the global copy from the team member that executed the last iteration of the `for`.

In addition to the `shared` and `private` clauses, file-scope and namespace-scope variables can be made private to a thread using the `threadprivate` pragma. Threadprivate variables always have  $t$  copies for  $t$  team members. The master thread uses the global copy as its private copy for the duration of each parallel region.

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



## CHAPTER 3

*OpenMP Pragmas*

Assure uses OpenMP pragmas to support a single level of parallelism. Each pragma begins with `#pragma omp`. Please note that items enclosed in square brackets (`[ ]`) are optional. The syntax of the OpenMP pragmas accepted by Assure is presented below.

Many of the pragmas include references to “<new-line>” in their definition. This simply refers to the end of a line and should not be typed.

Many of the pragmas in this chapter include a reference to a <structured-block> in their description. A structured block has a single entry point and a single exit point. No statement is a structured block if there is a jump into or out of that statement (including a call to `longjmp( )` or a use of `throw`, but a call to `exit` is permitted). A compound statement is a structured block 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 Pragma*

### **parallel**

The `parallel` pragma defines a parallel region.

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

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

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

When the logical `if` clause exists, the `<scalar-expression>` is evaluated at run time. If the logical expression evaluates to 0, then all of the code in the parallel region is executed by a team of one thread. If the logical expression evaluates to *non-zero*, then the code in the parallel region may be executed by a team of multiple threads. When the `if` clause is not present, it is treated as if `if (1)` were present.

When a parallel region is encountered in the dynamic scope of another parallel region, the inner parallel region is executed using a team of one thread. The remaining clauses are described in “Data Scope Attribute Clauses” on page 41.

---

## *Worksharing Pragmas*

### **for**

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

```
#pragma omp for [ <clause> [ <clause> ] ... ] <new-line>
    <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 must have the following form:

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

where `<incr-expr>` is one of the following:

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

`<var>` is a signed integer variable that must not be modified in the body of the `for` statement.

`<logic-op>` is one of `<`, `<=`, `>`, or `>=`.

`<lb>`, `<ub>`, and `<incr>` are loop invariant integer expressions. Any side effects in these expressions may produce indeterminate results.

Without the `nowait` clause, all threads that reach the end of the loop will wait until all iterations have been completed. Specifying `nowait` allows early finishing threads to execute code that follows the loop. The `schedule` clause is described in more detail in “Scheduling Options” on page 45. The `ordered` clause is described on page 39.

## sections

The `sections` pragma delineates sections of code that can be executed on different threads. Each parallel section except the first must be enclosed by the `section` pragma. If the `sections` pragma is encountered in the execution of the program

while a parallel region is not active then the pragmas do not cause work to be distributed, and all the sections are executed on the thread that encounters this construct.

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

or,

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

where <clause> is one of the following:

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

The `ordered` clause is a KAP/Pro Toolset extension and is described on page 39.

## single

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

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

where <clause> is one of the following:

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

The first arriving thread is allowed to execute the <structured-block> of code following the `single` pragma. Other threads wait until this thread has fin-

ished the section of code, then they continue executing with the statement after the `single` block. If the `nowait` clause is present, then the other threads do not wait, but instead immediately skip the `<structured-block>`.

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

---

## *Workqueuing Pragmas*

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*.

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`. The `task` pragma cannot be orphaned.
- The number of `task` pragmas inside a `taskq` is determined at run time. For example, a `task` can occur inside a loop contained in a `taskq`.
- `taskq` pragmas can be recursively nested to support, e.g., parallelism in multi-dimensional loops, across linked lists, and over tree-based data.

## The Taskq Model

### taskq

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

```
#pragma omp taskq [ <clause> [ <clause> ] ... ] <new-line>
    <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. (In fact, execution of the `taskq` block can be and often is transferred from one thread to another, so assignment to data indexed by `omp_get_thread_num()` should be avoided.) When the controlling thread encounters a `task` pragma inside the `taskq` block, the work in the `task` block is enqueued, but not immediately executed. Any available worker thread can then dequeue and execute this task.

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 across available threads.

By default, 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 that construct until all enqueued work is complete. When the `nowait` clause is present on a `taskq` construct, however, a thread may proceed past the end of 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.

## task

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

where `<clause>` is the following:

```
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

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. When a task is enqueued, it receives a “snapshot” of the current state of all variables private to the `taskq`. 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.

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):

```
#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);
    }
}
```

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. 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

The `examples` directory includes `taskq` examples, which may serve to clarify the workqueuing model and illustrate its possible uses.

---

## *Combined Parallel and Worksharing/Workqueuing Pragma*

### parallel for

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

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

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

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

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

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

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

```
if (<scalar-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
```

## parallel sections

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

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

or,

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

where <clause> is one of the following:

```

if (<scalar-expression>)
default (shared | private | none)
shared (<list>)
private (<list>)
firstprivate (<list>)
lastprivate (<list>)
reduction (<operator> : <list>)
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> ] ... ] <new-line>
{
  #pragma omp sections nowait [ <sec-clause> \
    [ <sec-clause> ] ... ] <new-line>
  {
    [ #pragma omp section <new-line> ]
      <structured-block>
    [ #pragma omp section <new-line> ]
      <structured-block>
    .
    .
    . ]
  }
}

```

or,

```

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

```

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

```

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

```

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

```

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

```

## parallel taskq

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

where <clause> is one of the following:

```
if (<scalar-expression>)
default (shared | private | none)
shared (<list>)
private (<list>)
firstprivate (<list>)
lastprivate (<list>)
reduction (<operator> : <list>)
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> ] ... ] <new-line>
{
  #pragma omp taskq nowait [ <taskq-clause> \
    [ <taskq-clause> ] ... ] <new-line>
    <structured-block>
}
```

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

```
if (<scalar-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
```

---

## *Synchronization Pragmas*

### **critical**

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

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

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

### **ordered**

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

```
#pragma omp ordered <new-line>  
    <structured-block>
```

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

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

Each ordered section must only be entered once or not at all during the execution of each `for` iteration, `section`, or `task`.

Only one ordered section may be encountered during the execution of each `for` iteration, `section`, or `task`.

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

## master

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

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

Other threads of the team skip the following section of code and continue execution. Note that there is no implied `barrier` on entry to or exit from the master section.

## atomic

This pragma ensures atomic update of a location in memory that may otherwise be exposed to the possibility of multiple, simultaneous, writing threads.

```
#pragma omp atomic <new-line>  
    <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 `>>`.

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

## flush

This pragma 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.

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

The optional list may be used to specify variables that need to be flushed. If the list is absent, all variables are flushed to memory.

## barrier

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

```
#pragma omp barrier <new-line>
```

Barriers force team members to wait at that point in the code until all of the team members encounter that barrier. The `barrier` pragma 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 the 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 variables, place the variable in the `private` list.

The `private` clause is allowed on `parallel`, `for`, `sections`, `taskq` and `task` pragmas. The `default` and `shared` clauses are only allowed on `parallel` pragmas.

When a variable is not present in any list, its default sharing classification is determined based upon the `default` clause. `default(shared)` causes unlisted variables to be `shared`, `default(private)` causes unlisted variables to be `private`, and `default(none)` causes unlisted, but referenced, variables to generate an error. The only exceptions to the `default()` rules are loop control variables (loop indices) of `for` pragmas, `threadprivate` variables, and `const-qualified` variables. The first two are `private`, and the latter is `shared`, unless explicitly overridden. The default is `default(shared)`.

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

### **firstprivate (<list>)**

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

The `firstprivate` clause is allowed on `parallel`, `taskq`, `for`, `sections`, and `single` pragmas.

### **lastprivate (<list>)**

A variable in a `lastprivate()` list is copied back into the variable of the same name in the enclosing context before the execution terminates for the team member that executes the last dynamically encountered `task` of a `taskq` construct, the final iteration of the index set for a `for`, the last lexical `section` of a `sections` construct, or the code enclosed by a `single`, as appropriate. If the loop is executed and the `lastprivate` variable is not written in the last encountered `task` of a `taskq`, in the final iteration of the index set for a `for`, or the last lexical `section` in a `sections` construct, then the value of the shared variable is undefined.

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

### **reduction (<operator>:<list>)**

A variable or array element in the `reduction` list is treated as a reduction by creating a `private` temporary for that variable and updating the original variable after the end of the construct using a `critical` section. The allowed operators are `+`, `-`, `*`, `&`, `^`, `|`, `&&`, and `||`.

The `reduction` clause is allowed on `parallel`, `taskq`, `for`, `sections`, and `single` pragmas. The use of the `reduction` clause on a `single` or `taskq` is a KAP/Pro Toolset extension.

```
#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 example is equivalent to the following:

```
#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;  
    }  
}
```

### copyin (<list>)

The `copyin()` clause applies only to `threadprivate` variables. 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` pragma is only allowed on `parallel` pragmas.

---

## *Privatization of Global Variables*

OpenMP provides privatization of file-scope and namespace-scope variables 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:

```
#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 illegal:

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

while the following is legal:

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

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

main() {
}
```

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:

```
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 is guaranteed to persist only if the dynamic threads mechanism is disabled. Dynamic threads 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 `for` construct. They can be specified in the following three ways:

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

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

For loops that are processed with the `runtime` scheduling mechanism, described below, scheduling can be changed at run time with environment variables. Loop scheduling is dependent on the scheduling mechanism and the chunk parameter. The table below describes each scheduling option. Assume the following: the loop has  $l$  iterations,  $p$  threads execute the loop, and  $n$  is a positive integer specifying the chunk size.

**Table 3-1 Scheduling Options**

Scheduling Type	Chunk	Meaning
<b>static</b>	n	<p>Static scheduling with a chunk size of <math>n</math>. <math>n</math> iterations are dispatched statically to each thread (repeat until <math>l</math> iterations have been dispatched). If <math>n</math> is missing, this is the same as static even scheduling, <math>l/p</math> iterations are dispatched statically to each thread so that each thread gets only a single chunk <math>p</math> of the iteration space.</p> <p>To specify static scheduling with the <code>schedule</code> clause use:</p> <pre>schedule (static[,&lt;integer&gt;])</pre> <p>To specify static scheduling with the <code>OMP_SCHEDULE</code> environment variable use:</p> <pre>OMP_SCHEDULE = static[,&lt;integer&gt;]</pre>
<b>dynamic</b>	n	<p>Dynamic scheduling with a chunk size of <math>n</math>. <math>n</math> iterations are dispatched dynamically to each thread.</p> <p>To specify dynamic scheduling with the <code>schedule</code> clause use:</p> <pre>schedule (dynamic[,&lt;integer&gt;])</pre> <p>To specify dynamic scheduling with the <code>OMP_SCHEDULE</code> environment variable use:</p> <pre>OMP_SCHEDULE = dynamic[,&lt;integer&gt;]</pre> <p>If no chunk size is specified, a size of 1 will be used.</p>

Scheduling Type	Chunk	Meaning
<b>guided</b>	n	<p>Guided scheduling with a minimum chunk size of <i>n</i>. An exponentially decreasing number of iterations are dispatched dynamically to each thread. At least <i>n</i> iterations are dispatched every time except the last.</p> <p>To specify guided scheduling with the <code>schedule</code> clause use:</p> <pre>schedule (guided[,&lt;integer&gt;])</pre> <p>To specify guided scheduling with the <code>OMP_SCHEDULE</code> environment variable use:</p> <pre>OMP_SCHEDULE = guided[,&lt;integer&gt;]</pre> <p>If no chunk size is specified, a size of 1 will be used.</p>
<b>runtime</b>	ignored	<p>Runtime scheduling specifies the scheduling that will be determined via the <code>OMP_SCHEDULE</code> environment variable at run time.</p> <p>To specify runtime scheduling with the <code>schedule</code> clause, use:</p> <pre>schedule (runtime)</pre> <p>To specify runtime scheduling with the <code>OMP_SCHEDULE</code> environment variable use:</p> <pre>OMP_SCHEDULE = &lt;string&gt;[,&lt;integer&gt;]</pre> <p>where <code>&lt;string&gt;</code> is one of <code>static</code>, <code>dynamic</code>, or <code>guided</code> and the optional <code>&lt;integer&gt;</code> parameter is the chunk size for the dispatch method.</p> <p>If the <code>OMP_SCHEDULE</code> environment variable is not set, then the default is assumed to be “<code>dynamic, 1</code>”.</p>

### Scheduling Options Using Pragmas

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

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

The `<integer>` parameter is a chunk size for the dispatch method. If `<integer>` is not specified, it is assumed to be 1 for `dynamic` and `guided`, and assumed to be missing for `static`. See Table 3-1 on page 46 for a complete description of the scheduling options.

The default is `schedule (static)`.

### Scheduling Options Using Environment Variables

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

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

where `<string>` is one of `static`, `dynamic`, or `guided` and the optional `<integer>` parameter is a chunk size for the dispatch method.

## CHAPTER 4

*The Assure Driver*

---

*About Assurec*

The Assure driver, `assurec`, replaces native compiler drivers, such as `cc`. It combines Assure instrumentation and the compile/link step into one command line. In scripts and Makefiles, replacing the compiler with `assurec` will execute the necessary C preprocessor, Assure, and compiler commands automatically.

Assurec is based on KAI C++, a high-performance, ISO standard-compliant C and C++ compiler. This reference manual documents only the places where Assurec's default behavior differs from or extends upon KAI C++. Documentation for KAI C++ is located under the Assurecc installation directory, in:

```
<install-dir>/KCC_docs/.
```

Assurec's default language settings differ from those of KAI C++ in two ways. Assurec's default language is ANSI C, rather than C++. To enable C++, use the `--c++` command line switch or the `assurec++` driver. To improve performance, Assurec disables C++ exceptions by default. Exceptions can be enabled via the `--exceptions` command line switch.

---

## *Using the Driver*

To run Assure, use the following command line:

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

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

If a list of source files is specified on the `assurec` command line without the `-c` compiler option, and if Assure fails to process any of the files, then the driver will compile but not link all successfully processed files.

Instrumented source files (Assure output files) are removed by default after successful Assure instrumentation and compilation. If the **-WKeep** option is specified, however, Assure's output file is not removed.

Assure's output filename 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, Assure would generate a file called `foo.int.c` from a file called `foo.c`, but the object file would be called `foo.o`.

---

## *Driver Options*

The `assurec` driver recognizes all the KAI C++ compiler options, as well as several OpenMP-related options. If `assurec` fails to recognize a command line option, it simply ignores it and passes it directly to KAI C++. Documentation for the KAI C++ command line options is available in the directory `<install-dir>/KCC_docs/`.

In the following descriptions, **<integer>** indicates an integer number, **<path>** indicates a directory name, and **<file>** indicates a file name. Every driver option should be preceded by the “-” character. For example, to see Assure's usage message, add **-WAhelp** to the command line.

### **Displaying all Command Lines**

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

---

## *Driver-specific Options*

### **WAhelp**

This option directs the driver to print a usage message and exit.

### **WAversion**

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

### **WAp<sub>project\_name</sub>=<file>**

### **WAp<sub>name</sub>=<file>**

### **WAp<sub>rj</sub>=<file>**

Any of these equivalent options specifies a name for the Assure project file. 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:

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

### **WAstrict**

This option puts Assure in strict mode, in which it flags non-standard usage of OpenMP pragmas as errors.

### **WAnostrict**

This option instructs Assure to allow KAP/Pro Toolset extensions to OpenMP pragmas. KAP/Pro Toolset's OpenMP extensions include:

- `psingle`, `psections`, and `pfor` are accepted as synonyms for the `single`, `sections`, and `for` pragmas, respectively.
- `ordered` clause is allowed on the `sections` pragma and `ordered` pragmas are allowed within `section` blocks.
- The `lastprivate` and `reduction` clauses are allowed on `single` pragma.
- A `default(private)` clause is allowed on the `parallel` pragma.

- The `taskq` model of unstructured parallelism is enabled.
- A variable may be listed on the `reduction` clauses of both a `parallel` pragma and an enclosed worksharing or workqueuing construct.
- The curly braces may be omitted for the `sections` pragma if it contains only a single `section`.
- The `threadprivate` pragma can be used with local static variables in C.

This option is the default.

### **WAdefault=<class>**

This option specifies the default classification of unlisted variables in OpenMP `parallel` pragmas. Its effect is as if `default(<class>)` were placed on every `parallel` pragma without 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`.

### **WAsched=<type>[,<chunk>]**

This option specifies the default scheduling type and chunk size for OpenMP `for` pragmas. Its effect is as if `schedule(<type>[, <chunk>])` were placed on every `parallel for` and `for` pragma without an explicit `schedule(...)` clause. Allowed values of `<type>` are `static`, `dynamic`, `guided`, and `runtime`. Valid values of the optional `<chunk>` are positive integers. The default value is `static`, with no chunk size. For `dynamic` and `guided`, the default chunk size is 1.

### **WAopt=<integer>**

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

Level 0 optimization disables all pragma optimizations.

Level 1 optimization attempts to remove unnecessary `barrier` pragmas 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.

### **W**Aprocess

The **-WAprocess** option instructs Assure to process OpenMP pragmas into parallel code. This is the default.

### **W**Anoprocess

This is the opposite of **-WAprocess** and instructs Assure to ignore OpenMP pragmas but otherwise process files as usual.

### **W**Aonly

This option instructs the driver to process source files with Assure but not compile them. The default is to compile Assure-processed source files.

### **W**Akeep

Normally, the `assurec` driver removes intermediate files created while processing source files. This option instructs the driver to leave these intermediate files intact.

### **W**Anokeep

This is the opposite of **-WAkeep**. It forces the removal of intermediate files after successful processing. This is the default.

### **W**Anowork

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

### **W**Acritname=<pattern>

This option applies to mixed language programs to allow matching of named and unnamed `critical` and `ordered` pragmas in 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.

### **W**Astatic\_library

By default, `assurec` links using shared libraries where possible. This option instructs the driver to statically link the Assure libraries into the generated executable.

### **W**Apath=<path>

This advanced option is used to specify an alternate path to the Assure executable. The default is determined at the time Assure is installed.

### **W**Acompiler=<path>

This option specifies an alternative path to the native C compiler chosen when Assure was installed.

### **W**Acc=<path>

This is an alternate form of the **-W**Acompiler option.

### **W**Alibpath=<directory>

This option instructs `assurec` to find the Assure libraries in a different location than the default installation directory.

### **W**Acatch=<class>

This option instructs 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 value “none” is the default.

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 within 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 “catch=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 “`catch=safe`” setting causes the program to intercept and report exceptions which violate the OpenMP API for only the `parallel`, `parallel for`, `parallel sections`, `taskq` and `task` constructs. This option has medium run-time overhead.

The “`catch=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.

### WAnorpath

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

---

## *Environment Variables*

The following environment variables control the running Assure-processed executable.

### **KDD\_OUTPUT <filename>**

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 not be acceptable. 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 5 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.

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 if interval updates to the results file are requested.

The exact value to use for the `KDD_DELAY` variable is sometimes not easily predictable. Even when the instrumented program is not checking for errors, it may exhibit a slowdown compared to the original program. For example, a program has three phases, and the user wants to skip error checking until the third phase. The first two phases of the uninstrumented program execute for an hour each

before the third phase starts. Setting `KDD_DELAY` to a value between 2h and 6h should delay error checking long enough to begin checking only after the first two phases are mostly completed.

If you only want to calculate the `STACKSIZE`, 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.

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`.



## CHAPTER 5

*AssureView*

---

*Introduction*

AssureView displays the results of Assure by using the project file information produced by Assurec and the simulation output produced by running the Assurec-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.

The AssureView browser reads a configuration file named `.assureviewrc` or `assure.ini` when it starts up. It looks for the configuration file, in order, in the current directory, in your home directory, and in the directory in which AssureView was installed.

Several options can be configured using this file to control fonts, colors, window size, window location, line numbering, tab expansion in source, and other features of the GUI.

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/assure39/class/example.assureviewrc
```

If the install location 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 Assurec was run on a single-file program and no **-WApname=** was specified to Assurec. Output the results as text to the standard output.

```
assureview myprogram
```

Run AssureView when Assurec was run on a multi-file program with **-WApname=myprogram** specified to Assurec. 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. 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 from which it was called. An icon on each line gives the depth (number of subroutines) 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. 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 on-line **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 “-”.

### **? 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.

### **gui**

The **-gui** option specifies that results should be displayed by using the AssureView GUI. This option is the default.

### **nogui**

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).

### **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> or prj=<file>**

The **-project\_name** option specifies the name of the project file to be used as input to AssureView (see “Using AssureView” on page 59). 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> or kdd=<file>**

The **-run\_data** 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.

### **suppress**

#### **nosuppress**

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 `PARALLEL DO`, or by using local reduction variables and correctly synchronized updates of a global result variable.

The default setting 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 AssureView GUI.

---

# *Index*

---

## **A**

atomic 40

## **B**

barrier 41  
bold typeface 2

## **C**

chunk 46  
control pragmas  
  parallel for 35  
copyin 43  
courier font 2  
critical 39

## **D**

data scope attribute clauses  
  copyin 43  
  default 41  
  firstprivate 42  
  lastprivate 42  
  private 41

  reduction 42  
  shared 41  
default 41

## **E**

environment variables 48  
  omp\_num\_threads 48  
  omp\_scheduling 48  
  scheduling options 48

## **F**

firstprivate 42  
flush 40  
for 28

## **L**

lastprivate 42

## **M**

master 40

## **O**

omp\_num\_threads 48

omp\_scheduling 48  
ordered 39

**P**

parallel 28  
parallel for 35  
parallel for 35  
parallel pragmas  
  parallel 28  
parallel sections 36  
parallel taskq 38  
pragmas  
  atomic 40  
  barrier 41  
  critical 39  
  flush 40  
  for 28  
  master 40  
  ordered 39  
  parallel 28  
  parallel for 35  
  parallel for 35  
  parallel sections 36  
  parallel taskq 38  
  sections 29  
  single 30  
  synchronization 39  
  task 33  
  taskq 32  
private 41

**R**

reduction 42

**S**

scheduling options 45  
  chunk size 46  
  environment variables 48  
sections 29  
shared 41  
single 30  
synchronization pragmas 39, 40  
  atomic 40  
  barrier 41  
  critical 39

flush 40  
master 40  
ordered 39

**T**

task 33  
taskq 32

**W**

workqueuing pragmas  
  task 33  
  taskq 32  
worksharing pragmas  
  for 28  
  parallel for 35  
  parallel sections 36  
  parallel taskq 38  
  sections 29  
  single 30