

***GUIDE**[™] Reference Manual
(C/C++ Edition)*

Version 3.9

GUIDE[™] Reference Manual
Version 3.8

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/guide/>

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

CHAPTER 1	1	<i>Introduction</i>
	1	About Guide
	2	Using this Reference Manual
	2	<i>Reference Manual Contents</i>
	3	<i>Reference Manual Conventions</i>
	3	Guide On-line
	3	Technical Support
	4	Comments
CHAPTER 2	5	<i>Using Guide</i>
	5	Parallel Processing Model
	5	<i>Overview</i>
	7	<i>Increasing Efficiency</i>
	9	<i>Data Sharing</i>
	9	Using Guide to Develop Parallel Programs

	9	<i>Analyze</i>
	10	<i>Restructure</i>
	10	<i>Tune</i>
	10	Orphaned Pragmas
	12	<i>A Few Rules about “Orphaned” Pragmas</i>
CHAPTER 3	15	<i>OpenMP Pragmas</i>
	16	Parallel Pragma
	16	parallel
	16	Worksharing Pragmas
	16	for
	17	sections
	18	single
	19	Workqueuing Pragmas
	20	<i>The Taskq Model</i>
	20	taskq
	21	task
	21	<i>Data Privatization</i>
	23	<i>Examples</i>
	23	Combined Parallel and Worksharing/Workqueuing Pragmas
	23	parallel for
	24	parallel sections
	26	parallel taskq
	27	Synchronization Pragmas
	27	critical
	27	ordered
	28	master
	28	atomic
	28	flush
	29	barrier
	29	Data Scope Attribute Clauses
	29	default (shared private none); shared (<list>); private (<list>)
	30	firstprivate (<list>)
	30	lastprivate (<list>)
	30	reduction (<operator>:<list>)
	31	copyin (<list>)

	31	Privatization of Global Variables
	32	<i>Initializing Threadprivate Variables</i>
	33	<i>Persistence of Threadprivate Variables</i>
	33	Scheduling Options
	36	<i>Scheduling Options Using Pragmas</i>
	36	<i>Scheduling Options Using Environment Variables</i>
	36	Environment Variables
	36	<i>KMP_BLOCKTIME=<integer>[<character>]</i>
	37	<i>KMP_LIBRARY=<string></i>
	37	<i>KMP_STACKSIZE=<integer>[<character>]</i>
	37	<i>KMP_STATSFILE=<file></i>
	38	<i>OMP_DYNAMIC=<boolean></i>
	38	<i>OMP_NUM_THREADS=<integer></i>
	38	<i>OMP_SCHEDULE=<string>[,<integer>]</i>
	39	<i>OMP_NESTED=<boolean></i>
	39	<i>LD_LIBRARY_PATH=<path></i>
CHAPTER 4	41	<i>The Guide Driver</i>
	41	About Guidec
	42	Using the Driver
	42	Driver Options
	43	Driver-specific Options
	43	<i>WGhelp</i>
	43	<i>WGversion</i>
	43	<i>WGproject_name=<file>; WGpname=<file>; WGprj=<file></i>
	43	<i>WGperview</i>
	43	<i>WGnoperview</i>
	43	<i>WGstats</i>
	44	<i>WGnostats</i>
	44	<i>WGstrict</i>
	44	<i>WGnostrict</i>
	44	<i>WGdefault=<class></i>

45	<i>WGsched=<type>[,<chunk>]</i>
45	<i>WGopt=<integer></i>
45	<i>WGprocess</i>
45	<i>WGnoprocess</i>
45	<i>WGonly</i>
46	<i>WGkeep</i>
46	<i>WGnokeep</i>
46	<i>WGnowork</i>
46	<i>WGcritname=<pattern></i>
46	<i>WGstatic_library</i>
46	<i>WGpath=<path></i>
46	<i>WGcompiler=<path></i>
47	<i>WGcc=<path></i>
47	<i>WGlibpath=<directory></i>
47	<i>WGcatch=<class></i>
48	<i>WGnorpath</i>
CHAPTER 5	<i>Libraries</i>
49	Selecting a Library
49	<i>Serial</i>
50	<i>Turnaround</i>
50	<i>Gang</i>
50	<i>Throughput</i>
51	The <i>Guide_stats</i> Library
52	The <i>Guide_perview</i> Library
52	Linking the Libraries
53	External Routines
53	<i>void mppbeg(void); void mppend(void)</i>
54	<i>kmp_get_blocktime(void)</i>
54	<i>kmp_get_library(void)</i>
54	<i>kmp_get_stacksize(void)</i>
55	<i>void kmp_set_blocktime(int)</i>

55	<code>void kmp_set_library(int)</code>
55	<code>void kmp_set_library_serial(void)</code>
55	<code>void kmp_set_library_throughput(void)</code>
55	<code>void kmp_set_library_turnaround(void)</code>
56	<code>void kmp_set_stacksize(int)</code>
56	<code>void kmp_set_parallel_name(char*)</code>
56	<code>void omp_destroy_lock(omp_lock_t *lock);</code>
56	<code>int omp_get_max_threads(void)</code>
56	<code>int omp_get_num_procs(void)</code>
57	<code>int omp_get_num_threads(void)</code>
57	<code>int omp_get_thread_num(void)</code>
57	<code>void omp_init_lock(omp_lock_t *lock);</code>
	<code>void omp_init_nest_lock(omp_nest_lock_t *lock);</code>
57	<code>void omp_set_lock(omp_lock_t *lock);</code>
	<code>void omp_set_nest_lock(omp_nest_lock_t *lock);</code>
58	<code>int omp_test_lock(omp_lock_t *lock);</code>
	<code>int omp_test_nest_lock(omp_nest_lock_t *lock);</code>
58	<code>void omp_unset_lock(omp_lock_t *lock);</code>
	<code>void omp_unset_nest_lock(omp_nest_lock_t *lock);</code>
58	<code>void omp_set_num_threads(int)</code>
58	<code>int omp_in_parallel(void)</code>
58	<code>void omp_set_dynamic(int)</code>
59	<code>int omp_get_dynamic(void)</code>
59	<code>void omp_set_nested(int)</code>
59	<code>int omp_get_nested(void)</code>
59	<i>Signal Handling</i>

CHAPTER 6

61	<i>GuideView</i>
61	Introduction
61	Using GuideView
62	Using Named Parallel Regions
64	GuideView Options
64	<code>mhz=<integer></code>

	64	<i>ovh=<file></i>
	65	<i>jpath=<file></i>
	65	<i>WJ,[java_option]</i>
	65	Java Options
	65	<i>ms<integer>[{{k,m}}</i>
	66	<i>mx<integer>[{{k,m}}</i>
	66	<i>nojit; Djava.compiler=none</i>
CHAPTER 7	67	<i>PerView</i>
	67	Introduction
	67	Enabling the PerView Server
	68	Security
	68	Running with PerView
	68	<i>Starting the Server</i>
	69	<i>kmp_http_port=<port></i>
	69	<i>kmp_http_home=<path></i>
	69	<i>kmp_http_access=<password></i>
	69	<i>Starting the Client</i>
	70	Using PerView
	71	<i>Performance</i>
	72	<i>Controls</i>
	73	<i>Status Bar</i>
	73	<i>Minimal Monitor</i>
	74	Progress Data
	74	<i>Progress Bar</i>
	75	<i>Progress Graph</i>
	75	<i>Progress String</i>
	76	<i>Extending PerView</i>
APPENDIX A	77	<i>Examples</i>
	78	for: A Simple Difference Operator
	79	for: Two Difference Operators

	80	for: Reduce Fork/Join Overhead
	81	sections: Two Difference Operators
	82	single: Updating a Shared Scalar
	83	sections: Updating a Shared Scalar
	84	for: Updating a Shared Scalar
	85	parallel for: A Simple Difference Operator
	86	parallel sections: Two Difference Operators
	87	Simple Reduction
	88	threadprivate: Private File-Scope Variable
	89	threadprivate: Private File-Scope Variable and Master Thread
	90	Avoiding External Routines: Reduction
	92	Avoiding External Routines: Temporary Storage
	93	firstprivate: Copying in Initialization Values
	94	threadprivate: Copying in Initialization Values
	95	taskq: Parallelizing across Loop Nests
 APPENDIX B	 97	 <i>Timing Guide Constructs</i>
	98	Typical Overhead

Table of Contents

CHAPTER 1

Introduction

About Guide

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

The KAP/Pro Toolset is intended for users who understand their application programs and understand parallel processing. The Guide component of the toolset implements the OpenMP API on all popular shared memory parallel (SMP) systems that support threads. The KAP/Pro Toolset uses the de facto industry standard OpenMP 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 input to Guide is a C/C++ program with OpenMP pragmas. The output of Guide is a C program with the pragma parallelism implemented using threads and the Guide support libraries. This output is then compiled using your existing C compiler.

Guide requires the native C compiler. GuideView requires a Java™ interpreter, which can be obtained from Sun or Microsoft via the world wide web. Links to these packages are available on the KAI web site at <http://www.kai.com/parallel/kapro/helpers/>.

Using this Reference Manual

Reference Manual Contents

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

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

Chapter 4, “The Guide Driver,” beginning on page 41, describes the Guide drivers, and it contains descriptions of all Guide command line options. These options allow you to alter Guide’s default behaviors.

Chapter 5, “Libraries,” beginning on page 49, explains the differences among Guide’s several run-time libraries.

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

Chapter 7, “PerView,” beginning on page 67, describes the PerView application manager and monitor.

Appendix A, “Examples,” beginning on page 103, contains code examples with OpenMP pragmas.

Appendix B, “Timing Guide Constructs,” beginning on page 97, shows the expense associated with using OpenMP pragmas.

Reference Manual Conventions

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

With Guide's *command line options* and *pragmas*, you can control a program's parallelization by providing information to Guide. 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, **-WGdefault=<string>** in this user's guide indicates that a string needs to be provided in order to change the **-WGdefault** option from the default value to a new value (such as **-WGdefault=private**).

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

In `Courier` typeface, indented where possible.

Guide On-line

Visit the Guide Home Page at <http://www.kai.com/parallel/kaprof/guide/> for the latest information on Guide.

Technical Support

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

Comments

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

CHAPTER 2

Using Guide

2

Using Guide

Parallel Processing Model

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

Increasing Efficiency

Scheduling options can be selected for the `for` worksharing construct to increase efficiency. Scheduling options specify the way processes are assigned iterations for a loop. A `nowait` option can be used to increase efficiency. The `nowait` option allows processes that finish their work to continue executing code. These processes do not wait at the end of the worksharing or workqueuing construct.

Enabling the option `-WGopt` can also help increase efficiency. For example, using `-WGopt=3` will perform optimizations, such as eliminating unnecessary barriers. The default setting for this option is `-WGopt=3`.

Figure 2-1 “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.

Using Guide to Develop Parallel Programs

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

Analyze

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

Restructure

- If the loop is parallel, introduce a `parallel for` pragma around this loop.
- List the variables that are present in the loop on the `shared()`, `private()`, `lastprivate()`, or `firstprivate()` clauses.
- List the `for` index of the parallel loop as `private()`.
- File-scope variables must not be placed on the `private()` list if their file-scope visibility is to be preserved. Instead, use the `threadprivate` pragma to make a variable private to a thread while preserving its file-scope visibility.
- Attempt to remove cross-iteration dependencies by rewriting the algorithm.
- Synchronize the remaining cross-iteration dependences by placing `critical` pragmas around the uses and assignments to variables involved in the dependences.
- Any I/O in the `parallel` region should be synchronized.
- Identify more parallel loops and restructure them.
- If possible, merge adjacent `parallel fors` into a single parallel region with multiple `fors` to reduce execution overhead.

Tune

- Guide supports the tuning process via the `guide_stats` library and Guide-View. The tuning process should include minimizing the sequential code in `critical` sections and load balancing by using the scheduling options listed in “Scheduling Options” on page 52.

Orphaned Pragas

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

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

```
#pragma omp parallel private(i) shared(n)
{
    #pragma omp for
    for(i=0; i < n; i++) {
        work(i);
    }
}
```

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

```
#pragma omp parallel private(i,j) shared(n)
{
    #pragma omp for
    for(i=0; i < n; i++) {
        some_work(i);
    }
    #pragma omp for
    for(j=0; j < n; j++) {
        more_work(j);
    }
}
```

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

```
void phase1(void) {
    for(i=0; i < n; i++) {
        some_work(i);
    }
}

void phase2(void) {
    for(j=0; j < n; j++) {
        more_work(j);
    }
}
```

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

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

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

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

Notice in this example, the pragmas specifying the parallelism are divided into three separate program units.

A Few Rules about “Orphaned” Pragmas

1. An orphaned worksharing construct (`for/section/single`) or work-queuing construct (`taskq`) that is dynamically executed outside of a parallel region will be executed sequentially. In the following example the first call to `phase0` is executed serially, and the second call is partitioned among the processors on the machine.

```
void main(...) {
    ...
    phase0();
    #pragma omp parallel
    {
        phase0();
    }
    ...
}

void phase0(void) {
    ...
    #pragma omp for
    for(i=0; i < n; i++) {
        other_work(i);
    }
}
```

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

```
void main(...) {
    ...
    #pragma omp parallel
    {
        #pragma omp for
        for(i=0; i < n; i++) {
            bar(i);
        }
    }
    ...
}

void bar(void) {
    #pragma omp barrier
}
```

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

```
void main(...) {
    ...
    #pragma omp parallel
    {
        #pragma omp critical
        test(i);
        ...
    }
}

void test(void) {
    #pragma omp for
    for(i=0; i < n; i++) {
        work(i);
    }
}
```

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

CHAPTER 3

OpenMP Pragmas

Guide 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 Guide 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 29.

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 33. The `ordered` clause is described on page 27.

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 27.

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 Pragas

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

Command line options and pragmas are used to specify the default scheduling mechanism when the source file is being processed by Guide. For loops that are processed with the `runtime` scheduling mechanism, described below, scheduling can be changed at run time with environment variables. Loop scheduling is dependent on the scheduling mechanism and the chunk parameter. The table below describes 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
static	<i>n</i>	<p>Static scheduling with a chunk size of <i>n</i>. <i>n</i> iterations are dispatched statically to each thread (repeat until <i>l</i> iterations have been dispatched). If <i>n</i> is missing, this is the same as static even scheduling, <i>l/p</i> iterations are dispatched statically to each thread so that each thread gets only a single chunk <i>p</i> of the iteration space.</p> <p>To specify static scheduling from the command line use:</p> <pre>-WGsched=static[,<integer>]</pre> <p>or</p> <pre>-WGsched=static [specifies even scheduling]</pre> <p>To specify static scheduling with the <code>schedule</code> clause use:</p> <pre>schedule (static[,<integer>])</pre> <p>To specify static scheduling with the <code>OMP_SCHEDULE</code> environment variable use:</p> <pre>OMP_SCHEDULE = static[,<integer>]</pre>
dynamic	<i>n</i>	<p>Dynamic scheduling with a chunk size of <i>n</i>. <i>n</i> iterations are dispatched dynamically to each thread.</p> <p>To specify dynamic scheduling from the command line use:</p> <pre>-WGsched=dynamic[,<integer>]</pre> <p>To specify dynamic scheduling with the <code>schedule</code> clause use:</p> <pre>schedule (dynamic[,<integer>])</pre> <p>To specify dynamic scheduling with the <code>OMP_SCHEDULE</code> environment variable use:</p> <pre>OMP_SCHEDULE = dynamic[,<integer>]</pre> <p>If no chunk size is specified, a size of 1 will be used.</p>

Scheduling Type	Chunk	Meaning
guided	<i>n</i>	<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 from the command line use:</p> <pre>-WGsched=dynamic[,<integer>]</pre> <p>To specify guided scheduling with the <code>schedule</code> clause use:</p> <pre>schedule (guided[,<integer>])</pre> <p>To specify guided scheduling with the <code>OMP_SCHEDULE</code> environment variable use:</p> <pre>OMP_SCHEDULE = guided[,<integer>]</pre> <p>If no chunk size is specified, a size of 1 will be used.</p>
runtime	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 scheduling at runtime, use the following from the command line:</p> <pre>-WGsched=runtime</pre> <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 = <string>[,<integer>]</pre> <p>where <code><string></code> is one of <code>static</code>, <code>dynamic</code>, or <code>guided</code> and the optional <code><integer></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 34 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.

Environment Variables

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

KMP_BLOCKTIME=<integer>[<character>]

This variable specifies the number of milliseconds that the Guide libraries should wait after completing the execution of a parallel region before putting threads to sleep. Use the optional suffix `s`, `m`, `h`, or `d` to specify seconds, minutes, hours, or days. The default is `1s` or one second. This default may be too

large if threads will be used to execute other threaded code between parallel regions. The default may be too small if threads are reserved solely for the use by the Guide library.

KMP_LIBRARY=<string>

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

- serial
- turnaround
- throughput

See Chapter 5, “Libraries,” beginning on page 49 for more information about the Guide libraries.

KMP_STACKSIZE=<integer>[<character>]

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

KMP_STATSFILE=<file>

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

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

- %H:** This expands into the hostname of the machine running the parallel program.
- %I:** This expands into a unique numeric identifier for this execution of the program. It is the process identifier of the program.

%P: This is replaced with the value of the `OMP_NUM_THREADS` environment variable which determines the number of threads that are created by the parallel program.

OMP_DYNAMIC=<boolean>

The `OMP_DYNAMIC` environment variable enables or disables dynamic adjustment of the number of threads between parallel regions. Enabling dynamic threads allows the Guide library to adjust the number of threads in response to system load. Such an adjustment can improve the turnaround time for all jobs on a loaded system. A value of `TRUE` for `<boolean>` enables dynamic adjustment, whereas a value of `FALSE` disables any change in the number of threads. If dynamic adjustment is enabled, the number of threads may be adjusted only at the beginning of each parallel region. No threads are created or destroyed during the execution of the parallel region.

The default value is `FALSE`.

OMP_NUM_THREADS=<integer>

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

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

OMP_SCHEDULE=<string>[,<integer>]

The `OMP_SCHEDULE` environment variable controls the schedule type and chunk size for `for` constructs with a `schedule(runtime)` clause or those with no `schedule` clause if the command line scheduling designator is set to `runtime`. The schedule type is given by `<string>`, which is one of `static`, `dynamic`, or `guided` and the optional chunk size is given by `<integer>` for those scheduling types which allow a chunk size. See “Scheduling Options” on page 52.

OMP_NESTED=<boolean>

The `OMP_NESTED` environment variable controls whether nested parallelism is enabled at run time. Nested parallelism with nested `parallel` pragmas is currently unimplemented, so this variable has no effect. This environment variable does not affect nested parallelism implemented via nested `taskq` pragmas within a single `parallel` pragma. Allowed values are `TRUE` and `FALSE`, and the default value is `FALSE`.

LD_LIBRARY_PATH=<path>

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

CHAPTER 4

The Guide Driver

About Guidec

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

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

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

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

Using the Driver

To run Guide, use the following command line:

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

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

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

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

Guide'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, Guide 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 `guidec` driver recognizes all the KAI C++ compiler options, as well as several OpenMP-related options. If `guidec` 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 Guide's usage message, add `-WGhelp` 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

WGhelp

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

WGversion

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

WGproject_name=<file>

WGpname=<file>

WGprj=<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:

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

WGperview

This option causes the driver to use the application management and monitoring version of Guide's run-time library. See Chapter 7, "PerView," beginning on page 67 for a complete description of this library.

WGnoperview

This option causes the driver to use the optimized version of the Guide's run-time library. This is the default. See also "WGnostats" on page 44.

WGstats

This option causes the driver to use the statistics version of Guide's run-time library. See Chapter 4, "GuideView," beginning on page 61 for more information on this option.

WGnostats

This option causes the driver to use the optimized version of the Guide's runtime library. This is the default.

WGstrict

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

WGnostrict

This option instructs Guide 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.

WGdefault=<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`.

WGsched=<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.

WGopt=<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.

WGprocess

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

WGnoprocess

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

WGonly

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

WGkeep

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

WGnokeep

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

WGnowork

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

WGcritname=<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.

WGstatic_library

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

WGpath=<path>

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

WGcompiler=<path>

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

WGcc=<path>

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

WGlibpath=<directory>

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

WGcatch=<class>

This option instructs Guide 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.

WGnorpath

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

CHAPTER 5

Libraries

Selecting a Library

Guide supplies three libraries, a development library, a management and monitoring library, and a production library. The production library is called the *guide* library. It should be used for normal or performance-critical runs on applications that have already been tuned. The development library is *guide_stats*. It provides performance information about the code, but it slightly degrades performance. It should be used to tune the performance of applications. The management and monitoring library is called the *guide_perview* library. It can be used to interactively and remotely monitor and manage the parallel performance of a running application program. This library degrades application performance slightly also. All three libraries contain the serial, turnaround, gang, and throughput modes described below. These modes are selected by using the `KMP_LIBRARY` environment variable at run-time; see “`KMP_LIBRARY=<string>`” on page 63.

Serial

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

Turnaround

In a dedicated (batch or single user) parallel environment where all of the processors for a program are exclusively allocated to the program for its entire run, it is most important to effectively utilize all of the processors all of the time. The turnaround mode is designed to keep all of the processors active and involved in the parallel computation to minimize the execution time of a single job. In this mode, the worker threads actively wait for more parallel work, without yielding to other threads.

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

Gang

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

Throughput

In a multi-user environment where the load on the parallel machine is not constant or where the job stream is not predictable, it may be better to design and tune for throughput. This minimizes the total time to run multiple jobs simultaneously. In this mode, the worker threads will yield to other threads while waiting for more parallel work.

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

This is the default.

The Guide_stats Library

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

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

The *guide_stats* library gathers performance statistics for each parallel region executed during a run. During a run, a program follows a number of different sequential code paths between the beginning of a program and the first parallel region, between the end of one parallel region and the beginning of the next, and between the end of the last parallel region and the end of the program. The *guide_stats* library also gathers statistics for each of these sequential paths.

The parallel regions are designated R1, R2, etc., following the order in which they are first executed. Similarly, the sequential code blocks are designated S1, S2, etc., following the order in which they are first executed.

If a parallel region contains one or more barriers, including implicit barriers after worksharing constructs, the parallel region statistics are further subdivided. For example, if the parallel region R3 contains two barriers with the second barrier being at the end of the parallel region, we have separate performance statistics for regions R3 (extending from the beginning of the parallel region to the first barrier) and R3B1 (extending from the first barrier to the end of the parallel region).

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

`kmp_set_parallel_name`. Default names can be restored by supplying an empty string as the argument.

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

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

The Guide_perview Library

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

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

See “PerView,” beginning on page 67 for more information about the use of the *guide_perview* library.

Linking the Libraries

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

```
guidec -WGstats source.c
```

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

External Routines

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

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

In most cases, pragmas can be used in place of these routines. For example, thread-private storage should be implemented by using the `PRIVATE ()` clause of the `parallel` pragma or the `threadprivate` pragma, rather than by explicit expansion and indexing with `omp_get_thread_num ()`. Appendix A, “Examples,” beginning on page 103, contains examples of coding styles that avoid the use of these routines.

To use these functions, include

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

in your source.

void mppbeg(void)

void mppend(void)

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

Calling these routines from another language requires knowledge of the cross-language calling conventions on your platform. A main program written in FORTRAN might look like:

```
program main
  call mppbeg
  call work
  call mppend
end
```

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

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

`kmp_get_blocktime(void)`

This routine returns the integer value of time, in milliseconds, that the Guide libraries wait after completing the execution of a parallel region before putting threads to sleep. This value can be changed via the `kmp_set_blocktime` routine or the `KMP_BLOCKTIME` environment variable. See the description of the `KMP_BLOCKTIME` environment variable on page 63 for more information.

`kmp_get_library(void)`

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

`kmp_get_stacksize(void)`

This routine returns the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can be changed via the `kmp_set_stacksize` routine, prior to the first parallel region or via the `KMP_STACKSIZE` environment variable. See the description of the `KMP_STACKSIZE` environment variable on page 63 for more information.

void kmp_set_blocktime(int)

This routine sets the number of milliseconds that the Guide libraries wait after completing the execution of a parallel region before putting threads to sleep. This value can also be changed via the `KMP_BLOCKTIME` environment variable. See the description of `KMP_BLOCKTIME` on page 63 for more information.

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

void kmp_set_library(int)

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

void kmp_set_library_serial(void)

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

void kmp_set_library_throughput(void)

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

void kmp_set_library_turnaround(void)

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

void kmp_set_stacksize(int)

This routine sets the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can also be changed via the `KMP_STACKSIZE` environment variable. See the description of `KMP_STACKSIZE` on page 63 for more information.

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

void kmp_set_parallel_name(char*)

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

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

void omp_destroy_lock(omp_lock_t *lock);

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

int omp_get_max_threads(void)

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

int omp_get_num_procs(void)

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

int omp_get_num_threads(void)

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

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

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

int omp_get_thread_num(void)

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

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

void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);

These routines initialize a lock associated with the parameter `lock` for use by subsequent calls. The lock parameter must be a pointer to type `omp_lock_t` or `omp_init_nest_lock_t` defined in the header file `omp.h`. The initial state is unlocked. The `lock` variable must only be accessed through the OpenMP library lock routines.

For a nestable lock, the initial nesting count is zero.

void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);

This routine forces the executing thread to wait until the specified lock is available. If the lock is not available, the thread is blocked from further execution until the thread is granted ownership of the lock. The `lock` parameter must be a pointer to a lock variable that was initialized by the `omp_init_lock()` routines.

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

```
int omp_test_lock( omp_lock_t *lock );  
int omp_test_nest_lock( omp_nest_lock_t *lock );
```

These routines try to obtain ownership of the lock, but do not block execution of the calling thread if the lock is not available. The routines return a non-zero value if the lock was successfully obtained; otherwise, they return zero. The `lock` parameter must be a pointer to an initialized lock.

```
void omp_unset_lock( omp_lock_t *lock );  
void omp_unset_nest_lock( omp_nest_lock_t *lock );
```

These routines release the executing thread from ownership of the lock. The behavior is undefined if the executing thread is not the owner of the lock. The `lock` parameter must be a pointer to an initialized lock.

```
void omp_set_num_threads(int)
```

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

```
int omp_in_parallel(void)
```

This function returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns zero.

```
void omp_set_dynamic(int)
```

This function enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. It has effect only when called from serial regions. If the argument is non-zero, the number of threads that are used for executing subsequent parallel regions may be adjusted automatically by the run-time environment to best utilize system resources. As a consequence, the number of threads specified by the `OMP_NUM_THREADS` variable or

`omp_set_num_threads` function is the maximum thread count. The number of threads always remains fixed over the duration of each parallel region. If the argument is zero, dynamic adjustment is disabled.

int omp_get_dynamic(void)

This function returns non-zero if dynamic thread adjustment is enabled and returns zero otherwise.

void omp_set_nested(int)

This function enables or disables nested parallelism. Nested parallelism is not implemented in the KAP/Pro Toolset, so this function has no effect. To exploit nested parallelism, please see “Workqueuing Pragmas” on page 19.

int omp_get_nested(void)

This function always returns zero in the current version of KAP/Pro Toolset.

Signal Handling

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

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

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

CHAPTER 6

GuideView

Introduction

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

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

Using GuideView

GuideView uses as input the statistics file that is output when a Guide instrumented program is run. See “Libraries,” beginning on page 49 to learn how to build an instrumented executable. The syntax for invoking GuideView is as follows:

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

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

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

An example initialization file is provided with your Guide installation. This example file contains comments that explain the meaning and usage of the supported options. If Guide is installed in directory `<install_dir>` on your machine, the example initialization file will be in

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

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

Detailed information about GuideView's operation can be found under its **Help** menu.

Using Named Parallel Regions

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

Once enabled, all following parallel regions are assigned the most recently supplied name, until named regions are disabled by a call to

`kmp_set_parallel_name` with an empty string. The `guide_stats` library gathers performance statistics separately for each named parallel region.

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

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

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

    ...

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

void work( int niter ) {

    int i;

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

        ...

    }
}
```

Named regions can also be used to split the performance statistics of a parallel region for different data sets. In the following example, the parallel region of inter-

est is assigned a name based upon the size of the data set. During a run, the parallel region is executed multiple times, each time with a different data set that activates different names for the parallel region. Performance statistics are gathered separately for each range of data sizes, and the statistics are associated with the appropriate names in the *guide_stats* report and GuideView display. The separate sets of statistics allow analysis of the parallel region as a function of the data set size.

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

    ...

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

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

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

GuideView Options

mhz=<integer>

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

ovh=<file>

The **-ovh=<file>** specifies an overheads file for the input statistics file. There are small overheads that exist in the GuideView library. These overheads can be measured in terms of the number of cycles for each library call or event. You can

override the default values to get more accurate overhead values for your machine by using the **-ovh=<file>** option to create a file that contains machine-specific values.

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

jpath=<file>

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

WJ,[java_option]

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

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

Java Options

The **-WJ** flag must prefix Java options. For example, to pass the **-ms5m** option to the Java interpreter, use **-WJ,-ms5m**.

ms<integer>[k,m]

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

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

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

nojit**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 GuideView GUI.

CHAPTER 7

PerView

7

Introduction

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

Enabling the PerView Server

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

Including the PerView server in your application is as simple as relinking your application with the *guide_perview* library, introduced in “Libraries,” beginning on page 49. To embed the PerView server in your application, add the **-WGperview** flag when linking with the Guidec driver. For example, to build a PerView-enabled application, issue the following commands:

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

Security

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

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

Running with PerView

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

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

Starting the Server

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

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

KMP_HTTP_PORT=<port>

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

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

KMP_HTTP_HOME=<path>

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

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

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

KMP_HTTP_ACCESS=<password>

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

Starting the Client

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

To start the PerView client, type:

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

or

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

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

```
server % guidec -o mondo mondo.c -WGperview
server % setenv KMP_HTTP_PORT 8000
server % setenv KMP_HTTP_ACCESS secret
server % ./mondo

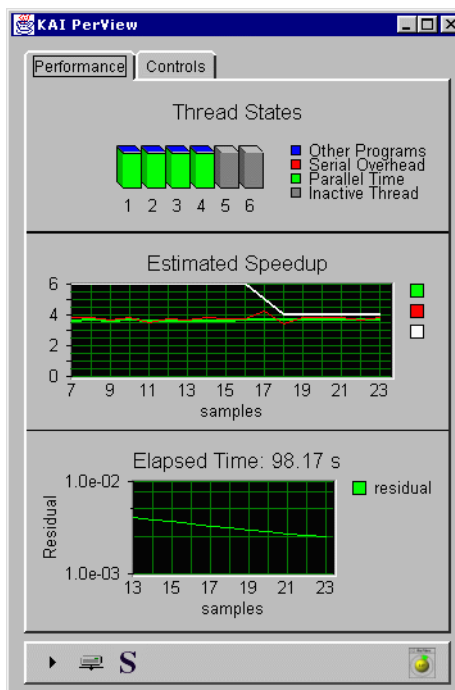
desktop % perview server 8000 secret
cc
```

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

Using PerView

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

Figure 7-1



Performance

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

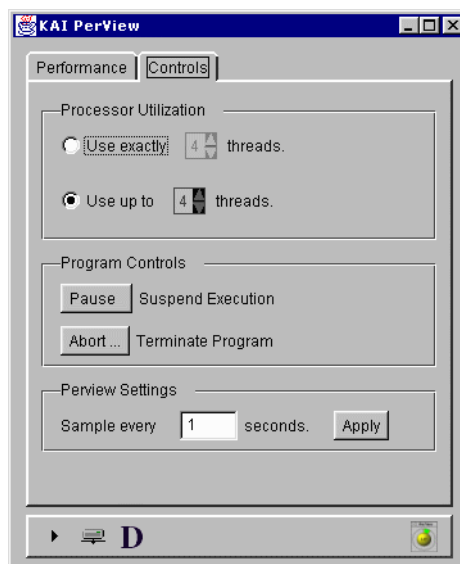
PerView uses this thread state data to estimate the parallel speedup of the application. This instantaneous speedup estimate is plotted, along with its time-averaged value and the thread count, in the center panel. PerView contacts the server at regular intervals to obtain new data. Each data set is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

The bottom panel displays the progress of the application. By default, only the elapsed time since the beginning of the application run is shown here. With the application's cooperation, however, PerView can display a percent completed graph, a string representing progress, or a convergence graph. See "Progress Data" on page 74 for details.

Controls

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

Figure 7-2



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

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

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

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

Status Bar

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

Figure 7-3



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

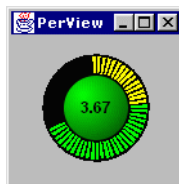
The connection icon indicates whether PerView is connected to the application. When the connection is broken, due to application completion, network failure, or application failure, the icon is obscured by a large, red X.

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

Minimal Monitor

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

Figure 7-4



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

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

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

Progress Data

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

Progress Bar

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

```
void kwebc_set_meter(char* meter_name, int icurrent, int istart, int iend);
```

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

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

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

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

Progress Graph

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

```
void kwebc_set_residual(char* meter_name, int current, int ymin, int ymax);
```

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

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

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

Progress String

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

```
void kwebc_set_string(char* meter_name, char* current_phase);
```

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

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

Extending PerView

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

APPENDIX A *Examples*

The following example programs illustrate the use of OpenMP pragmas.

for: A Simple Difference Operator

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

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

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

for: Two Difference Operators

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

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

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

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

for: Reduce Fork/Join Overhead

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

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

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

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

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

sections: Two Difference Operators

Identical to “for: Two Difference Operators” on page 79 but uses `sections` instead of `for`. Here the speedup is limited to 2 because there are only 2 units of work whereas in “for: Two Difference Operators” on page 79 there are $n-1 + m-1$ units of work.

```
void sections_1 (float a[], float b[], float c[], float d[], \
               int n, int m)
{
    int i, j;
    #pragma omp parallel shared(a,b,c,d,n,m) private(i,j)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for(i = 1; i < n; i++) {
                for(j = 0; j <= i; j++)
                    b[j + n*i] = ( a[j + n*i] + a[j + n*(i-1)] ) / 2.0;
            }

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

A

Examples

single: Updating a Shared Scalar

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

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

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

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

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

sections: Updating a Shared Scalar

Identical to “single: Updating a Shared Scalar” on page 82 but using different pragmas.

```
void sections_sp_1 (float a[], float b[], int n)
{
    int i;
    #pragma omp parallel shared(a,b,n) private(i)
    {
        #pragma omp for
        for(i = 0; i < n; i++)
            a[i] = 1.0 / a[i] ;

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

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

for: Updating a Shared Scalar

Identical to “single: Updating a Shared Scalar” on page 82 but using different pragmas.

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

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

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

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

parallel for: A Simple Difference Operator

Identical to “for: A Simple Difference Operator” on page 78 but using different pragmas.

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

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

parallel sections: Two Difference Operators

Identical to “sections: Two Difference Operators” on page 81 but using different pragmas.

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

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

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

Simple Reduction

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

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

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

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

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

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

A

Examples

threadprivate: Private File-Scope Variable

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

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

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

void tc_1(int n)
{
    int i;

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

threadprivate: Private File-Scope Variable and Master Thread

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

```
#include <stdio.h>

int j;
#pragma omp threadprivate(j)

int main()
{
    j = 1;

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

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

Avoiding External Routines: Reduction

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

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

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

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

    #pragma omp parallel shared(a,n,g) private(i,lx)
    {
        lx = 0.0;
        #pragma omp for nowait
        for(i = 0; i < n; i++)
            lx = lx + a[i];

        gx[ omp_get_thread_num() ] = lx;
    }

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

    printf("x = %f\n", x);
}
```

As shown below, this example can be written without the external routines.

```
#include <stdio.h>
void reduction_3b (int n, float a[])
{
    int i;
    float lx, x;

    x = 0.0;

    #pragma omp parallel shared(a,n) private(i,lx)
    {
        lx = 0.0;
        #pragma omp for nowait
        for(i = 0; i < n; i++)
            lx = lx + a[i];

        #pragma omp critical
        x = x + lx;
    }

    printf("x = %f\n", x);
}
```

This example can also be written more simply using the `reduction()` clause as follows:

```
#include <stdio.h>
void reduction_3c (int n, float a[])
{
    int i;
    float x;

    x = 0.0 ;

    #pragma omp parallel for shared(a,n) private(i) reduction(+:x)
    for(i = 0; i < n; i++)
        x = x + a[i];

    printf("x = %f\n", x) ;
}
```

Avoiding External Routines: Temporary Storage

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

```
#include <omp.h>

void local_1a (int n, float a[])
{
    int i, j;
    extern float t[8][100]; /* assume 8 processors max. */
    #pragma omp parallel for shared(a,t,n) private(i,j)
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++)
            t[omp_get_thread_num()][j] = a[i] * a[i];
        work( &(t[omp_get_thread_num()][0]) );
    }
}
```

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

```
void local_1b (int n, float a[])
{
    int i, j;
    float t[100];

    #pragma omp parallel for shared(a,n) private(i,t,j)
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++)
            t[j] = a[i] * a[i];
        work( t );
    }
}
```

If `t` is global, then the `threadprivate` pragma can be used instead.

```
float t[100];
#pragma omp threadprivate(t)

void local_1c (int n, float a[])
{
    int i, j;

    #pragma omp parallel for shared(a,n) private(i,j)
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++)
            t[j] = a[i] * a[i];
        work( t );
    }
}
```

firstprivate: Copying in Initialization Values

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

```
#include <stdio.h>

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

threadprivate: Copying in Initialization Values

Similar to “firstprivate: Copying in Initialization Values” on page 93 except using `threadprivate` variables. For `threadprivate`, `copyin` is used instead of `firstprivate` to copy initialization values from the shared (master) copies of `a` and `b` to the private copies.

```
float a[100], b[100];
#pragma omp threadprivate(a,b)

void dsq3_b_tc (float c[], int n) {
    int i, j;
    float x, y;

    init_a( a, n );
    init_b( b, n );

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

taskq: Parallelizing across Loop Nests

The OpenMP for pragma is limited in that it can only parallelize on a single for loop at a time. Using `taskq`, nested loops can be parallelized. Each iteration of the loop is independent and is enqueued as a task.

```
void multiple_doalls( int m, int n, int* sum_p ) {
    int i, j;
    int sum = 0;
    #pragma omp parallel taskq shared(n,m) private(i) \
                        lastprivate(j) reduction(+:sum)
    for( i = 0; i < n; ++i ) {
        for( j = 0; j < m; ++j ) {
            partial_sum( &sum );
            #pragma omp task
            do_work( i, j, &sum );
        }
    }
    *sum_p += sum;
    foo( &j );
}
```

A

Examples

APPENDIX B

*Timing Guide
Constructs*

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

```
void null(){};
```

In the table below, it took about 10 cycles to call the null function. A barrier construct is about 10 times slower for 1 processor, and about 70 times slower for 2 processors.

B

Timing Guide
Constructs

Typical Overhead

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

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

- A **barrier** statement is 30 to 50 percent less expensive than a **parallel region**.
- **barriers** and **singles** have roughly the same overhead.
- After 2 processors, all the costs follow a nearly linear pattern as you add processors.

Index

A

atomic 28

B

barrier 29

barrier 7

bold typeface 3

C

chunk 34

control pragmas
parallel for 23

copyin 31

courier font 3

critical 27

D

data scope attribute clauses

copyin 31

default 29

firstprivate 30

lastprivate 30

private 29

reduction 30

shared 29

default 29

E

eliminating 7

environment variables 36, 37, 38, 39

kmp_blocktime 36

kmp_library 37

kmp_stacksize 37

kmp_statsfile 37

ld_library_path 39

omp_dynamic 38

omp_num_threads 38

omp_schedule 38

omp_scheduling 36

scheduling options 36

external routines 53

kmp_get_blocktime 54

kmp_get_library 54

kmp_get_stacksize 54

kmp_set_blocktime 55
kmp_set_library 55
kmp_set_library_serial 55
kmp_set_library_throughput 55
kmp_set_library_turnaround 55
kmp_set_stacksize 56
mppbeg() 53
mppend() 53
omp_destroy_lock() 56
omp_get_max_threads() 56
omp_get_num_procs() 56
omp_get_num_threads() 57
omp_get_thread_num() 57
omp_init_lock() 57
omp_set_lock() 57
omp_test_lock() 58
omp_unset_lock() 58

F

firstprivate 30
flush 28
for 16

G

guideview 61

K

kmp_blocktime 36
kmp_get_blocktime 54
kmp_get_library 54
kmp_get_stacksize 54
kmp_library 37
kmp_set_blocktime 55
kmp_set_library 55
kmp_set_library_serial 55
kmp_set_library_throughput 55
kmp_set_library_turnaround 55
kmp_set_stacksize 56
kmp_stacksize 37
kmp_statsfile 37

L

lastprivate 30
ld_library_path 39
libraries 49, 52

linking 52
selecting 49

linking

libraries 52

M

master 28
mppbeg() 53
mppend() 53

O

omp_destroy_lock() 56
omp_dynamic 38
omp_get_max_threads() 56
omp_get_num_procs() 56
omp_get_num_threads() 57
omp_get_thread_num() 57
omp_init_lock() 57
omp_num_threads 38
omp_schedule 38
omp_scheduling 36
omp_set_lock() 57
omp_test_lock() 58
omp_unset_lock() 58
openmp environment variables 36–38
 kmp_stacksize 37
 ld_library_path 39
 omp_dynamic 38
 omp_schedule 38
ordered 27

P

parallel 16
parallel for 23
parallel for 23
parallel pragmas
 parallel 16
parallel sections 24
parallel taskq 26
perview 67–76
pragmas
 atomic 28
 barrier 29
 critical 27
 flush 28

- for 16
 - master 28
 - ordered 27
 - parallel 16
 - parallel for 23
 - parallel for 23
 - parallel sections 24
 - parallel taskq 26
 - sections 17
 - single 18
 - synchronization 27
 - task 21
 - taskq 20
 - private 29
- R**
- reduction 30
- S**
- scheduling options 33
 - chunk size 34
 - environment variables 36
 - sections 17
 - shared 29
 - single 18
 - synchronization pragmas 27, 28
 - atomic 28
 - barrier 29
 - critical 27
 - flush 28
 - master 28
 - ordered 27
- T**
- task 21
 - taskq 20
- W**
- workqueuing pragmas
 - task 21
 - taskq 20
 - worksharing pragmas
 - for 16
 - parallel for 23
 - parallel sections 24
- parallel taskq 26
 - sections 17
 - single 18