Cluster OpenMP*

User's Guide

Version 9.1

Copyright © 2005-2006 Intel Corporation
All Rights Reserved
Issued in U.S.A.
Document Number: **309076-002 US**

World Wide Web: http://developer.intel.com

Disclaimer and Legal Information

This document as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. INTEL PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS. INTEL MAY MAKE CHANGES TO SPECIFICATIONS AND PRODUCT DESCRIPTIONS AT ANY TIME, WITHOUT NOTICE.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Processors may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel, Intel XScale, and MMX are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands may be claimed as the property of others.

Copyright © 2005-2006 Intel Corporation.

Revision History

Version	Version Information	Date
Version 9.1, Rev 1.0	First version.	August 2005
Version 9.1, Rev 2.0	Added new debugging information.	January 2006
Version 9.1, Rev 3.0	Revised debugging information, added information about OpenMP* libraries supported by Cluster OpenMP*.	February 2006
Version 9.1, Rev 4.0	Added more specific download information.	March 2006
Version 9.1, Rev 4.1	Minor corrections.	May 2006

Contents

1 Introduction 5
1.1 About Cluster OpenMP* 5
1.2 About This Document 5
1.2.1 Intended Audience 5
1.2.2 Document Structure 5
1.2.3 Conventions and Symbols 6
1.3 Related Resources 7
2 Using Cluster OpenMP* 8
2.1 Getting Started 8
2.2 Examples 9
2.2.1 Running a Hello World Program 9
2.2.2 Sample Codes on the Web 10
3 When to Use Cluster OpenMP* 11
4 Porting Your Code 12
4.1 Memory Model and Sharable Variables 12
4.2 Before Porting 12
4.3 Identifying Sharable Variables 12
4.3.1 How the -clomp-sharable-propagation Option Works 13
4.4 Using the Disjoint Heap 15
4.4.1 How the Disjoint Heap Works 15
4.4.2 Language-Specific Steps 17
4.4.3 Using Default(none) to Find Sharable Variables 17
4.5 Promoting Variables to Sharable 18
4.5.1 Automatically Promoting Variables Using the Compiler 18
4.5.2 Manually Promoting Variables 18
4.5.3 Sharable Directive 18
4.5.4 Fortran Considerations 18
4.6 Declaring `omp_lock_t` Variables 19
4.7 Porting Tips 20
5 Compiling a Cluster OpenMP* Program 22
6 Running a Cluster OpenMP* Program 23
6.1 Cluster OpenMP* Startup Process 23
6.2 Cluster OpenMP* Initialization File 24
6.2.1 Overall Format 24
6.2.2 Options Line 25
6.2.3 Environment Variable Part 26
6.3 Input / Output in a Cluster OpenMP* Program 27
6.3.1 Input Files 27
6.3.2 Output Files 27

6.3.3 Mapping Files into Memory 27
6.4 System Heartbeat 27
6.5 Special Cases 28
6.5.1 Using ssh to launch a Cluster OpenMP program 28
6.5.2 Using a Cluster Queuing System 28
6.5.3 Sample PBS Batch File 29
7 Debugging a Cluster OpenMP* Program 30
7.1 Before Debugging 30
7.2 Using the Intel® Debugger 30
7.3 Using the gdb* Debugger 30
7.4 Using the Etnus* TotalView* Debugger 31
7.5 Redirecting I/O 31
8 Evaluating Cluster OpenMP* Performance 32
9 OpenMP* Usage with Cluster OpenMP* 35
9.1 Program Development for Cluster OpenMP* 35
9.1.1 Design the Program as a Parallel Program 35
9.1.2 Write the OpenMP* Program 36
9.2 Combining OpenMP* with Cluster OpenMP* 36
9.3 OpenMP* Implementation-Defined Behaviors in Cluster OpenMP* 37
9.3.1 Number of Threads to Use for a Parallel Region 37
9.3.2 Number of Processors 37
9.3.3 Creating Teams of Threads 37
9.3.4 Schedule(RUNTIME) 38
9.3.5 Various Defaults 38
9.3.6 Granularity of Data 38
9.3.7 Intel Extension Routines/Functions 38
9.4 Cluster OpenMP* Macros 38
9.5 Cluster OpenMP* Environment Variables 39
9.6 Cluster OpenMP* API Routines 39
9.7 Allocating Sharable Memory at Run-Time 40
9.7.1 C++ Sharable Allocation 41
10 Related Tools 43
10.1 Intel® Compiler 43
10.2 Intel® Threading Tools 43
10.2.1 Intel® Thread Checker 43
10.2.2 Intel® Thread Profiler 44
11 Technical Issues 45
11.1 How a Cluster OpenMP* Program Works 45
11.2 The Threads in a Cluster OpenMP* Program 46
11.2.1 OpenMP* Threads 46
11.2.2 DVSM Support Threads 46
11.3 Granularity of a Sharable Memory Access 46
11.4 Socket Connections Between Processes 47

11.5 Using X Window System* Technology with a Cluster OpenMP* Program 47
11.6 Memory Mapping Files 47
11.7 Tips and Tricks 48
11.7.1 Making Assumed-shape Variables Private 48
11.7.2 Missing Space on Partition Where /tmp is Allocated 49
12 Configuring a Cluster 50
12.1 Preliminary Setup 50
12.2 NIS Configuration 51
12.2.1 Head Node NIS Configuration 51
12.2.2 Compute Node NIS Configuration 52
12.3 NFS Configuration 52
12.3.1 Head Node NFS Configuration 52
12.3.2 Compute Node NFS Configuration 53
12.4 Gateway Configuration 53
12.4.1 Head Node Gateway Configuration 53
12.4.2 Compute Node Gateway Configuration 54
13 Reference 55
13.1 Using Foreign Threads in a Cluster OpenMP Program 55
13.2 Cluster OpenMP* Options Reference 55
Glossary 56
Index 58

1 Introduction

This chapter introduces Cluster OpenMP* and provides orientation to this *User's Guide*.

1.1 About Cluster OpenMP*

Cluster OpenMP* is a system that supports running an OpenMP program on a set of nodes connected by a communication fabric, such as Ethernet. Such nodes do not have the shared memory hardware that OpenMP is designed for, so Cluster OpenMP simulates that hardware with a software mechanism. The software mechanism used by Cluster OpenMP is commonly referred to as a *distributed shared memory* system.

1.2 About This Document

This *User's Guide* provides step-by-step instructions for using Cluster OpenMP*.

1.2.1 Intended Audience

This document is intended for users or potential users of Cluster OpenMP*. Users are expected to be familiar with OpenMP* programming and ideally have some experience using clusters and the Intel® compilers.

1.2.2 Document Structure

This User's Guide includes the following chapters:

- Chapter 2, *Using Cluster OpenMP** includes a general usage model for using Cluster OpenMP.
- Chapter3, *When to Use Cluster OpenMP** provides a test you can use to decide if Cluster OpenMP is right for you.
- Chapter 4, *Porting Your Code* describes how to prepare your OpenMP* code for use with Cluster OpenMP by making variables sharable.
- Chapter 5, *Compiling a Cluster OpenMP* Program* provides instructions and tips for compiling your ported Cluster OpenMP program.
- Chapter 6, *Running a Cluster OpenMP* Program* provides instructions and tips for running your compiled Cluster OpenMP program.
- Chapter 7, *Debugging a Cluster OpenMP* Program* provides suggestions for debugging your Cluster OpenMP* program.
- Chapter 8, *Evaluating Cluster OpenMP* Performance* explains how to evaluate your program's performance using Cluster OpenMP and how to determine the optimal number of nodes to use.
- Chapter 9, *OpenMP* Usage with Cluster OpenMP** describes a recommended programming model and provides a reference of OpenMP* information that is specific to Cluster OpenMP.
- Chapter 10, *Related Tools* describes how to use the Intel® Threading Tools to identify sharable variables and improve performance.
- Chapter 11, *Technical Issues* includes advanced technical information, including a description of how Cluster OpenMP* works.
- Chapter 12, *Configuring a Cluster* includes both general instructions for configuring a cluster as well as specific information for configuring a cluster to work with Cluster OpenMP.
- Chapter 13 *Reference* includes a command reference.
- The Glossary provides a guide to terminology used in this document.

1.2.3 Conventions and Symbols

The following conventions are used in this document.

Convention	Explanation	Example
`Monospace`	Filenames	`Ippsapi.h`
	Directory names and pathnames	`\alt\include`
	Commands and command-line options including OpenMP* directives, compiler options, and program text	`Ecl -02`
	Function names, methods, classes, data structures in running text	Use the `okCreateObjs` function to…
`Monospace italics`	Parameters or other placeholders	`ippMalloc(int` *`widthPixels`*`, int*` *`pStepByetes`*`);`
`Monospace bold`	User input	`[c:]` **`dir`**
Italic	Emphasis; introduce or denote items	The term *access* takes as its subject…
`[]`	Brackets indicate optional items	`-Fa[c]` Indicates these command-line options: `-Fa` and `-Fac`
{ | }	Braces and vertical bar(s) indicate choice of one item from a selection of two or more items	`-aX{K | W | P}` Indicates these command-line options: `-aXK, -aXW`, and `-aXP`.
Bold text	GUI elements	Select **Tools** > **Options…**

Notes

- All shell commands in this manual are given in the C shell (`csh`) syntax.
- Any screen shots which appear in this manual are provided for illustration purposes only. The actual program's graphical user interface may differ slightly from the images shown.

1.3 Related Resources

For detailed instructions on using the Intel® compilers, Intel® Thread Checker, or Intel® Thread Profiler, consult the documentation provided with the corresponding product.

For general information about Intel® Software Products, see the Intel® Software website at http://www.intel.com/software/products/index.htm.

Support materials for Cluster OpenMP*, including sample code files are available. To access them:

1. You must register for an account at http://premier.intel.com and login to the account.
2. Select **File Downloads**, then select one of the following products:
   - Intel C++ Compiler, Linux* Cluster OpenMP*
   - Intel Fortran Compiler, Linux* Cluster OpenMP*
3. Click **Display File List**. The extra support materials are found in the file `clomp_tools.tar.gz`.

The support materials contain scripts that should be placed in a directory that is in your path. In this User's Guide, that directory will be referred to as `<CLOMP tools dir>`. Some scripts must be used only on an Intel® EM64T processors and compatible processors and can be found in the support materials in a directory named `/32e`. Other scripts must be used only on an Intel® Itanium® processor, and can be found in a directory named `/64`.
For more information on X Window System* technology and standards, visit the X.Org Foundation at www.x.org.

2 Using Cluster OpenMP*

This chapter presents a recommended model for using Cluster OpenMP* and includes a simple example to illustrate how to use Cluster OpenMP*.

Before you begin, take a moment to consider whether your program can benefit from Cluster OpenMP*. Your program is probably a good candidate for porting to Cluster OpenMP* if one or more of the following conditions is met:

4. You need higher performance than can be achieved using a single node.
5. You want to use a cluster programming model that is easier to use and easier to debug than message-passing (MPI).
6. Your program gets excellent speedup with ordinary OpenMP*.
7. Your program has reasonably good locality of reference and little synchronization.

Tip!

If you are not sure whether Cluster OpenMP* is right for your needs, see Chapter 3, *When to Use Cluster OpenMP** for more details including a step-by-step instructions on how to evaluate the suitability of Cluster OpenMP* for your program.

2.1 Getting Started

At a high level, using Cluster OpenMP* involves the following basic steps. Each step is described in detail in the section noted:

1. Make sure the appropriate Intel Compiler is installed on your system. See the Release Notes for detailed requirements.
2. Make sure your cluster is correctly configured for Cluster OpenMP*. See Chapter 12, *Configuring a Cluster* for complete details.

Note

In most cases, you do not need to do anything special to configure your cluster. You must make sure your program is accessible by the same path in all nodes and that the appropriate compilers and their libraries are accessible with the same path on all nodes. If you output to an X Window, you must set up IP Forwarding for the cluster's interior nodes. See Section 12.4, Gateway Configuration for complete details.*

3. If you already have an existing parallel code using OpenMP*, skip to step 4.
   If you are still planning your code development, see Section 9.1, *Program Development for Cluster OpenMP**, for recommendations and considerations for working with Cluster OpenMP*.
4. Port your code for use with Cluster OpenMP*. Porting involves making variables sharable. You can use the compiler and the Cluster OpenMP run-time library to help you port your code. See Chapter 4, *Porting Your Code*.
5. Run your code using Cluster OpenMP* using a `kmp_cluster.ini` file. See Chapter 6, *Running a Cluster OpenMP* Program.*
6. Debug your code. See Chapter 7, *Debugging a Cluster OpenMP* Program*.

7. Cycle through steps 4 through 6 until your program runs correctly.
8. Tune your code to improve its performance using Intel® Thread Profiler. See Section 10.2.2, *Intel® Thread Profiler.*

2.2 Examples

2.2.1 Running a Hello World Program

Cluster OpenMP requires minimal changes to a conforming OpenMP program. The following example illustrates at a high level how to compile and run a cluster `hello world` program using Cluster OpenMP.

Consider the classic `hello world` program written in C:

```
#include <stdio.h>
main()
{
   printf("hello world\n");
}
```

The equivalent parallel OpenMP "hello world" program is:

```
#include <stdio.h>
main()
{
#pragma omp parallel
      {
#pragma omp critical
      printf("hello world\n");
      }
}
```

To run this program on a cluster:

1. Compile it with the Intel® C++ Compiler version 9.1 using the `-cluster-openmp` option.
2. If the code does not compile correctly, debug your code.
3. Supply a `kmp_cluster.ini` file.
4. Run the executable.

Compiling the program with `-cluster-openmp` inserts the proper code into the executable file for calling the Cluster OpenMP run-time library and links to that library. The `kmp_cluster.ini` file tells the Cluster OpenMP run-time system which nodes to use to run the program and enables you to set up the proper execution environment on all of them.

The following is a sample one-line `kmp_cluster.ini` file that runs the cluster `hello world` program on two nodes, with node names *home* and *remote*. The command is typed on the node *home*, and uses two OpenMP threads on each node for a total of four OpenMP threads:

```
--hostlist=home,remote --process_threads=2
```

With this `kmp_cluster.ini` file in the current working directory, build the OpenMP `hello world` program with the following commands:

```
$ icc  -cluster-openmp hello.c -o hello.exe
```

To run the program, run the resulting executable with:

```
$ ./hello.exe
```

This command produces the following output:

```
hello world

hello world
```

```
hello world

hello world
```

Note that you can change the number of threads per process by changing the value of the `--process_threads` option. You can change the number and identity of the nodes by changing or adding/deleting names in the `--hostlist` option in the `kmp_cluster.ini` file.

2.2.2 Sample Codes on the Web

You can download additional code samples from the Intel support website. See Section 1.3, *Related Resources* for pointers.

3 When to Use Cluster OpenMP*

The major advantage of Cluster OpenMP* is that it makes it relatively easy for you to do parallel programming on a distributed memory system since it uses the same fork-join, shared memory model of parallelism that OpenMP* uses. This model is often easier to use than message-passing paradigms like MPI* or PVM*.

OpenMP is a directive-based language that annotates an underlying serial program. The underlying serial program runs serially when you turn off OpenMP directive processing in the compiler. With planning, you can develop your program just as you would develop a serial program then turn on parallelism with OpenMP. Since you can parallelize your code incrementally, OpenMP usually helps you write a parallel program more quickly and easily than you could with other techniques.

Not all programs are suitable for Cluster OpenMP. If your code meets the following criteria, it is a good candidate for using Cluster OpenMP*:

- **Your code shows excellent speedup with ordinary OpenMP*.**
  If the scalability of your code is poor with ordinary OpenMP on a single node, then porting it to Cluster OpenMP is not recommended. The scalability for Cluster OpenMP will almost certainly be worse than for ordinary OpenMP because Cluster OpenMP has higher overheads for almost all constructs, and sharable memory accesses can be costly. **Ensure that your code gets good speedup with "ordinary" OpenMP*.**
  To test for this condition, run the OpenMP* form of the program (a program compiled with the –`openmp` option) on one node, once with one thread and once with *n* threads, where *n* is the number of processors on one node.
  For the most time-consuming parallel regions, if the speedup achieved for *n* threads is not close to *n*, then the code is not suitable for Cluster OpenMP. In other words, the following formula should be true:
  Speedup = Time(1 thread) / Time(*n* threads) = ~*n*
  Note that this measure of speedup is a *scalability* form of speedup. This measure is not the same as the speedup that measures the quality of the parallelization which uses the best serial time in place of the one thread time above.
- **Your code has good locality of reference and little synchronization.**
  An OpenMP program that gets excellent speedup is likely to get good speedup with Cluster OpenMP as well. However it still will not necessarily get good speedup with Cluster OpenMP. The data access pattern of your code can make Cluster OpenMP programs scale poorly even if it scales well with ordinary OpenMP. For example, if a thread typically accesses large amounts of data that were last written by a different thread, or if there is excessive synchronization, Cluster OpenMP will spend potentially large amounts of time sending messages between nodes.

If you are not sure if your code meets these criteria, you can use the Cluster OpenMP* Suitability Test described in the following section to verify that Cluster OpenMP* is appropriate for your code.

4 Porting Your Code

This chapter describes the memory model used by Cluster OpenMP*and provides instructions to port your code for use with Cluster OpenMP with help from other Intel tools.

4.1 Memory Model and Sharable Variables

The Cluster OpenMP* memory model is based on the OpenMP* memory model. One of the key concepts of this model is knowing whether a variable is used in a *shared* or *private* way in a parallel region. If a variable is shared in a parallel region because the variable name appears in a shared clause, or because of the defaults for a particular parallel region, then the variable is *used in a shared way*. If a variable is used in a shared way in at least one parallel region in a program, then it must be made sharable in a Cluster OpenMP program. If a variable has the sharable attribute, then it can be used in a shared way in any parallel region.

Specifying the difference between *sharable* and *shared* variables almost never arises for OpenMP programs because they run on shared memory multiprocessors, where all variables (except `threadprivate` variables) are automatically *sharable*.

The following table summarizes the assumptions made under OpenMP* versus the assumptions made by Cluster OpenMP* concerning sharability:

OpenMP*	Cluster OpenMP*
All variables are sharable except `threadprivate` variables.	Sharable variables are variables that either: • Are used in a shared way in a parallel region and allocated in an enclosing scope in the same routine. • Appear in a sharable directive.

Table 1: Assumptions about sharability of variables under OpenMP* and Cluster OpenMP*

The compiler automatically applies these assumptions when `-cluster-openmp` or `-cluster-openmp-profile` is specified. It automatically makes the indicated variables sharable. All other variables are non-sharable by default.

Use the Intel compiler's `sharable` directive to declare variables explicitly sharable, as described in Section 4.5 *Promoting Variables to Sharable*.

4.2 Before Porting

Before attempting to port your code, you should determine whether you need to change your code, or if you can port your code automatically. It is possible but unlikely that because of the compiler's defaults for sharability.that you can port your code seamlessly. To do this:

1. Verify that your code works correctly with OpenMP.
2. If your code works correctly with OpenMP, try running it with the`-cluster-openmp` option. If that also works correctly, then you are done porting your code.

If the program doesn't work, you need to identify the variables that must be made sharable with *sharable* directives as described in the following section.

4.3 Identifying Sharable Variables

The second step in porting your code is to use the `-clomp-sharable-propagation` compiler option to

identify variables that must be declared as sharable. To do this:

5. Compile all the source files in your program using the `-clomp-sharable-propagation and -ipo` compiler options and link the resulting object models to produce an executable.
6. Read the resulting compiler warnings and insert the indicated sharable directives in your code.
7. Rebuild and run the executable. If it runs correctly, you're done porting your code. If not, you need to use KMP_SHARABLE_WARNINGS as described in the next section.

4.3.1 How the -clomp-sharable-propagation Option Works

The `-clomp-sharable-propagation` option, used with the `-ipo` compiler option causes the compiler to do an interprocedural analysis of data usage in the program. It findsthe allocation point for variables that are eventually shared in a parallel region in the program. This process is useful for a variable in Fortran that is declared in one routine, passed as an argument in a subroutine or function call, and then shared in a parallel construct in some routine other than the one in which it was declared. It is likewise useful for data in a C program that is declared in one routine, pointed at by a pointer that is passed to a subroutine, then shared in a parallel construct by de-referencing the pointer. It can also be useful for C++ variables that are passed as references to other routines and shared in a parallel construct.

As an example of this analysis, consider the following source files, `pi.f` and `pi2.f`:

Source File pi.f

```
double precision pi
integer nsteps
nsteps = 1000000
call compute(nsteps, pi)
print *, nsteps, pi
end
subroutine calcpi(nsteps, pi,
sum)
double precision pi, sum, step
integer nsteps
double precision x
step = 1.0d0/nsteps
sum = 0.0d0
!$omp parallel private(x)
!$omp do reduction(+:sum)
do i=1, nsteps
  x = (i - 0.5d0)*step
  sum = sum + 4.0d0/(1.0d0 +
x*x)
end do
!$omp end do
```

Source File pi2.f

```
subroutine
compute(nsteps, pi)
double precision pi, sum
integer nsteps
call calcpi(nsteps, pi,
sum)
end
```

`!$omp end parallel`
`pi = step * sum`
`End`	

Table 2: Sample Fortran code with variables that should be made sharable.

To find the variables that must be declared sharable, use the following command:

```
$ ifort -cluster-openmp -clomp-sharable-propagation pi.f pi2.f -ipo
```

The resulting compiler warnings for this example are as follows:

```
IPO: performing multi-file optimizations

IPO: generating object file /tmp/ipo-ifortqKrZN4.o

fortcom: Warning: Sharable directive should be inserted by user as
'!dir$ omp sharable(nsteps)'

in file pi.f, line 2, column 16

fortcom: Warning: Sharable directive should be inserted by user as
'!dir$ omp sharable(sum)'

in file pi2.f, line 2, column 29

pi.f(18) : (col. 6) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.

pi.f(17) : (col. 6) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
```

The highlighted text tells indicates that the variables `nsteps` and `sum` should be made sharable by inserting sharable directives in the source code at the specified lines in `pi.f` and `pi2.f`. With the appropriate sharable directives, the corrected code is:

Source File pi.f

```
      double precision pi
      integer nsteps
!dir$ omp sharable(nsteps)

      nsteps = 1000000

      call compute(nsteps, pi)
      print *, nsteps, pi
      end
      subroutine calcpi(nsteps, pi, sum)
      double precision pi, sum, step
      integer nsteps

      double precision x

      step = 1.0d0/nsteps
      sum = 0.0d0
!$omp parallel private(x)
!$omp do reduction(+:sum)
      do i=1, nsteps
         x = (i - 0.5d0)*step
```

Source File pi2.f

```
      subroutine compute(nsteps, pi)
      double precision pi, sum
      integer nsteps
!dir$ omp sharable(sum)

      call calcpi(nsteps, pi, sum)
      end
```

``` sum = sum + 4.0d0/(1.0d0 + x*x) ```	
``` end do ```	

```
         sum = sum + 4.0d0/(1.0d0 + x*x)
      end do
!$omp end do
!$omp end parallel
      pi = step * sum
      End
```

Compile and execute the two source files by typing:

```
$ ifort –cluster-openmp pi.f pi2.f –o pi.exe
$ ./pi.exe
```

In this example, the compiler can identify all the variables that need to be made sharable for the program to function properly. This is not always the case. For various technical reasons, the compiler may not be able to find all such variables and additional steps are required to identify variables that should be made sharable..

4.4 Using the Disjoint Heap

The third step in porting your code is to use the disjoint heap mechanism. Set the environment variable `KMP_DISJOINT_HEAPSIZE` and then either run your code under a debugger (see Chapter 7, *Debugging a Cluster OpenMP* Program*) or just run it normally. If a heap block is misused, then the program will issue a `SIGSEGV` immediately, and, if you are running under a debugger, then it should show you the point of misuse.

You can use the disjoint heap with a program compiled with optimization, but you will get much more information about the source of the problem if you compile with debugging information (“`-g`”) before running the code using the disjoint heap and debugger.

For example, if you use `csh`, to run your code with the disjoint heap enabled with 128*1024*1024 bytes allocated for it in each process, you would do something like this :

```
% setenv KMP_DISJOINT_HEAP 128M
% ./a.out
Cluster OMP Fatal: Proc#1 Thread#3 (OMP): Segmentation fault
    (ip=0x40000000000013a0 address=0x20000000216159c8)
```

To convert the instruction pointer (“ip”) to a source line you can use Linux’ `addr2line` utility, like this:

```
% addr2line -e a.out 0x40000000000013a0
/usr/anon/tmp/foo.c:9
```

which shows that the access to the heap block which should have been allocated with `kmp_sharable_malloc` happened at line `9` in the file `foo.c`. Given that information you can then read the code to determine the point at which that block was allocated, and change the allocation routine as appropriate.

4.4.1 How the Disjoint Heap Works

When porting a C or C++ code to Cluster OpenMP it is often difficult to find all of the places where memory is allocated which need to be changed to use the routine `kmp_sharable_malloc`, rather than `malloc`. As a result, while you port, you might inadvertently pass pointers to blocks of store which are local to a particular process to other processes which then attempt to read from them. Often such pointers are also valid in the process to which they have been passed, as illustrated in Figure 1: Normal Heap. In such a case, accessing these pointers does not cause a `SIGSEGV` signal. However the data that is read corresponds to whatever data happens to be allocated at that address in the process doing the reading, rather

than the intended value.

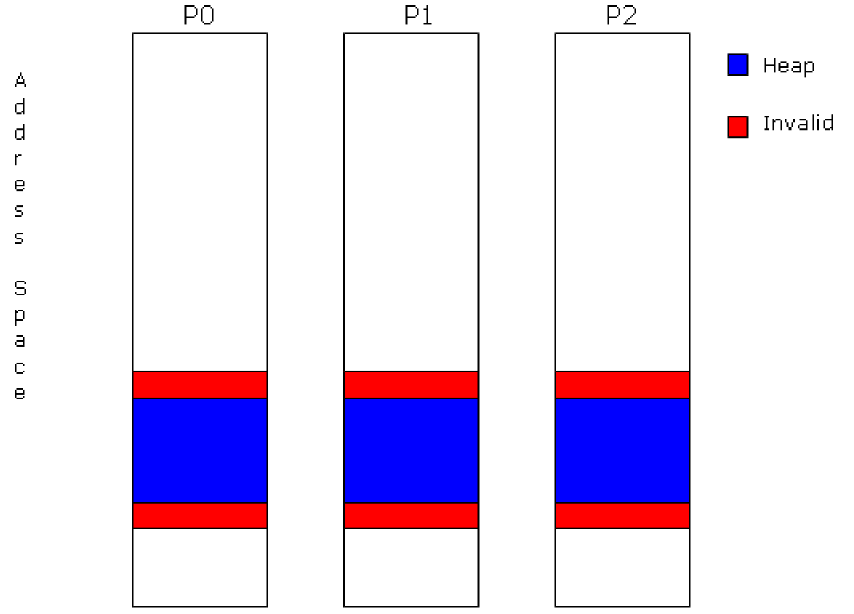


Figure 1: Normal Heap Address Space Layout.

To help you find such problems, you can direct the heap code in Cluster OpenMP to allocate the heap at a different address in each process which makes up the Cluster OpenMP program, as shown in Figure 2: Disjoint Heap Address Space Layout. This direction ensures that when the program attempts to access a pointer to an object in the local heap from a processor other than the one which allocated it the process immediately issues a `SIGSEGV`, rather than continuing to execute with wrong data values, making the problem much easier to find.

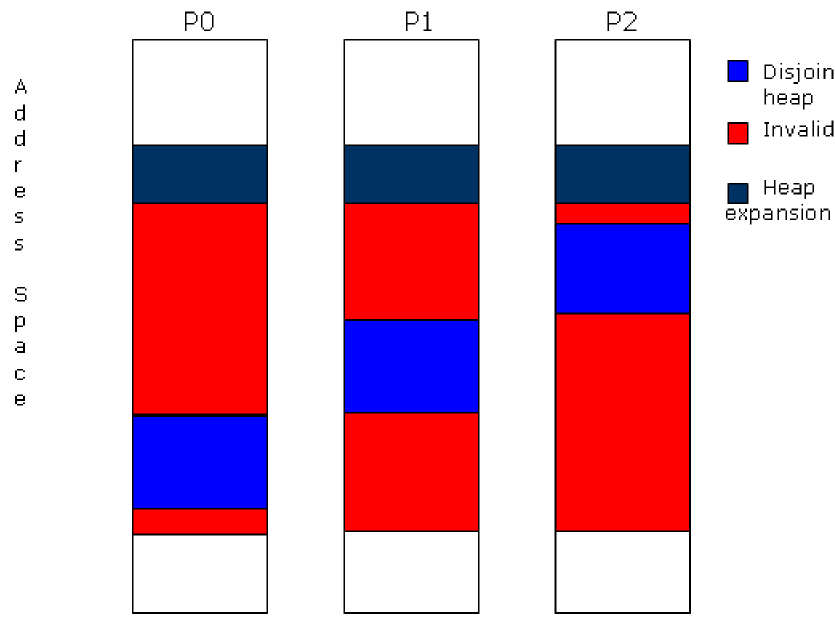


Figure 2: Disjoint Heap Address Space Layout.

Enable the disjoint heap by setting the environment variable `KMP_DISJOINT_HEAPSIZE` to a size. Use use 'K' for KiB (1KiB is 1024 bytes) or 'M', for MiB (1MiB is 1024*1024 bytes)). This environment variable sets the size of the disjoint heap in each process. The minimum value is 2MB. If you set a value lower than the minimum, it will be forced to 2MB. For example, a recommended value is:

```
KMP_DISJOINT_HEAPSIZE = 2M
```

The total address space consumed by the disjoint heap is the size you set for `KMP_DISJOINT_HEAPSIZE` multiplied by the number of processes.

If any process in your program uses more heap space than is allocated for the disjoint heap, a warning message appears. Allocation then continues from a heap expansion area which is very likely not disjoint.

Since the disjoint heap consumes much more address space than the normal heap it is recommended that you use `KMP_DISJOINT_HEAPSIZE` for debugging, but not for large production runs.

4.4.2 Language-Specific Steps

If the first three porting steps don't produce a working program, the fourth step is to try some language-specific fixes, as detailed in this section. For each language, it is important to check for the shared use of dynamically allocated memory. If dynamically-allocated variables are being shared in the parallel construct, or any of the routines called from inside the parallel construct, then you must allocate them out of sharable memory according to the demands of the language you use.

4.4.2.1 Fortran Code

In Fotran, try to isolate the offending variables by using the four Fortran-specific options: `-clomp-sharable-commons`, `-clomp-sharable-modvars`, `-clomp-sharable-localsaves`, and `–clomp-sharable-argexprs`. See Section 4.3.4 *Fortran Considerations* for more information. An `ALLOCATABLE` variable can be used in a parallel construct in a shared way. If this is the case, you should put the variable name of such a variable in a *sharable* directive.

4.4.2.2 C and C++ Code

In C, if memory is allocated with `malloc` or one of the other `malloc`-type routines and then used in a shared way, allocate it using `kmp_sharable_malloc` instead. See Section 9.6 *Cluster OpenMP* API Routines* for a list of the `malloc`-type routines available. Replace the `malloc`-type routine with its Cluster OpenMP analogue. Make sure to replace `free` calls for this memory with `kmp_sharable_free`.

In C++, memory allocated with new and shared in a parallel region will have to be allocated in sharable memory. See Section 9.7.1, *C++ Sharable Allocation*.

In C and C++, it is important to check whether a routine called from within a parallel region is using some file-scope data in a shared way, without the file-scope data being declared sharable.

4.4.3 Using Default(none) to Find Sharable Variables

If your program does not function correctly after the preceding four steps, use the `default(none)` clause to find variables that need to be made sharable. This final step should find all the remaining variables that need to be made sharable. To do this:

1. Place a `default(none)` clause on a parallel directive that seems to reference a non-sharable variable in a shared way. This clause causes the compiler to report all variables that are not explicitly mentioned in a data sharing attribute clause on the parallel directive, and it alerts you to all the variables that must be shared, and as a consequence, sharable.
2. Add variables mentioned in the messages to a `private` or `shared` clause for the parallel region and recompile, until no compiler `default(none)` appear.

3. Use the `-clomp-sharable-info` compiler option to report all variables automatically promoted to sharable.
4. Verify that all variables in the shared clause are either listed in a `-clomp-sharable-info` message or in an explicit `sharable` directive.
5. For a C/C++ program, verify that data shared by dereferencing a pointer is made sharable, since it does not not show up in a `default(none)` message

4.5 Promoting Variables to Sharable

This section describes how to promote variables to sharable. If the procedures described in section 4.4 indicate that certain variables need to be made sharable, follow the instructions in sections 4.5.2, 4.5.34.5.4 to make the variables sharable.

4.5.1 Automatically Promoting Variables Using the Compiler

You do not need to specify `sharable` for all variables that must be allocated in sharable memory. The appropriate Intel® compiler can automatically determine which variables must be sharable and can automatically promote these variables to sharable.

If a variable is stack-allocated in a certain program scope, for example, local variables in a Fortran program unit, or variables declared within a `{} scope` in C or C++), and the variable is also used as a shared variable or in a `firstprivate`, `lastprivate`, or `reduction` clause in any parallel region in that scope, then the compiler will automatically promote the variable to be `sharable`.

4.5.2 Manually Promoting Variables

Manually promoting variables means specifying variables in a sharable directive.

In C/C++, variables you need to specify `sharable` include:

- File-scope variables
- Static variables and stack-allocated variables that are shared in a parallel region outside the current lexical scope or are passed by-reference to a routine where it is used in a shared way
- Static variables and stack-allocated variables that are:
  - shared in a parallel region outside the current lexical scope, or
  - passed by-reference to a routine where it is used in a shared way

In Fortran, these are `COMMON` block names, module variables, variables with the `SAVE` attribute and variables declared locally in a routine and are shared in a parallel region outside the current routine.

4.5.3 Sharable Directive

Use the `sharable` directive to allocate a variable in sharable memory at compile time. The syntax of the *`sharable`* directive is as follows:

Language	Syntax
C/C++	`#pragma intel omp sharable(` *`variable [, variable . . . ]`* `)`
Fortran	`!dir$ omp sharable( variable [, variable . . . ] )`

4.5.4 Fortran Considerations

In Fortran, the `sharable` directive must be placed in the declaration part of a routine.

Common block members can not appear in a sharable directive variable list, since they could break storage association. A common block name (between slashes) can appear in a sharable list, however. For example:

`!dir$ omp sharable(/cname/)` is acceptable.

Variables appearing in an `EQUIVALENCE` statement should not appear in a sharable list since this could break storage association. If variables that appear in an `EQUIVALENCE` statement must be declared sharable, you must place them all together in a new `COMMON` statement, and use the common block name in the sharable directive.

Variables appearing in a Fortran `EQUIVALENCE` statement can not be used in a `SHARABLE` directive.

The Intel® Fortran compiler provides several options that you can use to make each of the following classes of variables sharable by default:

- `COMMON`s
- Module variables
- Local `SAVE` variables
- Temporary variables made for expressions in function and subroutine calls.

For Fortran, use the options in the following table to change how the defaults for making sharable variables are interpreted by the compiler.

Option	Description
`[-no]-clomp-sharable-argexprs`	An argument to any subroutine or function call that is an expression (rather than a simple variable) is assigned to a temporary variable that is allocated in sharable memory. Without this sub-option, such temporary variables are allocated in non-sharable memory. The default is `-no-clomp-sharable-argexprs`.
`[-no]-clomp-sharable-commons`	All common blocks are placed in sharable memory by default. Without this sub-option, all common blocks are placed in non-sharable memory, unless explicitly declared sharable. The default is `-no-clomp-sharable-commons`.
`[-no]-clomp-sharable-localsaves`	All variables declared in subroutines or functions that are not in common blocks, but have the Fortran SAVE attribute are placed in sharable memory by default. Without this sub-option, all such variables are placed in non-sharable memory, unless explicitly declared sharable. The default is `–no-clomp-sharable-localsaves`.
`[-no]-clomp-sharable-modvars`	All variables declared in modules are placed in sharable memory by default. Without this sub-option, all module variables are placed in non-sharable memory, unless explicitly declared sharable. The default is `--no-clomp-sharable-modvars`.

Table 3: Fortran options that control defaults for making variables sharable.

Each of these options makes all variables of a certain class sharable throughout the program. See Chapter 13 *Reference* for more information about these options. You can turn all of these options on to help ensure that all the right data will be made sharable. However, the fewer variables made sharable unnecessarily, the better. So it is best to use these switches as part of an investigation, then only make the necessary variables sharable with a sharable directive.

4.6 Declaring `omp_lock_t` Variables

The `omp_lock_t` variables are used for OpenMP locks and for the Cluster OpenMP condition variable API routines such as `kmp_lock_cond_wait()`. You must allocate these variables in sharable memory. Allocation is done automatically if they are allocated on the stack and are shared in a parallel region in the same routine. They may be mentioned in the list of a `sharable` directive, if necessary.

If you are using `omp_lock_t` variables, you must declare them sharable.

4.7 Porting Tips

The compiler does not automatically make sharable an expression that is passed as an actual argument to a Fortran routine and then used directly in a parallel region. You could create a new sharable variable and copy the value to it, then use that variable in the parallel region as shown in the following example. Note that the actual argument is an expression: `2*size`. To be passed to the argument of the subroutine `foo`, the compiler makes a temporary location to save the value of the expression in before passing it to subroutine `bound`. The temporary location is then passed to the subroutine.

```
      integer size
!dir$ omp sharable(size)
      size = 5
      call foo(2*size)
      end

      subroutine foo(bound)
      integer bound
!$omp parallel do
      do i=1,bound
!$omp critical
         print *,'hello i=',i
!$omp end critical
      enddo

      end
```

To automatically make the temporary variable passed to subroutine `foo` sharable, specify the compiler option `-clomp-sharable-argexprs` on the compile line. This option causes all such expressions used as arguments to function or subroutine calls to be transformed as follows:

```
      integer size
!dir$ omp sharable(size)
      size = 5
      call foo(2*size)
      end

      subroutine foo(bound)
      integer bound, SHbound
!dir$ omp sharable(SHbound)
      SHbound = bound
```

```
!$omp parallel do
      do i=1,SHbound
!$omp critical
         print *,'hello i=',i
!$omp end critical
      enddo

      end
```

5 Compiling a Cluster OpenMP* Program

To compile your ported program for use with Cluster OpenMP, use the appropriate `-cluster-openmp` compiler option. This option produces a production version of a Cluster OpenMP program.

Alternatively, you can use the `–cluster-openmp-profile` option to produce a program that includes detailed performance statistics. Use detailed performance statistics to analyze your program's performance using the Cluster OpenMP* suitability script, or Intel® Thread Profiler (see 8, Evaluating Cluster OpenMP* and 10.2.2, Intel® Thread Profiler, respectively).

You can use these options with both the Intel® C++ Compiler (`icc`) and the Intel® Fortran Compiler (`ifort`). Use one of the following compiler options to generated code for Cluster OpenMP:

For the Intel® C++ Compiler:

```
$ icc -cluster-openmp options source-file
$ icc -cluster-openmp-profile options source-file
```

For the Intel® Fortran Compiler:

```
$ ifort -cluster-openmp options source-file
$ ifort -cluster-openmp-profile options source-file
```

The `–cluster-openmp` and `–cluster-openmp-profile` options automatically link the proper run-time library. The `–cluster-openmp-profile` option also performs extra checking during execution to make sure that the OpenMP constructs are used properly.

6 Running a Cluster OpenMP* Program

To run your compiled Cluster OpenMP program, do the following:

1. Verify that a `kmp_cluster.ini` file exists in the current working directory.
2. Optionally, run the configuration checker script as follows:
   - Locate the configuration checker script in the `<CLOMP tools dir>` directory. See Section 1.3, *Related Resources* for instructions on downloading this script and other examples from the web.
   - In the command prompt, type

     `$ clomp_configchecker.pl` *`program-name`*

     Where ***`program name`*** is the name of your compiled executable.
   - The script does following:
     1. Verifies that the supplied argument is a valid executable
     2. Checks for and parses `kmp_cluster.ini` file.
     3. Pings each node to verify the connection to each node in the configuration file.
     4. Tests a simple `rsh` (or `ssh`) command.
     5. Confirms the existence of the executable on each node.
     6. Verifies the OS and library compatibility of each machine.
     7. If an inconsistency is detected, the script sends a warning to the terminal. If there is a configuration error, the script writes an error message and exits.
     8. Creates a log file, `clomp_configchecker.log`, in the current working directory.
   - Optionally, review the log file produced by the configuration checker script.
3. After correcting any errors reported by the script, type the name of the executable file to execute the program, for example: **`$ ./hello.exe`**. Your executable should run normally.

6.1 Cluster OpenMP* Startup Process

A Cluster OpenMP program does not need a special launch routine. To launch a Cluster OpenMP program, you type the name of the executable file, as you would with any Linux* executable.

The actual startup process that each Cluster OpenMP program goes through is somewhat complicated. It is not necessary to understand it in order to use Cluster OpenMP. However, it is described here in general terms to give you a sense of how it works.

First, the Cluster OpenMP runtime library queries your environment. The system makes an effort to duplicate the environment of the home process on each remote process. The system captures and stores the following key environment variable values for later transmission to the remote processes:

```
PATH,

SHELL,

LD_LIBRARY_PATH.
```

The following shell limits are captured, transmitted to, and applied in the remote processes:

```
core dump size,
cpu time,
file size,
locked-in memory addresses,
memory use,
number of file descriptors,
number of processes,
resident set size,
stack size, and
virtual memory use.
```

Next, the system establishes the Cluster OpenMP options that will be used for the current run. The following steps are used to find an initialization file in which the options are specified. At the first point in these steps where an initialization file is found, the process stops.

1. Look for a `kmp_cluster.ini` file in the current working directory at the time the program is run.
2. If the environment variable `KMP_CLUSTER_PATH` has a value, use it as a path in which to search for a `.kmp_cluster` file.
3. Check your home directory for a `.kmp_cluster` file.
4. Use the following built-in defaults: `--processes=1, --process_threads=1, and --hostlist=<current node>`

If an initialization file is found, it is read to establish values for the options. If not, default values are set, as described in step 4 above. Cluster OpenMP options are processed and any environment variable definitions in the file are applied to the home process and stored for transmission to the remote processes.

Then, the `KMP_CLUSTER_DEBUGGER` environment variable (that you can set) is checked. If it has a value, then the command that started the program is checked to see whether it matches that value (for example, `gdb`*)*. If it matches, then the system prepares to start up all remote processes in the same debugger. If there is no match, the program is started normally.

The home process then opens sockets for each remote process in turn and constructs a command string that is launched to remote processes through an appropriate remote shell command (`rsh` or `ssh`). One socket is set up for communication in each direction between each pair of processes for each thread.

Once communications are set up between the processes, the Cluster OpenMP runtime system initializes itself. Threads are started to handle asynchronous communication between the processes. The system-wide sharable memory is initialized and system control information is allocated there. System-wide locks are allocated and initialized, the OpenMP control structure is initialized, all OpenMP threads are started, and all except the master thread on the home process wait at a barrier for the first parallel region. The same number of OpenMP threads are started on each node, controlled by the `--process_threads` option.

Finally, control returns from the initialization and the master thread on the home node starts running your program.

6.2 Cluster OpenMP* Initialization File

This section describes how to use and customize the Cluster OpenMP* initialization file, `kmp_cluster.ini` for your use.

6.2.1 Overall Format

You put the Cluster OpenMP initialization file, `kmp_cluster.ini`, in the current working directory that is

active when you run your program. The initialization file consists of the following parts:

- **The options line.** The first non-blank, non-comment line in the file is considered to be the options line. You can continue this line on as many lines as you want by using “\” as the last character in each continued line.
- **The environment variable section**. All of the non-blank, non-comment lines following the options line are considered to be in the environment variable section. Each line in the environment variable part must be of the form: `<environment variable name> = <value>`. Where `<value>` is evaluated in the context of your shell. Any values that are permitted by the shell are acceptable as values. The `<value>` is resolved on the home process, then the value is transmitted to each remote process.
- **Comments**. Optionally, comments are designated by the # character as the first character on a line. “#” appearing in any other position in a line of the `kmp_cluster.ini` file has no special meaning, and there are no end-of-line comments.
- Blank lines can appear in the file.

The available options are described in the following section.

6.2.2 Options Line

The following table describes the options that may be specified in the options line of the `kmp_cluster.ini` file, their arguments, and rules for their use:

Option (preceded by --)	Default	Description	Notes
`processes=`*`integer`*	If a value for `omp_num_threads` is specified, the default value is equal to `omp_num_threads / process_threads` Otherwise, the default is equal to the number of hosts in the host pool.	Number of processes to use.	If the value set for `omp-num-threads` does not equal (`processes * process_threads`), Cluster OpenMP* issues an error message and exits.
`process_threads=`*`integer`*	`1`	Number of threads to use per process.	
`omp_num_threads=`*`integer`*	`processes * process_threads`	Number of OpenMP* threads.	
`hostlist=host,host,`…	(home node)	List of host names in the host pool.	These options are mutually exclusive. They specify the host pool, with the default pool consisting of the home node. Processes are started on hosts in the host pool in a round-robin fashion until the appropriate number of processes have been started.
`hostfile=`*`filename`*		Name of a hostname file. The hostname file consists of a list of hostnames, one per line, which defines the host pool.	
`launch=`*`keyword`*	`rsh`	Keywords: {`rsh, ssh`} The method for launching the Cluster OpenMP* program on remote nodes.	
`sharable_heap =`*`integer`*	`16384`	The initial number of pages to allocate for sharable memory.	
`transport=`*`keyword`*	`tcp`	Keywords: {`tcp, dapl`} The network transport to use for communication between Cluster OpenMP processes.	
`adapter=`*`name`*	*`none`*	Name of the DAPL adapter to use.	You must specify a value if `transport=dapl` is

		For example, `--adapter=Openib-ib0`.	specified.
suffix=*string*	*null*	Hostname suffix to append to host names in the host pool. This is useful when a cluster has multiple interconnects available.	
startup_timeout=*integer*	30	Set the number of seconds to wait for remote processes to startup. If any process takes longer than this time period to startup, the program is aborted.	
IO=*keyword*	system	Keywords: {system, debug, files} system writes stderr and stdout according to the rules of the shell. debug redirects stdout and stderr on remote nodes to stderr on home node and prefixes each remote line with Process *x*:, where *x* is the number of the remote process. files redirects stderr to a file named clomp-<process id>-stderr and stdout to a file named clomp-<process-id>-stdout.	
[no-]heartbeat	heartbeat	Turn on / off the heartbeat mechanism for ensuring that all processes are alive.	
backing_store =*string*	/tmp	Sets the directory where swap space is allocated on each process for the sharable heap. This option is useful if /tmp resides on a partition that lacks sufficient space for the sharable swap requirements of an application.	
[no-]divert_twins	no-divert_twins	Tells the runtime to reserve memory for twin pages in the backing store directory. Ordinarily, twins are allocated space in the system swap file.	Use this option if your system swap space is not large enough to accommodate your application's memory usage.

6.2.3 Environment Variable Part

The effect of the environment variable part is to assign the value to the variable in the environment during program startup, but before any OpenMP constructs are executed. This environment variable assignment is done in the context of the shell you are currently using.

The following variables are not allowed in the kmp_cluster.ini file:

```
PATH
```

```
SHELL
LD_LIBRARY_PATH
```

6.3 Input / Output in a Cluster OpenMP* Program

6.3.1 Input Files

When reading input files with a Cluster OpenMP program, you must note that each node is running a separate operating system. This means that there is a separate file system for each node. Therefore, there are separate file descriptors and file position pointers on each node. This can make a Cluster OpenMP program behave differently than the equivalent OpenMP program. Reading a sequential file advances the file pointer within each node independently because the file control structures are private to a node.

This means that the common practice of opening a file in the serial part of the program by the master thread and then reading it in parallel within a parallel region will not work for a Cluster OpenMP program. The file would have to be opened on each node for this to work. Care must be taken to make sure that each file open specifies the proper path for the file on that node. If the user launching the program is in a different group on a remote node, then there could be permission problems accessing the file on that node.

A program reading `stdin` within a parallel region will fail unless the read is inside a **master** construct, since no attempt is made to propagate `stdin` to remote nodes. The home process is the only process that has access to `stdin`.

Reading an input file from the serial part of the program should behave as expected since that is done only on the home process by a single thread.

6.3.2 Output Files

When creating output files with a Cluster OpenMP program, you must note (just as mentioned in the previous section) that each node is running a separate operating system. If all nodes try to create a file with the same file in the same shared directory, there will be a conflict that will have to be handled by the file system. Output should be written to separate files whenever possible, or should be written in the serial part of the program to avoid these conflicts.

For information on the options regarding stdout and stderr, see section 12.3.

6.3.3 Mapping Files into Memory

Files may be mapped into memory with special Cluster OpenMP routines that mirror the `mmap` and `munmap` system calls. There are read/write and read-only versions of `mmap` and `munmap` available within the Cluster OpenMP run-time library. Mapping a file into memory and then reading the memory has the effect of reading the file. If the read/write version of `mmap` is used, unmapping the file has the effect of writing the memory image back out to the file. See Section 11.6, Memory Mapping Files for more information.

6.4 System Heartbeat

In a multi-process program, the Cluster OpenMP run-time system uses a *heartbeat* mechanism to allow it to exit all processes cleanly in the event of a program crash. The heartbeat mechanism is enabled by default, although it is possible to turn it off with the `--no-heartbeat` option, if that is desired. The heartbeat adds very little overhead in the common case because it merely has to keep track of whether it has sent a message to a particular process during a given time period (called the *heartbeat period)*. If it has, it does nothing. If it has not, then a special heartbeat message is sent to that process.

If process `a` has not heard from process `b` in a certain number of heartbeat periods, then process `a` assumes that process `b` crashed and process `a` exits. Using this mechanism, all processes will shut down if any

process fails.

The heartbeat period is set at ten seconds. The number of heartbeat periods to wait before the program is killed is based on the number of processes in the cluster:

```
Number-of-heartbeat-periods = ceiling(number-of-processes / 10) + 1
```

If the `number-of-processes` is equal to one, then the heartbeat is disabled.

If there is no heartbeat mechanism and one process fails, the rest of the processes eventually attempt to synchronize with the failed process, and the program hangs as a result. The hanging process typically sends a message to the failed process, waits for the reply, times out, re-sends the message, and so on forever. To remove these hanging processes, you must kill each one by hand.

6.5 Special Cases

6.5.1 Using ssh to launch a Cluster OpenMP program

The default behavior for Cluster OpenMP is to launch remote processes with the remote shell `rsh`. If a more secure environment is required, you can use `ssh` to launch remote processes by specifying the `--launch=ssh option`. It is your responsibility to make sure that proper authentication is established between the home process and all remote processes before the Cluster OpenMP program is run.

It is most convenient if you configure the system not to require a password for `ssh`.

6.5.2 Using a Cluster Queuing System

Cluster OpenMP has been used with the PBS batch system and should work with similar systems. For PBS, you write a PBS batch script that conceptually has two parts: a part that requests resources from the system for your job (number and type of nodes, amount of wall-clock time, etc), and a part that runs your program after the resources become allocated to the program.

The PBS batch script goes into a queue, causing it to wait until the requested resources are available. When they are available, a file containing the names of the nodes assigned to the job is created, and the `PBS-NODE-FILE` environment variable receives the name of the file. The batch script can then use this file to create a file to use with the `--hostfile` option in the `kmp_cluster.ini` file. The reason that the `PBS-NODE-FILE` file cannot be used directly with Cluster OpenMP is that PBS puts one node name in the file for each processor on the node. Leaving the file in this form for Cluster OpenMP would cause Cluster OpenMP to start a separate process on each processor of each node. So, you should remove duplicate node names with the following command:

```
uniq $PBS_NODEFILE > node-file
```

This results in a file called `node-file` which can then be used directly in the `kmp_cluster.ini` file with the `--hostfile` option.

There is a known bug with OpenPBS that requires you to break the parent-child relationship with the PBS shell by logging into the home node once a PBS job has started. Not doing this can cause the Cluster OpenMP job to hang.

This extra step requires that information must be packaged and passed to the new login. For example, the following code passes the current working directory and the command to execute the Cluster OpenMP program to the new login.

```
# PBS bug workaround
set dir=`pwd`
cat > foo.csh << EOF
#!/bin/csh
cd $dir
```

```
<command line here>
EOF
chmod 755 foo.csh
ssh $HOST foo.csh
```

For a description of the problem and a patch to fix it, see:

http://www-unix.mcs.anl.gov/openpbs/patches/ncsa_mlockall_ia64/README.txt

6.5.3 Sample PBS Batch File

```
#!/bin/csh
#
#   Sample Batch Script for a titan cluster job
#   Submit this script using the command: qsub
#
#   Use the "qstat" command to check the status of a job.
#
# The following are embedded QSUB options. The syntax is #PBS (the # does
# _not_  denote that the lines are commented out so do not remove).
#
# resource limits  walltime: maximum wall clock time (hh:mm:ss)
#                  nodes: number of 2-processor nodes
#                  ppn: how many processors per node to use (1 or 2)
#PBS -l walltime=24:00:00,nodes=16:ppn=2:himem
#
# queue name
#PBS -q dque
#
# export all my environment variables to the job
#PBS -V
#
# job name (default = name of script file)
#PBS -N program
#
# filename for standard output (default = <job_name>.o<job_id>)
#PBS -o program.out
#
# filename for standard error (default = <job_name>.e<job_id>)
#PBS -e program.err
#
# send mail when the job begins and ends (optional)
#PBS -m be
# End of embedded QSUB options

set echo                # echo commands before execution; use for debugging

# Create the scratch directory for the job and cd to it
setenv SCR `set_SCR`
if ($SCR != "") cd $SCR

mkdir $PBS_JOBID    # make subdirectory based on jobID and
cd $PBS_JOBID           # cd to this directory


# Get file input from directory in UniTree
cp /home/ac/user1/program.tar .
tar xvf program.tar

chmod u+x *

# Run the program on all nodes/processors requested by the job
./program.csh
```

7 Debugging a Cluster OpenMP* Program

This chapter describes some strategies for debugging Cluster OpenMP* applications using the Intel® debugger (idb), and two common debuggers: the GNU* debugger (gdb*) and the Etnus* debugger (TotalView*).

7.1 Before Debugging

Before you begin any debugging, turn off the heartbeat mechanism with the `--no-heartbeat` option in the `kmp_cluster.ini` file. Turning off the heartbeat ensures that the Cluster OpenMP library does not time-out and kill the processes. See section 6.4, *System Heartbeat* for more on heartbeats.

Debuggers normally handle various signals, including `SIGSEGV`. This can be a problem when debugging a Cluster OpenMP program, which uses the the `SIGSEGV` signal as part of its normal operation. Cluster OpenMP installs its own handler for `SIGSEGV` and uses it as part of its memory consistency protocol. Unless you instruct it to do otherwise, every `SIGSEGV` signal that Cluster OpenMP causes is sent to `the debugger`.

To solve this problem, you must tell each debugger not to intercept SIGSEGV. This is done differently in each debugger, as described in the following sections.

To catch `SIGSEGV` signals that are caused by program errors, Cluster OpenMP causes them to call a routine called `__itmk_segv_break`. Therefore, you can be notified of a program error causing a `SIGSEGV` by setting a breakpoint in that routine. The following sections provide instructions for doing so for each debugger.

7.2 Using the Intel® Debugger

You must tell the Intel debugger (`idb`) not to intercept `SIGSEGV` signals by creating a `.dbxinit` file in your home directory containing the line:

```
ignore segv
```

To set a breakpoint in the routine `__itmk_segv_break` to catch addressing errors, use the following idb command:

```
stop in __itmk_segv_break
```

You can start remote processes in separate windows by using the `KMP_CLUSTER_DEBUGGER` environment variable, setting its value to `idb`. Execute the program as follows:

```
idb <executable>
```

The remote processes will startup in `idb`. The `DISPLAY` environment variable in the `kmp_cluster.ini` file determines where the remote `idb` sessions can be viewed.

7.3 Using the gdb* Debugger

To cause gdb to ignore SIGSEGV signals, the `.gdbinit` file must be located in your home directory and must contain the following line:

```
handle SIGSEGV nostop noprint
```

You should set a break point in the routine `__itmk_segv_break`, to catch errors in your code that cause

SIGSEGVs. Use the following command:

```
break __itmk_segv_break
```

You can cause each remote process to enter the debugger in a separate window by using the `KMP_CLUSTER_DEBUGGER` environment variable. If the `KMP_CLUSTER_DEBUGGER` environment variable is set to `gdb` and you start the program on the home process with:

```
gdb <executable>
```

The remote processes also start up in the `gdb` debugger. If the `DISPLAY` environment variable is also set in the `kmp_cluster.ini` file, then each remote process starts in the debugger and opens an X Window* for the debugger session to wherever the `DISPLAY` is pointing to.

7.4 Using the Etnus* TotalView* Debugger

You can tell Totalview* to pass any `SIGSEGV` signals on to the program by creating a `.tvdrc` file in your home directory, containing the line:

```
dset TV::signal_handling_mode {Resend=SIGSEGV}
```

You should set a breakpoint in `__itmk_segv_break` so that TotalView can catch addressing errors.

Execute the program with Totalview* as follows:

```
totalview <executable>
```

Totalview automatically acquires the Cluster OpenMP processes. Follow the instructions provided in the TotalView documentation as if you are debugging an MPI program.

7.5 Redirecting I/O

A debugging method that is sometimes useful is to separate the I/O streams of the various processes. The default option for Cluster OpenMP is to enable the system to redirect the standard output and standard error streams. Therefore there is no way to distinguish between outputs from two different processes without modifying your program. Cluster OpenMP supplies the following three `kmp_cluster.ini` file options to modify this behavior:

```
--IO=system // This is the default option.

--IO=debug //
```

The `--IO=debug` option redirects standard error and standard output for remote processes to standard error on the home process. It prefixes remote output lines with:

```
Process <process-id>
```

Where `process-id` is a numerical identifier of the process. IDs are assigned starting at `0` in the order that the hosts appear on the command line.

`--IO=files` This option takes standard error and standard output from remote processes and redirects them to files named `clomp-<process-id>-stderr` and `clomp-<process-id>-stdout`, respectively.

These options are for handling I/O of remote processes only. The system always handles the I/O for the home process.

8 Evaluating Cluster OpenMP* Performance

This section describes a set of steps you can use to test the performance of Cluster OpenMP* for your program. The process includes using a script packaged with the tool that can help you determine whether a given OpenMP* program is suitable for running on a cluster with Cluster OpenMP, and how many nodes are appropriate.

Requirement

This section assumes that you have access to at least one multi-processor Itanium®-based or Intel® EM64T processor or compatible processor.

To evaluate your code's performance with Cluster OpenMP, do the following in order:

1. Ensure that your code gets good speedup with Cluster OpenMP* in one process.
To do this:

- Port the code to Cluster OpenMP by adding `sharable` directives wherever they are needed. See Chapter 4, *Porting Your Code*.
- Run the one-process Cluster OpenMP form of the program (compiled with `-cluster-openmp`) with one thread and record runtime.
- Run the one-process Cluster OpenMP form of the program with *n* threads, where *n* is the number of processors in one node.
- If the speedup achieved for *n* threads is not close to *n*, then the code is not suitable for Cluster OpenMP. Again, Speedup = Time(1 thread) / Time(*n* threads). Speedup should be approximately equal to *n*.

2. **Run the Cluster OpenMP code as multiple processes on one node.**
Build the code with `-cluster-openmp-profile` and make (at least) two runs on one or more nodes of the cluster, collecting the `.gvs` files produced from each run:

one process, one thread per process (options --process_threads=1 --processes=1)

k processes, one thread per process (options --process_threads=1 --processes=*k*)

The projections are most accurate for *k* nodes.
This step simulates a multi-node run by using multiple processes on one or more nodes. Over-subscribing a node may cause the program to run much slower than it would on *k* nodes, but in this step execution time is not being measured. Rather, statistics are being gathered about how many messages are exchanged by the processes and the volume of data being transmitted in those messages.
This step produces files with the suffix `*.gvs`.
It is recommended that you name these to identify the run they each represent, for instance `t1n1.gvs` (for 1 thread and 1 node), and `t1n2.gvs` (for one thread and two nodes). The files are the inputs to step 3.

3. **Run the suitability script**.
Run the suitability program, giving the `*.gvs` files from the previous step as input:

```
clomp_forecaster [ options ] t1n1.gvs t1nk.gvs . . .
```

where *options* is one of the options shown in the following table:

Option	Description
`-b bandwidth`	Specifies the maximum bandwidth for the interconnect being used (in Mb/s).

`-l latency`	Specifies the minimum round-trip latency for UDP on the interconnect being used (in microseconds). Can be calculated with the `clomp_getlatency` script in `<CLOMP tools dir>`. See 1.3, *Related Resources* for instructions on downloading this script and other examples from the web.
`-t target`	Specifies that the output should project speedup up to *target* nodes.
`-w`	Eliminates all warnings.

Table 4: Options for `clomp_forecaster`.

The output of this step is a comma-separated-values (`CSV`) file written to `stdout`.

Note

The forecast results are for the specific workload you ran. Results may vary with different workloads.

4. **Open the *.`CSV` file in a spreadsheet program such as Microsoft Excel***.
   The following image shows the output of a sample `*.CSV` file when opened in Microsoft* Excel:

	A	B	C	D	E	F	G	H	I
1	Intel Cluster OMP Performance Forecaster								
2	Max processes	4							
3	Min CPUs	2							
4	Timing Nodes	2							
5	Timing Runs	2							
6	Total Runs	3							
7	Single Node Run	1							
8	Target	16							
9	Latency (us)	30	450						
10	Bandwidth (Mb/s)	142.86	1000						
11	TIME								
12		1	2	3	4	5	6	7	8
13	MIN	11.45	5.73	3.82	2.86	2.29	1.91	1.64	1.43
14	MAX	11.45	5.73	3.83	2.87	2.3	1.93	1.66	1.45
15									
16									
17	SCALABILITY								
18	PERFECT	1	2	3	4	5	6	7	8
19	BEST	1	1.9999	2.9995	3.9989	4.9978	5.9962	6.994	7.991
20	WORST	1	1.9979	2.993	3.9835	4.9679	5.9446	6.9123	7.8695
21									

Figure 8-1: Sample out put .CSV file. The numbers in rows 12 and 18 represent the number of nodes. The values MIN and MAX are estimates of execution time (in seconds) for executing the program on the indicated number of nodes (1 through 8). The values BEST and WORST are estimates of scalability speedup, which is the ratio of execution time on 1 node to the execution time on the corresponding number of nodes.

5. **Produce a chart using the data in the SCALABLITY section of the table.**
   Select the data in the SCALABILITY section of the table, typically rows 18-20 and select **Insert** > **Chart** to produce a chart such as the following:

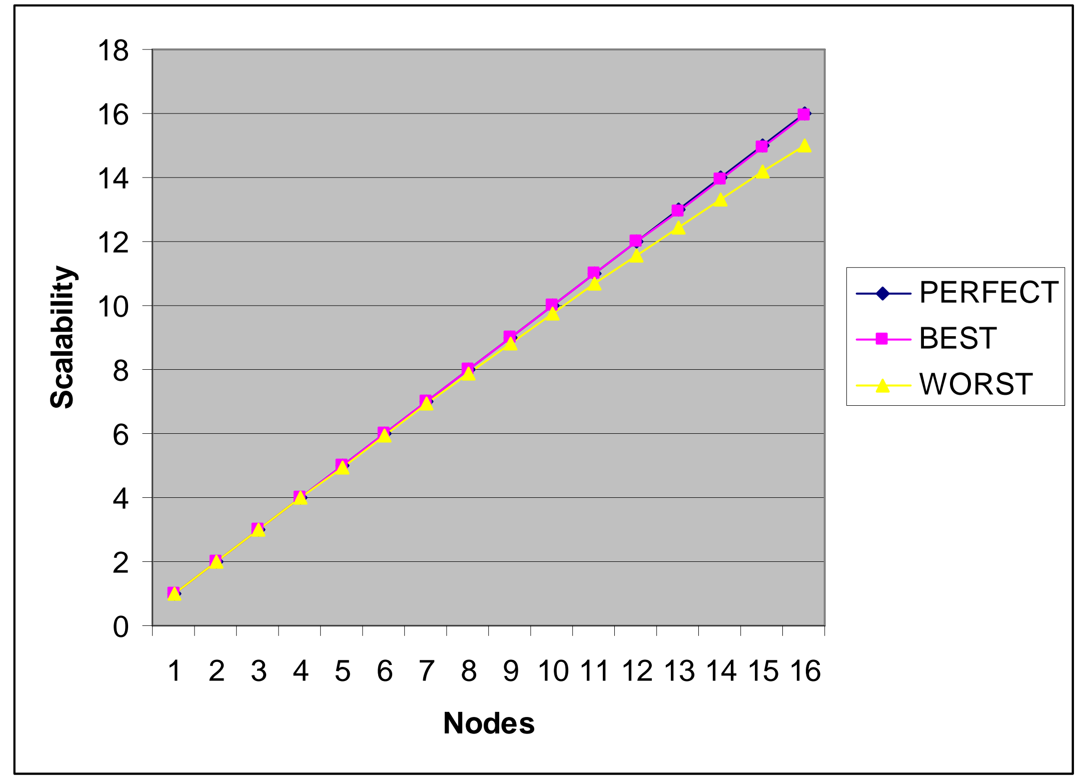


Figure 8-2: Chart showing the predicted scalability speedup using Cluster OpenMP*. In this example, the chart indicates that the application should scale extremely well with Cluster OpenMP. Worst-case performance is shown as a speedup of about 15 on 16 nodes.

6. **Determine the optimum number of nodes for your code.**
   At this point, you should decide based on cost/performance criteria how many nodes are right for you. Choosing the most appropriate number of nodes to use is probably workload dependent. The predictions are for applications using the stats-enabled library, `libclusterguide_stats.so`, typically installed in the `<compiler install directory>/lib`. This library has up to 10% overhead relative to the non-stats-enabled library.

Notes

- Actual performance is usually between BEST and WORST cases.
- Actual time is usually close to the average of the HIGH and LOW scalability predictions.
- The predictions are for applications using the stats-enabled library, `libclusterguide_stats.so`, typically installed in the `<compiler install directory>/lib`. This library has up to 10% overhead relative to the non-stats-enabled library.
- To improve performance, use Intel® Thread Profiler to tune your code. See Section 10.2.2, *Intel® Thread Profiler* for details.

---

9 OpenMP* Usage with Cluster OpenMP*

This chapter presents a program development model and describes OpenMP* considerations for working with Cluster OpenMP*.

9.1 Program Development for Cluster OpenMP*

This section presents an idealized program development model for Cluster OpenMP*. The steps described here are not required, but are recommended.

9.1.1 Design the Program as a Parallel Program

If you have the luxury of writing the program from scratch, it is important to design it with OpenMP parallelism in mind. A planned OpenMP program may differ significantly from a naïve serial program that is parallelized by adding OpenMP directives. Write your program according to the following guidelines:

- Make parallel regions as large as possible
- Use private data as much as possible
- Do as little synchronization as possible

9.1.2 Write the OpenMP* Program

To write the OpenMP* Program:

1. **Design a parallel program**.
Pay special attention to the tasks in your design that can be done in parallel. The ideal parallel application is one that has no serial code at all. However, most interesting codes require some synchronization and communication between threads. For best performance, synchronization and communication should be kept to a minimum. You can use various techniques to reduce synchronization and communication. For instance, instead of making a calculation on one thread and sending the result to the other threads, it may be faster to do the calculation redundantly on each thread. Also, avoid making the program depend on using a certain number of threads, or doing special things on certain threads other than the master thread. This strategy enables the program to run on any machine configuration.

2. **Debug the code serially**.
If you have an existing serial code, start by debugging it as you normally would.
Or, compile your OpenMP* Program without using `-openmp` or `-cluster-openmp` options to produce a serial program. Debug this program using a serial debugger until this serial version of the code is working.

3. **Debug the parallel form of the code**.
Add code as appropriate to parallelize the code. As in step 1, avoid making the program depend on using a certain number of threads, and avoid doing special things on threads other than the master thread. Compile using the `-openmp` option to produce the parallel form of the program. Debug until it works. You can use Intel® Thread Checker to help you identify threading inconsistencies in your parallel code.

4. **Mark sharable variables**.
Ensure that all variables that are in `firstprivate`, `lastprivate`, `reduction`, or `shared` clauses in the program are sharable.
Also, all variables that are shared by default in any parallel region must be sharable. Any of these that are declared in the same routine in which they are used in a parallel region are made sharable automatically by the compiler. You must mark others with `sharable` directives or by specifying an appropriate compiler option (for Fortran).

5. **Use the compiler to locate where sharable directives are needed**.
Compile your code using the appropriate Intel® Compiler with the`-clomp-sharable-propagation` and `-ipo` options. View the resulting output log contents to identify places in your code where sharable directives are needed and mark the identified variables with `sharable` directives.

6. **Build and run as a Cluster OpenMP program**.
Compile with the `-cluster-openmp` option. The program should execute correctly.

7. **Debug the program.**
If the program does not run correctly, follow the procedures outlined in section 4.4, *Using the Disjoint Heap* for finding variables that need to be made sharable. When done debugging, go back to step 6.

9.2 Combining OpenMP* with Cluster OpenMP*

Some libraries such as the Intel® Math Kernel Library and the Intel® Integrated Performance Primitives use OpenMP* directives for parallelism. These libraries are not designed to execute across a cluster. When using such libraries in a Cluster OpenMP* program, you must link the program with the Cluster OpenMP* runtime library instead of the OpenMP* runtime library.

The Cluster OpenMP* runtime library detects when a directive has not been compiled with `–cluster-`

`openmp`. It runs a parallel region with the number of threads specified by the `--process_threads` option if the region is encountered outside of any Cluster OpenMP* parallel region. Otherwise, it serializes the parallel region.

When linking such a library with a Cluster OpenMP* program, replace the OpenMP* option with the corresponding Cluster OpenMP* option as follows:

OpenMP* Option	Replace with Cluster OpenMP* Option
`-openmp`	`-cluster-openmp`
`-lguide`	`-lclusterguide`

To ensure that you linked with the correct library, use the `KMP_VERSION` environment variable as outlined in section 9.5, *Cluster OpenMP* Environment Variables*. If you linked correctly with Cluster OpenMP*, the version output should show `Intel(R) Cluster OMP`.

If not, the OpenMP* runtime library could be statically linked into the library you are trying to use. In this the case, compile the program to produce object files and link explicitly as follows:

```
<intel compiler> <obj files> -o <exe> -lclusterguide -l<other library>
```

As long as `-lclusterguide` appears before the other library on the link line, the OpenMP* runtime library symbols will be resolved from the Cluster OpenMP* runtime library instead of from the other library you are trying to use. To verify, use `KMP_VERSION` to make sure you linked with Cluster OpenMP*. If you linked correctly with Cluster OpenMP*, the version output should show `Intel(R) Cluster OMP`.

9.3 OpenMP* Implementation-Defined Behaviors in Cluster OpenMP*

The OpenMP* specification at www.openmp.org requires an implementation to document its behavior in a certain set of cases. This section documents these behaviors for Cluster OpenMP.

9.3.1 Number of Threads to Use for a Parallel Region

The number of OpenMP threads that are started at the beginning of a given program is the value of the `--omp_num_threads` option after proper defaults are applied. This is the maximum number of threads that can be used by any parallel region in the program.

A parallel region can use fewer than the maximum number of threads by specifying a value for the `OMP_NUM_THREADS` environment variable, or by using the `omp_set_num_threads()` routine.

9.3.2 Number of Processors

The number of processors reported by `omp_get_num_procs()` is the sum of the number of processors on all nodes.

9.3.3 Creating Teams of Threads

Cluster OpenMP does not support nested parallelism. If an inner parallel region is encountered by a thread while a parallel region is already active, then the inner parallel region is serialized and executed by a team of one thread.

9.3.4 Schedule(RUNTIME)

If the OMP_SCHEDULE environment variable is not set, the default schedule is static.

9.3.5 Various Defaults

In the absence of the schedule clause, Cluster OpenMP uses static scheduling.

Option	Description
ATOMIC	Cluster OpenMP replaces all atomic constructs with critical constructs with the same unique name.
omp_get_num_threads	If the number of threads has not been set by you, Cluster OpenMP sets it to the maximum number of threads (the product of the number of processes and the number of threads per process).
omp_set_dynamic	The default for dynamic thread adjustment is that it is disabled.
omp_set_nested	Cluster OpenMP supports only one level of parallelism.
OMP_SCHEDULE	The default schedule if OMP_SCHEDULE is not defined is static.
OMP_NUM_THREADS	If OMP_NUM_THREADS has not been defined by you, Cluster OpenMP uses the maximum number of threads (the value of the –omp_num_threads option after all defaults are evaluated).
OMP_DYNAMIC	The default for dynamic thread adjustment is that it is disabled.

9.3.6 Granularity of Data

The smallest unit of data that Cluster OpenMP can operate on in sharable memory is four bytes. This means that all sharable variables must be at least four bytes in length. Therefore the following parallel loop does not execute as expected:

```
int i;

char achars[1000], bchars[1000];

#pragma omp parallel for

for (i=0; i<N; i++) {

    achars[i] = bchars[i];

}
```

For a work-around to this limitation, see Section 11.3, *Granularity of a Sharable Memory Access.*

9.3.7 Intel Extension Routines/Functions

Intel's support for OpenMP* includes additional functions which provide a fast per-thread heap implementation. These functions are documented in the Intel® C++ Compiler documentation. They include kmp_malloc, kmp_calloc, kmp_realloc and kmp_free. In Cluster OpenMP these functions continue to allocate store with the same accessibility as malloc, providing a local, process-accessible store. They do not allocate sharable store. As a result blocks allocated by these routines can only be freed by threads which are running in the same process as the thread which allocated the store.

If sharable store allocation is required you must replace these allocation calls with calls to the corresponding kmp_sharable_ function.

9.4Cluster OpenMP* Macros

A given program can check to determine whether it was compiled with the –cluster-openmp or -cluster-openmp-profile options by checking whether the _CLUSTER_OPENMP macro has a value. If it does, then one of the Cluster OpenMP* options was used.

9.5 Cluster OpenMP* Environment Variables

The following table defines a set of environment variables you can set from the shell to control the behavior of a Cluster OpenMP* program.

Variable name	Value	Default Value	Description
KMP_STACKSIZE	*size* [K | M]	1M	Stacksize for subordinate threads in each Cluster OpenMP process, in kilobytes (K) or megabytes (M). The stacksize of each principle thread is determined by your original shell stack size.
KMP_SHARABLE_STACKSIZE	*size* [K / M]	1M	Size of stack to be used for allocation of stack-allocated sharable data on each OpenMP thread. The value for KMP_STACKSIZE is independent of the value for KMP_SHARABLE_STACKSIZE.
KMP_STATSFILE	*filename*	guide.gvs	Filename to use for the statistics file (build with -cluster-openmp-profile)
KMP_CLUSTER_DEBUGGER	*Name*	none	Debugger executable name (must be in your path).
KMP_WARNINGS	0 or off 1 or on	on	0 turns off run-time warnings from the Cluster OpenMP* run-time library.
KMP_SHARABLE_WARNINGS	0 or off 1 or on	off	1 turns on warnings for variables that may be shared in parallel regions but are not sharable.
KMP_CLUSTER_SETTINGS	None	None	Causes system to output the current values of all options specifiable in kmp_cluster.ini and all environment variable values.
KMP_CLUSTER_PATH	None	Nome directory	In case there is no kmp_cluster.ini file in the current working directory where you start the Cluster OpenMP* program, specifies a path along which to find the first instance of a .kmp_cluster file.
KMP_CLUSTER_HELP	None	None	Causes system to output text describing the use of the kmp_cluster.ini file options, then exits.
KMP_VERSION	None	None	Causes system to dump its version information at run-time
KMP_DISJOINT_HEAPSIZE	*size* [K / M]	None	Enable the disjoint heap porting mechanism. See Section 4.4 *Using the Disjoint Heap.* Causes diagnostic information to appear at runtime. Minimum assigned value is 2K.

Table 5: Cluster OpenMP* Environment Variables

9.6 Cluster OpenMP* API Routines

The following table defines a collection of API routines that you can call from inside your code to control ClusterOMP* program behavior.

API Routine	Description
void *kmp_sharable_malloc(size_t *size*)	Allocate sharable memory space.
void *kmp_aligned_sharable_malloc(size_t *size*)	Allocate sharable memory space aligned on a page boundary.
void *kmp_sharable_calloc(size_t *n*, size_t *size*)	Allocate sharable memory space for an array of *n* (each of size size) and zero it.
void *kmp_sharable_realloc(void **ptr*, size_t	De-allocates previously allocated sharable

`size)`	memory space (pointed to by *ptr* and allocates a new block of size *size.*
`void kmp_sharable_free(void *ptr)`	Free sharable memory space.
`int kmp_private_mmap(char *filename, size_t *len, void ** addr)`	Read-only version of `mmap`. See also Section 11.6, *Memory Mapping Files.*
`int kmp_sharable_mmap(char *filename, size_t *len, void **addr)`	Read/write version of `mmap`.
`int kmp_private_munmap(void *start);`	Read-only version of `munmap`. See also Section 11.6, *Memory Mapping Files.*
`int kmp_sharable_munmap(void * start);`	Read/write version of `munmap`.
`void kmp_lock_cond_wait(omp_lock_t *lock)`	Wait on a condition.
`void kmp_lock_cond_signal(omp_lock_t *lock)`	Signal a condition.
`void kmp_lock_cond_broadcast(omp_lock_t *lock)`	Broadcast a condition.
`void kmp_nest_lock_cond_wait(omp_nest_lock_t *lock)`	Wait on a condition with nested lock.
`kmp_nest_lock_cond_signal(omp_nest_lock_t *lock)`	Signal a condition with a nested lock.
`kmp_nest_lock_cond_broadcast(omp_nest_lock_t *lock)`	Broadcast a condition with a nested lock.
`void  kmp_set_warnings_on( void )`	Enable run-time warnings.
`void  kmp_set_warnings_off( void )`	Disable run-time warnings.
`omp_int_t  kmp_get_process_num( void )`	Return the process number of the current process.
`omp_int_t  kmp_get_num_processes( void )`	Return the number of processes involved in the computation.
`omp_int_t  kmp_get_process_thread_num( void )`	Return the thread number of the current thread with respect to the current process.

Table 6: Cluster OpenMP* API Routines

9.7 Allocating Sharable Memory at Run-Time

This section describes routines you can use that are specific to C, C++ or Fortran programming to help allocate sharable memory at runtime.

Allocating sharable memory at run-time is possible in C, C++ and Fortran. In C and C++, you can call one of two `malloc`-like routines:

```
void * kmp_sharable_malloc( int size);

void * kmp_aligned_sharable_malloc( int size );
```

These routines both allocate the given number of bytes out of the sharable memory and return the address. The `_aligned_` version allocates memory that is guaranteed to start at a page boundary, which may reduce false sharing at runtime. Memory allocated by one of these routines must be deallocated with:

```
void kmp_sharable_free(void *ptr)
```

In Fortran, the `ALLOCATE` statement will allocate a variable declared with the *sharable* directive in sharable memory. For example:

```
integer, allocatable :: x(:)

!dir$ omp sharable(x)

    allocate(x(500))     ! allocates x in sharable memory
```

9.7.1 C++ Sharable Allocation

This section describes sharable allocation requirements for C++ applications.

9.7.1.1 Header Files

All of the definitions required for using shared allocation in C++ are included in the file `kmp_sharable.h`. Use:

```
#include <kmp_sharable.h>
```

9.7.1.2 Creating Sharable Dynamically Allocated Objects

If you determine that only some objects of a given class need to be sharably allocated, then you must modify the allocation points of the objects which need to be sharable.

Suppose you are allocating objects of `class foo`. If your initial code is:-

```
foo * fp = new foo (10);
```

convert this to code which allocates a sharable `foo`, as follows:-

```
foo * fp = new kmp_sharable foo (10);
```

Adding the kmp_sharable macro ensures that your code continues to compile correctly when it is not compiled with Cluster OpenMP enabled. When not compiling with Cluster OpenMP*, the `kmp_sharable` macro expands to nothing . When compiling with Cluster OpenMP*, this macro inserts a bracketed expression which invokes a different `operator new`.

For example, if the initial code is:

```
foo * fp = new foo [20];
```

Change the code to include the kmp_sharable macro call as follows:

```
foo * fp = new kmp_sharable foo [20];
```

Note

Implementing a `new kmp_sharable` requires the overloading of the global operator new. If your code already replaces `::operator new` then you need to resolve the conflict.

9.7.1.3 Creating a Class of Sharable Allocated Objects

If you determine that all dynamically allocated objects of a particular class should be allocated as sharable, you can modify the class declaration to apply to all objects within it instead of modifying all of the points at which objects are allocated.

For example if the initial class declaration is:

```
class foo : public foo_base
{
// … contents of class foo
};
```

Change the declaration to allocate all objects as sharable as follows:

```
class foo : public foo_base, public kmp_sharable_base
{
```

```
// … contents of class foo
};
```

Note

Implementing `kmp_sharable_base` provides the derived class with `operator new` and `operator delete` methods which use `kmp_sharable_malloc`. If your class is already providing its own `operator new` and `operator delete` then you need to reconsider how to manage sharable store allocation for the class..

9.7.1.4 Sharable STL Containers

STL containers add another level of complication to programming since the container has two separate store allocations to manage:

1. The store allocated for the container object itself. To allocate sharably, add the `kmp_sharable` macro after the `new` command.
2. The space dynamically allocated internally by the container class to hold its contents. To cause that to be allocated sharably, pass in an allocator class to the STL container instantiation.

If the initial allocation is: -

```
std::vector<int> * vp = new std::vector<int>;
```

Make it sharable as follows:

```
std::vector<int,kmp_sharable_allocator<int> > * vp = new
kmp_sharable std::vector<int, kmp_sharable_allocator<int> >;
```

The kmp_sharable_allocator cause the vector's contents to be allocated in sharable space, while the `new kmp_sharable` causes the vector object itself to be allocated in sharable space.

Since the allocator is a part of the vector's type, you must also modify any iterators which iterate over the vector so that they are aware of the non-default allocator.

9.7.1.5 Complicated STL Containers

Some of the more complicated STL containers, such as `std::map`, use additional template arguments before the allocator, as in the following example:

```
std::map<int,float> * ifm = new std::map<int, float>;
```

Change the container to dynamically allocate sharable variables as follows:

```
std::map<int,float,std::less<int>,kmp_sharable_allocator<float> > *
ifm = new kmp_sharable
std::map<int,float,std::less<int>,kmp_sharable_allocator<float> >;
```

10 Related Tools

This chapter describes additional tools that can help you get the most out of Cluster OpenMP*. The following sections include specific suggestions for using these tools with Cluster OpenMP*. The tools are available from http://www.intel.com/cd/software/products/asmo-na/eng/index.htm.
For complete details, consult each product's documentation.

10.1 Intel® Compiler

The Intel® Compiler version 9.1 or later must be installed in order to use Cluster OpenMP*.

10.2 Intel® Threading Tools

Use the Intel® Threading Tools to speed and simplify the development and maintenance of threaded applications.

10.2.1 Intel® Thread Checker

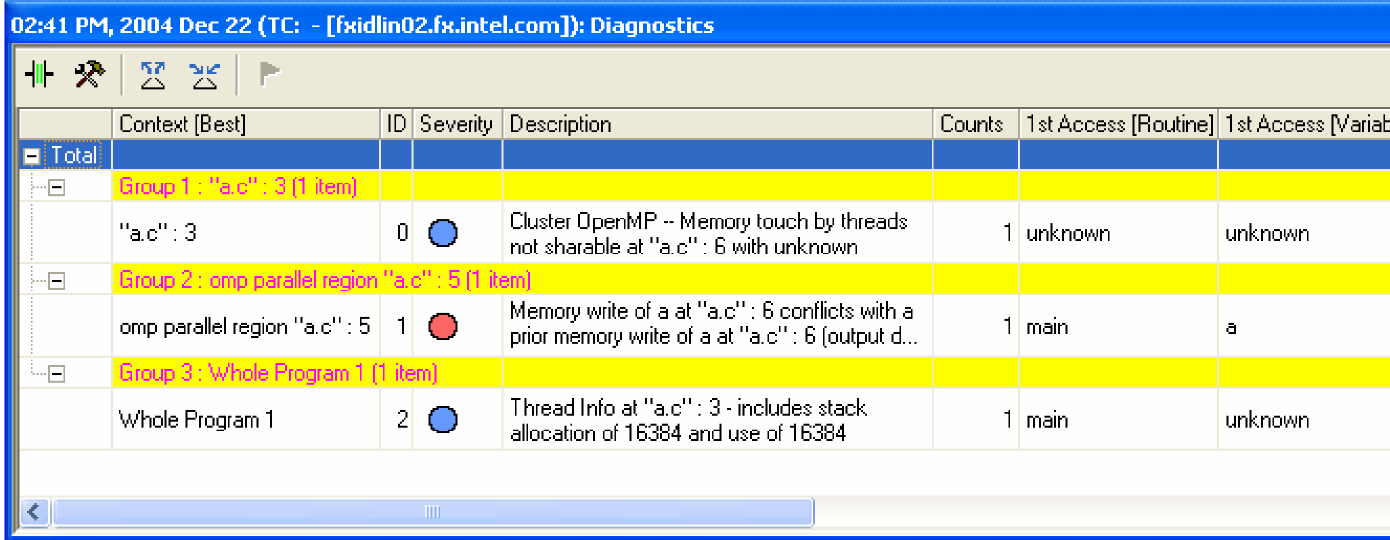
Intel® Thread Checker saves valuable time otherwise spent searching for hard-to-find threading errors. You can use Thread Checker to help you identify variables in your code to make sharable, including variables not identified by the compiler.

To produce Thread Checker diagnostics that indicate variables to make sharable, do the following:

To use Intel® Thread Checker to discover sharable errors, you perform normal remote data collection using a special environment variable that causes Thread Checker to identify sharable variables.

Consult the Intel® Thread Checker documentation for detailed information about the following steps:

1. Start the `ittserver` which is installed as part of the Intel® Thread Checker package for Linux*.
2. On your Windows* machine running Intel® Thread Checker, follow the instructions in the Thread Checker **Help** for remote data collection on Linux*. Make sure to compile your application with `-tcheck` and `-cluster-openmp` options in Linux.
3. Right-click on the Activity in the Tuning Browser and select **Modify Activity**.
4. Select your Cluster OpenMP* program from the list of **Application Modules/Profiles** and select **Configure…**.
5. In the **Application/Module Profile Configuration** dialog box, click **Advanced**.
6. In the **Application to Launch Options** dialog box:
    a. Uncheck **Use default environment**.
    b. Enter the following environment variable:
    `KDD_OPTIONS=cluster_openmp,mem=4,verbose`
1. Click **OK** to close all open dialog boxes.
2. Run your application remotely as usual. Thread Checker identifies sharable errors with diagnostics that read **Cluster OMP: memory touch by threads not sharable at:**.



	Context [Best]	ID	Severity	Description	Counts	1st Access [Routine]	1st Access [Variabl
Total							
	Group 1 : "a.c" : 3 (1 item)						
	"a.c" : 3	0		Cluster OpenMP -- Memory touch by threads not sharable at "a.c" : 6 with unknown	1	unknown	unknown
	Group 2 : omp parallel region "a.c" : 5 (1 item)						
	omp parallel region "a.c" : 5	1		Memory write of a at "a.c" : 6 conflicts with a prior memory write of a at "a.c" : 6 (output d...	1	main	a
	Group 3 : Whole Program 1 (1 item)						
	Whole Program 1	2		Thread Info at "a.c" : 3 - includes stack allocation of 16384 and use of 16384	1	main	unknown

Figure 10-1: Sharable diagnostics identified by Intel(R) Thread Checker

In the example shown in Figure 10-1, Thread Checker identified two problems in the code:

1) There is a race on the update of **a** in the parallel region in **Group 2**.
2) The variable **a** should be declared sharable in **Group 1**.

Notes

For complete instructions on using Intel® Thread Checker, see that product's online **Help**.

10.2.2 Intel® Thread Profiler

Intel® Thread Profiler locates performance issues in your threaded code.

Using Intel® Thread Profiler to find performance issues in Cluster OpenMP* programs is very similar to using Thread Profiler on traditional multi-threaded codes.

To use Thread Profiler with your Cluster OpenMP program:

1. Compile your Cluster OMP application with the `-cluster-openmp-profile` option to obtain a version of the run-time library that collects statistics.
2. Run the application as usual, but using a reduced dataset or iteration space if possible since statistics collection slow the application down.
   By default, Thread Profiler produces a `guide.gvs` file in the current working directory. You can change this default using the `KMP_STATSFILE` environment variable.
3. Open the `guide.gvs` file in Intel® Thread Profiler on your Windows* client to view performance data.

Notes

For complete instructions on using Intel® Thread Profiler, see that product's online **Help**.

Thread Profiler does not report additional information about Cluster OpenMP statistics. See Section 8, *Evaluating Cluster OpenMP**, for details about using the `*.gvs` files for analyzing application communication overheads.

11 Technical Issues

This chapter provides technical details on Cluster OpenMP*.

11.1 How a Cluster OpenMP* Program Works

In the following description, the assumption is that the Cluster OpenMP program is running on a cluster with one process per node.

Each sharable page is represented by a set of associated pages, one on each process. Each such page is at the same virtual address within each process. The access protection of each sharable page is managed according to a protocol within each process, based on the accesses made to the page by that process, and the accesses made to the associated pages on the other processes.

The basic idea of the protocol is that whenever a page is not fully up-to-date with respect to the associated pages on other processes, the page is protected against reading and writing. Then, whenever your program accesses the page in any way, the protection is violated, the Cluster OpenMP library gets notified of the protection failure, and it sends messages to the other processes to get the current up-to-date version of the page. When the data is received from the other processes and the page is brought up-to-date, the protection is removed, the instruction that accessed the page in the first place is re-started and this time the access succeeds.

In order for each process to know which other processes modified which pages, information about the modifications is exchanged between the processes. At cross-thread synchronizations (barriers and lock synchronizations), information is exchanged about which pages were modified since the last cross-thread synchronization. This information is in the form of a set of *write notices*. A write notice gives the page number that a process wrote to and the *vector time* stamp of the write.

The vector time stamp is an array of synchronization epoch values, one per process in the system. A particular process increments its epoch value each time it synchronizes with at least one other process. The epoch values on that process for all the other processes in the system represent the epoch values of each at the last synchronization point between that process and the current process. The vector time stamps are associated with a sharable page to show the state of the information on that page with respect to each process. This enables the process to check to see whether it needs updated information from a given process for a given page.

At each barrier synchronization point, as a barrier arrival message, each process sends write notices about which sharable pages it or other processes have modified, since the last synchronization point, to the master process. Then the master process combines all the write notices, determines which write notices are covered by which other write notices, and as a barrier departure message, sends the combined set of write notices to each remote process.

When a page is protected from any access, and a read is done to an address on the page, a SIGSEGV occurs and is caught by the `SIGSEGV` handler in the Cluster OpenMP run-time library. The handler checks the write notices it has stored for that page and then requests updated information from each process from whom it has a write notice. In most cases, the updated information comes in the form of a *diff*. A diff requires a comparison between the current information stored in a page and an old copy or *twin page* that the process made at some point in the past. So, only the locations that have changed since the twin was made are sent to the requesting process. The request for this diff information is call a *diff request.*

Each process keeps a database of write notice and diff information, sorted by vector time and organized by page. The diffs are stored so that the diff only has to be calculated once. After the diff is stored, the associated twin can be deallocated. Any future diff request for the current vector time and page are retrieved from the database and transmitted.

While executing a barrier synchronization, if this write notice database gets too large on any process, a *repo*

is done. A repo is the mechanism where each process is able to delete its write notice database, by bringing each page up-to-date on some process. The processes agree on which process should be considered to be the *owner* for each page. Each process brings the pages for which it is the owner up-to-date during the repo, and marks those pages *private*. The pages for which a process is not the owner are marked *empty* (and protected against reading and writing), but the owner for the page is remembered. Then, immediately after the repo, on a process's first access to a particular *empty* page, the process sends a page-request to the page's owner to retrieve the fully up-to-date page.

11.2 The Threads in a Cluster OpenMP* Program

This section describes the different kinds of threads used in a Cluster OpenMP* program.

11.2.1 OpenMP* Threads

The thread that starts the execution of a Cluster OpenMP program is called the *master thread*. The rest of the threads started in a parallel region are called *worker threads*. Nested parallel regions are serialized (at the present time) and the thread that executes the serialized parallel region becomes the *master thread* of the team of one that executes that serialized region.

The threads of each process are divided into two kinds of threads. The thread that initiates processing on the process is called the *principal* thread for the process and the other threads are the *subordinate* threads for the process. So, the OpenMP master thread is the principal thread on the home process. The OpenMP worker threads are all the subordinate threads on all the processes, plus the principal threads on all the remote processes.

The OpenMP threads are also referred to as *top-half* threads on any given process.

11.2.2 DVSM Support Threads

Every process in a Cluster OpenMP program has a set of *bottom-half* threads (part of the DVSM mechanism) that handles asynchronous communication chores for the process. That is, the bottom-half threads are activated by messages that come to the process from other processes. When a thread *k* on one process sends a message to a second process, the message is handled on the second process by a bottom-half thread.

Additional threads are used to handle mundane chores. If you use the `--IO=debug option` (see Chapter 13, *Reference*), the home process uses an *output listener* thread to handle text written to `stdout` by all remote processes. Also, the heartbeat operation, if enabled, is handled by its own thread on each process.

11.3 Granularity of a Sharable Memory Access

The smallest size for a memory access operation that can be kept consistent automatically is four bytes. However, consistency **can** be guaranteed for accesses of less than four bytes if the access is placed inside a critical section. For example, the following loop will not work for Cluster OpenMP:

```
char buffer[SIZE];

#pragma omp parallel for

for (i=0; i<SIZE; i++)

{

    buffer[i] = …;
```

```
}
```

The example must be modified as follows:

```
char buffer[SIZE];
#pragma omp parallel for
for (i=0; i<SIZE; i++)
{
 #pragma omp critical
   {
     buffer[i] = …;
   }
}
```

Note that such a parallel loop has poor performance with Cluster OpenMP.

11.4 Socket Connections Between Processes

At program startup, each bottom-half thread connects a socket to the same numbered thread on each other process for sending requests. So,

Number-of-sockets-on-one-process = 2*(number-of-threads/process)*(number-of-processes – 1)

The total number of socket connections between all processes of the cluster is

Number-of-connections = (number-of-threads/process)*(number-of-processes – 1)*(number-of-processes)

11.5 Using X Window System* Technology with a Cluster OpenMP* Program

If you want to use X Window System* calls within a Cluster OpenMP program, set the `DISPLAY` environment variable appropriately in the `kmp_cluster.ini` file and run the program. The `DISPLAY` environment variable is automatically propagated to the remote processes, so they will receive the same value. In this way, all processes are capable of starting an X Window System* session on the same display.

11.6 Memory Mapping Files

The `mmap` system call maps a file into the address space of the program. Since Cluster OpenMP employs `mmap` internally, it must be used with care. Cluster OpenMP supplies replacement routines for `mmap` and `munmap` that are compatible with the underlying DVSM mechanism. Two types of `mmap` are available: read/write and read-only.

The read/write version maps the entire file into the sharable memory on the home process. The normal DVSM mechanism propagates the information to remote processes as different parts of the file are read and written by different threads. When the associated `munmap` routine is called, the current memory image of

the file is written back to the file.

The read-only version of `mmap` maps the entire file into process-private memory, starting at the same virtual address on each process. Since process-private memory is used, no attempt is made to keep the copies of the file consistent, and nothing is written back to the file when the associated `munmap` routine is called. Nothing prevents the program from writing to the mapped version of the data, but any changes will be lost.

The memory mapping routines are:

Read/write version:

```
int  kmp_sharable_mmap(char * filename, size_t * len,  void **
addr);

int kmp_sharable_munmap(void * start);
```

Read-only version:

```
int kmp_private_mmap(char * filename, size_t * len, void ** addr);

int kmp_private_munmap(void *start);
```

The return values of each of these routines are `0` for success and `-1` for failure. If an `mmap` routine returns success, then it also returns the length of the file in bytes in the *`len`* parameter and the starting address in the *`addr`* parameter.

All of these calls must be made from the serial part of the program. Any use of these routines from a parallel region is unsupported.

11.7 Tips and Tricks

11.7.1 Making Assumed-shape Variables Private

An assumed shape array may be used in a private clause in an OpenMP program. If it is, however, the variable in the outer scope that the private variable is modeled on must be declared sharable, because the information relating to the shape of the array must be available across all nodes of the system.. The array must be declared sharable at its declaration point. For example, consider the following code:

```
    interface
       subroutine B ( A )
          integer A(:)
       end subroutine B
    end interface

    integer A(100000)
!dir$ omp sharable(A)

    call B(A)

    . . .
```

```
    subroutine B( A )
    integer  A(:)

!$omp parallel private(A)

    . . .
```

In this situation, if A were not declared to be sharable, then the information about its shape would not be available to all nodes of the cluster. The sharable directive is necessary to make this work. Without it, a variable of the proper shape could not be made by all threads.

11.7.2 Missing Space on Partition Where /tmp is Allocated

If you notice that the partition where `/tmp` is allocated is running low on space and the lack of space does not seem to be due to files residing on that partition, it is possible that there are Cluster OpenMP programs that are either still running on the cluster, or are halted in the debugger. If you kill all such programs, the space should reappear in the partition.

This is due to the anonymous space used for swap space in the `/tmp` partition by Cluster OpenMP.

12 Configuring a Cluster

This section provides instructions for configuring a cluster that you can use with Cluster OpenMP*. The instructions include both general steps and steps that are specific to configuring a cluster for Cluster OpenMP*.

Note

In most cases, you do not have to do anything special to prepare your cluster for use with Cluster OpenMP*. Special configuration is required if you intend to work with X Windows*. See 12.4, *Gateway Configuration* for details.

Configuring a cluster for the purpose of using Cluster OpenMP involves making a few decisions about how the cluster will be administered and how it fits in with the computing environment. One node of the cluster is distinguished as the *head node*. The other nodes of the cluster are referred to as the *compute nodes*.

1. **Decide how the cluster will appear in the external environment.** Will all cluster nodes be visible to external machines, or will only the head node be visible? If all nodes are visible to external machines, then the cluster becomes much more accessible to external users and the cluster is more likely to be disturbed during the run of a Cluster OpenMP program.
2. **Decide how to manage user accounts within the cluster.** Will all user accounts for the compute nodes be exported from outside, or will user accounts on the compute nodes be exported from the head node? Note that the user account that launches a Cluster OpenMP program must exist on all nodes.
3. **Decide how to organize the file systems on the compute nodes.** Will the head node export directories to the compute nodes, while accessing its directories from the outer domain? Or will the compute nodes access directories from the outer domain using a hub uplink or by using the head node as a gateway? It is recommended that the head node export directories to the compute nodes because the directory path to the executable and the Cluster OpenMP library on the home node must exist on the remote nodes.

12.1 Preliminary Setup

The following are general instructions for setting up a cluster. If you already have a cluster set up, you can skip to the next section.

These instructions assume that the outer domain in which the cluster is being setup is called *outerdomain.company.com* and the yellow pages server for the outer domain is called *ypserver001*.

1. Distribute a `/etc/hosts` file.

- Include cluster IP addresses and hostnames to all nodes in the cluster. Use separate names for the IP address of the head node's external ethernet port and the name for the internal ethernet port. For example: `headnode-external` and `headnode`.
- If you decided not to resolve host names via DNS or NIS, include entries for mounted file systems.
- To prevent problems with `rsh` and `X Windows`, ensure that the `127.0.0.1` line is filled out as follows on each host: `127.0.0.1 localhost.localdomain localhost`

2. Set up `rsh, rlogin,` and `rexec`. For the head node and each of the compute nodes:

  - At the end of `/etc/securetty` add `rsh`, `rexec`, and `rlogin`.
  - Create `/etc/hosts.equiv` file containing hostnames of head node and compute nodes.
  - Copy `/etc/hosts.equiv` to `/root/.rhosts`
  - Set `rsh` to `run` in `runlevel 3`, then do the same for `rexec` and `rlogin`, that is: *`/sbin/chkconfig --level 3 rsh on`*
  - To test, as root run `rsh localhost` and `rsh hostname` If these commands do not work, verify that a correct `127.0.0.1` line is in the */etc/hosts* file.

12.2 NIS Configuration

This section contains instructions for configuring user accounts for the cluster.

12.2.1 Head Node NIS Configuration

If compute nodes use outer domain logins and home directories, skip to Section 12.2.2, *Compute Node NIS Configuration* and configure the head node the same way as the other compute nodes.
To configure the head node:

1. Make sure that `ypserv rpm` is installed.
2. Configure the head node as a client of the outer NIS domain:
   `/bin/domainname outerdomain.company.com`
3. To survive a reboot, in the file `/etc/sysconfig/network` add the line:
   `NISDOMAIN=`*`outerdomain.company.com`*
4. Edit `/etc/yp.conf` and add the line:
   `ypserver` *`ypserver001`*
5. Start `ypbind` with:
   `/etc/rc.d/init.d/ypbind start`
6. Set `ypbind` to run in runlevel 3 after reboot:
   `/sbin/chkconfig --level 3 ypbind on`

Configure the head node to export its local user accounts (not outer domain user accounts) to the compute nodes, as follows:

7. Switch to an internal domain name:
   `/bin/domainname` *`your_cluster_nis_domain`*
8. Start `ypserv` and `yppasswdd`:
   `/etc/rc.d/init.d/ypserv start`
   `/etc/rc.d/init.d/yppasswdd start`

9. Run `/usr/lib/yp/ypinit -m`. Type the hostname of the head node when prompted.
10. Change back to the outer NIS domain as in step 2.
11. Add `ypserv` and `yppasswdd` to runlevel 3 with chkconfig as in step 6.
12. Whenever a new user is created, update the NIS maps as follows:
    ```
    cd /var/yp
    /bin/domainname your_cluster_nis_domain
    make
    /bin/domainname outerdomain.company.com
    ```

12.2.2 Compute Node NIS Configuration

To configure compute nodes for NIS configuration, do the following:

1. Edit `/etc/yp.conf` as follows:
    - If you configured the head node so that it exports its local user accounts via NIS (that is, if you followed the steps in *Head Node NIS Configuration*), add the line `ypserver` *`your-head-node-hostname`*.
    - If compute node accounts are all resolved from the outer domain NIS servers (if you skipped the *Head Node NIS Configuration* section), add the line `ypserver` *`ypserver001`*
2. Start `ypbind` with:
    `/etc/rc.d/init.d/ypbind start`
3. Set `ypbind` to run in runlevel 3 after reboot:
    `/sbin/chkconfig --level 3 ypbind on`
4. Edit `/etc/nsswitch.conf` and make sure the following lines (or similar lines) are present: Note: This must be `nis`. Using `nisplus` does not work.
    ```
    passwd:     files nis
    shadow:     files nis
    group:      files nis
    ```
5. To survive a reboot, in the file `/etc/sysconfig/network` add the appropriate line as follows:
    - If you are using internal logins: `NISDOMAIN=`*`your-cluster-nis-domain`*
    - If you are using external logins: `NISDOMAIN=`*`outerdomain.company.com`*

12.3 NFS Configuration

This section contains instructions for configuring the file systems for the nodes of the cluster.

12.3.1 Head Node NFS Configuration

To configure head nodes for NFS:

1. Set up the node to receive outer domain user account home directories. Assuming NIS configuration is already working, start the automounter:
    `/etc/rc.d/init.d/autofs start`
2. Set `autofs` to run in runlevel 3 after reboot:
    `/sbin/chkconfig --level 3 autofs on`
3. Setup `/etc/exports` to cause directories to be exported to compute nodes.

Note

4. This step is essential for using Cluster OpenMP*: You need directories to be exported from the head node to ensure that your program can find the Cluster OpenMP library and your home directory.

Tip!

Using `man` exports may be helpful.

- Edit `/etc/exports` to contain the following line, modifying for the correct network, netmask, and options:
  `/opt      10.0.1.0/255.255.255.0(ro)`
- If clients are using user accounts local to the head node rather than the outer domain user accounts, export user home directories as follows:
  `/home     10.0.1.0/255.255.255.0(rw,no-root-squash)`
- Optionally, add the following lines if you want to share these directories. You must ensure that these directories exist on the compute nodes and, preferably, are empty:
  `/usr      10.0.1.0/255.255.255.0(rw,no-root-squash)`
  `/shared  10.0.1.0/255.255.255.0(rw,no-root-squash)`

5. Start or restart `nfs` with:
   `/etc/rc.d/init.d/nfs restart`
6. Set `nfs` to run in runlevel 3 after reboot:
   `/sbin/chkconfig --level 3 nfs on`

12.3.2 Compute Node NFS Configuration

To configure compute nodes for NFS:

1. Type :
   `mount` *`your-head-node`* `/opt /opt`
2. Edit the *`/etc/fstab`* file, and add the following line:
   `your-head-node``:/opt          /opt   nfs    defaults      0 0`
3. If compute nodes receive user accounts and directories from the outer network do the following:
   `/etc/rc.d/init.d/autofs start`
   `/sbin/chkconfig --level 3 autofs on`
4. If compute nodes receive user accounts and directories from the head node, type:
   `mount` *`your-head-node`*`:/home /home`
   and edit the `/etc/fstab` file to add the following line:
   `your-head-node``:/home         /home nfs    defaults     0 0`

12.4 Gateway Configuration

The configuration steps in this section are recommended for using Cluster OpenMP* and are required if you want the head node of the cluster to act as a gateway. This enables a Cluster OpenMP program to write to an external X Window.

12.4.1 Head Node Gateway Configuration

To configure the head node:

1. Turn on IP forwarding:
   `echo 1 > /proc/sys/net/ipv4/ip-forward`
2. To survive a reboot, add the following line to `/etc/sysctl.conf`:
   `net.ipv4.ip-forward = 1`
3. Save the `iptables` configuration. The following line writes the `iptables` rules to the `/etc/sysconfig/iptables` file, which you must define prior to running `iptables`:
   `/etc/rc.d/init.d/iptables save`
4. Turn off `ipchains` and turn on `iptables`:
   `/etc/rc.d/init.d/ipchains stop`
   `/etc/rc.d/init.d/iptables start`
5. Do the same in runlevel 3 to survive a reboot:
   `/sbin/chkconfig --level 3 iptables on`
   `/sbin/chkconfig --level 3 ipchains off`
6. Add a rule to forward packets from the internal nodes with a source IP of the head node:
   `/sbin/iptables -t nat -A POSTROUTING -o` ***`external-ethernet-port`*** **`-j SNAT -to-source`** ***`external-ip-address`***
7. Save this rule:
   `/etc/rc.d/init.d/iptables save`

12.4.2 Compute Node Gateway Configuration

To configure the compute nodes:

Add the following line to the `/etc/sysconfig/network` file:
`GATEWAY=`***`head-node-ip-address`***

13 Reference

13.1 Using Foreign Threads in a Cluster OpenMP Program

It is possible for a program to start its own POSIX* threads that access the sharable memory provided by Cluster OpenMP. Threads started explicitly by your program are called *foreign threads*.

Foreign threads can access `sharable` memory, call OpenMP and Cluster OpenMP API routines, and execute OpenMP constructs. However, all OpenMP constructs executed by foreign threads will be serialized, that is, executed by just one thread.

13.2 Cluster OpenMP* Options Reference

You can access brief descriptions of the following commands by typing the –help command. The Cluster OpenMP* (CLOMP or Cluster OMP) options are available if you have a separate license for the Cluster OpenMP product.

These options can be used on Linux* systems running on Intel® Itanium® processors or IA-32 processors with Intel® Extended Memory 64 Technology (Intel® EM64T).

Command	Description
`-[no-]cluster-openmp`	Enables you to run an OpenMP program on a cluster.
`-[no-]cluster-openmp-profile`	Link a Cluster OpenMP program with profiling information.
`-[no-]clomp-sharable-propagation`	Reports variables that need to be made sharable by you with Cluster OpenMP.
`-[no-]clomp-sharable-info`	Reports variables that the compiler automatically makes sharable for Cluster OpenMP.
`-[no-]clomp-sharable-commons`	(Fortran only) Makes all `COMMONs` sharable by default for Cluster OpenMP.
`-[no-]clomp-sharable-modvars`	(Fortran only) Makes all variables in modules sharable by default for Cluster OpenMP.
`-[no-]clomp-sharable-localsaves`	(Fortran only) Makes all `SAVE` variables sharable by default for Cluster OpenMP.
`-[no-]clomp-sharable-argexprs`	(Fortran only) Makes all expressions in function and subroutine call statements sharable by default for Cluster OpenMP.

Glossary

The following definitions of terms related to Cluster OpenMP* are used throughout this document:

backing store – file space assigned to hold a backup copy of system memory.

Cluster OpenMP* – the Intel® implementation of OpenMP for a distributed memory environment.

compute node – one of the nodes of a cluster that is not the head node.

DVSM – distributed virtual shared memory – the underlying mechanism that provides the shared memory space required by OpenMP.

foreign thread – a thread started by you through an explicit thread creation call.

head node – the node of a cluster visible outside the cluster. Users usually login to a cluster through its head node.

home node – the node of a cluster where a Cluster OpenMP program is originally started by you.

home process – the process started on the home node to run the Cluster OpenMP program.

host – see the definition for node.

host pool – the set of hosts that run a Cluster OpenMP program.

master thread – the thread that runs the serial code at the beginning of an OpenMP program. The master thread forms each Cluster OpenMP parallel team.

multi-node program – a Cluster OpenMP program that runs on more than one node. There is a minimum of one process per node, so a multi-node program is also a multi-process program.

multi-process program – a Cluster OpenMP program that includes more than one process.

node – a computer with its own operating system. In a cluster, the nodes are connected together by a communications fabric (i.e., a network).

OpenMP thread – a thread started on behalf of you, due to the semantics of the OpenMP program that executes user-written code.

OpenMP* – a directive-based parallel programming language, for annotating Fortran, C, and C++ programs. See http://www.openmp.org.

principal thread – the distinguished thread in a Cluster OpenMP process that begins the execution in that process. The principal thread in the home process is called the master thread for the Cluster OpenMP program.

process – an operating-system-schedulable unit of execution, including one or more threads, a virtual memory and access to resources such as disk files. A Cluster OpenMP program consists of one or more processes running on one or more nodes of a cluster.

remote node – a node of a cluster, different from the home node that runs part of a Cluster OpenMP program.

remote process – a process spawned from the home process, usually on a remote node, for the purpose of

executing a Cluster OpenMP program.

sharable memory – the memory space in a Cluster OpenMP program that is kept consistent across all the Cluster OpenMP processes.

socket – a communication channel opened between processes, used for passing messages.

subordinate threads – all OpenMP threads in a Cluster OpenMP process that are not the principal thread.

thread – an entity of program execution, including register state and a stack. A Cluster OpenMP program includes threads for executing your code and threads for supporting the Cluster OpenMP mechanism.

twin – a read-only copy of a sharable memory page.

worker threads – all OpenMP threads that are not the master thread.

Index

/etc/hosts, 49
/tmp partition, 47
ALLOCATE, 38
assumed-shape, 46
bottom-half threads, 44
clomp_forecaster, 30
configuration checker, 21
DISPLAY, 45
environment, 21
environment variable, 24
foreign threads, 54
Gateway configuration, 51
gdb, 28
heartbeat, 25
-IOdebug, 29
-IOfiles:, 29
-IOsystem, 29
kmp_aligned_sharable_malloc, 37
kmp_cluster.ini, 9, 22
KMP_CLUSTER_DEBUGGER, 22, 37
kmp_get_num_processes, 38
kmp_get_process_num, 38
kmp_get_process_thread_num, 38
kmp_lock_cond_broadcast, 37
kmp_lock_cond_signal, 37
kmp_lock_cond_wait, 37
kmp_nest_lock_cond_broadcast, 37
kmp_nest_lock_cond_signal, 37
kmp_nest_lock_cond_wait, 37
kmp_private_munmap, 37
kmp_set_warnings_off, 38
kmp_set_warnings_on, 38
kmp_sharable_calloc, 37
kmp_sharable_free, 37
kmp_sharable_malloc, 37
kmp_sharable_munmap, 37
kmp_sharable_realloc, 37
KMP_SHARABLE_STACKSIZE, 36
KMP_STACKSIZE, 36
KMP_STATSFILE, 37
KMP_VERSION, 37
master thread, 44
mmap, 25, 45
munmap, 25, 45
NFS Configuration, 50
NIS configuration, 49
OMP_DYNAMIC, 36
omp_get_num_procs, 35
omp_get_num_threads, 35
omp_lock_t, 18
OMP_NUM_THREADS, 35
OMP_SCHEDULE, 35
omp_set_num_threads, 35
OpenPBS, 26
output listener thread, 44
PBS, 26
PBS bug, 26
principal thread, 44
queueing system, 26
repo, 44
rsh, 22
SIGSEGV, 28, 43
socket, 45
ssh, 22, 26
stderr, 24, 25
stdin, 25
stdout, 24, 25, 44
subordinate threads, 44
top-half threads, 44
vector time stamp, 43
write notice, 43