OpenMP API 用户指南

Sun™ Studio 10

Sun Microsystems, Inc.
www.sun.com

文件号码 819-1617-10
2005 年 1 月，修订 (版) A

请将有关本文档的意见和建议提交至： http://www.sun.com/hwdocs/feedback

目录

表

开始之前

*《OpenMP API 用户指南》*概述了用于生成多重处理应用程序的 OpenMP Fortran 95、 C 和 C++ 应用程序接口 (API)。 Sun™ Studio 编译器支持 OpenMP API。

本指南专供科学工作者、工程技术人员以及具有 Fortran、 C 或 C++ 语言及 OpenMP 并行编程模型的应用知识的程序员使用。通常，还假定他们熟悉 Solaris™ 操作环境或 UNIX®。

印刷约定

表 P-1 字体约定

字体	含义	示例
`AaBbCc123`	命令、文件和目录的名称；计算机屏幕输出	编辑您的 `.login` 文件。 使用 `ls -a` 列出所有文件。 `% You have mail.`
`AaBbCc123`	输入的内容，以便与计算机屏幕输出相区别	`% su` `Password:`
AaBbCc123	书名、新词或术语以及要强调的词	请阅读 *《用户指南》*的第 6 章。 这些称作*类*选项。 您*必须*是超级用户才能执行此操作。
AaBbCc123	命令行占位符文本；用实际名称或值替换	要删除文件，请键入 **`rm`** *filename*。

表 **P-2** 代码约定

代码符号	含义	表示法	代码示例
[ ]	括号包含可选参数。	O[*n*]	-O4, -O
{ }	大括号包含所需选项的选项集合。	d{y|n}	-dy
|	分隔变量的“|”或“-”符号，只能选择其一。	B{dynamic|static}	-Bstatic
:	与逗号一样，冒号有时可用于分隔参数。	R*dir*[:*dir*]	-R/local/libs:/U/a
…	省略号表示一系列省略。	-xinline=*fl* [, …*fn*]	-xinline=alpha,dos

Shell 提示符

Shell	提示符
C shell	*machine-name*%
C shell 超级用户	*machine-name*#
Bourne shell 和 Korn shell	$
Bourne shell 和 Korn shell 的超级用户	#

受支持的平台

此 Sun Studio 发行版本支持使用如下 SPARC® 和 x86 系列处理器架构的系统：UltraSPARC®、ARC64、AMD64、Pentium 和 Xeon EM64T。可从以下位置获得硬件兼容性列表，在列表中可以查看您正在使用的 Solaris 操作系统版本所支持的系统：http://www.sun.com/bigadmin/hcl。这些文档中给出了平台类型间所有实现的区别。

在本文档中，术语“x86”指采用兼容 AMD64 或 Intel Xeon/Pentium 产品系列处理器的 64 位和 32 位系统。有关受支持的系统，请参阅硬件兼容性列表。

访问 Sun Studio 软件和手册页

编译器和工具以及它们的手册页并没有安装到标准的 /usr/bin/ 和 /usr/share/man 目录中。要访问编译器和工具，必须正确设置 PATH 环境变量 （请参阅第 xi 页 “访问编译器和工具”）。要访问手册页，必须正确设置 MANPATH 环境变量 （请参阅第 xii 页 “访问手册页”）。

关于 PATH 变量的更多信息，请参阅 csh(1)、 sh(1) 和 ksh(1) 手册页。关于 MANPATH 变量的更多信息，请参阅 man(1) 手册页。关于设置 PATH 变量和 MANPATH 变量以访问此发行版本的更多信息，请参阅安装指南或询问系统管理员。

注 – 本节中的信息假设 Sun ONE Studio 编译器和工具安装在 /opt 目录中。如果软件没有安装在 /opt 目录中，请询问系统管理员以获取系统中的等效路径。

访问编译器和工具

使用下列步骤来决定是否需要更改 PATH 变量以访问编译器和工具。

决定是否需要设置 PATH 环境变量

1. **通过在命令提示符后输入下列内容以显示 PATH 变量的当前值。**

```
% echo $PATH
```

2. **查看输出中是否有包含 /opt/SUNWspro/bin/ 的路径字符串。**

   如果找到该路径，您的 PATH 变量已经设置好，可以访问编译器和工具了。如果没有找到该路径，按照下一步中的说明来设置 PATH 环境变量。

设置 PATH 环境变量以访问编译器和工具

1. **如果使用的是 C shell，请编辑起始 .cshrc 文件。如果使用的是 Bourne shell 或 Korn shel，请编辑起始 .profile 文件。**

2. **将下列内容添加到 PATH 环境变量。如果已安装 Forte Developer 软件、 Sun ONE Studio 软件或 Sun Studio 软件的其他发行版本，则将以下路径添加到这些安装的路径之前。**

   /opt/SUNWspro/bin

访问手册页

使用下列步骤来决定是否需要更改 MANPATH 变量以访问手册页。

决定是否需要设置 MANPATH 环境变量

1. **通过在命令提示符后输入下列内容以请求 dbx 手册页。**

```
% man dbx
```

2. **如果有输出的话，请查看输出。**

   如果 dbx(1) 手册页无法找到或者显示的手册页不是用于安装软件的当前版本，请按照下一步中的说明来设置 MANPATH 环境变量。

设置 MANPATH 环境变量以访问手册页

1. **如果使用的是 C shell，请编辑起始 .cshrc 文件。如果使用的是 Bourne shell 或 Korn shel，请编辑起始 .profile 文件。**

2. **将下列内容添加到 MANPATH 环境变量。**

   /opt/SUNWspro/man

访问集成开发环境

Sun Studio 集成开发环境 (IDE) 提供了创建、编辑、生成、调试和分析 C、C++ 或 Fortran 应用程序性能的模块。

启动 IDE 的命令是 sunstudio。有关该命令的详细信息，请参阅 sunstudio(1) 手册页。

IDE 是否能够正确操作取决于 IDE 能否找到核心平台。sunstudio 命令查找两个位置的核心平台：

- 该命令首先在缺省安装目录 /opt/netbeans/3.5V 中查找。
- 如果该命令在缺省目录未找到核心平台，则它将假设包含 IDE 的目录和包含核心平台的目录均安装在同一位置上。例如，如果包含 IDE 的目录的路径是 /foo/SUNWspro，该命令将在 /foo/netbeans/3.5V 中查找核心平台。

如果核心平台未安装在 sunstudio 命令查找它的任一位置上，客户端系统上的每个用户必须将环境变量 SPRO_NETBEANS_HOME 设置为安装核心平台的位置 (/*installation_directory*/netbeans/3.5V)。

IDE 的每个用户还必须将 /*installation_directory*/SUNWspro/bin 添加到其他任何 Forte Developer 软件、 Sun ONE Studio 软件或 Sun Studio 软件发行版本路径前面的 $PATH 中。

路径 /*installation_directory*/netbeans/3.5V/bin 不可添加到用户的 $PATH 中。

访问编译器和工具文档

您可以访问下列位置的文档：

- 可以在随软件一起安装的文档索引 file:/opt/SUNWspro/docs/index.html 中（位于本地系统或网络上）获取文档。

  如果软件没有安装在 /opt 目录中，请询问系统管理员以获取系统中的等效路径。

- 大多数的手册都可以从 docs.sun.com[sm] web 站点上获得。下列书目只能从您所安装的软件中找到：
  - *《标准 C++ 库类参考》*
  - *《标准 C++ 库用户指南》*
  - *《Tools.h++ 类库参考》*
  - *《Tools.h++ 用户指南》*
- 发行说明可以从 docs.sun.com web 站点上获得。
- 在 IDE 中通过 “帮助”菜单或窗口和对话框上的 “帮助”按钮可以访问 IDE 所有组件的联机帮助。

您可以通过因特网在 docs.sun.com web 站点 (http://docs.sun.com) 上阅读、打印和购买 Sun Microsystems 的各种手册。如果找不到手册，请参阅和软件一起安装在本地系统或网络中的文档索引。

注 – Sun 对本文档中提到的第三方 Web 站点的可用性不承担任何责任。对于此类站点或资源中的（或通过它们获得的）任何内容、广告、产品或其他材料， Sun 并不表示认可，也不承担任何责任。对于因使用或依靠此类站点或资源中的（或通过它们获得的）任何内容、产品或服务而造成的或连带产生的实际或名义损坏或损失， Sun 概不负责，也不承担任何责任。

使用易读格式的文档

该文档以易读格式提供，以方便残障用户使用辅助技术进行阅读。您还可以按照下表所描述的信息找到文档的易读版本。如果软件没有安装在 /opt 目录中，请询问系统管理员以获取系统中的等效路径。

文档类型	易读版本的格式和位置
手册 （第三方手册除外）	HTML，位于 http://docs.sun.com
第三方手册： • *《标准 C++ 库类参考》* • *《标准 C++ 库用户指南》* • *《Tools.h++ 类库参考》* • *《Tools.h++ 用户指南》*	HTML，位于安装的软件中的文档索引 file:/opt/SUNWspro/docs/index.html
自述文件和手册页	HTML，位于安装的软件中的文档索引 file:/opt/SUNWspro/docs/index.html
联机帮助	通过 IDE 中的 “帮助”菜单可以使用 HTML
发行说明	HTML，位于 http://docs.sun.com

相关编译器和工具文档

下表描述的相关文档可以在 file:/opt/SUNWspro/docs/index.html 和 http://docs.sun.com 上获得。如果软件没有安装在 /opt 目录中，请询问系统管理员以获取系统中的等效路径。

文档标题	描述
Fortran 编程指南	描述如何在 Solaris 环境中编写高效 Fortran 代码；输入 / 输出、库、性能、调试和并行处理。
Fortran 库参考	详细说明 Fortran 库和内部例程
Fortran 用户指南	描述 f95 编译器的编译时环境和命令行选项。还包括关于将以前的 f77 程序迁移到 f95 的说明。
C 用户指南	描述 cc 编译器的编译时环境和命令行选项。
C++ 用户指南	描述 CC 编译器的编译时环境和命令行选项。
数值计算指南	描述关于浮点计算数值精确性的问题。

访问相关的 Solaris 文档

下表描述了可从 docs.sun.com web 站点上获得的相关文档。

文档集合	文档标题	描述
Solaris 参考手册集合	请参阅手册页部分的标题。	提供关于 Solaris 操作环境的信息。
Solaris 软件开发人员集合	*链接程序和库指南*	描述了 Solaris 链接编辑器和运行时链接程序的操作。
Solaris 软件开发人员集合	*多线程编程指南*	涵盖 POSIX® 和 Solaris 线程 API、使用同步对象进行程序设计、编译多线程程序和多线程程序的查找工具。

开发人员资源

访问 http://developers.sun.com/prodtech/cc 以查找以下经常更新的资源：

- 关于编程技术和最佳方法的文章
- 短小编程提示的知识库
- 编译器和工具组件的文档以及与软件同时安装的文档的更正
- 支持等级信息
- 用户论坛
- 可下载代码样例
- 新技术预览

您可以在 http://developers.sun.com 上找到开发人员的额外资源。

与 Sun 技术支持联系

如果您有关于本产品的技术问题而本文档未予以解答，请访问：

http://www.sun.com/service/contacting

Sun 欢迎您提出意见和建议

Sun 致力于提高文档质量，并欢迎您提出宝贵的意见和建议。请将您的意见发送至以下 URL：

`http://www.sun.com/hwdocs/feedback`

请在电子邮件的主题行中注明文档的文件号码 (819-1617-10)。

当您提供意见和建议时，可能需要在表单中提供文档英文版本的标题和文件号码。本文档英文版本的文件号码和标题是：819-0501-10， OpenMP API User's Guide。

第 1 章

OpenMP API 概述

OpenMP™ 应用程序接口是与多家计算机供应商联合开发的、针对共享内存多处理器体系结构的可移植并行编程模型。其规范由“OpenMP 体系结构审核委员会”创立并公布。有关 OpenMP 的更多信息（包括教程和其他资源），请访问其网站：`http://www.openmp.org/`。

OpenMP API 是 Solaris™ 操作系统平台上所有 Sun Studio 编译器的建议并行编程模型。有关将传统 Fortran 和 C 并行化指令转换为 OpenMP 指令的指导，请参阅第 6 章。

本章概述组成“OpenMP 2.0 版应用程序接口”并由 Sun Studio Fortran 95、C 和 C++ 编译器实现的指令、运行时库例程及环境变量。

1.1 哪里有 OpenMP 规范

简洁起见，本章中提供的材料有意略去了许多详细信息，*只是一个概述*。随时都可参阅 OpenMP 规范文档来了解完整的详细信息。

Fortran 和 C/C++ OpenMP 2.0 规范可以在 OpenMP 官方网站 `http://www.openmp.org/` 上获得。

1.2 本章所使用的特殊约定

在以下表格和示例中， Fortran 指令和源代码虽以大写形式出现，但实际上不区分大小写。

*结构化块*指无进或出传输的 Fortran 或 C/C++ 语句块。

方括号 [...] 内的构造为可选构造。

本手册中，“Fortran”指 Fortran 95 语言和编译器 **f95**。

本手册中，“指令”和 “pragma”互换使用。

1.3 指令格式

每个指令行中只能指定一个*指令名*，且该指令名应用于随后的程序语句。

Fortran：

Fortran 固定格式可以接受三个指令 “标记”，而自由格式只能接受一个。在下面的 Fortran 示例中，将使用自由格式。

C/C++：

C 和 C++ 使用以 **#pragma omp** 开头的标准预处理指令。

OpenMP 2.0 Fortran

固定格式：

C$OMP *directive-name optional_clauses...*
!$OMP *directive-name optional_clauses...*
*$OMP *directive-name optional_clauses...*

标记必须从第一列开始；续行的第 6 列必须含非空白或非零字符。
指令行第 6 列后可以有注释，注释以感叹号 (**!**) 开头。行中 **!** 后的其余内容都会被忽略。

自由格式：

!$OMP *directive-name optional_clauses...*

可以出现在行内任何位置，之前只能使用空白；行尾的和号 (**&**) 用于标识续行。
指令行中可以有注释，注释以感叹号 (!) 开头。行中其余内容都会被忽略。

OpenMP 2.0 C/C++

#pragma omp *directive-name optional_clauses...*
每个 pragma 必须以换行符结尾，并遵循标准的 C 和 C++ 编译器编译指示约定。
Pragma 区分大小写。子句出现的顺序并不重要。 **#** 前后和字间可以有空白。
指令应用于随后的语句，该语句必须为结构化块。

1.4 条件编译

OpenMP API 定义预处理程序符号 **_OPENMP** 用于条件编译。此外，OpenMP Fortran API 也接受条件编译标记。

OpenMP 2.0 Fortran

固定格式：

```
!$      fortran_95_statement
C$      fortran_95_statement
*$      fortran_95_statement
c$      fortran_95_statement
```

标记必须从第 1 列开始，且不能包含中间空白。启用 OpenMP 编译时，该标记用两个空白代替。行内其余内容必须符合标准的 Fortran 固定格式约定。示例：

```
C23456789
!$ 10  iam  =  OMP_GET_THREAD_NUM()  +
!$    1         index
```

自由格式：

```
!$  fortran_95_statement
```

此标记可以出现在任意列，之前只能为空白，且必须以单个字形式出现。Fortran 自由格式约定应用于行的其余内容。示例：

```
C23456789
 !$  iam  =  OMP_GET_THREAD_NUM()  +     &
 !$&          index
```

Fortran 预处理程序：

启用 OpenMP 时编译会定义预处理程序符号 **_OPENMP**。

```
#ifdef _OPENMP
     iam  =  OMP_GET_THREAD_NUM()+index
#endif
```

OpenMP 2.0 C/C++

C/C++ 预处理程序：

启用 OpenMP 时编译会定义宏 **_OPENMP**。

```
#ifdef _OPENMP
     iam  =  omp_get_thread_num()  +  index;
#endif
```

1.5 **PARALLEL**——并行区域构造

PARALLEL 指令定义并行区域，该区域多个线程以并行方式执行的程序区域。

OpenMP 2.0 Fortran

```
!$OMP  PARALLEL  [clause[[,]clause]...]
   结构化块
!$OMP  END  PARALLEL
```

OpenMP 2.0 C/C++

```
#pragma  omp  parallel  [clause[[,]clause]...]
      结构化块
```

有许多特殊条件和限制。有关详细信息，强烈建议编程人员参阅相应的 OpenMP 规范文档。

表 1-1 列出了可随此构造出现的子句。

1.6 工作共享构造

工作共享构造在遇到该构造的线程组成员中分配封装代码区域的执行。要使工作共享构造以并行方式执行，构造必须封装在并行区域内。

这些指令及其所应用到的代码有许多特殊条件和限制。有关详细信息，强烈建议编程人员参阅相应的 OpenMP 规范文档。

1.6.1 DO 和 for 构造

指定随后的 **DO** 或 **for** 循环的迭代应该以并行方式执行。

OpenMP 2.0 Fortran

```
!$OMP DO [clause[[,] clause]...]
  do_loop
[!$OMP END DO [NOWAIT]]
```

DO 指令指定其后紧跟的 **DO** 循环的迭代应以并行方式执行。循环迭代将分配到执行绑定了循环的并行区域的线程组中的已有线程。此指令必须出现在并行区域内才有效。

OpenMP 2.0 C/C++

```
#pragma omp for [clause[[,]clause]...]
  for-loop
```

for pragma 指定其后紧跟的 *for- loop* 的迭代应以并行方式执行。循环迭代将分配到执行绑定了循环的并行区域的线程组中的已有线程。此 pragma 必须出现在并行区域内才有效。 **for** pragma 对相应 **for** 循环的结构有限制，且必须有*规范格式*：

for （*initexpr*； *var logicop b*； *increxpr*）

其中：

- *initexpr* 为以下之一：
  var = lb
  integer_type var = lb
- *increxpr* 为以下表达式形式之一：
  ++var
  var++
  --var
  var--
  var += incr
  var -= incr
  var = var + incr
  var = incr + var
  var = var - incr
- *var* 是有符号整型变量，被隐式设置为供 **for** 范围专用。切勿修改 **for** 语句体内的 *var*。除非指定 **lastprivate**，否则其值在循环后不确定。
- *logicop* 为以下逻辑操作符之一：
  < <= > >=
- *lb*、*b* 和 *incr* 是循环不变量整型表达式。

对 < 或 <= 和 > 或 >= 作为 **for** 语句中的 *logicalop* 使用尚有其他限制。有关详细信息，请参阅 OpenMP C/C++ 规范。

表 1-1 列出了可随此构造出现的子句。

1.6.2 **SECTIONS** 构造

SECTIONS 构造用于封装要在组内的线程中分配的一组结构化代码块。每个块由组内的线程执行一次。

每段均以 **SECTION** 指令开头，该指令对第一段为可选指令。

OpenMP 2.0 Fortran

```
!$OMP SECTIONS [clause[[,] clause]...]
[!$OMP SECTION]
    结构化块
[!$OMP SECTION
 结构化块 ]
...
!$OMP END SECTIONS [NOWAIT]
```

OpenMP 2.0 C/C++

```
#pragma omp sections [clause[[,]clause]...]
  {
   [#pragma omp section ]
          结构化块
   [#pragma omp section
          结构化块 ]
    ...
  }
```

表 1-1 列出了可随此构造出现的子句。

1.6.3 **SINGLE** 构造

用 **SINGLE** 封装的结构化块只由组内的一个线程来执行。除非指定 **NOWAIT**，否则组内未执行 **SINGLE** 块的线程会在块结尾处等待。

OpenMP 2.0 Fortran

```
!$OMP SINGLE [clause[[,] clause]...]
   结构化块
!$OMP END SINGLE [end-modifier]
```

OpenMP 2.0 C/C++

```
#pragma omp single [clause[[,] clause]...]
   结构化块
```

表 1-1 列出了可随此构造出现的子句。

1.6.4 Fortran **WORKSHARE** 构造

WORKSHARE 构造将执行封装代码块的工作划分为独立的工作单元，并使组内的线程共享工作，这样每个单元便只执行一次。

OpenMP 2.0 Fortran

```
!$OMP WORKSHARE
   结构化块
!$OMP END WORKSHARE [NOWAIT]
```

没有与 Fortran WORKSHARE 构造等效的 C/C++ 指令。

1.7 合并的并行工作共享构造

合并的并行工作共享构造是指定包含一个工作共享构造的并行区域的捷径。

这些指令及其所应用到的代码有许多特殊条件和限制。有关完整的详细信息，请参阅相应的 OpenMP 规范文档。以下说明只是概述，内容并不详尽。

表 1-1 列出了可随这些构造出现的子句。

1.7.1 **PARALLEL DO** 和 **parallel for** 构造

指定包含单个 **DO** 或 **for** 循环的并行区域的捷径。等价于 **PARALLEL** 指令后紧跟 **DO** 或 **for** 指令。除 **NOWAIT** 修饰符外，*子句*可以是 **PARALLEL** 和 **DO/for** 指令可接受的任意子句。

OpenMP 2.0 Fortran

```
!$OMP PARALLEL DO [clause[[,] clause]...]
   do_loop
[!$OMP END PARALLEL DO ]
```

OpenMP 2.0 C/C++

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

1.7.2 **PARALLEL SECTIONS** 构造

指定包含单个 **SECTIONS** 指令的并行区域的捷径。等效于 **PARALLEL** 指令后紧跟 **SECTIONS** 指令。除 **NOWAIT** 修饰符外，*子句*可以是 **PARALLEL** 和 **SECTIONS** 指令可接受的任意子句。

OpenMP 2.0 Fortran

```
!$OMP PARALLEL SECTIONS [clause[[,] clause]...]
[!$OMP SECTION]
   结构化块
[!$OMP SECTION
  结构化块 ]
...
!$OMP END PARALLEL SECTIONS
```

OpenMP 2.0 C/C++

```
#pragma omp parallel sections [clause[[,] clause]...]
  {
  [#pragma omp section]
       结构化块
  [#pragma omp section
       结构化块]
  ...
  }
```

1.7.3 PARALLEL WORKSHARE 构造

Fortran **PARALLEL WORKSHARE** 构造为指定包含单个 **WORKSHARE** 指令的并行区域提供了一条捷径。 *clause* 可以是 **PARALLEL** 指令可接受的子句之一。

OpenMP 2.0 Fortran

```
!$OMP PARALLEL WORKSHARE [clause[[,] clause]...]
   结构化块
!$OMP END PARALLEL WORKSHARE
```

没有等效的 C/C++ 指令。

1.8 同步构造

以下构造指定线程同步。有关这些构造的特殊条件和限制非常多，无法在此处完全概述。有关完整的详细信息，强烈建议编程人员参阅相应的 OpenMP 规范文档。

1.8.1 **MASTER** 构造

只有组内的主线程才执行此指令所封装的块。其他线程会跳过此块，然后继续执行。主构造的入口或出口处无暗含障碍。

OpenMP 2.0 Fortran

```
!$OMP MASTER
   结构化块
!$OMP END MASTER
```

OpenMP 2.0 C/C++

```
#pragma omp master
     结构化块
```

1.8.2 **CRITICAL** 构造

每次限一个线程可访问结构化块。可选的 *name* 参数标识临界区域。所有未命名的 **CRITICAL** 指令都映射到同一名称。临界段名称是程序的全局实体，必须唯一。对于 Fortran，如果 *name* 出现在 **CRITICAL** 指令中，便也必须出现在 **END CRITICAL** 指令中。对于 C/C++，用于给临界区域命名的标识符有外部链接，且其所在的名字空间与标签、标记、成员及普通标识符所使用的名字空间不同。

OpenMP 2.0 Fortran

```
!$OMP CRITICAL [(name)]
   结构化块
!$OMP END CRITICAL [(name)]
```

OpenMP 2.0 C/C++

```
#pragma omp critical [(name)]
     结构化块
```

1.8.3 **BARRIER** 构造

同步组内的所有线程。每个线程都等到组内所有其他线程都到达此点为止。

OpenMP 2.0 Fortran

```
!$OMP BARRIER
```

OpenMP 2.0 C/C++

```
#pragma omp barrier
```

组内的所有线程都遇到障碍后，组内的每个线程便都开始执行 **BARRIER** 指令后的语句。

请注意，由于 **barrier** pragma 不使用 C/C++ 语句作为其语法的一部分，因此有对其在程序内位置的限制。有关详细信息，请参阅 C/C++ OpenMP 规范。

1.8.4 **ATOMIC** 构造

确保特定内存位置自动更新，而不要将其交由不确定的多个同时写入线程来支配。

OpenMP 2.0 Fortran

```
!$OMP ATOMIC
```
表达式语句

该指令只应用于随后紧跟的 *表达式语句*，且语句必须采用以下这些形式之一：

x = x operator expression

x = expression operator x

x = intrinsic(x, expr-list)

x = intrinsic(expr-list, x)

其中：

- *x* 为内在类型的标量
- *expression* 为不引用 *x* 的标量表达式
- *expr-list* 为不引用 x、非空、以逗号分隔的标量表达式列表 （*有关详细信息，请参阅 OpenMP 2.0 Fortran 规范*）
- *intrinsic* 为 **MAX**、**MIN**、**IAND**、**IOR** 或 **IEOR** 之一。
- *operator* 为 **+ - * / .AND..OR..EQV..NEQV.** 之一

OpenMP 2.0 C/C++

```
#pragma omp atomic
    表达式语句
```

该指令只应用于随后紧跟的 *表达式语句*，且语句必须采用以下这些形式之一：

x binop = expr

x++

++*x*

x--

--*x*

其中：

- *x* 为带标量类型的左值表达式。
- *expr* 为不引用 *x* 的带标量类型的表达式。
- *binop* 不是重载操作符，也不是以下操作符之一：**+**、 *****、 **-**、 **/**、 **&**、 **^**、 **|**、 **<<** 或 **>>**。

此实现在临界区域中封装表达式语句来替换所有 ATOMIC 指令。

1.8.5 **FLUSH** 构造

线程可见的 Fortran 变量或 C 对象被写回到出现此指令的内存位置。**FLUSH** 指令只提供执行线程和全局内存内操作间的一致性。可选的 *variable-list* 由需要刷新的变量或对象的列表组成，变量或对象间以逗号分隔。不带 *variable-list* 的 **FLUSH** 指令会同步所有线程可见的共享变量或对象。

OpenMP 2.0 Fortran

```
!$OMP FLUSH  [(variable-list)]
```

OpenMP 2.0 C/C++

```
#pragma omp flush [(variable-list)]
```

请注意，由于 **flush** pragma 不使用 C/C++ 语句作为其语法的一部分，因此有对其在程序内位置的限制。有关详细信息，请参阅 C/C++ OpenMP 规范。

1.8.6 **ORDERED** 构造

封装的块按迭代在循环的顺序执行中的执行顺序执行。

OpenMP 2.0 Fortran

```
!$OMP ORDERED
   结构化块
!$OMP END ORDERED
```

封装的块按迭代在循环的顺序执行中的执行顺序执行。它只会出现在 **DO** 或 **PARALLEL DO** 指令的动态范围内。 **ORDERED** 子句必须在封装该块的最近 **DO** 指令中指定。
每次迭代时， **DO** 指令应用到的循环不能执行同一 **ordered** 指令超过一次，且不能执行超过一个 **ordered** 指令。

OpenMP 2.0 C/C++

```
#pragma omp ordered
    结构化块
```

封装的块按迭代在循环的顺序执行中的执行顺序执行。它只能出现在指定了 **ordered** 子句的 **for** 或 **parallel for** 指令的动态范围内。
每次迭代时，有 **for** 构造的循环不能执行同一 **ordered** 指令超过一次，且不能执行超过一个 **ordered** 指令。

1.9 数据环境指令

以下指令用于在并行构造执行期间控制数据环境。

1.9.1 **THREADPRIVATE** 指令

将对象*列表*（Fortran 公共块和命名变量、 C 和 C++ 命名变量）设置为某线程专用，但在线程内为全局性。

有关完整的详细信息和限制，请参阅 OpenMP 规范。

OpenMP 2.0 Fortran

```
!$OMP THREADPRIVATE(list)
```

公共块名必须出现在斜杠间。要设置某公共块为 **THREADPRIVATE**，此指令必须出现在该块的每个 **COMMON** 声明后。

OpenMP 2.0 C/C++

`#pragma omp threadprivate (`*list*`)`

文件、名字空间或块作用域处的 *list* 中的每个变量都必须引用词典上位于 pragma 之前的文件、名称空间或块作用域处的变量声明。

1.10 OpenMP 指令子句

本节概述可以出现在 OpenMP 指令中的数据作用域和调度子句。

1.10.1 数据作用域子句

有几个指令接受允许用户在构造范围内控制变量的作用域属性的子句。如果未给指令指定数据作用域子句，则受指令影响的变量的缺省作用域为 **SHARED**。

Fortran：*list* 是以逗号分隔、可在作用域单元中访问的命名变量或公共块列表。公共块名必须出现在斜杠内 （例如， **/ABLOCK/**）。

*对这些作用域子句的使用有一些重要的限制。*有关完整的详细信息，请参阅 OpenMP 规范的相应章节。

表 1-1 列出可出现这些子句的指令。

1.10.1.1 **PRIVATE** 子句

private(*list***)**

将可选的以逗号分隔的 *list* 中的变量声明为供组内各线程专用。

1.10.1.2 **SHARED** 子句

shared(*list***)**

组内所有线程都共享 *list* 中出现的变量，并访问同一存储区域。

1.10.1.3 DEFAULT 子句

Fortran

```
DEFAULT(PRIVATE | SHARED | NONE)
```

C/C++

```
default(shared | none)
```

指定并行区域内所有变量的作用域属性。 **THREADPRIVATE** 变量不受此子句影响。未指定时使用 **DEFAULT(SHARED)**。可以使用 **private**、 **firstprivate**、**lastprivate**、 **reduction** 及 **shared** 子句覆盖变量的缺省数据共享属性。

1.10.1.4 FIRSTPRIVATE 子句

firstprivate(*list*)

列表中的变量为 **PRIVATE**。此外，变量的专用副本从构造前便已存在的原始对象中初始化得来。

1.10.1.5 LASTPRIVATE 子句

lastprivate(*list*)

列表中的变量为 **PRIVATE**。此外，**LASTPRIVATE** 子句在 **DO** 或 **for** 指令中出现时，执行序列末位的最后一个迭代的线程会更新原始对象。在 **SECTIONS** 指令中，执行按词汇顺序位于最后的 **SECTION** 的线程会更新原始对象。

1.10.1.6 COPYIN 子句

Fortran

COPYIN(*list*)

COPYIN 子句只应用于变量、公共块和声明为 **THREADPRIVATE** 的公共块中的变量。在并行区域中，**COPYIN** 指定组主线程中的数据复制到并行区域开头的线程专用副本中。

C/C++

copyin(*list*)

COPYIN 子句只应用于声明为 **THREADPRIVATE** 的变量。在并行区域中，**COPYIN** 指定组主线程中的数据复制到并行区域开头的线程专用副本中。

1.10.1.7 **COPYPRIVATE** 子句

Fortran

COPYPRIVATE(*list*)

使用专用变量将值或指向共享对象的指针从组的一个成员广播给其他成员。**COPYPRIVATE** 子句只能在 **END SINGLE** 指令中出现。广播发生在执行与 **single** 关联的结构化块后，组中任何线程在构造尾部留下障碍前。*list* 中的变量决不可在指定 **COPYPRIVATE** 的 **SINGLE** 构造的 **PRIVATE** 或 **FIRSTPRIVATE** 子句中出现。

C/C++

copyprivate(*list*)

使用专用变量将值从组的一个成员广播给其他成员。**copyprivate** 子句只能在 **single** 指令中出现。广播发生在执行与 **single** 关联的结构化块后，组中任何线程在构造尾部留下障碍前。*list* 中的变量决不可在同一 **single** 指令的 **private** 或 **firstprivate** 子句中出现。

1.10.1.8 **REDUCTION** 子句

Fortran

REDUCTION(*operator*|*intrinsic*:*list*)

operator 为以下操作符之一：**+**、 ***、 -**、 **.AND.**、 **.OR.**、 **.EQV.**、 **.NEQV.**

intrinsic 为下列值之一：**MAX**、 **MIN**、 **IAND**、 **IOR**、 **IEOR**

list 中的变量必须为内在类型的命名变量。

C/C++

reduction(*operator*:*list*)

operator 为以下操作符之一：**+**、 ***、 -**、 **&**、 **^**、 **|**、 **&&**、 **||**

REDUCTION 子句专供约简变量只在约简语句中使用的区域中使用。*list* 中的变量在封装上下文中必须为 **SHARED**。为每个线程创建每个变量的专用副本，仿佛它便是 **PRIVATE**。在约简尾部，将原始值与每个专用副本的最终值合并来更新共享变量。

有关 **REDUCTION** 子句和构造完整的详细信息和限制，请参阅 OpenMP 规范的相应章节。

1.10.2 调度子句

SCHEDULE 子句指定 Fortran **DO** 循环或 C/C++ **for** 循环中的迭代是如何在组中的线程间分配的。表 1-1 显示哪些指令允许 **SCHEDULE** 子句。

对这些调度子句的使用有一些重要的限制。有关完整的详细信息，请参阅 Fortran 规范中的 2.3.1 节和 C/C++ 规范中的 2.4.1 节。

schedule(*type* [**,***chunk*]**)**

指定如何在组的线程间分配 **DO** 或 **for** 循环的迭代。*type* 可以是 **STATIC**、**DYNAMIC**、**GUIDED** 或 **RUNTIME** 之一。缺少 **SCHEDULE** 子句时， Sun Studio 编译器使用 **STATIC** 调度。 *chunk* 必须为整型表达式。

1.10.2.1 **STATIC** 调度

schedule(static[**,***chunk*]**)**

迭代被分为由 *chunk* 指定大小的块。这些块按线程号顺序，以循环方式静态地分配给组中的线程。未指定时系统会选择 *chunk*，迭代会被划分为大小近似相同的连续块，每个线程分配一块。

1.10.2.2 **DYNAMIC** 调度

schedule(dynamic[**,***chunk*]**)**

迭代被分为由 *chunk* 指定大小的块，并分配给等待的线程。每个线程都完成其迭代空间块时，会动态地获取下一组迭代。未指定 *chunk* 时，缺省值为 1。

1.10.2.3 **GUIDED** 调度

schedule(guided[**,***chunk*]**)**

使用 **GUIDED** 时，每分发一个迭代块，块大小便以指数方式递减一次。 *chunk* 指定每次分发的最小迭代数。（块大小是由取决于实现的公式确定的；请参阅第 4-2 页 “GUIDED：确定块大小”。）未指定 *chunk* 时，缺省值为 2.0。

1.10.2.4 **RUNTIME** 调度

schedule(runtime)

调度被延迟到运行时为止。调度*类型*和*块*大小将根据 **OMP_SCHEDULE** 环境变量的值来确定。（缺省值为 **SCHEDULE(STATIC)**。

1.10.3 **NUM_THREADS** 子句

OpenMP API 在 **PARALLEL**、**PARALLEL SECTIONS**、**PARALLEL DO**、**PARALLEL for** 和 **PARALLEL WORKSHARE** 指令上提供了 **NUM_THREADS** 子句。

num_threads(*scalar_integer_expression*)

指定某线程进入并行区域时组内所创建的线程数。*scalar_integer_expression* 是请求的线程数，它会代替通过先前调用 **OMP_SET_NUM_THREADS** 库函数定义的线程数或 **OMP_NUM_THREADS** 环境变量的值。如果启用了动态线程管理，请求便是要使用的*最大*线程数。

请注意， **num_threads** 不应用于后续区域。

1.10.4 指令中出现的子句

表 1-1 显示了可以在以下指令和 pragma 中出现的子句：

- **PARALLEL**
- **DO**
- **for**
- **SECTIONS**
- **SINGLE**
- **PARALLEL DO**
- **parallel for**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**

表 1-1 Pragma 及可以出现的子句

子句 /Pragma	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS	PARALLEL WORKSHARE[3]
IF	•				•	•	•
PRIVATE	•	•	•	•	•	•	•
SHARED	•				•	•	•
FIRSTPRIVATE	•	•	•	•	•	•	•
LASTPRIVATE		•	•		•	•	
DEFAULT	•				•	•	•
REDUCTION	•	•	•		•	•	•
COPYIN	•				•	•	•

表 1-1 Pragma 及可以出现的子句 *(续)*

子句 /Pragma	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS	PARALLEL WORKSHARE[3]
COPYPRIVATE				•[1]			
ORDERED		•			•		
SCHEDULE		•			•		
NOWAIT		•[2]	•[2]	•[2]			
NUM_THREADS	•				•	•	•

1. 仅限 Fortran：**COPYPRIVATE** 可以在 **END SINGLE** 指令中出现。
2. 对于 Fortran， **NOWAIT** 修饰符只出现在 **END DO**、 **END SECTIONS**、 **END SINGLE** 或 **END WORKSHARE** 指令中。
3. 只有 Fortran 支持 **WORKSHARE** 和 **PARALLEL WORKSHARE**。

1.11 OpenMP 运行时库例程

OpenMP 提供了一组用来控制和查询并行执行环境的可调用的库例程、一组通用锁定例程和两个可移植计时器例程。 Fortran 和 C/C++ OpenMP 规范中有完整的详细信息。

1.11.1 Fortran OpenMP 例程

Fortran 运行时库例程是外部过程。在以下概述中， *int_expr* 是标量整型表达式；*logical_expr* 是标量逻辑表达式。

返回 **INTEGER(4)** 和 **LOGICAL(4)** 的 **OMP_** 函数不是内在函数，必须进行正确声明。否则，编译器将假定其为 **REAL**。如 Fortran OpenMP 规范中所述，以下概述的 OpenMP Fortran 运行时库例程的接口声明由 Fortran 的包含文件 **omp_lib.h** 和 Fortran **MODULE omp_lib** 提供。

在引用这些库例程的每个程序单元中提供一个 **INCLUDE 'omp_lib.h'** 语句或 **#include "omp_lib.h"** 预处理程序指令或 **USE omp_lib** 语句。

使用 **-Xlist** 编译将报告所有类型不匹配情况。

整型参数 **omp_lock_kind** 定义在 **OMP_*_LOCK** 例程中用于简单锁定变量的 **KIND** 类型参数。

整型参数 **omp_nest_lock_kind** 定义在 **OMP_*_NEST_LOCK** 例程中用于可嵌套锁定变量的 **KIND** 类型参数。

整型参数 **openmp_version** 被定义为使用 **YYYYMM** 格式的预处理程序宏 **_OPENMP**。其中 **YYYY** 和 **MM** 是 OpenMP Fortran API 版本的年和月名称。

1.11.2 C/C++ OpenMP 例程

C/C++ 运行时库函数是外部函数。

头 **<omp.h>** 声明可用于控制和查询并行执行环境的两种类型的几个函数以及可用于同步数据访问的锁定函数。

类型 **omp_lock_t** 是能够代表锁定可用或线程拥有锁定的对象类型。这些锁定称为简单锁定。

类型 **omp_nest_lock_t** 是能够代表锁定可用或线程拥有锁定的对象类型。这些锁定称为可嵌套锁定。

1.11.3 运行时线程管理例程

有关详细信息，请参阅相应的 OpenMP 规范。

1.11.3.1 **OMP_SET_NUM_THREADS** 例程

设置用于未使用 **num_threads()** 子句指定后续并行区域的线程数。此调用只影响调用线程所遇到的同一级或内部嵌套级别的后续并行区域。

Fortran

```
SUBROUTINE OMP_SET_NUM_THREADS(int_expr)
```

C/C++

```
#include <omp.h>
void omp_set_num_threads(int num_threads);
```

1.11.3.2 **OMP_GET_NUM_THREADS** 例程

返回当前组中正在执行从中调用其的并行区域的线程的数量。

Fortran

```
INTEGER(4) FUNCTION OMP_GET_NUM_THREADS()
```

C/C++

```
#include <omp.h>
int omp_get_num_threads(void);
```

1.11.3.3 OMP_GET_MAX_THREADS 例程

如果在程序中此处遇到未使用 **num_threads()** 子句指定的活动并行区域，则返回将用于组成线程组的最大线程数。

Fortran

```
INTEGER(4) FUNCTION OMP_GET_MAX_THREADS()
```

C/C++

```
#include <omp.h>
int omp_get_max_threads(void);
```

1.11.3.4 OMP_GET_THREAD_NUM 例程

返回组内执行对此函数调用的线程的号码。此号码位于 0 和 OMP_GET_NUM_THREADS()-1 之间，0 为主线程。

Fortran

```
INTEGER(4) FUNCTION OMP_GET_THREAD_NUM()
```

C/C++

```
#include <omp.h>
int omp_get_thread_num(void);
```

1.11.3.5 OMP_GET_NUM_PROCS 例程

返回程序可用的处理器数。

Fortran

```
INTEGER(4) FUNCTION OMP_GET_NUM_PROCS()
```

C/C++

```
#include <omp.h>
int omp_get_num_procs(void);
```

1.11.3.6 **OMP_IN_PARALLEL** 例程

确定线程是否在并行区域的动态范围内执行。

Fortran

```
LOGICAL(4) FUNCTION OMP_IN_PARALLEL()
```

如果在活动并行区域的动态范围内调用，则返回 .TRUE.，否则将返回 .FALSE.。

C/C++

```
#include <omp.h>
int omp_in_parallel(void);
```

如果在活动并行区域的动态范围内调用，则返回非零值；否则，返回零值。

活动并行区域是指 **IF** 子句求值为 **TRUE** 的并行区域。

1.11.3.7 **OMP_SET_DYNAMIC** 例程

启用或禁用可用线程数的动态调整。（缺省情况下启用动态调整。）此调用只影响调用线程所遇到的同一级或内部嵌套级别的后续并行区域。

Fortran

```
SUBROUTINE OMP_SET_DYNAMIC(logical_expr)
```

logical_expr 的求值为 .TRUE 时启用动态调整；否则，禁用动态调整。

C/C++

```
#include <omp.h>
void omp_set_dynamic(int dynamic);
```

如果 *dynamic* 的求值为非零值，启用动态调整；否则，禁用动态调整。

1.11.3.8 **OMP_GET_DYNAMIC** 例程

确定在程序中此处是否启用了动态线程调整。

Fortran

```
LOGICAL(4) FUNCTION OMP_GET_DYNAMIC()
```

启用了动态线程调整时返回 .TRUE.；否则，返回 .FALSE.。

C/C++

```
#include <omp.h>
int omp_get_dynamic(void);
```

启用了动态线程调整时返回非零值；否则，返回零值。

1.11.3.9 OMP_SET_NESTED 例程

启用或禁用嵌套并行操作。此调用只影响调用线程所遇到的同一级或内部嵌套级别的后续并行区域。

Fortran

```
SUBROUTINE OMP_SET_NESTED(logical_expr)
```

logical_expr 的求值为 **.TRUE.** 时启用嵌套并行操作；否则，禁用嵌套并行操作。

C/C++

```
#include <omp.h>
void omp_set_nested(int nested);
```

nested 的求值为非零值时启用嵌套并行操作；否则，禁用嵌套并行操作。

缺省情况下，禁用嵌套并行操作。有关嵌套并行操作的信息，请参阅第 2 章。

1.11.3.10 OMP_GET_NESTED 例程

确定在程序中此处是否启用了嵌套并行操作。

Fortran

```
LOGICAL(4) FUNCTION OMP_GET_NESTED()
```

启用嵌套并行操作时返回 **.TRUE.**；否则，返回 **.FALSE.**。

C/C++

```
#include <omp.h>
int omp_get_nested(void);
```

启用嵌套并行操作时返回非零值；否则，返回零值。

有关嵌套并行操作的信息，请参阅第 2 章。

1.11.4 管理同步锁定的例程

支持两种类型的锁定：简单锁定和可嵌套锁定。可以在解锁前使用同一线程多次锁定可嵌套锁定，如果简单锁定已处于锁定状态，便不能再行锁定。简单锁定变量只能传递给简单锁定例程，嵌套锁定变量只能传递给嵌套锁定例程。

Fortran：

锁定变量 *var* 只能通过这些例程进行访问。为此，请使用参数 **OMP_LOCK_KIND** 和 **OMP_NEST_LOCK_KIND**（在 **omp_lib.h INCLUDE** 文件和 **omp_lib MODULE** 中定义）。例如，

```
INTEGER(KIND=OMP_LOCK_KIND)     ::var
INTEGER(KIND=OMP_NEST_LOCK_KIND) :: nvar
```

C/C++：

简单锁定变量的类型必须为 **omp_lock_t**，且只能通过这些函数来访问。所有简单函数都需要指向 **omp_lock_t** 类型的参数。

嵌套锁定变量的类型必须是 **omp_nest_lock_t**，同样所有嵌套锁定函数也都需要指向 **omp_nest_lock_t** 类型的参数。

1.11.4.1 OMP_INIT_LOCK 和 OMP_INIT_NEST_LOCK 例程

为后续调用初始化锁定变量。

Fortran

```
SUBROUTINE OMP_INIT_LOCK(var)
SUBROUTINE OMP_INIT_NEST_LOCK(nvar)
```

C/C++

```
#include <omp.h>
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

1.11.4.2 OMP_DESTROY_LOCK 和 OMP_DESTROY_NEST_LOCK 例程

删除锁定变量。

Fortran

```
SUBROUTINE OMP_DESTROY_LOCK(var)
SUBROUTINE OMP_DESTROY_NEST_LOCK(nvar)
```

C/C++

```
#include <omp.h>
void omp_destroy_lock(omp_lock_t *lock);
```

```
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

1.11.4.3 OMP_SET_LOCK 和 OMP_SET_NEST_LOCK 例程

强制正在执行的线程等到指定的锁定可用为止。锁定可用时，线程会被授予对锁定的所有权。

Fortran

```
SUBROUTINE OMP_SET_LOCK(var)
SUBROUTINE OMP_SET_NEST_LOCK(nvar)
```

C/C++

```
#include <omp.h>
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

1.11.4.4 OMP_UNSET_LOCK 和 OMP_UNSET_NEST_LOCK 例程

释放正在执行的线程对锁定的所有权。线程不拥有该锁定时，行为不确定。

Fortran

```
SUBROUTINE OMP_UNSET_LOCK(var)
SUBROUTINE OMP_UNSET_NEST_LOCK(nvar)
```

C/C++

```
#include <omp.h>
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

1.11.4.5 OMP_TEST_LOCK 和 OMP_TEST_NEST_LOCK 例程

OMP_TEST_LOCK 会尝试设置与锁定变量关联的锁定。调用不会阻碍线程的执行。

如果锁定设置成功，**OMP_TEST_NEST_LOCK** 返回新嵌套计数；否则，返回 0。调用不会阻碍线程的执行。

Fortran

```
LOGICAL(4) FUNCTION OMP_TEST_LOCK(var)
```

设置了锁定时返回 **.TRUE.**；否则，返回 **.FALSE.**。

```
INTEGER(4) FUNCTION OMP_TEST_NEST_LOCK(nvar)
```

锁定设置成功时返回嵌套计数；否则，返回零。

C/C++

```
#include <omp.h>
int omp_test_lock(omp_lock_t *lock);
```

锁定设置成功时返回非零值；否则，返回零值。

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

锁定设置成功时返回锁定嵌套计数；否则，返回零。

1.11.5 计时例程

两个函数支持可移植挂钟计时器。

1.11.5.1 **OMP_GET_WTIME** 例程

返回挂钟 “自过去任意时刻以来” 经过的时间 （秒）。

Fortran

```
REAL(8) FUNCTION OMP_GET_WTIME()
```

C/C++

```
#include <omp.h>
double omp_get_wtime(void);
```

1.11.5.2 **OMP_GET_WTICK** 例程

返回连续时钟滴答声之间的秒数。

Fortran

```
REAL(8) FUNCTION OMP_GET_WTICK()
```

C/C++

```
#include <omp.h>
double omp_get_wtick(void);
```

第 2 章

嵌套并行操作

本章讨论 OpenMP 嵌套并行操作特性。

2.1 执行模型

OpenMP 采用 fork-join （分叉 - 合并）并行执行模式。线程遇到并行构造时，就会创建由其自身及其他一些额外 （可能为零个）线程组成的线程组。遇到并行构造的线程成为新组中的主线程。组中的其他线程称为组的从属线程。所有组成员都执行并行构造内的代码。如果某个线程完成了其在并行构造内的工作，它就会在并行构造末尾的隐式屏障处等待。当所有组成员都到达该屏障时，这些线程就可以离开该屏障了。主线程继续执行并行构造之后的用户代码，而从属线程则等待被召集加入到其他组。

OpenMP 并行区域之间可以互相嵌套。如果禁用嵌套并行操作，则由遇到并行区域内并行构造的线程所创建的新组仅包含遇到并行构造的线程。如果启用嵌套并行操作，则新组可以包含多个线程。

OpenMP 运行时库维护一个线程池，该线程池中的线程可用作并行区域中的从属线程。当线程遇到并行构造并需要创建包含多个线程的线程组时，该线程将检查该池，从池中获取空闲线程，将其作为组的从属线程。如果池中没有足够的空闲线程，则主线程获取的从属线程可能会比所需的要少。组完成执行并行区域时，从属线程就会返回到池中。

2.2 控制嵌套并行操作

通过在执行程序前设置各种环境变量，可以在运行时控制嵌套并行操作。

2.2.1 OMP_NESTED

可通过设置 **OMP_NESTED** 环境变量或调用 omp_set_nested() 函数 （第 1-23 页第 1.11.3.9 节的 “OMP_SET_NESTED 例程”）来启用或禁用嵌套并行操作。

以下示例说明在启用嵌套并行操作时包含多个执行嵌套并行区域的线程的组。

代码样例 2-1 嵌套并行操作示例

```
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
    #pragma omp single
    {
        printf("Level %d:number of threads in the team - %d\n",
                 level, omp_get_num_threads());
    }
}

int main()
{
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(2)
        {
            report_num_threads(2);
            #pragma omp parallel num_threads(2)
            {
                report_num_threads(3);
            }
        }
    }
    return(0);
}
```

启用嵌套并行操作时，编译和运行此程序会产生以下输出：

```
% setenv OMP_NESTED TRUE
% a.out
Level 1:number of threads in the team - 2
Level 2:number of threads in the team - 2
Level 2:number of threads in the team - 2
Level 3:number of threads in the team - 2
Level 3:number of threads in the team - 2
Level 3:number of threads in the team - 2
Level 3:number of threads in the team - 2
```

比较禁用嵌套并行操作时运行相同程序的输出结果：

```
% setenv OMP_NESTED FALSE
% a.out
Level 1:number of threads in the team - 2
Level 2:number of threads in the team - 1
Level 3:number of threads in the team - 1
Level 2:number of threads in the team - 1
Level 3:number of threads in the team - 1
```

2.2.2 SUNW_MP_MAX_POOL_THREADS

OpenMP 运行时库维护一个线程池，该线程池中的线程可用作并行区域中的从属线程。设置 **SUNW_MP_MAX_POOL_THREADS** 环境变量可控制池中线程的数量。缺省值是 1023。

线程池只包含运行时库创建的非用户线程。它不包含主线程或用户程序显式创建的任何线程。如果将此环境变量设置为零，则线程池为空，并且的并行区域均由一个线程执行。

以下示例说明如果池中没有足够的线程，并行区域可能获取较少的线程。代码与上面的代码相同。使所有并行区域同时处于活动状态所需的线程数为 8 个。池需要包含至少 7 个空闲线程。如果将 **SUNW_MP_MAX_POOL_THREADS** 设置为 5，则四个最里面的并行区域中的两个区域可能无法获取所请求的所有从属线程。一种可能的结果如下所示。

```
% setenv OMP_NESTED TRUE
% setenv SUNW_MP_MAX_POOL_THREADS 5
% a.out
Level 1:number of threads in the team - 2
Level 2:number of threads in the team - 2
Level 2:number of threads in the team - 2
Level 3:number of threads in the team - 2
Level 3:number of threads in the team - 2
Level 3:number of threads in the team - 1
Level 3:number of threads in the team - 1
```

2.2.3 SUNW_MP_MAX_NESTED_LEVELS

环境变量 **SUNW_MP_MAX_NESTED_LEVELS** 控制需要多个线程的嵌套并行区域的最大深度。

活动嵌套深度大于此环境变量值的任何并行区域将仅由一个线程来执行。如果并行区域是具有假 IF 子句的 OpenMP 并行区域，则不会将该区域视为活动区域。

以下代码将创建 4 级嵌套并行区域。如果将 **SUNW_MP_MAX_NESTED_LEVELS** 设置为 2，则嵌套深度为 3 和 4 的嵌套并行区域将由单个线程来执行。

```
#include <omp.h>
#include <stdio.h>
#define DEPTH 5
void report_num_threads(int level)
{
    #pragma omp single
    {
        printf("Level %d:number of threads in the team - %d\n",
               level, omp_get_num_threads());
    }
}
void nested(int depth)
{
    if (depth == DEPTH)
        return;

    #pragma omp parallel num_threads(2)
    {
        report_num_threads(depth);
        nested(depth+1);
    }
}
int main()
{
    omp_set_dynamic(0);
    omp_set_nested(1);
    nested(1);
    return(0);
}
```

使用最大嵌套级别 4 来编译和运行此程序会产生以下可能的输出。（实际结果取决于操作系统调度线程的方式）。

```
% setenv SUNW_MP_MAX_NESTED_LEVELS 4
% a.out |sort +2n
Level 1:number of threads in the team - 2
Level 2:number of threads in the team - 2
Level 2:number of threads in the team - 2
Level 3:number of threads in the team - 2
Level 3:number of threads in the team - 2
Level 3:number of threads in the team - 2
Level 3:number of threads in the team - 2
Level 4:number of threads in the team - 2
Level 4:number of threads in the team - 2
Level 4:number of threads in the team - 2
Level 4:number of threads in the team - 2
Level 4:number of threads in the team - 2
Level 4:number of threads in the team - 2
Level 4:number of threads in the team - 2
Level 4:number of threads in the team - 2
```

使用设置为 2 的嵌套级别来运行产生以下可能的结果：

```
% setenv SUNW_MP_MAX_NESTED_LEVELS 2
% a.out |sort +2n
Level 1:number of threads in the team - 2
Level 2:number of threads in the team - 2
Level 2:number of threads in the team - 2
Level 3:number of threads in the team - 1
Level 3:number of threads in the team - 1
Level 3:number of threads in the team - 1
Level 3:number of threads in the team - 1
Level 4:number of threads in the team - 1
Level 4:number of threads in the team - 1
Level 4:number of threads in the team - 1
Level 4:number of threads in the team - 1
```

2.3 在嵌套并行区域中使用 OpenMP 库函数

在嵌套并行区域中调用以下 OpenMP 例程需要仔细斟酌。

- `omp_set_num_threads()`
- `omp_get_max_threads()`
- `omp_set_dynamic()`
- `omp_get_dynamic()`
- `omp_set_nested()`
- `omp_get_nested()`

“set”调用只影响调用线程所遇到的在同一级别或内部嵌套级别的并行区域。它们不影响其他线程遇到的并行区域，也不影响调用线程稍后在任何外部级别所遇到的并行区域。

“get”调用将返回由调用线程设置的值。创建组后，从属线程将继承主线程的值。

代码样例 2-2 并行区域中的 OpenMP 函数调用

```
#include <stdio.h>
#include <omp.h>
int main()
{
    omp_set_nested(1);
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2)
    {
        if (omp_get_thread_num() == 0)
            omp_set_num_threads(4);       /* 行 A */
        else
            omp_set_num_threads(6);       /* 行 B */

        /* 以下语句将打印
         *
         * 0: 2 4
         * 1: 2 6
         *
         * omp_get_num_threads() 返回组中的线程数
         * 因此，对于
         * 组中的两个线程来说情况是相同的。
         */
        printf("%d:%d %d\n", omp_get_thread_num(),
               omp_get_num_threads(),
               omp_get_max_threads());

        /* 将创建两个内部的并行区域
         * 一个区域带有包含 4 个线程的组；
         * 另一个区域带有包含 6 个线程的组。
         */
        #pragma omp parallel
        {
            #pragma omp master
            {
                /* 以下语句将打印
                 *
                 * 内部： 4
                 * 内部： 6
                 */
                printf("Inner:%d\n", omp_get_num_threads());
            }
            omp_set_num_threads(7);      /* line C */
        }
```

```
        /* 将再次创建两个内部的并行区域，
         * 一个区域带有包含 4 个线程的组；
         * 另一个区域带有包含 6 个线程的组。
         *
         * C 行的 omp_set_num_threads(7) 调用
         * 此处无效，因为它只影响
         * 与 C 行在相同级别或内部嵌套级别
         * 的并行区域。
         */

        #pragma omp parallel
        {
            printf("count me.\n");
        }
    }
    return(0);
}
```

编译和运行此程序会产生一种以下可能的结果：

```
% a.out
0: 2 4
Inner: 4
1: 2 6
Inner: 6
count me.
count me.
count me.
count me.
count me.
count me.
count me.
count me.
count me.
count me.
count me.
```

2.4 有关使用嵌套并行操作的一些提示

- 嵌套并行区域提供一种直接的方法来允许多个线程参与到计算中。

  例如，假定您的程序包含两级并行操作，并且每个级别的并行操作等级为 2。同时，您的系统有 4 个 CPU，您要使用所有 4 个 CPU 来加快此程序的执行速度。如果只并行化其中任意一个级别，则只需使用两个 CPU。您想要并行化两个级别。

- 嵌套并行区域容易创建过多的线程，从而占用过多的系统资源。适当地设置 **`SUNW_MP_MAX_POOL_THREADS`** 和 **`SUNW_MP_MAX_NESTED_LEVELS`** 以限制使用的线程数，防止系统资源枯竭。

- 创建嵌套并行区域会增加开销。如果在外部级别有足够的并行操作并且负载平衡，则通常在计算外部级别使用所有线程比在内部级别创建嵌套区域更有效。

  例如，假定您的程序包含两级并行操作。外部级别的并行操作等级为 4，并且负载平衡。您的系统具有四个 CPU，您要使用所有四个 CPU 来加快此程序的执行速度。那么，通常将所有 4 个线程用于外部级别比将 2 个线程用于外部并行区域而将其他 2 个线程用作内部并行区域的从属线程的性能要好。

第 3 章

Fortran 中的自动作用域

在 OpenMP 并行区域内声明变量的作用域属性称为*作用域*。通常，如果将一个变量的作用域确定为 **SHARED**，则所有线程共享该变量的一个副本。如果将一个变量的作用域确定为 **PRIVATE**，则每个线程拥有其自己的变量副本。OpenMP 拥有丰富的数据环境。除了 **SHARED** 和 **PRIVATE** 之外，还可以将变量的作用域声明为 **FIRSTPRIVATE**、**LASTPRIVATE**, **REDUCTION** 或 **THREADPRIVATE**。

OpenMP 要求用户声明在并行区域中使用的每个变量的作用域。这是一个单调乏味，极易出错的过程，并且公认是使用 OpenMP 并行化程序过程中最艰难的部分。

Fortran 95 编译器的 Sun Studio 9 发行版 **f95** 提供了自动作用域功能。该编译器不仅可以分析并行区域的执行和同步模式，而且可以基于一组作用域规则确定变量的作用域。

3.1 自动作用域数据范围子句

自动作用域数据范围子句是 Sun 对 Fortran OpenMP 规范的扩展。通过使用以下两种子句之一，用户可以指定要自动确定作用域的变量。

3.1.1 **__AUTO** 子句

__AUTO （*变量列表*）

该编译器将确定在并行区域内列出的各个变量的作用域。（注意 **AUTO** 之前的*两个*下划线）。

__AUTO 子句出现在 **PARALLEL**、**PARALLEL DO**、**PARALLEL SECTIONS** 或 **PARALLEL WORKSHARE** 指令中。

如果变量列在 **__AUTO** 子句中，则不能在其他任何数据作用域子句中将其指定。

3.1.2 **DEFAULT(__AUTO)** 子句

将此并行区域中缺省作用域设置为 **__AUTO**。

DEFAULT(__AUTO) 子句出现在 **PARALLEL**、**PARALLEL DO**、**PARALLEL SECTIONS** 或 **PARALLEL WORKSHARE** 指令中。

3.2 作用域规则

在自动作用域下，编译器应用以下规则来确定并行区域中变量的作用域。

这些规则并不适用于由 OpenMP 规范隐式确定作用域的变量，如共享任务 **DO** 循环的循环索引变量。

3.2.1 标量变量的作用域规则

- **S1**：如果对于组中执行区域的线程而言，在并行区域中使用变量的环境是自由的*数据争用*[1] 条件，则变量的作用域为 **SHARED**。
- **S2**：如果在每个执行并行区域的线程中，变量始终在相同线程读取之前写入，则将变量的作用域确定为 **PRIVATE**。如果变量的作用域确定为 **PRIVATE**，并且在其写入并行区域之前读取，而其构造为 **PARALLEL DO** 或 **PARALLEL SECTIONS**，则将其作用域确定为 **LASTPRIVATE**。
- **S3**：如果在编译器识别的约简操作中使用变量，则将该变量的作用域确定为具有此特定操作类型的 **REDUCTION**。

3.2.2 数组的作用域规则

- **A1**：对于组中执行区域的线程而言，如果在并行区域中使用变量的环境是自由的数据争用条件，则数组的作用域为 **SHARED**。

1. 当两个线程可以同时访问相同的共享变量（其中至少有一个线程可以修改该变量）时，存在*数据争用*。要删除数据争用条件，请将访问放入临界段或将线程同步。

3.3 关于自动作用域的通用注释

如果用户指定以下由 **__AUTO**（*变量列表*）或 **DEFAULT**（**__AUTO**）自动确定作用域的变量，则编译器将按照 OpenMP 规范中的隐含作用域规则确定变量的作用域。

- **THREADPRIVATE** 变量
- Cray *指针对象*。
- 循环迭代变量只能在区域词汇范围内的顺序循环或绑定到区域的共享任务 DO 循环中使用。
- 隐含的 **DO** 或 **FORALL** 索引。
- 变量只能在绑定到区域的工作共享构造中使用，并且在每个这种构造的数据作用域属性子句中指定。

自动确定没有隐含作用域的变量的作用域时，编译器将根据上述规则，按照显示的顺序检查变量的用法。如果符合某个规则，编译器将按照匹配的规则确定变量的作用域。如果不符合某个规则，编译器将尝试使用下一个规则。如果编译器找不到符合的规则，该变量作用域的自动确定将失败。

自动确定变量的作用域失败后，变量的作用域将被确定为 **SHARED**，并且绑定并行区域将被序列化，如同指定了 **IF (.FALSE.)** 子句。

作用域的自动确定失败的原因有两个。一个原因是使用的变量不匹配任何规则。第二个原因是源代码对于编译器来说过于复杂，因而无法执行全面的分析。函数调用、复杂的数组子脚本、内存别名和用户实现的同步都是常见原因。（请参阅第 3-7 页第 3.5 节的“当前实现的已知限制”。）

3.4 检查自动作用域的结果

使用*编译器注释*检查自动作用域结果，并且查看并行区域是否因为自动作用域失败而序列化。

使用 **-g** 调试选项进行编译时，编译器将生成内联注释。可以使用 **er_src** 命令查看这个生成的注释，如代码样例 3-2 所示。（**er_src** 命令作为 Sun Studio 软件的一部分提供；有关详细信息，请参阅 er_src(1) 手册页或 《*Sun Studio 性能分析器*》手册。）

使用 **-vpara** 选项进行编译是一个良好的开端。如果自动作用域失败，将打印一条警告消息（如代码样例 3-1 所示）。

代码样例 3-1 使用 **-vpara** 进行编译

```
>cat t.f
      INTEGER X(100), Y(100), I, T
C$OMP PARALLEL DO DEFAULT(__AUTO)
      DO I=1, 100
         T = Y(I)
         CALL FOO(X)
         X(I) = T*T
      END DO
C$OMP END PARALLEL DO
      END
>f95 -xopenmp -xO3 -vpara -c t.f
"t.f", line 3:Warning: 并行区域已序列化
       因为下列变量的自动作用域失败
       - x
```

代码样例 3-2 使用编译器注释

```
>cat t.f
      INTEGER X(100), Y(100), I, T
C$OMP PARALLEL DO DEFAULT(__AUTO)
      DO I=1, 100
         T = Y(I)
         X(I) = T*T
      END DO
C$OMP END PARALLEL DO
      END

>f95 -xopenmp -xO3 -g -c t.f
>er_src t.o
源文件：./t.f
目标文件：./t.o
加载对象：./t.o

     1.       INTEGER X(100), Y(100), I, T
     2.

OpenMP 构造中的专用变量如下：t,i
OpenMP 构造中的共享变量如下：y,x
OpenMP 构造中自动作用域为 PRIVATE 的变量如下：
       i, t
OpenMP 构造中自动作用域为 SHARED 的变量如下：
       y, x
     3. C$OMP PARALLEL DO DEFAULT(__AUTO)

由显式用户指令并行化以下循环
     4.       DO I=1, 100

以下循环调度时的稳态循环计数 = 3
以下循环展开了 2 次
以下循环每次迭代具有 1 次加载、1 次存储、0 次预取、0 次浮点加、0 次浮点减和
0 次浮点除
     5.          T = Y(I)
     6.          X(I) = T*T
     7.       END DO
     8. C$OMP END PARALLEL DO
     9.
    10.       END
```

接下来，使用一个更复杂的示例来解释自动作用域规则的应用方式。

代码样例 3-3 更复杂的示例

```
 1.       REAL FUNCTION FOO (N, X, Y)
 2.       INTEGER       N, I
 3.       REAL          X(*), Y(*)
 4.       REAL          W, MM, M
 5.
 6.       W = 0.0
 7.
 8. C$OMP PARALLEL DEFAULT(__AUTO)
 9.
10. C$OMP SINGLE
11.        M = 0.0
12. C$OMP END SINGLE
13.
14.        MM = 0.0
15.
16. C$OMP DO
17.        DO I = 1,N
18.           T = X(I)
19.           Y(I) = T
20.           IF ( MM .GT.T) THEN
21.              W = W + T
22.              MM = T
23.           END IF
24.        END DO
25. C$OMP END DO
26.
27. C$OMP CRITICAL
28.        IF ( MM .GT.M ) THEN
29.           M = MM
30.        END IF
31. C$OMP END CRITICAL
32.
33. C$OMP END PARALLEL
34.
35.       FOO = W - M
36.
37.       RETURN
38.       END
```

函数 **FOO()** 包含一个并行区域，其包含一个 **SINGLE** 构造、一个工作共享 **DO** 构造和一个 **CRITICAL** 构造。如果我们忽略所有 OpenMP 并行构造，则并行区域中使用的代码如下：

1. 将数组 **X** 中的值复制到数组 **Y**

2. 查找 **X** 中的最大正数，并将其存储在 **M** 中

3. **X** 一些元素的值累积为变量 **W**。

让我们来看一看编译器如何使用上述规则来查找并行区域中变量的适当作用域。

在并行区域中使用下列变量：**I**、**N**、**MM**、**T**、**W**、**M**、**X** 和 **Y**。编译器将确定以下内容。

- 标量 **I** 是工作共享 **DO** 循环的循环索引。 OpenMP 规范要求将 **I** 的作用域确定为 **PRIVATE**。
- 标量 **N** 在并行区域中是只读的，因此不会造成数据争用，按照以下规则 **S1**，将其作用域确定为 **SHARED**。
- 执行并行区域的任何线程将执行语句 14，该语句将标量 **MM** 的值设置为 0.0。这种写入方式将造成数据争用，因此规则 **S1** 不适用。在同一线程中读取 **MM** 之前出现这种写入方式，因此按照规则 **S2** 将 **MM** 的作用域确定为 **PRIVATE**。
- 同样，将标量 **T** 的作用域确定为 **PRIVATE**。
- 读取标量 **W**，然后写入语句 21，因此规则 **S1** 和 **S2** 不适用。加法运算是关联、相互通信的，因此按照规则 **S3** 将 **W** 的作用域确定为 **REDUCTION(+)**。
- 标量 **M** 写入语句 11，该语句位于 **SINGLE** 构造内。 **SINGLE** 构造末尾的隐式屏障可确保不在读取语句 28 或写入语句 29 的同时写入语句 11，而后面两者不会同时发生，因为它们都位于同一 **CRITICAL** 构造内。没有两个线程可以同时访问 **M**。因此，在并行区域中写入和读取 **M** 不会造成数据争用，按照规则 **S1**，将 **M** 的作用域确定为 **SHARED**。
- 数组 **X()** 是只读的，并且没有写入区域，因此按照规则 **A1** 将其作用域确定为 **SHARED**。
- 写入数组 **Y()** 分布在线程之间，没有两个线程可以写入相同的 **Y()** 元素。因为不存在数据争用，因此按照规则 **A1** 将 **Y()** 的作用域确定为 **SHARED**。

3.5 当前实现的已知限制

此处介绍了 Sun Studio 9 Fortran 95 编译器中自动作用域的已知限制。

- 只识别 OpenMP 指令，并且只能在分析中使用。不识别 OpenMP API 函数调用。例如，如果程序使用 **OMP_SET_LOCK()** 和 **OMP_UNSET_LOCK()** 来实现临界段，则编译器不能检测到是否存在临界段。如果可能的话，使用 **CRITICAL** 和 **END CRITICAL** 指令。
- 只有在 OpenMP 同步指令中指定的同步（如 **BARRIER** 和 **MASTER**）才能被识别，并且在分析中使用。不识别用户实现的同步，如忙等待。
- 使用 **-xopenmp=noopt** 编译时不支持自动作用域。

第 4 章

实现定义的行为

本章说明 OpenMP 2.0 Fortran 和 C/C++ 规范中依赖实现的某些特定问题。有关最新版编译器的最新信息，请参阅 C、 C++ 和 Fortran 自述文件。

- **调度**

  在缺少显式 **OMP_SCHEDULE** 环境变量或显式 **SCHEDULE** 子句的情况下，缺省值为 **static** 调度。

- **线程数**

  如果没有显式 **num_threads()** 子句、对 **omp_set_num_threads**() 函数的调用或 **OMP_NUM_THREADS** 环境变量的显式定义，组中缺省线程数为 1。

- **线程的动态调整**

  如果启用了动态调整，则将组内的线程数被调整为以下值中的最小值：

  用户请求的线程数

  1 + 池中可用线程数

  可用处理器数

  如果禁用了动态调整，则组内的线程数为以下值中的最小值：

  用户请求的线程数

  1 + 池中可用线程数

  如果提供的线程数少于用户所请求的数量，并且将 **SUNW_MP_WARN** 设置为 **TRUE** 或者通过调用 `sunw_mp_register_warn()` 来注册回调函数，则会报告警告消息。

  在异常情况下，如缺少系统资源，提供的线程数少于上述值。在这些情况下，如果禁用了动态调整并将 **SUNW_MP_WARN** 设置为 **TRUE** 或者通过调用 `sunw_mp_register_warn()` 来注册回调函数，则会报告警告消息。

  有关线程池和嵌套并行操作执行模型的更多信息，请参阅第 2 章。

- **嵌套并行操作**

  支持嵌套并行操作。可以由多个线程来执行嵌套并行区域。缺省情况下，禁用嵌套并行操作。要启用它，请设置 `OMP_NESTED` 环境变量或调用 `omp_set_nested()` 函数。请参阅第 2 章。

- **`ATOMIC` 指令**

  此实现在临界区域中封装目标语句来替换所有 **`ATOMIC`** 指令和 pragma。

- **`GUIDED`：确定块大小**

  如果未指定 *chunksize*，则 **`SCHEDULE(GUIDED)`** 的缺省块大小为 1。OpenMP 运行时库使用以下公式来计算使用 **`GUIDED`** 调度的循环的块大小。

  chunksize = *unassigned_iterations* / (*weight* * *num_threads*)

  其中：

  unassigned_iterations 是循环中尚未分配给任何线程的迭代数；

  weight 是可由用户在运行时使用 **`SUNW_MP_GUIDED_WEIGHT`** 环境变量设置的浮点常量（第 5-5 页第 5.3 节的“OpenMP 环境变量”）。如果未指定，则当前的缺省设置假定 *weight* 为 2.0；

  num_threads 是用于执行循环的线程数。

  加权值的选择会影响分配给循环中线程的迭代的初始块和后续块的大小，并且直接影响负载平衡。实验结果表明缺省加权值 2.0 通常效果比较好。但是，某些应用程序使用其他加权值可能会获得较好的效果。

- **显式线程程序**

  使用 POSIX 或 Solaris 线程的程序可以包含 OpenMP 指令或调用包含 OpenMP 指令的例程。

- **运行时警告**

  - 设置 **`SUNW_MP_WARN`** 环境变量（第 5-5 页第 5.3 节的“OpenMP 环境变量”）就会启用由 OpenMP 多任务库执行的运行时有效性检查。

例如，因为线程在不同的屏障处等待，所以以下代码陷入无限循环，必须从终端使用 Control-C 来终止它：

```
% cat bad1.c

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

int
main(void)
{
   omp_set_dynamic(0);
   omp_set_num_threads(4);

   #pragma omp parallel
   {
       int i = omp_get_thread_num();

       if (i % 2) {
           printf("At barrier 1.\n");
           #pragma omp barrier
       }
   }
   return 0;
}
% cc -xopenmp -xO3 bad1.c
% ./a.out                          run the program
At barrier 1.
At barrier 1.
                  program hung in endless loop
Control-C   to terminate execution
```

但是，如果在执行前设置了 **SUNW_MP_WARN**，运行时库将检测该问题：

```
% setenv SUNW_MP_WARN TRUE
% ./a.out
At barrier 1.
At barrier 1.
WARNING (libmtsk):Threads at barrier from different directives.
    Thread at barrier from bad1.c:11.
    Thread at barrier from bad1.c:17.
    Possible Reasons:
    Worksharing constructs not encountered by all threads in the team in the
       same order.
    Incorrect placement of barrier directives.
```

- C 编译器还提供了一个函数，可用于在检测到错误时注册回调函数。只要检测到错误，就会调用注册的回调函数，并将指向错误消息字符串的指针作为参数传递给它。

  ```
  int sunw_mp_register_warn(void (*func) (void *) )
  ```

  要访问此函数的原型，您需要添加
  `#include <sunw_mp_misc.h>`

例如：

```
% cat bad2.c
#include <omp.h>
#include <sunw_mp_misc.h>
#include <stdio.h>

void handle_warn(void *msg)
{
    printf("handle_warn:%s\n", (char *)msg);
}

void set(int i)
{
    static int k;
#pragma omp critical
    {
        k++;
    }
#pragma omp barrier
}

int main(void)
{
  int i, rc;
  omp_set_dynamic(0);
  omp_set_num_threads(4);
  if (sunw_mp_register_warn(handle_warn) != 0) {
      printf ("Installing callback failed\n");
  }
#pragma omp parallel for
  for (i = 0; i < 20; i++) {
      set(i);
  }
  return 0;
}

% cc -xopenmp -xO3 bad2.c
% a.out
handle_warn:WARNING (libmtsk):at bad2.c:21 Barrier is not
permitted in dynamic extent of for / DO.
```

如果 OpenMP 运行时库检测到错误，则将 `handle_warn()` 安装为回调处理程序函数。本示例中的处理程序只打印从库传递给它的错误消息，但可以将其用于捕获某些错误。

第 5 章

OpenMP 编译

本章介绍如何利用 OpenMP API 编译程序。

要在多线程环境下运行并行化的程序，必须在执行程序前设置 **OMP_NUM_THREADS** 环境变量。这通知运行时系统程序可以创建的最大线程数。缺省值为 1。一般将 **OMP_NUM_THREADS** 的值设置为不大于目标平台上可用的处理器数。可以将 **OMP_DYNAMIC** 设置为 **FALSE** 以使用 **OMP_NUM_THREADS** 指定的线程数。

编译器自述文件包含与其 OpenMP 实现有关的限制和已知缺陷方面的信息。使用 `-xhelp=readme` 标志调用编译器，或将 HTML 浏览器指向已安装软件的文档索引所在的以下路径，可直接查看自述文件

```
file:/opt/SUNWspro/docs/index.html
```

5.1 要使用的编译器选项

要使 OpenMP 指令可以进行显式并行化，请使用 **cc**、 **CC** 或 **f95** 选项标志 **-xopenmp** 编译程序。此标志可带有可选关键字参数。（**f95** 编译器将 **-xopenmp** 和 **-openmp** 均按同义字接受。）

-xopenmp 标志接受下列关键字子选项。

-xopenmp=parallel	启用 OpenMP pragma 的识别。-xopenmp=parallel 的最低优化级别是 -xO3。如有必要，编译器将优化级别从较低级别更改为 -xO3，并发出警告。
-xopenmp=noopt	启用 OpenMP pragma 的识别。如果级别低于 -xO3，编译器不会提高级别。 如果将优化级别显式地设置为低于 -xO3 的级别，如 -xO2 -openmp=noopt，编译器会报告错误。 如果没有使用 -openmp=noopt 指定优化级别，则识别 OpenMP 编译指示，并相应地并行化程序，但不执行优化。 （此子选项只适用于 cc 和 f95；指定时 CC 会发出警告，且不执行 OpenMP 并行化。）
-xopenmp=stubs	不再支持此选项。OpenMP 桩模块库是为了方便起见提供给用户的。要编译调用 OpenMP 库函数但忽略 OpenMP pragma 的 OpenMP 程序，请不要使用 -xopenmp 选项编译该程序，并使用 libompstubs.a 库链接对象文件。例如， **% cc omp_ignore.c -lompstubs** 不支持同时使用 libompstubs.a 和 OpenMP 运行时库 libmtsk.so 进行链接，这种链接方式可能会导致出现意外行为。
-xopenmp=none	禁用 OpenMP 编译指示的识别，并且不更改优化级别。

附加说明：

- 如果未在命令行指定 **-xopenmp**，编译器会假定使用 **-xopenmp=none**（禁用对 OpenMP pragma 的识别）。
- 如果指定 **-xopenmp** 时不带关键字子选项，编译器会假定使用 **-xopenmp=parallel**。
- 不要将 **-xopenmp** 与 **-xparallel** 或 **-xexplicitpar** 在命令行上一并指定。
- 指定 **-xopenmp=parallel** 或 **noopt** 将把 **_OPENMP** 预处理程序标记定义为 YYYYMM 格式（具体地讲，C/C++ 定义为 **200203L**，Fortran 95 定义为 **200011**）。
- 使用 dbx 调试 OpenMP 程序时，请使用 **-xopenmp=noopt -g** 进行编译。

- **-xopenmp** 的缺省优化级别在未来版本中可能会发生变化。显式地指定适当的优化级别可避免出现编译警告消息。
- 如果是 Fortran 95，**-xopenmp**、**-xopenmp=parallel**、**-xopenmp=noopt** 会自动添加 **-stackvar**。
- 如果在构建动态 (**.so**) 库时使用 **-xopenmp** 进行编译，则必须在链接可执行文件时同时指定 **-xopenmp**，并且用于创建可执行文件的编译器至少必须与使用 **-xopenmp** 构建动态库的编译器版本相同。使用带有 **-xopenmp** 不同版本的编译器来创建可执行文件和库将导致出现意外行为。

5.2 Fortran 95 OpenMP 验证

使用 **f95** 编译器的全局程序检查功能可以实现对 Fortran 95 程序的 OpenMP 指令的静态、过程间验证。使用 **-XlistMP** 标志进行编译来启用 OpenMP 检查。（来自 **-XlistMP** 的诊断消息会出现在一个单独的文件中，该文件的名称由源文件名和 **.lst** 扩展名构成）。编译器将诊断下列违规和并行化抑制因素：

- 并行指令规范中的违规，包括不当嵌套。
- 因使用数据而被过程间依存分析检测出的并行化抑制因素。
- 过程间指针分析检测出的并行化抑制因素。

例如，使用 -XlistMP 编译源文件 ord.f 会生成诊断文件 ord.lst：

```
FILE  "ord.f"
     1  !$OMP PARALLEL
     2  !$OMP DO ORDERED
     3                  do i=1,100
     4                          call work(i)
     5                  end do
     6  !$OMP END DO
     7  !$OMP END PARALLEL
     8
     9  !$OMP PARALLEL
    10  !$OMP DO
    11                  do i=1,100
    12                          call work(i)
    13                  end do
    14  !$OMP END DO
    15  !$OMP END PARALLEL
    16              end
    17                  subroutine work(k)
    18  !$OMP ORDERED
         ^
**** ERR-OMP:It is illegal for an ORDERED directive to bind to a
directive (ord.f, line 10, column 2) that does not have the
ORDERED clause specified.
    19                  write(*,*) k
    20  !$OMP END ORDERED
    21                  return
    22              end
```

本例中，**WORK** 子例程中的 **ORDERED** 指令收到有关第二个 **DO** 指令的诊断，因为该指令缺少 **ORDERED** 子句。

5.3 OpenMP 环境变量

OpenMP 规范定义了四个用来控制 OpenMP 程序执行的环境变量。下表对它们进行了概括。尽管其他多重处理环境变量也影响 OpenMP 程序的执行，但它们不是 OpenMP 规范的一部分。下表对它们进行了概括。

表 5-1 OpenMP 环境变量

环境变量	功能
OMP_SCHEDULE	为指定了 **RUNTIME** 调度类型的 **DO**、**PARALLEL DO**、**parallel for**、**for** 及指令 /pragma 设置调度类型。未定义时使用缺省值 **STATIC**。*值*为 *"type[,chunk]"* 示例：`setenv OMP_SCHEDULE "GUIDED,4"`
OMP_NUM_THREADS *或* **PARALLEL**	并行区域执行时设置需使用的线程数。**NUM_THREADS** 子句或 **OMP_SET_NUM_THREADS()** 调用可以覆盖该线程数。未设置时使用缺省值 1。*值*为正整数。为与传统程序兼容，设置 **PARALLEL** 环境变量和设置 **OMP_NUM_THREADS** 环境变量的效果相同。但如果将这两个环境变量都设置为不同的值，运行时库会发出一个错误消息。 示例：`setenv OMP_NUM_THREADS 16`
OMP_DYNAMIC	为并行区域的执行启用或禁用对可用线程数的动态调整。未设置时使用缺省值 **TRUE**。*值*为 **TRUE** 或 **FALSE**。 示例：`setenv OMP_DYNAMIC FALSE`
OMP_NESTED	启用或禁用嵌套并行操作。 *值*为 **TRUE** 或 **FALSE**。缺省值为 **FALSE**。 示例：`setenv OMP_NESTED FALSE`

表 5-2　多重处理环境变量

环境变量	功能
SUNW_MP_WARN	控制 OpenMP 运行时库发出的警告消息。设置为 **TRUE** 时，运行时库会给 stderr 发出警告消息；设置为 **FALSE** 时禁用警告消息。缺省值为 **FALSE**。 OpenMP 运行时库可以检查很多常见的 OpenMP 违规行为，如错误的嵌套和死锁。运行时检查会增加程序执行的开销。请参阅第 4-2 页 “运行时警告”。 示例： **setenv SUNW_MP_WARN TRUE**
SUNW_MP_THR_IDLE	控制执行程序的并行部分的每个帮助程序线程的任务结束状态。可以将值设置为 **SPIN**、**SLEEP** *n***s** 或者 **SLEEP** *n***ms**。缺省为 **SLEEP** - 线程在完成并行任务后进入休眠状态，直到有新的并行任务到达为止。 选择 **SLEEP** *time* 指定完成并行任务后帮助程序线程应旋转等待的时间。如果在线程空转期间此线程有新任务到达，此线程会立即执行新任务。否则，线程便进入休眠状态，新任务到达时再被唤醒。*time* 可以秒、（*n***s**）或只使用（*n*）、或毫秒、（*n***ms**）为单位指定。 不带参数的 **SLEEP** 在完成并行任务后即将线程置于休眠状态。**SLEEP**、**SLEEP (0)**、**SLEEP (0s)** 和 **SLEEP (0ms)** 均等价。 示例： **setenv SUNW_MP_THR_IDLE SLEEP (50ms)**
SUNW_MP_PROCBIND	**SUNW_MP_PROCBIND** 环境变量可用于将 OpenMP 程序的 LWP（轻量进程）绑定到处理器。虽然可以通过处理器绑定来增强性能，但是如果将多个 LWP 绑定到相同的处理器，则将导致性能下降。详细信息，请参阅第 5-7 页第 5.4 节的 “处理器绑定”。
SUNW_MP_MAX_POOL_THREADS	指定线程池的最大大小。线程池只包含 OpenMP 运行时库创建的非用户线程。它不包含主线程或用户程序显式创建的任何线程。如果将此环境变量设置为零，则线程池为空，并且将由一个线程执行所有的并行区域。如果未指定，则缺省值是 1023。有关详细信息，请参阅第 2-1 页第 2.2 节的 “控制嵌套并行操作”。

表 5-2　多重处理环境变量 *（续）*

环境变量	功能
SUNW_MP_MAX_NESTED_LEVELS	指定活动嵌套并行区域的最大深度。活动嵌套深度大于此环境变量值的任何并行区域将仅由一个线程来执行。如果并行区域是具有假 **IF** 子句的 OpenMP 并行区域，则不会将该区域视为活动区域。如果未指定，则缺省值是 4。有关详细信息，请参阅第 2-1 页第 2.2 节的 “控制嵌套并行操作”。
STACKSIZE	设置每个线程的栈大小。值以千字节为单位。缺省线程栈大小在 32 位 SPARC V8 和 x86 平台上为 4 兆字节，在 64 位 SPARC V9 和 x86 平台上为 8 兆字节。 示例： **setenv STACKSIZE 8192** *将线程栈大小设置为 8 兆字节*
SUNW_MP_GUIDED_WEIGHT	设置加权因子，这些因子用于确定在使用 **GUIDED** 调度的循环中为线程分配的块的大小。该值应该是正浮点数，并且应用于在程序中使用 **GUIDED** 调度的所有循环。如果未设置，则假定的缺省值为 2.0。

5.4 处理器绑定

处理器绑定与静态调度一起使用时，将有益于展示某个数据重用模式的应用程序，在该应用程序中，由并行区域中的线程访问的数据将位于上一次所调用并行区域的本地缓存中。

缺省情况下，轻量进程 LWP 不会绑定到处理器。此任务会移交到 Solaris OS，通过它来调度处理器上的 LWP。OpenMP 运行时库 `libmtsk` 中的多任务例程始终使用一对一的线程模型；即每个线程均与一个 LWP 相对应。

SUNW_MP_PROCBIND 环境变量指定的值表示 LWP 将绑定到的 “逻辑”处理器标识符 (ID)。逻辑处理器 ID 是从 0 开始的连续整数，与实际的处理器 ID 可能相同，也可能不相同。如果在线可用 *n* 个处理器，则它们的虚拟处理器 ID 是 0、1、 ...、 *n*-1，采用 `psrinfo`(1M) 显示的顺序。

逻辑处理器 ID 和实际处理器 ID 之间的映射取决于系统。在大多数系统上，实际处理器 ID 是有序的；然而，删除系统板可能会造成范围内出现空缺。在某些系统上， ID 采用 4 个为一组的形式，其中每组开头之间的间隙是 32 ；因此，处理器的编号可以为 0、1、2、 3、 32、 33、 34、 35，依次类推。

`libmtsk` 创建的线程数是由用户程序中的环境变量和 / 或 API 调用决定的。**SUNW_MP_PROCBIND** 指定一组逻辑处理器，如下所述。LWP 以循环的方式绑定到这组逻辑处理器上。如果 LWP 的数量比处理器少，则一些处理器就不会绑定 LWP。如果 LWP 的数量比处理器多，则一些处理器会绑定多个 LWP。

为 **SUNW_MP_PROCBIND** 指定的值可以是下列值之一：

- 字符串 **TRUE** 或 **FALSE**（或小写字母）。例如，
  ```
  % setenv SUNW_MP_PROCBIND false
  ```
- 非负整数。例如，
  ```
  % setenv SUNW_MP_PROCBIND 2
  ```
- 由一个或多个空格来分隔两个或多个非负整数的列表。
  例如，
  ```
  % setenv SUNW_MP_PROCBIND "0 2 4 6"
  ```
- 两个非负整数 *n1* 和 *n2* 之间由减号（“-”）分隔； *n1* 必须小于或等于 *n2*。例如，
  ```
  % setenv SUNW_MP_PROCBIND "0-6"
  ```

如果为 **SUNW_MP_PROCBIND** 指定的值是 **FALSE**，则不执行处理器绑定。这是缺省行为。

如果为 **SUNW_MP_PROCBIND** 指定的值是 **TRUE**，则该值就相当于整数 0。

如果为 **SUNW_MP_PROCBIND** 指定的值是非负整数，则该整数指定 LWP 应该绑定到的起始逻辑处理器 ID。LWP 将以循环的方式绑定到处理器，以指定的逻辑处理器 ID 开头，并且在逻辑处理器 ID *n*-1 之后环绕逻辑处理器 ID 0。

如果为 **SUNW_MP_PROCBIND** 指定的值是两个或多个非负整数的列表，则 LWP 将以循环的方式绑定到指定的逻辑处理器 ID。将不会使用列表以外的 ID。

如果为 **SUNW_MP_PROCBIND** 指定的值是两个用减号（“-”）分隔的非负整数，则 LWP 将以循环的方式绑定到处理器，处理器位于以第一个逻辑处理器 ID 开头，并以第二个逻辑处理器 ID 结尾的范围内。将不会使用此范围之外的 ID。

如果为 **SUNW_MP_PROCBIND** 指定的值不符合上述形式之一，或如果指定无效的逻辑处理器 ID，则将忽略环境变量 **SUNW_MP_PROCBIND**，并且 LWP 将不绑定到处理器。如果已启用警告，则在这种情况下将发出一条警告消息。

5.5 栈和栈大小

正在执行的程序为执行该程序的初始线程保留主内存栈，并为每个从属线程保留不同的栈。栈是临时内存地址空间，用于保留子程序或函数引用调用期间的参数和自动变量。

主栈的缺省大小一般约为 8 兆字节。使用 `f95` **-stackvar** 选项编译 Fortran 程序会强制将局部变量和数组像自动变量一样在栈中分配。显式并行化的程序暗指对 OpenMP 程序使用 **-stackvar**，因为该选项会提高优化器在循环中并行化调用的能力。（请参阅 *《Fortran 用户指南》* 中有关 **-stackvar** 标志的论述。）但如果分配给栈的内存不足，这样会导致栈溢出。

使用 **limit** C-shell 命令或 `ulimit` ksh/sh 命令来显示或设置主栈的大小。

OpenMP 程序的每个从属线程均具有其自身的线程栈。此栈模拟初始（或主）线程栈，但归线程专有。线程的 **PRIVATE** 数组和变量（线程的局部变量）在线程栈中分配。在 32 位 SPARC V8 和 x86 平台上的缺省大小为 4 兆字节；在 64 位 SPARC V9 和 x86 平台上为 8 兆字节。帮助程序线程栈的大小使用 **STACKSIZE** 环境变量来设置。

```
demo% setenv STACKSIZE 16384     <- 将线程栈大小设置为 16 兆字节 (C shell)

demo% STACKSIZE=16384            <- 相同，使用 Bourne/Korn shell
demo% export STACKSIZE
```

可能需要反复试验才能确定最佳栈大小。如果栈小到不足以满足线程的运行需要，可能会导致邻近线程中发生静态数据损坏或段故障。如果无法确定是否有栈溢出，请使用 **-xcheck=stkovf** 标志编译 Fortran、C 或 C++ 程序来强制栈溢出时发生段故障。这样便可以在发生数据损坏前停止程序。

第 6 章

转换为 OpenMP

本章提供使用 Sun 或 Cray 指令和 pragma 将传统程序转换为 OpenMP 的指导。

6.1 转换传统 Fortran 指令

传统 Fortran 程序使用 Sun 或 Cray 风格的并行化指令。*《Fortran 编程指南》*的*并行化*一章中有对这些指令的描述。

6.1.1 转换 Sun 风格的 Fortran 指令

以下表格提供与 Sun 并行化指令及其子子句近似等效的 OpenMP 指令和子子句。这些只是建议值。

表 6-1 将 Sun 并行化指令转换为 OpenMP

Sun 指令	**等效 OpenMP 指令**
`C$PAR DOALL` *[qualifiers]*	`!$omp parallel do` *[qualifiers]*

表 6-1　将 Sun 并行化指令转换为 OpenMP *(续)*

Sun 指令	等效 OpenMP 指令
C$PAR DOSERIAL	无完全等效指令。可以使用： !$omp master *loop* !$omp end master
C$PAR DOSERIAL*	无完全等效指令。可以使用： !$omp master *loopnest* !$omp end master
C$PAR TASKCOMMON *block[,...]*	!$omp threadprivate (/block/*[,...]*)

DOALL 指令可带有下列可选限定符子句。

表 6-2　DOALL 限定符子句和等效的 OpenMP 子句

Sun DOALL 子句	等效的 OpenMP PARALLEL DO 子句
PRIVATE(*v1,v2,...*)	private(*v1,v2,...*)
SHARED(*v1,v2,...*)	shared(*v1,v2,...*)
MAXCPUS(*n*)	num_threads(*n*)。无完全等效指令。
READONLY(*v1,v2,...*)	无完全等效指令。使用 firstprivate(*v1,v2,...*) 可以获得相同效果。
STOREBACK(*v1,v2,...*)	lastprivate (*v1,v2,...*)。
SAVELAST	无完全等效指令。使用 lastprivate(*v1,v2,...*) 可以获得相同效果。
REDUCTION(*v1,v2,...*)	reduction(operator:*v1,v2,...)* 必须提供约简操作符和变量列表。
SCHEDTYPE(*spec*)	schedule(*spec*)（请参阅表 6-3）

SCHEDTYPE(*spec*) 子句接受下列调度规范。

表 6-3　SCHEDTYPE 调度和等效的 OpenMP schedule

SCHEDTYPE(spec)	等效的 OpenMP schedule(*spec*) 子句
SCHEDTYPE(STATIC)	schedule(static)

表 6-3　SCHEDTYPE 调度和等效的 OpenMP schedule *(续)*

SCHEDTYPE(spec)	等效的 OpenMP schedule(*spec*) 子句
SCHEDTYPE(SELF(*chunksize*))	schedule(dynamic,*chunksize*) 缺省 *chunksize* 为 1。
SCHEDTYPE(FACTORING(*m*))	无完全等效指令。
SCHEDTYPE(GSS(*m*))	schedule(guided, *m*) 缺省 *m* 为 1。

6.1.1.1 Sun 风格的 Fortran 指令和 OpenMP 指令间的问题

- 私有变量作用域必须使用 OpenMP 加以显式声明。对于 Sun 指令，编译器对未在 **PRIVATE** 或 **SHARED** 子句中显式确定作用域的变量使用其自己的缺省作用域规则：所有标量均按 **PRIVATE** 处理；所有数组引用均按 **SHARED** 处理。对于 OpenMP，除非 **PARALLEL DO** 指令中出现 **DEFAULT(PRIVATE)** 子句，否则缺省数据作用域为 **SHARED**。 **DEFAULT(NONE)** 子句会使编译器标记那些未显式确定作用域的变量。有关在 Fortran 中自动确定作用域的信息，请参阅第 3 章。
- 由于没有 **DOSERIAL** 指令，因此混合自动和显式 OpenMP 并行化的结果可能会不同：某些使用 Sun 指令不能自动并行化的循环可能会被自动并行化。
- OpenMP 提供并行区域和并行段来提供更丰富的并行操作模型。使用 Sun 指令来重新设计程序的并行操作策略，以利用 OpenMP 的这些功能，便有可能获得更高的性能。

6.1.2 转换 Cray 风格的 Fortran 指令

Cray 风格的 Fortran 并行化指令与 Sun 风格的并行化指令几乎完全相同，只不过标识这些指令的标记是 **!MIC$**。此外， **!MIC$ DOALL** 上的限定符子句集也不同。

表 6-4　Cray 风格的 DOALL 限定符子句的等效 OpenMP 子句

Cray DOALL 子句	等效的 OpenMP PARALLEL DO 子句
SHARED(*v1,v2,...*)	SHARED(*v1,v2,...*)
PRIVATE(*v1,v2,...*)	PRIVATE(*v1,v2,...*)
AUTOSCOPE	无等效子句。确定作用域必须是显式的，或者使用 **DEFAULT** 子句或 **__AUTO** 子句
SAVELAST	无完全等效指令。使用 lastprivate 可以获得相同效果。
MAXCPUS(n)	num_threads(*n*) . 无完全等效指令。
GUIDED	schedule(guided, *m*) 缺省 *m* 为 1。

表 6-4　Cray 风格的 DOALL 限定符子句的等效 OpenMP 子句 *(续)*

Cray DOALL 子句	等效的 OpenMP PARALLEL DO 子句
SINGLE	schedule(dynamic,1)
CHUNKSIZE(n)	schedule(dynamic,*n*)
NUMCHUNKS(m)	schedule(dynamic,*n/m*) 其中 *n* 为迭代次数

6.1.2.1 Cray 风格的 Fortran 指令和 OpenMP 指令间的问题

差别基本上与和 Sun 风格指令的差别相同，只不过没有与 Cray AUTOSCOPE 等效的指令。

6.2 转换传统 C Pragma

C 编译器接受传统 pragma 来进行显式并行化。*《C 用户指南》*中有对这些内容的描述。与 Fortran 指令相同，这些只是建议值。

传统并行化 pragma 为：

表 6-5　将传统 C 并行化 Pragma 转换为 OpenMP

传统 C Pragma	等效的 OpenMP Pragma
#pragma MP taskloop *[clauses]*	#pragma omp parallel for *[clauses]*
#pragma MP serial_loop	无完全等效指令。可以使用 #pragma omp master *循环*
#pragma MP serial_loop_nested	无完全等效指令。可以使用 #pragma omp master *loopnest*

taskloop pragma 可带有一个或多个下列可选子句。

表 6-6　taskloop 可选子句和等效的 OpenMP 子句

taskloop 子句	等效的 OpenMP parallel for 子句
maxcpus(*n*)	无完全等效指令。使用 num_threads(*n*)
private(*v1,v2,...*)	private(*v1,v2,...*)
shared(*v1,v2,...*)	shared(*v1,v2,...*)

表 6-6　taskloop 可选子句和等效的 OpenMP 子句

taskloop 子句	等效的 OpenMP parallel for 子句
readonly(*v1,v2,...*)	无完全等效指令。使用 firstprivate(*v1,v2,...*) 可以获得相同效果。
storeback(*v1,v2,...*)	使用 lastprivate(*v1,v2,...*) 可以获得相同效果。
savelast	无完全等效指令。使用 lastprivate(*v1,v2,...*) 可以获得相同效果。
reduction(*v1,v2,...*)	reduction(operator:*v1,v2,...*)。必须提供约简操作符和变量列表。
schedtype(*spec*)	schedule(*spec*)（请参阅表 6-7）

schedtype(*spec*) 子句接受下列调度规范。

表 6-7　SCHEDTYPE 调度和等效的 OpenMP schedule

schedtype(spec)	等效的 OpenMP schedule(*spec*) 子句
SCHEDTYPE(STATIC)	schedule(static)
SCHEDTYPE(SELF(*chunksize*))	schedule(dynamic,*chunksize*) 注意：缺省 *chunksize* 为 1。
SCHEDTYPE(FACTORING(*m*))	无完全等效指令。
SCHEDTYPE(GSS(*m*))	schedule(guided, *m*) 缺省 *m* 为 1。

6.2.1　传统 C Pragma 与 OpenMP 间的问题

- OpenMP 作用域变量在并行构造内声明为 **private**。**#pragma omp parallel** for 指令中的 **default(none)** 子句会使编译器标记未显式确定作用域的变量。
- 由于没有 **serial_loop** 指令，因此混合自动和显式 OpenMP 并行化的结果可能会不同：某些使用传统 C 指令不能自动并行化的循环可能会被自动并行化。
- 由于 OpenMP 提供了更丰富的并行操作模型，因此使用传统 C 指令来重新设计程序的并行操作策略，以利用这些功能，往往可能会获得更高的性能。

第 7 章

性能注意事项

拥有正确、可执行的 OpenMP 程序之后，应该考虑其整体性能。您可以利用一些常规技术和 Sun 平台专有技术来改善 OpenMP 应用程序的效率和可伸缩性。我们将在此进行简单地介绍。

有关详细信息，请参阅 *Techniques for Optimizing Applications：High Performance Computing*， Rajat Garg 和 Ilya Sharapov 编著，可以从 `http://www.sun.com/books/catalog/garg.xml` 获得。

此外，有关 OpenMP 应用程序的性能分析和优化方面的参考文章和案例研究，请访问 Sun 开发人员门户网站，网址是 `http://developers.sun.com/prodtech/cc/`。

7.1 一般性建议

以下技术是用于改善 OpenMP 应用程序性能的一些常规技术。

- 将同步降至最低。
  - 避免或尽量不使用 **BARRIER**、 **CRITICAL** 区、 **ORDERED** 区域和锁定。
  - 使用 **NOWAIT** 子句可能消除冗余或不必要的障碍。例如，在并行区域末端总是有一个隐含的障碍。将 **NOWAIT** 添加到区域最后的 **DO** 中可以消除一个冗余的障碍。
  - 使用已命名的 **CRITICAL** 段进行细化锁定。
  - 使用显式 **FLUSH** 时要非常小心。刷新将造成数据高速缓存恢复到内存，而随后的数据访问可能需要从内存重新加载，从而会降低效率。
- 如果并行区域中的 **SHARED** 变量由执行该区域的线程读取，但是不通过任何线程写入，则将该变量指定为 **FIRSTPRIVATE**，而不是 **SHARED**。从而可以避免通过非关联化指针访问变量，以及避免出现高速缓存冲突。
- 通过请求比计划使用的线程数多的线程，可以避免浪费资源。使用 **SUNW_MP_THR_IDLE** 进行实验，以在不需要时将空转工作线程置于休眠状态。请参阅第 5-5 页第 5.3 节的 “OpenMP 环境变量”。

- 以尽可能高的级别并行化，如外部 **DO/FOR** 循环。在一个并行区域中封闭多个循环。通常，使并行区域尽可能大以降低并行化开销。例如：

此构造效率较低：

```
!$OMP PARALLEL
  ....
  !$OMP DO
    ....
  !$OMP END DO
  ....
!$OMP END PARALLEL

!$OMP PARALLEL
  ....
   !$OMP DO
     ....
   !$OMP END DO
  ....
!$OMP END PARALLEL
```

此构造效率较高：

```
!$OMP PARALLEL
  ....
  !$OMP DO
    ....
  !$OMP END DO
  .....

  !$OMP DO
    ....
  !$OMP END DO

!$OMP END PARALLEL
```

- 在并行区域中使用 **PARALLEL DO/FOR** 指令，而不是工作共享 **DO/FOR** 指令。与可能包含几个循环的常规并行区域相比，可以更有效地实现 **PARALLEL DO/FOR**。例如：

此构造效率较低：

```
!$OMP PARALLEL
  !$OMP DO
    .....
  !$OMP END DO
!$OMP END PARALLEL
```

此构造效率较高：

```
!$OMP PARALLEL DO
   ....
!$OMP END PARALLEL
```

- 使用 **SUNW_MP_PROCBIND** 将轻量进程 (LWP) 绑定到处理器。处理器绑定与静态调度一起使用时，将有益于展示某个数据重用模式的应用程序，在该应用程序中，由并行区域中的线程访问的数据将位于上一次所调用并行区域的本地缓存中。请参阅第 5-7 页第 5.4 节的 “处理器绑定”。
- 尽可能使用 **MASTER**，而不是 **SINGLE**。
  - **MASTER** 指令按照不带隐式 **BARRIER** 的 **IF** 语句来实现：
    IF(omp_get_thread_num() == 0) {...}
  - **SINGLE** 指令的实现方式类似于其他工作共享构造。跟踪哪个线程首先到达 **SINGLE** 将添加额外的运行时开销。如果未指定 **NOWAIT**，则存在一个隐式 **BARRIER**。这样效率较低。
- 选择适当的循环调度。
  - **STATIC** 不会造成同步开销，并且可以在数据适合高速缓存时保持数据的局域性。但是 **STATIC** 可能导致负载失衡。
  - **DYNAMIC**，**GUIDED** 由于要跟踪已经分配了哪些块，因此会发生同步开销。虽然这些调度会导致数据局域性较差，但是可以改善负载平衡。使用不同的块大小进行实验。
- 使用 **LASTPRIVATE** 时要非常小心，因为其有可能导致很高的开销。
  - 从并行构造返回时，数据需要从专用区复制到共享存储区。
  - 编译器代码检查哪个线程逻辑上执行最后一个迭代。从而将在并行 **DO**/**FOR** 中每个块的末尾添加额外的工作。如果块数很多，开销将会增加。
- 使用有效的线程安全内存管理。
  - 应用程序可以在编译器生成的代码中显式或隐式使用 **malloc()** 和 **free()**，以支持动态 / 可分配数组、向量化内例程等。

- **libc** 中的线程安全 **malloc()** 和 **free()** 由内部锁定造成高同步开销。可以在 **libmtmalloc** 库中找到更快的版本。用 **-lmtmalloc** 进行链接以使用 **libmtmalloc**。

7.2 伪共享及其避免方法

不小心使用 OpenMP 应用程序共享的内存结构将导致性能下降及可伸缩性受限。多个处理器更新内存中相邻共享数据将导致多处理器互连的通信过多，因而造成计算序列化。

7.2.1 何为*伪共享*?

大多数高性能处理器（如 UltraSPARC III）在 CPU 的慢速内存和高速寄存器之间插入一个高速缓存缓冲区。访问内存位置要求将包含该内存位置的一部分实际内存（*缓存代码行*）复制到高速缓存。随后可能在高速缓存外即可满足对同一内存位置或其周围位置的引用，直至系统决定有必要保持高速缓存和内存之间的一致性，并将缓存代码行恢复到内存。

然而，同时更新来自不同处理器的相同缓存代码行中的单个元素会使整个缓存代码行无效，即使这些更新在逻辑上是彼此独立的。对缓存代码行的单个元素进行更新会将此代码行标记为*无效*。其他访问同一代码行中不同元素的处理器将看到该代码行已标记为*无效*。即使所访问的元素未被修改，也会强制它们从内存提取该代码行的刷新副本。这是因为基于缓存代码行保持缓存一致性，而不是针对单个元素的。因此，互连通信和开销方面都将所有增长。并且，正在进行缓存代码行更新的时候，禁止访问该代码行中的元素。

这种情况称为*伪共享*，这是造成 OpenMP 应用程序性能和可伸缩性显著下降的主要原因。

在发生以下所有条件时，伪共享会使性能下降。

- 由多个处理器修改共享数据。
- 多个处理器更新同一缓存代码行中的数据。
- 这种更新发生的频率非常高（例如，在紧凑循环中）。

请注意，在循环中只读状态的共享数据不会导致伪共享。

7.2.2 减少伪共享

在执行应用程序时，对占据主导地位的并行循环进行仔细分析即可揭示伪共享造成的性能可伸缩性问题。通常可以通过以下方式减少伪共享

- 将共享数据的结构和用法更改为专用数据，

- 增加问题量 （迭代长度），
- 更改迭代到处理器的映射，以针对每个迭代赋予每个处理器更多的工作 （*块大小*），
- 利用编译器的优化功能来消除内存加载和存储。

7.3 操作系统调节功能

从 Solaris 9 发行版本开始，操作系统为 SunFire™ 系统提供了可伸缩性和较高的性能。在 Solaris 9 操作系统中引入了新的特性，这些特性无需进行硬件升级即可提高 OpenMP 程序性能，这些特性包括内存定位优化 (MPO) 和多页大小支持 (MPSS)。

MPO 功能允许操作系统分配页，这些页位于访问它们的处理器附近。 SunFire 6800、SunFire 15K 和 SunFire E25K 系统在相同的 UniBoard™ 中及不同的 UniBoards 之间有不同的内存延时。称为 *初次接触*的缺省 MPO 策略在包含第一次接触内存的处理器的 UniBoard 上分配内存。对于大部分数据访问是针对每个处理器 （处于初次接触定位状态）的局部内存的应用程序，初次接触策略可以显著提高该应用程序的性能。与内存平均分布在系统上的随机内存定位策略相比，此策略即可以降低应用程序的内存延迟，又能提高带宽，从而获得更高的性能。

MPSS 功能允许程序针对虚拟内存的不同区域使用不同的页大小。 Solaris 9 操作系统的缺省页大小是 8KB。采用 8KB 的页大小，使用大内存的应用程序会有大量 TLB 缺失，因为 UltraSPARC™ III Cu 和 UltraSPARC IV 上 TLB 的条目数只允许访问几兆字节内存。 UltraSPARC III Cu 和 UltraSPARC IV 支持四种不同的页大小：8 KB、64 KB、512 KB 和 4MB。运用 MPSS，用户进程可以请求这四种页大小之一。因此 MPSS 可以显著降低 TLB 的缺失数，并提高使用大量内存的应用程序的性能。

索引

X

Z