Oracle® Solaris Studio 12.3 ： OpenMP API 用户指南

文件号码 E26443
2011 年 12 月，E26443-01

ORACLE

120220@25097

目录

前言

本指南介绍了 Oracle Solaris Studio 12.3 C、C++ 和 Fortran 编译器支持的 OpenMP 共享内存 API 的特定信息。

受支持的平台

此 Oracle Solaris Studio 发行版支持使用以下体系结构的平台：运行 Oracle Solaris 操作系统的 SPARC 系列处理器体系结构，以及运行 Oracle Solaris 或特定 Linux 系统的 x86 系列处理器体系结构。

本文档使用以下术语说明 x86 平台之间的区别：

- "x86" 泛指 64 位和 32 位的 x86 兼容产品系列。
- "x64" 指特定的 64 位 x86 兼容 CPU。
- “32 位 x86”指出了有关基于 x86 的系统的特定 32 位信息。

在 SPARC 和 x86 系统中，特定于 Linux 系统的信息仅指受支持的 Linux x86 平台，而特定于 Oracle Solaris 系统的信息仅指受支持的 Oracle Solaris 平台。

有关受支持的硬件平台和操作系统发行版的完整列表，请参见《Oracle Solaris Studio 12.3 发行说明》。

Oracle Solaris Studio 文档

可以查找 Oracle Solaris Studio 软件的完整文档，如下所述：

- 产品文档位于 Oracle Solaris Studio 文档 Web 站点，包括发行说明、参考手册、用户指南和教程。
- 代码分析器、性能分析器、线程分析器、dbxtool、DLight 和 IDE 的联机帮助可以在这些工具中通过 "Help"（帮助）菜单以及 F1 键和许多窗口和对话框上的 "Help"（帮助）按钮获取。
- 命令行工具的手册页介绍了工具的命令选项。

相关的第三方 Web 站点引用

本文档引用了第三方 URL，以用于提供其他相关信息。

注 – Oracle 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、产品或其他资料，Oracle 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，Oracle 概不负责，也不承担任何责任。

开发者资源

对于使用 Oracle Solaris Studio 的开发者，可访问 Oracle 技术网 Web 站点来查找以下资源：

- 有关编程技术和最佳做法的文章
- 软件最新发行版完整文档的链接
- 有关支持级别的信息
- 用户论坛。

获取 Oracle 支持

Oracle 客户可通过 My Oracle Support 获取电子支持。有关信息，请访问 `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info`，或访问 `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs`（如果您听力受损）。

印刷约定

下表介绍了本书中的印刷约定。

表 P–1 印刷约定

字体或符号	含义	示例
`AaBbCc123`	命令、文件和目录的名称；计算机屏幕输出	编辑 `.login` 文件。 使用 `ls -a` 列出所有文件。 `machine_name% you have mail.`
`AaBbCc123`	用户键入的内容，与计算机屏幕输出的显示不同	`machine_name%` **`su`** `Password:`

表 P–1 印刷约定 （续）

字体或符号	含义	示例
aabbcc123	要使用实名或值替换的命令行占位符	删除文件的命令为 `rm` *filename*。
AaBbCc123	保留未译的新词或术语以及要强调的词	这些称为 *Class* 选项。 **注意**：有些强调的项目在联机时以粗体显示。
新词术语强调	新词或术语以及要强调的词	**高速缓存**是存储在本地的副本。 请**勿**保存文件。
《书名》	书名	阅读《用户指南》的第 6 章。

命令中的 shell 提示符示例

下表显示了 Oracle Solaris OS 中包含的缺省 UNIX shell 系统提示符和超级用户提示符。请注意，在命令示例中显示的缺省系统提示符可能会有所不同，具体取决于 Oracle Solaris 发行版。

表 P–2 shell 提示符

shell	提示符
Bash shell、Korn shell 和 Bourne shell	`$`
Bash shell、Korn shell 和 Bourne shell 超级用户	`#`
C shell	`machine_name%`
C shell 超级用户	`machine_name#`

◆ ◆ ◆ 第 1 章

1 OpenMP API 简介

OpenMP 应用程序接口 (Application Program Interface, API) 是适用于共享内存多线程体系结构的可移植并行编程模型，它是由本公司与多家计算机供应商联合开发的。其规范由“OpenMP 体系结构审核委员会”创立并公布。

OpenMP API 是 Oracle Solaris 平台上所有 Oracle Solaris Studio 编译器的推荐并行编程模型。

1.1 哪里有 OpenMP 规范

本手册中的材料介绍了 OpenMP 3.1 API 的 Oracle Solaris Studio 实现所特有的问题，可以在官方 OpenMP Web 站点 `http://www.openmp.org` 中找到相关信息。

有关 OpenMP 的其他信息（包括教程和其他开发者资源），请访问 cOMPunity Web 站点 `http://www.compunity.org`。

有关 Oracle Solaris Studio 编译器发行版及其 OpenMP API 实现的最新信息，请访问 Oracle Solaris Studio 门户 `http://www.oracle.com/technetwork/server-storage/solarisstudio`。

1.2 特殊约定

在本文档中的表格和示例中，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。

术语**结构化块**是指无数据输入或输出的 Fortran 或 C/C++ 语句块。

方括号 [...] 中的构造是可选的。

在本手册中，"Fortran" 是指 Fortran 95 语言和 Oracle Solaris Studio 编译器 **f95**(1)。

在本手册中，术语“指令”和 "pragma" 互换使用。OpenMP API 指令是程序员插入的重要注释，用于指示编译器使用专用功能。注释不是 C、C++ 或 Fortran 主语言的一部分，根据编译器选项不同，可能会被忽略或实施。

◆ ◆ ◆ 第 2 章

2 编译并运行 OpenMP 程序

本章介绍编译器选项和运行时设置，这些选项和设置会影响使用 OpenMP API 的程序。

注 – 从 Oracle Solaris Studio 12.3 开始，用于 OpenMP 程序的缺省线程数为 2，而不是 1。可以通过以下方法更改该线程数：在运行程序之前设置 **OMP_NUM_THREADS** 环境变量，或调用 **omp_set_num_threads()** 例程，或者在 **PARALLEL** 指令中使用 **num_threads** 子句。

2.1 编译器选项

要使用 OpenMP 指令实现显式并行化，请使用 **cc**、**CC** 或 **f95** 选项标志 **-xopenmp** 编译程序。（**f95** 编译器将 **-xopenmp** 和 **-openmp** 作为同义词接受。）

-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。 如果优化级别低于 **-xO3**，则编译器不提升它。 如果将优化级别显式设置为低于 **-xO3** 的级别，如 **-xO2 -openmp=noopt**，则编译器会报告错误。 如果没有使用 **-openmp=noopt** 指定优化级别，则会识别 OpenMP pragma，并相应地并行化程序，但不执行优化。

-xopenmp=stubs	不再支持此选项。 OpenMP 桩模块库是为方便用户而提供的。 要编译调用 OpenMP 库例程的 OpenMP 程序但忽略其 OpenMP pragma，请在编译该程序时不要使用 **-xopenmp** 选项，并且将目标文件与 **libompstubs.a** 库链接。 例如，**% cc omp_ignore.c -lompstubs** 不支持同时与 **libompstubs.a** 和 OpenMP 运行时库 **libmtsk.so** 进行链接，因为这样可能会导致意外的行为。
-xopenmp=none	禁用对 OpenMP pragma 的识别，并且不更改优化级别。

附加说明：

- 为了在 Oracle Solaris 平台上获得最佳性能和功能，请确保正在运行的系统上已安装了最新的 OpenMP 运行时库 **libmtsk.so**。
- 如果未在命令行中指定 **–xopenmp**，则编译器会假定使用 **–xopenmp=none**（禁用对 OpenMP pragma 的识别）。
- 如果指定了 **–xopenmp** 但不带关键字子选项，则编译器会假定使用 **–xopenmp=parallel**。
- 指定 **-xopenmp=parallel** 或 **noopt** 时，会将 **_OPENMP** 预处理程序标记定义为 YYYYMM（具体地讲，对于 C/C++，将其定义为 **201107L**；对于 Fortran 95，将其定义为 **201107**）。
- 使用 **dbx** 调试 OpenMP 程序时，请使用 **-xopenmp=noopt -g** 进行编译。
- **-xopenmp** 的缺省优化级别在以后的发行版中可能会更改。通过显式指定适当的优化级别，可避免出现编译警告消息。
- 对于 Fortran 95，**-xopenmp**、**-xopenmp=parallel**、**-xopenmp=noopt** 会自动添加 **-stackvar**。请参见第 21 页中的 “2.4 堆栈和堆栈大小”。
- 在分步编译并链接 OpenMP 程序时，请在各个编译及链接步骤中包含 **-xopenmp**。
- 使用带有 **–xopenmp, –xopenmp=parallel** 或 **–xopenmp=noopt** 的 **–xvpara** 选项，以显示有关 OpenMP 编程潜在问题的编译器警告，以及与自动确定作用域（请参见第 6 章，自动确定变量的作用域）和作用域检查（请参见第 7 章，作用域检查）相关的消息。例如，如果编译器检测到以下列表中的情况，则将发出警告消息：
  - 循环是使用 OpenMP 指令并行化的，而这些指令中的不同循环迭代之间存在数据依赖性
  - OpenMP 数据共享属性子句存在问题。例如，声明在 OpenMP 并行区域中的访问可能导致数据争用的变量 **shared**，或者声明其在并行区域中的值在并行区域之后使用的变量 **private**。

2.2 OpenMP 环境变量

OpenMP 规范定义了若干用于控制 OpenMP 程序执行的环境变量。有关详细信息，请参阅 http://openmp.org 中的 OpenMP API 版本 3.1 规范。有关 Oracle Solaris Studio 编译器实现 OpenMP 环境变量的具体信息，另请参见第 26 页中的 “3.8 环境变量”。

Oracle Solaris Studio 编译器定义了未包括在 OpenMP 规范中的其他环境变量，第 15 页中的 “2.2.2 Oracle Solaris Studio 环境变量”中对这些变量进行了总结。

2.2.1 OpenMP 环境变量缺省值

本节介绍了 OpenMP 环境变量的缺省值。

OMP_SCHEDULE	如果未设置，则使用缺省值 **STATIC**。 示例： **setenv OMP_SCHEDULE 'GUIDED,4'**
OMP_NUM_THREADS	如果未设置，则使用缺省值 2。 示例： **setenv OMP_NUM_THREADS 16**
OMP_DYNAMIC	如果未设置，则使用缺省值 **TRUE**。 示例： **setenv OMP_DYNAMIC FALSE**
OMP_NESTED	如果未设置，则使用缺省值 **FALSE**。 示例： **setenv OMP_NESTED FALSE**
OMP_STACKSIZE	对于 32 位应用程序，缺省值为 4 MB；对于 64 位应用程序，缺省值为 8 MB。 示例： **setenv OMP_STACKSIZE 10M**
OMP_WAIT_POLICY	如果未设置，则使用缺省值 **PASSIVE**。
OMP_MAX_ACTIVE_LEVELS	如果未设置，则使用缺省值 4。
OMP_THREAD_LIMIT	如果未设置，则使用缺省值 1024。
OMP_PROC_BIND	如果未设置，则使用缺省值 **FALSE**。

2.2.2 Oracle Solaris Studio 环境变量

以下其他环境变量影响 OpenMP 程序的执行，但它们不是 OpenMP 规范的一部分。请注意，为以下环境变量指定的值为不区分大小写，可以采用大写或小写形式。

2.2.2.1 PARALLEL

为与传统程序兼容，设置 **PARALLEL** 环境变量的效果与设置 **OMP_NUM_THREADS** 的效果相同。然而，如果同时设置 **PARALLEL** 和 **OMP_NUM_THREADS**，则必须将它们设置为相同的值。

2.2.2.2 SUNW_MP_WARN

控制 OpenMP 运行时库发出的警告消息。如果将 **SUNW_MP_WARN** 设置为 **TRUE**，运行时库会向 **stderr** 发送警告消息。此外，运行时库还会输出所有环境变量的设置，以供参考。如果将该环境变量设置为 **FALSE**，运行时库将不发出任何警告消息或输出任何设置。缺省值为 **FALSE**。

示例：

```
setenv SUNW_MP_WARN TRUE
```

如果程序注册一个回调函数以接受警告消息，则运行时库也将发出警告消息。程序可通过调用以下函数来注册用户回调函数：

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

回调函数的地址将作为参数传递给 **sunw_mp_register_warn()**。如果成功注册了回调函数，该函数将返回 0，如果注册失败则返回 1。

如果程序已注册了回调函数，运行时库将调用该注册的函数，并将一个指针传递给包含错误消息的本地化字符串。从回调函数返回后，指向的内存将不再有效。

注 – OpenMP 运行时库能够检查很多常见的 OpenMP 违规行为，如错误的嵌套和死锁。但是运行时检查会增加程序执行的开销。测试或调试程序时，将 **SUNW_MP_WARN** 设置为 **TRUE**，以便可以显示来自 OpenMP 运行时库的警告消息。

2.2.2.3 SUNW_MP_THR_IDLE

控制 OpenMP 程序中空闲线程的状态，这些空闲线程正在某个屏障处等待或者正在等待要处理的新并行区域。可以将该值设置为下列某个值：**SPIN**、**SLEEP**、**SLEEP(***time***s)**、**SLEEP(***time***ms)**、**SLEEP(***time***mc)**，其中 *time* 是一个整数，指定时间量，**s**、**ms** 和 **mc** 指定时间单位（分别为秒、毫秒和微秒）。

SPIN 指定空闲线程在屏障处等待时或等待要处理的新并行区域时应旋转。不带时间参数的 **SLEEP** 指定空闲线程应立即休眠。带时间参数的 **SLEEP** 指定线程进入休眠状态前应旋转等待的时间量。

缺省情况下，空闲线程经过一段时间的旋转等待后将进入休眠状态。**SLEEP**、**SLEEP(0)**、**SLEEP(0s)**、**SLEEP(0ms)** 和 **SLEEP(0mc)** 都是等效的。

如果同时设置 **SUNW_MP_THR_IDLE** 和 **OMP_WAIT_POLICY**，则将忽略 **OMP_WAIT_POLICY**。

示例：

```
setenv SUNW_MP_THR_IDLE SPIN
setenv SUNW_MP_THR_IDLE SLEEP
setenv SUNW_MP_THR_IDLE SLEEP(2s)
setenv SUNW_MP_THR_IDLE SLEEP(20ms)
setenv SUNW_MP_THR_IDLE SLEEP(150mc)
```

2.2.2.4 SUNW_MP_PROCBIND

此环境变量可在 Oracle Solaris 和 Linux 系统上工作。**SUNW_MP_PROCBIND** 环境变量可用于将 OpenMP 程序的线程绑定到正在运行的系统上的虚拟处理器。虽然可以通过处理器绑定来增强性能，但是如果将多个线程绑定到同一虚拟处理器，则会导致性能下降。如果同时设置 **SUNW_MP_PROCBIND** 和 **OMP_PROC_BIND**，则必须将它们设置为相同的值。有关详细信息，请参见第 19 页中的 “2.3 处理器绑定”。

2.2.2.5 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、在并行区域中运行的非用户线程。该池不包含初始线程（主线程）或由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并行区域。如果未指定，则使用缺省值 1023。有关详细信息，请参见第 29 页中的 “4.2 控制嵌套并行操作”。

请注意，**SUNW_MP_MAX_POOL_THREADS** 指定用于整个程序的**非用户** OpenMP 线程的最大数量，而 **OMP_THREAD_LIMIT** 指定用于整个程序的**用户和非用户** OpenMP 线程的最大数量。如果同时设置 **SUNW_MP_MAX_POOL_THREADS** 和 **OMP_THREAD_LIMIT**，则它们的值必须一致，以便将 **OMP_THREAD_LIMIT** 设置为比 **SUNW_MP_MAX_POOL_THREADS** 的值大一。

2.2.2.6 SUNW_MP_MAX_NESTED_LEVELS

设置嵌套活动并行区域的最大数量。如果由包含多个线程的组执行并行区域，则该并行区域处于活动状态。如果未指定 **SUNW_MP_MAX_NESTED_LEVELS**，则使用缺省值 4。有关详细信息，请参见第 29 页中的 “4.2 控制嵌套并行操作”。

请注意，如果同时设置 **SUNW_MP_MAX_NESTED_LEVELS** 和 **OMP_MAX_ACTIVE_LEVELS**，则必须将它们设置为相同的值。

2.2.2.7 STACKSIZE

设置每个线程的堆栈大小。值以千字节为单位。在 32 位 SPARC V8 和 x86 平台上，缺省线程堆栈大小为 4 MB；在 64 位 SPARC V9 和 x86 平台上，缺省线程堆栈大小为 8 MB。

STACKSIZE 环境变量接受带有 **B, K, M**（字节、千字节、兆字节）或 **G**（千兆字节）后缀的数值。如果未指定后缀，则使用缺省单位千字节。

示例：

```
setenv STACKSIZE 8192  // sets the thread stack size to 8 Megabytes
setenv STACKSIZE 16M  // sets the thread stack size to 16 Megabytes
```

请注意，如果同时设置 **STACKSIZE** 和 **OMP_STACKSIZE**，则必须将它们设置为相同的值。如果值不相同，则会出现运行时错误。

2.2.2.8 SUNW_MP_GUIDED_WEIGHT

设置加权因子，该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，并且应用于程序中所有使用 **GUIDED** 调度的循环。如果未设置，则采用缺省值 2.0。

2.2.2.9 SUNW_MP_WAIT_POLICY

允许更精确地控制程序中等待工作（空闲）、在屏障处等待或等待任务完成的线程的行为。上述各种等待类型的行为有三种可能：旋转片刻、让出 CPU 片刻、休眠直至被唤醒。

语法为（使用 csh 显示）：

setenv SUNW_MP_WAIT_POLICY IDLE=*val* **:BARRIER=***val***:TASKWAIT=** *val*

IDLE=*val*、**BARRIER=** *val* 和 **TASKWAIT=** *val* 是可选关键字，用于指定所控制的等待类型。

上述每个关键字都有一个 *val* 设置来描述等待行为：**SPIN**、**YIELD** 或 **SLEEP**。

SPIN(*time***)** 指定线程在让出 CPU 之前应旋转多长时间。*time* 可以是秒、毫秒或微秒（分别用 **s**、**ms** 和 **mc** 表示）。如果不指定时间单位，则使用秒。如果 **SPIN** 不带时间参数，表示线程在等待时应持续旋转。

YIELD(*number***)** 指定线程在休眠之前应让出 CPU 的次数。每次让出 CPU 后，线程会在操作系统将其调度为运行时再次运行。如果 YIELD 不带 *number* 参数，表示线程在等待时应持续让出。

SLEEP 指定线程应立即转入休眠。

请注意，可以按任意顺序指定特定等待类型的 **SPIN**、**SLEEP** 和 **YIELD** 设置。这些设置之间用逗号分隔。**"SPIN(0),YIELD(0)"** 与 **SLEEP**（立即休眠）相同。在处理 **IDLE**、**BARRIER** 和 **TASKWAIT** 的设置时，采用左侧优先规则。

如果同时设置 **SUNW_MP_WAIT_POLICY** 和 **OMP_WAIT_POLICY**，则将忽略 **OMP_WAIT_POLICY**。

示例：

```
% setenv SUNW_MP_WAIT_POLICY “BARRIER=SPIN”
```

在屏障等待的线程将一直旋转，直到组中的所有线程都到达该屏障。

```
% setenv SUNW_MP_WAIT_POLICY “IDLE=SPIN(10ms),YIELD(5)”
```

等待工作（空闲）的线程旋转 10 毫秒，然后让出 CPU 5 次，再转入休眠。

```
% setenv SUNW_MP_WAIT_POLICY \
“IDLE=SPIN(10ms),YIELD(2):BARRIER=SLEEP:TASKWAIT=YIELD(10)”
```

等待工作（空闲）的线程旋转 10 毫秒，然后让出 CPU 2 次，再转入休眠；在屏障处等待的线程让出 CPU 10 次，再转入休眠。

2.3 处理器绑定

通过处理器绑定，程序员可指示操作系统在整个程序执行期间在同一处理器上运行该程序中的线程。

在将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，在该模式中，由并行区域或工作共享区域中的线程访问的数据将位于上次调用的并行区域或工作共享区域的本地缓存中。

从硬件的角度看，计算机系统是由一个或多个物理处理器组成的。从操作系统的角度看，其中每个物理处理器都映射到可运行程序中的线程的一个或多个虚拟处理器。如果 *n* 个虚拟处理器可用，则可同时调度运行 *n* 个线程。根据系统的不同，虚拟处理器可能是处理器、内核、硬件线程等。

例如，SPARC T3 物理处理器具有八个内核，每个内核可运行八个同时进行处理的线程。从 Oracle Solaris 系统的角度看，共有 64 个虚拟处理器，可以在其中调度要运行的线程。在 Oracle Solaris 平台上，可以使用 **psrinfo**(1M) 命令来确定虚拟处理器的数量。在 Linux 系统上，文本文件 `/proc/cpuinfo` 提供有关可用处理器的信息。

当操作系统将线程绑定到处理器时，实际上是将线程绑定到特定的**虚拟**处理器而不是**物理**处理器。

设置 **SUNW_MP_PROCBIND** 环境变量，可以绑定 OpenMP 程序中的线程。为 **SUNW_MP_PROCBIND** 指定的值可以是下列值之一：

- 字符串 "**TRUE**"、"**FALSE**"、"**COMPACT**" 或 "**SCATTER**"（或小写的 "**true**"、"**false**"、"**compact**" 或 "**scatter**"）。例如：`% setenv SUNW_MP_PROCBIND "TRUE"`

- 非负整数。
  例如，**% setenv SUNW_MP_PROCBIND "2"**
- 由一个或多个空格分隔的两个或更多非负整数的列表。
  例如，如果使用四个线程，则 **% setenv SUNW_MP_PROCBIND "2 2 4 6"** 将两个线程绑定到处理器 2，一个绑定到处理器 4，另一个绑定到处理器 6。
- 两个非负整数 *n1* 和 *n2*，二者之间由减号 ("-") 分隔。*n1* 必须小于或等于 *n2*。
  例如，**% setenv SUNW_MP_PROCBIND "0-6"**

请注意，上文提到的非负整数表示逻辑标识符 (ID)。逻辑 ID 可能不同于**虚拟**处理器 ID。下一节将介绍二者之间的差异。

2.3.1 虚拟和逻辑处理器 ID

系统中的每个虚拟处理器都有唯一的处理器 ID。Oracle Solaris **psrinfo**(1M) 命令显示有关系统中虚拟处理器的信息，包括其虚拟处理器 ID。**prtdiag**(1M) 命令显示系统配置和诊断信息。

可以使用 **psrinfo -pv** 列出系统中的所有物理处理器以及与每个物理处理器关联的虚拟处理器。

虚拟处理器 ID 可能是连续的，但也可能是不连续的。例如，在具有 8 个 UltraSPARC IV 处理器（16 个内核）的 Sun Fire 4810 上，虚拟处理器 ID 可能是：0、1、2、3、8、9、10、11、512、513、514、515、520、521、522、523。

另一方面，逻辑处理器 ID 是从 0 开始的连续整数。如果系统中可用的虚拟处理器数为 *n*，则其逻辑 ID 为 0、1、...、*n*-1（按 **psrinfo**(1M) 显示的顺序）。

有关 **SUNW_MP_PROCBIND** 接受的值的解释，请参见第 20 页中的 “2.3.2 解释为 **SUNW_MP_PROCBIND** 指定的值”。

2.3.2 解释为 SUNW_MP_PROCBIND 指定的值

如果为 **SUNW_MP_PROCBIND** 指定的目标为 **FALSE**，线程将不绑定到任何处理器。这是缺省设置。

如果为 **SUNW_MP_PROCBIND** 指定的值为 **TRUE**，线程将以循环（共享）方式绑定到虚拟处理器。绑定的起始处理器由运行时库以获得最佳性能为目标进行确定。

如果为 **SUNW_MP_PROCBIND** 指定的值为 **COMPACT**，则线程将被绑定到系统中最靠近的虚拟处理器。**COMPACT** 允许线程共享数据高速缓存，从而改进数据局域性。

如果为 **SUNW_MP_PROCBIND** 指定的值为 **SCATTER**，线程将被绑定到远离的虚拟处理器。这将允许每个线程具有更高的内存带宽。**SCATTER** 与 **COMPACT** 相反。

如果为 **SUNW_MP_PROCBIND** 指定的值是非负整数，则该整数表示线程应绑定到的虚拟处理器的起始逻辑 ID。线程会从具有指定逻辑 ID 的处理器开始，以循环方式绑定到虚拟处理器，在绑定到逻辑 ID 为 *n-1* 的处理器后，返回到逻辑 ID 为 0 的处理器。

如果为 **SUNW_MP_PROCBIND** 指定的值是包含两个或更多非负整数的列表，则线程将以循环方式绑定到具有指定逻辑 ID 的虚拟处理器。将不会使用其逻辑 ID 不是指定逻辑 ID 的处理器。

如果为 **SUNW_MP_PROCBIND** 指定的值是用减号 ("-") 分隔的两个非负整数，则线程将以循环方式绑定到如下范围的虚拟处理器：以第一个逻辑 ID 开头，并以第二个逻辑 ID 结尾。将不会使用其逻辑 ID 在此范围之外的处理器。

如果为 **SUNW_MP_PROCBIND** 指定的值不符合上述任何一种形式，或者给定的逻辑 ID 无效，则会发出一条错误消息，并终止程序的执行。

如果 OpenMP 程序中的线程数大于可用的虚拟处理器数，则一些虚拟处理器将绑定多个线程。这可能会对性能有负面影响。

2.3.3 与 OS 处理器集进行交互

在 Oracle Solaris 平台上使用 **psrset** 实用程序，或者在 Linux 平台上使用 **taskset** 命令，可以指定处理器集。**SUNW_MP_PROCBIND** 没有将处理器集考虑在内。如果您使用处理器集，则您应负责确保 **SUNW_MP_PROCBIND** 的设置与所用的处理器集一致。否则，在 Linux 系统上，**SUNW_MP_PROCBIND** 的设置将覆盖处理器集设置，而在 Oracle Solaris 系统上，将会发出错误消息。

2.4 堆栈和堆栈大小

正在执行的程序为执行该程序的初始（或主）线程维护一个主堆栈，并为每个从属线程维护不同的堆栈。堆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。如果线程堆栈的大小太小，则可能会出现堆栈溢出，从而导致无提示数据损坏或段故障。

使用 **f95 -stackvar** 选项编译 Fortran 程序会强制在堆栈中分配局部变量和数组，就好像它们是自动变量。显式并行化的程序暗指对 OpenMP 程序使用 **-stackvar**，因为该选项可提高优化器将循环中的调用并行化的能力。（有关 **-stackvar** 标志的讨论，请参见 Fortran 用户指南。）但是，如果为堆栈分配的内存不足，该使用会导致堆栈溢出。

使用 **limit** C-shell 命令或者 **ulimit** Bourne 或 Korn shell 命令可显示或设置初始线程（或主线程）的堆栈大小。一般而言，初始线程的缺省堆栈大小为 8 MB。

OpenMP 程序的每个从属线程均具有其自身的线程堆栈。此堆栈模拟初始（或主）线程堆栈，但对于线程是唯一的。线程的 **PRIVATE** 数组和变量（对于线程是局部的）在线

程堆栈中分配。在 32 位 SPARC V8 和 x86 平台上，缺省大小为 4 MB；在 64 位 SPARC V9 和 x86 平台上，缺省大小为 8 MB。从属线程堆栈的大小通过 **OMP_STACKSIZE** 环境变量来设置。

```
demo% setenv OMP_STACKSIZE 16384    <-Set thread stack size to 16 Mb (C shell)

demo$ OMP_STACKSIZE=16384    <-Set thread stack size to 16 Mb (Bourne/Korn shell)
demo$ export OMP_STACKSIZE
```

要检测堆栈溢出，请使用 **-xcheck=stkovf** 编译器选项编译 Fortran、C 或 C++ 程序，以强制在堆栈溢出时发生段故障，从而在发生任何数据损坏前停止程序。

2.5 检查和分析 OpenMP 程序

Oracle Solaris Studio 提供了几种工具来帮助调试和分析 OpenMP 程序。

- **dbx** 是一种交互式调试工具，可提供相应的功能以受控方式运行程序，并检查已停止程序的状态。有关更多信息，请参阅 **dbx**(1)。
- 线程分析器是用于检测多线程程序中数据争用和死锁的工具。有关详细信息，请参阅线程分析器手册以及 **tha**(1) 和 **libtha**(3) 手册页。
- 性能分析器用于分析 OpenMP 程序的性能。有关详细信息，请参考性能分析器手册或 **collect**(1) 和 **analyzer**(1) 手册页。

◆ ◆ ◆ 第 3 章

3 实现定义的行为

本章说明使用 Oracle Solaris Studio 编译器编译程序时特定 OpenMP 功能的行为方式。（请参见 OpenMP 3.1 API 规范的附录 E。）

3.1 任务调度点

非绑定 (untied) 任务区域中的任务调度点与绑定 (tied) 任务区域中的任务调度点出现在相同的点。因此，在非绑定 (untied) 任务区域内，OpenMP 规范会定义以下任务调度：

- 遇到的任务构造
- 遇到的任务等待 (taskwait) 构造
- 遇到的任务让出 (taskyield) 构造
- 遇到的屏障指令
- 隐式屏障区域
- 非绑定 (untied) 任务区域末尾

3.2 内存模型

当多个线程异步访问同一变量时，这些线程执行的内存访问不一定互为原子操作。一些依赖实现的因素和依赖应用程序的因素会对访问是否为原子操作产生影响。某些变量占用的内存空间可能比目标平台上最大的原子内存操作所占用的空间大。某些变量的存储方式可能是未对齐的或者其对齐方式是未知的，因此编译器或运行时系统可能需要使用多个 load/store 操作来访问变量。有时，使用多个 load/store 操作会让代码序列的运行速度更快。

3.3 内部控制变量

实现定义了以下内部控制变量：

- *nthreads-var*：控制为遇到的并行区域请求的线程数。*nthreads-var* 的初始值为 1。
- *dyn-var*：控制是否为遇到的并行区域启用线程数动态调整。*dyn-var* 的初始值为 `TRUE`（即启用动态调整）。
- *run-sched-var*：控制运行时调度子句针对循环区域使用的调度。*run-sched-var* 的初始值为 `static`（不指定块大小）。
- *def-sched-var*：控制实现定义的循环区域缺省调度。*def-sched-var* 的初始值为 `static`（不指定块大小）。
- *bind-var*：控制处理器的线程绑定。*bind-var* 的初始值为 `FALSE`。
- *stacksize-var*：控制 OpenMP 实现创建的线程的堆栈大小。*stacksize-var* 的初始值为 4 MB（对于 32 位应用程序）和 8 MB（对于 64 位应用程序）。
- *wait-policy-var*：控制等待线程的所需行为。*wait-policy-var* 的初始值为 `PASSIVE`。
- *thread-limit-var*：控制参与 OpenMP 程序的最大线程数量。*thread-limit-var* 的初始值为 1024。
- *max-active-levels-var*：控制嵌套活动并行区域的最大数量。*max-active-levels-var* 的初始值为 4。

3.4 线程的动态调整

实现提供了动态调整线程数量的功能。缺省情况下会启用动态调整。通过将 **`OMP_DYNAMIC`** 环境变量设置为 **`FALSE`**，或使用适当的参数调用 **`omp_set_dynamic()`** 例程，可以禁用动态调整。

当线程遇到并行构造时，此实现提供的线程数将根据 OpenMP 3.1 规范中的算法 2.1 来确定。在异常情况下，例如当缺少系统资源时，提供的线程数将少于算法 2.1 中所述的线程数。在这些情况下，如果将 **`SUNW_MP_WARN`** 设置为 **`TRUE`**，或者通过调用 **`sunw_mp_register_warn()`** 注册回调函数，则将发出警告消息。

3.5 循环指令

用于计算折叠 (collapsed) 循环的迭代计数的整数类型为 **`long`**。

将 *run-sched-var* 内部控制变量设置为 *auto* 时，**`schedule(runtime)`** 子句的效果为 `static`（不指定块大小）。

3.6 构造

3.6.1 SECTIONS

sections 构造中的结构化块在静态（不指定块大小）方式下分配给组中的线程，从而使每个线程获得的连续结构化块数量大致相等。

3.6.2 SINGLE

遇到 **single** 构造的第一个线程将会执行该构造。

3.6.3 ATOMIC

此实现通过使用一个名为 **critical** 的特殊构造封闭目标语句来替换所有 **atomic** 指令。此操作会在程序中的所有原子区域之间强制进行独占访问，无论这些区域是否更新相同或不同的存储位置。

3.7 例程

3.7.1 omp_set_num_threads()

如果 **omp_set_num_threads()** 的参数不是正整数，则忽略调用。如果将 **SUNW_MP_WARN** 设置为 **TRUE**，或者通过调用 **sunw_mp_register_warn()** 注册回调函数，则将发出警告消息。

3.7.2 omp_set_schedule()

Oracle Solaris Studio 特定的 **sunw_mp_sched_reserved** 调度的行为与 `static`（不指定块大小）相同。

3.7.3 omp_set_max_active_levels()

如果从活动并行区域中调用 **omp_set_max_active_levels()**，则忽略调用。如果将 **SUNW_MP_WARN** 设置为 `TRUE`，或者通过调用 **sunw_mp_register_warn()** 注册回调函数，则将发出警告消息。

如果 **omp_set_max_active_levels()** 的参数不是非负整数，则忽略调用。如果将 **SUNW_MP_WARN** 设置为 TRUE，或者通过调用 **sunw_mp_register_warn()** 注册回调函数，则将发出警告消息。

3.7.4 omp_get_max_active_levels()

可以从程序中的任何位置调用 **omp_get_max_active_levels()**。调用将返回 *max-active-levels-var* 内部控制变量的值。

3.8 环境变量

变量名称	实现
OMP_SCHEDULE	如果为 **OMP_SCHEDULE** 指定的调度类型不是有效类型（**static**、**dynamic**、**guided** 或 **auto**）之一，则将忽略该环境变量，并将使用缺省调度（**static**（不指定块大小））。如果将 **SUNW_MP_WARN** 设置为 TRUE，或者通过调用 **sunw_mp_register_warn()** 注册回调函数，则将发出警告消息。 如果为 **OMP_SCHEDULE** 环境变量指定的调度类型为 **static**、**dynamic** 或 **guided**，但是指定的块大小为负整数，则使用的块大小将如下：对于 **static**，将不使用块大小；对于 **dynamic** 和 **guided**，块大小为 1。如果将 **SUNW_MP_WARN** 设置为 TRUE，或者通过调用 **sunw_mp_register_warn()** 注册回调函数，则将发出警告消息。
OMP_NUM_THREADS	如果变量的值不是正整数，则将会忽略该环境变量，并且如果将 **SUNW_MP_WARN** 设置为 TRUE，或者通过调用 **sunw_mp_register_warn()** 注册回调函数，则将发出警告消息。 如果变量的值大于该实现可支持的线程数，则将执行以下操作： ■ 如果启用了线程数的**动态**调整，则线程数将会减少，并且如果将 **SUNW_MP_WARN** 设置为 TRUE，或者通过调用 **sunw_mp_register_warn()** 注册回调函数，则将发出警告消息。 ■ 如果禁用了线程数的动态调整，则将发出错误消息，并且程序将会停止。
OMP_PROC_BIND	如果为 **OMP_PROC_BIND** 指定的值既不是 **TRUE** 也不是 **FALSE**，则将发出错误消息，并且程序将会停止。
OMP_DYNAMIC	如果为 **OMP_DYNAMIC** 指定的值既不是 TRUE 也不是 FALSE，则将忽略该值，并将使用缺省值 TRUE。如果将 **SUNW_MP_WARN** 设置为 TRUE，或者通过调用 **sunw_mp_register_warn()** 注册回调函数，则将发出警告消息。
OMP_NESTED	如果为 **OMP_NESTED** 指定的值既不是 TRUE 也不是 FALSE，则将忽略该值，并将使用缺省值 FALSE。如果将 **SUNW_MP_WARN** 设置为 TRUE，或者通过调用 **sunw_mp_register_warn()** 注册回调函数，则将发出警告消息。

变量名称	实现
OMP_STACKSIZE	如果为 **OMP_STACKSIZE** 提供的值不符合指定格式，则将忽略该值，并将使用缺省值（对于 32 位应用程序为 4 MB，对于 64 位应用程序为 8 MB）。如果将 **SUNW_MP_WARN** 设置为 TRUE，或者通过调用 **sunw_mp_register_warn()** 注册回调函数，则将发出警告消息。
OMP_WAIT_POLICY	线程的 ACTIVE 行为是**旋转**。线程的 PASSIVE 行为是经过一段可能的旋转之后**休眠**。
OMP_MAX_ACTIVE_LEVELS	如果为 **OMP_MAX_ACTIVE_LEVELS** 指定的值不是非负整数，则将忽略该值，并将使用缺省值 (4)。如果将 **SUNW_MP_WARN** 设置为 TRUE，或者通过调用 **sunw_mp_register_warn()** 注册回调函数，则将发出警告消息。
OMP_THREAD_LIMIT	如果为 **OMP_THREAD_LIMIT** 指定的值不是正整数，则将忽略该值，并将使用缺省值 (1024)。如果将 **SUNW_MP_WARN** 设置为 TRUE，或者通过调用 **sunw_mp_register_warn()** 注册回调函数，则将发出警告消息。

3.9 Fortran 问题

以下问题仅适用于 Fortran。

3.9.1 THREADPRIVATE 指令

如果要在两个连续的活动并行区域之间保持的线程（初始线程除外）的线程专用 (thread-private) 对象中的数据值条件不能全部成立，则第二个区域中的可分配数组的分配状态可能为“当前未分配”。

3.9.2 SHARED 子句

如果将共享变量传递到非内在过程，可能导致该共享变量的值在过程引用之前被复制到临时存储中，并在过程引用之后又从临时存储中复制回到实际参数存储中。仅当 OpenMP 3.1 规范 **shared** 子句一节中的条件成立时，才会发生这种向临时存储复制数据以及从临时存储向外复制数据的情况，即：

- 实际参数为以下参数之一：
  - 共享变量
  - 共享变量的子对象
  - 与共享变量关联的对象
  - 与共享变量子对象关联的对象
- 实际参数也可以是以下参数之一：
  - 数组段
  - 带有向量下标的数组段

- 假定形状数组
- 指针数组

- 此实际参数的关联伪参数是显式形状数组或假定大小数组。

3.9.3 运行时库定义

此实现中同时提供了头文件 **omp_lib.h** 和模块文件 **omp_lib**。

在 Oracle Solaris 平台中，采用参数的 OpenMP 运行时库例程是通过通用接口扩展的，因此可以适应不同 Fortran **KIND** 类型的参数。

◆ ◆ ◆ 第 4 章

4 嵌套并行操作

本章讨论 OpenMP 嵌套并行操作的特性。

4.1 执行模型

OpenMP 采用 fork-join（派生-连接）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。如果池中不包含足够的空闲线程数，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。

4.2 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。

4.2.1 OMP_NESTED

可通过设置 **`OMP_NESTED`** 环境变量或调用 **`omp_set_nested()`** 来启用或禁用嵌套并行操作。

以下示例中的嵌套并行构造具有三个级别。

示例 4–1 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（经过排序的）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
```

以下示例运行相同程序，但是禁用了嵌套并行操作：

```
% 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
```

4.2.2 OMP_THREAD_LIMIT

OpenMP 运行时库维护一个线程池，该线程池可用作并行区域中的从属线程。可通过设置 **OMP_THREAD_LIMIT** 环境变量来控制池中的线程数。缺省情况下，池中的线程数最多为 1023。

线程池只包含运行时库创建的非用户线程。该池不包含初始线程或由用户程序显式创建的任何线程。

如果将 **OMP_THREAD_LIMIT** 设置为 1（或将 **SUNW_MP_MAX_POOL_THREADS** 设置为 0），则线程池将为空，并且将由一个线程执行所有并行区域。

以下示例表明，如果池中的线程不足，并行区域将获得较少的从属线程。代码与示例 4–1 中的相同。使所有并行区域同时处于活动状态所需的线程数为 8 个。所以，池至少需要包含 7 个线程。如果将 **OMP_THREAD_LIMIT** 设置为 6（或将 **SUNW_MP_MAX_POOL_THREADS** 设置为 5），则池最多包含 5 个从属线程。这意味着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。以下示例显示一个可能的结果：

```
% setenv OMP_NESTED TRUE
% OMP_THREAD_LIMIT 6
% 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
```

4.2.3 OMP_MAX_ACTIVE_LEVELS

环境变量 **OMP_MAX_ACTIVE_LEVELS** 可控制嵌套活动并行区域的最大数量。如果由包含多个线程的组执行并行区域，则该并行区域处于活动状态。嵌套活动并行区域的缺省最大数量为 4。

请注意，设置该环境变量不足以启用嵌套并行操作。该环境变量仅控制嵌套活动并行区域的最大数量，并不启用嵌套并行操作。要启用嵌套并行操作，必须将 **OMP_NESTED** 设置为 **TRUE**，或者必须使用求值结果为 *true* 的参数调用 **omp_set_nested()**。

以下代码将创建 4 级嵌套并行区域。如果将 **OMP_MAX_ACTIVE_LEVELS** 设置为 2，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);
}
```

以下示例说明了使用最大嵌套级别 4 编译并运行此程序所产生的可能结果。（实际结果取决于 OS 调度线程的方式。）

```
% setenv OMP_MAX_ACTIVE_LEVELS 4
% a.out |sort
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
```

以下示例说明了将嵌套级别设置为 2 时运行的可能结果：

```
% setenv OMP_MAX_ACTIVE_LEVELS 2
% a.out |sort
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
```

此外，这些示例只显示了一些**可能**的结果。实际结果取决于 OS 调度线程的方式。

4.3 在嵌套并行区域中使用 OpenMP 库例程

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处于同一嵌套级别或内部嵌套级别的后续并行区域。它们不影响其他线程遇到的并行区域。

get 调用将返回由调用线程设置的值。当某个线程成为执行并行区域的组的主线程后，所有其他的组成员会继承该主线程的值。当主线程退出嵌套并行区域，并继续执行封闭并行区域时，该线程的值会恢复为刚执行嵌套并行区域之前封闭并行区域中的值。

示例 4–2 在并行区域中调用 OpenMP 例程

```
#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);       /* line A */
        else
            omp_set_num_threads(6);       /* line B */

        /* The following statement will print out
         *
         * 0: 2 4
         * 1: 2 6
         *
         * omp_get_num_threads() returns the number
         * of the threads in the team, so it is
         * the same for the two threads in the team.
         */
        printf("%d: %d %d\n", omp_get_thread_num(),
               omp_get_num_threads(),
               omp_get_max_threads());
```

示例 4–2 在并行区域中调用 OpenMP 例程 （续）

```
        /* Two inner parallel regions will be created
         * one with a team of 4 threads, and the other
         * with a team of 6 threads.
         */
        #pragma omp parallel
        {
            #pragma omp master
            {
                /* The following statement will print out
                 *
                 * Inner: 4
                 * Inner: 6
                 */
                printf("Inner: %d\n", omp_get_num_threads());
            }
            omp_set_num_threads(7);      /* line C */
        }

        /* Again two inner parallel regions will be created,
         * one with a team of 4 threads, and the other
         * with a team of 6 threads.
         *
         * The omp_set_num_threads(7) call at line C
         * has no effect here, since it affects only
         * parallel regions at the same or inner nesting
         * level as line C.
         */

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

以下示例说明了编译并运行此程序所产生的可能结果：

```
% 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.
```

4.4 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。此外，还假定您的系统有四个 CPU，您要使用全部四个 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。适当设置 **OMP_THREAD_LIMIT** 和 **OMP_MAX_ACTIVE_LEVELS** 可限制使用中的线程数，并防止过多占用系统资源而失控。

- 创建嵌套并行区域会增加开销。如果外部级别有足够的并行操作并且负载平衡，在计算的外部级别使用所有线程要比在内部级别创建嵌套并行区域更有效。

  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四，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。通常，将所有四个线程用于外部级别在性能上要优于将两个线程用于外部并行区域而将其他两个线程用作内部并行区域的从属线程。

◆ ◆ ◆ 第 5 章

5 任务处理

本章介绍 OpenMP 任务处理模型。

5.1 任务处理模型

任务处理功能有助于应用程序的并行化，其中任务单元是动态生成的，就像在递归结构或 *while* 循环中一样。

在 OpenMP 中，使用 **task** 指令指定**显式**任务。**task** 指令定义了与任务及其数据环境关联的代码。任务构造可以放置在程序中的任何位置：只要线程遇到任务构造，就会生成新任务。

当线程遇到任务构造时，可能会选择立即执行任务或延迟执行任务直到稍后某个时间再执行。如果延迟执行任务，则任务会被放置在与当前并行区域关联的概念任务池中。当前组中的线程会将任务从该池中取出，并执行这些任务，直到该池为空。执行任务的线程可能与最初遇到该任务的线程不同。

与任务构造关联的代码将只被执行一次。如果代码从始至终都由相同的线程执行，则任务为**绑定** *(tied)* 任务。如果代码可由多个线程执行，使得不同的线程执行代码的不同部分，则任务为**非绑定** *(untied)* 任务。缺省情况下，任务为**绑定** *(tied)* 任务，可以通过将 **untied** 子句与 **task** 指令一起使用来将任务指定为**非绑定** *(untied)* 任务。

为了执行不同的任务，允许线程在任务调度点暂停执行任务区域。如果暂停的任务为绑定 (tied) 任务，则同一线程稍后会恢复执行暂停的任务。如果暂停的任务为非绑定 (untied) 任务，则当前组中的任何线程都可能会恢复执行该任务。

OpenMP 规范为**绑定** *(tied)* 任务定义了以下任务调度点：

- 遇到任务构造的点
- 遇到任务等待 (taskwait) 构造的点
- 遇到任务让出 (taskyield) 构造的点
- 遇到隐式或显式屏障的点

- 任务的完成点

在 Oracle Solaris Studio 编译器中实现时，这些调度点也是**非绑定** *(untied)* 任务的任务调度点。

除了使用 task 指令指定的显式任务外，OpenMP 规范还介绍了**隐式**任务的概念。隐式任务是由隐式并行区域生成的任务，或是在执行期间遇到并行构造时生成的任务。每个隐式任务的代码都是 **parallel** 构造内的代码。每个隐式任务会分配给组中的不同线程，且隐式任务为绑定 (tied) 任务，即隐式任务从始至终总是由最初分配给的线程执行。

对于在遇到 **parallel** 构造时生成的所有隐式任务，都要保证在主线程退出并行区域末尾的隐式屏障时完成。另一方面，对于在并行区域中生成的所有显式任务，都要保证在从并行区域中的下一个隐式或显式屏障退出时完成。

OpenMP 3.1 规范定义了各种类型的任务，程序员可能会选择这些任务用来减少任务处理开销。

未延迟 *(undeferred)* 任务是指相对于其生成任务区域，不会延迟执行的任务。即，其生成任务区域会暂停，直至未延迟 (undeferred) 任务执行完成。例如，**if** 子句表达式求值结果为 **false** 的任务即是一个未延迟 (undeferred) 任务。在这种情况下，将会生成未延迟 (undeferred) 任务，并且遇到该任务的线程必须暂停当前任务区域，直到完成具有 **if** 子句的任务后才能恢复执行该任务区域。

包括 *(included)* 任务是指其执行按顺序包括在生成任务区域的任务。即，该任务不会被延迟，由遇到该任务的线程立即执行。例如，属于**最终** *(final)* 任务（如下所述）后代的任务即是一个包括 (included) 任务。

包括 (included) 任务和未延迟 (undeferred) 任务之间的差异很细微。对于未延迟 (undeferred) 任务，生成任务区域会暂停，直至未延迟 (undeferred) 任务执行完成，但是未延迟 (undeferred) 任务可能不会在被遇到时立即执行。未延迟 (undeferred) 任务可能会被放置在概念池中，由遇到该任务的线程或某个其他线程稍后执行；同时，生成任务将被暂停。未延迟 (undeferred) 任务执行完成后，生成任务才能恢复。

对于包括 (included) 任务，一旦被遇到，就立即由遇到该任务的线程执行该包括 (included) 任务。该任务不会被放置在池中以在稍后执行。生成任务会暂停，直到包括 (included) 任务执行完成。包括 (included) 任务执行完成后，生成任务才能恢复。

合并 *(merged)* 任务是指其数据环境与其生成任务区域的数据环境相同的任务。当 **mergeable** 子句存在于 **task** 构造中，且生成的任务为未延迟 (undeferred) 任务或包括 (included) 任务时，则实现可能会选择生成合并 (merged) 任务。如果生成合并 (merged) 任务，则相应行为就好像根本没有 task 指令一样。

最终 *(final)* 任务是指强制其所有子任务成为最终 (final) 和包括 (included) 任务的任务。当 **final** 子句存在于 **task** 构造中，且 **final** 子句表达式求值结果为 **TRUE** 时，生成的任务将为**最终** *(final)* 任务。该最终 (final) 任务的所有后代将既是**最终** *(final)* 任务，又是**包括** *(included)* 任务。

5.2 数据环境

task 指令采用以下数据属性子句，这些子句可定义任务的数据环境：

- **default** (**private** | **firstprivate** | **shared** | **none**)
- **private** (*list*)
- **firstprivate** (*list*)
- **shared** (*list*)

在任务内对 **shared** 子句中列出的变量的所有引用是指在 **task** 指令之前一看便知的同名变量。

对于每个 **private** 和 **firstprivate** 变量，都会创建一个新存储，并且对 **task** 构造词法范围内的原始变量的所有引用都会被对新存储的引用所替换。遇到任务时，将会使用原始变量的值初始化 **firstprivate** 变量。

OpenMP 版本 3.0 规范（第 2.9.1 节）介绍了如何确定在并行、任务和工作共享区域中所引用变量的数据共享属性。

构造中引用的变量的数据共享属性可以是以下属性之一：**预先确定**、**显示确定**或**隐式确定**。具有显式确定数据共享属性的变量是那些在给定构造中引用，并在构造的数据共享属性子句中列出的变量。具有隐式确定数据共享属性的变量是那些在给定构造中引用、不具有预先确定数据共享属性，并且不在构造的数据共享属性子句中列出的变量。

有关如何隐式确定变量的数据共享属性的规则可能并不总是很直观。为避免意外，请确保使用数据共享属性子句显式确定任务构造中引用的所有变量的作用域，而不是依赖 OpenMP 隐式作用域规则。

5.3 任务处理示例

以下 C/C++ 程序说明为什么 OpenMP **task** 指令和 **taskwait** 指令可用于递归计算斐波纳契数。

在该示例中，**parallel** 指令指示一个将由四个线程执行的并行区域。在并行构造中，**single** 指令用于指示只有其中一个线程将执行调用 `fib(n)` 的 **print** 语句。

对 `fib(n)` 的调用会生成两个任务（由 **task** 指令指示）。其中一个任务计算 `fib(n-1)`，另一个任务计算 `fib(n-2)`，将返回值加在一起即可产生由 `fib(n)` 返回的值。对 `fib(n-1)` 和 `fib(n-2)` 的每个调用反过来又会生成两个任务。将会以递归方式生成任务，直到传递到 `fib()` 的参数小于 2。

请注意每个 **task** 指令上的 **final** 子句。如果 **final** 子句表达式 (**n <= THRESHOLD**) 求值结果为 true，则生成的任务将为 **final** 任务。执行最终 (final) 任务期间遇到的所有 **task** 构造将生成包括 (included)（和最终 (final)）任务。因此，当使用参数 *n = 9, 8, ..., 2* 调用 `fib` 时，将生成包括 (included) 任务。这些包括 (included) 任务将由遇到这些任务的线程立即执行，从而减少在概念池中放置任务的开销。

taskwait 指令可确保在调用 fib() 的过程中生成的两个任务（即计算 i 和 j 的任务）在对 fib() 的调用返回之前已完成。

请注意，虽然只有一个线程执行 **single** 指令（因而也只有一个线程对 fib(n) 进行调用），但是所有四个线程都将参与执行生成的任务。

以下示例是使用 Oracle Solaris Studio C++ 编译器编译的。

示例 5–1 任务处理示例：计算斐波纳契数

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

#define THRESHOLD 5

int fib(int n)
{
  int i, j;

  if (n<2)
    return n;

  #pragma omp task shared(i) firstprivate(n) final(n <= THRESHOLD)
  i=fib(n-1);

  #pragma omp task shared(j) firstprivate(n) final(n <= THRESHOLD)
  j=fib(n-2);

  #pragma omp taskwait
  return i+j;
}


int main()
{
  int n = 30;
  omp_set_dynamic(0);
  omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  {
     #pragma omp single
     printf ("fib(%d) = %d\n", n, fib(n));
  }
}

% CC -xopenmp -xO3 task_example.cc

% a.out
fib(30) = 832040
```

5.4 编程注意事项

任务处理功能使 OpenMP 程序的复杂性有所增加。程序员需要特别注意带有任务的程序的工作原理。本节讨论一些要考虑的编程问题。

5.4.1 THREADPRIVATE 和线程特定的信息

当线程遇到任务调度点时，实现可能会选择暂停当前任务并安排线程处理另一个任务。该行为意味着 **threadprivate** 变量的值或线程特定的其他信息（如线程数）可能会在任务调度点发生变化。

如果暂停的任务为**绑定** *(tied)* 任务，则恢复执行该任务的线程与暂停该任务的线程将是同一线程。因此，恢复该任务后，线程数将保持相同。但是，**threadprivate** 变量的值可能会更改，原因是可能会安排线程处理另一个任务，这样会在恢复暂停的任务之前修改 **threadprivate** 变量。

如果暂停的任务为**非绑定** *(untied)* 任务，则恢复执行该任务的线程可能与暂停该任务的线程不同。因此，线程数和 **threadprivate** 变量的值在任务调度点之前和之后都可能不同。

5.4.2 锁

OpenMP 指定，锁不再归线程所有，而是归任务所有。一旦获取了锁，当前任务就会拥有该锁，同一任务必须先释放锁才能完成任务。

另一方面，**critical** 构造仍保留采用**基于线程的互斥机制**。

使用锁时，需要格外小心锁所有权的变化。以下示例（在 OpenMP 3.1 规范的附录 A 中出现）符合 OpenMP 2.5，因为在并行区域中释放锁 `lck` 的线程与在该程序顺序部分中获取锁的线程为同一线程。并行区域的主线程与初始线程相同。但是，该示例不符合 OpenMP 3.1，因为释放锁 `lck` 的任务区域与获取锁的任务区域不同。

示例 5–2 使用锁的示例：不符合 OpenMP 3.0

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

int main()
{
  int x;
  omp_lock_t lck;

  omp_init_lock (&lck);
  omp_set_lock (&lck);
  x = 0;
```

示例 5–2 使用锁的示例：不符合 OpenMP 3.0 （续）

```
  #pragma omp parallel shared (x)
  {
    #pragma omp master
    {
      x = x + 1;
      omp_unset_lock (&lck);
    }
  }
  omp_destroy_lock (&lck);
}
```

5.4.3 对堆栈数据的引用

任务可能会引用任务构造所在的例程的堆栈数据。由于任务的执行可能会延迟，直至下一个隐式或显式屏障，所以有可能出现这样的情况：给定的任务将在任务所在的例程的堆栈已经弹出，且堆栈数据被覆盖（从而销毁由任务列为共享的堆栈数据）之后执行。

程序员应负责插入所需的同步，以确保任务引用变量时这些变量仍在堆栈中，如以下两个示例所示。

在第一个示例中，在 **task** 构造中将 `i` 指定为 **shared**，任务会访问在 `work()` 的堆栈中分配的 `i` 的副本。

任务的执行可能会延迟，使得任务将在 `work()` 例程已返回后，在 `main()` 中的并行区域末尾的隐式屏障处执行。因此当任务引用 `i` 时，会访问当时碰巧在堆栈中的某个不确定的值。

为了得到正确的结果，程序员需要确保 `work()` 不会在任务完成前退出。这可以通过在 **task** 构造之后插入 **taskwait** 指令来实现。或者，可以在 **task** 构造中将 `i` 指定为 **firstprivate** 而不是 **shared**。

示例 5–3 堆栈数据：第一个示例－不正确的版本

```
#include <stdio.h>
#include <omp.h>
void work()
 {
   int i;

   i = 10;
   #pragma omp task shared(i)
   {
     #pragma omp critical
     printf("In Task, i = %d\n",i);
   }
 }

int main(int argc, char** argv)
```

示例 5–3 堆栈数据：第一个示例－不正确的版本 （续）

```
 {
    omp_set_num_threads(8);
    omp_set_dynamic(0);

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

示例 5–4 堆栈数据：第一个示例－更正的版本

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

void work()
 {
   int i;

   i = 10;
   #pragma omp task shared(i)
   {
     #pragma omp critical
     printf("In Task, i = %d\n",i);
   }

   /* Use TASKWAIT for synchronization. */
   #pragma omp taskwait
 }

int main(int argc, char** argv)
 {
    omp_set_num_threads(8);
    omp_set_dynamic(0);

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

在第二个示例中，**task** 构造中的 j 引用 **sections** 构造中的 j。因此，任务会访问 **sections** 构造中 j 的 **firstprivate** 副本，该副本（在某些实现中，包括 Oracle Solaris Studio 编译器）是 **sections** 构造的概要例程的堆栈中的局部变量。

任务的执行可能会延迟，使得任务将在 **sections** 构造的概要例程退出后，在 **sections** 区域末尾的隐式屏障处执行。因此当任务引用 j 时，会访问堆栈中的某个不确定的值。

为了得到正确的结果，程序员需要确保任务在 **sections** 区域达到其隐式屏障前执行，这可以通过在 **task** 构造之后插入 **taskwait** 指令来实现。或者，可以在 **task** 构造中将 j 指定为 **firstprivate** 而不是 **shared**。

示例 5–5 第二个示例－不正确的版本

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

int main(int argc, char** argv)
 {
    omp_set_num_threads(2);
    omp_set_dynamic(0);
    int j=100;

    #pragma omp parallel shared(j)
    {
       #pragma omp sections firstprivate(j)
       {
          #pragma omp section
          {
             #pragma omp task shared(j)
             {
               #pragma omp critical
               printf("In Task, j = %d\n",j);
             }
          }
       }
    }

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

示例 5–6 第二个示例－更正的版本

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

int main(int argc, char** argv)
 {
    omp_set_num_threads(2);
    omp_set_dynamic(0);
    int j=100;

    #pragma omp parallel shared(j)
    {
       #pragma omp sections firstprivate(j)
       {
          #pragma omp section
          {
             #pragma omp task shared(j)
             {
               #pragma omp critical
               printf("In Task, j = %d\n",j);
             }

             /* Use TASKWAIT for synchronization. */
             #pragma omp taskwait
          }
       }
    }

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

示例 5–6 第二个示例－更正的版本 （续）

```
}
```

◆ ◆ ◆ 第 6 章

6 自动确定变量的作用域

声明 OpenMP 构造中所引用变量的数据共享属性的过程称为**确定作用域**。有关每个数据共享属性的说明，请参见 OpenMP 规范的“数据共享属性子句”一节（第 2 章）。

在 OpenMP 程序中，会为 OpenMP 构造中引用的每个变量确定作用域。通常，可通过两种方法之一来确定构造中所引用变量的作用域。程序员可使用**数据共享属性子句**来显式声明变量的作用域，或者，编译器中的 OpenMP API 实现可根据 OpenMP 规范的“数据环境：数据作用域规则”一节（第 2 章）自动对预先确定或隐式确定的作用域应用规则。

大多数用户会发现，确定作用域是使用 OpenMP 范例中最难的部分。显式确定变量作用域的过程非常乏味，而且容易出错，尤其是对于大型和复杂的程序而言。而且，在 OpenMP 3.0 规范中针对隐式确定和预先确定变量作用域所指定的规则可能会产生某些意外结果。OpenMP 规范 3.0 中引入的 **task** 指令增加了确定作用域的复杂性和难度。

自动确定作用域功能（称为**自动确定作用域**）受 Oracle Solaris Studio 编译器支持，使得程序员无需显式确定变量的作用域，因此是一个很有用的工具。通过自动确定作用域功能，编译器可在非常简单的用户模型中使用一些智能规则来确定变量的作用域。

早期编译器发行版将自动确定作用域功能仅限于 **parallel** 构造中的变量。当前的 Oracle Solaris Studio 编译器对自动确定作用域功能的范围进行了扩展，包含了 **task** 构造中引用的变量。

6.1 自动确定作用域数据范围子句

可通过在 **__auto** 数据作用域子句中指定要自动确定作用域的变量或使用 **default(__auto)** 子句，来调用自动确定作用域功能。这两种方法都是 Oracle Solaris Studio 编译器提供的 OpenMP 规范的扩展。

6.1.1 __auto 子句

语法：**__auto(***list-of-variables***)**

对于 Fortran，也接受 **__AUTO(***list-of-variables***)**。

并行或任务构造中的 **__auto** 子句可指示编译器自动确定构造中已命名变量的作用域。（请注意 **auto** 前面的两个下划线）。

__auto 子句可以出现在 **PARALLEL**、**PARALLEL DO/for**、**PARALLEL SECTIONS**、Fortran 95 **PARALLEL WORKSHARE** 或 **TASK** 指令中。

如果在 **__auto** 子句中指定了变量，将不能在任何其他数据共享属性子句中指定该变量。

6.1.2 default(__auto) 子句

语法：**default(__auto)**

对于 Fortran，也接受 **DEFAULT(__AUTO)**。

并行或任务构造中的 **default(__auto)** 子句可指示编译器自动确定构造中引用的所有未在任何数据作用域子句中显式确定作用域的变量的作用域。

default(__auto) 子句可以出现在 **PARALLEL**、**PARALLEL DO/for**、**PARALLEL SECTIONS**、Fortran 95 **PARALLEL WORKSHARE** 或 **TASK** 指令中。

6.2 并行构造的作用域规则

在自动确定作用域的情况下，编译器应用本节中介绍的规则来确定并行构造中变量的作用域。

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

6.2.1 标量变量的作用域规则

在自动确定并行构造中引用的且没有预先确定或隐式确定作用域的标量变量的作用域时，编译器会按给定顺序根据以下规则 **PS1-PS3** 来检查变量的使用。

- **PS1**：对于组中执行并行区域的线程而言，如果在该区域中使用变量不会导致**数据争用**情形，则将变量的作用域确定为 **SHARED**。

- **PS2**：如果在每个执行并行区域的线程中，在读取变量之前始终先由同一线程写入，则将变量的作用域确定为 **PRIVATE**。如果可以将变量的作用域确定为 **PRIVATE**，并且该变量在写入（在并行区域之后写入）之前被读取，而构造为 **PARALLEL DO** 或 **PARALLEL SECTIONS**，则将其作用域确定为 **LASTPRIVATE**。
- **PS3**：如果在编译器可以识别的归约操作中使用变量，则将该变量的作用域确定为具有该特定操作类型的 **REDUCTION**。

6.2.2 数组的作用域规则

- **PA1**：对于组中执行并行区域的线程而言，如果在该区域中使用数组不会导致数据争用情形，则将数组的作用域确定为 **SHARED**。

6.3 任务构造的作用域规则

在自动确定作用域的情况下，编译器应用本节中介绍的规则来确定 **task** 构造中变量的作用域。

这些规则不适用于由 OpenMP 规范隐式确定作用域的变量，如 **PARALLEL DO**/**for** 循环的循环索引变量。

6.3.1 标量变量的作用域规则

在自动确定任务构造中引用的且没有预先确定或隐式确定作用域的标量变量的作用域时，编译器会按给定顺序根据以下规则 **TS1**-**TS5** 来检查变量的使用。

- **TS1**：如果变量的使用在 **task** 构造中是只读的，并且在包括该任务构造的并行构造中也是只读的，则自动将变量的作用域确定为 **FIRSTPRIVATE**。
- **TS2**：如果变量的使用不会导致数据争用，并且可在执行任务时访问该变量，则自动将变量的作用域确定为 **SHARED**。
- **TS3**：如果变量的使用不会导致数据争用，并且在任务构造中是只读的，但在执行任务时不可访问该变量，则自动将变量的作用域确定为 **FIRSTPRIVATE**。
- **TS4**：如果变量的使用会导致数据争用，并且在每个执行任务区域的线程中，在读取变量之前始终先由同一线程写入，而且向任务中的变量指定的值不在任务区域之外使用，则自动将变量的作用域确定为 **PRIVATE**。
- **TS5**：如果变量的使用会导致数据争用，该变量在任务区域中不为只读，并且在任务区域中执行某些读取操作可能会获取在任务之外定义的值，而且向任务中的变量指定的值不在任务区域之外使用，则自动将变量的作用域确定为 **FIRSTPRIVATE**。

6.3.2 数组的作用域规则

自动确定任务的作用域时不会处理数组。

6.4 关于自动确定作用域的通用注释

注意，在将来的版本中，任务自动确定作用域规则和自动确定作用域的结果可能会有所更改。而且，隐式确定作用域规则和自动确定作用域规则的应用顺序在将来的发行版中也会发生更改。

程序员使用 **_auto(***list-of-variables***)** 子句或 **default(_auto)** 子句显式请求自动确定作用域。为 **parallel** 构造指定 **default(_auto)** 或 **_auto(***list-of-variables***)** 子句，并不意味着将同一子句应用于在语法上或动态包含在 **parallel** 构造中的 **task** 构造。

在对没有预先确定隐式作用域的变量自动确定作用域时，编译器会按给定顺序根据上述规则来检查变量的使用。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。如果没有匹配的规则或自动确定作用域无法处理变量（如下节所述，由于存在某些限制），则编译器会将变量的作用域确定为 **SHARED**，并将 **parallel** 或 **task** 构造视为如同指定了 **IF (.FALSE.)** 或 **if(0)** 子句一样。

通常，由于以下两个原因之一，自动确定作用域功能会失败。一个原因是使用的变量不匹配任何规则。另一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、复杂的数组下标、内存别名和用户实现的同步都是常见原因。

6.5 限制

- 要启用自动确定作用域功能，必须使用 **-xopenmp** 在优化级别 **-xO3** 或更高级别上编译程序。如果仅使用 **-xopenmp=noopt** 编译程序，将不会启用自动确定作用域功能。
- C 和 C++ 中的并行和任务自动作用域只能处理基本数据类型：整型、浮点和指针。
- 任务自动确定作用域功能不能处理数组。
- C 和 C++ 中的任务自动确定作用域不能处理全局变量。
- 任务自动确定作用域不能处理非绑定任务。
- 任务自动确定作用域不能处理在语法上包含在其他任务中的任务。例如：

```
#pragma omp task /* task1 */
{
  ...
  #pragma omp task /* task 2 */
  {
    ...
  }
 ...
}
```

在示例中，由于 task2 在语法上嵌套在 task1 中，因此编译器不会尝试对其启用自动确定作用域功能。编译器会将 task2 中引用的所有变量的作用域确定为 **SHARED**，并将 task2 视为如同指定了 **IF(.FALSE.)** 或 **if(0)** 子句一样。

- 只识别 OpenMP 指令，并且只能在分析中使用。无法识别对 OpenMP 运行时例程的调用。例如，如果程序使用 **omp_set_lock()** 和 **omp_unset_lock()** 来实现临界段，编译器将无法检测是否存在临界段。如果可能，请使用 **CRITICAL** 和 **END CRITICAL** 指令。
- 在数据争用分析中，只能识别和使用通过 OpenMP 同步指令（如 **BARRIER** 和 **MASTER**）指定的同步。不识别用户实现的同步，如忙等待。

6.6 检查自动确定作用域的结果

使用**编译器注释**可检查自动确定作用域结果，并确定是否因自动确定作用域功能失败而对并行区域进行了序列化。

使用 **-g** 调试选项进行编译时，编译器将生成行内注释，可以使用 **er_src** 命令查看这个生成的注释，如以下示例所示。**er_src** 命令作为 Oracle Solaris Studio 软件的一部分提供。有关更多信息，请参见 **er_src**(1) 手册页或 Oracle Solaris Studio 性能分析器手册。

使用 **-xvpara** 选项进行编译是一个良好的开端。使用 **–xvpara** 进行编译可大体确定针对特定构造的自动确定作用域是否成功。

示例 6–1 使用 -vpara 自动确定作用域

```
%cat source1.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 -vpara -c -g source1.f
"source1.f", line 2: Autoscoping for OpenMP construct succeeded.
Check er_src for details
```

如果针对特定构造的自动确定作用域失败，将发出以下示例中所示的警告消息（使用 **-xvpara**）。

示例 6–2 使用 -vpara 自动确定作用域失败

```
%cat source2.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
```

示例 6–2 使用 -vpara 自动确定作用域失败 （续）

```
%f95 -xopenmp -xO3 -vpara -c -g source2.f
"source2.f", line 2: Warning: Autoscoping for OpenMP construct failed.
 Check er-src for details. Parallel region will be executed by
 a single thread.
```

er_src 所显示的编译器注释中显示了更详细的信息：

示例 6–3 使用 er_src

```
% er_src source2.o
Source file: source2.f
Object file: source2.o
Load Object: source2.o

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

   Source OpenMP region below has tag R1
   Variables autoscoped as SHARED in R1: y
   Variables autoscoped as PRIVATE in R1: t, i
   Variables treated as shared because they cannot be autoscoped in R1: x
   R1 will be executed by a single thread because
     autoscoping for some variable s was not successful
   Private variables in R1: i, t
   Shared variables in R1: y, x
     2. C$OMP PARALLEL DO DEFAULT(__AUTO)

   Source loop below has tag L1
   L1 parallelized by explicit user directive
   L1 autoparallelized
   L1 parallel loop-body code placed in function _$d1A2.MAIN_
      along with 0 inne r loops
   L1 could not be pipelined because it contains calls
     3.         DO I=1, 100
     4.                 T = Y(I)
     5.                 CALL FOO(X)
     6.                 X(I) = T*T
     7.         END DO
     8. C$OMP END PARALLEL DO
     9.         END
    10.
```

6.7 自动确定作用域示例

本节提供了一些示例来说明自动确定作用域规则的工作原理。

示例 6–4 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
```

示例 6–4 更复杂的示例 （续）

```
 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，该并行区域包含一个 **SINGLE** 构造、一个工作共享 **DO** 构造和一个 **CRITICAL** 构造。除 OpenMP 并行构造外，并行区域中的代码还执行以下操作：

1. 将数组 **X** 中的值复制到数组 **Y**
2. 查找 **X** 中的最大正数值，并将其存储在 **M** 中
3. 将 **X** 的一些元素的值累加到变量 **W** 中。

编译器如何使用这些规则正确地确定并行区域中变量的作用域?

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

- 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** 在并行区域中只被进行读取，因此不会导致数据争用，这样，按照规则 **S1**，将其作用域确定为 **SHARED**。

- 任何执行并行区域的线程都会执行语句 14，以便将标量 **MM** 的值设置为 0.0。这一写入操作会导致数据争用，因此规则 **S1** 不适用。由于是在同一线程中读取 **MM** 之前出现该写入操作，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**。
- 先读取标量 **W**，然后将其写入语句 21，因此，规则 **S1** 和 **S2** 不适用。加法运算同时符合结合律和交换律，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写入语句 11 不会与读取语句 28 或写入语句 29 同时发生，并且后两者不会同时发生，因为它们都位于同一 **CRITICAL** 构造内。没有任何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** 都不会导致数据争用，按照规则 **S1**，将 **M** 的作用域确定为 **SHARED**。
- 数组 **X** 在区域中只被进行读取，不进行写入，因此，按照规则 **A1**，将其作用域确定为 **SHARED**。
- 写入数组 **Y** 的操作分布在各个线程中，没有任何两个线程会写入 **Y** 的相同元素。由于未发生数据争用，因此，按照规则 **A1** 将 **Y** 的作用域确定为 **SHARED**。

示例 6–5 使用 QuickSort 的示例

```
static void par_quick_sort (int p, int r, float *data)
{
   if (p < r)
   {
      int q = partition (p, r, data);

      #pragma omp task default(__auto) if ((r-p)>=low_limit)
      par_quick_sort (p, q-1, data);

      #pragma omp task default(__auto) if ((r-p)>=low_limit)
      par_quick_sort (q+1, r, data);
   }
}

int main ()
{
  ...
  #pragma omp parallel
  {
     #pragma omp single nowait
     par_quick_sort (0, N-1, &Data[0]);
  }
  ...
}

er_src result:

      Source OpenMP region below has tag R1
      Variables autoscoped as FIRSTPRIVATE in R1: p, q, data
      Firstprivate variables in R1: data, p, q
        47. #pragma omp task default(__auto) if ((r-p)>=low_limit)
        48. par_quick_sort (p, q-1, data);
```

示例 6–5 使用 QuickSort 的示例 （续）

```
      Source OpenMP region below has tag R2
      Variables autoscoped as FIRSTPRIVATE in R2: q, r, data
      Firstprivate variables in R2: data, q, r
        49. #pragma omp task default(__auto) if ((r-p)>=low_limit)
        50. par_quick_sort (q+1, r, data);
```

标量变量 p 和 q 以及指针变量数据在任务构造和并行区域中都是只读的。因此，根据 **TS1** 自动将其作用域确定为 **FIRSTPRIVATE**。

示例 6–6 另一个 Fibbonacci 示例

```
int fib (int n)
{
   int x, y;
   if (n < 2) return n;

   #pragma omp task default(__auto)
   x = fib(n - 1);

   #pragma omp task default(__auto)
   y = fib(n - 2);

   #pragma omp taskwait
   return x + y;
}

er_src result:

   Source OpenMP region below has tag R1
   Variables autoscoped as SHARED in R1: x
   Variables autoscoped as FIRSTPRIVATE in R1: n
   Shared variables in R1: x
   Firstprivate variables in R1: n
    24.         #pragma omp task default(__auto) /* shared(x) firstprivate(n) */
    25.         x = fib(n - 1);

   Source OpenMP region below has tag R2
   Variables autoscoped as SHARED in R2: y
   Variables autoscoped as FIRSTPRIVATE in R2: n
   Shared variables in R2: y
   Firstprivate variables in R2: n
    26.         #pragma omp task default(__auto) /* shared(y) firstprivate(n) */
    27.         y = fib(n - 2);
    28.
    29.         #pragma omp taskwait
    30.         return x + y;
    31. }
```

标量 n 在任务构造和并行构造中都是只读的。因此，根据 TS1 自动将 n 的作用域确定为 **FIRSTPRIVATE**。

标量变量 x 和 y 是函数 `fib()` 的局部变量。在两个任务中访问 x 和 y 不会导致数据争用。由于存在一个 **taskwait**，因此这两个任务将在执行 `fib()` 的线程（遇到了这两个任务）退出 `fib()` 之前完成执行。这意味着 x 和 y 将在这两个任务执行时处于活动状

示例 6–6 另一个 Fibbonacci 示例 （续）

态。因此，根据 TS2 自动将 x 和 y 的作用域确定为 **SHARED**。

示例 6–7 另一个自动确定作用域示例

```
int main(void)
{
  int yy = 0;

  #pragma omp parallel default(__auto) shared(yy)
  {
    int xx = 0;

    #pragma omp single
    {
       #pragma omp task default(__auto) // task1
       {
          xx = 20;
       }
    }

    #pragma omp task default(__auto) // task2
    {
       yy = xx;
    }
  }

  return 0;
}

er_src result:

   Source OpenMP region below has tag R1
   Variables autoscoped as PRIVATE in R1: xx
   Private variables in R1: xx
   Shared variables in R1: yy
     7.   #pragma omp parallel default(__auto) shared(yy)
     8.   {
     9.     int xx = 0;
    10.

   Source OpenMP region below has tag R2
    11.     #pragma omp single
    12.     {

   Source OpenMP region below has tag R3
   Variables autoscoped as SHARED in R3: xx
   Shared variables in R3: xx
    13.        #pragma omp task default(__auto) // task1
    14.        {
    15.           xx = 20;
    16.        }
    17.     }
    18.

   Source OpenMP region below has tag R4
   Variables autoscoped as PRIVATE in R4: yy
```

示例 6–7 另一个自动确定作用域示例 （续）

```
   Variables autoscoped as FIRSTPRIVATE in R4: xx
   Private variables in R4: yy
   Firstprivate variables in R4: xx
    19.     #pragma omp task default(__auto) // task2
    20.     {
    21.        yy = xx;
    22.     }
    23.   }
```

在此示例中，xx 是并行区域中的专用变量。组中的其中一个线程会修改 xx 的初始值（通过执行 task1）。然后，所有线程都会遇到使用 xx 执行某些计算的 task2。

在 task1 中，使用 xx 不会导致数据争用。由于单个构造结尾有一个隐式屏障，而且 task1 应在退出此屏障之前完成，因此，xx 将在 task1 执行时处于活动状态。所以，根据 TS2，在 task1 中自动将 xx 的作用域确定为 **SHARED**。

在 task2 中，xx 的使用是只读的。但是，xx 的使用在包含它的并行构造中不是只读的。由于 xx 在并行构造中预先确定为 **PRIVATE**，因此无法确定 xx 是否将在执行 task2 时处于活动状态。所以，根据 TS3，在 task2 中自动将 xx 的作用域确定为 **FIRSTPRIVATE**。

在 task2 中，使用 yy 会导致数据争用，在每个执行 task2 的线程中，在读取变量 yy 之前始终先由同一线程写入该变量。所以，根据 TS4，在 task2 中自动将 yy 的作用域确定为 **PRIVATE**。

示例 6–8 最后一个示例

```
int foo(void)
{
  int xx = 1, yy = 0;

  #pragma omp parallel shared(xx,yy)
  {
    #pragma omp task default(__auto)
    {
       xx += 1;

       #pragma omp atomic
       yy += xx;
    }

    #pragma omp taskwait
  }
  return 0;
}

er_src result:

   Source OpenMP region below has tag R1
   Shared variables in R1: yy, xx
     5.   #pragma omp parallel shared(xx,yy)
     6.   {
```

示例 6–8 最后一个示例 （续）

```
Source OpenMP region below has tag R2
Variables autoscoped as SHARED in R2: yy
Variables autoscoped as FIRSTPRIVATE in R2: xx
Shared variables in R2: yy
Firstprivate variables in R2: xx
  7.     #pragma omp task default(__auto)
  8.     {
  9.        xx += 1;
 10.
 11.        #pragma omp atomic
 12.        yy += xx;
 13.     }
 14.
 15.     #pragma omp taskwait
 16.   }
```

在 **task** 构造中，xx 的使用不是只读的，且会导致数据争用。但在任务区域中读取 x 会获取在任务之外定义的 x 值。（在此示例中，由于 xx 对于并行区域确定为 **SHARED**，因此 x 的定义实际上在并行区域之外。）所以，根据 TS5，自动将 xx 的作用域确定为 **FIRSTPRIVATE**。

在 **task** 构造中，yy 的使用不是只读的，但不会导致数据争用。由于存在 **taskwait**，因此可在执行任务时访问 yy。所以，根据 TS2，自动将 yy 的作用域确定为 **SHARED**。

◆ ◆ ◆ 第 7 章

7 作用域检查

自动确定作用域功能可以帮助程序员决定如何确定变量的作用域。但是，对于一些复杂程序，自动确定作用域可能不会成功，或者无法实现程序员期望的结果。错误确定作用域可能引发许多不引人注意但很严重的问题。例如，将某些变量的作用域错误地确定为 **SHARED** 可能会导致数据争用；将变量错误地专有化可能会在构造之外为变量使用未定义的值。

Oracle Solaris Studio C、C++ 和 Fortran 编译器提供了一个编译时作用域检查功能，编译器可以通过该功能来确定 OpenMP 程序中的变量是否正确确定了作用域。

根据编译器的功能，作用域检查可以发现数据争用、不适当专有化、变量归约等潜在问题以及其他作用域问题。在作用域检查期间，编译器会对程序员指定的数据共享属性、编译器确定的隐式数据共享属性和自动确定作用域结果进行检查。

7.1 使用作用域检查功能

要启用作用域检查，请使用 **-xvpara** 和 **-xopenmp** 选项在优化级别 **-xO3** 或更高级别上编译 OpenMP 程序。如果只使用 **-xopenmp=noopt** 来编译程序，作用域检查将不起作用。如果优化级别低于 **-xO3**，编译器将发出警告消息，且不执行任何作用域检查。

在作用域检查期间，编译器将检查所有 OpenMP 构造。如果确定有些变量的作用域时引发问题，编译器会发出警告消息，有时还会建议使用正确的数据共享属性子句。

例如：

示例 7–1 作用域检查

```
% cat t.c

#include <omp.h>
#include <string.h>

int main()
{
```

示例 7–1 作用域检查 （续）

```
  int g[100], b, i;

  memset(g, 0, sizeof(int)*100);

  #pragma omp parallel for shared(b)
  for (i = 0; i < 100; i++)
  {
    b += g[i];
  }

  return 0;
}

%  cc -xopenmp -xO3 -xvpara source1.c
"source1.c", line 10: Warning: inappropriate scoping
         variable 'b' may be scoped inappropriately as 'shared'
         . write at line 13 and write at line 13 may cause data race

"source1.c", line 10: Warning: inappropriate scoping
         variable 'b' may be scoped inappropriately as 'shared'
         . write at line 13 and read at line 13 may cause data race
```

如果优化级别低于 **-xO3**，编译器将不进行作用域检查：

```
%  cc -xopenmp=noopt  -xvpara source1.c
 "source1.c", line 10: Warning: Scope checking under vpara compiler
option is supported with optimization level -xO3 or higher.
 Compile with a higher optimization level to enable this feature
```

以下示例显示了潜在的作用域错误：

示例 7–2 作用域错误示例

```
% cat source2.c

#include <omp.h>

int main()
{
  int g[100];
  int r=0, a=1, b, i;

  #pragma omp parallel for private(a) lastprivate(i) reduction(+:r)
  for (i = 0; i < 100; i++)
  {
    g[i] = a;
    b = b + g[i];
    r = r * g[i];
  }

  a = b;
  return 0;
}
```

示例 7–2 作用域错误示例 （续）

```
% cc -xopenmp -xO3 -xvpara source2.c
"source2.c", line 8: Warning: inappropriate scoping
        variable 'r' may be scoped inappropriately as 'reduction'
        . reference at line 13 may not be a reduction of the specified type

"source2.c", line 8: Warning: inappropriate scoping
        variable 'a' may be scoped inappropriately as 'private'
        . read at line 11 may be undefined
        . consider 'firstprivate'

"source2.c", line 8: Warning: inappropriate scoping
        variable 'i' may be scoped inappropriately as 'lastprivate'
        . value defined inside the parallel construct is not used outside
        . consider 'private'

"source2.c", line 8: Warning: inappropriate scoping
        variable 'b' may be scoped inappropriately as 'shared'
        . write at line 12 and write at line 12 may cause data race

"source2.c", line 8: Warning: inappropriate scoping
        variable 'b' may be scoped inappropriately as 'shared'
        . write at line 12 and read at line 12 may cause data race
```

此虚构示例显示了作用域检查可以检测到的一些典型作用域错误。

1. r 被指定为归约变量，其操作为 **+**，但实际上操作应为 *****。
2. a 的作用域显式确定为 **PRIVATE**。由于 **PRIVATE** 变量没有初始值，因此第 11 行中对 a 的引用可能会显示乱码。编译器会指出此问题，并建议程序员考虑将 a 的作用域确定为 **FIRSTPRIVATE**。
3. 变量 i 是循环索引变量。有些情况下，如果在循环后使用循环索引值，程序员可能希望将其指定为 **LASTPRIVATE**。但在示例中，在循环后根本没有引用 i。编译器会发出警告，并建议程序员将 i 的作用域确定为 **PRIVATE**。使用 **PRIVATE** 而不是 **LASTPRIVATE** 可以提高性能。
4. 程序员不为变量 b 显式指定数据共享属性。根据 OpenMP 3.0 规范第 79 页的第 27-28 行，将 b 的作用域隐式确定为 **SHARED**。但是，将 b 的作用域确定为 **SHARED** 将导致数据争用。b 的正确数据共享属性应为 **REDUCTION**。

7.2 限制

- 如所述，作用域检查仅适用于优化级别 **-xO3** 或更高级别。如果只使用 **-xopenmp=noopt** 来编译程序，作用域检查将不起作用。
- 在数据争用分析中，只能识别和使用通过 OpenMP 同步指令（如 **BARRIER** 和 **MASTER**）指定的同步。不识别用户实现的同步，如**忙等待**。

◆ ◆ ◆ 第 8 章

8 性能注意事项

拥有正确且可执行的 OpenMP 程序之后，应该考虑其整体性能。本章介绍了一些可以用于改善 OpenMP 应用程序的效率和可伸缩性的常规技术。

Oracle Solaris Studio 门户发布了有关 OpenMP 应用程序性能分析和优化的临时文章和案例研究，网址为 http://www.oracle.com/technetwork/server-storage/solarisstudio。

8.1 一些常规性能建议

用于改善 OpenMP 应用程序性能的一些常规技术如下：

- 将同步降至最低。
  - 避免或尽量不使用 **BARRIER**、**CRITICAL** 段、**ORDERED** 区域和锁定。
  - 使用 **NOWAIT** 子句可以消除冗余或不必要的屏障。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屏障。
  - 使用已命名的 **CRITICAL** 段进行细粒度锁定。
  - 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。

  缺省情况下，空闲线程将在特定的超时期限后进入休眠状态。如果线程在超时期限结束时未找到工作，则会进入休眠状态，从而避免以其他线程为代价浪费处理器周期。缺省超时期限对于您的应用程序而言可能过短，从而导致线程过早或过晚地进入休眠状态。通常，如果应用程序在专用处理器上运行，则 **SPIN** 将会提供最佳的应用程序性能。如果某一应用程序与其他应用程序同时共享处理器，则缺省设置或者不使线程过久旋转的设置对于系统总处理能力来说最佳。使用 **SUNW_MP_THR_IDLE** 环境变量覆盖缺省超时期限，甚至可以使空闲线程永不进入休眠状态并始终处于活动状态。
- 尽可能在最高级别（如外部 **DO/FOR** 循环）执行并行化。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，以下构造效率不高：

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

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

更有效的构造：

```
!$OMP PARALLEL
  ....
  !$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，以下构造效率不高：

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

而以下构造更有效：

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

- 在 Oracle Solaris 系统中，使用 **SUNW_MP_PROCBIND** 将线程绑定到处理器。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。请参见第 19 页中的 “2.3 处理器绑定”。
- 尽可能使用 **MASTER**，而不是 **SINGLE**。
  - **MASTER** 指令作为不带隐式 **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** 会增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，并且可以在数据装入高速缓存时保持数据的局域性。但是，**STATIC** 可能导致负载不平衡。
- 由于 **DYNAMIC** 和 **GUIDED** 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，因为它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。这会在并行 **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()** 具有由内部锁定造成的高同步开销。可以在 **libmtmalloc** 库中找到更快的版本。请用 **-lmtmalloc** 进行链接以使用 **libmtmalloc**。

在数据量较少的情况下，OpenMP 并行循环可能性能不佳。使用 **PARALLEL** 构造中的 **if** 子句指示循环仅应在预期可以提高一定性能的情况下并行运行。

- 如果可能，请合并循环。例如，合并两个循环：

```
!$omp parallel do
  do i = ...
.........statements_1...
  end do
!$omp parallel do
  do i = ...
.........statements_2...
  end do
```

合并为单个循环：

```
!$omp parallel do
  do i = ...
.........statements_1...
.........statements_2...
  end do
```

- 如果应用程序缺乏超出某个级别的可伸缩性，请尝试使用嵌套并行操作。有关 OpenMP 中嵌套并行操作的更多信息，请参见第 11 页中的 “1.2 特殊约定”。

8.2 伪共享及其避免方法

如果不慎将共享内存结构与 OpenMP 应用程序一起使用，可能导致性能下降且可伸缩性受限制。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。

8.2.1 什么是伪共享？

大多数高性能处理器（如 UltraSPARC 处理器）在 CPU 的低速内存和高速寄存器之间插入一个高速缓存缓冲区。访问内存位置时，会使包含所请求内存位置的一部分实际内存（**缓存代码行**）被复制到高速缓存中。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。

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。每次对缓存代码行的单个元素进行更新时，都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**。即使所访问的元素尚未被修改，也会强制它们从内存或其他位置获取该代码行的较新副本。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。因此，互连通信和开销方面都将有所增长。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**。如果此情况频繁发生，OpenMP 应用程序的性能和可伸缩性就会显著下降。

在出现以下所有情况时，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。
- 数据更新发生的频率非常高（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。

8.2.2 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，可以通过以下方法减少伪共享：

- 尽可能使用专用数据
- 利用编译器的优化功能来消除内存负载和存储

在特定情况下，如果处理较大的问题而共享较少，可能较难看到伪共享的影响。

处理伪共享的方法与特定应用程序紧密相关。在某些情况下，更改数据的分配方式可以减少伪共享。在其他情况下，通过更改迭代到线程的映射，为每个线程的每个块分配更多的工作（通过更改 *chunksize* 值），也可以减少伪共享。

8.3 Oracle Solaris OS 调优特性

Oracle Solaris 支持无需进行硬件升级即可提高 OpenMP 程序性能的特性，其中包括内存定位优化 (Memory Placement Optimization, MPO) 和多页大小支持 (Multiple Page Size Support, MPSS) 等。

MPO 使 OS 可以将页面分配到访问这些页面的处理器附近。SunFire E20K 和 SunFire E25K 系统在相同的 UniBoard 内（与不同的 UniBoard 之间相比）有不同的内存延迟。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。

通过 MPSS 功能，程序可以对虚拟内存的不同区域使用不同的页面大小。缺省 Oracle Solaris 页面大小相对较小（在 UltraSPARC 处理器上为 8 KB，在 AMD64 Opteron 处理器上为 4 KB）。通过使用较大的页面大小，过多发生 TLB 未命中情况的应用程序可以提高性能。

使用 Oracle Solaris Studio 性能分析器可以测量 TLB 未命中次数。

使用以下 Oracle Solaris OS 命令可以获取特定平台的缺省页面大小：**/usr/bin/pagesize**。在此命令中使用 **-a** 选项可列出所有受支持的页面大小。（有关详细信息，请参见 **pagesize**(1) 手册页。）

三种可更改应用程序的缺省页面大小的方法为：

- 使用 Oracle Solaris OS 命令 **ppgsz**(1)。
- 使用 **-xpagesize**、**-xpagesize_heap** 和 **-xpagesize_stack** 选项编译应用程序。有关详细信息，请参见编译器手册页。
- 使用 MPSS 特有的环境变量。有关详细信息，请参见 **mpss.so.1**(1) 手册页。

◆ ◆ ◆ 附 录 A

子句在指令中的放置

表 A–1 拥有子句的 Pragma

子句/Pragma	parallel	do/for	sections	single	parallel do/for	parallel sections	parallel workshare	task
if	+				+	+	+	+
private	+	+	+	+	+	+	+	+
shared	+				+	+	+	+
firstprivate	+	+	+	+	+	+	+	+
lastprivate		+	+		+	+		
default	+				+	+	+	+
reduction	+	+	+		+	+	+	
copyin	+				+	+	+	
copyprivate				+ (1)				
ordered		+			+			
schedule		+			+			
nowait		+ (2)	+ (2)	+ (2)				
num_threads	+				+	+	+	
untied								+
final								+
mergeable								+
__auto	+				+	+	+	+

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

索引

自