OpenMP API 用户指南

Sun™ Studio 11

Sun Microsystems, Inc.
www.sun.com

文件号码 819-4818-10
2005 年 11 月，修订版 A

请将关于本文档的意见和建议提交至：http://www.sun.com/hwdocs/feedback

目录

阅读本书之前　ix

印刷约定　ix

Shell 提示符　x

支持的平台　x

访问 Sun Studio 软件和手册页　xi

访问编译器和工具文档　xiii

访问相关的 Solaris 文档　xv

开发者资源　xvi

联系 Sun 技术支持　xvi

Sun 欢迎您提出意见　xvi

1.　OpenMP API 简介　1–1

1.1　哪里有 OpenMP 规范　1–1

1.2　本章所使用的特殊约定　1–2

2.　嵌套并行操作　2–1

2.1　执行模型　2–1

2.2　控制嵌套并行操作　2–2

2.2.1　**`OMP_NESTED`**　2–2

2.2.2　**`SUNW_MP_MAX_POOL_THREADS`**　2–3

2.2.3 **SUNW_MP_MAX_NESTED_LEVELS** 2–4
2.3 在嵌套并行区域中使用 OpenMP 库例程 2–7
2.4 有关使用嵌套并行操作的一些提示 2–10

3. 自动确定变量的作用域 3–1
3.1 自动确定作用域数据范围子句 3–1
3.1.1 **__AUTO** 子句 3–1
3.1.2 **DEFAULT(__AUTO)** 子句 3–2
3.2 作用域规则 3–2
3.2.1 标量变量的作用域规则 3–2
3.2.2 数组的作用域规则 3–3
3.3 关于自动确定作用域的通用注释 3–3
3.3.1 Fortran 95 的自动确定作用域规则： 3–3
3.3.2 C/C++ 的自动确定作用域规则： 3–3
3.4 检查自动确定作用域的结果 3–4
3.5 当前实现的已知限制 3–8

4. 实现定义的行为 4–1

5. OpenMP 编译 5–1
5.1 要使用的编译器选项 5–1
5.2 Fortran 95 OpenMP 验证 5–3
5.3 OpenMP 环境变量 5–4
5.4 处理器绑定 5–7
5.5 栈和栈大小 5–10

6. 转换为 OpenMP 6–1
6.1 转换传统 Fortran 指令 6–1
6.1.1 转换 Sun 风格的 Fortran 指令 6–2
6.1.2 转换 Cray 风格的 Fortran 指令 6–3

6.2 转换传统 C Pragma 6–4

6.2.1 传统 C Pragma 与 OpenMP 间的问题 6–5

7. 性能注意事项 7–1

7.1 一般性建议 7–1

7.2 伪共享及其避免方法 7–4

7.2.1 何为*伪共享*? 7–4

7.2.2 减少伪共享 7–5

7.3 操作系统优化功能 7–5

A. 子句在指令中的放置 A–1

索引 索引 –1

表

表 5-1 OpenMP 环境变量 5–4

表 5-2 多重处理环境变量 5–5

表 6-1 将 Sun 并行化指令转换为 OpenMP 6–2

表 6-2 `DOALL` 限定符子句和等效的 OpenMP 子句 6–2

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

表 6-4 Cray 风格的 `DOALL` 限定符子句的等效 OpenMP 子句 6–3

表 6-5 将传统 C 并行化 Pragma 转换为 OpenMP 6–4

表 6-6 `taskloop` 可选子句和等效的 OpenMP 子句 6–5

表 6-7 `SCHEDTYPE` 调度和等效的 OpenMP `schedule` 6–5

表 A-1 拥有子句的 Pragma A–1

阅读本书之前

《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。

本指南专供具有 Fortran、C 或 C++ 语言及 OpenMP 并行编程模型工作经验的科学工作者、工程技术人员以及编程人员使用。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[1]	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。 `% You have mail.`
`AaBbCc123`	用户键入的内容，与计算机屏幕输出的显示不同。	`% su` `Password:`
AaBbCc123	保留未译的新词或术语以及要强调的词。要使用实名或值替换的命令行变量。	这些称为 *class* 选项。 要删除文件，请键入 **`rm`** *filename*。
新词术语强调	新词或术语以及要强调的词。	您**必须**成为超级用户才能执行此操作。
《书名》	书名	阅读 《用户指南》的第 6 章。

1 浏览器的设置可能会与这些设置不同。

表 P-2 代码约定

代码符号	含义	表示法	代码示例
[ ]	方括号包含可选参数。	O[*n*]	-O4, -O
{ }	花括号中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，分号有时可用于分隔参数。	R*dir*[:*dir*]	-R/local/libs:/U/a
…	省略号表示一系列的省略。	-xinline=*f1*[,…*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 超级用户	#

支持的平台

此 Sun Studio 发行版本支持使用 SPARC® 和 x86 系列处理器体系结构（UltraSPARC®、SPARC64、 AMD64、 Pentium 和 Xeon EM64T）的系统。通过访问 http://www.sun.com/bigadmin/hcl 中的硬件兼容性列表，可以了解您在使用的 Solaris 操作系统版本的支持系统。这些文档列出了实现各个平台类型的所有差别。

在本文档中，这些与 x86 有关的术语具有以下含义：

- “x86”是指较大的 64 位和 32 位 x86 兼容产品系列。
- “x64”表示有关 AMD64 或 EM64T 系统的特定 64 位信息。
- “32 位 x86”表示有关基于 x86 的系统的特定 32 位信息。

有关所支持的系统，请参见硬件兼容性列表。

访问 Sun Studio 软件和手册页

Sun Studio 软件及其手册页未安装到 /usr/bin/ 和 /usr/share/man 标准目录中。要访问该软件，必须正确设置 PATH 环境变量（请参见 第 xi 页的 “访问软件”）。要访问手册页，必须正确设置 MANPATH 环境变量 （请参见第 xii 页的 “访问手册页”）。

有关 PATH 变量的详细信息，请参见 csh(1)、 sh(1)、 ksh(1) 和 bash(1) 手册页。有关 MANPATH 变量的详细信息，请参见 man(1) 手册页。有关设置 PATH 变量和 MANPATH 变量以访问此发行版本的详细信息，请参见安装指南或询问系统管理员。

注 – 本节中的信息假设 Sun Studio 软件安装在 Solaris 平台上的 /opt 目录和 Linux 平台上的 /opt/sun 目录中。如果未将软件安装在默认的目录中，请询问系统管理员以获取系统中的相应路径。

访问软件

使用以下步骤决定是否需要更改 PATH 变量以访问该软件。

决定是否需要设置 PATH 环境变量

1. **通过在命令提示符后键入以下内容以显示 PATH 变量的当前值。**

```
% echo $PATH
```

2. **在 Solaris 平台上，查看输出中是否包含有 /opt/SUNWspro/bin 的路径字符串。在 Linux 平台上，查看输出中是否包含有 /opt/sun/sunstudio11/bin 的路径字符串。**

   如果找到该路径，则说明已设置了访问该软件的 PATH 变量。如果没有找到该路径，则按照下一步中的说明设置 PATH 环境变量。

设置 PATH 环境变量以访问软件

- **在 Solaris 平台上，将以下路径添加到 PATH 环境变量中。如果以前安装了 Forte Developer 软件、 Sun ONE Studio 软件、或其他发行版本的 Sun Studio 软件，则将以下路径添加到这些软件安装路径之前。**

  /opt/SUNWspro/bin

- **在 Linux 平台上，将以下路径添加到 PATH 环境变量中。**
  /opt/sun/sunstudio10u1/bin

访问手册页

使用以下步骤决定是否需要更改 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键入以下内容以请求 dbx 手册页。**

```
% man dbx
```

2. **请查看输出 （如果有）。**
   如果找不到 dbx(1) 手册页或者显示的手册页不是软件当前版本的手册页，请按照下一步的说明来设置 MANPATH 环境变量。

设置 MANPATH 环境变量以实现对手册页的访问

- **在 Solaris 平台上，将以下路径添加到 MANPATH 环境变量中。**
  /opt/SUNWspro/man
- **在 Linux 平台上，将以下路径添加到 MANPATH 环境变量中。**
  /opt/sun/sunstudio11/man

访问集成开发环境

Sun Studio 集成开发环境 (integrated development environment, IDE) 提供了创建、编辑、生成、调试 C、 C++ 或 Fortran 应用程序并分析其性能的模块。

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

IDE 是否可以正确操作取决于 IDE 能否找到核心平台。 sunstudio 命令会查找两个位置的核心平台：

- 该命令首先查找 Solaris 平台上的默认安装目录 /opt/netbeans/3.5V11 和 Linux 平台上的默认安装目录 /opt/sun/netbeans/3.5V11。

- 如果该命令在默认目录中找不到核心平台，则它会假设包含 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，在 Solaris 平台上，如果包含 IDE 的目录的路径是 `/foo/SUNWspro`，则该命令会在 `/foo/netbeans/3.5V11` 中查找核心平台。在 Linux 平台上，如果包含 IDE 的目录的路径是 `/foo/sunstudio11`，则该命令会在 `/foo/netbeans/3.5V11` 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，则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/3.5V11`)。

在 Solaris 平台上，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用户 `$PATH` 中。在 Linux 平台上， IDE 的每个用户还必须将 */installation_directory*`/sunstudio11/bin` 添加到其他任何 Sun Studio 软件发行版本路径前面的用户 `$PATH` 中。

路径 */installation_directory*`/netbeans/3.5V11/bin` 不能添加到用户的 `$PATH` 中。

访问编译器和工具文档

您可以访问以下位置的文档：

- 可以通过随软件一起安装在本地系统或网络中的文档索引获取文档，位置为 Solaris 平台上的 `file:/opt/SUNWspro/docs/zh/index.html` 和 Linux 平台上的 `file:/opt/sun/sunstudio11/docs/zh/index.html`。

  如果软件未安装在 Solaris 平台的 `/opt` 目录或 Linux 平台的 `/opt/sun` 目录中，请询问系统管理员以获取系统中的相应路径。

- 大多数的手册都可以从 `docs.sun.com`℠ Web 站点上获取。以下书目只能从 Solaris 平台上安装的软件中找到：
  - 《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》
- 适用于 Solaris 平台和 Linux 平台的发行说明可以从 `docs.sun.com` Web 站点获取。
- 在 IDE 中通过“帮助”菜单以及许多窗口和对话框中的“帮助”按钮，可以访问 IDE 的所有组件的联机帮助。

您可以通过 Internet 访问 `docs.sun.com` Web 站点 (`http://docs.sun.com`) 以阅读、打印和购买 Sun Microsystems 的各种手册。如果找不到手册，请参见与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、产品或其他资料，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、物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

该文档采用易读格式提供，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述，找到文档的易读版本。如果未将软件安装在 `/opt` 目录中，请询问系统管理员以获取系统中的相应路径。

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》	安装软件所包含的 HTML，位于 Solaris 平台上的文档索引 `file:/opt/SUNWspro/docs/zh/index.html` 中
自述文件	HTML，位于开发者门户 `http://developers.sun.com/prodtech/cc/documentation/ss11/mr/READMEs` 中
手册页	安装软件所包含的 HTML，位于 Solaris 平台的文档索引 `file:/opt/SUNWspro/docs/zh/index.html` 和 Linux 平台的文档索引 `file:/opt/sun/sunstudio11/docs/index.html` 中
联机帮助	HTML，可通过 IDE 中的 “帮助” 菜单和 “帮助” 按钮访问
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通过 `file:/opt/SUNWspro/docs/zh/index.html` 和 `http://docs.sun.com` 站点获取。如果未将软件安装在 `/opt` 目录中，请询问系统管理员以获取系统中的相应路径。

文档标题	描述
Fortran 编程指南	描述了如何在 Solaris 环境中编写高效的 Fortran 代码；并且描述了输入 / 输出、库、性能、调试和并行处理信息。
《Fortran 库参考》	详细说明了 Fortran 库和内部例程
《Fortran 用户指南》	描述了 `f95` 编译器的编译时环境和命令行选项。还包括了关于将以前的 `f77` 程序迁移到 `f95` 中的说明。
《C 用户指南》	描述了 `cc` 编译器的编译时环境和命令行选项。
《C++ 用户指南》	描述了 `CC` 编译器的编译时环境和命令行选项。
《数值计算指南》	描述了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 站点上获取的相关文档。

文档集合	文档标题	描述
Solaris 参考手册集合	请参见手册页部分的标题。	提供有关 Solaris 操作系统的信息。
Solaris 软件开发者集合	《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 软件开发者集合	《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。

开发者资源

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

- 有关编程技术和最佳实例的文章
- 有关编程小技巧的知识库
- 有关编译器和工具组件的文档以及与软件安装在一起的文档的修正内容
- 有关支持级别的信息
- 用户论坛
- 可下载的代码样例
- 新技术预览

您可以通过访问 http://developers.sun.com 找到其他开发者资源。

联系 Sun 技术支持

如果您遇到通过本文档无法解决的技术问题，请访问以下网址：

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

Sun 欢迎您提出意见

Sun 致力于提高其文档的质量，并十分乐意收到您的意见和建议。您可以通过以下网址提交您的意见和建议：

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

请在您的电子邮件主题行中注明文档的文件号码 (819-4818-10)。

第1章

OpenMP API 简介

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。

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，请参见第 6 章。

1.1 哪里有 OpenMP 规范

本手册所提供的材料描述了 OpenMP API 的 Sun Studio 实现所特有的问题。有关完整的详细信息，请参阅 OpenMP 规范文档。本手册直接引用了 OpenMP 2.5 API 规范中的部分。

C、C++ 和 Fortran 95 的 OpenMP 2.5 规范可通过访问 OpenMP 官方网站 `http://www.openmp.org/` 获取。

有关 OpenMP 的其他信息（包括教程和其他开发者资源）可通过访问 cOMPunity 网站 `http://www.compunity.org/` 获取。

有关 Sun Studio 编译器发行版本及其 OpenMP API 实现的最新信息可通过访问 Sun Developer Network 门户 `http://developers.sun.com/sunstudio` 获取。

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。

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() 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。

代码示例 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。默认值是 1023。

线程池只包含运行时库创建的非用户线程。它不包含初始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活动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活动并行区域将仅由一个线程来执行。如果并行区域是 OpenMP 并行区域，且该并行区域的 **IF** 子句（如果指定）的值为 True，则视该并行区域为活动区域。仅对活动并行区域进行计算。默认的最大活动嵌套级别数为 4。

以下代码将创建 4 个级别的嵌套并行区域。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。（实际结果将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
```

这些示例仅显示了一些**可能的**结果。实际结果将取决于操作系统调度线程的方式。

2.3 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” 调用只影响调用同一级或内部嵌套级别的线程所遇到的并行区域。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。

代码示例 2-2 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);       /* 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);      /* 行 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可以使多个线程参与计算。

  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。再假定您的系统有 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。

- 嵌套并行区域可以轻松地创建过多的线程，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，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 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。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

自动确定变量的作用域

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。

Sun Studio C、 C++ 和 Fortran 95 编译器提供了自动确定作用域的功能。这些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自动确定变量的作用域。

3.1 自动确定作用域数据范围子句

自动确定作用域数据子句是 Sun 对 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 (*list-of-variables***)** Fortran 95 指令
__auto (*list-of-variables***)** C 和 C++ Pragma

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** 和 **auto** 之前的**两个**下划线）。

__AUTO 或 **__auto** 子句可以出现在 **PARALLEL**、**PARALLEL DO**、**PARALLEL SECTIONS** 中或 Fortran 95 的 **PARALLEL WORKSHARE** 指令中。

如果变量列在该子句中，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

DEFAULT(__AUTO)	Fortran 95 指令
default(__auto)	C 和 C++ Pragma

将此并行区域中的默认作用域设置为 **__AUTO**。

DEFAULT(__AUTO) 子句可以出现在 **PARALLEL**、**PARALLEL DO**、**PARALLEL SECTIONS** 中或 Fortran 95 的 **PARALLEL WORKSHARE** 指令中。

3.2 作用域规则

在自动确定作用域下，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** 或 **FOR** 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**。

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。

3.2.2 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**。

3.3 关于自动确定作用域的通用注释

自动确定不具有隐式作用域的变量的作用域时，编译器将根据规则，按照所给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。如果编译器找不到匹配规则，它将放弃确定该变量的作用域，而将其作用域确定为 **SHARED**，并像指定了 **IF (.FALSE.)** 或 **if(0)** 子句一样，将绑定并行区域序列化。

自动确定作用域失败的原因有两个。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过于复杂，无法进行全面的分析。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引起失败的常见原因。（请参见第 3-8 页 3.5 节的“当前实现的已知限制”。）

3.3.1 Fortran 95 的自动确定作用域规则：

对于 Fortran，如果指定由 **__AUTO** 或 **DEFAULT(__AUTO)** 指令自动确定以下种类变量的作用域，将导致编译器按照 OpenMP 规范中的隐式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共享任务构造中使用，并且在每个这种构造的数据作用域属性子句中指定。

3.3.2 C/C++ 的自动确定作用域规则：

对于 C/C++，如果用户指定由 **__auto** 或 **default(__auto)** Pragma 自动确定以下变量的作用域，编译器将按照 OpenMP 规范的隐式作用域规则确定变量的作用域：

- 变量在并行构造中被声明。
- 变量具有 **THREADPRIVATE** 属性。
- 变量为 const 限定类型。

- 变量是紧跟在 **for** 或 **parallel for** Pragma 之后的 **for** 循环的循环控制变量，而且变量引用出现在循环内。

C 和 C++ 中的自动确定作用域仅适用于基本数据类型：整型、浮点型和指针。如果用户指定了要自动确定作用域的结构变量或类变量，编译器会将变量的作用域确定为 **shared**，并由单线程执行对并行区域的封闭。

3.4 检查自动确定作用域的结果

使用**编译器注释**检查自动确定作用域结果，并且查看并行区域是否因为自动确定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 代码示例 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，请参见 er_src(1) 手册页或 《Sun 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。如果自动确定作用域失败，将打印一条警告消息 （如代码示例 3-1 所示）。

代码示例 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
       因为以下变量的自动确定作用域失败
       - x
```

用 f95 的 -vpara 或 cc 的 -xvpara 进行编译。（CC 中尚未实现此选项。）

代码示例 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 omp_t.o
源文件：./omp_t.f
目标文件：./omp_t.o
加载对象：./omp_t.o

     1.       INTEGER X(100), Y(100), I, T
        <Function:MAIN_>

    Source OpenMP region below has tag R1
    Variables autoscoped as PRIVATE in R1:t, i
    Variables autoscoped as SHARED in R1:x, y
    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
    Source loop below has tag L1
    L1 parallelized by explicit user directive
    Discovered loop below has tag L2
    L-unknown scheduled with steady-state cycle count = 3
    L-unknown unrolled 4 times
    L-unknown has 0 loads, 0 stores, 2 prefetches, 0 FPadds, 0 FPmuls, and 0
FPdivs per iteration
   L-unknown has 1 int-loads, 1 int-stores, 4 alu-ops, 1 muls, 0 int-divs and
1 shifts per iteration
     3.       DO I=1, 100
     4.          T = Y(I)
     5.          X(I) = T*T
     6.       END DO
     7. C$OMP END PARALLEL DO
     8.       END
```

接下来，用一个更复杂的示例来解释自动确定作用域规则的应用方式。

代码示例 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。

在并行区域中使用以下变量：**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

此处介绍了当前 Sun Studio Fortran 95 编译器中自动确定作用域的已知限制。

- 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，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。
- 只有通过使用 OpenMP 同步指令指定的同步（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** 编译时不支持自动确定作用域。

第 4 章

实现定义的行为

本章说明 OpenMP 2.5 规范中依赖实现的特定行为。有关最新编译器发行版本的最新信息，请参见 Sun Developer Network 门户 `http://developers.sun.com/sunstudio` 上的编译器文档。

- **内存模型**

  多个线程*异步*访问同一变量时，无需保证内存访问空间的大小。

  某些依赖实现以及依赖应用程序的因素会对是否为原子访问产生影响。有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需的空间大。有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 loads/stores 来访问变量。有时，使用多个 loads/stores 会让代码序列的运行速度更快。

- **内部控制变量**

  OpenMP 运行时库维护以下内部控制变量：

  nthreads-var - 存储以后的并行区域所需的线程数。

  dyn-var - 控制是否为以后的并行区域所用的线程数启用动态调整。

  nest-var - 控制是否为以后的并行区域启用嵌套并行操作。

  run-sched-var - 使用 **`RUNTIME`** 调度子句，存储用于循环区域的调度信息。

  def-sched-var - 为循环区域存储实现定义的默认调度信息。

  运行时库一方面维护每个线程的 *nthreads-var*、*dyn-var* 和 *nest-var* 各自的单独副本。另一方面，维护应用于所有线程的 *run-sched-var* 和 *def-sched-var* 各自的一个副本。

- **线程数**

  nthreads-var 的默认值为 1。即，如果没有显式的 **`num_threads()`** 子句、**`omp_set_num_threads()`** 例程的调用或 **`OMP_NUM_THREADS`** 环境变量的显式定义，则组内的默认线程数为 1。

  调用 **`omp_set_num_threads()`** 仅修改调用线程的 *nthreads-var* 值，并应用于调用线程遇到的同一嵌套级别或内部嵌套级别的并行区域。

  如果所需的线程数大于实现可以支持的线程数，或者 *nthreads-var* 值不是正整数，在将 **`SUNW_MP_WARN`** 设置为 **`TRUE`** 或者通过调用 **`sunw_mp_register_warn()`** 注册回调函数时，编译器会发出警告信息。

- **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。

  nest-var 的默认值为 *False*。即，默认情况下禁用嵌套并行操作。要启用嵌套并行操作，请设置 **OMP_NESTED** 环境变量或调用 **omp_set_nested()** 例程。

  调用 **omp_set_nested()** 仅修改调用线程的 *nest-var* 值，并应用于调用线程遇到的同一嵌套级别或内部嵌套级别的并行区域。

  默认情况下，支持的最大活动嵌套级别数为 4。可通过设置环境变量 **SUNW_MP_MAX_NESTED_LEVELS** 更改该最大值。

- **线程的动态调整**

  dyn-var 的默认值为 *True*。即，默认情况下启用动态调整。要禁用动态调整，请设置 **OMP_DYNAMIC** 环境变量或调用 **omp_set_dynamic()** 例程。

  调用 **omp_set_dynamic()** 仅修改调用线程的 *dyn-var* 值，并应用于调用线程遇到的同一嵌套级别或内部嵌套级别的并行区域。

  如果启用了动态调整，则将组内的线程数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

  相反，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

  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 **SUNW_MP_WARN** 设置为 **TRUE** 或者通过对 **sunw_mp_register_warn()** 的调用注册回调函数，将出现警告消息。

  有关线程池和嵌套并行操作执行模型的详细信息，请参阅第 2 章。

- **循环调度**

  def-sched-var 的默认值为 **STATIC** 调度。要指定循环区域的其他调度，请使用 **SCHEDULE** 子句。

  run-sched-var 的默认值也是 **STATIC** 调度。可通过设置 **OMP_SCHEDULE** 环境变量更改该默认值

- **GUIDED：确定块大小**

  如果未指定 *chunksize*，则 **SCHEDULE(GUIDED)** 的默认块大小为 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** 环境变量指定的浮点常量（第 5-4 页 5.3 节的“OpenMP 环境变量”）。如果未指定，则当前的默认设置假定 *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 线程的显式线程程序可以包含 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 5-4 页 5.3 节的“OpenMP 环境变量”）就会启用由 OpenMP 运行时库执行的运行时有效性检查。

    例如，以下代码属于无限循环，因为线程在不同的屏障处等待，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**，运行时库将检测到该问题：

```
% 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 和 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。在检测到错误时，将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()` 安装为回调函数。此示例中的回调函数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。

- **关于特定构造：**

  sections 构造

  在执行段区域的组成员中分配 **sections** 构造中的结构化块，以便线程执行的段数大致相等。

  single 构造

  single 构造的结构化块将由首先遇到单个区域的线程执行。

  atomic 构造

  此实现通过在 **CRITICAL** 构造中封装目标语句来替换所有 **ATOMIC** 指令和 Pragma。

- **OpenMP 库的绑定线程集例程：**

  omp_set_num_threads 例程

  当从显式并行区域内调用时， omp_set_num_threads 区域的绑定线程集就是调用线程。

  omp_get_max_threads 例程

  当从显式并行区域内调用时， omp_get_max_threads 区域的绑定线程集就是调用线程。

  omp_set_dynamic 例程

  当从任何显式并行区域内调用时，只有 omp_set_dynamic 区域的绑定线程集才是调用线程。

  omp_get_dynamic 例程

  当从显式并行区域内调用时，只有 omp_get_dynamic 区域的绑定线程集才是调用线程。

  omp_set_nested 例程

  当从显式并行区域内调用时，只有 omp_set_nested 区域的绑定线程集才是调用线程。

  omp_get_nested 例程

  当从显式并行区域内调用时，只有 omp_get_nested 区域的绑定线程集才是调用线程。

- **Fortran 95 特定的问题：**

  threadprivate 指令

  如果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）的线程专用对象中数据值的条件并不都成立，则第二个区域中可分配数组的分配状态可能为“不是当前分配的”。

`shared` 子句

将共享变量传递到非内在过程可能会产生共享变量的值，该共享变量在过程引用之前要复制到临时存储，并在过程引用之后又从临时存储复制到实际的参数存储。仅当 OpenMP 2.5 规范的 2.8.3.2 节中的条件 a、b 和 c 同时成立时，才会发生复制进和复制出临时存储这种情况。

包含文件和模块文件

此实现中同时提供了包含文件 `omp_lib.h` 和模块文件 `omp_lib`。

采用参数的 OpenMP 运行时库例程是通过通用接口扩展的，因此可以提供不同 Fortran `KIND` 类型的参数。

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。默认值为 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。

有关 Sun Studio 编译器和 OpenMP 的最新信息可以通过访问 Sun Developer Network 门户 `http://developers.sun.com/sunstudio` 获取

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** 作为同义字接受。）

-xopenmp 标志接受以下关键字子选项。

-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 Pragma，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用户而提供的。要编译调用 OpenMP 库例程但忽略 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 Pragma 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 **200505L**，Fortran 95 定义为 **200505**）。
- 使用 dbx 调试 OpenMP 程序时，请使用 **-xopenmp=noopt -g** 进行编译
- **-xopenmp** 的默认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。
- 使用 C 和 Fortran 95 的 **-xvpara** 选项可以显示编译器并行性信息。

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）。编译器将诊断以下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**、**for**、**parallel for** 指令 /pragma 设置调度类型。未定义时使用默认值 **STATIC**。值为 *"type[,chunk]"* 示例：`setenv OMP_SCHEDULE 揌UIDED,4î`
OMP_NUM_THREADS 或 **PARALLEL**	并行区域执行时设置需使用的线程数。通过 **NUM_THREADS** 子句或调用 **OMP_SET_NUM_THREADS()** 可以覆盖此值。未设置时使用默认值 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。未设置时使用默认值 **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**。默认值为 **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。默认值为 **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。请参见第 3 页的 “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**。默认为 **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 程序的线程绑定到处理器。虽然可以通过处理器绑定来增强性能，但是如果将多个线程绑定到相同的处理器，则将导致性能下降。详细信息，请参见第 5-7 页 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，则默认值是 1023。有关详细信息，请参见第 2-2 页 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，则默认值是 4。有关详细信息，请参见第 2-2 页 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。默认线程栈大小在 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** **STACKSIZE** 环境变量值还可以是单位后缀为 **B**、**K**、**M** 或 **G**（分别表示字节、千字节、兆字节或千兆字节）的数值。默认单位为千字节。
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，则假定的默认值为 2.0。

5.4 处理器绑定

通过处理器绑定，编程人员可以指示操作系统 (Solaris) 程序中的线程在程序的整个执行过程中应该在同一处理器上运行。

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，由并行区域或共享任务区域中的线程访问的数据将位于上一次所调用并行区域或共享任务区域的本地缓存中。

从硬件的角度看，计算机系统由一个或多个**物理**处理器组成。从操作系统 (Solaris) 的角度看，其中每个**物理**处理器都映射到程序中线程可以在其上运行的一个或多个**虚拟**处理器。例如，每个 UltraSPARC IV 物理处理器都有两个核心。从 Solaris 操作系统的角度看，其中每个核心都是可以调度线程在其上运行的**虚拟**处理器。

当操作系统将线程绑定到处理器时，这些线程实际上是被绑定到特定的**虚拟**处理器，而不是**物理**处理器。

要将 OpenMP 程序中的线程绑定到特定的虚拟处理器，请设置 **SUNW_MP_PROCBIND** 环境变量。为 **SUNW_MP_PROCBIND** 指定的值可为以下任一值：

- 字符串 "**TRUE**" 或 "**FALSE**"（或者小写形式 "**true**" 或 "**false**"）。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*，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，上面提到的非负整数指逻辑标识符 (ID)。逻辑 ID 可能与**虚拟**处理器 ID 不同。这一不同将在下面进行说明。

虚拟处理器 ID：

系统中的每个虚拟处理器都有唯一的处理器 ID。您可以使用 Solaris 操作系统的 psrinfo(1M) 命令显示有关系统中处理器的信息（包括其处理器 ID）。此外，您还可以使用 prtdiag(1M) 命令显示系统配置和诊断信息。

在以后的 Solaris 发行版本中，您可以使用 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，也可能是不连续的。例如，在具有 8 个 UltraSPARC IV 处理器（16 个核心）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。

逻辑 ID：

如上所述，为 SUNW_MP_PROCBIND 指定的非负整数是逻辑 ID。逻辑 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*，其逻辑 ID 则为 0、 1、 ...、 *n*-1（按 psrinfo(1M) 命令显示顺序）。以下 Korn shell 脚本可用于显示从虚拟处理器 ID 到逻辑 ID 的映射。

```
#!/bin/ksh

NUMV=`psrinfo | fgrep "on-line" | wc -l`
set -A VID `psrinfo | cut -f1`

echo "Total number of on-line virtual processors = $NUMV"
echo

let "I=0"
let "J=0"
while [[ $I -lt $NUMV ]]
do
  echo "Virtual processor ID ${VID[I]} maps to logical ID ${J}"
  let "I=I+1"
  let "J=J+1"
done
```

在单个物理处理器映射到多个虚拟处理器的系统上，知道哪些逻辑 ID 与属于同一物理处理器的虚拟处理器相对应会很有用。以下 Korn shell 脚本可在以后的 Solaris 发行版本中显示此信息。

```
#!/bin/ksh

NUMV=`psrinfo | grep "on-line" | wc -l`
set -A VLIST `psrinfo | cut -f1`
set -A CHECKLIST `psrinfo | cut -f1`

let "I=0"

while [ $I -lt $NUMV ]
do
  let "COUNT=0"
  SAMELIST="$I"

  let "J=I+1"

  while [ $J -lt $NUMV ]
  do
    if [ ${CHECKLIST[J]} -ne -1 ]
    then
      if [ `psrinfo -p ${VLIST[I]} ${VLIST[J]}` = 1 ]
      then
       SAMELIST="$SAMELIST $J"
       let "CHECKLIST[J]=-1"
       let "COUNT=COUNT+1"
      fi
    fi
    let "J=J+1"
  done

  if [ $COUNT -gt 0 ]
  then
    echo "The following logical IDs belong to the same physical
processor:"
    echo "$SAMELIST"
    echo " "
  fi

  let "I=I+1"
done
```

解释为 SUNW_MP_PROCBIND 指定的值：

如果为 **SUNW_MP_PROCBIND** 指定的值是非负整数，则该整数表示线程应该绑定到的虚拟处理器起始逻辑 ID。线程会从有指定逻辑 ID 的处理器开始，以循环的方式绑定到虚拟处理器，在绑定到逻辑 ID 为 *n*-1 的处理器之后，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多个非负整数的列表，则线程将以循环的方式绑定到有指定逻辑 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** 指定的值是两个用减号 ("-") 分隔的非负整数，则线程将以循环的方式绑定到虚拟处理器，处理器位于以第一个逻辑 ID 开头，并以第二个逻辑 ID 结尾的范围内。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形式之一，或者提供了无效的逻辑 ID，系统将发出错误消息，并终止程序的执行。

请注意，微任务化库 libmtsk 创建的线程数取决于环境变量、用户程序中的 API 调用和 **num_threads** 子句。 **SUNW_MP_PROCBIND** 指定线程应该绑定到的虚拟处理器的逻辑 ID。线程将以循环的方式绑定到该处理器集。如果程序中使用的线程数比 **SUNW_MP_PROCBIND** 指定的逻辑 ID 数少，程序将不使用某些虚拟处理器。如果线程数比 **SUNW_MP_PROCBIND** 指定的逻辑 ID 数多，一些虚拟处理器将绑定多个线程。

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。

主栈的默认大小一般约为 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。（请参见《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，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。此栈与初始（或主）线程栈相似，但归从属线程专有。线程的 **PRIVATE** 数组和变量（线程的局部变量）在线程栈中分配。从属线程的线程栈在 32 位 SPARC V8 和 x86 平台上的默认大小为 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。如果栈的尺寸太小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。

注 – 传统的 Sun 和 Cray 并行化指令现已过时，不再受 Sun Studio 编译器支持。

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]*

`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 指令可带有以下可选限定符子句。

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*`)`（请参见表 6-3）

SCHEDTYPE(*spec*) 子句接受以下调度规范。

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

SCHEDTYPE(spec)	等效的 OpenMP schedule(*spec*) 子句
SCHEDTYPE(STATIC)	schedule(static)
SCHEDTYPE(SELF(*chunksize*))	schedule(dynamic,*chunksize*) 默认 *chunksize* 为 1。
SCHEDTYPE(FACTORING(*m*))	无完全等效指令。
SCHEDTYPE(GSS(*m*))	schedule(guided, *m*) 默认 *m* 为 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默认作用域规则：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，否则默认数据作用域为 **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，请参见第 3 章。
- 由于没有 **DOSERIAL** 指令，因此混合使用自动和显式 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。重新设计使用 Sun 指令的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。

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
GUIDED	schedule(guided, *m*) 默认 *m* 为 1。
SINGLE	schedule(dynamic,1)
CHUNKSIZE(n)	schedule(dynamic,*n*)
NUMCHUNKS(m)	schedule(dynamic,*n/m*) 其中 *n* 为迭代次数

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 *loop*
#pragma MP serial_loop_nested	无完全等效指令。可以使用 #pragma omp master *loopnest*

taskloop Pragma 可带有一个或多个以下可选子句。

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,...*)
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*)（请参见表 6-7）

schedtype(*spec*) 子句接受以下调度规范。

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：默认 *chunksize* 为 1。
SCHEDTYPE(FACTORING(*m*))	无完全等效指令。
SCHEDTYPE(GSS(*m*))	schedule(guided, *m*) 默认 *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，因此混合使用自动和显式 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，因此重新设计使用传统 C 指令的程序的并行操作策略，以利用 OpenMP 的这些功能，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，请参见 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 开发者门户网站，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`** 段、 **`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存在一个隐含的障碍。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。
- 默认情况下，空闲线程将在某一超时期限后进入休眠状态。如果默认超时期限对于应用程序过短，也会导致线程过早或过晚地进入休眠状态。 **`SUNW_MP_THR_IDLE`** 环境变量可以替换默认超时期限，甚至可以一直替换到空闲线程永不进入休眠状态且始终保持活动状态的时刻。

- 尽可能在最高级别并行化，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，而不用共享任务 **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 5-7 页 5.4 节的 “处理器绑定”。
- 尽可能使用 **MASTER**，而不用 **SINGLE**。
  - 将 **MASTER** 指令作为不带隐式 **BARRIER** 的 **IF** 语句来执行：
    IF(omp_get_thread_num() == 0) {...}
  - **SINGLE** 指令的实现方式类似于其他共享任务构造。跟踪哪个线程首先到达 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。
  - 由于 **DYNAMIC**、**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，数据需要从专用存储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()** 具有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。用 **-lmtmalloc** 进行链接以使用 **libmtmalloc**。

- 数据量少的情况下，OpenMP 并行循环可能运行不佳。使用 **PARALLEL** 构造上的 **IF** 子句，以指示循环仅应在那些可以预期某些性能增益的情况下运行并行。
- 如有可能，请合并循环。例如：

```
将以下两个循环
!$omp parallel do
  do i = ...
         statements_1
  end do
!$omp parallel do
  do i = ...
         statements_2
  end do

合并为一个循环
!$omp parallel do
  do i = 1,1024
     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 嵌套并行操作的详细信息，请参见第 2 章。

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 处理器）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。

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或其他位置提取该代码行的较新

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**。如果频繁出现伪共享， OpenMP 应用程序的性能和可伸缩性就会显著下降。

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

- 尽可能多地使用专用数据；
- 利用编译器的优化功能来消除内存加载和存储。

在特定情况下，当处理较大的问题时由于存在较少共享，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，数据分配方式的更改可能减少伪共享。在其他情况下，更改迭代到线程的映射，赋予每个块的每个线程更多的工作（通过更改 *chunksize* 值），也可能减少伪共享。

7.3 操作系统优化功能

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)。

操作系统可以利用 MPO 功能分配页，这些页位于访问它们的处理器附近。SunFire E20K 和 SunFire E25K 系统在相同的 UniBoard™ 中及不同的 UniBoard 之间有不同的内存延时。称为**初次接触**的默认 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。

从 Solaris 9 操作系统发行版本开始，系统支持 MPSS 功能，且程序可以利用该功能对虚拟内存的不同区域使用不同的页面大小。默认的 Solaris 页面大小相对较小（在 UltraSPARC 处理器上为 8KB，在 AMD64 Opteron 处理器上为 4KB）。使用较大的页面大小，可以使缺少大量 TLB 的应用程序的性能得以提高。

使用 Sun 性能分析器可以测量 TLB 缺少。

使用以下 Solaris 操作系统命令可以获得特定平台上的默认页面大小：
/usr/bin/pagesize。此命令中的 **-a** 选项列出了所有受支持的页面大小。（有关详细信息，请参见 pagesize(1) 手册页。）

更改应用程序的默认页面大小有以下三种方法：

- 使用 Solaris 操作系统命令 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 特定的环境变量。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

下表将子句与指令和 Pragma 联系起来：

表 A-1 拥有子句的 Pragma

子句 /Pragma	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS	PARALLEL WORKSHARE[3]
IF	•				•	•	•
PRIVATE	•	•	•	•	•	•	•
SHARED	•				•	•	•
FIRSTPRIVATE	•	•	•	•	•	•	•
LASTPRIVATE		•	•		•	•	
DEFAULT	•				•	•	•
REDUCTION	•	•	•		•	•	•
COPYIN	•				•	•	•
COPYPRIVATE				•[1]			
ORDERED		•			•		
SCHEDULE		•			•		
NOWAIT		•[2]	•[2]	•[2]			
NUM_THREADS	•				•	•	•
__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

A

__AUTO, 3-1

B

变量的作用域
- 编译器注释，3-4
- 规则，3-2
- 自动，3-1
- 自动确定作用域的限制，3-8

编译器，访问，-xi

并行操作，嵌套，2-1

D

调度，4-2
- **OMP_SCHEDULE**, 5-4

调度子句
- **SCHEDULE**, 4-2

动态线程，4-2

动态线程调整，5-4

G

guided 调度，5-6

guided 加权，4-2

H

缓冲代码行，7-4

环境变量，5-4

J

加权因子，4-2, 5-6

警告消息，5-5

K

可伸缩性，7-4

空闲线程，5-5

M

MANPATH 环境变量，设置，-xii

N

内存定位优化 (MPO), 7-5

O

OMP_DYNAMIC, 5-4

OMP_NESTED, 2-2, 5-4

OMP_NUM_THREADS, 5-4

OMP_SCHEDULE, 5-4

OpenMP API 规范，1-1

OpenMP 编译，5-1

P

PATH 环境变量，设置，-xi

Pragma
- 请参见指令

平台，受支持的，-x

Q

嵌套并行操作，2-1, 2-2, 4-2, 5-4

S

Shell 提示符，-x
SLEEP，5-5
Solaris 操作系统优化，7-5
SPIN，5-5
STACKSIZE，5-6
-stackvar，5-10
SUNW_MP_GUIDED_WEIGHT，4-3，5-6
SUNW_MP_MAX_NESTED_LEVELS，2-4，5-6
SUNW_MP_MAX_POOL_THREADS，2-3，5-5
sunw_mp_misc.h，4-4
SUNW_MP_PROCBIND，5-5
sunw_mp_register_warn()，4-4
SUNW_MP_THR_IDLE，5-5
SUNW_MP_WARN，4-3，5-5
实现，4-1
手册页，访问，-xi

W

伪共享，7-4
文档，访问，-xiii 至 -xv
文档索引，-xiii

X

-XlistMP，5-3
-xopenmp，5-1
线程数，4-1
  OMP_NUM_THREADS，5-4
线程栈大小，5-6
显式线程程序，4-3
性能，7-1

Y

易读文档，-xiv
印刷约定，-ix
运行时检查，4-3

Z

栈，5-10
栈大小，5-6，5-10
支持的平台，-x
指令
  请参见 Pragma
  验证 (Fortran 95)，5-3
指令验证 (Fortran 95)，5-3
转换为 OpenMP
  Cray 风格的 Fortran 指令，6-3
  Sun 风格的 Fortran 指令，6-2
  传统 C pragma，6-4
自动确定作用域，3-1
自动确定作用域规则，3-3