OpenMP API 用户指南

Sun™ Studio 8

Sun Microsystems, Inc.
www.sun.com

部件号 817-5814-10
2004 年 4 月，修订版 A

请将关于本文档的意见发送至：http://www.sun.com/hwdocs/feedback

请回收

Adobe PostScript

目录

表

开始之前

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

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

印刷惯例

表 P-1 字样惯例

字样	含义	示例
`AaBbCc123`	命令、文件及目录的名称；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`	键入的内容，用于与计算机屏幕输出相对比	`% su` `Password:`
AaBbCc123	书名、新词或术语、要强调的词语	参阅 *《用户指南》*第 6 章。 这些称为*类*选项。 您*必须*是超级用户才能执行此项操作。
AaBbCc123	命令行占位文字；用实际名称或值替换	要删除文件，请键入 **`rm`** *filename*。

表 **P-2** 代码惯例

代码符号	含义	表示法	代码示例
[ ]	方括号中的参数为可选参数。	O[*n*]	-O4、O
{ }	大括号中包含必需选项的选择集。	d{y|n}	-dy
|	“管道”或“竖线”符号用于分隔参数，只能选择其中之一。	B{dynamic|static}	-Bstatic
:	与逗号类似，冒号有时用于分隔参数。	R*dir*[:*dir*]	-R/local/libs:/U/a
…	省略号指略去了一个系列项中的某些项。	-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 的超级用户	#

访问 Sun Studio 软件和手册页

编译器和工具及它们的手册页并未安装到标准的 /usr/bin/ 和 /usr/share/man 目录中。要访问这些编译器和工具，必须正确设置 PATH 环境变量 （参见第 xi 页中的“访问编译器和工具”）。要访问手册页，必须正确设置 MANPATH 环境变量 （参见第 xii 页中的 “访问手册页”）。

有关 PATH 变量的详细信息，参见 csh(1)、sh(1) 和 ksh(1) 手册页。有关 MANPATH 变量的详细信息，参见 man(1) 手册页。有关设置 PATH 变量和 MANPATH 变量以访问本版本的详细信息，参见安装指南或咨询系统管理员。

注意 – 本部分中的信息假定您的 Sun Studio 编译器和工具安装在 /opt 目录中。如果软件未安装在 /opt 目录中，请向系统管理员询问您系统中的相当路径。

访问编译器和工具

采用以下步骤确定是否需要更改 PATH 变量才能访问编译器和工具。

▼ 确定是否需要设置 PATH 环境变量

1. **在命令提示符处键入以下命令，显示 PATH 变量的当前值。**

```
% echo $PATH
```

2. **查看输出结果，查找包含 /opt/SUNWspro/bin/ 的路径字符串。**

   如果找到了该路径，表明 PATH 变量已设置为可以访问编译器和工具。如果未找到该路径，请按照下一过程中的说明设置 PATH 环境变量。

▼ 设置 PATH 环境变量以便能够对编译器和工具进行访问

1. **如果使用 C shell，请编辑您起始目录下的 .cshrc 文件。如果使用 Bourne shell 或 Korn shell，请编辑您起始目录下的 .profile 文件。**

2. **将以下路径添加至 PATH 环境变量。如果您已安装了 Sun ONE Studio 软件或 Forte Developer 软件，请将以下路径添加到这些安装软件的路径之前。**

   /opt/SUNWspro/bin

访问手册页

使用下列步骤确定是否需要更改 MANPATH 变量才能访问手册页。

▼ 确定是否需要设置 MANPATH 环境变量

1. **在命令提示符处键入以下命令来请求 dbx 手册页。**

```
% man dbx
```

2. **查看输出结果 （如果有）。**

   如果无法找到 dbx(1) 手册页，或者所显示的手册页并非对应于所安装软件的当前版本，请按照下一过程中的说明设置 MANPATH 环境变量。

▼ 设置 MANPATH 环境变量以便能够对手册页进行访问

1. **如果使用 C shell，请编辑您起始目录下的 .cshrc 文件。如果使用 Bourne shell 或 Korn shell，请编辑您起始目录下的 .profile 文件。**

2. **将以下路径添加至 MANPATH 环境变量。**

   /opt/SUNWspro/man

访问集成开发环境

Sun Studio 8 集成开发环境 (IDE) 提供了用于创建、编辑、生成、调试以及分析 C、C++ 或 Fortran 应用程序性能的模块。

此 IDE 需要 Sun Studio 8 的核心平台组件。如果核心平台组件没有安装在下列位置之一，必须将 SPRO_NETBEANS_HOME 环境变量设置为核心平台组件的安装位置 (*installation_directory*/netbeans/3.5R)：

- 缺省安装目录 /opt/netbeans/3.5R

- 与 Sun Studio 8 编译器和工具组件相同的位置 （例如，编译器和工具组件安装在 `/foo/SUNWspro`，核心平台组件安装在 `/foo/netbeans/3.5R`）

用于启动 IDE 的命令是 `sunstudio`。有关此命令的详细信息，参见 `sunstudio`(1) 手册页。

访问编译器和工具文档

可在下列位置访问文档：

- 文档可从随软件一同安装在本地系统或网络上的文档索引中获得，位置在 `file:/opt/SUNWspro/docs/index.html`。

  如果软件未安装在 `/opt` 目录中，请向系统管理员询问您系统中的相当路径。

- 大多数手册都可以从 `docs.sun.com`sm 网站获得。下列书目只能通过所安装的软件得到：
  - *《标准 C++ 库类参考》*
  - *《标准 C++ 库用户指南》*
  - *《Tools.h++ 类库参考》*
  - *《Tools.h++ 用户指南》*
- 发行说明可从 `docs.sun.com` 网站获得。
- 可通过 [帮助] 菜单获得 IDE 所有组件的联机帮助，也可通过 IDE 许多窗口和对话框中的 [帮助] 按钮来获得。

在 `docs.sun.com` 网站 (`http://docs.sun.com`) 上，您可以通过因特网阅读、打印和购买 Sun Microsystems 的手册。如果找不到手册，参见随软件一同安装在本地系统或网络上的文档索引。

注意 – Sun 对本文档中提及的第三方网站的可用性概不负责，并且不认可也不对此类站点或资源上或通过其获得的任何内容、广告、产品或其它资料负任何责任或义务。对于因使用或信赖此类站点或资源中提供或通过其提供的任何此类内容、商品或服务而导致或声称导致或与之有关的任何损害或损失， Sun 将不负任何责任或义务。

易访问格式文档

我们还提供了易访问格式的文档，便于残疾用户通过辅助技术阅读。您可以按下表所述找到文档的易访问版本。如果软件未安装在 /opt 目录中，请向系统管理员询问您系统中的相当路径。

文档类型	易访问版本的格式和位置
手册 （不包括第三方手册）	HTML 格式，位于 http://docs.sun.com
第三方手册： • *《标准 C++ 库类参考》* • *《标准 C++ 库用户指南》* • *《Tools.h++ 类库参考》* • *《Tools.h++ 用户指南》*	HTML 格式，在已安装软件中通过文档索引访问，位置在 file:/opt/SUNWspro/docs/index.html
自述文件和手册页	HTML 格式，在已安装软件中通过文档索引访问，位置在 file:/opt/SUNWspro/docs/index.html
联机帮助	HTML 格式，可通过 IDE 的 [帮助] 菜单获得
发行说明	HTML 格式，位于 http://docs.sun.com

相关编译器和工具文档

下表介绍了可在 file:/opt/SUNWspro/docs/index.html 和 http://docs.sun.com 上获得的相关文档。如果软件未安装在 /opt 目录中，请向系统管理员询问您系统中的相当路径。

文档名称	说明
《Fortran 编程指南》	介绍如何在 Solaris 环境中编写高效的 Fortran 代码；输入 / 输出、库、性能、调试和并行处理。
《Fortran 库参考》	详细介绍 Fortran 库和内在例程
《Fortran 用户指南》	介绍 f95 编译器的编译时环境以及命令行选项。另外还包括传统 f77 程序到 f95 的迁移指导。

文档名称	说明
《C 用户指南》	介绍 cc 编译器的编译时环境以及命令行选项。
《C++ 用户指南》	介绍 CC 编译器的编译时环境以及命令行选项。
《数值计算指南》	说明浮点计算数值精度涉及到的问题。

访问 Solaris 相关文档

下表介绍可通过 docs.sun.com 网站获得的相关文档。

文档集	文档名称	说明
Solaris 参考手册集	参见手册页各部分的书名。	提供 Solaris 操作环境的有关信息。
Solaris 软件开发人员集	*《链接程序和库指南》*	介绍 Solaris 链接编辑器及运行时链接程序的操作。
Solaris 软件开发人员集	*《多线程编程指南》*	内容包括 POSIX 和 Solaris 线程 API、使用同步对象编程、编译多线程程序以及查找多线程程序工具。

开发人员资源

访问 http://developers.sun.com/prodtech/cc 可找到以下经常更新的资源：

- 介绍编程技术和最佳办法的文章
- 内含简短编程技巧的知识库

- 编译器和工具组件文档，以及对随软件安装文档的更正
- 支持级别信息
- 用户论坛
- 可下载代码范例
- 新技术预见

还可以在 `http://developers.sun.com` 上找到其它开发人员资源。

联系 Sun 技术支持

如果您对于本产品有任何技术问题在本文档中找不到答案，请访问：

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

发送意见

Sun 一直致力于提高其文档质量，欢迎您提出意见和建议。请将您的意见通过电子邮件发送给 Sun，邮件地址：

`docfeedback@sun.com`

请在您电子邮件的主题行中加入文档的部件号 (817-5814-10)。

第 1 章

OpenMP API 概述

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

OpenMP API 是 SPARC® 和 UltraSPARC® 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，参见第 4 章。

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

1.1 哪里有 OpenMP 规范

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

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

1.2 本章所使用的特殊惯例

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

*结构化块*指无进或出传输的 Fortran 或 C/C++ 语句块。

方括号 [...] 内的构造为可选构造。

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

本手册中，“指令”和 “pragma”互换使用。

1.3 指令格式

每个指令行中只能指定一个*指令名*，且该指令名应用于随后的程序语句。

Fortran：

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

C/C++：

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

OpenMP 2.0 Fortran

固定格式：

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

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

自由格式：

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

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

OpenMP 2.0 C/C++

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

1.4 条件编译

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

OpenMP 2.0 Fortran

固定格式：

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

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

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

自由格式：

```
!$ fortran_95_statement
```

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

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

Fortran 预处理程序：

启用 OpenMP 时编译会定义预处理程序符号 **_OPENMP**。

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

OpenMP 2.0 C/C++

C/C++ 预处理程序：

启用 OpenMP 时编译会定义宏 **_OPENMP**。

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

1.5 PARALLEL - 并行区域构造

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

OpenMP 2.0 Fortran

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

OpenMP 2.0 C/C++

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

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

表 1-1 列出了可随此构造出现的子句。

1.6 工作共享构造

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

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

1.6.1 **DO** 和 **for** 构造

指定随后的 **DO** 或 **for** 循环的迭代必须以并行方式执行。

OpenMP 2.0 Fortran

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

DO 指令指定其后紧跟的 **DO** 循环的迭代应以并行方式执行。此指令必须出现在并行区域内才有效。

OpenMP 2.0 C/C++

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

for pragma 指定其后紧跟的 *for- 循环*的迭代应以并行方式执行。此 pragma 必须出现在并行区域内才有效。 **for** pragma 对相应 **for** 循环的结构有限制，且必须有*规范形状*：

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

其中：

- *initexpr* 为以下之一：

  var = lb
  integer_type var = lb

- *increxpr* 为以下表达式形式之一：

  ++var
  var++
  --var
  var--
  var += incr
  var -= incr
  var = var + incr
  var = incr + var
  var = var - incr

- *var* 是有符号整型变量，被隐式设置为供 **for** 范围专用。切勿修改 **for** 语句体内的 *var*。除非指定 **lastprivate**，否则其值在循环后不确定。

- *logicop* 为以下逻辑操作符之一：

  `<  <=  >  >=`

- *lb*、 *b* 和 *incr* 是循环不变量整型表达式。

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

1.6.2 **SECTIONS** 构造

SECTIONS 构造用于封装要在组内的线程中分配的非迭代代码块。每个块由组内的线程执行一次。

每段均以 **SECTION** 指令开头，该指令对第一段为可选指令。

OpenMP 2.0 Fortran

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

OpenMP 2.0 C/C++

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

表 1-1 列出了可随此构造出现的子句。

1.6.3 **SINGLE** 构造

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

OpenMP 2.0 Fortran

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

OpenMP 2.0 C/C++

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

表 1-1 列出了可随此构造出现的子句。

1.6.4 Fortran **WORKSHARE** 构造

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

OpenMP 2.0 Fortran

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

没有与 Fortran WORKSHARE 构造等价的 C/C++ 指令。

1.7 合并的并行工作共享构造

合并的并行工作共享构造是指定包含一个工作共享构造的并行区域的捷径。

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

表 1-1 列出了可随这些构造出现的子句。

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

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

OpenMP 2.0 Fortran

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

OpenMP 2.0 C/C++

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

1.7.2 **PARALLEL SECTIONS** 构造

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

OpenMP 2.0 Fortran

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

OpenMP 2.0 C/C++

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

1.7.3 **PARALLEL WORKSHARE** 构造

Fortran **PARALLEL WORKSHARE** 构造为指定包含单个 **WORKSHARE** 指令的并行区域提供了一种捷径。 *clause* 可以是 **PARALLEL** 或 **WORKSHARE** 指令可接受的任意子句。

OpenMP 2.0 Fortran

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

没有等价的 C/C++ 指令。

1.8 同步构造

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

1.8.1 **MASTER** 构造

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

OpenMP 2.0 Fortran

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

OpenMP 2.0 C/C++

```
#pragma omp master
     结构化块
```

1.8.2 **CRITICAL** 构造

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

OpenMP 2.0 Fortran

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

OpenMP 2.0 C/C++

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

1.8.3 **BARRIER** 构造

同步组内的所有线程。每个线程都等到组内所有其它线程都到达此点为止。

OpenMP 2.0 Fortran

```
!$OMP BARRIER
```

OpenMP 2.0 C/C++

```
#pragma omp barrier
```

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

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

1.8.4 **ATOMIC** 构造

确保特定内存位置自动更新，而不要将其交由不确定的多个同时写入线程来支配。

此实现在临界区域中封装表达式语句来替换所有 ATOMIC 指令。

OpenMP 2.0 Fortran

```
!$OMP ATOMIC
```
表达式语句

该指令只应用于随后紧跟的语句，且语句必须采用以下这些形式之一：

x = x operator expression

x = expression operator x

x = intrinsic(x, expr-list)

x = intrinsic(expr-list, x)

其中：

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

OpenMP 2.0 C/C++

```
#pragma omp atomic
```
表达式语句

该 pragma 只应用于随后紧跟的语句，且语句必须采用以下这些形式之一：

x binop = expr

x++

++x

x--

--x

其中：

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

1.8.5 **FLUSH** 构造

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

OpenMP 2.0 Fortran

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

OpenMP 2.0 C/C++

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

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

1.8.6 **ORDERED** 构造

封装的块按迭代在循环的顺序执行中的执行顺序执行。

OpenMP 2.0 Fortran

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

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

OpenMP 2.0 C/C++

```
#pragma omp ordered
    结构化块
```

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

1.9 数据环境指令

以下指令用于在并行构造执行期间控制数据环境。

1.9.1 THREADPRIVATE 指令

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

有关完整细节和限制，参见 OpenMP 规范（Fortran 规范中的 2.6.1 节； C/C++ 规范中的 2.7.1 节）。

OpenMP 2.0 Fortran

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

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

OpenMP 2.0 C/C++

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

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

1.10 OpenMP 指令子句

本节概述可以出现在 OpenMP 指令中的数据作用域和调度子句。

1.10.1 数据作用域子句

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

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

*对这些作用域子句的使用有一些重要的限制。*有关完整细节，请参阅 Fortran 规范的 2.6.2 节和 C/C++ 规范中的 2.7.2 节。

表 1-1 列出可出现这些子句的指令。

1.10.1.1 **PRIVATE** 子句

private(*list***)**

将可选的以逗号分隔的 *list* 中的变量声明为供组内各线程专用。

1.10.1.2 **SHARED** 子句

shared(*list***)**

组内所有线程都共享 *list* 中出现的变量，并访问同一存储区域。

1.10.1.3 **DEFAULT** 子句

Fortran

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

C/C++

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

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

1.10.1.4 **FIRSTPRIVATE** 子句

firstprivate(*list*)

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

1.10.1.5 **LASTPRIVATE** 子句

lastprivate(*list*)

列表中的变量为 **PRIVATE**。此外， **LASTPRIVATE** 子句在 **DO** 或 **for** 指令中出现时，执行序列末位的最后一个迭代的线程会在构造前更新变量的版本。在 **SECTIONS** 指令中出现时，执行按词汇顺序位于最后的 **SECTION** 的线程会在构造前更新对象的版本。

1.10.1.6 **COPYIN** 子句

Fortran

COPYIN(*list*)

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

C/C++

copyin(*list*)

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

1.10.1.7 **COPYPRIVATE** 子句

Fortran

COPYPRIVATE(*list*)

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

C/C++

copyprivate(*list*)

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

1.10.1.8 **REDUCTION** 子句

Fortran

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

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

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

list 中的变量必须为内在类型的命名变量。

C/C++

reduction(*operator*:*list*)

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

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

有关完整细节和对 **REDUCTION** 子句和构造的限制，*参见* Fortran OpenMP 规范中的 2.6.2.6 节和 C/C++ 规范中的 2.7.2.6 节。

1.10.2 调度子句

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

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

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

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

1.10.2.1 **STATIC** 调度

schedule(static[,*chunk*])

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

1.10.2.2 **DYNAMIC** 调度

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

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

1.10.2.3 **GUIDED** 调度

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

使用 **GUIDED** 时，每分发一个迭代块，块大小便以指数方式递减一次。*chunk* 指定每次分发的最小迭代数。（初始迭代块的大小要依赖实现，参见 “第 2 章”）。未指定 *chunk* 时，缺省值为 1。

1.10.2.4 **RUNTIME** 调度

schedule(runtime)

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

1.10.3 **NUM_THREADS** 子句

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

num_threads(*scalar_integer_expression***)**

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

请注意， **num_threads** 不应用于后续区域。

1.10.4 子句在指令中的放置

表 1-1 显示可以在这些指令和 pragma 中出现的子句：

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

表 1-1 可以有子句的 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	•				•	•	•

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

1.11 OpenMP 运行时库例程

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

1.11.1 Fortran OpenMP 例程

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

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

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

使用 **-Xlist** 编译将报告所有类型不匹配情况。

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

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

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

1.11.2 C/C++ OpenMP 例程

C/C++ 运行时库函数是外部函数。

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

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

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

1.11.3 运行时线程管理例程

有关详细信息，请参阅相应的 OpenMP 规范。

1.11.3.1 OMP_SET_NUM_THREADS 例程

设置供后续并行区域使用的线程数

Fortran

```
SUBROUTINE OMP_SET_NUM_THREADS(int_expr)
```

C/C++

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

1.11.3.2 OMP_GET_NUM_THREADS 例程

返回当前组中正在执行从中调用其的并行区域的线程的数量。

Fortran

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

C/C++

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

1.11.3.3 OMP_GET_MAX_THREADS 例程

返回调用 OMP_GET_NUM_THREADS 函数所返回的最大值。

Fortran

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

C/C++

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

1.11.3.4 **OMP_GET_THREAD_NUM** 例程

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

Fortran

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

C/C++

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

1.11.3.5 **OMP_GET_NUM_PROCS** 例程

返回程序可用的处理器数。

Fortran

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

C/C++

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

1.11.3.6 **OMP_IN_PARALLEL** 例程

确定线程是否在并行区域的动态范围内执行。

Fortran

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

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

C/C++

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

如果在并行区域的动态范围内调用，则返回非零值；否则，返回零值。

1.11.3.7 OMP_SET_DYNAMIC 例程

启用或禁用可用线程数的动态调整。（缺省情况下启用动态调整。）

Fortran

```
SUBROUTINE OMP_SET_DYNAMIC(logical_expr)
```

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

C/C++

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

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

1.11.3.8 OMP_GET_DYNAMIC 例程

确定是否启用了动态线程调整。

Fortran

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

启用了动态线程调整时返回 .TRUE.；否则，返回 .FALSE.。

C/C++

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

启用了动态线程调整时返回非零值；否则，返回零值。

1.11.3.9 OMP_SET_NESTED 例程

启用或禁用嵌套并行操作。（*不支持嵌套并行操作，缺省情况下禁用。*）

Fortran

```
SUBROUTINE OMP_SET_NESTED(logical_expr)
```

C/C++

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

1.11.3.10 **OMP_GET_NESTED** 例程

确定是否启用了嵌套并行操作。（*不支持嵌套并行操作，缺省情况下禁用。*）

Fortran

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

返回 **.FALSE.**。*不支持嵌套并行操作。*

C/C++

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

返回零值。*不支持嵌套并行操作。*

1.11.4 管理同步锁定的例程

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

Fortran：

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

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

C/C++：

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

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

1.11.4.1 **OMP_INIT_LOCK** 和 **OMP_INIT_NEST_LOCK** 例程

为后续调用初始化锁定变量。

Fortran

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

C/C++

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

1.11.4.2 OMP_DESTROY_LOCK 和 OMP_DESTROY_NEST_LOCK 例程

删除锁定变量。

Fortran

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

C/C++

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

1.11.4.3 OMP_SET_LOCK 和 OMP_SET_NEST_LOCK 例程

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

Fortran

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

C/C++

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

1.11.4.4 OMP_UNSET_LOCK 和 OMP_UNSET_NEST_LOCK 例程

释放正在执行的线程对锁定的所有权。线程不拥有该锁定时，行为不确定。

Fortran

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

C/C++

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

1.11.4.5 OMP_TEST_LOCK 和 OMP_TEST_NEST_LOCK 例程

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

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

Fortran

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

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

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

锁定设置成功时返回嵌套计数；否则，返回零。

C/C++

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

锁定设置成功时返回非零值；否则，返回零值。

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

锁定设置成功时返回锁定嵌套计数；否则，返回零。

1.11.5 计时例程

两个函数支持可移植挂钟计时器。

1.11.5.1 OMP_GET_WTIME 例程

返回挂钟 “自过去任意时刻以来”经过的时间 （秒）。

Fortran

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

C/C++

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

1.11.5.2 OMP_GET_WTICK 例程

返回连续时钟滴答声之间的秒数。

Fortran

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

C/C++

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

第 2 章

依赖实现问题

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

调度

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

线程数

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

动态线程

- 如果没有对 **omp_set_dynamic()** 函数的显式调用或 **OMP_DYNAMIC** 环境变量的显式定义，缺省值是启用动态线程调整。启用动态线程调整后，线程数被限定为可用处理器数。

嵌套并行操作

- 此实现中不支持嵌套并行操作，缺省情况下会禁用嵌套并行操作。嵌套并行区域只由单个线程来执行。

ATOMIC 指令

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

GUIDED 初始和最小块大小

- **SCHEDULE(GUIDED,** *chunk***)** 的缺省最小块大小为 1。缺省初始块大小为循环内的迭代数与执行循环的线程数之商。

显式线程程序

- 使用线程的程序可以调用包含 OpenMP 指令的例程。

第 3 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。设置的目的是告知运行时系统程序可以创建的最大线程数。缺省值为 1。一般将 **OMP_NUM_THREADS** 设置为目标平台上可用的处理器数。

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

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

3.1 要使用的编译器选项

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

-xopenmp 标志接受下列关键字子选项。

-xopenmp=parallel	启用对 OpenMP pragma 的识别。 -xopenmp=parallel 的最低优化级别是 -xO3。必要时，编译器可把优化从较低级别更改为 -xO3，并发出警告。
-xopenmp=noopt	启用对 OpenMP pragma 的识别。如果级别低于 -xO3，编译器不会提高级别。 如果将优化级别显式地设置为低于 -xO3 的级别，如 -xO2 -openmp=noopt，编译器会报告错误。 如果未使用 -openmp=noopt 指定优化级别，系统会识别 OpenMP pragma，程序会被相应并行化，但不会进行优化。 （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	禁用对 OpenMP pragma 的识别、链接到桩模块库例程，且不更改优化级别。如果应用程序显式地调用 OpenMP 运行时库例程，并要编译程序使其顺次执行，请使用此选项。
-xopenmp=none	禁用对 OpenMP pragma 的识别，不更改优化级别。（缺省值）

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。

使用 **parallel**、 **noopt** 或 **stubs** 指定 **-xopenmp=** 将把 **_OPENMP** 预处理程序标记定义为 YYYYMM 格式（具体地讲， C/C++ 定义为 **200203L**， Fortran 95 定义为 **200011**）。

使用 dbx 调试 OpenMP 程序时，请使用 **-xopenmp=noopt -g** 进行编译

-xopenmp 的缺省优化级别在未来版本中可能会发生变化。显式地指定适当的优化级别可避免出现警告消息。

如果是 Fortran 95， **-xopenmp**、 **-xopenmp=parallel**、 **-xopenmp=noopt** 会自动添加 **-stackvar**。

3.2 Fortran 95 OpenMP 验证

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

- 并行指令规范中的违例，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`：

```
文件 "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  !$OMP ORDERED
          ^
**** ERR-OMP: ORDERED 指令绑定到
没有指定 ORDERED 子句的指令（ord.f 第 10 行，第 2 列）
上是非法的。
    19                    write(*,*) k
    20  !$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。

3.3 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。

表 3-1 OpenMP 环境变量

环境变量	功能
OMP_SCHEDULE	为指定了 **RUNTIME** 调度类型的 **DO**、 **PARALLEL DO**、 **parallel for**、 **for** 及指令 /pragma 设置调度类型。未定义时使用缺省值 **STATIC**。*值*为 *"type[,chunk]"* 示例：`setenv OMP_SCHEDULE "GUIDED,4"`
OMP_NUM_THREADS *或* **PARALLEL**	如果未使用 **NUM_THREADS** 子句或调用 **OMP_SET_NUM_THREADS()** 进行设置，则设置执行期间使用的线程数。未设置时使用缺省值 1。*值*为正整数。（*当前最大值为 128*）。为与传统程序兼容，设置 **PARALLEL** 环境变量和设置 **OMP_NUM_THREADS** 环境变量的效果相同。但如果将这两个环境变量都设置为不同的值，运行时库会发出一个错误消息。 示例：`setenv OMP_NUM_THREADS 16`
OMP_DYNAMIC	为并行区域的执行启用或禁用对可用线程数的动态调整。未设置时使用缺省值 **TRUE**。*值*为 **TRUE** 或 **FALSE**。 示例：`setenv OMP_DYNAMIC FALSE`
OMP_NESTED	启用或禁用嵌套并行操作。（*不支持嵌套并行操作*。） *值*为 **TRUE** 或 **FALSE**。（*此变量无效*。） 示例：`setenv OMP_NESTED FALSE`

尽管其它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 3-2　多重处理环境变量

环境变量	功能
SUNW_MP_WARN	控制 OpenMP 运行时库发出的警告消息。设置为 **TRUE** 时，运行时库会给 stderr 发出警告消息；设置为 **FALSE** 时禁用警告消息。缺省值为 **FALSE**。 示例： **setenv SUNW_MP_WARN FALSE**
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 **sleep** *n***ms**。缺省值为 **SPIN** — 完成并行任务后线程会旋转 （或占线等待），直到新并行任务到达为止。 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。如果在线程旋转期间有新任务到达，线程便会立即执行新任务。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)**
STACKSIZE	设置每个线程的栈大小。值以千字节为单位。 缺省线程栈大小在 32 位 SPARC V8 平台上为 4 兆字节；在 64 位 SPARC V9 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*

3.4 栈和栈大小

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

主栈的缺省大小一般约为 8 兆字节。使用 `f95` **`-stackvar`** 选项编译 Fortran 程序会强制将局部变量和数组像自动变量一样在栈中分配。显式并行化的程序暗指对 OpenMP 程序使用 **`-stackvar`**，因为该选项会提高优化器在循环中并行化调用的能力。（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，这样会导致栈溢出。

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

多线程程序的每个帮助程序线程都有自己的线程栈。此栈模拟初始（或主）线程栈，但归线程专有。线程的 **`PRIVATE`** 数组和变量（线程的局部变量）在线程栈中分配。在 32 位系统上的缺省大小为 4 兆字节；在 64 位系统上为 8 兆字节。帮助程序线程栈的大小使用 **`STACKSIZE`** 环境变量来设置。

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

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

可能需要反复试验才能确定最佳栈大小。如果栈小到不足以满足线程的运行需要，可能会导致邻近线程中发生静态数据损坏或段故障。如果无法确定是否有栈溢出，请使用 **`-xcheck=stkovf`** 标志编译 Fortran 或 C 程序来强制于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 4 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。

4.1 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。

4.1.1 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价的 OpenMP 指令和子子句。这些只是建议值。

表 4-1 将 Sun 并行化指令转换为 OpenMP

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

`C$PAR DOSERIAL`	无完全等价指令。可以使用：
	`!$omp master`
	loop
	`!$omp end master`

`C$PAR DOSERIAL*`	无完全等价指令。可以使用：
	`!$omp master`
	loopnest
	`!$omp end master`

`C$PAR TASKCOMMON` *block[,...]*	`!$omp threadprivate (/block/[,...])`

DOALL 指令可带有下列可选限定符子句。

表 4-2　DOALL 限定符子句和等价的 OpenMP 子句

Sun DOALL 子句	等价的 OpenMP PARALLEL DO 子句
PRIVATE(*v1,v2,...*)	private(*v1,v2,...*)
SHARED(*v1,v2,...*)	shared(*v1,v2,...*)
MAXCPUS(*n*)	num_threads(*n*)。无完全等价指令。
READONLY(*v1,v2,...*)	无完全等价指令。使用 firstprivate(*v1,v2,...*) 可以获得相同效果。
STOREBACK(*v1,v2,...*)	无完全等价指令。使用 lastprivate(*v1,v2,...*) 可以获得相同效果。
SAVELAST	无完全等价指令。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*)（参见表 4-3）

SCHEDTYPE(*spec*) 子句接受下列调度规范。

表 4-3　SCHEDTYPE 调度和等价的 OpenMP schedule

SCHEDTYPE(spec)	等价的 OpenMP schedule(*spec*) 子句
SCHEDTYPE(STATIC)	schedule(static)
SCHEDTYPE(SELF(*chunksize*))	schedule(dynamic,*chunksize*) 缺省 *chunksize* 为 1。
SCHEDTYPE(FACTORING(*m*))	无等价 OpenMP 指令。
SCHEDTYPE(GSS(*m*))	schedule(guided, *m*) 缺省 *m* 为 1。

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

- 变量作用域（共享或专用）必须使用 OpenMP 加以显式声明。对于 Sun 指令，编译器对未在 PRIVATE 或 SHARED 子句中显式确定作用域的变量使用其自己的缺省作用域规则：所有标量均按 PRIVATE 处理；所有数组引用均按 SHARED 处理。对于 OpenMP，除非 PARALLEL DO 指令中出现 DEFAULT(PRIVATE) 子句，否则缺省数据作用域为 SHARED。DEFAULT(NONE) 子句会使编译器标记那些未显式确定作用域的变量。
- 由于没有 DOSERIAL 指令，因此混合自动和显式 OpenMP 并行化的结果可能会不同：某些使用 Sun 指令不能自动并行化的循环可能会被自动并行化。

- OpenMP 提供并行区域和并行段来提供更丰富的并行操作模型。使用 Sun 指令来重新设计程序的并行操作策略，以利用 OpenMP 的这些功能，便有可能获得更高的性能。

4.1.2 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 4-4 Cray 风格的 `DOALL` 限定符子句的等价 OpenMP 子句

Cray DOALL 子句	**等价的 OpenMP PARALLEL DO 子句**
`SHARED(`*v1,v2,...*`)`	`SHARED(`*v1,v2,...*`)`
`PRIVATE(`*v1,v2,...*`)`	`PRIVATE(`*v1,v2,...*`)`
`AUTOSCOPE`	无等价子句。作用域必须为显式，或带有 `DEFAULT` 子句。
`SAVELAST`	无完全等价指令。使用 `lastprivate` 可以获得相同效果。
`MAXCPUS(n)`	`num_threads(`*n*`)`。无完全等价指令。
`GUIDED`	`schedule(guided,` *m*`)` 缺省 *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

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

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

4.2 转换传统 C Pragma

C 编译器接受传统 pragma 来进行显示并行化。*《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 4-5　将传统 C 并行化 Pragma 转换为 OpenMP

传统 C Pragma	等价的 OpenMP Pragma
`#pragma MP taskloop` *[clauses]*	`#pragma omp parallel for` *[clauses]*
`#pragma MP serial_loop`	无完全等价指令。可以使用 `#pragma omp master` *loop*
`#pragma MP serial_loop_nested`	无完全等价指令。可以使用 `#pragma omp master` *loopnest*

`taskloop` pragma 可带有一个或多个下列可选子句。

表 4-6　`taskloop` 可选子句和等价的 OpenMP 子句

`taskloop` 子句	等价的 OpenMP `parallel for` 子句
`maxcpus(`*n*`)`	无等价子句。使用 `num_threads(`*n*`)`
`private(`*v1,v2,...*`)`	`private(`*v1,v2,...*`)`
`shared(`*v1,v2,...*`)`	`shared(`*v1,v2,...*`)`
`readonly(`*v1,v2,...*`)`	无完全等价指令。使用 `firstprivate(`*v1,v2,...*`)` 可以获得相同效果。
`storeback(`*v1,v2,...*`)`	无完全等价指令。使用 `lastprivate(`*v1,v2,...*`)` 可以获得相同效果。
`savelast(`*v1,v2,...*`)`	无完全等价指令。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*`)`（参见表 4-7）

`schedtype(`*spec*`)` 子句接受下列调度规范。

表 4-7　`SCHEDTYPE` 调度和等价的 OpenMP `schedule`

schedtype(spec)	等价的 OpenMP schedule(*spec*) 子句
`SCHEDTYPE(STATIC)`	`schedule(static)`

表 4-7　SCHEDTYPE 调度和等价的 OpenMP schedule

schedtype(spec)	等价的 OpenMP schedule(*spec*) 子句
SCHEDTYPE(SELF(*chunksize*))	schedule(dynamic,*chunksize*) 注意：缺省 *chunksize* 为 1。
SCHEDTYPE(FACTORING(*m*))	无等价 OpenMP 指令。
SCHEDTYPE(GSS(*m*))	schedule(guided, *m*) 缺省 *m* 为 1。

4.2.1 传统 C Pragma 与 OpenMP 间的问题

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

索引

P

Q

S

T

W

X

Y

Z