Oracle® Developer Studio 12.5：OpenMP API 用户指南

文件号码 E71962
2016 年 7 月

ORACLE

文件号码 E71962

文档可访问性

有关 Oracle 对可访问性的承诺，请访问 Oracle Accessibility Program 网站 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc。

获得 Oracle 支持

购买了支持服务的 Oracle 客户可通过 My Oracle Support 获得电子支持。有关信息，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info；如果您听力受损，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs。

目录

示例

使用本文档

- **概述** – 介绍 Oracle Developer Studio 12.5 C、C++ 和 Fortran 编译器支持的 OpenMP API 的特性
- **目标读者** – 应用程序开发者、系统开发者、架构师、支持工程师
- **必备知识** – 编程经验、软件开发测试以及构建和编译软件产品的能力

产品文档库

有关该产品及相关产品的文档和资源，可从以下网址获得：http://docs.oracle.com/cd/E60778_01。

反馈

可以在 http://www.oracle.com/goto/docfeedback 上提供有关本文档的反馈。

♦ ♦ ♦ 第 1 章

1 OpenMP API 简介

OpenMP 应用程序接口 (API) 是与许多计算机供应商、学者和研究人员合作开发的一个用于编写多线程程序的可移植并行编程模型。OpenMP 规范由“OpenMP 体系结构审核委员会”创立并公布。

OpenMP API 是所有 Oracle Developer Studio 编译器的推荐并行编程模型。

1.1 支持的 OpenMP 规范

本手册介绍了遵循 OpenMP API 规范版本 4.0（在本手册中称为 OpenMP 4.0）的 Oracle Developer Studio 实现所特有的问题。可以在 OpenMP 官方网站 http://www.openmp.org 找到该规范。

注 - 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版本的 OpenMP 运行时库 `libmtsk.so`。

有关 Oracle Developer Studio 编译器发行版及其 OpenMP API 实现的最新信息可在 Oracle Developer Studio 门户上找到，网址为：http://www.oracle.com/technetwork/server-storage/solarisstudio。

注 - 此发行版的 Oracle Developer Studio 完全支持 OpenMP 4.0 规范。但是，应注意以下几点：

- 接受 SIMD 构造。但是，SIMD 构造不能导致使用任何 SIMD 指令。
- 接受设备构造。但是，所有代码都将在主机设备上执行。唯一可用的设备是主机设备。

1.2 本文档的特殊约定

术语结构化块是指无数据输入或输出的 C、C++ 或 Fortran 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 Developer Studio 编译器 `f95`(1)。

在本手册中，术语“指令”和 "pragma" 互换使用。OpenMP 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。

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` 编译器选项 `-xopenmp` 编译程序。`f95` 编译器将 `-xopenmp` 和 `-openmp` 作为同义词接受。

`-xopenmp` 标志接受下表中列出的关键字子选项。

`-xopenmp=parallel`	启用 OpenMP 指令的识别。 `-xopenmp=parallel` 的最低优化级别是 `-xO3`。 如果优化级别低于 `-xO3`，则编译器会将优化级别提高到 `-xO3` 并发出警告。
`-xopenmp=noopt`	启用 OpenMP 指令的识别。 如果优化级别低于 `-xO3`，则编译器不提升它。 如果使用 `-xopenmp=noopt` 将优化级别显式设置为低于 `-xO3`，如 `-xO2 -xopenmp=noopt`，则编译器会发出错误。 如果没有使用 `-xopenmp=noopt` 指定优化级别，则会识别 OpenMP 指令，并相应地并行化程序，但不执行优化。
`-xopenmp=stubs`	不再支持此选项。 仅限于 *C* 和 *C++* 程序： OpenMP 桩模块库是为方便用户而提供的。要编译调用 OpenMP 运行时例程但忽略 OpenMP 指令的 OpenMP 程序，请在编译该程序时不要使用 `-xopenmp` 选项，并且将对象文件与 `libompstubs.a` 库链接。例如， `% cc omp_ignore.c -lompstubs` 注 - 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 指令的识别，并且不更改优化级别。

请注意以下其他几点：

- 如果未在命令行中指定 `-xopenmp`，则缺省情况下编译器会采用 `-xopenmp=none`（禁用对 OpenMP 指令的识别）。
- 如果指定 `-xopenmp` 但不带关键字子选项，则编译器会采用 `-xopenmp=parallel`。
- 指定 `-xopenmp=parallel` 或 `-xopenmp=noopt` 会将 `_OPENMP` 宏定义为具有十进制值 `201307L`（在 C/C++ 中），或具有值 `201307`（在 Fortran 中），其中 2013 是 OpenMP 4.0 规范的年份，07 是月份。
- 使用 `dbx` 调试 OpenMP 程序时，请使用 `-xopenmp=noopt -g` 进行编译，以启用全部调试功能。
- 为避免出现编译警告消息，请显式指定适当的优化级别，而不是依赖于可能会发生变化的缺省值。
- 对于 Fortran，使用 `-xopenmp`、`-xopenmp=parallel` 或 `-xopenmp=noopt` 编译表示 `-stackvar`。请参见第 2.3 节 “堆栈和堆栈大小” [23]。
- 在单独的步骤中编译和链接 OpenMP 程序时，请在各个编译及链接步骤中包含 `-xopenmp`。
- 将 `-xvpara` 选项与 `-xopenmp` 选项一起使用可显示有关潜在 OpenMP 编程问题的编译器警告（请参见第 7 章 作用域检查）。

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 4.0 规范。有关在 Oracle Developer Studio 中实现 OpenMP 环境变量的信息，另请参见第 9 章 OpenMP 实现定义的行为。

Oracle Developer Studio 还支持第 2.2.2 节 “Oracle Developer Studio 环境变量” [18]中汇总的不属于 OpenMP 规范的其他环境变量。

注 - OpenMP 和 `autopar` 程序的缺省线程数是每个插槽核心数（即，每个处理器芯片的核心数）的倍数，此数值小于等于 MIN（核心总数 32）。

2.2.1 OpenMP 环境变量行为和缺省值

下表介绍了 Oracle Developer Studio 支持的 OpenMP 环境变量的行为及其缺省值。请注意，为环境变量指定的值不区分大小写，可以采用大写或小写形式。

环境变量	行为、缺省值和示例
`OMP_SCHEDULE`	如果为 `OMP_SCHEDULE` 指定的调度类型不是有效类型（`static`、`dynamic`、`guided`、`auto`、`sunw_mp_sched_reserved`）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

环境变量	行为、缺省值和示例
	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，不指定块大小；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。 如果未设置，则使用缺省值 static（不指定块大小）。 示例：% **setenv OMP_SCHEDULE "GUIDED,4"**
OMP_NUM_THREADS	如果为 OMP_NUM_THREADS 指定的值不是正整数，则会忽略环境变量。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。 如果指定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，并且程序将会停止执行。 如果未设置，则缺省线程数是每个插槽核心数（即，每个处理器芯片的核心数）的倍数，此数值小于等于 MIN（核心总数 32）。 示例：% **setenv OMP_NUM_THREADS 16**
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。 如果未设置，则使用缺省值 TRUE。 示例：% **setenv OMP_DYNAMIC FALSE**
OMP_PROC_BIND	如果为 OMP_PROC_BIND 指定的值不是 TRUE、FALSE 或者逗号分隔的 master、close 或 spread 列表，则以非零状态退出进程。 如果无法将初始线程绑定到 OpenMP 位置列表中的第一个位置，则以非零状态退出进程。 如果未设置，则使用缺省值 FALSE。 示例：% **setenv OMP_PROC_BIND spread**
OMP_PLACES	如果为 OMP_PLACES 指定的值无效或无法实现，则以非零状态退出进程。 如果未设置，则使用缺省值 cores。 示例：% **setenv OMP_PLACES sockets**
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。 如果未设置，则使用缺省值 FALSE。 示例：% **setenv OMP_NESTED TRUE**

环境变量	行为、缺省值和示例
OMP_STACKSIZE	如果为 OMP_STACKSIZE 指定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。 对于 32 位应用程序，辅助线程的缺省堆栈大小为 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	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。 如果未设置，则使用缺省值 PASSIVE。 示例：% **setenv OMP_WAIT_POLICY ACTIVE**
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。 如果未设置，则使用缺省值 4。 示例：% **setenv OMP_MAX_ACTIVE_LEVELS 8**
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。 如果未设置，则使用缺省值 1024。 示例：% **setenv OMP_THREAD_LIMIT 128**
OMP_CANCELLATION	如果为 OMP_CANCELLATION 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。 如果未设置，则使用缺省值 FALSE。 示例：% **setenv OMP_CANCELLATION TRUE**
OMP_DISPLAY_ENV	如果为 OMP_DISPLAY_ENV 指定的值不是 TRUE、FALSE 和 VERBOSE，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。 如果未设置，则使用缺省值 FALSE。 示例：% **setenv OMP_DISPLAY_ENV VERBOSE**

2.2.2 Oracle Developer 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。

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

OpenMP 运行时库能够发出有关许多常见 OpenMP 违规的警告，例如区域嵌套错误、显式屏障位置错误、死锁、环境变量的设置无效等等。

环境变量 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 发出警告消息。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。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回调函数：

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

回调函数的地址作为参数传递给 sunw_mp_register_warn（）。sunw_mp_register_warn（） 在成功注册回调函数后返回 0，失败后返回 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警告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 - 测试或调试程序时，将 SUNW_MP_WARN 设置为 TRUE 可启用运行时检查并显示来自 OpenMP 运行时库的警告消息。请注意，运行时检查会增加程序执行的开销。

2.2.2.3 SUNW_MP_THR_IDLE

控制 OpenMP 程序中正在等待工作（空闲）或者正在屏障处等待的线程的行为。可以将该值设置为以下值之一：SPIN、SLEEP、SLEEP(*time* s)、SLEEP(*time* ms)、SLEEP(*time* mc)，其中 *time* 是一个指定时间量的整数，s、ms 和 mc 是可选后缀，指定时间单位（分别为秒、毫秒和微秒）。如果未指定时间单位，则采用秒作为时间单位。

SPIN 指定线程在等待工作（空闲）或在屏障处等待时应旋转。不带时间参数的 SLEEP 指定等待线程应立即休眠。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。

缺省行为是经过一段时间的旋转等待后进入休眠状态。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
```

以下项等效：

```
% setenv SUNW_MP_THR_IDLE SLEEP(5)
% setenv SUNW_MP_THR_IDLE SLEEP(5s)
% setenv SUNW_MP_THR_IDLE SLEEP(5000ms)
% setenv SUNW_MP_THR_IDLE SLEEP(5000000mc)
```

2.2.2.4 SUNW_MP_PROCBIND

SUNW_MP_PROCBIND 环境变量可用于将 OpenMP 线程绑定到正在运行的系统上的硬件线程。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硬件线程，则会导致性能下降。无法同时设置 SUNW_MP_PROCBIND 和 OMP_PROC_BIND。如果未设置 SUNW_MP_PROCBIND，则缺省值为 FALSE。有关更多信息，请参见第 5 章 处理器绑定（线程关联性）。

2.2.2.5 SUNW_MP_MAX_POOL_THREADS

指定 OpenMP 辅助线程池的最大大小。OpenMP 辅助线程是 OpenMP 运行时库创建的用于在并行区域上运行的那些线程。辅助线程池不包含初始线程（或主线程）和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 OpenMP 辅助线程池将为空，并且将由初始线程（或主线程）执行所有并行区域。如果未设置，则使用缺省值 1023。有关更多信息，请参见第 3.2 节 “控制嵌套并行操作” [27]。

请注意，SUNW_MP_MAX_POOL_THREADS 指定用于程序的非用户 OpenMP 线程的最大数量，而 OMP_THREAD_LIMIT 指定用于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，则必须将它们设置为相同的值。OMP_THREAD_LIMIT 的值必须比 SUNW_MP_MAX_POOL_THREADS 的值大 1。

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。如果未设置 SUNW_MP_MAX_NESTED_LEVELS，则缺省值为 4。有关更多信息，请参见第 3.2 节 “控制嵌套并行操作” [27]。

2.2.2.7 STACKSIZE

设置每个 OpenMP 辅助线程的堆栈大小。该环境变量接受带可选后缀 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。如果未指定后缀，则使用缺省单位千字节。

如果未设置，则对于 32 位应用程序，缺省 OpenMP 辅助线程堆栈大小为 4 MB；对于 64 位应用程序，缺省值为 8 MB。

示例：

```
% setenv STACKSIZE 8192  <- sets the OpenMP helper thread stack size to 8 Megabytes
% setenv STACKSIZE 16M   <- sets the OpenMP helper 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。

2.2.2.8 SUNW_MP_GUIDED_WEIGHT

设置加权因子，该因子用于确定使用 guided 调度的循环中的块大小。该值应为正浮点数，并且应用于程序中所有使用 guided 调度的循环。如果未设置，则缺省加权因子为 2.0。

当使用 for/do 指令指定 schedule(guided, *chunk_size*) 子句时，会在线程请求时将循环迭代以块形式分配给线程，块大小一直减小到 *chunk_size*，最后一个块的大小可能小于该值。线程执行一个迭代块，然后请求另一个块，直到没有块可以分配为止。对于 *chunk_size* 为 1 的情况，每个块的大小按照未分配的迭代数除以线程数的值成比例分配，一直减小到 1。对于 *chunk_size* 为 *k*（这里 *k* 大于 1）的情况，每个块的大小按照相同方式进行确定但遵循如下限制：块包含的迭代不少于 *k* 次（最后一个块除外）。未指定 *chunk_size* 时，该值缺省为 1。

OpenMP 运行时库 libmtsk.so 使用以下公式来计算使用 guided 调度的循环的块大小：

chunk_size = *num-unassigned-iters* / (*guided-weight* * *num-threads*)

- *num-unassigned-iters* 是循环中尚未分配给任何线程的迭代数。
- *guided-weight* 是 SUNW_MP_THR_GUIDED_WEIGHT 环境变量指定的加权因子（如果未设置环境变量，则为 2.0）。
- *num-threads* 是用于执行循环的线程数。

举例来说，假定有一个使用 guided 调度的 100 次迭代循环。如果 *num-threads* = 4，加权因子 = 1.0，那么块大小将是：

25、18、14、10、8、6、4、3、3、2、1...

另一方面，如果 *num-threads*= 4，加权因子 = 2.0，那么块大小将是：

12、11、9、8、7、6、5、5、4、4、3...

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 OpenMP 线程的行为。在这些等待类型中，每一类型的行为都有三种可能：旋转片刻、让出处理器片刻或休眠直至被唤醒。

语法（使用 csh 显示）如下：

```
% setenv SUNW_MP_WAIT_POLICY "IDLE=val:BARRIER=val:TASKWAIT=val"
```

IDLE、BARRIER 和 TASKWAIT 是指定所控制的等待类型的可选关键字。IDLE 指等待运行。BARRIER 指在显式或隐式屏障处等待。TASKWAIT 指在 taskwait 区域等待。上述每个关键字都后跟一个 *val* 设置，使用关键字 SPIN、YIELD 或 SLEEP 来描述等待行为。

SPIN(*time*) 指定等待线程在让出处理器之前应旋转多长时间。*time* 可以是秒、毫秒或微秒（分别用 s、ms 或 mc 表示）。如果不指定 *time* 单位，则使用秒。如果 SPIN 不带 *time* 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处理器的次数。每次让出处理器后，线程会在操作系统调度其运行时再次运行。如果 YIELD 不带 *number* 参数，表示线程在等待时应持续让出。

SLEEP 指定等待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。各个设置必须使用逗号分隔。"SPIN(0),YIELD(0)" 与 "YIELD(0),SPIN(0)" 相同，等效于 SLEEP 或立即休眠。在处理 IDLE、BARRIER 和 TASKWAIT 的设置时，采用左侧优先规则。“左侧优先”规则表示，如果为相同等待类型指定不同的值，那么最左侧的值是要应用的值。在以下示例中，为 IDLE 指定了两个值。第一个是 SPIN，第二个是 SLEEP。因为 SPIN 首先出现（它在字符串的最左侧），所以这是 OpenMP 运行时库将要应用的值。

```
% setenv SUNW_MP_WAIT_POLICY "IDLE=SPIN:IDLE=SLEEP"
```

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

示例 1：

```
% 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。

示例 2：

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

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

示例 3:

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

等待工作（空闲）的线程旋转 2 秒，然后让出处理器 2 次，再转入休眠；在屏障处等待的线程立即转入休眠；在 `taskwait` 处等待的线程让出处理器 10 次，再转入休眠。

2.3 堆栈和堆栈大小

堆栈是临时内存地址空间，用于保留子程序或函数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，从而导致无提示的数据损坏或段故障。

正在执行的程序为执行该程序的初始线程（或主线程）维护一个主堆栈。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。

另外，程序中的每个 OpenMP 辅助线程有自己的线程堆栈。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` 变量在线程堆栈上进行分配。对于 32 位应用程序，辅助线程堆栈的缺省大小为 4 MB；对于 64 位应用程序，缺省值为 8 MB。使用 `OMP_STACKSIZE` 环境变量设置辅助线程堆栈的大小。

请注意，使用 `-stackvar` 选项编译 Fortran 程序将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。对于使用 `-xopenmp`、`-xopenmp=parallel` 或 `-xopenmp=noopt` 选项编译的程序，`-stackvar` 为隐式选项。如果为堆栈分配的内存不足，会导致堆栈溢出。请务必确保堆栈足够大。

C shell 示例：

```
% limit stacksize 32768    <- Sets the main thread stack size to 32 Megabytes
% setenv OMP_STACKSIZE 16384    <- Sets the helper thread stack size to 16 Megabytes
```

Bourne shell 或 Korn shell 示例：

```
$ ulimit -s 32768    <- Sets the main thread stack size to 32 Megabytes

$ OMP_STACKSIZE=16384    <- Sets the helper thread stack size to 16 Megabytes
$ export OMP_STACKSIZE
```

2.3.1 检测堆栈溢出

要检测堆栈溢出，请使用 `-xcheck=stkovf` 编译器选项编译 C、C++ 或 Fortran 程序。语法如下所示：

```
-xcheck=stkovf[:detect | :diagnose]
```

如果指定了 `-xheck=stkovf:detect`，则通过执行通常与错误关联的信号处理程序来处理检测到的堆栈溢出错误。

如果指定了 `-xcheck=stkovf:diagnose`，则通过捕获关联的信号来处理检测到的堆栈溢出错误并通过调用 `stack_violation`(3C) 来诊断错误。如果诊断到堆栈溢出错误，则会向 `stderr` 输出错误消息。如果只指定了 `-xcheck=stkovf`，这是缺省行为。

有关 `-xcheck=stkovf` 编译器选项的更多信息，请参见 `cc`(1)、`CC`(1) 或 `f95`(1) 手册页。

2.4 OpenMP 运行时例程

本节介绍使用 Oracle Developer Studio 编译器编译程序时某些 OpenMP 运行时例程的行为。

2.4.1 `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。

2.4.2 `omp_set_schedule`（）

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

2.4.3 `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。

2.4.4 `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。

2.5 检查和分析 OpenMP 程序

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

- `dbx` 是一个交互式调试工具，它具有以受控方式运行程序的功能并能检查已停止程序的状态。`dbx` 提供了一些为 OpenMP 定制的功能，例如单步执行并行区域；输出区域中的 shared、private 和 threadprivate 变量；输出有关并行区域和任务区域的信息；以及跟踪同步事件。有关更多信息，请参阅《Oracle Developer Studio 12.5：使用 dbx 调试程序》。
- 代码分析器是一个提供静态源代码检查及运行时内存访问检查的工具。检测到的静态错误包括缺少 `malloc`（） 返回值检查、空指针解除引用、缺少函数返回等等。检测到的内存访问错误包括未分配的内存读/写、未初始化的内存读取、释放的内存读/写等等。有关更多信息，请参阅《Oracle Developer Studio 12.5：代码分析器用户指南》。
- 线程分析器是用于检测多线程应用程序中数据争用和死锁的工具。该工具适用于使用 OpenMP、POSIX 线程、Oracle Solaris 线程或这些线程组合编写的应用程序。有关更多信息，请参阅《Oracle Developer Studio 12.5：线程分析器用户指南》、`tha`(1) 和 `libtha`(3) 手册页。
- 性能分析器是用于分析应用程序性能的工具。该工具根据调用堆栈的统计抽样收集性能数据，并显示函数、调用方和被调用方、源代码行和指令的性能度量。性能分析器提供了一些有助于了解 OpenMP 性能的功能，例如 OMP 工作、OMP 等待和 OMP 开销度量，以及应用程序的用户模式和计算机模式视图。有关更多信息，请参阅《Oracle Developer Studio 12.5：性能分析器》、`collect`(1) 和 `analyzer`(1) 手册页。

♦ ♦ ♦ 第 3 章

3 OpenMP 嵌套并行操作

本章讨论 OpenMP 嵌套并行操作的特性。

3.1 OpenMP 执行模型

OpenMP 使用 fork-join（派生-连接）并行执行模型。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）辅助线程组成的线程组。遇到并行构造的线程成为新组中的主线程。所有组成员都执行并行区域中的代码。如果某个线程完成了其在并行区域内的工作，它就会在并行区域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。主线程继续执行程序中并行构造之后的用户代码，而辅助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，则执行嵌套并行区域的组仅由一个线程（遇到嵌套 `parallel` 构造的线程）组成。如果启用嵌套并行操作，则新组可以包含多个线程。

OpenMP 运行时库维护一个辅助线程池，该线程池可用于处理并行区域。当线程遇到并行构造并请求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，将其作为组的一部分。如果池中未包含足够数量的空闲线程，遇到该构造的线程获得的辅助线程可能会比它请求的要少。组完成执行并行区域时，辅助线程就会返回到池中。

3.2 控制嵌套并行操作

可通过在执行程序之前设置各种环境变量，或者通过调用 `omp_set_nested（）` 运行时例程来控制嵌套并行操作。本节讨论可用于控制嵌套并行操作的各种环境变量。

3.2.1 OMP_NESTED

可通过设置 `OMP_NESTED` 环境变量来启用或禁用嵌套并行操作。缺省情况下，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。

例 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 | 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
```

在禁用嵌套并行操作的情况下运行程序会生成以下输出：

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

3.2.2 OMP_THREAD_LIMIT

OMP_THREAD_LIMIT 环境变量的设置控制可用于整个程序的最大 OpenMP 线程数。这一数量包括初始（或主）线程以及 OpenMP 运行时库创建的 OpenMP 辅助线程。缺省情况下，可用于整个程序的最大 OpenMP 线程数为 1024（一个初始线程或主线程及 1023 个 OpenMP 辅助线程）。

请注意，线程池仅由 OpenMP 运行时库创建的 OpenMP 辅助线程组成。该池不包含初始（或主）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，则辅助线程池将为空，并且所有并行区域将由一个线程（初始线程或主线程）执行。

以下示例输出表明，如果池中不包含足够数量的辅助线程，并行区域可能会获得较少的辅助线程。该代码与例 1 “嵌套并行操作示例” 中的代码相同，除了将环境变量 OMP_THREAD_LIMIT 设置为 6。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 个辅助线程。如果将 OMP_THREAD_LIMIT 设置为 6，则池中将最多包含 5 个辅助线程。因此，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辅助线程。以下示例显示一个可能的结果。

```
% setenv OMP_NESTED TRUE
% OMP_THREAD_LIMIT 6
% a.out | 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 = 1
Level 3: number of threads in the team = 1
```

3.2.3 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。如果未设置，则使用缺省值 4。

请注意，设置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（）。

以下样例代码将创建 4 级嵌套并行区域。

```
#include <omp.h>
#include <stdio.h>
#define DEPTH 4
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);
}
```

以下输出显示将 DEPTH 设置为 4 时编译和运行样例代码可能产生的结果。实际结果取决于操作系统调度线程的方式。

```
% setenv OMP_NESTED TRUE
% 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
```

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。以下示例显示一个可能的结果。

```
% setenv OMP_NESTED TRUE
% 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
```

3.3 在嵌套并行区域中调用 OpenMP 运行时例程

本节讨论在嵌套并行区域中调用以下 OpenMP 运行时例程：

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

`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` 调用返回调用线程的值。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。

例 2 在并行区域中调用 OpenMP 运行时例程

```
#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());

        /* 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);
}
```

以下示例显示运行以上程序可能产生的结果：

```
% 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.
```

3.4 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，可让更多线程参与到计算中。

  例如，假定您的程序包含两级并行操作，并且将 OMP_NUM_THREADS 设置为 2。同时，假定您的系统有四个硬件线程，并且您希望使用全部四个硬件线程来加速程序的执行。只并行化任何一级将仅使用两个硬件线程。通过启用嵌套并行操作，即可使用全部四个硬件线程。
- 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。
- 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且对负载进行平衡。假定您的系统有四个硬件线程，并且您希望使用全部四个硬件线程来加速此程序的执行。通常，与对外部并行区域使用两个线程并对内部并行区域使用另外两个线程作为辅助线程相比，对外部并行区域使用全部四个线程能产生更高的性能，因为嵌套并行区域会带来其他屏障。

♦ ♦ ♦ 第 4 章

4 OpenMP 任务处理

本章介绍 OpenMP 任务处理模型。

4.1 OpenMP 任务处理模型

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

4.1.1 OpenMP 任务执行

在 OpenMP 中，显式任务使用 `task` 构造指定，该构造可放置在程序中的任一位置。只要线程遇到 `task` 构造，就会生成一个新任务。

当线程遇到 `task` 构造时，可以选择立即执行任务或将任务延迟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不是最初遇到该任务并将该任务放在池中的线程。

与任务关联的代码仅执行一次。如果代码从始至终都必须由相同的线程执行，则任务为绑定 *(tied)* 任务。如果代码可由多个线程执行，即由不同的线程执行任务代码的不同部分，则任务为非绑定 (untied) 任务。缺省情况下，任务为绑定 (tied) 任务，可以通过在 `task` 指令中使用 `untied` 子句将任务指定为非绑定 (untied) 任务。

允许线程在任务调度点暂停执行任务区域，以便执行另一任务。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。

任务调度点隐含在多个位置中，包括以下位置：

- 紧随在显式任务生成之后的点
- `task` 区域完成点之后的位置
- 在 `taskyield` 区域中

- 在 `taskwait` 区域中
- `taskgroup` 区域末尾
- 在隐式和显式 `barrier` 区域中

除了使用 `task` 构造指定的显式任务外，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 `parallel` 构造时生成的任务。在后一种情况中，每个隐式任务的代码都是 `parallel` 构造内的代码。每个隐式任务分配给组中不同的线程，且为绑定 *(tied)* 任务。

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。

4.1.2 OpenMP 任务类型

OpenMP 规范定义了各种类型的任务，程序员可以使用这些任务来减少任务处理开销。

不延迟 *(undeferred)* 任务是指相对于生成任务，不会延迟执行的任务；也就是说，在不延迟 (undeferred) 任务执行完成前，生成任务区域会暂停。不延迟 (undeferred) 任务可能不会被遇到该任务的线程立即执行。它可能会被放置在池中，然后由遇到该任务的线程或某个其他线程稍后执行。任务执行完成后，生成任务才会恢复执行。例如，`if` 子句表达式求值结果为 *false* 的任务即是一个不延迟 (undeferred) 任务。在这种情况下，将生成不延迟 (undeferred) 任务，且遇到该任务的线程必须暂停当前的任务区域。在包含 `if` 子句的任务完成前，不会恢复当前任务区域的执行。

与不延迟 (undeferred) 任务不同，包括 *(included)* 任务将由遇到该任务的线程立即执行，而不会放在池中稍后执行。此类任务的执行按顺序包含在生成任务区域中。与不延迟 (undeferred) 任务一样，在包括 (included) 任务的执行完成前将暂停生成任务，执行完成后才会恢复生成任务。举例来说，包括 (included) 任务是最终 *(final)* 任务的子孙。

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一样的任务。如果 `mergeable` 子句存在于 `task` 指令中，且生成的任务为不延迟 (undeferred) 任务或包括 (included) 任务，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)* 任务是指会强制其所有子孙任务都成为最终 (final) 和包括 (included) 任务的任务。当 `final` 子句存在于 `task` 指令中，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。

4.2 OpenMP 数据环境

`task` 指令采用以下数据共享属性子句，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遇到 `task` 构造时已知的同名变量。

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。遇到任务构造时，将会使用原始变量的值初始化 `firstprivate` 变量。

OpenMP 规范介绍了并行、任务或工作共享构造中所引用变量的数据共享属性如何确定。

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、显式确定或隐式确定。特定变量有预先确定的数据共享属性；例如，`parallel for/do` 构造中的循环迭代变量是专用变量。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。

注 - 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，而不要依赖 OpenMP 隐式确定作用域规则。

4.3 任务处理示例

本节中的 C/C++ 示例说明如何用 OpenMP `task` 和 `taskwait` 指令递归计算斐波纳契数。

在本例中，并行区域由四个线程执行。`single` 区域确保只由其中一个线程执行调用 `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）。其中一个任务调用 `fib(n-1)`，另一个任务调用 `fib(n-2)`，将这些调用的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)` 的每个调用会进而生成两个任务，这两个任务递归生成，直到传递给 `fib（）` 的参数小于 2。

请注意每个 `task` 指令中的 `final` 子句。如果 `final` 子句表达式 (`n <= THRESHOLD`) 求值结果为 *true*，则生成的任务将为最终 (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。当使用参数 *n* = 9, 8, ..., 2 调

用 fib（） 时，将生成包括 (included) 任务。这些任务将由遇到这些任务的线程立即执行，从而减少在池中放置任务的开销。

taskwait 指令可确保在调用 fib（） 的同一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（因而只有第一个线程调用 fib（）），但是所有四个线程都将参与执行生成的任务和放在池中的任务。

例 3 使用任务计算斐波纳契数

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

#define THRESHOLD 9

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
```

4.4 任务调度约束

OpenMP 规范列出了 OpenMP 任务调度程序必须遵守的几项任务调试约束。

1. 包括 (included) 任务生成后立即执行。
2. 新绑定 (tied) 任务的调度受当前绑定到线程且未在屏障区域中暂停的任务区域集约束。如果此集为空，则可以调度任何新的绑定 (tied) 任务。否则，新绑定 (tied) 任务只有在它为集中各个任务的子孙任务时才会进行调度。
3. 从属任务在其任务依赖项得到满足前不会进行调度。
4. 当显式任务由包含 `if` 子句且该子句表达式求值结果为 *false* 的构造生成且已满足之前的约束，任务将在生成后立即执行。

如果程序还依赖任何与任务调度有关的其他假设，则程序不符合规范。

约束 1 和 4 是 OpenMP 任务应立即执行的两种情况。

约束 2 是为了防止出现死锁。在例 4 “说明任务调度约束 2”中，任务 A、B 和 C 是绑定 (tied) 任务。执行任务 A 的线程将进入临界 `taskyield` 区域且该线程拥有与该临界区域关联的锁。由于 `taskyield` 是任务调度点，因此执行任务 A 的线程可以选择暂停任务 A，改为执行其他任务。假设任务 B 和 C 在任务池中。根据约束 2，执行任务 A 的线程不能执行任务 B，因为任务 B 不是任务 A 的子孙。此时只能调度任务 C，因为任务 C 是任务 A 的子孙。

如果在暂停任务 A 的同时调度任务 B，则任务 A 所绑定到的线程无法进入任务 B 的临界区域，因为该线程已拥有与该临界区域关联的锁。因此，此时将出现死锁现象。约束 2 的目的是避免在代码符合规范时出现此类死锁现象。

请注意，如果编程人员在任务 C 中嵌套了临界段，也会发生死锁现象，但这属于编程错误。

例 4 说明任务调度约束 2

```
#pragma omp task // Task A
{
    #pragma omp critical
    {
        #pragma omp task // Task C
        {
        }
        #pragma omp taskyield
     }
}

#pragma omp task // Task B
{
    #pragma omp critical
     {
     }
}
```

4.5 任务依赖性

OpenMP 4.0 规范介绍了 `task` 指令中的 `depend` 子句，该子句会对任务调度强制使用其他约束。这些约束仅在同级任务之间建立依赖关系。同级任务是指属于同一任务区域的子任务的 OpenMP 任务。

当使用 `depend` 子句指定 `in` 依赖类型时，生成的任务将是所有以前生成的至少引用了 `out` 或 `inout` 依赖类型列表中一个列表项目的同级任务的从属任务。在在 `depend` 子句中指定了 `out` 或 `inout` 依赖类型时，生成的任务将是所有以前生成的至少引用了 `in`、`out` 或 `inout` 依赖类型列表中一个列表项目的同级任务的从属任务。

下面的示例对任务依赖性进行了说明。

例 5 说明仅同步同级任务的 `depend` 子句

```
% cat -n task_depend_01.c
     1  #include <omp.h>
     2  #include <stdio.h>
     3  #include <unistd.h>
     4
     5  int main()
     6  {
     7      int a,b,c;
     8
     9      #pragma omp parallel
    10      {
    11          #pragma omp master
    12          {
    13              #pragma omp task depend(out:a)
    14              {
    15                  #pragma omp critical
    16                  printf ("Task 1\n");
    17              }
    18
    19              #pragma omp task depend(out:b)
    20              {
    21                  #pragma omp critical
    22                  printf ("Task 2\n");
    23              }
    24
    25              #pragma omp task depend(in:a,b) depend(out:c)
    26              {
    27                  printf ("Task 3\n");
    28              }
    29
    30              #pragma omp task depend(in:c)
    31              {
    32                  printf ("Task 4\n");
    33              }
    34          }
    35          if (omp_get_thread_num () == 1)
```

```
    36            sleep(1);
    37      }
    38      return 0;
    39  }

% cc -xopenmp -O3 task_depend_01.c
% a.out
Task 2
Task 1
Task 3
Task 4

% a.out
Task 1
Task 2
Task 3
Task 4
```

在本例中，任务 1、2、3 和 4 都是同一隐式任务区域的子任务，因此都是同级任务。任务 3 是任务 1 和 2 的从属任务，因为在 depend 子句中的 *a* 参数指定了依赖性。因此，在任务 1 和 2 完成前，无法调度任务 3。同样，任务 4 是任务 3 的从属任务，因此在任务 3 完成前，无法调度任务 4。

请注意，depend 子句仅同步同级任务。下面的示例（例 6 “说明不影响非同级任务的 depend 子句”）显示了 depend 字句不影响非同级任务的情况。

例 6　　说明不影响非同级任务的 depend 子句

```
% cat -n task_depend_02.c
     1  #include <omp.h>
     2  #include <stdio.h>
     3  #include <unistd.h>
     4
     5  int main()
     6  {
     7      int a,b,c;
     8
     9      #pragma omp parallel
    10      {
    11          #pragma omp master
    12          {
    13              #pragma omp task depend(out:a)
    14              {
    15                  #pragma omp critical
    16                  printf ("Task 1\n");
    17              }
    18
    19              #pragma omp task depend(out:b)
    20              {
    21                  #pragma omp critical
    22                  printf ("Task 2\n");
    23
```

```
24                  #pragma omp task depend(out:a,b,c)
25                  {
26                    sleep(1);
27                    #pragma omp critical
28                    printf ("Task 5\n");
29                  }
30              }
31
32              #pragma omp task depend(in:a,b) depend(out:c)
33              {
34                  printf ("Task 3\n");
35              }
36
37              #pragma omp task depend(in:c)
38              {
39                  printf ("Task 4\n");
40              }
41          }
42          if (omp_get_thread_num () == 1)
43            sleep(1);
44      }
45      return 0;
46  }
```

```
% cc -xopenmp -O3 task_depend_02.c
% a.out
Task 1
Task 2
Task 3
Task 4
Task 5
```

在上例中，任务 5 是任务 2 的子任务，不是任务 1、2、3 或 4 的同级任务。因此，尽管 `depend` 子句引用相同的变量（*a*、*b*、*c*），任务 5 与任务 1、2、3 或 4 之间也不存在依赖关系。

4.5.1 关于任务依赖性的说明

请注意以下有关任务依赖性的提示：

- `depend` 子句中的 `in`、`out` 和 `inout` 依赖类型类似于读写操作，但是 `in`、`out` 和 `inout` 依赖类型只用于建立任务依赖性。它们不指示任务区域内的任何内存访问模式。包含 `depend(in:`*a*`)`、`depend(out:`*a*`)` 或 `depend(inout:`*a*`)` 子句的任务可能可以在其区域内读取或写入变量 *a*，也可能根本无法访问变量 *a*。
- 如果在同一任务指令中同时包含 `if` 子句和 `depend` 子句，当 `if` 子句的条件求值结果为 *false* 时，开销可能很大。当某个任务包含 `if(`*false*`)` 子句时，遇到该任务的线程必须暂停当前的任务区域，直到生成的任务（包含 `if(`*false*`)` 子句的任务）完成为止。同时，任务调度程序在所生成任务的任务依赖项满足之前，无法调度该任务。由于紧随在显式任务生成之后的点是任务调度点，因此任务调度程序将尝试调度任务，

以便满足不延迟 (undeferred) 任务的任务依赖项。查找和调度池中正确任务的开销可能会很高。在最糟糕的情况下，开销可能与拥有一个 taskwait 区域那么大。

- 相同任务或同级任务的 depend 子句中使用的列表项目必须指明相同存储或不相交存储。因此，如果 depend 子句中出现数组段，请确保数组段指明了相同存储或不相交存储。

4.6 使用 taskwait 和 taskgroup 同步任务

您可以使用 taskwait 或 taskgroup 指令同步任务。

当线程遇到 taskwait 构造时，当前任务会暂停，直到它在 taskwait 区域前生成的所有子任务均已完成执行为止。

当线程遇到 taskgroup 构造时，它将开始执行 taskgroup 区域。在 taskgroup 区域末尾，当前任务暂停，直到它在 taskgroup 区域中生成的所有子任务及其所有子孙任务均已完成执行为止。

请注意 taskwait 与 taskgroup 之间的不同。使用 taskwait，当前任务仅等待其子任务。使用 taskgroup，当前任务不仅等待在 taskgroup 中生成的子任务，还等待这些子任务的所有子孙任务。下面两个示例介绍了二者之间的差异。

例 7 taskwait 示例

```
% cat -n taskwait.c
     1  #include <omp.h>
     2  #include <stdio.h>
     3  #include <unistd.h>
     4
     5  int main()
     6  {
     7    #pragma omp parallel
     8    #pragma omp single
     9    {
    10      #pragma omp task
    11      {
    12        #pragma omp critical
    13        printf ("Task 1\n");
    14
    15        #pragma omp task
    16        {
    17          sleep(1);
    18          #pragma omp critical
    19          printf ("Task 2\n");
    20        }
    21      }
    22
    23      #pragma omp taskwait
    24
```

```
    25      #pragma omp task
    26      {
    27        #pragma omp critical
    28        printf ("Task 3\n");
    29      }
    30    }
    31
    32    return 0;
    33  }
```

例 8 taskgroup 示例

```
% cat -n taskgroup.c
     1  #include <omp.h>
     2  #include <stdio.h>
     3  #include <unistd.h>
     4
     5  int main()
     6  {
     7    #pragma omp parallel
     8    #pragma omp single
     9    {
    10      #pragma omp taskgroup
    11      {
    12        #pragma omp task
    13        {
    14          #pragma omp critical
    15          printf ("Task 1\n");
    16
    17          #pragma omp task
    18          {
    19            sleep(1);
    20            #pragma omp critical
    21            printf ("Task 2\n");
    22          }
    23        }
    24      } /* end taskgroup */
    25
    26      #pragma omp task
    27      {
    28        #pragma omp critical
    29        printf ("Task 3\n");
    30      }
    31    }
    32
    33    return 0;
    34  }
```

虽然 taskwait.c 和 taskgroup.c 的源代码几乎相同，但是 taskwait.c 在第 23 行有 taskwait 指令，而 taskgroup.c 在第 10 行有包含任务 1 和任务 2 的 taskgroup 构造。在这两个程序中，taskwait 和 taskgroup 指令同步任务 1 和任务 3 的执行；不同之处在于它们是否同步任务 2 和任务 3 的执行。

在使用 taskwait.c 的情况下，任务 2 不是 taskwait 区域绑定到的并行区域所生成的隐式任务的子任务。因此，任务 2 不必在 taskwait 区域末尾完成。可以在完成任务 2 之前调度任务 3。

对于 taskgroup.c，任务 2 是任务 1 的子任务，该任务是在 taskgroup 区域中生成的。因此，任务 2 必须在 taskgroup 区域末尾完成，然后才能遇到和调度任务 3。

4.7 OpenMP 编程注意事项

任务处理功能使 OpenMP 程序的复杂性有所增加。本节讨论一些要考虑的与任务相关的编程问题。

4.7.1 Threadprivate 和线程特定的信息

当线程遇到任务调度点时，实现可能会暂停当前任务并安排线程处理另一个任务。该行为意味着任务中的 threadprivate 变量或其他线程特定的信息（如线程数）在任务调度点前后可能会有所不同。

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。

4.7.2 OpenMP 锁

从 OpenMP 3.0 开始，锁为任务所有，而不再由线程所有。某个任务获取锁之后，该任务就会拥有该锁，同一任务必须在任务完成前释放锁。但是，critical 构造仍然保留了基于线程的互斥机制。

由于锁为任务所有，因此使用锁时应格外小心。下例符合 OpenMP 2.5 规范，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相同。并行区域的主线程与初始线程相同。但是，该示例不符合最近的规范，因为释放锁 *lck* 的任务区域不是获取锁的任务区域。

例 9　　OpenMP 3.0 之前的锁应用

```
#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;

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

4.7.3 对堆栈数据的引用

任务可以引用任务构造所在例程（主机例程）的堆栈中的数据。由于任务的执行可能会延迟到下一个隐式或显式屏障，所以任务有可能在主机例程的堆栈已经弹出，堆栈数据被覆盖（从而销毁任务引用的堆栈数据）之后执行。

确保插入所需的同步，以便当任务引用变量时，这些变量仍在堆栈中，如本节中的两个示例所示。

在例 10 “堆栈数据：引用不正确”中，在 `task` 构造中将 *i* 指定为 `shared`，任务会访问在 `work（）` 例程的堆栈中分配的 *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。那时，当任务引用 *i* 时，它会访问当时碰巧在堆栈中不确定的某个值。

为了获得正确的结果，请确保 `work（）` 不在任务完成前返回。通过将 `taskwait` 指令插在 `task` 构造之后可以实现此目标，如例 11 “堆栈数据：引用正确”中所示。或者，可以在 *task* 构造中将 `i` 指定为 `firstprivate` 而不是 `shared`。

例 10 堆栈数据：引用不正确

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

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

例 11　　堆栈数据：引用正确

```
#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();
    }
 }
```

在以下示例中，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 Oracle Developer Studio 中是 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，请确保任务在 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，如例 13 “段数据：引用正确” 中所示。或者，可以在 *task* 构造中将 j 指定为 firstprivate 而不是 shared。

例 12 段数据：引用不正确

```
#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);
             }
          }
       } /* Implicit barrier for sections */
    } /* Implicit barrier for parallel */

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

例 13 段数据：引用正确

```
#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
         }
      } /* Implicit barrier for sections */
   }/* Implicit barrier for parallel */

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

♦ ♦ ♦ 第 5 章

处理器绑定（线程关联性）

本章介绍处理器绑定。

5.1 处理器绑定概述

通过处理器绑定（也称为线程关联性），程序将指示操作系统，程序中的线程在整个执行过程中均应在计算机上的相同位置运行，不应移到其他位置。此处的位置是指插槽、核心或硬件线程的特定分组。

处理器绑定可以改善具有特定数据重用模式的应用程序的性能，其模式为：并行区域或工作共享区域中的线程访问的数据位于上次调用的并行区域或工作共享区域的本地高速缓存中。

计算机系统可被视为插槽、核心和硬件线程组成的层次结构。每个插槽包含一个或多个核心，每个核心又包含一个或多个硬件线程。

在 Oracle Solaris 平台上，可以使用 `psrinfo`(1M) 命令列出可用的硬件线程。在 Linux 平台上，文本文件 `/proc/cpuinfo` 提供有关可用硬件线程的信息。

当操作系统将某个线程绑定到处理器时，实际上是将该线程绑定到特定的硬件线程或一组硬件线程。

要控制 OpenMP 线程到处理器的绑定，您可以使用 OpenMP 4.0 环境变量 `OMP_PLACES` 和 `OMP_PROC_BIND`。或者，您可以使用 Oracle 特定的环境变量 `SUNW_MP_PROCBIND`。这两组环境变量不得混用。有关环境变量的介绍，请参见第 5.2 节 “OMP_PLACES 和 OMP_PROC_BIND” [52]。

注 - 本章中介绍的 OpenMP 环境变量仅控制 OpenMP 线程的绑定（即 OpenMP 运行时库中记录的所有用户线程，以及该库创建的辅助线程）。这些环境变量不控制其他用户线程的绑定。如果用户线程遇到 OpenMP 构造或调用 OpenMP 运行时例程，该库将记录该用户线程。

5.2 OMP_PLACES 和 OMP_PROC_BIND

OpenMP 4.0 提供 OMP_PLACES 和 OMP_PROC_BIND 环境变量来指定程序中的 OpenMP 线程如何绑定到处理器。这两个环境变量通常结合使用。OMP_PLACES 用于指定线程将绑定到的计算机位置。OMP_PROC_BIND 用于指定绑定策略（线程关联性策略），这项策略指定如何将线程分配到位置。仅设置 OMP_PLACES 不会启用绑定。您还需设置 OMP_PROC_BIND。

根据 OpenMP 规范，OMP_PLACES 的值可以是以下两种类型值之一：用于描述一组位置（线程、核心或插槽）的抽象名称或非负数描述的明确位置列表。通过使用 <lowerbound> : <length> : <stride> 表示法表示以下编号列表，还可以使用间隔来定义位置："<lower-bound>, <lower-bound> + <stride>, ..., <lower-bound> + (<length>-1) *<stride>"。省略 <stride> 时，将采用单元距值。如果不设置 OMP_PLACES，则缺省值为核心。

例 14 一个位置中一个硬件线程

```
% OMP_PLACES="{0:1}:8:32"

{0:1} defines a place which has one hardware thread only, namely place {0}. The interval {0:
1}:8:32 is therefore equivalent to
{0}:8:32, which defines 8 places starting with place {0}, and the stride is 32. So the list of
 places is as follows:

Place 0: {0}
Place 1: {32}
Place 2: {64}
Place 3: {96}
Place 4: {128}
Place 5: {160}
Place 6: {192}
Place 7: {224}
```

例 15 一个位置中两个硬件线程

```
% OMP_PLACES="{0:2}:32:8"

{0:2} defines a place which has two hardware threads, namely place {0,1}. The interval {0:2}:
24:8 is therefore equivalent to
{0,1}:24:8 which defines 24 places starting with place {0,1}, and the stride is 8. So the list
 of places is as follows:

Place 0: {0,1}
Place 1: {8,9}
Place 2: {16,17}
Place 3: {24,25}
Place 4: {32,33}
Place 5: {40,41}
Place 6: {48,49}
```

```
Place 7: {56,57}
Place 8: {64,65}
Place 9: {72,73}
Place 10: {80,81}
Place 11: {88,89}
Place 12: {96,97}
Place 13: {104,105}
Place 14: {112,113}
Place 15: {120,121}
Place 16: {128,129}
Place 17: {136,137}
Place 18: {144,145}
Place 19: {152,153}
Place 20: {160,161}
Place 21: {168,169}
Place 22: {176,177}
Place 23: {184,185}
```

除了 OMP_PLACES 和 OMP_PROC_BIND 这两个环境变量外，OpenMP 4.0 还提供可在 parallel 指令中使用的 proc_bind 子句。proc_bind 子句用于指定如何将执行并行区域的线程组绑定到处理器。

有关 OMP_PLACES 和 OMP_PROC_BIND 环境变量以及 proc_bind 子句的详细信息，请参阅 OpenMP 4.0 规范。

5.2.1 控制 OpenMP 4.0 中的线程关联性

本节详细介绍 OpenMP 4.0 规范中的 2.5.2 节“控制 OpenMP 线程关联性”。

当线程遇到包括 proc_bind 子句的并行构造时，OMP_PROC_BIND 环境变量用于确定将线程绑定到位置的策略。如果并行构造包括 proc_bind 子句，则 proc_bind 子句指定的绑定策略将覆盖 OMP_PROC_BIND 指定的策略。组中的线程分配到某个位置后，实现不会将其移到其他位置。

master 线程关联性策略指示执行环境将组中的每个线程分配到与主线程相同的位置。此策略不会更改位置分区，且每个隐式任务将继承父隐式任务的 *place-partition-var* 内部控制变量 (Internal Control Variable, ICV)。

close 线程关联性策略指示执行环境将组中的线程分配到靠近父线程位置的位置。此策略不会更改位置分区，且每个隐式任务将继承父隐式任务的 *place-partition-var* ICV。如果 T 指组中的线程数量，P 指父线程所在位置分区中的位置数量，那么将按以下方法将组中的线程分配到位置：

- T <= P。主线程在父线程（即遇到并行构造的线程）的位置上执行。下一个线程编号最小的线程在位置分区的下一个位置上执行，以此类推，相对主线程的位置分区进行绕回分配。
- T > P。每个位置 P 将包含 S_p 个线程编号连续的线程，其中 floor(T/P) <= S_p <= ceiling(T/P)。前 S_0 个线程（包括主线程）分配到父线程的位置。接下来的 S_1 个线

程分配到位置分区的下一个位置，以此类推，相对主线程的位置分区进行绕回分配。当 P 不能整除 T 时，特定位置的具体线程数由实现定义。

spread 线程关联性策略的目的是将一组 T 个线程稀疏地分布到父线程所在位置分区的 P 个位置。要实现稀疏分布，首先将父分区分割为 T 个子分区（如果 T <= P）或 P 个子分区（如果 T > P）。然后，将一个线程 (T <= P) 或一组线程 (T > P) 分配到每个子分区。每个隐式任务的 *place-partition-var* ICV 均设置为其子分区。分割为子分区不仅是实现稀疏分布的机制，还定义了在创建嵌套并行区域时可供线程使用的位置子集。将线程分配到位置的方法如下所示：

- T <= P。父线程的位置分区拆分为 T 个子分区，其中每个子分区包含 floor(P/T) 个或 ceiling(P/T) 个连续位置。向每个子分区分配一个线程。主线程在父线程的位置上执行，并分配到包含该位置的子分区。下一个线程编号最小的线程分配到下一个子分区的第一个位置，以此类推，相对主线程的原始位置分区进行绕回分配。
- T > P。父线程的位置分区拆分为 P 个子分区，每个子分区包含一个位置。每个子分区分配有 S_p 个线程编号连续的线程，其中 floor(T/P) <= S_p <= ceiling(T/P)。前 S_0 个线程（包括主线程）分配到包含父线程位置的子分区。接下来的 S_1 个线程分配到下一个子分区，以此类推，相对主线程的原始位置分区进行绕回分配。当 P 不能整除 T 时，特定子分区中的具体线程数由实现定义。

注 - 如果在未分配完所有线程前就到达分区位置的末端，则需要绕回分配机制。例如，在采用 *close* 策略且 T <= P 时，如果主线程分配到位置分区中第一个位置以外的位置，那么就有可能需要绕回分配机制。在这种情况下，线程 1 分配到主线程位置之后的位置，线程 2 分配到线程 1 位置之后的位置，以此类推。这样，可能在未分配完所有线程前，就已到达位置分区的末端。在这种情况下，将继续从位置分区的第一个位置开始分配线程。

5.3 SUNW_MP_PROCBIND

`SUNW_MP_PROCBIND` 是特定于 Oracle 的传统环境变量，用于指定处理器绑定。本节介绍可以为此变量设置的值。

注 - `SUNW_MP_PROCBIND` 的值为非负整数，表示逻辑硬件线程 ID，可能与实际硬件线程 ID 不同。虽然硬件线程 ID 可能是连续的，但是仍然会存在间隔。例如，在 16 核 SPARC 系统中，硬件线程 ID 可以为 0、1、2、3、8、9、10、11、512、513、514、515、520、521、522、523。但是，逻辑处理器 ID 是以 0 开头的连续整数。如果系统中可用的硬件线程数为 *n*，则其逻辑处理器 ID 为 0、1...*n*-1。

`SUNW_MP_PROCBIND` 的可能值包括：

- 大写或小写的字符串 `FALSE`、`TRUE`、`COMPACT` 或 `SCATTER`。例如：

```
% setenv SUNW_MP_PROCBIND "TRUE"
```

- FALSE – OpenMP 线程将不绑定到任何处理器。这是缺省设置。
- TRUE – OpenMP 线程将以循环方式绑定到硬件线程。绑定的起始硬件线程由运行时库根据性能最佳这一目标来确定。
- COMPACT – OpenMP 线程将绑定到系统上尽可能靠近的硬件线程上。COMPACT 允许线程共享数据高速缓存，从而改进数据局域性。
- SCATTER – OpenMP 线程将绑定到彼此远离的硬件线程上。此设置将为每个线程提供更多的内存带宽。

- 非负整数 – 指示 OpenMP 线程应绑定到的硬件线程的起始逻辑 ID。OpenMP 线程将以循环方式绑定到硬件线程，从具有指定逻辑 ID 的硬件线程开始，在绑定到逻辑 ID 为 *n*-1 的硬件线程之后，将绕回到逻辑 ID 为 0 的硬件线程。

  例如：

  ```
  % setenv SUNW_MP_PROCBIND "2"
  ```

- 包含两个或更多个非负整数的列表 – OpenMP 线程将以循环方式绑定到具有指定逻辑 ID 的硬件线程。将不使用逻辑 ID 不在指定之列的硬件线程。

  下例将两个线程绑定到硬件线程 2，一个线程绑定到硬件线程 4，一个线程绑定到硬件线程 6（如果使用了 4 个线程）。

  ```
  % setenv SUNW_MP_PROCBIND "2 2 4 6"
  ```

- 由连字符 ("-") 分隔的两个非负整数 – OpenMP 线程将以循环方式绑定到某一范围内的硬件线程，该范围以第一个逻辑 ID 开头、以第二个逻辑 ID 结尾。第一个整数必须小于或等于第二个整数。将不使用逻辑 ID 不在指定范围内的硬件线程。

  例如：

  ```
  % setenv SUNW_MP_PROCBIND "0-6"
  ```

如果为 SUNW_MP_PROCBIND 指定的值无效，或者如果给定的逻辑 ID 无效，将出现错误消息且程序将终止执行。

如果 OpenMP 线程数大于可用的硬件线程数，则对于某些硬件线程，将有多个 OpenMP 线程绑定到它们。这种情况可能会对性能有负面影响。

5.4 与处理器集进行交互

处理器集是留出的专供指定进程使用的系统处理器集的一部分。处理器集允许将进程绑定到处理器组，而非单个处理器。要指定处理器集，可以在 Oracle Solaris 平台上使用 psrset(1M) 实用程序，或者在 Linux 平台上使用 taskset 命令。当前，处理器绑定不考虑在 Linux 上使用 taskset 命令指定的处理器集。

♦ ♦ ♦ 第 6 章

6 自动确定变量的作用域

确定 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。本章介绍自动确定变量的作用域。

6.1 确定变量作用域概述

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。编程人员使用数据共享属性子句显式声明变量的作用域，或者编译器根据 OpenMP 4.0 规范第 2.14.1 节的“数据共享属性规则”自动应用有关预先确定或隐式确定作用域的规则。有关数据共享属性的更多信息，请参见 OpenMP 4.0 规范的第 2.14.3 节“数据共享属性子句”。

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，数据共享属性规则可能会产生一些意外的结果。`task` 指令增加了确定作用域的复杂性和难度。

Oracle Developer Studio 编译器支持的自动确定作用域功能（称为自动确定作用域）使得程序员无需显式确定变量的作用域。通过自动确定作用域功能，编译器可在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。

在早期编译器发行版中，自动确定作用域功能只能用于 `parallel` 构造中的变量。当前的 Oracle Developer Studio 编译器将自动确定作用域功能的范围扩展到了 `task` 构造中引用的标量变量。

6.2 自动确定作用域数据范围子句

要调用自动确定作用域功能，可通过在 `__auto` 数据作用域子句中指定要确定作用域的变量或使用 `default(__auto)` 子句。这两种方法都是 Oracle Developer Studio 对 OpenMP 规范的扩展。

6.2.1 __auto 子句

语法：__auto(*list-of-variables*)

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

__auto 子句可以出现在 parallel 指令（包括 parallel for/do、parallel sections 和 Fortran parallel workshare 指令）或 task 指令中。

parallel 或 task 构造中的 __auto 子句指示编译器自动确定构造中指定变量的作用域。（请注意 auto 前面的两个下划线）。

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。

6.2.2 default(__auto) 子句

语法：default(__auto)

对于 Fortran，也接受 DEFAULT(__AUTO)。

default(__auto) 子句可以出现在 parallel 指令（包括 parallel for/do、parallel sections 和 Fortran parallel workshare 指令）或 task 指令中。

parallel 或 task 构造中的 default(__auto) 子句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。

6.3 parallel 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 parallel 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，如工作共享 for/do 循环的循环索引变量。

6.3.1 parallel 构造中标量变量的作用域规则

在自动确定 parallel 构造中引用的不是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 parallel 构造的线程而言，如果在该构造中使用变量不会导致数据争用情形，则将该变量的作用域确定为 shared。
- PS2：如果在执行 parallel 构造的每个线程中，变量始终先写入后再由同一线程读取，则将该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，并且该变量先读取后写入（在并行构造后写入），而构造为 parallel for/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。

6.3.2 parallel 构造中数组的作用域规则

- PA1：对于组中执行并行构造的线程而言，如果在该构造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。

6.4 任务构造中标量变量的作用域规则

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 构造中标量变量的作用域。

注 - 在此发行版的 Oracle Developer Studio 中，任务的自动确定作用域功能不处理数组。

在自动确定 task 构造中引用的不是预先确定或隐式确定作用域的标量变量的作用域时，编译器会按数字顺序根据规则 TS1 - TS5 来检查变量的使用。这些规则不适用于由 OpenMP 规范隐式确定作用域的变量，如 parallel for/do 循环的循环索引变量。

- TS1：如果变量的使用在 task 构造中为只读，并且在包括该 task 构造的 parallel 构造中也为只读，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 task 构造中为只读，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构造的线程中，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 task 构造中不为只读，并且在任务中执行的某些读取操作可能会获取在任务之外分配的值，而且向任务中的变量分配的值不在任务之外使用，则自动将变量的作用域确定为 firstprivate。

6.5 关于自动确定作用域的说明

在 `parallel` 构造上指定 `_auto(`*list-of-variables*`)` 或 `default(_auto)` 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。

在对不是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相应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。如果不匹配任何规则或自动确定作用域功能无法处理变量，则编译器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(0)`（Fortran 中为 `if(.false.)`）子句一样。有关更多信息，请参见第 6.6 节 “使用自动确定作用域的限制” [60]。

如果变量的使用不匹配任何规则，或者如果源代码过于复杂，编译器无法进行充分的分析，通常无法确定变量的作用域。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。

6.6 使用自动确定作用域的限制

- 要启用自动确定作用域功能，必须使用 `-xopenmp` 选项在优化级别 `-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确定作用域功能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 /* task 1 */
{
  ...
  #pragma omp task /* task 2 */
  {
    ...
  }
 ...
}
```

在此示例中，由于 `task 2` 在语法上嵌套在 `task 1` 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 `task 2` 中引用的所有变量的作用域确定为 `shared`，并将 `task 2` 视为如同对任务指定了 `if(0)`（在 Fortran 中为 `if(.false.)`）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。请在可能的情况下使用 `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。

6.7 检查自动确定作用域的结果

自动确定作用域的详细结果显示在编译器注释中。在使用 `-g` 选项编译源时，编译器将生成内联注释。可以使用 `er_src` 命令查看该注释，如以下示例所示。`er_src` 命令作为 Oracle Developer 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 Developer Studio 12.5：性能分析器》。

要快速检查自动确定作用域的结果，请使用 `-xvpara` 选项进行编译。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。

例 16 使用 `-xvpara` 检查自动确定作用域的结果

```
% 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 -xvpara -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，指定 `-xvpara` 时将发出警告消息，如以下示例所示。

例 17 使用 `-xvpara` 时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

% f95 -xopenmp -xO3 -xvpara -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.
```

有关自动确定作用域的更多详细信息将显示在 `er_src` 显示的编译器注释中，如以下示例所示。

例 18 使用 `er_src` 显示的自动确定作用域详细结果

```
% 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 _$d1A2.MAIN_
      along with 0 inner loops
   L1 could not be pipelined because it contains calls
     3.         DO I=1, 100
     4.                 T = Y(I)
     5.                 CALL FOO(X)
     6.                 X(I) = T*T
     7.         END DO
     8. C$OMP END PARALLEL DO
     9.         END
    10.
```

6.8 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。这些规则在第 6.3 节 “parallel 构造的作用域规则” [58]和第 6.4 节 “任务构造中标量变量的作用域规则” [59]中进行了介绍。

例 19　　说明自动确定作用域规则的复杂示例

```
 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
```

在此示例中，函数 FOO（） 包含一个 parallel 构造，其中包含一个 single 构造、一个工作共享 do 构造以及一个 critical 构造。

在 parallel 构造中使用了变量 I、N、MM、T、W、M、X 和 Y。编译器按照如下方式确定这些变量的作用域：

- 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 在并行构造中是只读的，因此不会导致数据争用，这样，按照规则 PS1，将其作用域确定为 shared。

- 任何执行并行构造的线程都会执行第 14 行，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 PS1 不适用。由于在同一线程中该写入操作发生在读取 MM 之前，因此，按照规则 PS2 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，然后在第 21 行写入该标量，因此，规则 PS1 和 PS2 不适用。加法运算同时符合结合律和交换律，因此，按照规则 PS3，将 W 的作用域确定为 reduction (+)。
- 在第 11 行写入标量 M，该行位于 single 构造内。single 构造末尾的隐式屏障可确保第 11 行的写入不会与第 28 行的读取或第 29 行的写入同时发生，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 parallel 构造中写入和读取 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 在构造中是只读的，不进行写入，因此，按照规则 PA1，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对 Y 的相同元素进行写入。由于不会发生数据争用，因此，按照规则 PA1 将 Y 的作用域确定为 shared。

例 20 QuickSort 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 shows the following compiler commentary:

      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);

      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 `task` 构造和 `parallel` 构造中都是只读的。因此，根据 `TS1` 自动将其作用域确定为 `firstprivate`。

例 21　　斐波纳契示例

```
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 shows the following compiler commentary:

   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 task 构造和 parallel 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（遇到并生成了这两个任务）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可访问状态。因此，根据 TS2 自动将 *x* 和 *y* 的作用域确定为 shared。

例 22 使用 single 和 task 构造的示例

```
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 shows the following compiler commentary:

   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
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 `parallel` 构造中的专用变量。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 `single` 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，因此在 `task1` 正在执行时可以访问 *xx*。因此，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 `parallel` 构造中不是只读的。由于 *xx* 在 `parallel` 构造中预先确定为 `private`，因此无法确定在 `task2` 正在执行时是否可以访问 *xx*。因此，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。因此，根据 `TS4`，在 `task2` 中自动将 `yy` 的作用域确定为 `private`。

例 23　使用 `task` 和 `taskwait` 构造的示例

```
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 shows the following compiler commentary:

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

   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。但是，在任务中读取 *x* 将获取在任务外定义的 *x* 值（因为 *xx* 在 parallel 构造中为 shared）。因此，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。因此，根据 TS2，自动将 *yy* 的作用域确定为 shared。

♦ ♦ ♦ 第 7 章

7 作用域检查

Oracle Developer Studio C、C++ 和 Fortran 编译器提供了一个作用域检查功能，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。本章介绍如何使用作用域检查功能。

7.1 作用域检查概述

自动确定作用域功能可以帮助您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，或者无法实现您期望的结果。错误确定作用域可能引发一些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内为变量使用未定义的值。

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，编译器将检查编程人员指定的数据共享属性、预先确定和隐式确定的数据共享属性以及自动确定作用域结果。

7.2 使用作用域检查功能

要启用作用域检查，请使用 `-xvpara` 和 `-xopenmp` 选项编译 OpenMP 程序。优化级别应为 `-xO3` 或更高。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。如果某些变量的作用域会引发问题，编译器将发出警告消息，有时还会建议使用正确的数据共享属性子句。例如，编译器检测到下列情形时会发出警告消息：

- 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
- 例如，指定了要在并行区域中共享的变量但访问并行区域中的该变量可能会导致数据争用，或者指定了要在并行区域中专用的变量但分配给并行区域中该变量的值在并行区域后使用，这样 OpenMP 数据共享属性子句就会出现问题。

以下示例对作用域检查进行了说明。

例 24 使用 -xvpara 进行作用域检查

```
% cat t.c

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

int main()
{
  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
```

以下示例说明了如何报告潜在的作用域错误。

例 25 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;
}

% 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
```

此示例显示了作用域检查可以检测到的一些典型错误。

1. *r* 被指定为归约变量，其运算为 `+`，但实际上运算应为 `*`。
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* 的引用可能会读取未定义的值。编译器会指出此问题，并建议将 `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 `parallel for` 循环后使用循环索引值，程序员可能希望将其指定为 `LASTPRIVATE`。但在上述示例中，在循环后根本没有引用 *i*。编译器会发出警告，并建议将 *i* 的作用域确定为 `private`。使用 `private` 而不是 `lastprivate` 可以提高性能。
4. 未显式指定变量 *b* 的任何数据共享属性。根据 OpenMP 规范，`b` 的作用域将隐式确定为 `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`。

7.3 使用作用域检查时的限制

- 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。
- 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。请在可能的情况下使用 `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。

注 - 使用 `-xvpara` 编译器选项进行的作用域检查使用静态（编译时）分析确定程序中的潜在问题。另一方面，线程分析器工具使用动态（运行时）分析检查程序中的数据争用和死锁。结合使用这两种方法可以尽可能多地检测出程序中的错误。

♦ ♦ ♦ 第 8 章

8 性能注意事项

具备了正确无误、可以正常运行的 OpenMP 应用程序后，请注意其整体性能。本章提供了一些可以提高 OpenMP 应用程序效率和可伸缩性的最佳做法。

8.1 一些常规性能建议

本节介绍了几项可以提高 OpenMP 应用程序性能的通用技术。

- 将同步降至最少。
  - 避免使用或最大限度地少用同步功能，如 `barrier`、`critical`、`ordered`、`taskwait` 和锁。
  - 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。如果区域中的工作共享循环后面不跟区域中的任何代码，可以向其添加 `nowait`，从而消除一个冗余屏障。
  - 适用的时候，对细粒度的锁定使用命名的 `critical` 段，以免程序中的所有 `critical` 段都使用相同的缺省锁。
- 使用 `OMP_WAIT_POLICY`、`SUNW_MP_THR_IDLE` 或 `SUNW_MP_WAIT_POLICY` 环境变量控制等待线程的行为。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可能不合适，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，使等待线程旋转的主动等待策略将会提供更高的性能。如果应用程序与其他应用程序同时运行，将等待线程置于休眠状态的被动等待策略将会改善系统吞吐量。
- 尽可能在最高级别（如最外面的循环）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：

```
#pragma omp parallel
{
   #pragma omp for
   {
      ...
   }
}
```

```
#pragma omp parallel
{
   #pragma omp for
   {
      ...
   }
}
```

更有效的构造：

```
#pragma omp parallel
{
   #pragma omp for
   {
      ...
   }

   #pragma omp for
   {
      ...
   }
}
```

- 使用 `parallel for/do` 构造，而不是嵌套在 `parallel` 构造中的工作共享 `for/do` 构造。例如，以下构造效率不高：

```
#pragma omp parallel
{
   #pragma omp for
   {
      ... statements ...
   }
}
```

此构造更有效：

```
#pragma omp parallel for
{
   ... statements ...
}
```

- 如有可能，合并并行循环以避免并行开销。例如，合并两个 `parallel for` 循环：

```
#pragma omp parallel for
for (i=1; i<N; i++)
   {
     ... statements 1 ...
   }
```

```
#pragma omp parallel for
for (i=1; i<N; i++)
   {
     ... statements 2 ...
   }
```

合并后的一个 parallel for 循环更高效：

```
#pragma omp parallel for
for (i=1; i<N; i++)
   {
     ... statements 1 ...
     ... statements 2 ...
   }
```

- 使用 the OMP_PROC_BIND 或 SUNW_MP_PROCBIND 环境变量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 章 处理器绑定（线程关联性）。
- 尽可能使用 master，而不是 single。
  - master 指令作为不带隐式屏障的 if 语句实现：if (omp_get_thread_num() == 0) {...}
  - single 构造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。此外，如果未指定 nowait，则存在一个隐式屏障，这样效率较低。
- 选择适当的循环调度。
  - static 循环调度不要求同步，在数据与高速缓存相符的情况下可保持数据局域性。但是，static 调度可能导致负载不平衡。
  - 由于 dynamic 和 guided 循环调度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（） 函数，以支持动态数组、可分配数组、向量化内部函数等。标准 C 库 libc.so 中的线程安全 malloc（） 和 free（） 具有由内部锁定造成的高同步开销。可以在其他库（如 libmtmalloc.so 库）中找到更快的版本。指定 -lmtmalloc 与 libmtmalloc.so 链接。
- 如果数据集较小，可能会导致 OpenMP 并行区域性能不佳。在 parallel 构造中使用 if 子句指定区域仅应在预期可以提高一定性能的情况下并行运行。
- 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。不过，使用嵌套并行操作时要非常谨慎，它会增加同步开销，因为每个嵌套并行区域的线程组均需在屏障处进行同步。另外，嵌套并行操作可能会占用过多的计算机资源，从而导致性能下降。
- 使用 lastprivate 时要非常谨慎，因为它有可能导致很高的开销。

- 在从区域返回之前，必须将数据从线程的专用内存复制到共享内存。
- `lastprivate` 会增加额外的检查。例如，包含 `lastprivate` 子句的工作共享循环的编译代码会检查哪个线程执行顺序中的上一个迭代。这会在循环中每个块的末尾增加额外的工作，如果有许多个块，这些额外的工作还会累加。

- 使用显式 `flush` 时要非常谨慎。刷新将造成数据存储到内存，而随后的数据访问可能需要从内存重新加载，这些都会降低效率。

8.2 避免伪共享

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？

在大多数共享内存多处理器计算机中，每个处理器都有自己的本地高速缓存。高速缓存充当着慢速内存与处理器的高速寄存器之间的缓冲区。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。对同一内存位置或其周围位置的后续引用将在高速缓存外满足，直至系统决定有必要在高速缓存和内存之间保持一致性。

当不同处理器上的线程修改驻留在同一缓存代码行上的变量时，就会发生伪共享。这种情况称为伪共享（与真正意义上的共享加以区分），因为线程并没有访问相同的变量，而是访问正好驻留在同一缓存代码行上的不同变量。

当线程修改其高速缓存中的变量时，该变量所在的整个缓存代码行将被标为无效。如果另一个线程尝试访问同一缓存代码行上的变量，则修改后的缓存代码行将写回到内存，线程将从内存中获取缓存代码行。这是因为基于缓存代码行保持高速缓存一致性，而不是针对单个变量或元素。发生伪共享后，线程将被强制从内存中获取缓存代码行的最新副本，即使它尝试访问的变量并未修改。

如果经常发生伪共享，互连流量将增加，而 OpenMP 应用程序的性能和可伸缩性则会显著下降。在出现以下所有情况时，伪共享会使性能下降。

- 多个线程修改共享数据
- 多个线程修改同一缓存代码行中的数据
- 数据修改频率非常高（如在紧凑循环中）

请注意，访问只读共享数据不会发生伪共享。

8.2.2 减少伪共享

当访问特定变量的开销似乎特别高时，通常会检测到伪共享。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，可以使用以下方法减少伪共享：

- 尽可能使用专用或线程专用数据。
- 利用编译器的优化功能来消除内存负载和存储。
- 填充数据结构，以使每个线程的数据都驻留在不同的缓存代码行上。填充的大小取决于系统，等于将线程的数据推入到另一缓存代码行所需的大小。
- 修改数据结构，以减少线程之间的数据共享。

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，为每个块中的每个线程分配更多的工作（通过更改 *chunk_size* 值），也可以减少伪共享。

8.3 Oracle Solaris OS 调优功能

Oracle Solaris 操作系统支持可能提高 OpenMP 程序性能的功能。这些功能包括内存定位优化 (Memory Placement Optimization, MPO) 和多页大小支持 (Multiple Page Size Support, MPSS)。

8.3.1 内存定位优化

共享内存多处理器计算机包含多个处理器。每个处理器均可访问计算机中的所有内存。在某些共享内存多处理器中，内存体系结构使每个处理器访问某些内存区域的速度快于访问其他区域的速度。因此，向访问内存的处理器分配其附近的内存，可以缩短延迟和提高应用程序性能。

Oracle Solaris 操作系统引入了地址组 (locality group, `lgroup`) 摘要，它属于 MPO 功能的一部分。`lgroup` 是一组类似处理器和类似内存的设备，在此类设备中，组中的每个处理器都可在有限的延迟间隔内访问该组中的任何内存。库 `liblgrp.so` 导出应用程序的 `lgroup` 摘要，以便进行观察和性能调优。应用程序可以使用 `liblgrp.so` API 执行以下任务：

- 遍历组分层结构
- 搜索给定 `lgroup` 的内容和特征
- 影响 `lgroup` 的线程和内存定位

缺省情况下，Oracle Solaris 操作系统尝试从线程的主 `lgroup` 为线程分配资源。例如，缺省情况下，操作系统会尝试调度线程在线程主 `lgroup` 中的处理器上运行，并在线程的主 `lgroup` 中分配线程的内存。

以下机制可用于发现和影响与 lgroup 有关的线程和内存定位方式：

- meminfo（） 系统调用可用于发现内存定位方式。
- lgrp_home（） 函数可用于发现线程定位方式。
- 通过设置线程与给定 lgroup 的关联性，lgrp_affinity_set（） 函数可以影响线程和内存定位方式。
- 标准 C 库中的 madvise（） 函数用于提醒操作系统：用户虚拟内存的区域遵守特定的使用模式。传递给 madvise（） 的 MADV_ACCESS 标志用于影响 lgroup 之间的内存分配。例如，使用 MADV_ACCESS_LWP 标志调用 madvise（） 将提醒操作系统：下一个访问指定地址范围的线程将是访问内存区域最多的线程。操作系统将放置与此范围对应的内存并会相应地放置线程。

有关 lgroup API 的更多信息，请参阅《Oracle Solaris 11.3 Programming Interfaces Guide》 中的 第 4 章, “Locality Group APIs”。有关 madvise（） 函数的更多信息，请参见 madvise(3C) 手册页。

8.3.2 多页大小支持

Oracle Solaris 中的多页大小支持 (Multiple Page Size Support, MPSS) 功能允许应用程序对不同的虚拟内存区域使用不同的页面大小。使用 pagesize 命令可以获取特定平台的缺省页面大小。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。

转换后备缓冲器 (Translation Lookaside Buffer, TLB) 是一种数据结构，用于将虚拟内存地址映射到物理内存地址。访问 TLB 中没有虚拟-物理映射信息的内存，会产生一些性能损失。页面大小越大，TLB 使用固定数量的 TLB 项就可以映射越多的物理内存。因此，较大的页面会降低虚拟-物理内存映射的开销，从而提高整体系统性能。

更改应用程序的缺省页面大小的方法有多种：

- 使用 Oracle Solaris 命令 ppgsz(1)。
- 使用 -xpagesize、-xpagesize_heap 或 -xpagesize_stack 选项编译应用程序。有关详细信息，请参见 cc(1)、CC(1) 或 f95(1) 手册页。
- 预装入 mpss.so.1 共享对象，可以使用环境变量设置页面大小。有关详细信息，请参见 mpss.so.1(1) 手册页。

♦ ♦ ♦ 第 9 章

9 OpenMP 实现定义的行为

本章介绍在使用 Oracle Developer Studio 编译器编译程序时某些 OpenMP 功能如何运行。有关在 OpenMP 4.0 规范中描述为实现定义的行为的汇总信息，请参见该规范的附录 D。

9.1 OpenMP 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。某些变量可能未对齐或者对齐方式未知。有时，使用多个 load（或 store）操作会让代码序列的运行速度更快。因此，编译器和运行时系统可能需要使用多个 load（或 store）操作来访问变量。

当内存更新针对位域变量时，该内存更新还可以读取和写回相邻变量（属于另一个变量的一部分，例如数组或结构元素）的最小大小与基本语言的要求相同。当内存更新针对非位域变量的变量时，该更新将不读取和写回属于另一个变量一部分的相邻变量，例如数组或结构元素。

9.2 OpenMP 内部控制变量

实现定义了以下内部控制变量：

- *bind-var*：控制线程的位置绑定。*bind-var* 的初始值为 `FALSE`。
- *default-device-var*：控制缺省目标设备。*default-device-var* 的初始值为 0（主机设备）。
- *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）。
- *dyn-var*：控制是否为遇到的 `parallel` 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）。
- *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。

- *nthreads-var*：控制为遇到的并行区域请求的线程数。*nthreads-var* 的初始值等于核心数，上限为 32。
- *place-partition-var*：控制遇到的并行区域的执行环境可用的位置分区。*place-partition-var* 的初始值为 `cores`。
- *run-sched-var*：控制 `schedule(runtime)` 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）。
- *stacksize-var*：控制 OpenMP 实现创建的线程（也称为辅助线程）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）。
- *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。
- *wait-policy-var*：控制等待线程的所需行为。*wait-policy-var* 的初始值为 `PASSIVE`。

9.3 线程数的动态调整

该实现能够动态调整线程数。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 *false* 参数调用 `omp_set_dynamic（）` 例程，可以禁用动态调整。

当线程遇到 `parallel` 构造时，此实现提供的线程数将根据 OpenMP 4.0 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。

如果实现无法提供请求数量的线程，并且已启用线程数动态调整，则程序执行将采用较小的线程数继续进行。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。

如果实现无法提供请求数量的线程，并且已禁用线程数动态调整，则程序将发出错误消息并停止执行。

9.4 OpenMP 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）。

9.5 OpenMP 构造

`sections`	`sections` 构造中的结构化块以 `static`（不指定块大小）方式分配给组中的线程，从而使每个线程获得的连续结构化块数量大致相等。
`single`	遇到 `single` 构造的第一个线程将会执行该构造。
`atomic`	此实现通过使用一个名为 `critical` 的特殊构造封闭目标语句或结构化块来处理所有 `atomic` 指令。此操作会强制在程序中的所有原子区域之间进行独占访问，无论这些区域是否更新相同或不同的内存位置。

9.6 处理器绑定（线程关联性）

OpenMP 4.0 规范将术语处理器定义为实现定义的硬件单元，在处理器上可以执行一个或多个 OpenMP 线程。

在该实现中，术语处理器定义为可以调度、绑定和执行一个或多个 OpenMP 线程的最小硬件执行单元，如 `processor_bind`(2) Oracle Solaris 手册页中所述。处理器的同义词包括 CPU、虚拟处理器和硬件线程。为了清楚起见，在本手册中统一使用术语硬件线程。

在该实现中，与 `OMP_PLACES` 环境变量一起使用的抽象名称线程、核心和插槽的精确定义如下所示：

- 线程指计算机上的硬件线程。
- 核心指计算机上的物理核心。
- 插槽指计算机上的物理插槽（处理器芯片）。

有关更多信息，请参见第 5.2 节 “`OMP_PLACES` 和 `OMP_PROC_BIND`” [52]。Oracle Developer Studio 与 OpenMP 4.0 线程关联性相关的实现定义行为如下所示：

- 如果采用 *close* 线程绑定策略，当 T > P 并且 P 不能整除 T 时，向位置分配线程的方法如下所示：首先，为 P 个位置中的每个位置分配 S = floor(T/P) 个线程；分配给一个位置的线程 ID 是组中线程 ID 的连续子集。其次，向前 T - (P*S) 个位置（从父线程的位置开始，并采用绕回方法）中的每个位置分配一个额外的线程。
- 如果采用 *spread* 线程绑定策略，当 T > P 并且 P 不能整除 T 时，向子分区分配线程的方法如下所示：首先，向 P 个子分区中的每个分区分配 S = floor(T/P) 个线程；分配给一个子分区的线程 ID 是组中线程 ID 的连续子集。其次，向前 T - (P*S) 个子分区（从包含父线程位置的子分区开始，并采用绕回方法）中的每个子分区分配一个额外的线程。
- 如果无法完成关联性请求，将以非零状态退出进程。
- 在 `OMP_PLACES` 环境变量中指定的数字指硬件线程 ID。

- 通过将编号 *n* 附加到抽象名称来创建 *n* 个元素的位置列表时，该位置列表将由 *N* 个连续资源组成，从构造位置列表时正在执行主线程的硬件线程所在的资源开始，在到达最后一个可用命名资源后将发生绕回分配。
- 如果请求的资源超过计算机上可用的资源，将发出错误消息，并以非零状态退出进程。如果一个资源至少包含一个联机硬件线程，则该资源可用。
- 当执行环境无法将 `OMP_PLACES` 列表中的数值（显式定义或从某个间隔隐式派生）映射到目标平台上的硬件线程时，或者如果它映射到不可用的硬件线程，将发出错误消息，并以非零状态退出进程。
- 当使用抽象名称定义 `OMP_PLACES` 环境变量时，该抽象名称表示的每个资源单元将分配为一个位置。可通过计数 *n*（其值不大于计算机上的可用单元总数）指定分配的单元数。在 Oracle Solaris 平台上，管理员使用 `psrset`(1M) 优先保留的硬件线程被视为不可用。如果 `OMP_PLACES` 定义的集中没有可用的硬件线程，将发出错误消息并以非零状态退出进程。
- 如果无法实现并行构造的关联性请求（例如，由于将 OpenMP 线程绑定到硬件线程的系统调用失败），则产生的行为不明确。
- 当使用 `OMP_PLACES` 时，可以使用间隔指定位置。该实现假定，当间隔指定位置序列时，*length* 是该序列中的位置数，而 *stride* 是分隔序列中连续位置的硬件线程 ID 数。如果未指定 *stride* 值，将采用单元距值。

9.7 Fortran 问题

本节中描述的问题仅适用于 Fortran。

9.7.1 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 threadprivate 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”。

9.7.2 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。仅当以下三个有关实际参数的条件成立时，才可以使用中间的临时存储：

1. 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

2. 实际参数为以下参数之一：
   - 数组段
   - 带有向量下标的数组段
   - 假定形状数组
   - 指针数组
3. 该实际参数的关联伪参数是显式形状数组或假定大小数组。

9.7.3 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。

索引

S

T

W

X

Y

Z