OpenMP 并行编程基础

李会民

hmli@ustc.edu.cn, lihm@qibebt.ac.cn

中国科学院青岛生物能源与过程研究所 超级计算中心

2009 年 12 月

OpenMP 编程培训内容

并行计算简介

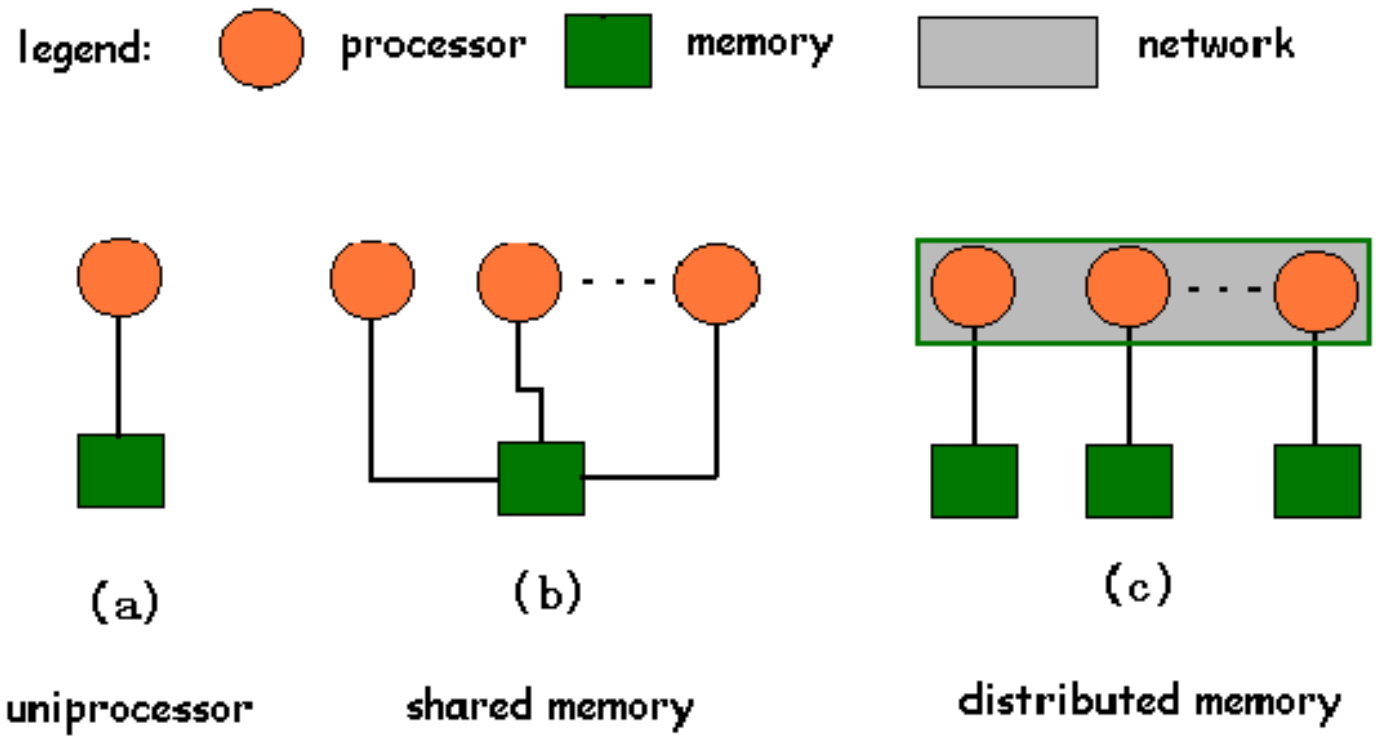
为什么要采用并行计算？

- 串行程序速度提升缓慢
- 可以加快速度
- 可以加大规模

并行计算设计的分类

- 共享内存：OpenMP
  - 多个处理器共享同一内存
  - ccNUMA(Cache-Coherent Non-Uniform Memory Access)、SMP(Symmetric MultiProcessing)
- 分布式内存：MPI
  - 每个处理器都有自己的内存
  - 处理器之间通过传递消息交换信息
  - Cluster、MPP(Massively Parallel Processing)

三种计算模型


legend:
processor
memory
network
(a)
(b)
(c)
uniprocessor
shared memory
distributed memory

并行化分解方法

- 任务分解：多任务并发执行
- 功能分解：分解被执行的计算
- 区域分解：分解被执行的数据

什么是 OpenMP

- OpenMP 是通过在源代码中添加一些 OpenMP 编译指令、调用 OpenMP 库函数来实现在共享内存的系统上并行运行的一种标准
- OpenMP 提供给共享内存并行程序员一种简单灵活的用于开发并行应用的接口模型，使程序既可以运行在台式机，又可以运行在超级计算机上，并具有良好可移植性和可缩放性等
  - 未打开 OpenMP 编译选项的编译器将忽略 OpenMP 编译指令而直接编译成串行可执行程序
  - 打开 OpenMP 编译选项的将对 OpenMP 编译指令进行处理编译成 OpenMP 并行可执行程序
  - 运行时的并行线程数可以在程序启动时利用环境变量等动态设置
  - 支持与 MPI 的混合编程，在节点内部运行 OpenMP 并行，节点之间运行 MPI 并行
- OpenMP 支持 C/C++ 和 Fortran 语言
- OpenMP 程序可以运行在 Unix、Linux 和 Windodws 等操作系统上

OpenMP 参与的主要公司、机构和人员

OpenMP Architecture Review Board(ARB) 永久成员

- AMD (David Leibs)
- Cray (James Beyer)
- Fujitsu (Matthijs van Waveren)
- HP (Uriel Schafer)
- IBM (Kelvin Li)
- Intel (Sanjiv Shah)
- NEC (Kazuhiro Kusano)
- The Portland Group, Inc. (Michael Wolfe)
- SGI (Lori Gilbert)
- Sun Microsystems (Nawal Copty)
- Microsoft (-)

ARB 辅助成员

- ASC/LLNL (Bronis R. de Supinski)
- cOMPunity (Barbara Chapman)
- EPCC (Mark Bull)
- NASA (Henry Jin)
- RWTH Aachen University (Dieter an Mey)

官方站点：http://www.openmp.org

OpenMP 发布历史

- Version 3.0 Complete Specifications - (May, 2008)
- Version 2.5 - (May 2005, combined C/C++ and Fortran)
- C/C++ version 2.0 - (March 2002)
- C/C++ version 2.0 with change bars reflecting changes from 1.0 - (March 2002)
- Fortran version 2.0 - (November 2000)
- Fortran version 2.0 with change bars reflecting changes from 1.1 (November 2000)
- C/C++ version 1.0 - (October 1998)
- Fortran version 1.1 - (November 1999 - incorporates April 1999 Interpretations and Errata)
- Fortran version 1.0 - (October 1997)

支持 OpenMP 的编译器以及对应编译参数

公司或组织	编译器	对应的编译参数
GNU	gcc(4.3.2)	-fopenmp
Intel	C/C++/Fortran(10.1)	Linux: -openmp, Win: /Qopenmp
Portland Group	C/C++/Fortran	-mp
IBM	XL C/C++/Fortran	-qsmp=omp
HP	C/C++/Fortran	+Oopenmp
Microsoft	Visual Studio 2008 C++	/openmp
Sun Microsystems	C/C++/Fortran	-xopenmp
Absoft Pro FortranMP	Fortran	-openmp
Lahey/Fujitsu Fortran95	C/C++/Fortran	–openmp –threadstack –threadheap
PathScale	C/C++/Fortran	-apo -mp

一维矢量点乘的串行代码

两个一维矢量的点乘：$sum = \mathbf{A} \cdot \mathbf{B}$

```
int main(argc,argv)
int argc;
char *argv[];
{
  double sum;
  double a[256], b[256];
  int i,n;
  n = 256;
  for (i = 0; i < n; i++) {
     a[i] = i * 0.5;
     b[i] = i * 2.0;
  }
  sum = 0;
  for (i = 1; i <= n; i++ ) {
     sum = sum + a[i]*b[i];
  }
  printf ("sum = %f\n", sum);
}
```

一维矢量点乘的 MPI 并行代码

```
#include "mpi.h"
int main(argc,argv)
int argc;
char *argv[];
{
  double sum, sum_local;
  double a[256], b[256];
  int i, n, numprocs, myid, my_first, my_last;
  n = 256;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  my_first = myid * n/numprocs;
  my_last = (myid + 1) * n/numprocs;
  for (i = 0; i < n; i++) {
     a[i] = i * 0.5;
     b[i] = i * 2.0;
  }
  sum_local = 0;
  for (i = my_first; i < my_last; i++) {
       sum_local = sum_local + a[i]*b[i];
  }
  MPI_Allreduce(&sum_local, &sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  if (myid==0) printf ("sum = %f\n", sum);
  MPI_Finalize();
}
```

一维矢量点乘的 OpenMP 并行代码

```
int main(argc,argv)
int argc; char *argv[];
{
  double sum;
  double a[256], b[256];
  int status;
  int i,n=256;
  for (i = 0; i < n; i++) {
     a[i] = i * 0.5;
     b[i] = i * 2.0;
  }
  sum = 0;
 #pragma omp parallel for reduction(+:sum)
  for (i = 1; i <= n; i++ ) {
     sum = sum + a[i]*b[i];
  }
   printf ("sum = %f \n", sum);
}
```

基础

OpenMP 是下面的集合：

- 编译指令
- 库函数
- 环境变量

OpenMP 格式：C/C++

C/C++ 中 OpenMP 利用 pragma 预处理指令做为 OpenMP 的指令，语法按照以下格式：

#pragma omp directive−name [clause[[,] clause]. . .] new−**line**

- 每个指令以 **#pragma** omp 开始，
- 指令其余部分按照 C/C++ 编译指令标准，并区分大小写
- # 之前和之后可有空白，且有时候必须用空白来分隔指令中的文字
- **#pragma** omp 之后的预处理目标在编译时将被宏替换
- 每个 OpenMP 执行指令必须应用于至少一个随后的语句，并且必须是一个结构块
- 每行只能有一个 OpenMP 指令
- 参数的位置无前后之分，需要时可以重复

OpenMP 格式：Fortran

Fortran 下的 OpenMP 指令语法按照以下格式：

sentinel directive –**name** [clause[[,] clause]...]

OpenMP 指令需要满足以下条件：

- 所有 OpenMP 编译指令必须以标志符开始
- 固定格式和自由格式的 OpenMP 标识符不同
- OpenMP 指令不区分大小写
- OpenMP 指令不能内嵌在连续的语句中，语句也不能内嵌在指令中
- 每行只能有一个 OpenMP 指令
- 参数的位置无前后之分，需要时可以重复

对于 Fortran，今后除非特殊说明，否则以自由格式表述

OpenMP 格式：Fortran 固定格式

固定格式 Fortran 的 OpenMP 指令标识符为 *!$omp*、c$omp 或 *$omp，需满足以下条件：

- 标识符必需在第一列开始，并且做为一个单词，之间不得有其它字符
- 固定格式 Fortran 的行宽、空白、续行和列规则对指令行一样需遵守
- 初始指令行必须有一个空格或 0 在第六列，指令续行时必须有一个非空格或 0 的字符在第六列
- 注释可以与指令同一行，在第六列之后的 ! 表明注释开始
- 如果指令标志符之后跟的第一个字符是非空格字符，或者指令续行是一个 !，那么此行被忽略

```
!$omp directive-name [clause[[,] clause]. . . ] new-line
c$omp directive-name [clause[[,] clause]. . . ] new-line
*$omp directive-name [clause[[,] clause]. . . ] new-line
```

以下三种表示方式等价

```
c23456789 表示列号
!$omp parallel do shared(a,b,c)

c$omp parallel do
c$omp+shared(a,b,c)

*$omp paralleldoshared(a,b,c)
```

OpenMP 格式：Fortran 自由格式

自由格式 Fortran 的 OpenMP 指令标识符为 *!$omp*，需满足以下条件：

- 标识符可在任何位置开始，但之前的字符必须只能为空白而不能为其它字符
- 必须做为一个单独的单词，之间不能有其它字符
- 自由格式 Fortran 的行宽、空白、续行和列规则对指令行一样需遵守
- 初始指令行必须在标识符后有一个空格
- 需要续行的行的最后，必须为 &，其续行可以在标识符之后有一个 &，且 & 前后可有空白
- 注释可以与指令同一行，! 表明注释开始
- 如果指令标志符之后跟的第一个字符是非空字符，或指令续行仅是一个 !，那么此行被忽略
- 一个或更多的空格或制表符用于分割邻近指令中的关键词，以下的情形空格是可选的：
  end critical、end do、end master、end ordered、end parallel、end sections、end single、end task、end workshare、parallel do、parallel sections、parallel workshare

```
!$omp directive-name [clause[[,] clause]. . . ] new-line
```

```
!23456789 表示列号
      !$omp parallel do &
            !$omp shared(a,b,c)

      !$omp parallel &
     !$omp&do shared(a,b,c)

!$omp paralleldo shared(a,b,c)
```

条件编译

对于支持预处理的 OpenMP 实现，宏 _OPENMP 的值被定义为 yyyymm 格式，其中 yyyy 和 mm 分别为此编译器使用的 OpenMP 标准发布的年和月，如此宏另由 #define 和 #undef 声明，那么编译行为将无法预料，因此尽量不要自己定义此宏。

条件编译：Fortran 固定格式

标志符可为 *!$*、**$* 或 c$ 之一，并需要满足以下条件：

- 标记符必需从第一列开始，并且是做为一个之间没有空白的整体
- 标记符被替换为两个空格之后，初始行必须在第六列有一个空格或者 0，并且在第一到第五列之间只能为空白或数字
- 标记符被替换为两个空格之后，续行必须在第六列有一个非空白或者 0 的字符，并且在第一到第五列之间只能为空白

如果上述满足，那么编译时此行的标识符将被将替换为两个空格，否则将不做处理

条件编译：Fortran 固定格式

```
!23456789 表示列号
!$      interval=l*omp_get_thread_num() / &
!$      (omp_get_num_threads()-1)
!下行的 !$ 由于的前面有非空格字符而不表示 OpenMP 条件编译，而是注释
      Do i=1,100, !$ OMP_get_num_threads()
```

以下两种方式等价：

```
!23456789 表示列号
!$ 10 iam = omp_get_thread_num() +
!$    &           index

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

条件编译：Fortran 自由格式

标志符为 *!$*，并需满足以下条件：

- 以 *!$* 而不以 *!$omp* 为开始标记，且前面没有空格以外的其它字符
- 标记符是做为一个之间没有空白的整体
- 初始行必须在标识符之后有一个空格
- 需要续行的行的最后必须为 &，续行可在标识符之后有一个 &，& 前后可有空白

如果上述满足，那么编译时此行的标识符将被将替换为两个空格，否则将不做处理

条件编译：Fortran 自由格式

```
!$interval=l*omp_get_thread_num() / &
!$(omp_get_num_threads()-1)
!下行的 !$ 由于的前面有非空格字符而不表示 OpenMP 条件编译，而是注释
Do i=1,100, !$ OMP_get_num_threads()
```

以下两种方式等价

```
c23456789
 !$ iam = omp_get_thread_num() + &
 !$&      index

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

内在控制变量（ICV）

内在控制变量（ICV）可是内置的用于控制 OpenMP 程序行为的变量，比如存储线程数、线程号等信息

- 影响并行块的内在控制变量：
  - dyn-var：控制影响的并行区域是否允许动态调整线程数。每任务一个 ICV 副本。
  - nest-var：控制影响的并行区域是否允许嵌套并行。每任务一个 ICV 副本。
  - nthreads-var：控制影响的并行区域的线程数。每任务一个 ICV 副本。
  - thread-limit-var：控制程序最大允许参与的线程数。整个任务共用一个 ICV 副本。
  - max-active-levels-var：控制嵌套的激活的并行区域的最大数。整个任务共用一个 ICV 副本。
- 影响循环区域操作的内在控制变量：
  - run-sched-var：控制循环区域的实时调度策略。每任务一个 ICV 副本。
  - def-sched-var：控制循环区域的默认实时调度策略。整个任务共用一个 ICV 副本。
- 影响程序执行的内在控制变量：
  - stacksize-var：控制 OpenMP 实现产生的线程的堆栈（stack）的大小。整个任务共用一个 ICV 副本。
  - wait-policy-var：控制等待线程的期望行为。整个任务共用一个 ICV 副本。

更改和获取内在控制变量的值

内在控制变量	作用范围	变量值的设置方式	获取值	初始值
dyn-var	单个任务	OMP_DYNAMIC omp_set_dynamic()	omp_get_dynamic()	参看注解
nest-var	单个任务	OMP_NESTED omp_set_nested()	omp_get_nested()	false
nthreads-var	单个任务	OMP_NUM_THREADS omp_set_num_threads()	omp_get_max_threads()	执行时定义
run-sched-var	单个任务	OMP_SCHEDULE omp_set_schedule()	omp_get_schedule()	执行时定义
def-sched-var	全局	(none)	(none)	执行时定义
stacksize-var	全局	OMP_STACKSIZE	(none)	执行时定义
wait-policy-var	全局	OMP_WAIT_POLICY	(none)	执行时定义
thread-limit-var	全局	OMP_THREAD_LIMIT	omp_get_thread_limit()	执行时定义
max-active-levels-var	全局	OMP_MAX_ACTIVE_LEVELS omp_set_max_active_levels()	omp_get_max_active_levels()	参看注解

注解：

- 如 OpenMP 的执行支持动态调整线程，dyn-var 的初始值由执行时定义，否则为 false
- max-active-levels-var 的初始值是执行时支持的并行级别数目，详细信息参考手册。

程序开始运行时，变量的初始值将被赋予，并会在 OpenMP 结构或 API 子程序中执行，用户设置的 OpenMP 环境变量的值被读取并赋予对应的内部控制变量，之后任何 OpenMP 环境变量的改变都不会再影响内部控制变量。OpenMP 结构的参数不会影响任何内部控制变量。

各任务的内在控制变量的工作方式

各任务区域具有初始变量 dyn-var、nest-var、nthreads-var 和 run-sched-var 自己的副本。当一个任务结构或并行结构在一个任务中执行时，其产生的子任务继承这些变量。调用 omp_set_num_threads()、omp_set_dynamic()、omp_set_nested() 和 omp_set_schedule() 仅仅会更新与其相关的任务区域的内在控制变量。

当执行到指定 schedule(runtime) 的循环工作共享区域时，所有组成绑定 parallel 区域的任务区域的 run-sched-var 必须有同样的值，否则行为将无法预料。

优先级

结构参数	API 子程序	环境变量	内在控制变量
(none)	omp_set_dynamic()	OMP_DYNAMIC	dyn-var
(none)	omp_set_nested()	OMP_NESTED	nest-var
num_threads	omp_set_num_threads()	OMP_NUM_THREADS	nthreads-var
schedule	omp_set_schedule()	OMP_SCHEDULE	run-sched-var
schedule	(none)	(none)	def-sched-var
(none)	(none)	OMP_STACKSIZE	stacksize-var
(none)	(none)	OMP_WAIT_POLICY	wait-policy-var
(none)	(none)	OMP_THREAD_LIMIT	thread-limit-var
(none)	omp_set_max_active_levels()	OMP_MAX_ACTIVE_LEVELS	max-active-levels-va

左边优先级高右边

并行结构：parallel Construct

指明随后区域中的代码将并行执行，并且并行是可以嵌套的

- C/C++：

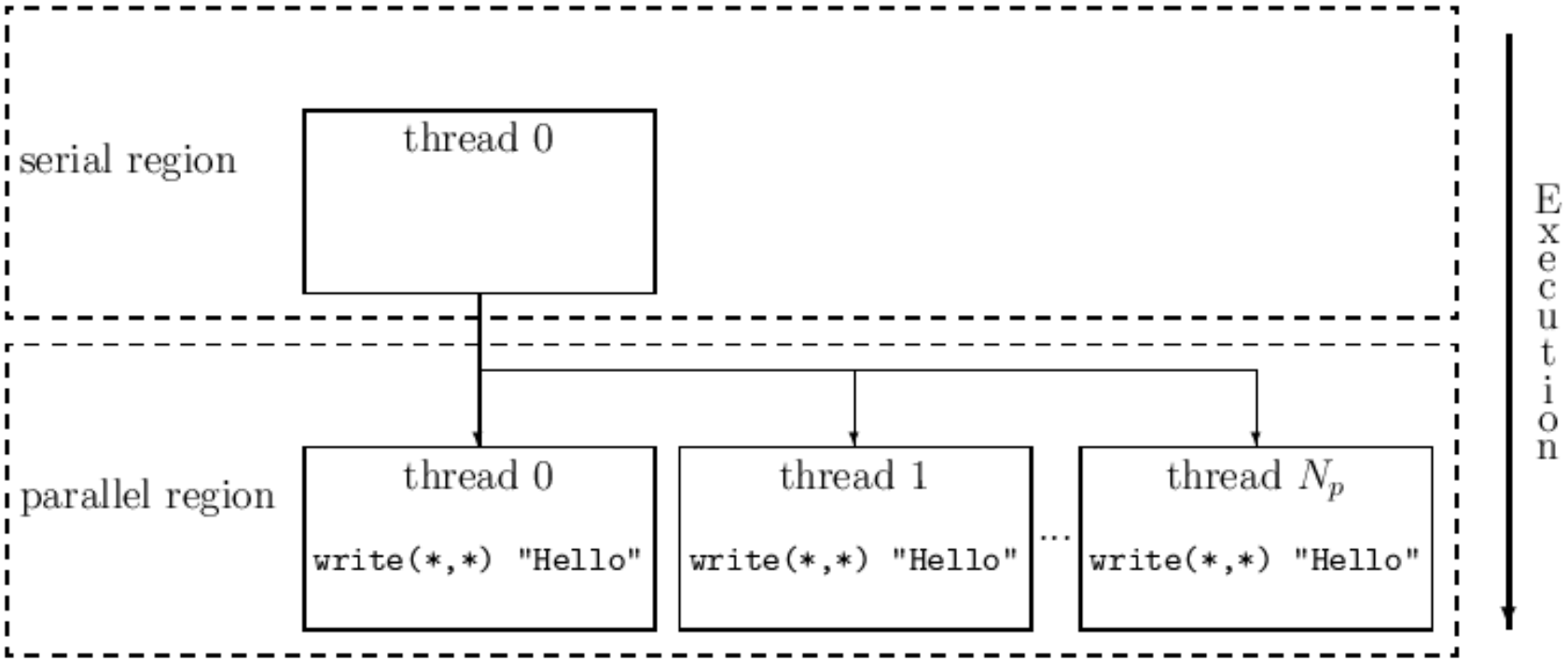
```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
structured-block
```

- Fortran：

```
!$omp parallel [clause [[,] clause ]...]
    structured-block
!$omp end parallel
```

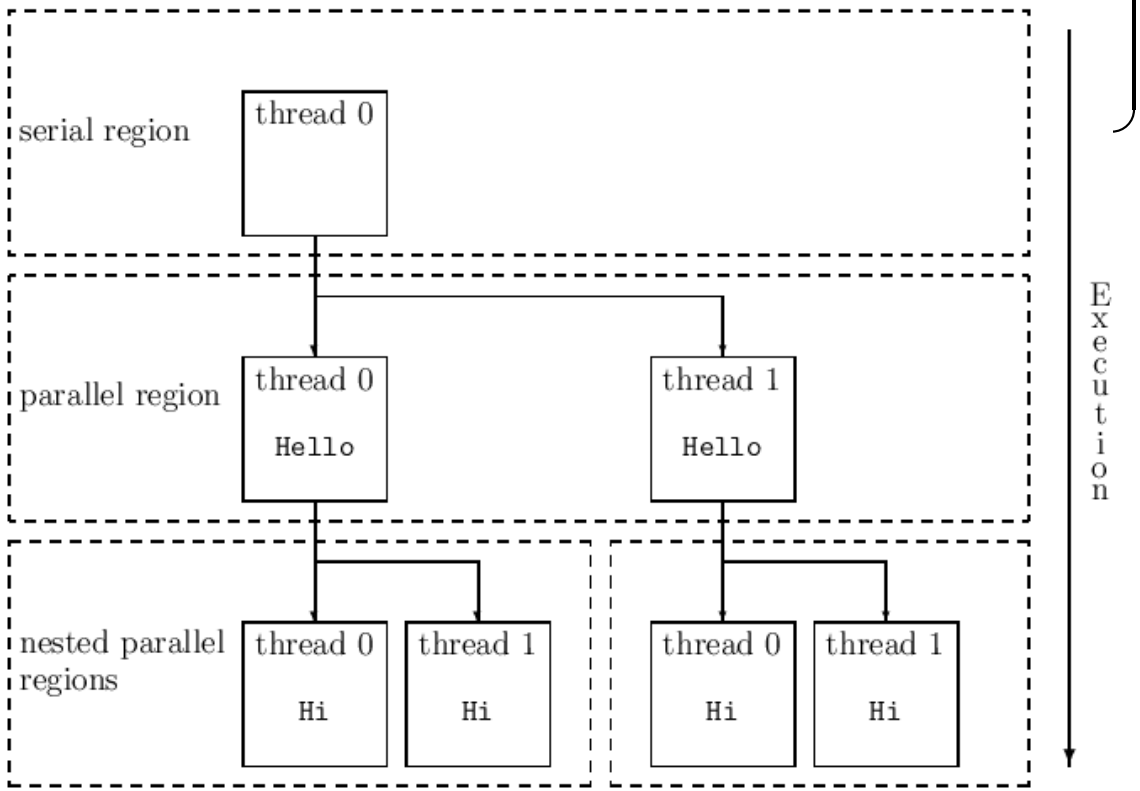
并行结构图示

```
!$omp parallel
      write(*,*) "Hello"
!$omp end parallel
```



并行嵌套图示

```
!$omp parallel
      write(*,*) "Hello"
      !$omp parallel
            write(*,*) 'Hi'
      !$omp end parallel
!$omp end parallel
```



工作共享结构：Worksharing Constructs

工作共享结构会在执行到此结构的线程中分配与之相关区域的到各线程执行。线程会执行在每个正在执行任务的情况下隐含的部分区域。在每个工作共享结构的入口处并不需进行同步，但在工作共享区域的结束处除非有 nowait 参数，否则默认会进行同步。如果有 nowait，那么执行到此处时将会忽略此工作共享区域的同步，早执行完此结构的线程会直接执行下面的指令而不需等待其它进程执行到此，也不需要进行刷新（flush）操作。OpenMP 定义了以下工作共享结构：

- 循环结构：loop structure
- 分块结构：sections structure
- 单执行结构：single structure
- 工作共享结构：workshare structure

限制：

- 工作共享区域必须为所有组内线程都执行到或都不执行到
- 一组内每个线程执行到工作共享区域和同步区域的顺序必须一致

循环结构：Loop Constructs

循环结构指明一个或多个与之相关的循环迭代将被一组线程执行

- C/C++:

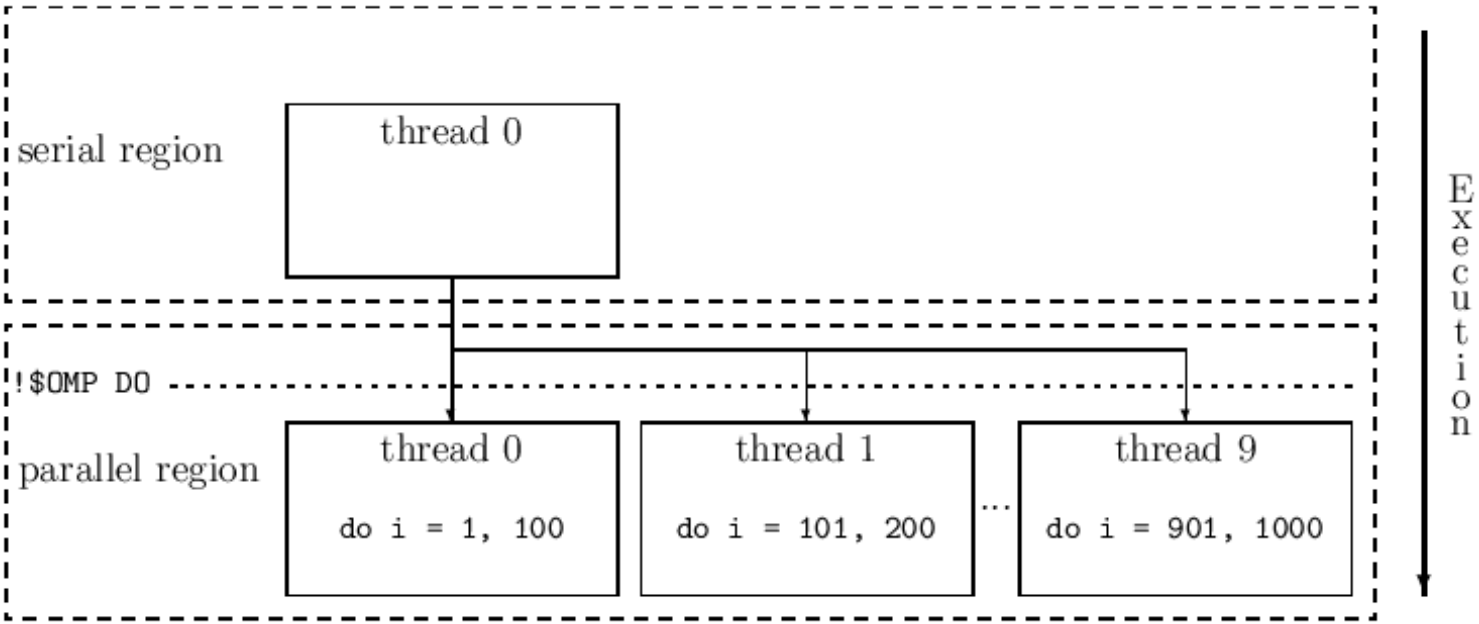
```
#pragma omp for [clause[[,] clause] ... ] new-line
    for-loops
```

- Fortran:

```
!$omp do [clause[[,] clause] ... ]
     do-loops
[!$omp end do [nowait] ]
```

循环结构图示

```
!$omp do,clause ···
do i = 1, 1000
...
enddo
!$omp end do
```



分块结构：sections Construct

不同的线程执行不同区域内的代码

- C/C++:

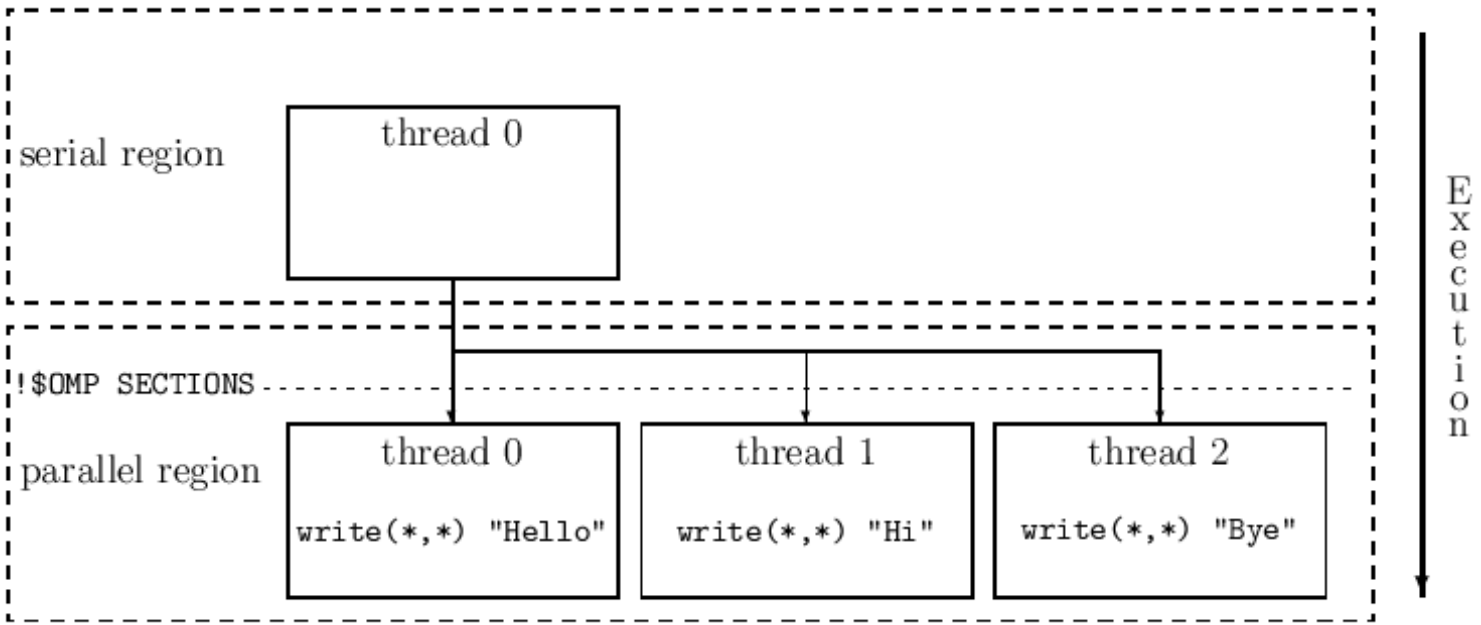
```
#pragma omp sections [clause[[,] clause] ...] new-line
   {
   [#pragma omp section new-line]
         structured-block
   [#pragma omp section new-line
         structured-block ]
     ...
   }
```

- Fortran:

```
!$omp sections [clause[[,] clause] ...]
  [!$omp section]
        structured-block
  [!$omp section
        structured-block ]
    ...
!$omp end sections [nowait]
```

分块结构图示

```
!$omp sections
    !$omp section
        write(*,*) "hello"
    !$omp section
        write(*,*) "hi"
    !$omp section
    write(*,*) "bye"
!$omp end sections
```



单执行结构：single structure

指明相关代码只能有一个线程（并不必须是雇主线程）执行，一般为先到的线程执行

- C/C++:

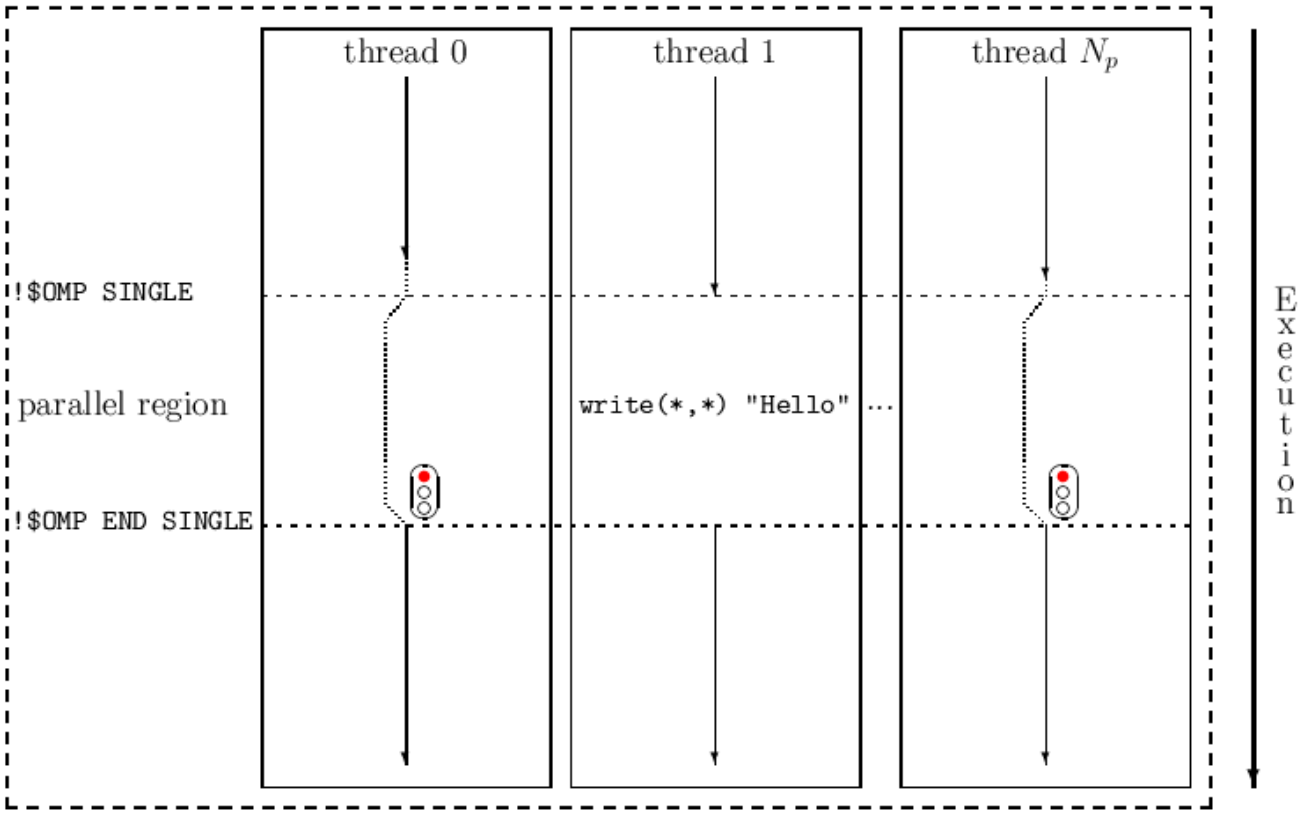
```
#pragma omp single [clause[[,] clause] ...] new-line
    structured-block
```

- Fortran:

```
!$omp single [clause [[,] clause] ...]
    structured-block
!$omp end single [end_clause[[,] end_clause] ...]
```

单执行结构图示

```
!$omp single clause1 clause2 ···
      ···
!$omp end single end_clause
```



工作共享结构：workshare Construct

指明相关的代码被分割成不同的单元以供不同线程执行，不同单元只能被某个线程执行一次

- C/C++：不支持此 OpenMP 指令
- Fortran：

```
!$omp workshare
    structured-block
!$omp end workshare [nowait]
```

structured-block 需要满足以下条件：

- 数组赋值
- 标量赋值
- forall 语句
- forall 结构
- where 语句
- where 结构
- atomic 结构
- critical 结构
- parallel 结构

critical 结构中封入的语句也受此限制，parallel 结构中封入的语句不受此限制。

联合并行工作共享结构

联合并行共享结构实际上是在并行结构中嵌套共享结构的一个快捷方式，这些指令将同时指明一个工作共享结构具有并行性，而不需其它语句另行指明。联合并行共享结构允许一些并行结构和工作共享结构都允许的特定参数。如程序含有对并行结构或工作共享结构具有不同行为的参数，那么其行为将不可预测。联合并行共享结构包含以下结构：

- 并行循环结构
- 并行分块结构
- 并行工作共享结构

并行循环结构：Parallel Loop Constructs

并行循环结构指明一个或多个相关的循环迭代将被一组线程并行执行

- C/C++：

```
#pragma omp parallel for [clause[[,] clause] ... ] new-line
    for-loops
```

- Fortran：

```
!$omp parallel do [clause [[,] clause] ... ]
      do-loops
[!$omp parallel end do [nowait] ]
```

并行分块结构：parallel sections Construct

并行分块结构指明与之相关的不同区域的代码将被不同线程并行执行

- C/C++:

```
#pragma omp parallel sections [clause[[,] clause]  ...]  new-line
   {
   [#pragma omp section new-line]
         structured-block
   [#pragma omp section new-line
         structured-block ]
     ...
   }
```

- Fortran:

```
!$omp parallel sections [clause[[,] clause]  ...]
  [!$omp section]
        structured-block
  [!$omp section
        structured-block ]
    ...
!$omp end parallel sections [nowait]
```

并行工作共享结构：parallel workshare Construct

并行工作共享结构指明相关的代码被分割成不同的单元以供不同线程并行执行，不同单元只能被某一线程执行一次

- C/C++：不支持此 OpenMP 指令
- Fortran：

```
!$omp parallel workshare
    structured-block
!$omp parallel end workshare [nowait]
```

structured-block 需要满足以下条件：

- 数组赋值
- 标量赋值
- forall 语句
- forall 结构
- where 语句
- where 结构
- atomic 结构
- critical 结构
- parallel 结构

critical 结构中封入的语句也受此限制，parallel 结构中封入的语句不受此限制。

任务结构：task Construct

任务结构定义一个明确的任务

- C/C++：

```
#pragma omp task [clause[[,] clause] ...] new-line
    structured-block
```

- Fortran：

```
!$omp task [clause [[,] clause] ...]
    structured-block
!$omp end task
```

雇主结构和同步结构

雇主结构和同步结构主要包含以下结构

- 雇主结构：master construct
- 临界结构：critical construct
- 屏障结构：barrier construct
- 任务等待结构：taskwait construct
- 原子结构：atomic construct
- 刷新结构：flush construct
- 有序结构：ordered construct

雇主结构：master construct

雇主结构指明结构块需要雇主线程（主线程）执行

- C/C++:

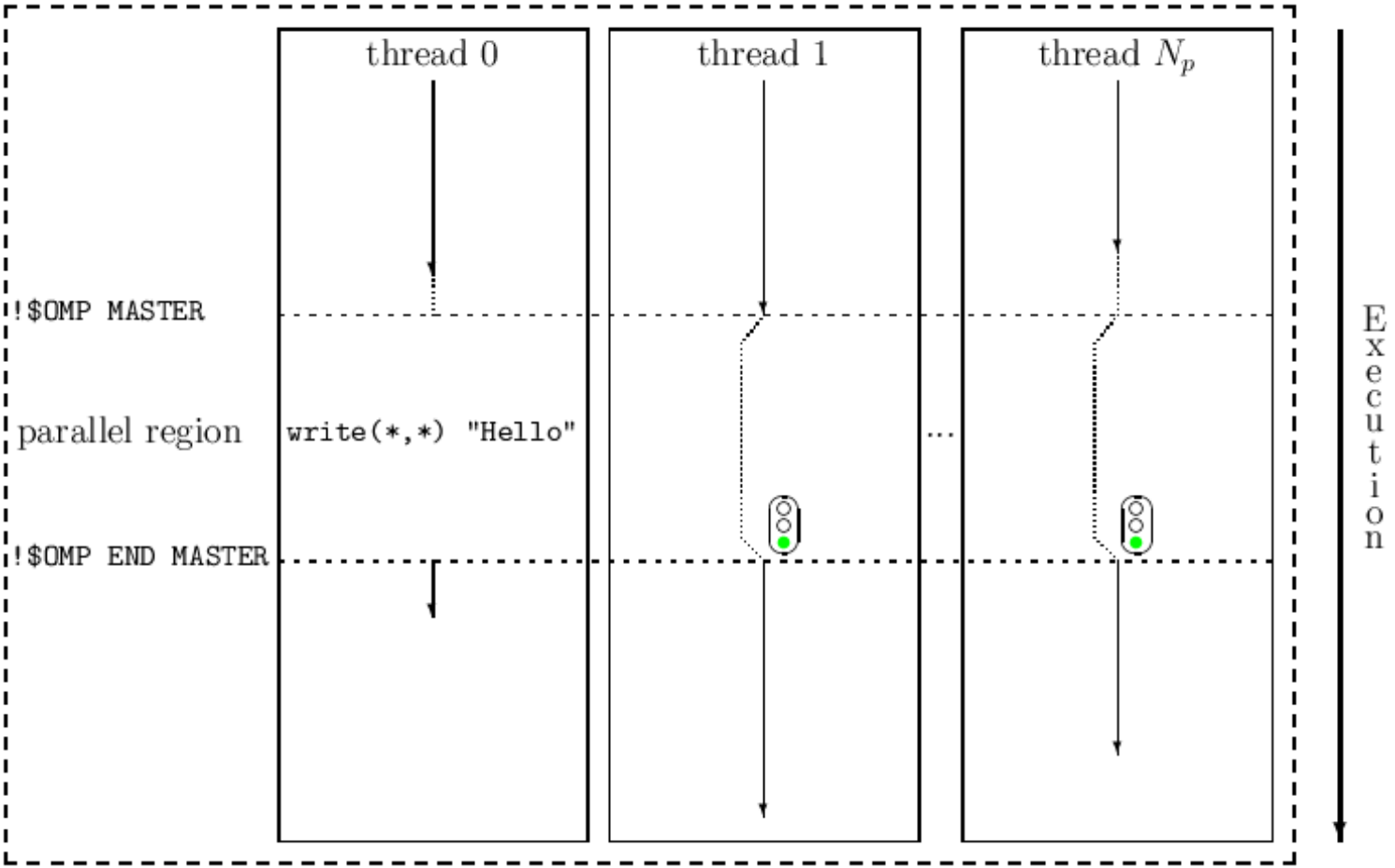
```
#pragma omp master new-line
    structured-block
```

- Fortran:

```
!$omp master
    structured-block
!$omp end master
```

雇主结构图示

```
!$omp master
      write(*,*) "Hello"
!$omp end master
```



临界结构：critical construct

临界结构指明结构块在同一时间只能有一个线程执行

- C/C++：

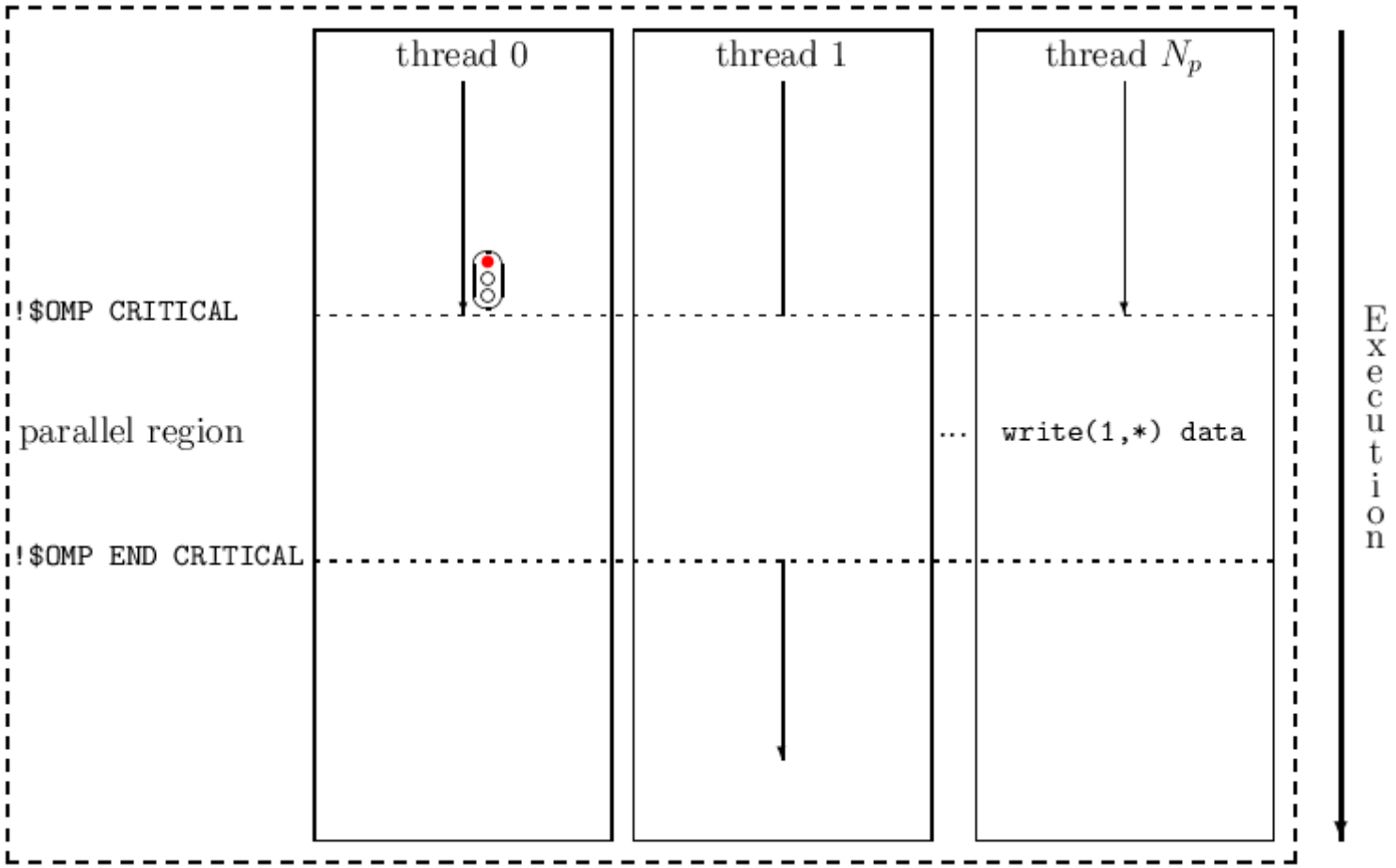
```
#pragma omp critical [(name)] new-line
    structured-block
```

- Fortran：

```
!$omp critical [(name)]
    structured-block
!$omp end critical [(name)]
```

临界结构图示

```
!$omp critical write_file
    write(1,*) data
!$omp end critical write_file
```



屏障结构：barrier construct

屏障结构指明在此处需显式屏障，即需本组内所有线程都要运行到此后才执行后续代码

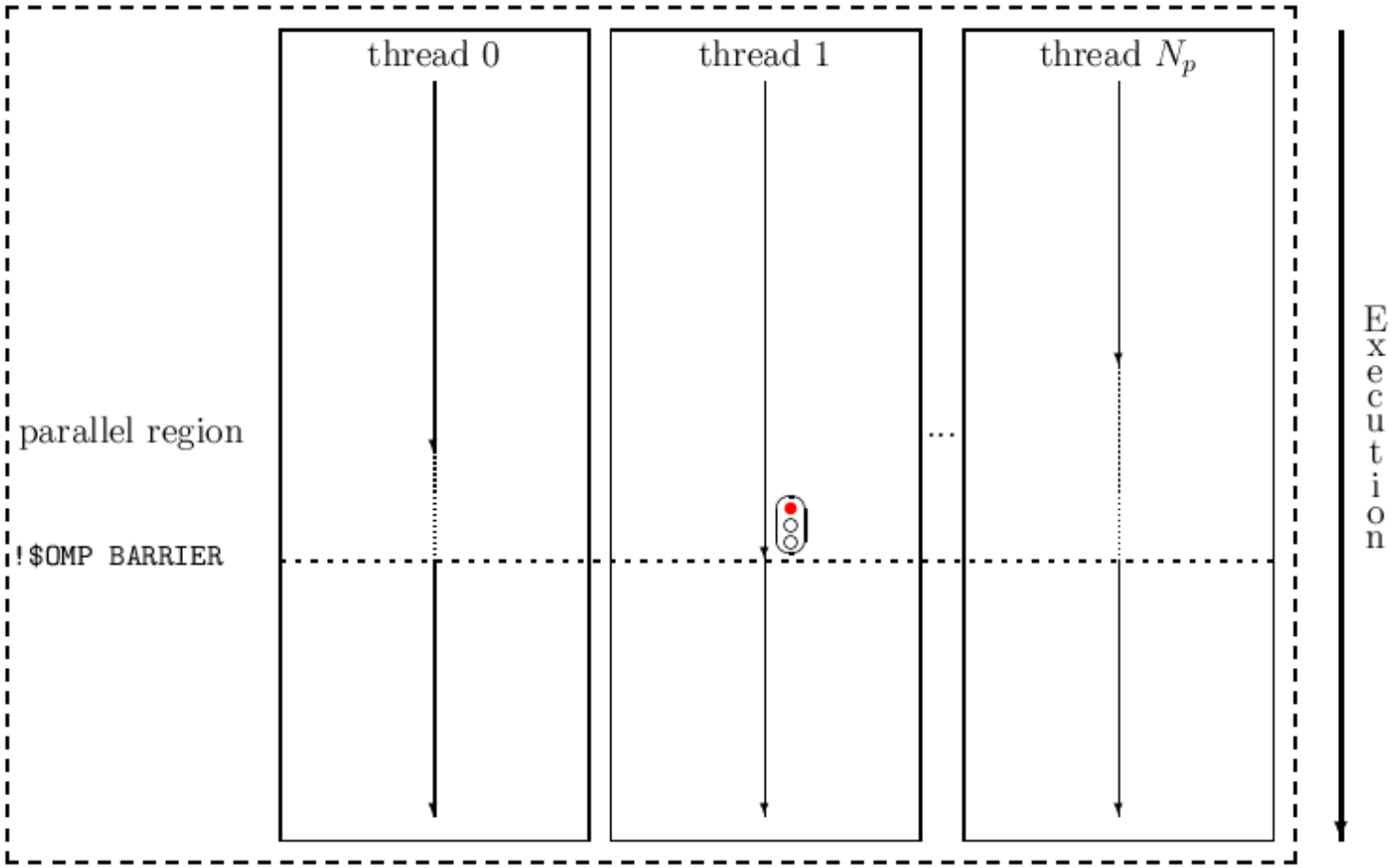
- C/C++:

```
#pragma omp barrier new-line
```

- Fortran:

```
!$omp barrier
```

屏障结构图示


thread 0
thread 1
thread N_p
parallel region
!$OMP BARRIER
...
Execution

屏障结构常见错误举例

注意：barrier 必须在所有线程的同样顺序中

常见错误举例：

```
...
!$omp critical
...
!$omp barrier
...
!$omp end critical
...
```

```
...
!$omp master
...
!$omp barrier
...
!$omp end master
...
```

```
...
!$omp single
...
!$omp barrier
...
!$omp end single
...
```

```
...
!$omp sections
...
!$omp section
...
!$omp barrier
...
!$omp end section
...
!$omp end sections
...
```

任务等待结构：taskwait construct

任务等待结构指明需要自当前任务开始就需等待产生的子任务完成

- C/C++：

```
#pragma omp taskwait newline
```

- Fortran：

```
!$omp taskwait
```

原子结构：atomic construct

原子结构用于确保指明的存储区域对某项操作进行原子更新，而不使其暴露给多个同步进行写的线程，避免同时多个线程对这些变量进行更新

- C/C++:

```
#pragma omp atomic new-line
    expression-stmt
```

- Fortran:

```
!$omp atomic
    statement
```

举例：

```
!$omp do
do i = 1, 1000
!$omp atomic +
    a = a + i
enddo
!$omp end do
```

刷新结构：flush construct

刷新结构指明对指定的变量执行内存刷新操作以保证线程此时看到这些变量在内存中的值一致

- C/C++:

```
#pragma omp flush [(list)] new-line
```

- Fortran:

```
!$omp flush [( list )]
```

有序结构：ordered construct

有序结构指明循环结构中结构块的执行按照循环的迭代顺序，将排序此结构块中的执行顺序，并允许结构块之外的代码可以并行执行

- C/C++:

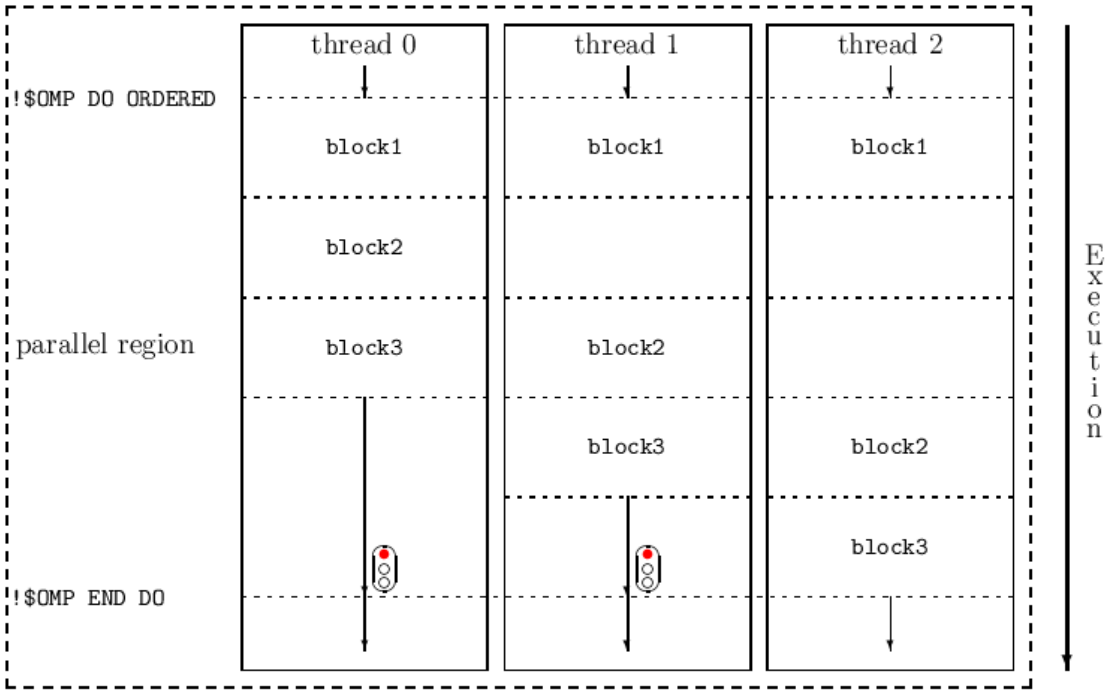
```
#pragma omp ordered new-line
    structured-block
```

- Fortran:

```
!$omp ordered
    structured-block
!$omp end ordered
```

有序结构图示

```
!$omp do ordered
    do i = 1, 100
        block1
        !$omp ordered
            block2
        !$omp end ordered
        block3
    enddo
!$omp end do
```



结构内引用的变量的数据共享属性法则

结构中引用变量的数据共享属性可为预先决定的、显示决定的或隐式决定的之一。封闭结构中的 firstprivate、lastprivate 或 reduction 参数声明的变量会导致此结构中变量采用隐式的引用方式，并遵守以下规则：

- C/C++：
  - threadprivate 指令指明的变量是线程私有的
  - 结构中某个范围内申明的具有自动存储周期的变量是私有的
  - 具有堆分配存储的变量是共享的
  - 静态数据成员是共享的
  - 关联的 **for** 循环或 parallel **for** 循环结构中的循环迭代变量是私有的
  - 不具有易变成员的保留常数变量是共享的
  - 结构中某个范围内声明的静态变量是共享的
- Fortran：
  - threadprivate 指令指明的变量和 **common** 块是线程私有的
  - 关联的 **do** 循环或 parallel **do** 循环结构中的循环迭代变量是私有的
  - parallel 或 task 结构中的顺序循环的循环迭代变量在封闭此循环的最内部结构中是私有的
  - 隐式 **do** 和 forall 的索引是私有的
  - Cray 指针继承其关联存储的数据共享属性

数据共享属性法则

除了在以下情形中，具有预定义数据共享属性的变量也许没有在数据共享属性参数中列出，对这些例外而言，在数据共享属性参数中允许列出一些预定义的变量，并且将取代这些变量的预先定义数据共享属性。

- C/C++：
  - **for** 循环或 parallel **for** 循环结构中的循环迭代变量可以在 private 或 lastprivate 参数中被列出
- Fortran：
  - **do** 循环或 parallel **do** 循环结构中的循环迭代变量可以在 private 或 lastprivate 参数中被列出
  - parallel 或 task 结构中的顺序循环的循环迭代变量可以在 private、firstprivate、lastprivate、shared、或 reduction 参数中列出，并且在封闭结构中受其它的限制
  - 假定大小的数组可以在 share 参数中被列出

隐式数据共享法则

具有显式决定或隐式决定的数据共享属性的变量：

- 具有显式决定的数据共享属性的变量指的是在一个给定的结构中被引用并且在此结构的数据共享参数中被列出的变量
- 具有隐式决定的数据共享属性的变量指的是在一个给定的结构中被引用，但没有被预先决定数据共享属性，并且没有在此结构的数据共享属性参数中列出的变量

隐式决定的数据共享属性规则如下：

- 在 parallel 或 task 结构中，如果存在 default 参数，那么这些变量的数据共享属性由 default 参数决定
- 在 parallel 结构中，如果没有 default 参数，那么这些变量是共享的
- 对于不是 task 的结构来说，如果没有 default 参数，那么这些变量将从此封闭上下文中继承数据共享属性
- 在 task 结构中，如果没有 default 参数，那么在所有封闭结构并一直到最内部的封闭 parallel 结构中变量被决定为共享的
- 在 task 结构中，如果没有 default 参数且数据共享属性没有被上述规则决定，那么变量是 firstprivate 的

区域而非结构中的数据共享属性规则

- C/C++:
  - 在被调用子程序中声明的静态变量在此区域中是共享的
  - 不具有易变成员且在被调用子程序中声明的保留常数变量是共享的
  - 除非是在 threadprivate 指令中出现，否则区域中在被调用子程序中引用的文件范围或者名字空间范围变量是共享的
  - 具有堆分配存储的变量是共享的
  - 除非是在 threadprivate 指令中出现，否则静态数据成员是共享的
  - 区域中被调用子程序的通过引用传递的正式参数继承关联的实际参数数据的数据共享属性
  - 区域中被调用子程序中的其它变量是私有的
- Fortran:
  - 区域中被调用子程序中声明的本地变量如有 save 属性，或数据是被初始化的，且未在 threadprivate 指令中出现的变量是共享的
  - 区域中被调用子程序中引用的 **common** 块或 **module** 中的变量是共享的，除非是在 threadprivate 指令中出现
  - 区域中被调用子程序中的哑元参数如通过引用传递，那么将继承与之关联的实际参数的数据共享属性
  - Cray 指针继承其关联存储的数据共享属性
  - 区域中被调用子程序中的隐式 **do** 和 forall 索引及声明的其它本地变量是私有的

threadprivate 指令

threadprivate 指令指明变量是线程私有的

- C/C++:

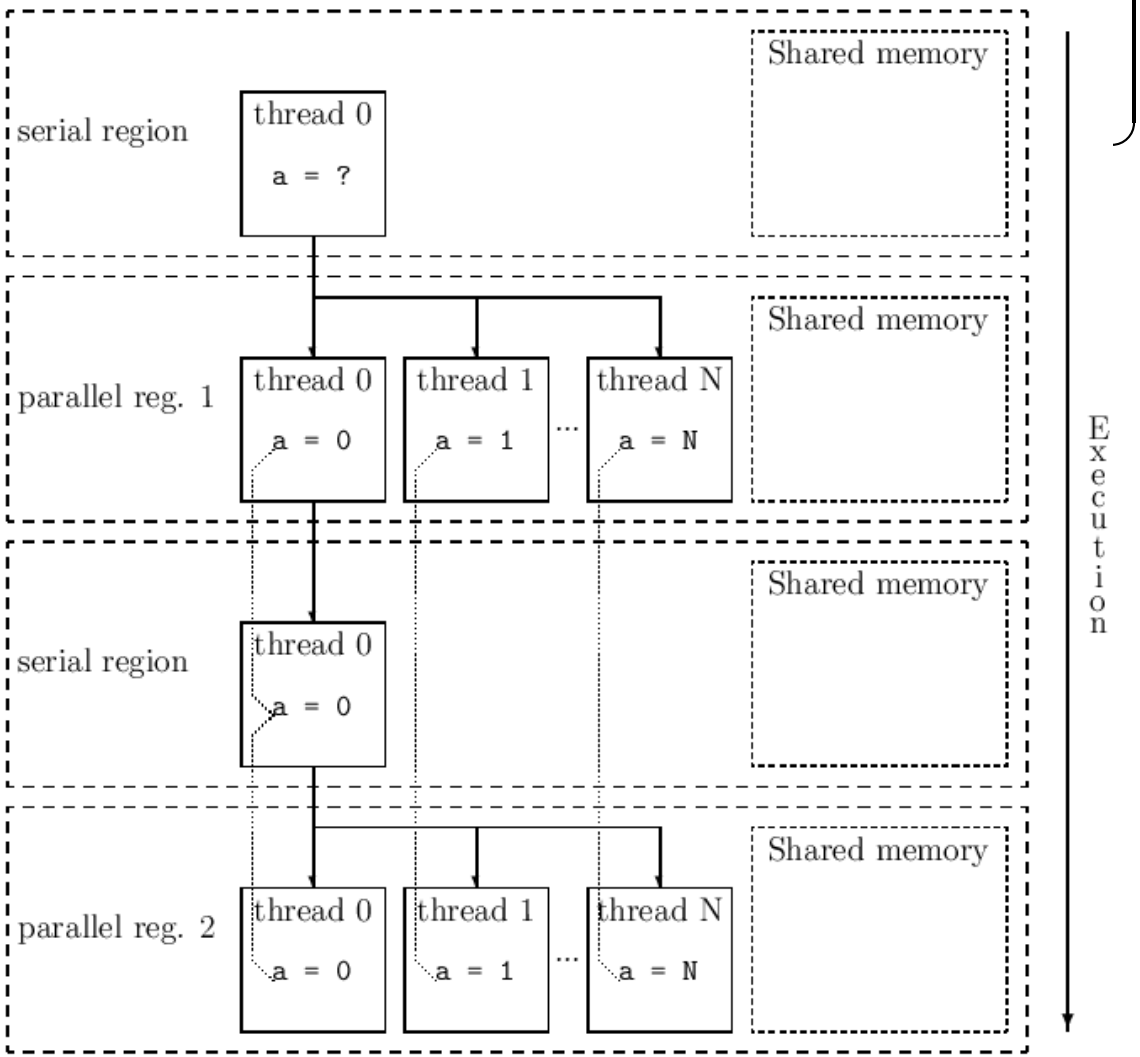
```
#pragma omp threadprivate(list) new-line
```

- Fortran:

```
!$omp threadprivate(list)
```

threadprivate 指令图示

```
real(8), save :: A(100), B
integer, save :: C
...
!$omp threadprivate(a, b, c)
```



数据共享属性参数

- default：设置默认的共享方式
  - C/C++：

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

  - Fortran：

```
default(private | firstprivate | shared | none)
```

- shared：线程共享
- private：线程私有
- firstprivate：线程私有，其值为开始此结构时从原始变量继承的值
- lastprivate：线程私有，并在此区域结束时更新原始变量的值
- reduction：声明对变量进行规约操作
  - C/C++：

```
reduction(operator: list )
```

  - Fortran：

```
reduction({operator | intrinsic_procedure_name}:list)
```

数据复制参数

- copyin：在执行 parallel 结构时，将主线程的线程私有变量的值复制给所有其它线程

```
copyin( list )
```

- copyprivate：提供一种机制可以将隐式任务数据环境的某个私有变量的值广播给此 parallel 结构中的其它数据环境

```
copyprivate( list )
```

线程嵌套

线程嵌套的规则如下

- 工作共享区域可没被嵌套封闭在工作共享、显式 task、critical、ordered 或 master 区域
- barrier 区域可没被嵌套封闭在工作共享、显式 task、critical、ordered 或 master 区域
- master 区域可没有被嵌套封闭在工作共享或显式 task 区域
- ordered 区域可没被嵌套封闭在critical 或显式 task 区域
- ordered 区域必须被嵌套封闭在具有 ordered 参数的循环区域或并行循环区域中
- critical 区域可没被嵌套（封闭或其它）在具有同样名字的 critical 区域中，此限制并不充分能防止一个死锁

运行库定义

- C/C++:
  在头文件 omp.h 中定义，此文件定义了:
  - 所有 OpenMP 子程序的原型
  - omp_lock_t 类型
  - omp_nest_lock_t 类型
  - omp_sched_t 类型
- Fortran:
  提供 F77 的头文件 omp_lib.h 和 F90 的 module 文件 omp_lib，定义了:
  - 所有 OpenMP 子程序的接口
  - **integer parameter** omp_lock_kind
  - **integer parameter** omp_nest_lock_kind
  - **integer parameter** omp_sched_kind
  - **integer parameter** openmp_version

执行环境子程序

- omp_set_num_threads：设置线程数
- omp_get_num_threads：获取线程数
- omp_get_max_threads：获取最大进程数
- omp_get_thread_num：获取进程号
- omp_get_num_procs：获取处理器数
- omp_in_parallel：判断是否在 parallel 区域
- omp_set_dynamic：设置是否允许动态调整线程数
- omp_get_dynamic：获取是否允许动态调整线程数
- omp_set_nested：设置是否允许嵌套
- omp_get_nested：获取是否允许嵌套
- omp_set_schedule：设置调度策略
- omp_get_schedule：获取调度策略
- omp_get_thread_limit：获取程序允许的最大线程数
- omp_set_max_active_levels：设置允许最大嵌套激活的层数
- omp_get_max_active_levels：获取允许最大嵌套激活的层数
- omp_get_level：获取当前嵌套 parallel 区域的层级
- omp_get_ancestor_thread_num：获取父线程或者当前线程号
- omp_get_team_size：获取某嵌套层级的父或当前线程允许的组线程数
- omp_get_active_level：获取嵌套的活跃的 parallel 区域号

锁子程序

- OpenMP 运行库包含一系列专门用途的锁子程序，可以用于同步等。这些锁子程序操作那些用 OpenMP 锁变量表示的 OpenMP 锁。OpenMP 锁只能被这些子程序存取，其余方式存取 OpenMP 锁的方式不被确认。
- OpenMP 锁可有 uninitialized、unlocked 或 locked 状态。如一个锁在 unlocked 状态，一个任务可设置该锁为 locked 状态。设置此锁的任务将拥有此锁，拥有此锁的任务可设置此锁为 unlocked 状态。一个任务复位属于另一个任务的锁是无法保证的。
- OpenMP 支持两种类型的锁：简单锁和可嵌套锁。可嵌套锁可被同一个任务在被复位之前多次设置；简单锁在属于的任务尝试再次设置的时候将不被设置。简单锁变量与简单锁相关联，且只能被传递给简单锁子程序。可嵌套锁变量与可嵌套锁相关联，并且只能传递给可嵌套锁子程序。
- 每个锁子程序存取的锁状态和所有者的约束在此子程序中描述。如果这些约束没有被执行，这些子程序的行为是非特定的。
- OpenMP 锁子程序存取锁变量通过以下方式进行：它们总是读取和更新锁变量的最当前值。锁子程序暗含更新（flush）过程；对这些锁变量值的读取和更新必须按照具有原子性（atomic）更新操作来实现，并不需要 OpenMP 包含显示更新指令来保证锁变量在不同任务中的一致性。

简单锁子程序

- omp_init_lock：初始化简单锁
- omp_destroy_lock：销毁简单锁
- omp_set_lock：设置简单锁
- omp_unset_lock：复位简单锁
- omp_test_lock：测试简单锁，如果存在则设置

嵌套锁子程序

- omp_init_nest_lock：初始化嵌套锁
- omp_destroy_nest_lock：销毁嵌套锁
- omp_set_nest_lock：设置嵌套锁
- omp_unset_nest_lock：复位嵌套锁
- omp_test_nest_lock：测试嵌套锁，如果存在则设置

omp_init_lock 和 omp_init_nest_lock

分别初始化简单锁和嵌套锁

- C/C++:

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

- Fortran:

```
subroutine omp_init_lock(svar)
integer (kind=omp_lock_kind) svar
subroutine omp_init_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

omp_destroy_lock 和 omp_destroy_nest_lock

分别销毁简单锁和嵌套锁

- C/C++:

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

- Fortran:

```
subroutine omp_destroy_lock(svar)
integer (kind=omp_lock_kind) svar
subroutine omp_destroy_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

omp_set_lock 和 omp_set_nest_lock

分别设置简单锁和嵌套锁

- C/C++:

```
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

- Fortran:

```
subroutine omp_set_lock(svar)
integer (kind=omp_lock_kind) svar
subroutine omp_set_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

omp_unset_lock 和 omp_unset_nest_lock

分别复位简单锁和嵌套锁

- C/C++:

```
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

- Fortran:

```
subroutine omp_unset_lock(svar)
integer (kind=omp_lock_kind) svar
subroutine omp_unset_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

omp_test_lock 和 omp_test_nest_lock

分别测试简单锁和嵌套锁，如果存在则设置

- C/C++:

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

- Fortran:

```
logical function omp_test_lock(svar)
integer (kind=omp_lock_kind) svar
integer function omp_test_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

时间子程序

提供计时器

- omp_get_wtime：获取以秒为单位的当前时间
  - C/C++：

```
double omp_get_wtime(void);
```

  - Fortran：

```
double precision function omp_get_wtime()
```

- omp_get_wtick：获取计时器的精度
  - C/C++：

```
double omp_get_wtick(void);
```

  - Fortran：

```
double precision function omp_get_wtick()
```

时间子函数举例

利用时间函数可以计算程序中某段计算花费的时间

- C/C++:

```
double start, end;
start = omp_get_wtime();
... work to be timed ...
end = omp_get_wtime();
printf("Work took %f seconds\n", end - start);
```

- Fortran:

```
double precision start, end
start = omp_get_wtime()
... work to be timed ...
end = omp_get_wtime()
print *, "Work took", end - start, "seconds"
```

环境变量

利用环境变量可以控制 OpenMP 程序的运行

- OMP_SCHEDULE：设置 run-sched-var 内在控制变量以控制运行调度类型和并行块大小，可设为 static、dynamic、guided 或 auto
- OMP_NUM_THREADS：设置 nthreads-var 内在控制变量以设置线程数
- OMP_DYNAMIC：设置 dyn-var 内在控制变量以设置是否允许动态调整线程
- OMP_NESTED：设置 nest-var 内在控制变量以设置是否允许嵌套
- OMP_STACKSIZE：设置 stacksize-var 内在控制变量以设置堆栈大小
- OMP_WAIT_POLICY：设置 wait-policy-var 内在控制变量以设置线程等待策略
- OMP_MAX_ACTIVE_LEVELS：设置 max-active-levels-var 内在控制变量以设置允许的最大嵌套的激活的并行区域
- OMP_THREAD_LIMIT：设置 thread-limit-var 内在控制变量以设置在 OpenMP 程序中允许的最大线程分割数

环境变量设置方法

需要在程序运行前设置，程序运行后再设置将不起作用，以下为几种常见 shell 下的设置方法：

- csh：
  setenv OMP_SCHEDULE dynamic
- sh、bash、ksh：
  export OMP_SCHEDULE=dynamic
- DOS：
  set OMP_SCHEDULE=dynamic

OpenMP 与 MPI 混合并行编程

MPI 并行进程内部可以再执行 OpenMP 并行线程以联合进行并行计算，一般用于节点内部为共享内存，节点间为分布式内存的超算系统

```
program main
use omp_lib
use mpi
implicit none
integer, save :: iErr, MyID, NumNodes
integer i, j

call mpi_init(ierr)
call mpi_comm_rank(mpi_comm_world, MyID ,iErr)
call mpi_comm_size(mpi_comm_world, NumNodes, iErr)
print*,'MPI: My ID: ', MyID, ', Number of Processes: ', NumNodes
!$omp parallel
i=omp_get_num_threads()
j=omp_get_thread_num()
print*, 'OMP: Thread Number:', j, ', Number of Threads: ', i
!$omp end parallel
call mpi_finalize(ierr)
end
```

OpenMP 程序在 Unix/Linux 下的编译

在 Unix/Linux 平台下常见 GCC、Intel、PGI 编译器的编译方式

- C：
  - GCC： **gcc -o prog-omp -fopenmp prog-omp.c**
  - Intel： **icc -o prog-omp -openmp prog-omp.c**
  - PGI： **pgcc -o prog-omp -mp prog-omp.c**
- C++：
  - GCC： **g++ -o prog-omp -fopenmp prog-omp.cpp**
  - Intel： **icpc -o prog-omp -openmp prog-omp.cpp**
  - PGI： **pgCC -o prog-omp -mp prog-omp.cpp**
- Fortran：
  - GCC： **gfortran -o prog-omp -fopenmp prog-omp.f90**
  - Intel： **ifort -o prog-omp -openmp prog-omp.f90**
  - PGI： **pgf90 -o prog-omp -mp prog-omp.f90**
- 编译器是从某个版本开始支持 OpenMP 的，如 GCC 从 4.3.2 开始
- 对 OpenMP 与 MPI 混合编程的程序，只要将编译命令换成对应的 MPI 编译命令即可

Unix/Linux 下的运行

运行：

- csh、tcsh：
  setenv OMP_NUM_THREADS 8
  prog-omp >log &
- bash、ksh、zsh：
  export OMP_NUM_THREADS=8
  prog-omp >log &
- 对 OpenMP 与 MPI 混合编程的程序的执行，只要设置 OpenMP 环境变量后用 MPI 程序运行方式运行即可
- 很多超算系统采用作业调度系统管理用户作业，此时需通过作业调度系统来运行作业，需参看对应的作业调度系统手册

并行效率

$$S = \frac{1}{f_{par}/P + (1 - f_{par})}$$

Amdahl 定律，其中 P 是并行的核数，f_{par} 为可并行的部分串行时占用的计算时间的百分比
假设 $f_{par} = 80\%$，那么 16 和 32 核时最大的并行效率是多少？

- 16 核：4
- 32 核：4.4

尽可能多地并行化代码，特别是耗时部分

辅助软件工具

- 程序性能分析：
  - Intel VTune：
    http://software.intel.com/en-us/intel-vtune/
  - AMD CodeAnalyst：
    http://developer.amd.com/cpu/CodeAnalyst/
- 程序源码结构分析：
  - Understand：
    http://www.scitools.com/products/understand/

联系方式

- 中国科大超算中心：
  - 主页：http://scc.ustc.edu.cn
  - 电话：0551-3602248
  - 信箱：sccadmin@ustc.edu.cn
- 青能所超算中心：
  - 当前主页：http://124.16.151.186
  - 将来域名：http://scc.qibebt.cas.cn
  - 电话：0532-80662613
  - 信箱：scc@qibebt.ac.cn
- 李会民：
  - 主页：http://staff.ustc.edu.cn/~hmli/
  - 电话：0532-80662613
  - 信箱：hmli@ustc.edu.cn、lihm@qibebt.ac.cn

欢迎指出错误和改进意见。