OpenMP 并行编程

（三）

—— 运行库函数
—— 环境变量

目录页

Contents

目录页

Contents

1 运行库函数

Runtime Library Routines

- 运行环境函数
- 锁函数
- 时间函数

运行环境函数

Execution Environment Routines

`omp_set_num_threads(int)`	设置并行域中的线程个数（用在串行域中）
`omp_get_num_threads()`	返回当前并行域中的线程个数
`omp_get_max_threads()`	返回并行域中缺省可用的最大线程个数
`omp_get_thread_num()`	返回当前线程的线程号，0号为主线程
`omp_get_num_procs()`	返回系统中处理器的个数
`omp_in_parallel()`	判断是否在并行域中
`omp_set_dynamic(int)`	启用或关闭线程数目动态改变功能 （用在串行域中）
`omp_get_dynamic()`	判断系统是否支持动态改变线程数目
`omp_set_nested(int)`	启用或关闭并行域嵌套功能（缺省为关闭）
`omp_get_nested()`	判断系统是否支持并行域的嵌套

运行环境函数

- **omp_set_num_threads**

Fortran	**subroutine omp_set_num_threads(*num_threads*)** **integer** *num_threads*
C/C++	**void omp_set_num_threads(int** *num_threads***);**

- **omp_get_num_threads**

Fortran	**integer function omp_get_num_threads()**
C/C++	**int omp_get_num_threads(void);**

- **omp_get_max_threads**

Fortran	**integer function omp_get_max_threads()**
C/C++	**int omp_get_max_threads(void);**

运行环境函数

- `omp_get_thread_num`

Fortran	integer function omp_get_thread_num()
C/C++	int omp_get_thread_num(void);

- `omp_get_num_procs`

Fortran	integer function omp_get_num_procs()
C/C++	int omp_get_num_procs(void);

- `omp_in_parallel`

Fortran	logical function omp_in_parallel()
C/C++	int omp_in_parallel(void);

运行环境函数

- `omp_set_dynamic`

Fortran	**subroutine omp_set_dynamic (***dynamic_threads***)** **logical** *dynamic_threads*
C/C++	**void omp_set_dynamic(int** *dynamic_threads***);**

- `omp_get_dynamic`

Fortran	**logical function omp_get_dynamic()**
C/C++	**int omp_get_dynamic(void);**

- `omp_set_nested`

Fortran	**subroutine omp_set_nested (***nested***)** **logical** *nested*
C/C++	**void omp_set_nested(int** *nested***);**

运行环境函数

- `omp_get_nested`

Fortran	**logical function omp_get_nested()**
C/C++	**int omp_get_nested(void);**

运行环境函数

OpenMP 3.1 中新函数

- `omp_set_schedule`

Fortran	subroutine omp_set_schedule(kind, modifier) integer (kind=omp_sched_kind) kind integer modifier
C/C++	void omp_set_schedule(omp_sched_t kind, int modifier);

- `omp_get_schedule`

Fortran	subroutine omp_get_schedule(kind, modifier) integer (kind=omp_sched_kind) kind integer modifier
C/C++	void omp_get_schedule(omp_sched_t * kind, int * modifier);

运行环境函数

- `omp_get_thread_limit`

returns the maximum number of OpenMP threads available to the program

Fortran	**integer function omp_get_thread_limit()**
C/C++	**int omp_get_thread_limit(void);**

- `omp_set_max_active_levels`

limits the number of nested active parallel regions

Fortran	**subroutine omp_set_max_active_levels (max_levels)** **integer max_levels**
C/C++	**void omp_set_max_active_levels (int max_levels);**

运行环境函数

- `omp_get_max_active_levels`

returns the maximum number of nested active parallel regions

Fortran	**integer function omp_get_max_active_levels()**
C/C++	**int omp_get_max_active_levels(void);**

- `omp_get_level`

returns the number of nested parallel regions enclosing the task that contains the call

Fortran	**integer function omp_get_level()**
C/C++	**int omp_get_level(void);**

运行环境函数

- `omp_get_ancestor_thread_num`

returns, for a given nested level of the current thread, the thread number of the ancestor or the current thread

Fortran	integer function omp_get_ancestor_thread_num(level) integer level
C/C++	int omp_get_ancestor_thread_num(int level);

- `omp_get_team_size`

returns, for a given nested level, the size of the thread team to which the ancestor or the current thread belongs

Fortran	integer function omp_get_team_size(level) integer level
C/C++	int omp_get_team_size(int level);

运行环境函数

- `omp_get_active_level`

returns the number of nested, active parallel regions enclosing the task that contains the call

Fortran	**integer function omp_get_active_level()**
C/C++	**int omp_get_active_level(void);**

- `omp_in_final`

returns true if the routine is executed in a final task region; otherwise, it returns false.

Fortran	**logical function omp_in_final()**
C/C++	**int omp_in_final(void);**

锁函数 Lock Routines

什么是锁

- OpenMP 提供了一些锁函数用于避免任务竞争，作用类似于 `atomic` 和 `critical`。
- OpenMP 锁是一个特殊的变量，称为锁变量。
- 锁变量的取值有：`uninitialized, unlocked, locked`
- 如果一个锁处于 unlocked 状态，则任务就可以设置该锁，并拥有这个锁，只有拥有该锁的任务才能解锁，其他任务只有等解锁后才能使用这个锁。
- 锁变量有两类：简单锁和嵌套锁，后者的区别在于可以设置多次而不阻塞，分别是 `omp_lock_t` 和 `omp_nest_lock_t`

锁函数 Lock Routines

锁操作：初始化，销毁，上锁，解锁，测试

- 锁函数一般按如下顺序进行调用

① 用 **omp_init_lock / omp_init_nest_lock** 初始化锁变量。

② 调用 **omp_set_lock / omp_set_nest_lock** 上锁，是阻塞型函数。

③ 线程须调用 **omp_unset_lock / omp_unset_nest_lock** 来解锁。

④ 当不再需要该锁，调用 **omp_destroy_lock / omp_destroy_nest_lock** 来释放锁资源，即设为为初始化

⑤ 函数 **omp_test_lock / omp_test_nest_lock** 尝试去上锁，非阻塞型，若上锁成功，返回 1（简单锁）或嵌套层数（嵌套锁），否则返回 0

锁函数

■ OpenMP 锁函数（Lock Routines）

`OMP_INIT_LOCK(omp_lock_t * lock)`	初始化一个简单锁
`OMP_DESTROY_LOCK(omp_lock_t * lock)`	销毁一个简单锁
`OMP_SET_LOCK(omp_lock_t * lock)`	上锁操作
`OMP_UNSET_LOCK(omp_lock_t * lock)`	解锁操作
`OMP_TEST_LOCK(omp_lock_t * lock)`	非阻塞上锁操作

`OMP_INIT_NEST_LOCK(omp_nest_lock_t * lock)`	初始化一个嵌套锁
`OMP_DESTROY_NEST_LOCK(omp_nest_lock_t * lock)`	销毁一个嵌套锁
`OMP_SET_NEST_LOCK(omp_nest_lock_t * lock)`	上锁操作
`OMP_UNSET_NEST_LOCK(omp_nest_lock_t * lock)`	解锁操作
`OMP_TEST_NEST_LOCK(omp_nest_lock_t * lock)`	非阻塞上锁操作

† 详情参见 OpenMP 手册

锁举例

OMP_lock.c

```
int main()
{
    int k;
    omp_lock_t lock;

    omp_init_lock(&lock); // 初始化锁
    #pragma omp parallel num_threads(4) private(k)
    {
        omp_set_lock(&lock); // 上锁
        for(k=0; k<4; k++)
            printf("myid=%d, k=%d\n", omp_get_thread_num(), k);
        omp_unset_lock(&lock); // 解锁
    }
    omp_destroy_lock(&lock); // 释放锁
    return 0;
}
```

OMP_lock_atomic_critical.c

时间函数

OMP_GET_WTIME()	获取 wall time，以秒为单位，双精度型的实数
OMP_GET_WTICK()	获取每个时钟周期的秒数，即 omp_get_wtime 的精度

- **omp_get_wtime**

returns elapsed wall clock time in seconds

Fortran	**double precision function omp_get_wtime()**
C/C++	**double omp_get_wtime(void);**

- **omp_get_wtick**

returns the number of seconds between successive clock ticks

Fortran	**double precision function omp_get_wtick()**
C/C++	**double omp_get_wtick(void);**

时间函数

Fortran

```
real(8) :: t0, t1
t0=omp_get_wtime()
... work to be timed ...
t1=omp_get_wtime()
print *, "Work took", t1-t0, "seconds"
```

C/C++

```
double t0, t1;
t0=omp_get_wtime();
... work to be timed ...
t1=omp_get_wtime();
printf("Work took %f seconds\n", t1-t0);
```

目录页

Contents

2 环境变量 Environment Variables

- 环境变量
- 内在控制变量（ICV）

环境变量

OpenMP 提供了一环境变量来控制并行代码的执行

OMP_SCHEDULE	设置循环任务的调度模式
OMP_NUM_THREADS	设置线程个数
OMP_DYNAMIC	设置是否开启线程数的动态变化功能
OMP_PROC_BIND	设置线程是否与处理器绑定
OMP_NESTED	设置是否开启并行域的嵌套功能
OMP_STACKSIZE	线程的栈的大小，缺省单位是 K
OMP_WAIT_POLICY	设置线程等待时是否占用处理器资源
OMP_MAX_ACTIVE_LEVELS	设置并行域嵌套最大层数
OMP_THREAD_LIMIT	设置整个程序所能使用的最大线程数

环境变量

- 在 Linux 下修改环境变量的一般格式

```
export 环境变量名 = 值
```

```
echo $环境变量名   // 查看变量的值
```

```
export OMP_SCHEDULE=dynamic
export OMP_SCHEDULE="guided,4"
```

```
export OMP_NUM_THREADS=4
```

```
export OMP_DYNAMIC=true
export OMP_DYNAMIC=false
```

环境变量

```
export OMP_PROC_BIND=true
export OMP_PROC_BIND=false
```

```
export OMP_NESTED=true
export OMP_NESTED=false
```

```
export OMP_STACKSIZE=2000
export OMP_STACKSIZE=2G
```

```
export OMP_WAIT_POLICY=ACTIVE
export OMP_WAIT_POLICY=PASSIVE
```

```
export OMP_MAX_ACTIVE_LEVELS=3
```

```
export OMP_THREAD_LIMIT=16
```

ICV内置控制变量

内在控制变量（ICV）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。

详情参见 OpenMP 手册

编程注意事项

- 数据竞争问题；共享内存或伪共享内存引起的访存冲突；
- 私有与共享变量的确定
- 线程间的同步；
- 并行执行的程序比例及其可扩展性；
- 并行执行循环任务时，要检查并重构热点循环，确保循环迭代相互独立；
- 优良的并行算法和精心调试是好的性能的保证，糟糕的算法即使使用手工优化的汇编语言来实现，也无法获得好的性能；
- 创建在单核或单处理器上出色运行的程序同创建在多核或单处理器上出色运行的程序是不同的；
- 可以借助一些性能分析工具，如 Intel VTune Amplifier（Intel 线程监测器）

目录页

Contents

3 编程示例

- Jacobi 迭代法
- 并行 Jacobi 迭代法

Jacobi 迭代法

考虑线性方程组 $Ax = b$

其中 $A=[a_{ij}]_{n\times n}$ 非奇异，且对角线元素全不为 0

- 将 A 分裂成 $A = D - L - U$， 其中

$$D = \mathrm{diag}(a_{11}, a_{22}, \ldots, a_{nn})$$

$$L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ -a_{21} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ -a_{n1} & \cdots & -a_{n,n-1} & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 0 & -a_{12} & \cdots & -a_{1n} \\ 0 & 0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & -a_{n-1,n} \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

Jacobi 迭代公式

$$x^{(k+1)} = D^{-1}(L+U)x^{(k)} + D^{-1}b$$

分量形式

$k = 0, 1, 2, \ldots$

$$\begin{cases} x_1^{(k+1)} = \left(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1n}x_n^{(k)}\right)/a_{11} \\ x_2^{(k+1)} = \left(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \cdots - a_{2n}x_n^{(k)}\right)/a_{22} \\ \quad\vdots \\ x_n^{(k+1)} = \left(b_n - a_{n1}x_1^{(k)} - a_{n2}x_2^{(k)} - \cdots - a_{n,n-1}x_{n-1}^{(k)}\right)/a_{nn} \end{cases}$$

串行代码

用 C 语言实现 Jacobi 迭代算法，取 $A=\begin{bmatrix} 2 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{bmatrix}$ $b=\begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$

```
{
    x[i] = b[i];
    for(j=0; j<i; j++)
        x[i] = x[i] - A[i][j]*x[j];
    for(j=i+1; j<n; j++)
        x[i] = x[i] - A[i][j]*x[j];
    x[i] = x[i]/A[i][i];
}
```