一.MPI

1.获取信息

```
/*
MPI_Comm 是一个通信器。通常，所有进程都要相互通信。
MPI定义一个默认的通信器 MPI_Comm WORLD，它包含所有参与执行的进程。
函数MPI_Comm_size在变量size中返回属于通信器comm的进程数目。
*/
int MPI_Comm_size(MPI_Comm comm,int *size);

/*
进程可以用 MPI_Comm_rank 函数确定其在通信器中的等级，变量rank存储进程的等级。
*/
int MPI_Comm_rank(MPI_comm comm,int *rank);

/*
在每个MPI进程中，该函数必须是最后一个被调用，它只能在所有其他MPI例程完成后被调用。
*/
int MPI_Comm_Finalize();

/*
返回（参数group指向）与通信子相关的组的句柄。
*/
int MPI_Comm_group(MPI_Comm comm,MPI_Group *group);

/*
函数 MPI_Group_incl 将从现有的组中选出进程，已创建新组。
参数group是现有的组，size是新组的大小，rank指向欲包含的进程的序号，newGroup指想欲创建的新组。
*/
int MPI_Group_incl(MPI_Group group,int size,int *rank,MPI_Group *newGroup);

/*
参数origComm是现有的通信子，参数newgroup是新组，参数newComm指向新建的通信子。
*/
int MPI_Comm_create(MPI_Comm oirgComm,MPI_Group newgroup,MPI_Com *new Comm);
```

2.发送和接收消息

```
/*
函数MPI_Send发送存储在有buf指向的缓冲区的数据。
count给出缓冲区中项目的个数，datatype指定数据类型，dest参数是通信器comm指定的通信域中的目标进程的等级，
tag用来区分不同类型的消息。
*/
int MPI_Send(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm);

/*
```

```
函数MPI_Recv接收到由一个进程发来的消息，该进程的等级由comm参数指定的通信域中的source给出，接收的消息存储在有buf指定的缓冲区中的连续单元。
已发送消息的标志必须由tag参数指定，如果来自同一进程的许多消息的标志相同，那么这些消息中的任意一个被接收。
MPI运行对source和tag使用通配符，如果source被设置为MPI_ANY_SOURCE,那么通信域中的任意进程都能成为消息源；
如果tag被设置为MPI_ANY+TAY，那么具有任何标准的消息都被接收。
count参数用来指定提供的缓冲区的长度，datatype参数用来指定数据类型。
可以用status变量获取有关MPI_Recv操作的信息。在C语言中，status以MPI_Status数据结构存储：
    typedef struct MPI_Status{
        int MPI_SOURCE; //保存接收的消息源
        int MPI_TAG;    //保存接收的标志
        int MPI_ERROR;  //保存接收的消息的出错码
    }
*/
int MPI_Recv(void *buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm
comm,MPI_Status *Status);

/*
函数MPI_Get_count用来返回有关接收消息的长度信息。count参数用来记录实际接收到的数据项的数目。
*/
int MPI_Get_count(MPI_Status *status,MPI_Datatype datatype,int *count);
```

3.同时发送和接收消息

```
/*
函数 MPI_Sendrecv 既发送消息又接受消息，MPI_Sendrecv 中的参数实际上是 MPI_Send 和 MPI_Recvv 中参数的结合。
发送和接收缓冲区必需分开，消息源和目标可以相同，也可以不同。
*/
int MPI_Sendrecv(void *sendbuf,int sendcount,MPI_Datatype senddatatype,int dest,int
sendtag,void * recvbuf,int recvcount,MPI_Datatype recvdatatype,int source,int
recvtag,MPI_Comm comm,MPI_Status *status);

/*
函数 MPI_Sendrecv_replace 执行阻塞式发送和接收操作，但对发送和接收操作只使用单一的缓冲区，也就是说，接收到的数据替换发送出缓冲的数据。注意，发送和接收操作必须传递同一种数据类型的数据。
*/
int MPI_Sendrecv_replace(void *buf,int count,MPI_Datatype datatype,int dest,int sendtag,int
source,int recvtag,MPI_Comm comm,MPI_Status*status);
```

4.创建和使用笛卡尔拓扑结构

```
/*
函数 MPI_Cart_create 用来获取属于通信器comm_old的一组进程，并创建一个虚拟进程结构，结构信息附属在新通信器comm_cart上。任何后面的MPI例行程序，如果想利用这种新的笛卡尔结构，就必须使用com_cart作为通信器参数。
参数ndims指定拓扑结构的维数，数组dims指定拓扑结构中每一维的大小，数组periods用于指定拓扑结构中是否有环绕连接，如果periods[i]为真，那么拓扑结构沿i维有环绕连接。
参数reorder用于确定新通信器中的进程是否需要进行重新排序。
*/
int MPI_Cart_create(MPI_Comm comm_old,int ndims,int *dims,int *periods,int reorder,MPI_Comm
*comm_cart);
```

```
/*
函数 MPI_Cart_rank 以数组coords中的进程的坐标为参数，并用rank返回进程的等级。
*/
int MPI_Cart_rank(MPI_Comm comm_cart,int *coords,int *rank);

/*
函数 MPI_Cart_coord 进程的等级作为参数，并返回数组coords中的笛卡尔坐标以及长度maxdims。要注意，maxdims
至少要等于通信器comm_cart指定的笛卡尔结构的维数。
*/
int MPI_Cart_coord(MPI_Comm comm_cart,int rank,int maxdims,int *coords);

/*
函数 MPI_Cart_shift 用来结算在数据交换操作中源进程和目标进程的等级，计算出来的等级在rank_source和
rank_dest中返回。如果笛卡尔结构是环绕连接的，则数据也是环绕交换，否者，对于在拓扑结构外的进程，对其
rank_source和rank_dest返回值MPI_PROC_NULL。
参数dir指定数据交换的方向，并且是拓扑结构的一个维。
参数s_step指定交换数据的大小。
*/
int MPI_Cart_shift(MPI_Comm comm_cart,int dir,int s_step,int *rank_source,int *rank_dest) ;
```

5.无阻塞式通信操作

```
/*
MPI提供的一对执行无阻塞式发送和接收操作的函数，分别是 MPI_Isend 和 MPI_Irecv 。MPI_Isend 开始一个发送
操作，但不等它完成，在数据复制出缓冲区前就返回； MPI_Irecv 开始一个接收操作，但在数据接收完成完毕且复制到缓
冲区前就返回。
两个函数分配一个请求（request）对象并返回一个指向request变量的指针，用来确定需要查询其状态或等待其完成的操
作。
*/
int MPI_Isend(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm
comm,MPI_Request *request);
int MPI_Irecv(void *buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm
comm,MPI_Request *request);

/*
MPI_Test测试有request确定的无阻塞发送或接收操作否是完成。如果已完成，则返回flag={true}（在c语言中的非0
值），否则返回flag={false}。
在无阻塞操作完成后，变量request指向的请求对象被解除分配，request被设置为MPI_REQUEST_NULL。同时status对
象被设置为包含与操作相关的信息。
如果操作未完成，request就不被修改，status对象中的值未被定义。
*/
int MPI_Test(MPI_Request *request,int *flag,MPI_Statuts *status);

/*
MPI_Wait函数由request确定的无阻塞操作完成前一直阻塞。在这种情况下，它解除分配request对象，将其设置为
MPI_REQUEST_NULL,并在status对象中返回有关已完成操作的信息。
*/
int MPI_Wait(MPI_Request *request,MPI_Status *status);

/*
函数 MPI_Request_free 可以显示解除分配一个请求对象。
```

```
注意，请求对象的解除分配对相关的无阻塞式发送和接收操作没有任何影响，也就是说，如果这些操作未完成，它们弧继续
进行下去。因此要小心对请求对象的显示解除分配你，没有请求对象就无法测试无阻塞式操作是否已完成。
*/
int MPI_Request_free(MPI_Request *request);
```

6.聚合的通信和计算操作

```
/*
6.1障碍
函数 MPI_Barrier 实现障碍同步操作，函数中只有通信器一个参数，它定义被同步的进程组，
对 MPI_Barrier 的调用只有在组内的所有进程调用了这个函数后才返回。
*/
int MPI_Barrier(MPI_Comm comm);

/*
6.2广播
函数 MPI_Bcast 将存储在进程source的缓冲区buf中的数据发送到组中的所有其他进程。
被广播的数据中，类型datatype的数据有count项。
*/
int MPI_Bcast(void *buf,int count,MPI_Datatype datatype,int source,MPI_Comm comm);

/*
6.3归约
函数 MPI_Reduce 使用由op指定的操作符将组内每个进程中存储在缓冲区sendbuf的元素组合起来，并用等级为target
的进程中的缓冲区recvbuf返回组合后的值。
sendbuf和recvbuf中类型为datatype的数据项数目都必须为count。
所用进程都必须提供一个recvbuf数组，即使这些进程不是归约操作的目标（target）。
*/
int MPI_Reduce(void *sendbuf,void *recvbuf,int count,MPI_Datatype datatype,,MPI_Op op,int
target,MPI_Comm comm);

/*
6.3归约
函数 MPI_Allreduce 提供全归约操作，所有的进程都接收归约操作的结果，此函数不含target参数。
*/
int MPI_Allreduce(void sendbuf,void *recvbuf,int count,MPI_Datatype datatype,MPI_Op
op,MPI_Comm comm);

/*
6.4前缀
函数 MPI_Scan 对存储在每一个进程的缓冲区sendbuf中的数据进行前缀归约，并在换乘区revcbuf中返回结果。
在归约操作的最后，等级为i的进程的接收缓冲区，将存储等级由0到i（包括i）的进程的发送缓冲区中的归约结果。
*/
int MPI_Scan(void *sendbuf,void *recvbuf,int count,MPI_Datatype datatype,MPI_Op,MPI_Comm
comm);

/*
6.5收集
每个进程（包括target进程）将存储在数组sendbuf中的数据发送给target进程。
如果p是通信comm中处理器的数目，则目标进程将接收的缓冲区总数为p。
数按等级顺序存储在目标的recvbuf数组中。
```

```
每个进程发送的数据必须具有相同的类型和大小，即sendcount和senddatatype必须在每个进程中具有相同的值。参数
recvcount指定的是每个进程节接收的元素数目而不是接收的元素总数目，所以recvcount必须和sendcount相同，两者
的类型也必须匹配。
*/
int MPI_Gather(void *sendbuf,int sendcount,MPI_Datatype senddatatype,void *recvbuf,int
recvcount,MPI_Datatype recvdatatype,int target,MPI_Comm comm);

/*
6.5收集
函数 MPI_Allgather 为所有进程收集数据而不仅给目标进程。每个进程必须提供一个recvbuf数组来存储收集的数据。
*/
int MPI_Allgather(void *sendbuf,int sendcount,MPI_Datatype senddatatype,void *recvbuf,int
recvcount,MPI_Datatype recvdatatype,MPI_Comm comm);

/*
6.5收集
上述的6.5收集中，每个进程发送的数组大小都相等，下面的两个函数提供了数组大小可以不等的收集操作。MPI称这种操作
为 向量（vecto）变体。
将recvcount参数用数组recvcounts代替，进程i发送的数据等于recvcounts[i]，displs用来确定由每个进程发送出
的数据存储于revcbuf中的位置，由进程i发送出来的数据存储与recvbuf中起始单元为displs[i]的地方。对于不同的进
程，参数sendcount也可以不同。
*/
int MPI_Gatherv(void *sendbuf,int sendcount,MPI_Datatype senddatatype,void *recvbuf,int
*recvcounts,int *displs,MPI_Datatype recvdatatype,int target,MPI_Comm comm);
int MPI_Allgatherv(void *sendbuf,int sendcount,MPI_Datatype senddatatype,void *recvbuf,int
*recvcounts,int *displs,MPI_Datatype recvdatatype,int target,MPI_Comm comm);

/*
6.6散发
源程序将发送缓冲区sendbuf中的不同部分发送给每个进程，也包括它自身，接收到的数据存储于recvbuf中。
进程i接收sendcount个类型为senddatatype连续的元素，这些元素从源进程缓冲区sendbuf的i*sendcount单元开
始。再次注意，sendcount是发送每一个进程的元素数目。
*/
int MPI_Scatter(void *sendbuf,int sendcount,MPI_Datatype senddatatype,void *recvbuf,int
recvcount,MPI_Datatype recvdatatype,int source,MPI_Comm comm);

/*
6.6散发
允许将不同数量的数据发送给不同的进程。数组sendcounts替代了参数sendcount，该数值确定发送给每个进程的元素数
目。
数组displs用来确定这些元素从sendbuf的何处发送出。
*/
int MPI_Scatterv(void *sendbuf,int *sendcounts,int *displs,MPI_Datatype senddatatype,void
*recvbuf,int recvcount,MPI_Datatype recvdatatype,int source,MPI_Comm comm);

/*
6.7多对多
每个进程发送数组sendbuf中的不同部分给其他每个进程，包括他们自身。
每个进程i sendcount个类型为senddatatype连续的元素，这些元素从它的sendbuf数组中i*sendcount单元开始。接
收到的数据存储于数组recvbuf中，每个进程从进程i接收recvcount个类型为recvdatatype的元素，并将元素存储在
recvbuf数组中从i*sendcount开始的单元。
所有进程调用 MPI_Alltoall 时，必须使用同样值的sendcount、senddatatype、recvcount、recvdatatype以及
comm参数。
```

```
*/
int MPI_Alltoall(void *sendbuf,int sendcount,MPI_Datatype senddatatype,void *recvbuf,int
recvcount,MPI_Datatype recvdatatype,MPI_Comm comm);

/*
6.7多对多
允许每个进程发送或接收不同数量的数据。
*/
int MPI_Alltoallv(void *sendbuf,int *sendcounts,int *sdispls,MPI_Datatype senddatatype,void
*recvbuf,int *recvcounts,int *rdispls,MPI_Datatype recvdatatype,MPI_Comm comm);
```

7.进程组和通信器

```
/*
函数 MPI_Comm_split 将属于某一通信器的进程划分为子组，每个子组都与不同的通信器相对应。
此函数是聚合操作函数，必须由通信器comm中的所有进程调用。
每个子组包含对color参数提供同样值的所有进程，在每个子组内部，进程按由参数key的值定义的顺序确定等级，与它们在
老的通信器中进程等级的划分相联系。
newcomm参数为每个子组返回一个新的通信器。
*/
int MPI_Comm_split(MPI_Comm comm,int color,int key,MPI_Comm *newcomm);

/*
函数 MPI_Cart_sub 用于将笛卡尔拓扑结构划分成构成低维网格的子结构。
数组keep_dims用于指定如何划给笛卡尔拓扑结构，如果keep_dims[i]为真（在C语言中为非零），那么在新的子结构
中，第i维保存不变，注意，新创建的子结构的数目等于沿未保留的各维上的进程数目的乘积。
例如：对于一个2*4*7的三维结构，keep_dims的值是{true，false，true}，那么原始结构将分裂为 4 个 2*7 的二
维子结构；keep_dims的值是{false，false，true}，那么原始结构将分裂为 8 个大小为 1 的一维子结构；
*/
int MPI_Cart_sub(MPI_Comm comm_cart,int *keep_dims,MPI_Comm *comm_subcart);
```

二.Pthread

1.线程的创建和终止

```
/*
函数 pthread_create 创建一个单一的线程，对应thread_function函数的调用。成功创建一个线程后，有唯一的标识
符被分配到由thread_handle指向的位置；pthread_create返回0，否者返回一个错误代码。
参数attribute给出线程的属性，该参数为NULL时，创建一个默认属性的线程。
字段arg指定一个指针，指向函数thread_function的参数，这个参数通常用来将工作区或其他线程指定的数据传送给一个
线程；使用一个结构可以向起始函数传递多个参量。
*/
#include<pthread.h>
int pthread_create(pthread_t *thread_handle,const pthread_attr_t *attribute,void*
(*thread_function)(void *),void* arg);

/*
函数 pthread_join 挂起正在调用的线程，直到指定的线程终止，对这个函数的调用将有thread给出其id的线程终止。
pthread_join 调用成功完成后返回0，否则返回一个错误代码。
```

```
在对函数的每一次成功调用中，传递给pthread_exit的值返回到由ptr指向的位置；如果status不为NULL，退出线程的
完成转态将复制到*ptr中，否者，完成状态将不会复制。
*/
int pthread_join(pthread_t thread,void **ptr);

/*
调用该函数的线程的id
*/
pthread_t pthread_self();

/*
如果2个线程id相同，则返回非0值；否者返回0
*/
int pthread_equal(pthread_t t1,pthread_t t2);

/*
用来终止线程的函数。参数status用来保存已退出的线程的完成状态，该指针可供其他线程使用。
*/
void pthread_exit(void *status);
```

2.互斥

```
/*
对函数 pthread_mutex_lock 的调用将试图锁定互斥锁mutex_lock。如果mutex_lock已经锁定，则线程阻塞，否则
mutex_lock被锁定，并且调用的线程返回数值0。
*/
int pthread_mutex_lock(pthread_mutex_t *mutex_lock);

/*
对函数 pthread_mutex_unlock 调用时，杜宇正常的mutex_lock，锁被放弃，阻塞的线程之一被调度进入临界区。指定
的线程由调度策略确定。
如果试图对已经解锁的互斥锁或对被另一个线程锁定的互斥锁，进行pthread_mutex_unlock操作，其结果是不确定的。
*/
int pthread_mutex_unlock(pthread_mutex_t *mutex_lock);

/*
使用互斥锁之前，需要使用函数 pthread_mutex_init 将互斥锁初始化为未锁定状态的函数。
*/
int pthread_mutex_init(pthread_mutex_t *mutex_lock,const pthread_mutexattr_t *lock_attr);

/*
销毁互斥体
*/
int pthread_mutex_destroy(pthread_mutex_t *mutex_lock);

/*
函数 pthread_mutexattr_init 用来初始化互斥体属性
函数 pthread_mutexattr_init 用来销毁互斥体属性
*/
int pthread_mutexattr_init(pthread_mutexattr_t *lock_attr);
int pthread_mutexattr_init(pthread_mutexattr_t *lock_attr);
```

```
/*
函数 pthread_mutex_trylock 可以减少与锁相关的空闲开销。
这个函数试图锁定mutex_lock，如果锁定成功，函数返回0；如果锁已被其他线程锁定，函数返回一个值EBUSY而不是阻塞线程操作，这样就使能使线程执行其他任务并轮询互斥锁。
在典型的系统中，通常 pthread_mutex_trylock 要比 pthread_mutex_lock快得多。
*/
int pthread_mutex_trylock(pthread_mutex_t *mutex_lock);
```

3.同步原语

```
/*
用于同步的条件变量：条件变量是用来同步进程的数据对象，这个变量运行某一线程阻塞自己，直到指定的数据达到预定的状态。
一个条件变量总是有一个互斥锁与之对应。一个线程锁定这个互斥锁并测试对共享变量定义的谓词；如果谓词非真，则线程等待与使用函数 pthread_cond_wait 的谓词相关的条件变量。
函数 pthread_cond_wait 的调用将阻塞进程的执行，直至线程收到来自另一个线程的信号或者被一个操作系统的信息中断；并且解除对互斥锁的锁定。
函数 pthread_cond_signal 对当前等待条件变量cond的一个线程解除阻塞。
*/
int pthread_cond_wait(pthread_cont_t *cond,pthread_mutex_t *mutex));
int pthread_cond_signal(phtread_cond_t * cond);

/*
函数 pthread_cond_init 初始化一个条件变量（有cond指定），其属性由属性对象attr定义，将attr置为NULL对条件变量赋予默认的属性。
如果在程序某处，条件变量不在需要，可以使用函数 pthread_cond_destroy 将它废弃。
*/
int pthread_cond_init(pthread_cond *cons,const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);

/*
函数 pthread_cond_broadcast 的调用可以唤醒所有等待条件变量的线程而不是单个线程可能会有利。
*/
int pthread_cond_broadcast(pthread_cond_t *cond);

/*
使用函数 pthread_cond_timewait ，线程就会执行等待条件变量，直至指定的时间届满。此外，如果线程没有收到信号或广播，就会自己被唤醒。
如果在收到信号或广播前，指定的绝对时间abstime届满，函数返回一个错误信息。当函数成为可用时，它也会重新获得互斥锁。
*/
int pthread_cond_timewait(pthread_cond_t *cond,pthread_mutex_t *mutex,const struct timespec
*abstime);

//线程指定数据：POSIX Threads使用了一组键，既能在一个进程中被所有线程共享，又能为每个线程映射不同的指针值。
int pthread_key_create(pthread_key_t *key,           //要创建的键
                       void(*destructor)(void *));  //析构函数，NULL表示无析构函数
int pthread_key_delete(
    pthread_key_t *key);    //要删除的键
int pthread_setspecific(pthread_key_t *key,  //要设置的键
                        void *value);       //要设置的键
int pthread_getspecific(
```

```
    phtead_key_t *key);      //映射到键的值
```

4.控制线程及同步的属性

```
/*
函数 pthread_attr_init 用来创建线程的属性对象，用默认值初始化属性对象attr。
初始化成功后，函数返回0，否者返回一个错误代码。
*/
int pthread_attr_init(pthread_attr_t *attr);

/*
函数 pthread_attr_destroy 用来取消线程的属性对象。
成功地取消属性对象attr后，函数的调用返回0。与属性有关的性质可用如下函数修改：
pthread_attr_setdetachstate，pthread_attr_setguardsize_np，pthread_attr_setstacksize，
pthread_attr_setinheritshed，pthread_attr_setschedpolicy以及pthread_attr_setschedparam，这些函数分别用来设置线程属性对象的分离状态、堆栈保护大小、堆栈大小、调度策略是否从创建继承线程继承、调度策略（在未继承的情况下）以及调度参数。
*/
int pthread_attr_destroy(pthread_attr_t *attr);

/*
Pthread API支持三种不同的锁：
1.PTHREAD_MUTEX_NORMAL_NP：正常互斥锁，是一种默认类型的锁，在任何时刻，只允许一个线程锁定互斥锁。
2.PTHREAD_MUTEX_RECURSIVE_NP：递归互斥锁，允许单一的线程多次锁定互斥锁，线程每次锁定一个互斥锁时，锁计数器加1，每次解锁计数器减1。如果其他任何线程想成功锁定一个递归互斥锁，锁计数器必须为0。当线程函数需要递归调用自己时，就需要用到递归互斥锁。
3.PTHREAC_MUTEX_ERRORCHEck_NP：错误检查互斥锁，一个线程只能锁定一个互斥锁一次，但是，当一个线程试图锁定一个已经锁定的互斥锁时，错误检查锁将返回一个错误而不是死锁。因此，错误检查互斥锁在排错时更有用。
函数 pthread_mutexattr_init 创建并初始化一个互斥锁对象attr，互斥锁的默认类型为正常互斥锁。
函数 pthread_mutexattr_settype_np 用来设置互斥锁属性对象指定的类型（type），与正常、递归和错误检查这三种互斥锁的类型相对应。
*/
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_settype_np(pthread_mutexattr_t *attr,int type);
```

5.线程注销

```
/*
函数 pthread_cancel 提供线程注销功能，参数thread是被注销线程的句柄。
一个线程线程可以注销自己，也可以注销其他线程。调用此函数后，注销就发送给指定的线程，但不能保证指定的线程一定会收到注销，也不能保证指定的线程执行注销。线程可以保护自己不被注销哦。
当真正注销是，调用清理函数恢复线程数据结构，这一过程与调用函数pthread_exit终止线程类似。
函数 pthread_cancel 在注销发送后返回，二注销操作可能推迟执行。注销成功时返回0，并不代表请求的线程已被注销，仅说明指定的线程是有效的注销线程。
*/
int pthread_cancel(pthread_t thread);
```

6.复合同步结构

```
// (P211) 设计读写锁，基于一种mylib_rwlock_t的数据结构
mylib_rwlock_rlock       //读出锁函数
mulib_rwlock_wlock       //写入锁函数
mylib_rwlock_unlock      //解锁函数

// (P225) 设计障碍
```

三.OpenMP

1.编程模型

```
//C语言及C++中南的OpenMP命名基于#pragma编译器命令，命令本身由命令名及后面的子句构造。除非遇到parallel命令，否者OpenMP程序串行执行。
#pragma omp directive [clause list]

/*
parallel命名用来创建一组线程。线程的确切数目可由命令指定，用一个环境变量指定，或者在运行时用OpenMP函数指定。每条由这个命令创建的线程执行有parallel命令指定的structured block。
子句列表用来指定条件并行、线程数目以及数据处理：
1.条件并行：子句if(标量表达式)决定并行结构是否将导致常见进程；一个parallel命令只能有一个if子句。
2.并发度：子句num_threads(整数表达式)指定parallel命令创建的线程数目。
3.数据处理：
     (1) 子句private(变量表)表示指定的变量集对每个线程都是本地的，即每个线程对表中的每个变量都有一个副本。
     (2) 子句firstprivate (变量表) 与子句private类似，还需要进入线程变量的值要在parallel命令前初始化成相应的值。可以复制全局变量的值到本地线程版本的变量，但是，并行区域的私有化本地线程变量的修改任然不会改变全局变量的值。
     (3) 子句shared (变量表) 表示表中的所有变量对于所有线程都是共享的，即只有一个副本。
     (4) 由子句default (shared) 或default(none)指定变量的默认状态。子句default (shared)表明，在默认情况下，一个变量被所有线程共享；子句default (none) 表明，线程中的每个变量的状态必须显示指定。
     (5) 子句reduction指定不同线程中的变量的多个副本如何在主线程中组合一个副本。子句reduction的用法为reduction (操作符：变量表) ，表中的变量被隐式指定为对线程私有，操作符可以是+，*，-，&，|，^，&&或||。
*/
#pragma omp parallel [clause list]
/* structured block */
```

2.并发任务

```
//OpenMP提供两条命令--for和sections--用来指定并发的迭代和任务。

/*
for命令可用的子句有private、firstprivate、lastprivate、reduction、schedule、nowait和ordered。
(1) 子句lastprivate用来处理变量的多个本地副本在并行for循环结束时如何写回单一副本中。
(2) 子句schedule用来将迭代分配给线程。schedule命令的一般形式为
schedule (scheduling_class[,parameter]) 。OpenMP支持4种调度类：
    1) static调度：一般形式为schedule (static[,chunk-size])。将迭代空间分割成大小为chunk-seiz的相等的块，并将这些块循环地分配给线程。如果没有指定chunk-size，默认的迭代空间和线程数目一样多。
    2) dynamic调度：一般形式为schedule (dynamic[,chunk-size])。迭代空间被分为chunk-size大小的块，但是这些块只有在线程空闲时才分配给它们。如果没有指定chunk-size，则默认每一个块一个迭代。
    3) guided调度：一般形式为schedule (guided[,chunk-size])。当每个块分配给线程时，块的大小按指数缩小。
```

```
chunk-size为应分配的最小块的大小。如果chunk-size没有指定，则默认为1。
    4）运行时间：将调度设置为runtime，由环境变量OMP_SCHEDULE决定调度类以及块的大小。
（3）子句nowait用来表明线程可以进入下一个语句，无需等待其他所有的线程结束for循环执行。
（4）ordered命令代表for循环的按序执行，它必须位于 for 或 parallel for 命名的作用域内。进而， for 或 parallel for 命令中必须含有ordered子句。
*/
#pragma omp for [cluse list]
/* for loop */

/*
sections命令执行非循环的并行任务分配，将于每个段对应的结构化块分配给一个线程（可以将一个以上的段分配给一个进程）。
cluse list 中可能包含以下子句--private、firstprivate、lastprivate、reduction以及nowait。在sections命名的后面，子句nowait指定所有线程没有隐式同步。
*/
#pragma omp sections [clause list]
{
    [#pragma omp section
        /* structured block */
    ]
    [#pragma omp section
        /* structured block */
    ]
    ...
}

//合并命令：如果没有指定parallel命令，则for命令和sections命令将串行执行（主线程执行）。OpenMP运行程序员把parallel命令分别合并成 parallel for 和 parallel sections。合并后命令的子句列表可以是parallel命令的子句列表，也可以是for/sections命令的子句列表。

//嵌套并行：必须用 OMP_NESTED 环境变量来启用嵌套并行。如果 OMP_NESTED 被设置为FALSE，那么内部的parallel区域将串行化，并由一个线程执行；如果 OMP_NESTED 被设置为TRUE，则嵌套并行被启动。环境变量默认为FALSE。
```

3.同步结构

```
//遇到此命令，队中所有线程都要等待其他线程跟上，然后释放。
#pragma omp barrier

/*
single命令指定一个由单一（任意的）线程执行的结构化块，子句表（clause list）可以是子句private、firstprivate以及nowat。
遇到single块时，第一个线程进入块，所有其他线程进入到块的末尾，如果指定nowait，则其他的线程继续，否者在末尾等待。
这条命令在计算全局数据和进程I/O操作时很有用。
*/
#pragma omp single [cluse list]
/* structured block */

/*
master命令是single命令的特殊情况，它指定只能有主线程执行结构化块。并且，不伴随隐式障碍。
*/
#pragma omp master
```

```
/* structured block */

/*
critical命令用来实现临界区域。可选标识符 name 用来表示临界区，使用 name 可以使不同的线程执行不同的代码，且相互保护。如果某一线程已经在临界段内部（已命名），所有其他的线程必须等待该线程完毕后才能进入命名的临界段。
如果没有指定名称，则临界段映射到一个默认名称，这个名称和所有未命名的临界段的名称相同；在整个程序中，临界段的名称都是全局的。
指令block必须表示结构化块，即不允许跳入块或从块中跳出。跳入块将导致非临界访问；跳出块将导致未释放的锁，引起线程无限等待。
所有的 atomic 命令都可以被 critical 命令替代，但是原子化硬件指令的实用性可以优化程序的性能你。
*/
#pragma omp critical [(name)]
/* structured block */

/*
ordered命令代表for循环的按序执行，它必须位于 for 或 parallel for 命名的作用域内。进而， for 或 parallel for 命令中必须含有ordered子句。
注意，如果循环的大部分包含在ordered命令中，则相应的加速比就不能达到。
*/
#pragma omp ordered
/* structured block */

/*
flush命令提供在线程间实现内存一致的机制，强制变量写入内存系统或将变量从内存系统读出。选项 list 指定需要被刷新的变量，默认情况下，所有共享变量都被刷新。
flush命名提供一个内存栅栏，所有对共享变量的写操作必须在一个flush中提交到内存，所有在栅栏后对共享变量的引必须从内存实现。由于私有变量之和一个线程有关，flush命名只应用于共享变量。
有的OpenMP命令有隐式flush，特别是flush隐含在barrier中，在critical、ordered、parallel for以及parallel sections块的入口和出口处，在for、sections以及single块的出口处。如果有nowait子句，则不隐含flush。在for、sections以及single块的入口处，以及master块的入口或出口处，也没有隐含flush。
*/
#pragma omp flush [(list)]
```

4.数据处理

```
/*
threadprivate命令隐含在variable_list中的所有变量对每个线程而言都是本地的，且在一个并行区域内被访问前初始化一次。此外，如果线程数目的动态调整被禁止和线程数目不变，则这些变量在不同的并行区域中能保持不变。
copyin命令将同样的值分配个一个并行区域的所有线程的threadprivate变量，它可以和parallel命名一起使用。
*/
#pragma omp threadprivate(variable_list)
#pragma omp copyin(variable_list)
```

5.库函数

```
/////////控制线程和处理器数目
//设置默认的线程数
void omp_set_num_threads(int num_threads);
//返回并行范围内的线程数目
int omp_get_num_threads();
```

```
//返回可能有遇到的parallel命令创建的最大线程数目
int omp_get_max_threads();
//返回每个线程的整数id（主线程的id为0）
int omp_get_thread_num();
//返回在那一点可用来执行线程程序的处理器数目
int omp_get_num_procs();
//对于从并行区域内的调度返回一个非0值，从并行区外的调用返回0
int omp_in_parallel();


/////////控制和监控线程的创建
//运行程序员动态地改变在遇到并行区时创建的线程数目
void omp_set_dynamic(int dynamic_threads);
//在动态调整时返回一个非0值，没有启用时返回0
int omp_get_dynamic();
//在其参数nested为非0时运行嵌套并行，为0时停止嵌套并行
void omp_set_nested(int nested);
//查询嵌套并行的转态，它在嵌套并行时返回一个非0值，停止使用时返回0
int omp_get_nested();


////////互斥
//锁额初始化
void omp_init_lock(omp_lock_t * lock);  //锁数据结构的类型为omp_lock_t
//废弃不需要的锁
void omp_destroy_lock(omp_lock_t * lock);
//锁定
void omp_set_lock(omp_lock_t * lock);
//解锁
void omp_unset_lock(omp_lock_t * lock);
//测试锁是否已被设置，如果锁已被成功设置则返回一个非0值
int opm_test_lock(omp_lock_t * lock);

//与Pthreads中的递归互斥锁一样，OpenMP也支持可嵌套锁，同一线程对可嵌套锁可以锁定多次，对应的函数如下：
void omp_init_nest_lock(omp_nest_lock_t * lock);//锁数据结构的类型为omp_nest_lock_t
void omp_destroy_nest_lock(omp_nest_lock_t * lock);
void omp_se_nestt_lock(omp_nest_lock_t * lock);
void omp_unset_nest_lock(omp_nest_lock_t * lock);
int opm_test_nest_lock(omp_nest_lock_t * lock);
```

6.环境变量

(1) OPM_NUM_THREADS:
    指定进入parallel区域时创建线程的默认数目。线程的数目可以用函数omp_set_num_threads改变，也可以用parallel命令的num_threads子句改变。注意只有在变量OPM_SEt_DYNAMIC设置为TRUE或者用非0调用函数omp_set_dynamic时，才能动态地改变线程数目。例如，如果程序执行前在csh中输入如下命令，那么默认的线程数目设置为8，
    1 setenv OPM_NUM_THREADS 8
(2) OPM_DYNAMIC:
    设置为TRUE时，运行线程的数目在运行时用函数opm_set_num_threads或num_threads子句进行控制。如果用参数0调用omp_set_dynamic,则对线程数目的动态控制被停止。
(3) OPM_NESTED:
    设置为TRUE时，启用嵌套并行，用参数0调用函数omp_set_nested停用嵌套并行。
(4) OPM_SCHEDULE:
    控制与使用runtime调度类的for命令相关的迭代空间的指定。变量的取值可以是static、dynamic以及guided，在加上可选的块的大小。例如，
    1 setenv OPM_SCHEDULE "static,4"
指定在默认情况下，所有的for命令使用static调度，块大小为4。指定的其他例子包括：
    1 setenv OPM_SCHEDULE "dynamic"
    2 setenv OPM_SCHEDULE "guided"

四.其他

1.利用chrono计算程序的运行时间

```
#include<iostream>
using namespace std;
#include<omp.h>
#include<vector>
#include<chrono>

chrono::high_resolution_clock::time_point start, stop;  //定义变量

//获取起始时间
template<typename T>
void timeStart(T t) {
    start = chrono::high_resolution_clock::now();
}
//获取结束时间
template<typename T>
void timeEnd(T t) {
    stop = chrono::high_resolution_clock::now();
    //获取时间差 microseconds代表微秒，nanoseconds代表纳秒
    auto duration = chrono::duration_cast<chrono::microseconds>(stop - start);

    //将微秒转换为秒
    printf(t);
    printf(": %.7f s\n", (float)duration.count() * 1e-6);
    //cout << t << ": " << (float)duration.count() * 1e-6 << "s" << endl;
}

int main() {
    const int n = 5;
    const int m = 5;
```

```
    timeStart("alloc");
    vector<int>A(m * n);
    vector<int>x(n);
    vector<int>b(m);
    timeEnd("alloc");

    system("pause");
    return 0;
}
```