**OpenMP**是**共享存储体系结构**上的一个基于**线程**的**并行编程模型**。适合于**共享内存多处理系统**和**多核处理器**体系结构。并行化控制的三要素：**编译制导**、**运行时库函数**和**环境变量**组成。

编译制导指令以制导标识符#pragma omp开始，后边跟具体的功能指令，格式如：#pragma omp 指令[子句[,子句] …] 一个并行域就是一个能被多个线程并行执行的，在并行域结尾有一个隐式同步（barrier）

**功能指令 parallel**用在一个结构块之前，表示这段代码将被多个线程并行执行**for**：用于for循环语句之前，表示将循环计算任务分配到多个线程中并行执行，以实现任务分担 **parallel for**表示for循环体的代码将被多个线程并行执行，它同时具有并行域的产生和任务分担两个功能；**sections**实现多个结构块语句的任务分担，可并行执行的代码段各自用section指令标出**parallel sections** parallel和sections两个语句的结合**single**用在并行域内，表示一段只被单个线程执行的代码；**critical**用在一段代码临界区之前，保证每次只有一个OpenMP线程进入；临界区指的是一个访问共用资源（例如：共用设备或是共用存储器）的程序片段，而这些共用资源又无法同时被多个线程访问的特性。当有线程进入临界区段时，其他线程或是进程必须等待**barrier**用于并行域内代码的线程同步，线程执行到barrier时要停下等待，直到所有线程都执行到barrier时才继续往下执行**atomic**用于指定一个数据操作需要原子性地完成原子操作是指不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行到结束。critical 可以对代码块进行临界区设置，而atomic只能对代码语句进行加持。 原子操作是要独占处理器，其他的线程必须等原子操作完了才可以运行。**master**用于指定一段代码由主线程执行；**threadprivate**用于指定一个或多个变量是线程专用

**子句 private**指定一个或多个变量在每个线程中都有它自己的私有副本**reduction**用来指定一个或多个变量是私有的，并且在并行处理结束后这些变量要执行指定的归约运算，并将结果返回给主线程同名变量**num\_threads**指定并行域内的线程的数目**shared**指定一个或多个变量为多个线程间的共享变量；

**API函数omp\_get\_thread\_num**返回线程号**omp\_set\_num\_threads**设置后续并行域中的线程格式**omp\_get\_num\_threads**返回当前并行域中的线程数**omp\_get\_max\_threads** 获得并行域中可用的最大线程数**omp\_get\_num\_procs**返回系统中处理器的个数。**Mpi和openmp的区别**MPI主要实现的是多主机联网协作进行并行计算，OpenMP更适合单台计算机（多核）共享内存结构上的并行计算。MPI针对的是进程，openMP针对的是线程。**MPI和openmp混合运算**在只使用MPI进行计算时，每个进程会分配一些迭代，即每个进程内部仍然使用了for循环。当使用了openmp后，每个进程for循环的的迭代有不同的线程来分担，从而实现进程间核线程间的并行计算，速度变的更快了

**MPI 消息传递函数库**的标准规范, 每个**并行进程**均有自己独立的地址空间，相互之间访问不能直接进行，必须通过显式的**消息传递**来实现。通讯器定义了**一组能够互相发消息的进程**。在这组进程中，每个进程会被分配一个序号，称作**秩**（rank），进程间显性地通过指定秩来进行通信。通信的基础建立在不同进程间发送和接收操作。一个进程可以通过指定另一个进程的秩以及一个独一无二的消息**标签（tag）**来发送消息给另一个进程。接受者可以发送一个接收特定标签标记的消息的请求（或者也可以完全不管标签，接收任何消息），然后依次处理接收到的数据。类似这样的涉及一个发送者以及一个接受者的通信被称作点对点（point-to-point）通信。**MPI 通信函数MPI\_INIT**MPI程序的初始化**MPI\_Comm\_Size** 获取通信域包含的进程数 **MPI\_Comm\_rank**获取当前进程标识**MPI\_Send (void\* buf,int count,MPI\_Datatype datatype, int dest,int tag,MPI\_Comm comm)**源程序将缓存的数据阻塞式发送到目的进程。buf : 发送缓存的起始地址**count**: 发送的数据个数**datatype**: 数据类型**des**标识号**Tag:** 消息标志**comm:** 通信域**request:** 非阻塞通信完成对象（句柄）**MPI\_Recv(void \* buf,int count,MPI\_Datatype datatype,int source,int tag,MPI\_Comm comm,MPI\_Status \* status)** Status:返回状态MPI\_Finalize():结束MPI程序的执行**MPI聚合通信**MPI\_Bcast 一对多广播同样的消息，根进程既是发送缓冲区也是接收缓冲区**MPI\_gather:** 多对一收集各个进程的消息，Root进程从n个进程的每一个接收各个进程(包括它自已)的消息. 这n个消息的连接按序号rank进行, 存放在Root进程的接收缓冲中。MPI\_Isend (\*buf, count, datatype,destination, tag, comm, MPI\_Request \*request) 非阻塞发送MPI\_Irecv (\*buf, count, datatype,source, tag, comm, MPI\_Request\*request)非阻塞接收 **MPI\_Wait(MPI\_Request \*request,MPI\_Status \*status)**通信检查，必须等待指定的通信请求完成后才能返回，成功返回时，status 中包含关于所完成的通信的消息，相应的通信请求被释放，即request 被置成MPI\_REQUEST\_NULL。阻塞通信时保证了数据的安全性，所以通信还是返回后，其他的函数或者语句能够对这些数据资源直接访问。但是非阻塞通信函数立即返回，不能保证数据已经被传送出去或者被备份或者已经读入使用，所以即使你没有阻塞，后面的语句可以继续执行，如果你要操纵上面所说的数据，将会导致发送或接收产生错误。所以对于非阻塞操作，要先调用等待MPI\_Wait()或测试MPI\_Test()函数来结束或判断该请求，然后再向缓冲区中写入新内容或读取新内容。

**OpenCl** 平台通常包括一个主机(Host)和多个OpenCL设备（device），每个OpenCL设备包括一个或多个CU(compute units)，每个CU包括又一个或多个PE（process element）。

1选择openCL平台 **clGetPlatformIDs** 第一次得到系统中可使用的平台数目，然后为（Platform）平台对象分配空间，第二次调用就是查询所有的平台，选择自己需要的OpenCL平台。2 **clCreateContest**这个函数中指定了和context关联的一个或多个设备对象,properties参数指定了使用的平台。Context是指管理OpenCL对象和资源的上下文环境。下面的一些对象都要和Context关联起来•设备（Devices）执行Kernel程序对象。•程序对象（Programobjects）:kernel程序源代码•Kernels:运行在OpenCL设备上的函数•内存对象（Memoryobjects）:设备上存放数据•命令队列（Commandqueues）:设备的交互机制•内存命令（Memory commands）(用于在主机内存和设备内存之间拷贝数据)•Kernel执行(Kernelexecution)•同步（Synchronization）3 **clCreateCommandQueue** 命令队列：主机的请求，在设备上执行的一种机制。对于不同的设备，它们都有自己的独立的命令队列 4 **clCreateBuffer**该函数会在指定的context上创建一个buffer对象，函数创建OpenCLbuffer对象后，会把对应hostbuffer的内容拷贝到OpenCLbuffer中。opencl内存对象，就是一些OpenCL数据，这些数据一般在设备内存中，能够被拷入也能够被拷出。5. 在Kernel执行之前，host中原始输入数据必须显式的传到device中，Kernel执行完后，结果也要从device内存中传回到host内存中。我们主要通过函数**clEnqueue{Read/Write}Buffer }**来实现这两种操作。clEnqueueWrite命令包括初始化内存对象以及把host数据传到device内存这两种操作。6. **clCreateProgramWithSource**通过源代码(strings)，创建一个程序对象。程序对象就是通过读入Kernel函数源代码或二进制文件，然后在指定的设备上进行编译而产生的OpenCL对象。7. **clBuildProgram** 对context中的每个设备，这个函数编译、连接源代码对象，产生device可以执行的文件 8 **clCreateKernel** Kernel对象通过程序对象以及指定的函数名字创建。Kernel就是在程序代码中的一个函数，这个函数能在OpenCL设备上执行。一个Kernel对象就是kernel函数以及其相关的输入参数。9. **clSetKernelArg**为Kernel对象设置参数。 该函数的第二个参数指定该参数为 Kernel 函数中的第几个参数 (比如第一个参数为 0，第二个为 1,…)。内存对象和单个的值都 可以作为 Kernel 参数。10. Kernel的每个运行实例称作WorkItem(也就是线程），WorkItem组织在一起称作WorkGroup，OpenCL中，每个workgroup之间都是相互独立的。通过一个globalid(在索引空间，它是唯一的）或者一个workgroupid和一个workgroup内的localid，就能标定一个workitem。WorkGroup 被映射到硬件的 CU 上执行。每个线程（workitem）都有一个 kenerl 函数的实例。**get\_global\_id(dim)，get\_global\_size(dim)** 这两个函数能得到每个维度上的global id 11 Kernel函数的写法：每个线程（workitem）都有一个kenerl函数的实例。通过函数**clEnqueueNDRangeKerne**把Kernel放在指定的命令队列里。每个线程执行的代码都是相同的，但是它们执行数据却是不同的12**clEnqueueReadBuffer**当Kernel函数执行完毕后，我们要把数据从devicememory中拷贝到hostmemory中去：13**clRelase{Resource}** 释放资源。

 