多核并行计算之 OpenMP / MPI介绍

2022.05.15

MPI 篇

MPI的定义

- MPI（Message Passing Interface）是一个库，而不是一门语言。
- MPI是一种标准或规范的代表，而不特指某一个对它的具体实现。
- MPI是一种消息传递编程模型并成为这种编程模型的代表和事实上的标准。

MPI的产生与发展

- MPI的标准化开始于1992年4月19日至30在威吉尼亚的威廉姆斯堡召开的分布存储环境中消息传递标准的讨论会，由Dongarra等建议的初始草案于1992年11月推出，并在1993年2月完成了修订版，即MPI-1.0。
- 1995年6月：MPI的新版本MPI -1.1，对原来的MPI作了进一步的修改、完善和扩充，2008年MPI-1.3标准作为MPI-1系列的最终结束发布。
  - MPI-1（1.3版本缩写）提供了C和Fortran 77的语言绑定
- 1997年7月：推出了MPI的扩充部分MPI-2.0，2009年9月发布最后一个版本MPI-2.2，MPI-2（2.2版本缩写）相当于MPI-1的超集（C/C++的关系），MPI-2的扩充很多，但主要是三个方面：并行I/O、远程存储访问和动态进程管理。
  - MPI-2支持了C++和Fortran 90接口

MPI的产生与发展

- 2012年9月：发布了MPI的第三个主要版本MPI-3.0，2015年6月发布了MPI-3.1，MPI-3（3.1版本缩写）包含非阻塞聚合通信、相邻聚合通信等新特性，并对单边通信接口进行了大量扩展。
  - MPI-3支持了Fortran 2008的语言绑定。

- 2012年9月：MPI-4.0标准发布，包含更多新的技术特性，扩展以更好地支持混合编程模型，支持MPI 应用程序中的容错，持久性集合体，性能断言和提示，RMA/单边通信等。

- 未来：MPI论坛同时在积极发布MPI-4.1和未来的MPI-5.0。

MPI Forum (mpi-forum.org)

MPI的语言绑定与实现

- MPICH

MPICH是一个与MPI规范同步发展的版本，每当MPI推出新的版本就会有相应的MPICH的实现版本，Argonne国家试验室和MSU对MPICH作出了重要的贡献。

Ubuntu20.04支持apt安装mpich

- OpenMPI

OpenMPI是一种高性能消息传递库，最初是作为融合的技术和资源从其他几个项目（FT- MPI, LA-MPI, LAM/MPI, 以及 PACX-MPI），它是MPI-2标准的一个开源实现，由一些科研机构和企业一起开发和维护。

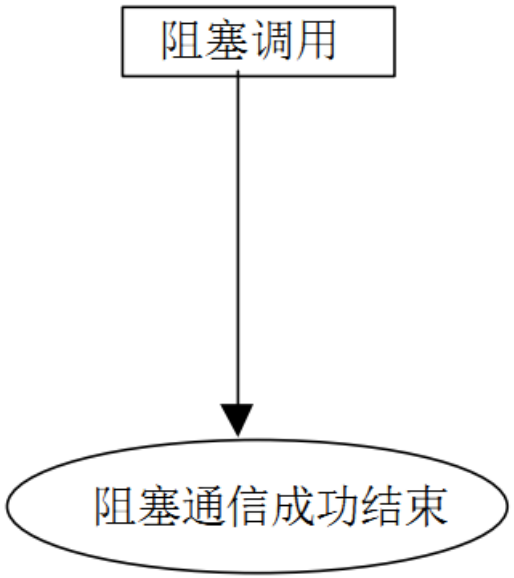
- LAM

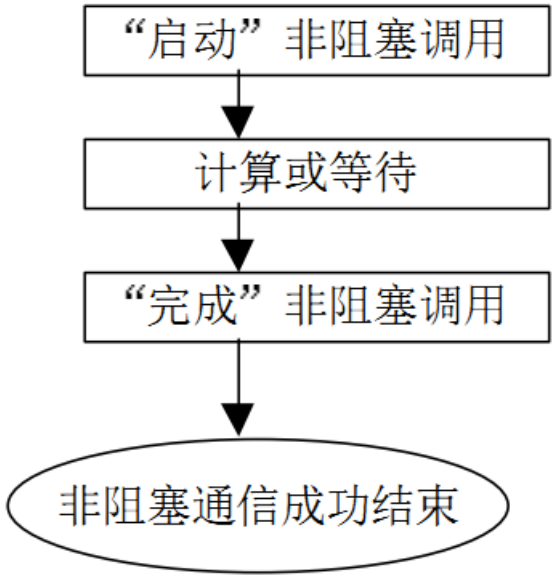
LAM (Local Area Multicomputer)也是免费的MPI实现，由Ohio State University开发。它主要用于异构的计算机网络计算系统。

Ubuntu20.04默认的mpi实现为LAM

MPI阻塞和非阻塞

对于阻塞通信，只需要一个调用函数即可以完成。但是对于非阻塞通信一般需要两个调用函数。首先是非阻塞通信的启动，但启动并不意味着该通信过程的完成，因此，为了保证通信的完成，还必须调用与该通信相联系的通信完成调用接口。通信完成调用才真正将非阻塞通信完成。





MPI函数介绍

- MPI对参数说明的方式有三种 分别是IN、OUT和INOUT。
  - IN（输入）：调用部分传递给MPI的参数，MPI除了使用该参数外不允许对这一参数做任何修改。
  - OUT（输出）： MPI返回给调用部分的结果参数，该参数的初始值对MPI没有任何意义。
  - INOUT（输入输出）： 调用部分首先将该参数传递给MPI，MPI对这一参数引用、修改后，将结果返回给外部调用，该参数的初始值和返回结果都有意义。

- MPI提供的调用虽然很多，但最常使用的只有6个，通过使用这6个函数就可以完成几乎所有的通信功能 。
  - MPI_Init; MPI_Finalize; 环境开始和结束
  - MPI_Comm_size; MPI_Comm_rank; 进程总数和当前进程号
  - MPI_Send; MPI_Recv; 消息发送和接收

int MPI_Init (int* argc ,char argv[])**

该函数通常应该是第一个被调用的MPI函数用于并行环境初始化，其后面的代码到 MPI_Finalize()函数之前的代码在每个进程中都会被执行一次。除MPI_Initialized()外，其余所有的MPI函数应该在其后被调用。MPI系统将通过argc,argv得到命令行参数（也就是说main函数必须带参数，否则会出错）。

int MPI_Finalize (void)

退出MPI系统， 所有进程正常退出都必须调用。表明并行代码的结束,结束除主进程外其它进程。串行代码仍可在主进程(rank = 0)上运行， 但不能再有MPI函数（包括MPI_Init()）。

int MPI_Comm_size (MPI_Comm comm ,int* size)

得到通信域中的总进程数。

- IN　comm　通信域（句柄）
- OUT　size　通信域comm内包括的进程数（整数）

常用宏MPI_COMM_WORLD：包含所有进程。

int MPI_Comm_rank (MPI_Comm comm ,int* rank)

得到本进程在通信域中的rank值,即在组中的逻辑编号(该 rank值为0到p-1间的整数,相当于进程的ID。)

- IN comm 该进程所在的通信域 (句柄)
- OUT rank 调用进程在comm中的标识号

int MPI_Send(void *buff, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

发送缓冲区中的count个datatype数据类型的数据发送到目的进程，目的进程在通信域中的标识号是dest，本次发送的消息标志是tag，使用这一标志，就可以把本次发送的消息和本进程向同一目的进程发送的其它消息区别开来。MPI_SEND操作指定的发送缓冲区是由count个类型为datatype的连续数据空间组成，起始地址为buf。注意这里不是以字节计数，而是以数据类型为单位指定消息的长度。其中datatype数据类型可以是MPI的预定义类型，也可以是用户自定义的类型，通过使用不同的数据类型调用MPI_SEND 可以发送不同类型的数据。

int MPI_Comm_rank (MPI_Comm comm ,int* rank)

得到本进程在通信域中的rank值,即在组中的逻辑编号(该 rank值为0到p-1间的整数,相当于进程的ID。)

- IN comm 该进程所在的通信域 (句柄)
- OUT rank 调用进程在comm中的标识号

int MPI_Send(void *buff, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

发送缓冲区中的count个datatype数据类型的数据发送到目的进程，目的进程在通信域中的标识号是dest，本次发送的消息标志是tag，使用这一标志，就可以把本次发送的消息和本进程向同一目的进程发送的其它消息区别开来。MPI_SEND操作指定的发送缓冲区是由count个类型为datatype的连续数据空间组成，起始地址为buf。注意这里不是以字节计数，而是以数据类型为单位指定消息的长度。其中datatype数据类型可以是MPI的预定义类型，也可以是用户自定义的类型，通过使用不同的数据类型调用MPI_SEND 可以发送不同类型的数据。

- IN buf 发送缓冲区的起始地址(可选类型)
- IN count 将发送的数据的个数(非负整数)
- IN datatype 发送数据的数据类型(句柄)
- IN dest 目的进程标识号(整型)
- IN tag 消息标志(整型)
- IN comm 通信域(句柄)

int MPI_Recv(void *buff, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

- OUT buf 接收缓冲区的起始地址(可选数据类型)
- IN count 最多可接收的数据的个数(整型)
- IN datatype 接收数据的数据类型(句柄)
- IN source 接收数据的来源即发送数据的进程的进程标识号(整型)
- IN tag 消息标识 与相应的发送操作的表示相匹配相同(整型)
- IN comm 本进程和发送进程所在的通信域(句柄)

- OUT status 返回状态 (状态类型)

int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

启动一个标准的非阻塞发送操作，调用后立即返回。MPI_ISEND的调用返回并不意味着消息已经成功发送，它只表示该消息可以被发送，和阻塞发送调用相比，它多了一个参数request，这一参数是一个用来描述非阻塞通信状况的对象(非阻塞通信对象)，通过对这一对象的查询，就可以知道与之相应的非阻塞发送是否完成。

- IN buf 发送缓冲区的起始地址(可选数据类型)
- IN count 发送数据的个数(整型)
- IN datatype 发送数据的数据类型(句柄)
- IN dest 目的进程号(整型)
- IN tag 消息标志(整型)
- IN comm 通信域(句柄)
- OUT request 返回的非阻塞通信对象(句柄)

int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

启动一个标准的非阻塞发送操作，调用后立即返回。MPI_ISEND的调用返回并不意味着消息已经成功发送，它只表示该消息可以被发送，和阻塞发送调用相比，它多了一个参数request，这一参数是一个用来描述非阻塞通信状况的对象(非阻塞通信对象)，通过对这一对象的查询，就可以知道与之相应的非阻塞发送是否完成。

- IN buf　　发送缓冲区的起始地址(可选数据类型)
- IN count　　发送数据的个数(整型)
- IN datatype　　发送数据的数据类型(句柄)
- IN dest　　目的进程号(整型)
- IN tag　　消息标志(整型)
- IN comm　　通信域(句柄)
- OUT request　　返回的非阻塞通信对象(句柄)

int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

- IN buf　　发送缓冲区的起始地址(可选数据类型)
- IN count　　发送数据的个数(整型)
- IN datatype　　发送数据的数据类型(句柄)
- IN dest　　目的进程号(整型)
- IN tag　　消息标志(整型)
- IN comm　　通信域(句柄)
- OUT request　　返回的非阻塞通信对象(句柄)

int MPI_Wait(MPI_Request *request, MPI_Status *status)

MPI_WAIT以非阻塞通信对象为参数，一直等到与该非阻塞通信对象相应的非阻塞通信完成后才返回。同时释放该阻塞通信对象，因此程序员就不需要再显式释放该对象。与该非阻塞通信完成有关的信息放在返回的状态参数status中。INOUT request　　非阻塞通信对象 (句柄)OUT status　　返回的状态(状态类型)

int MPI_Test(MPI_Request*request, int *flag, MPI_Status *status)

与MPI_WAIT类似，MPI_TEST也以非阻塞通信对象为参数，但是它的返回不一定等到与非阻塞通信对象相联系的非阻塞通信的结束。若在调用MPI_TEST时，该非阻塞通信已经结束，则它和MPI_WAIT的效果完全相同，完成标志flag=true。若在调用MPI_TEST时，该非阻塞通信还没有完成，则它和MPI_WAIT不同，它不必等待该非阻塞通信的完成，可以直接返回。但是完成标志flag=false，同时也不释放相应的非阻塞通信对象。

- INOUT request　　非阻塞通信对象(句柄)
- OUT flag　　操作是否完成标志(逻辑型)
- OUT status　　返回的状态 (状态类型)

int MPI_Barrier(MPI_Comm comm)

MPI_BARRIER阻塞所有的调用者直到所有的组成员都调用了它，各个进程中这个调用才可以返回。

- IN　comm　通信域(句柄)

OpenMP篇

OpenMP的定义

OpenMP是由OpenMP Architecture Review Board牵头提出的，用于共享内存并行系统的多线程程序设计的一套编译指令 (Compiler Directive)。OpenMP支持的编程语言包括C、C++和Fortran。

- fork/join执行模式

fork创建新线程或者唤醒已有线程；join即多线程的会合。标准并行模式执行代码的基本思想是，程序开始时只有一个主线程，程序中的串行部分都由主线程执行，并行的部分是通过派生其他线程来执行，但是如果并行部分没有结束时是不会执行串行部分的。在并行代码执行结束后，派生线程退出或者阻塞，不再工作，控制流程回到单独的主线程中。

- OpenMP编程要素：

  - 编译制导
  - API函数集
  - 环境变量

OpenMP常用指令与子句

- 并行域控制类

➢ parallel指令：用在一个代码段之前，表示这段代码将被多个线程并行执行。

格式：**#pragma omp parallel [for | sections] [子句[子句]…]**

- 任务分担类

➢ for指令：用于for循环之前，将循环分配到多个线程中并行执行，必须保证每次循环之间无相关性。

格式：**#pragma omp [parallel] for [子句]**

➢ sections指令：用在可能会被并行执行的代码段之前，用于非迭代计算的任务分担。

格式：**#pragma omp [parallel] sections [子句] { #pragma omp section{…代码块…}}**

➢ task指令：定义一个显式的任务，任务的执行，依赖于OpenMP的任务调度，for和sections无法根据运行时的环境动态的进行任务划分，必须是预先能知道的任务划分的情况。

格式：**#pragma omp task**

- single指令：用在一段只被单个线程执行的代码段之前，表示后面的代码段将被单线程执行，其他线程等待。

格式：**#pragma omp single [nowait] [子句]**

没有nowait则其他线程需在该指令结束处隐式同步点同步，否则其他线程继续向下执行。

- schedule子句：**schedule(type [, size])**，迭代配额策略设置。

- static：每个线程分配任务均衡
- dynamic：较快的线程可以申请到更多线程数
- guided：开始时每个线程会分配到较大的迭代块，之后分配到的迭代块会逐渐递减
- runtime：根据环境变量确定上述调度策略中的某一种，默认也是静态的（static）

- 同步控制类（互斥锁和事件同步类）

critical指令：任意大小代码块，不允许相互嵌套，未命名的可对标记临界区进行强制互斥访问，命名的被保护的代码块可以同时执行，针对操作不同变量。

格式：**#pragam omp critical [(name)]**

- atomic指令：仅作用于单条赋值语句，实现互斥访问最快。

格式：**#pragma omp atomic**

- barrier指令：同步线程。

格式：**#pragma omp barrier**

- master指令：指定由主线程执行。

格式：**#pragma omp master**

- ordered子句： **ordered**，顺序制导

● 数据环境类

- 共享与私有化

- private子句：**private(a,b,…)**,每个线程都有它自己的列表变量私有副本。
- shared子句：**shared(a,b,…)**, 列表变量被当做共享变量处理，不会产生线程私有副本。
- default子句：**default(shared|none)**，shared表示并行区域内的共享变量在不指定的情况下都是shared属性。none表示必须显式指定所有共享变量的数据属性，否则会报错，除非变量有明确的属性定义。

- firstprivate子句：**firstprivate(a,b,…)**,并行域开始执行时私有变量通过主线程中的变量初始化一次。
- lastprivate子句：**lastprivate(a,b,…)**,将最后一个线程上的私有变量赋值给主线程的同名变量。

reduction子句：**reduction(+|-|*… : a,b,c,…)**，用来在运算结束时对并行区域内的一个或多个参数执行一个操作。每个线程将创建参数的一个副本，在运算结束时，将各线程的副本进行指定的操作，操作的结果赋值给原始的参数。reduction支持的操作符是有限的，支持+ - * / += -= *= /= |&^，且不能是C++重载后的运算符。

➢ 线程专有数据

- threadprivate指令：指定全局变量被OpenMP所有的线程各自产生一个私有的拷贝，即各个线程都有自己私有的全局变量。只能用于全局变量或静态变量。

格式：**#pragma omp threadprivate(a,b,c,…)**

- copyin子句：**copyin(a,b,c,…)**，copyin子句用于将主线程中threadprivate变量的值拷贝到执行并行区域的各个线程的threadprivate变量中，从而使得team内的子线程都拥有和主线程同样的初始值。
- copyprivate子句： **copyprivate(a,b,c,…)**，用于将线程私有副本变量的值从一个线程广播到执行同一并行区域的其他线程的同一变量，只能用于single指令的子句中，在一个single块的结尾处完成广播操作。

OpenMP与Cilk Plus

Cilk多线程编程技术最早由MIT开发，是一个基于GCC编译器的开源项目。后来开发者创建了一个创业公司，推出改进的私有版本，整合到Windows下的多种编译器中。之后它被英特尔公司收购，整合进英特尔的编译器中。后续英特尔将其开源，成为GCC4.7下的一个分支。现在在Intel的官网上Cilk Plus已经被oneTBB代替。

- Cilk Plus在数据并行时对程序源代码的改动非常小，只需要将for改写成clik_for即可，Cilk Plus的底层运行库能高效地完成工作量的划分、调度，不需要我们开发人员的干预。
- 任务并行方面，Cilk Plus提供cilk_spawn和cilk_sync这两个关键字，我们可以很灵活地将基于任务的串行代码并行化，比如递归函数。
- Cilk Plus提供了一些向量化方面的支持，包括Elemental Function、Array Notation，开发人员可以使用这些功能，将向量操作以自然方式来书写，极大的增强程序的可读性，编译器也更容易将向量操作向量化。

THANKS