计算机体系结构基础

胡伟武、苏孟豪

第10章：并行编程基础

- 程序的并行行为
- 并行编程模型
  - 单任务数据并行模型
  - 多任务共享存储编程模型
  - 多任务消息传递编程模型
- 典型的并行编程环境
  - SIMD编程
  - Posix编程环境
  - OpenMP编程环境
  - MPI编程环境

程序的并行行为

指令级并行性

- 不存在相关的指令可以并行执行
  - 只有 **RAW**数据相关和控制相关制约指令级并行执行
  - 编程模型内在的并行性：把全序变成部分序
- 微结构开发指令级并行性
  - 指令流水线
  - 动态调度：允许超车
  - 多发射：多车道
- 喂饱“饥饿”的运算器
  - 转移猜测：提供足够的指令
  - 存储管理：提供足够的数据
  - 冯诺依曼结构：存储程序和顺序执行

数据级并行

- 对集合或者数组中的元素同时执行相同的操作
  - 主要来源于程序中的循环语句，常见于科学与工程计算。
- 不同编程方式均支持数据级并行
  - 现代处理器的向量功能部件
  - 向量处理器：如早期的Cray并行机，Fortran90
  - 多处理器中SPMD（单程序、多数据）编程

```
for(i=0,i<N,i++)
    local[i] = (i+0.5)*w;
```

例子：数据并行的代码示意图

任务级并行性

- 将不同的任务（进程或者线程）分布到不同的处理单元上执行，可分为进程级或者线程级并行性
- 常见于商业应用领域（如大规模数据库的事务处理）、多道程序工作负载（**Multiprogramming workload**）等

```
if(processor_ID=”a”) {
  task A;
}else if (processor_ID=”b”){
  Task B;
}
```

例子：任务并行的代码示意图

并行编程模型

单任务数据并行SIMD

- 数据并行
  - 数据并行（**data parallel**）模型是指对集合或者数组中的元素同时（即并行）执行相同操作。
- 特点
  - 单线程
  - 同构并行
  - 全局命名空间
  - 隐式相互作用（同步）
  - 隐式数据分配

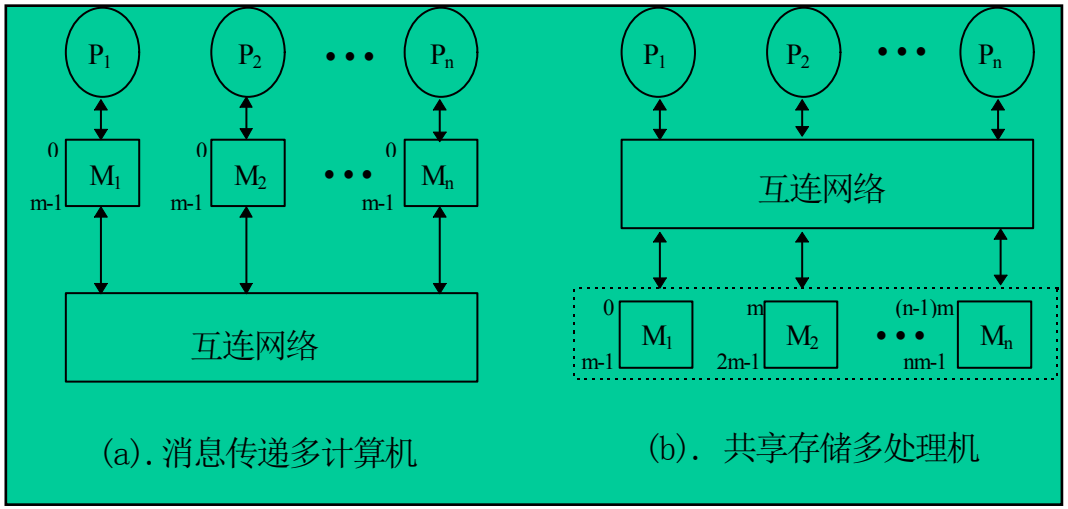
多任务并行：共享存储

- 多进程（或多线程）通过读/写共享存储器中的共享变量来相互通信
- 特点：
  - 多线程、异步执行的
  - 具有一个单一的全局名字空间（与数据并行模型类似）
  - 数据分配：不需要显式的分配数据
  - 负载分配：可以显式的分配也可以隐式的分配
  - 通信通过共享的读/写变量隐式的完成
  - 同步必须是显式的，以保持进程执行的正确顺序
- 例如：**Pthreads、OpenMP**

多任务并行：消息传递

- 在不同处理器上执行的进程，通过网络传递消息相互通信
- 特点
  - 多进程、异步的、具有独立地址空间
  - 显式相互作用：程序员必须解决包括数据映射、通信、同步和聚合等相互作用问题；通过消息传递操作来实现通信及同步
  - 负载和数据分配：程序员显式分配给进程，通过属主-计算规则来完成，即进程只能在其所拥有的数据上执行计算
  - 为了编程方便，通常采用SPMD方式编写程序
- 例如：**MPI、PVM**

两种并行系统：消息传递与共享存储



- 多地址空间
- 消息传递通讯
- 编程困难、程序移植困难
- 通用性差
- 可伸缩性好

- 单地址空间
- 共享存储通讯
- 编程容易、程序易移植
- 通用性强
- 可伸缩性一般

共享存储与消息传递的编程复杂度

- 任务划分与数据划分
  - 共享存储编程只需划分任务
  - 消息传递编程除了划分任务外，还需划分数据和考虑通信
  - **BBS与Email**
- 传递复杂的数据结构较困难
  - 多个指针组成的结构
  - **struct {int *pa; int pb*; int *pc}**
- 动态通信
  - **{for (i,j){ x=…; y=…; a[i][j]=b[x][y];}}**
  - 进程迁移及进程数目的变化

例子：积分求π

$$\pi = 4\int_0^1 \frac{1}{1+x^2}dx = \sum_{i=1}^{N} \frac{4}{1+\left(\frac{i-0.5}{N}\right)^2} \times \frac{1}{N}$$

串行程序

```
h = 1.0/N; pi = 0;
for(i=1; i<=N; i++){
        temp = (i-0.5)*h;
        pi = pi + 4/(1+temp*temp);
}
pi = pi * h;
printf pi;
```

积分求π的并行程序

JIAJIA

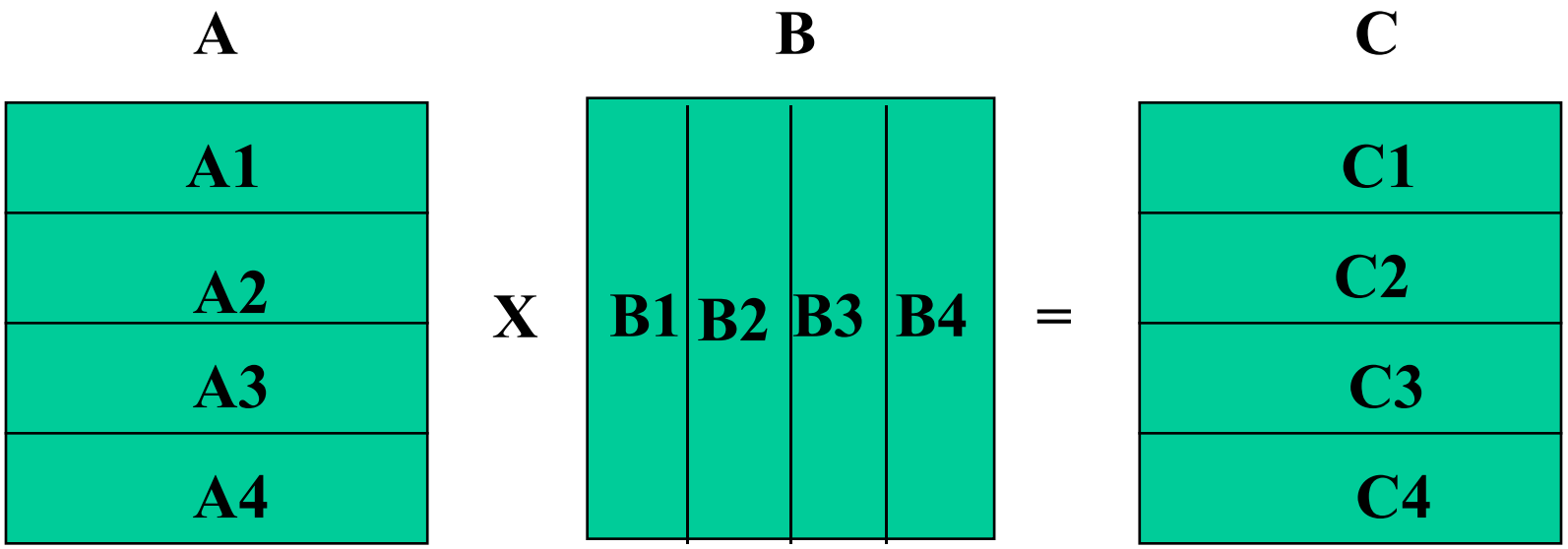
```
jia_init(argc,argv);
pi=jia_alloc(8);

h = 1.0/N;
for(i=jiapid+1;i<=N;i+=jiahosts)
   { mypi = ... }
jia_lock(1);
   pi += mypi;
jia_unlock(1);
printf pi;
jia_exit();
```

MPI

```
MPI_Init(argc,argv);
MPI_Comm_size();
MPI_Comm_rand();
h = 1.0/N;
for(i=myid+1;i<=N;i+=numprocs)
   { mypi = ... }
MPI_Reduce(mypi,pi,1   ... );
printf pi;
MPI_Finalize();
```

并行程序例子：矩阵乘法



- A × B = C的过程可分为四个独立的部分：

  Ai × B = Ci，i = 1, 2, 3, 4

- 每部分包含的运算可由一台处理机单独完成
- 矩阵较大时，需分开存放，即Ai,Bi,Ci放在结点i上
- 每个结点的工作分为AiB1,AiB2,AiB3,AiB4四个部分

矩阵乘法的并行程序

- 共享存储

```
double (*a)[N],(*b)[N],(*c)[N];
a=jia_alloc(N*N*8);
b=jia_alloc(...); c=jia_alloc(...);
if (jiapid==0) for (i...) for (j…){
  a[i][j]=1;b[i][j]=1;
}
jia_barrier();
begin=N*jiapid/jiahosts;
end=N*(jiapid+1)/jiahosts;
for (i=begin; i<end; i++)
    for (j=0; j<N; j++)
          for (k=0; k<N; k++)
            c[i][j]+=a[i][k]*b[k][j];
jia_barrier();
if (jiapid==0) printf C;
jia_exit();
```

- 消息传递

```
double (*a)[N],(*b)[N],(*c)[N];
a=malloc2(N*N*8);
b=malloc2(...); c=malloc(...);
if (mypid==0) {
  init a,b; send a,b;
}else{
  recv a,b;
}

for (i=0;i<N/hosts;i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      c[i][j]+=a[i][k]*b[k][j];

if (mypid!=0){  send c;}
else{ recv c; printf c;}
```

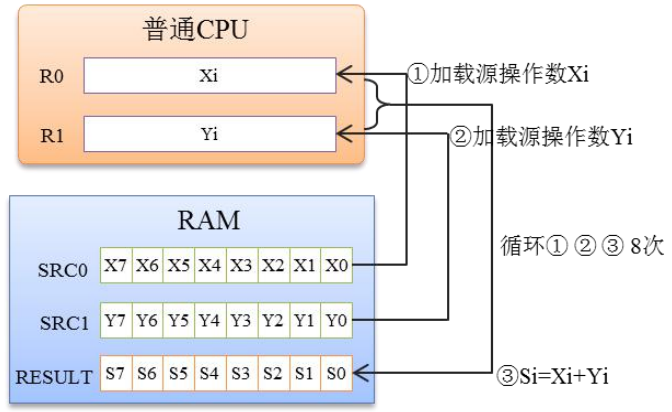
常见的并行处理结构

- SIMD结构
  - 早期Cray向量机，现代CPU的SIMD短向量指令（128、256、512位）
- SMP结构：多核共享存储
  - 早期多路服务器/工作站（DEC、SUN、SGI），现在片内多核
- CC-NUMA结构：更多核共享存储
  - SGI、IBM、HP高端服务器，几十到上千路共享内存服务器
  - 片内核数增加导致CC-NUMA
- MPP或机群结构
  - HPC，如曙光高性能机、太湖之光，主要用于科学计算
  - 云计算，机群数据库
- GPU（上千核）采用什么结构？

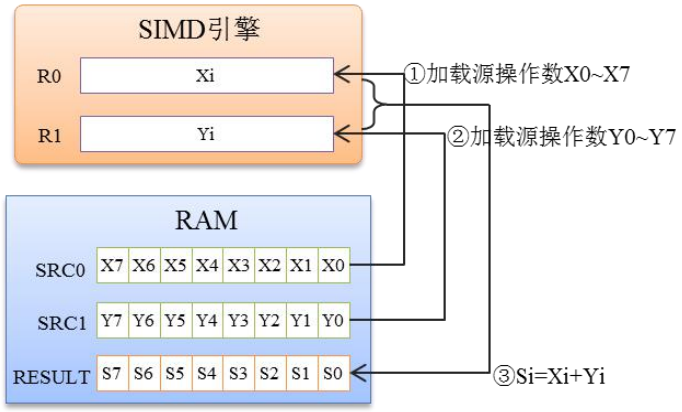
典型并行编程环境

数据并行编程-SIMD

- **SIMD编程**
  - **一条SIMD指令可以同时对一组数据进行相同的计算**
  - **龙芯CPU：gsldxc1（向量读）、gssdxc1（向量写）、paddb（向量加）**



单指令流单数据流(SISD, Single Instruction Single Data)



单指令流多数据流(SIMD, Single Instruction Multiple Data)

```
for(i=0; i<8; i++)
    resule[i] = src0[i] + src1[i];
```

C语言代码

```
gsldxc1 $f0, 0x0($src0, $0)
gsldxc1 $f2, 0x0($src1, $0)
paddb   $f0, $f0, $f2
gssdxc1 $f0, 0x0($result, $0)
```

龙芯SIMD向量指令代码

POSIX Threads （Pthread）标准

- **POSIX（Portable Operating System Interface）Threads：共享存储编程标准**
  - **官方IEEE POSIX1003.1C_1995线程标准**
  - **主要包含线程管理、线程调度、同步等原语**
  - **体现为C语言的一套函数库**

pthreads中基本线程管理一览表

功能	定义
int pthread_create(pthread_t *thread_id, pthread_attr_t *attr, (void *) (*myroutine)(void *), void *arg)	生成线程
void pthread_exit(void *status)	退出线程
int pthread_join(pthread_t thread,void ** status)	等待线程结束
pthread_t pthread_self(void)	获得调用线程ID

POSIX Threads标准（Cont.）

- **线程调度及同步原语**

表：pthreads中线程同步原语	
定义	功能
int pthread_yield(void)	调用者放弃处理器给其它线程
int pthread_cancel(pthread_t thread)	发终止信号给指定的线程
pthread_mutex_init(...)	生成新的互斥变量
pthread_mutex_destroy(...)	销毁互斥变量
pthread_mutex_lock(...)	锁住互斥变量
pthread_mutex_trylock(...)	尝试锁住互斥变量
pthread_mutex_unlock(...)	解锁互斥变量
pthread_cond_init(...)	生成新的条件变量
pthread_cond_destroy(...)	销毁条件变量
pthread_cond_wait(...)	等待（阻塞）条件变量
pthread_cond_timedwait(...)	等待条件变量直至到达时限
pthread_cond_signal(...)	投递一个事件，解锁一个等待进程
pthread_cond_broadcast(...)	投递一个事件，解锁所有等待进程

Pthreads并行程序举例-利用梯形规则求π

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4 //假设线程数目为4
int num_steps = 1000000;
double step = 0.0, sum = 0.0;
pthread_mutex_t mutex;

void *countPI(void *id) {
  int index = (int ) id;
  int start = index*(num_steps/NUM_THREADS);
  int end; double x = 0.0, y = 0.0;
  if (index == NUM_THREADS-1)
          end = num_steps;
  else
           end = start+(num_steps/NUM_THREADS);
  for (int i=start; i<end; i++)  {
             x=(i+0.5)*step;
             y +=4.0/(1.0+x*x);
  }
  pthread_mutex_lock(&mutex);
  sum += y;
  pthread_mutex_unlock(&mutex);
}
```

```
int main() {
      int i;
      double pi;
      step = 1.0 / num_steps;
      sum = 0.0;
      pthread_t tids[NUM_THREADS];

      pthread_mutex_init(&mutex, NULL);
      for(i=0; i<NUM_THREADS; i++) {
         pthread_create(&tids[i], NULL, countPI, (void *) i);
      }

      for(i=0; i<NUM_THREADS; i++)
         pthread_join(tids[i], NULL);

      pthread_mutex_destroy(&mutex);
      pi = step*sum;
      printf("pi %1f\n", pi);
      return 0;
}
```

OpenMP共享存储编程标准

- **OpenMP Architecture Review Board牵头提出**
  - **最初形成于1997年，2002年OpenMP2.0，2008年OpenMP3.0、2013年OpenMP4.0**
- **主要API（Application Program Interface）包含**
  - **制导指令、运行库和环境变量**
  - **针对基本编程语言（C、C++和Fortran）进行编译制导扩展，用注释符“#”符开始，不支持OpenMP的编译器中可以忽略，支持OpenMP的编译器才有用**
- **由于GCC的支持，主流处理器都支持OpenMP编译器**
  - **如Intel、AMD、IBM、龙芯等**

OpenMP的并行执行模型

- **一个OpenMP进程由多个线程组成，使用fork-join并行模型**
  - **OpenMP程序始于一个主线程（Master Thread），主线程串行执行，遇到一个并行域（parallel region）开始并行执行。**
  - **Fork：主线程创建一队并行线程，并行域的代码在不同线程中并行执行**
  - **Join：当主线程在并行域中执行完后，它们或被同步或被中断，所计算的结果会被主线程收集，最后只有主线程在执行**

```
#include <omp.h>
main(){
  int var1,var2,var3;
  ……
  #pragma omp parallel private(var1,var2)  shared(var3)
 { …… }
}
```

OpenMP程序的并行结构

OpenMP程序举例-积分法计算π值

```
#include <stdio.h>
#include <math.h>
int main(){
int i;
int num_steps=1000000;
double x,pi,step,sum=0.0;
step = 1.0/(double) num_steps;

for(i=0;i<num_steps;i++)  {
     x=(i+0.5)*step;
     sum = sum+4.0/(1.0+x*x);
}
   pi = step*sum;
   printf("pi %1f\n", pi);
   return 0;
}
```

利用梯形规则计算π的C串行代码

```
#include <stdio.h>
#include <omp.h>
int main(){
int i;
int num_steps=1000000;
double x,pi,step,sum=0.0;
step = 1.0/(double) num_steps;
#pragma omp parallel for private(i,x),reduction(+:sum)
for(i=0;i<num_steps;i++)   {
    x=(i+0.5)*step;
    sum = sum+4.0/(1.0+x*x);
}
pi = step*sum;
printf("pi %1f\n", pi);
return 0;
}
```

利用梯形规则计算π的OpenMP并行代码

OpenMP的制导

- **编译制导语句的格式**

#pragma omp	Directive-name	[clause,…]	newline
指导指令前缀。对所有的OpenMP语句都需要这样的前缀	OpenMP制导指令。在制导指令前缀和子句之间必须有一个正确的OpenMP制导指令	子句。在没有其他约束条件下，子句可以无序，也可以任意地选择。这一部分也可以没有	换行符。表明这条制导语句结束

例：#pragma omp parallel private(var1,var2)　shared(var3)

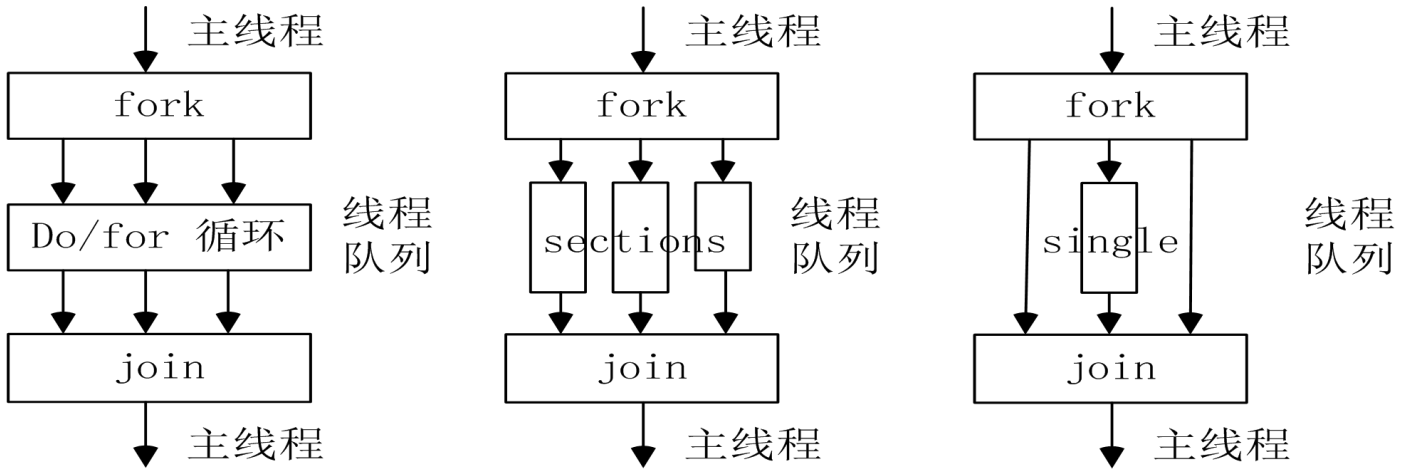
OpenMP的制导-并行域结构

- 一个并行域就是一个能被多个线程执行的程序块，它是最基本的OpenMP并行结构
  - 当一个线程执行到parallel这个指令时，就会生成一列线程，线程号依次从0到n-1，而自己会成为主线程（线程号为0）。线程数n由下面方法确定，优先前面方法，即：（1）使用库函数omp_set_num_threads()来设定；（2）设置环境变量，即OMP_NUM_THREADS；（3）所使用的默认值。

#pragma omp parallel [if(scalar_expression) | private(list) |shared(list) | default(shared|none) |firstprivate(list)|reduction(operator:list) |copyin(list) newline

OpenMP的制导-共享任务结构

- 共享任务结构将其内封闭的代码段划分给线程队列中的各线程执行，有3种类型
  - **do/for**：将循环分布到线程列中执行，表达数据并行
  - **sections**：将任务分成多个**setcion**，每个线程执行一个**section**，表达任务并行
  - **single**：由线程列中的一个线程串行执行



OpenMP的制导-组合并行共享任务结构

- **parallel for编译制导语句：该语句表明一个并行域包含一个单独的for语句。格式如下：**

#pragma omp parallel for [if(scalar_logical_expression) |default(shared|none) |schedule(type[, chunk]) shared(list) |private(list) |firstprivate(list) |lastprivate(list) |reduction(operator:list) |copyin(list)] newline

- **parallel sections编译制导语句：该语句表明一个并行域包含单独的一个sections语句语句。格式如下：**

#pragma omp parallel sections [default(shared|none) | shared(list) |private(list) |firstprivate(list) |lastprivate(list) |reduction(operator:list) |copyin(list) |ordered] newline

OpenMP的制导-同步结构

- **master编译制导语句：表明一个只能被主线程执行的域**
  - **格式为：#pragma omp master newline**
- **critical编译制导语句：表明域中代码一次只能由一个线程执行**
  - **格式为： #pragma omp critical[name] newline**
- **barrier编译指导语句：同步线程队列中的所有线程**
  - **格式为： #pragma omp barrier newline**
- **atomic编译制导语句：表明一个特别的存储单元只能原子的更新，而不允许让多个线程同时去写。**
  - **格式为：#pragma omp atomic newline**

OpenMP的制导-数据域属性子句

- **数据域属性子句用来定义变量范围，一般与parallel、for和sections语句配合使用**
  - **private子句：表示它列出的变量对于每个线程是局部的**
  - **shared子句：表示它列出的变量被线程队列中的所有线程共享**
  - **default子句：表示并行域的所有变量的默认属性（如可以是private、shared、none）**
  - **firstprivate子句：含private子句的操作，初始化为并行域外同名变量值**
  - **lastprivate子句：含private子句的操作，并将值复制给并行域外同名变量**
  - **threadprivate编译制导语句：将全局变量变成每个线程私有**
  - **copyin子句：赋予线程中变量与主线程中threadprivate同名变量的值**
  - **reduction子句：用来归约其列表中出现的变量。**

MPI（Message Passing Interface）标准

- MPI定义了一组消息传递函数库的编程接口标准
  - 1994年发布MPI-1，1997年发布MPI-2，2012年发布MPI-3
  - 支持C、C++、Fortran
  - 编译器：mpicc、mpic++、mpif90
  - 对硬件要求简单，不需要共享存储环境的支持
- MPI实现
  - 开源实现，如MPICH（Argonne National Laboratory (ANL) and Mississippi State University）、Open MPI、LAM/MPI（Ohio Supercomputer Center）等
  - 商业实现来自于Intel、Microsoft、HP等
- 具有高可移植性和易用性

最基本的MPI

- **MPI编程模型**
  - 由一个或多个彼此通过调用库函数进行消息收、发通信的进程所组成
- **MPI是复杂系统，但只要6个基本函数就能编写MPI程序**

序号	函数名	用途
1	MPI_Init（）	初始化MPI执行环境
2	MPI_Finalize	结束MPI执行环境
3	MPI_COMM_SIZE	确定进程数
4	MPI_COMM_RANK	确定自己的进程标识符
5	MPI_SEND	发送一条消息
6	MPI_RECV	接收一条信息

MPI函数格式举例-MPI_SEND&MPI_RECV

MPI_SEND（buf, count, datatype, dest, tag, comm）
//发送消息
IN buf address of send buffer(choice)
IN count number of elements in send buffer(integer≥0)
IN datatype datatype of send buffer elements(handle)
IN dest rank of destination(integer)
IN tag message tag(integer)
IN comm communicator(handle)

MPI_RECV（buf, count, datatype, source, tag, comm, status）
//接收消息
OUT buf address of receive buffer(choice)
IN count number of elements in receive buffer(integer≥0)
IN datatype datatype of each receive buffer elements(handle)
IN source rank of source or MPI_ANY_SOURCE (integer)
IN tag message tag or MPI_ANY_TAG (integer)
IN comm communicator(handle)
OUT status status object (Status)

点对点通信（MPI_SEND/MPI_Recv）

- 阻塞和非阻塞两种方式
  - 阻塞方式：必须等到消息从本地送出之后才可以执行后续的语句，保证了缓冲区等资源可再用
  - 非阻塞方式：不须等到消息从本地送出就可执行后续的语句，非阻塞调用的返回并不保证资源的可再用性
- 四种通信模式
  - 标准模式：是否缓存由**MPI**决定
  - 缓冲模式：在相匹配的接收未开始的情况下，将送出的消息放在缓冲区内，这样发送者可以很快地继续计算，多一次内存拷贝
  - 同步模式：只有接收操作开始后，发送才能返回
  - 就绪模式：只有接收操作启动后，发送操作才能开始

点对点通信函数

	标准模式	缓冲模式	同步模式	就绪模式
阻塞方式	MPI_SEND、MPI_RECV	MPI_BSEND、MPI_BRECV	MPI_SSEND、MPI_SRECV	MPI_RSEND、MPI_RRECV
非阻塞方式	MPI_ISEND、MPI_IRECV	MPI_IBSEND、MPI_IBRECV	MPI_ISSEND、MPI_ISRECV	MPI_IRSEND、MPI_IRRECV

集体通信

- 并行程序中经常需要一些进程组间的集体通信（Collective Communication）
  - 路障（MPI_BARRIER）：同步所有进程；
  - 广播（MPI_BCAST）：从一个进程发送一条数据给所有进程；
  - 收集（MPI_GATHER）：从所有进程收集数据到一个进程；
  - 散播（Scatter）：从一个进程散发多条数据给所有进程；
  - 归约（MPI_REDUCE、MPI_ALLREDUCE）：包括求和、求积等。

MPI程序例子-利用梯形规则计算π

```
#include <stdio.h>
#include “mpi.h”
int main(int argc, char **argv){
  int num_steps=1000000;
  double x,pi,step,sum,sumallprocs;
  int  i,start, end,temp;
  int num_procs; //组中的进程数量,
  int ID;  //进程编号,范围为0到num_procs-1
  MPI_Status status;

//Initialize the MPI environment
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&ID);
  MPI_Comm_size(MPI_COMM_WORLD,
                                  &num_procs);
```

```
//任务划分并计算
step = 1.0/num_steps;
start = ID *(num_steps/num_procs) ;
if (ID == num_procs-1)
     end = num_steps;
else
     end = start + num_steps/num_procs;
for(i=start; i<end;i++) {
    x=(i+0.5)*step;
    sum += 4.0/(1.0+x*x);
}
MPI_Barrier(MPI_COMM_WORLD);
MPI_Reduce(&sum,&sumallprocs,1,MPI_D
OUBLE,MPI_SUM,0, MPI_COMM_WORLD);
if(ID==0) {
    pi = sumallprocs*step;
    printf(“pi %1f\n”, pi);   }
 MPI_Finalize();
 return 0;}
```

作业