

IPCC

## 第三讲-并行开发技术概论

时间：2021年6月

主讲人：张力越

# 上讲回顾：性能优化方法论

- 1、优化方法
  - 自顶向下优化方法论
  - 通用的从收集数据-确认瓶颈-提出解决方案-应用方法-评估测试的优化循环
- 2、性能度量的指标
  - Amdahl定律，加速比，Flops，Cache miss率，CPU利用率，网络IO速率等
- 3、性能分析工具
  - perf, Gprof, valgrind
  - 并行科技的paramon paratune
- 4、案例分析
  - 矩阵乘法, stencil：优化cache利用率

# 本讲概要：

- 1、比赛平台使用示例
  - 2、OpenMP编程基础与示例
  - 3、MPI编程入门
-

# 比赛平台使用示例

---

## 1.1 北京超级云计算中心 A3分区计算资源

- 登录节点
- CPU 双路 Intel Xeon Silver 4208 8核 @2.1Ghz
- 内存 192GB
- 主要用于作业提交、结果查看、软件环境调配等
- 切勿在登录节点跑计算作业或长时间全核编译
- 计算节点
- CPU 双路 AMD EPYC 32核 @2.35Ghz
- 内存 256GB 每cpu各8通道
- 56Gb InfiniBand 互联网络
- 通过slurm系统管理使用

## 1.2 slurm简介

- SLURM (Simple Linux Utility for Resource Management) 是一种可用于大型计算节点集群的高度可伸缩和容错的**集群管理器**和**作业调度系统**，被世界范围内的超级计算机和计算集群广泛采用。
- SLURM 维护着一个待处理工作的队列并管理此工作的整体资源利用。
- SLURM 会为任务队列合理地分配资源，并监视作业至其完成。

## 1.2 slurm命令

- BSCC- A3分区使用 Slurm 作业管理系统，下面是slurm一些重要命令

命令	功能介绍	常用命令例子
sinfo	显示系统资源使用情况	sinfo
squeue	显示作业状态	squeue
srun	用于交互式作业提交	srun -N 1 -n 64 -p amd_256 A.exe
sbatch	用于批处理作业提交	sbatch -N 1 -n 64 job.sh
salloc	用于分配模式作业提交	salloc -p amd_256
scancel	用于取消已提交的作业	scancel JOBID
scontrol	用于查询节点信息或正在运行的作业信息	scontrol show job JOBID
sacct	用于查看历史作业信息	sacct -u sc90001 -S 01/01/21 -E 01/15/21 --field=jobid,partition,jobname,user,nnodes,start,end,elapsed,state

# 1.2 slurm 使用 sinfo

- sinfo
- 图中可以看到可用队列为amd\_256
- idle可以申请使用的节点数量为53个，以及他们对应的hostname

```
[sc92571@ln112%bscc-a3 ~]$ sinfo
PARTITION AVAIL TIMELIMIT NODES STATE NODELIST
all      inact  infinite    2 drain* ea[0409,1101]
all      inact  infinite    2 down* eb0803,ec0404
all      inact  infinite   22 drain ea[0108,0214,0512,0908,0913,1211,1315],eb0813,ec[010
6,0110,0215,0606,0808,0810],ed[0315,0405,0415,0512-0516]
all      inact  infinite   544 alloc ea[0101-0107,0109-0116,0201-0202,0204-0213,0215-0216
,0301-0307,0309-0316,0401-0408,0410-0416,0501-0511,0513-0516,0601-0616,0701-0716,0801-0816,09
01-0907,0909-0912,0914-0916,1001-1016,1102-1116,1201-1210,1212-1216,1301-1314,1316],eb[0202-0
204,0601-0605,0701-0716,0801-0802,0804-0807,0809-0812,0814-0816,0901-0908,0910-0916,1001-1016
,1101-1108,1110-1116,1201-1216,1301-1316],ec[0101-0105,0107-0109,0111-0116,0201-0214,0216,030
1-0316,0401-0403,0405-0416,0501-0516,0601-0605,0607-0608,0610-0616,0701-0707,0709-0716,0801,0
803-0805,0807,0809,0811-0812,0814-0816,0901-0905,0907-0909,0911-0915,1002-1007,1010-1012,1014
-1016,1101,1106-1116,1201-1204,1206-1212,1214-1216],ed[0101-0108,0110-0116,0201-0207,0209-021
1,0213-0216,0301-0304,0306-0314,0316,0401,0403-0404,0406-0407,0409-0412,0414,0501-0511,0705]
all      inact  infinite   55  idle eb[0201,0808],ec[0609,0708,0802,0806,0813,0906,0910,
0916,1001,1008-1009,1013,1102-1105,1205,1213],ed[0109,0208,0212,0402,0408,0413,0416,0601-0616
,0701-0704,0706-0713]
all      inact  infinite    5  down ea[0203,0308],eb[0909,1109],ed0305
amd_256*  up   infinite    2 drain* ea[0409,1101]
amd_256*  up   infinite    2 down* eb0803,ec0404
amd_256*  up   infinite   22 drain ea[0108,0214,0512,0908,0913,1211,1315],eb0813,ec[010
6,0110,0215,0606,0808,0810],ed[0315,0405,0415,0512-0516]
amd_256*  up   infinite   528 alloc ea[0101-0107,0109-0116,0201-0202,0204-0213,0215-0216
,0301-0307,0309-0316,0401-0408,0410-0416,0501-0511,0513-0516,0601-0616,0701-0716,0801-0816,09
01-0907,0909-0912,0914-0916,1001-1016,1102-1116,1201-1210,1212-1216,1301-1314,1316],eb[0202-0
204,0601-0605,0701-0716,0801-0802,0804-0807,0809-0812,0814-0816,0901-0908,0912-0916,1001-1016
,1101-1108,1110-1116,1201-1216,1301-1316],ec[0101-0105,0107-0109,0111-0116,0201-0214,0216,030
1-0316,0401-0403,0405-0416,0501-0516,0601-0605,0607-0608,0610-0616,0701-0707,0709-0716,0801,0
803-0805,0807,0809,0811-0812,0814-0816,0901-0905,0907-0909,0911-0915,1002-1007,1010-1012,1014
-1016,1101,1106-1116,1201-1204,1206-1212,1214-1216],ed[0101-0108,0110-0116,0201-0207,0209-021
1,0213-0216,0301-0303,0308-0314,0316,0401,0403-0404,0406-0407,0409-0412,0414,0511]
amd_256*  up   infinite   53  idle eb0808,ec[0609,0708,0802,0806,0813,0906,0910,0916,10
01,1008-1009,1013,1102-1105,1205,1213],ed[0109,0208,0212,0402,0408,0413,0416,0601-0616,0701-0
704,0706-0711,0713]
amd_256*  up   infinite    4  down ea[0203,0308],eb[0909,1109]
```

## 1.2 slurm 使用 sbatch

- sbatch提交的作业脚本：helloworld.slurm

```
#!/bin/bash
#SBATCH -J IPCC
#SBATCH -p amd_256
#SBATCH -N 1
#SBATCH --exclusive
```

```
source /public1/soft/modules/module.sh
echo "hello world"
```

-J是任务名称  
-p amd\_256指明使用amd\_256队列  
-N 1 申请的节点数量  
--exclusive 表示独占以防万一撞车

**sbatch helloworld.slurm # 提交，运行屏幕输出在slurm-xxx.out**

```
[sc92571@ln112%bscc-a3 ~]$
[sc92571@ln112%bscc-a3 ~]$ sbatch helloworld.slurm
Submitted batch job 398242
[sc92571@ln112%bscc-a3 ~]$ cat slurm-398242.out
hello world
[sc92571@ln112%bscc-a3 ~]$
[sc92571@ln112%bscc-a3 ~]$
```

## 1.2 slurm 使用 srun

```
srun -p amd_256 -w c[1100-1101] -n 64 -t 20 A.exe
```

交互式提交 A.exe 程序。如果不关心节点和时间限制，可简写为 srun -p amd\_256 -n 64 A.exe

其中，

-p amd\_256 指定提交作业到 amd\_256 队列；

-w c[1100-1101] 指定使用节点 c[1100-1101]；

-n 64 指定进程数为 64，BSCC-A3 超算 amd\_256 队列每一个节点 64 核，按核计费；

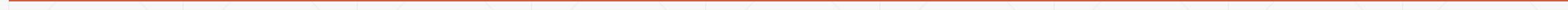
-t 20 指定作业运行时间限制为 20 分钟。

srun 的一些常用命令选项：

参数选项	功能
-N 3	指定节点数为 3
-n 64	指定进程数为 64
-c 64	指定每个进程（任务）使用的 CPU 核为 64
-p amd_256	指定提交作业到 amd_256 队列
-w c[100-101]	指定提交作业到 c100、c101 节点
-x c[100,106]	排除使用 c100、c106 节点

## 1.2 slurm 使用 srun

- srun可以理解成包装了mpirun 会自动mpirun -host到sbatch申请的节点里运行
- 如果srun -N 节点超过了sbatch申请的数量则会报错：
- srun: error: Only allocated 1 nodes asked for 2
- 建议将srun写在sbatch提交的脚本中，这样可以避免网络问题导致运行中断



## 1.2 slurm 使用 sbatch+srub

- 一个sbatch+srub的组合例子： test.slurm

```
#!/bin/bash
#SBATCH -J IPCC
#SBATCH -p amd_256
#SBATCH -N 2
#SBATCH --exclusive

source /public1/soft/modules/module.sh

srun -N 2 -n 4 hostname

# (2节点共4进程运行输出hostname)
```

运行sbatch test.slurm得到下图所示结果

```
[sc92571@ln112%bscc-a3 ~]$ sbatch test.slurm
Submitted batch job 398297
[sc92571@ln112%bscc-a3 ~]$ cat slurm-398297.out
eb0202.para.bscc
ec0113.para.bscc
eb0202.para.bscc
ec0113.para.bscc
[sc92571@ln112%bscc-a3 ~]$ █
```

## 1.2 slurm使用 squeue

squeue查看已经提交的作业情况

```
[sc92571@ln112%bscc-a3 ~]$ squeue
      JOBID PARTITION      NAME      USER ST          TIME  NODES NODELIST (REASON)
      401616  amd_256      ipcc    sc92571 R      0:05      2 eb1202,ec0910
[sc92571@ln112%bscc-a3 ~]$ squeue
      JOBID PARTITION      NAME      USER ST          TIME  NODES NODELIST (REASON)
      401616  amd_256      ipcc    sc92571 R      0:06      2 eb1202,ec0910
[sc92571@ln112%bscc-a3 ~]$ squeue
      JOBID PARTITION      NAME      USER ST          TIME  NODES NODELIST (REASON)
      401616  amd_256      ipcc    sc92571 R     1:30      2 eb1202,ec0910
```

# 1.2 slurm 使用 scancel

scancel 终止正在运行或者队列中等待运行的作业

比如终止401616号作业：

```
[sc92571@ln112%bscc-a3 ~]$ squeue
      JOBID PARTITION      NAME      USER ST          TIME  NODES NODELIST (REASON)
      401616  amd_256      ipcc    sc92571 R      1:30      2 eb1202,ec0910
[sc92571@ln112%bscc-a3 ~]$ cat slurm-401616.out
[sc92571@ln112%bscc-a3 ~]$ squeue
      JOBID PARTITION      NAME      USER ST          TIME  NODES NODELIST (REASON)
      401616  amd_256      ipcc    sc92571 R      2:11      2 eb1202,ec0910
[sc92571@ln112%bscc-a3 ~]$ scancel 401616
[sc92571@ln112%bscc-a3 ~]$ squeue
      JOBID PARTITION      NAME      USER ST          TIME  NODES NODELIST (REASON)
```

# 1.3 module 软件环境

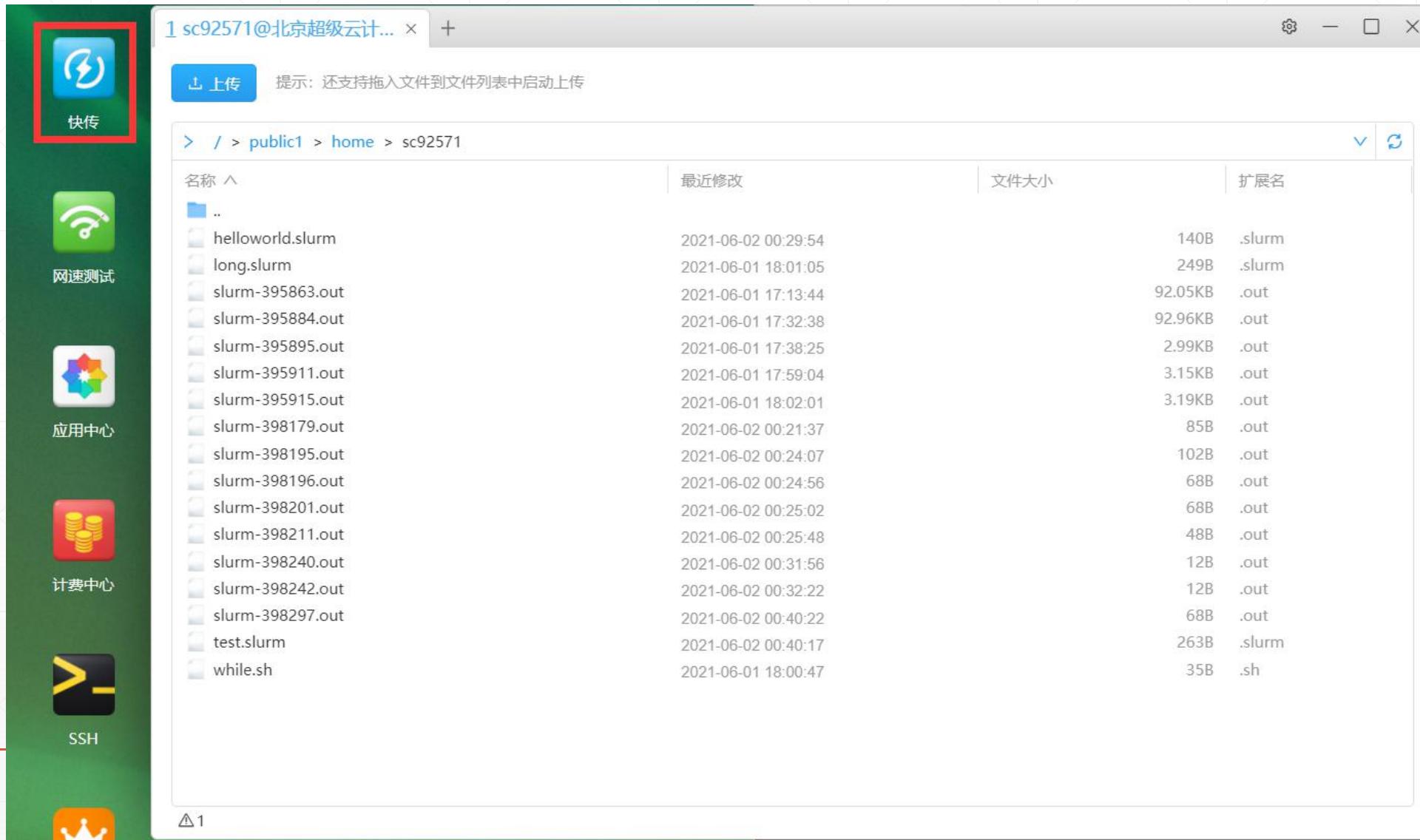
- source /public1/soft/modules/module.sh
- module avail
- module load xxx
- module unload xxx
- module list
- module purge

```
-- /public1/soft/modulefiles --
abinit/8.6.1          libxc/4.3.4-icc17      wrf/4.2
abinit/8.10.3         libxsmm/1.15-icc17    zlib/1.2.7
amber/18-gcc          manta/1.6.0          zlib/1.2.11
amber/18-intel        module-git           zstd/1.4.9
anaconda/2-Python2.7  module-info          module
anaconda/3-python-3.6.5-new  modules
anaconda/3-Python-3.6.5-phonopy   mpc/1.0.2
anaconda/3-python-3.7.9       mpfr/3.1.3
anaconda/3-python3.8       mpi/intel/20.4.3
angsd/0.934           mpi/intel/2015-public3
bedtools/2.30          mpi/intel/2017.5
BLAS/3.5.0            mpi/intel/2018.4
blast/blast-2.9       mpi/intel/2019.3.199
boost/163-intel17-kd   mpi/openmpi/1.6.5-nanodcal-kd
boost/164-intel        mpi/openmpi/2.0.4-gcc-4.9.0
boost/172-gcc-kd       mpi/openmpi/3.0.0-gcc
cmake/3.0.2            mpi/openmpi/3.1.4-gcc
cmake/3.19.4-wxl      mpi/openmpi/3.1.4-icc19
cmake/3.20.1           mpich/3.2-gcc
CNVnator/0.4.1         mpich/3.3.1
cp2k/6.1-icc17         namd/2.8
cp2k/6.1-plumed2-icc17 namd/2.12
cp2k/7.1-icc17         ncl/6.6.2
cp2k/8.1               nco/4.7.6
dot                   ncview/2.1.7
EIG/6.1.4              netcdf/3.6.3-icc17
elpa/intel17/2016.05.004 netcdf/4.4.1-icc17
elpa/intel17/2018.11.001 netcdf/4.6.3-gcc4.8.5
elpa/intel17/2019.11.001 netcdf/4.6.3-hpcx-intel20
emacs/emacs-27.2-cyc   netcdf/4.6.3-impi20-gcc7.3.0
esmf/8.0.1-intel20     netcdf/4.6.3-intel17
fftw/3.3.4-gcc          netcdf/4.6.3-intel17-gcc7.3.0
fftw/3.3.8-f            netcdf/4.6.3-intel20
fftw/3.3.8-intelmpi17-openmp-single null
fftw/3.3.8-mpi          octave/4.4.0
:
```

## 1.3 module 常用包

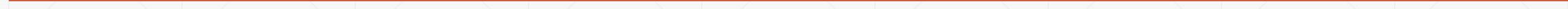
- 一些常用的包（按需使用，不要无脑load，导致环境冲突）
- module load intel/20.4.3
- module load mpi/intel/20.4.3
- module load gcc/7.3.0-kd
- module load mpi/openmpi/3.1.4-gcc
- 更多软件包可以 module avail 查看

# 1.4 文件传输



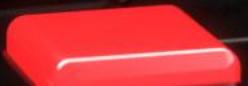
## 1.5 小结

- 编译节点不要搞大新闻
- 相比普通集群申请之后登录到计算节点机器操作的方式，此次比赛使用的A3分区不推荐也不能ssh到计算节点
- 由于登录节点跟计算节点环境不一样（CPU分别来自intel和amd）所以最好到计算节点进行编译，也就是把编译命令写在slurm脚本中提交运行
- 答疑~



# OpenMP编程

---



# OpenMP编程

- 1. OpenMP编程简介
  - 2. OpenMP编程制导
  - 3. OpenMP库函数
  - 4. OpenMP环境变量
  - 5. OpenMP并行注意的问题
-

# OpenMP编程

- 1. OpenMP编程简介
  - 2. OpenMP编程制导
  - 3. OpenMP库函数
  - 4. OpenMP环境变量
  - 5. OpenMP并行注意的问题
-

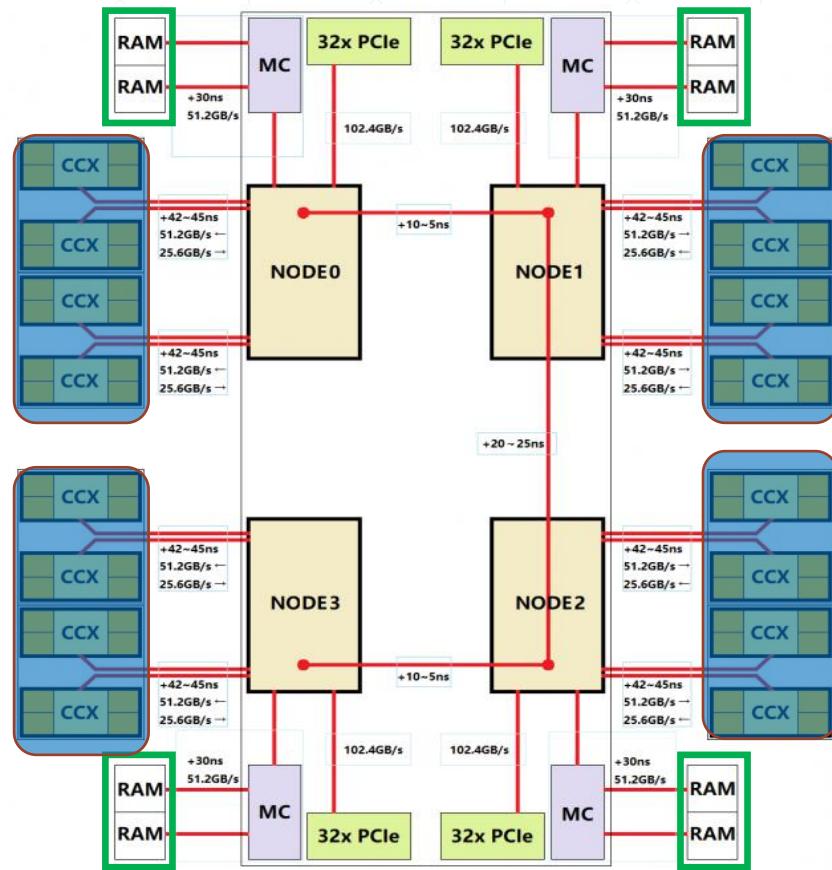
# 1.1 OpenMP简介

OpenMP (Open Multi-Processing) 是一套支持跨平台共享内存方式的多线程并发的编程API，支持C,C++和Fortran语言开发。适合于SMP共享内存多处理系统和多核处理器体系结构

- 与Pthreads和X3H5标准同为共享存储器编程标准
- 简单、移植性好和可扩展性等特点
- 提供了支持Fortran、C/C++的API和规范
- 由一组编译制导、运行时库函数和环境变量组成
- 工业标准
  - DEC、Intel、IBM、HP、Sun、SGI等公司支持
  - 包括Linux、Unix和NT等多种操作系统平台
- <http://www.openmp.org/>

# SMP共享内存系统

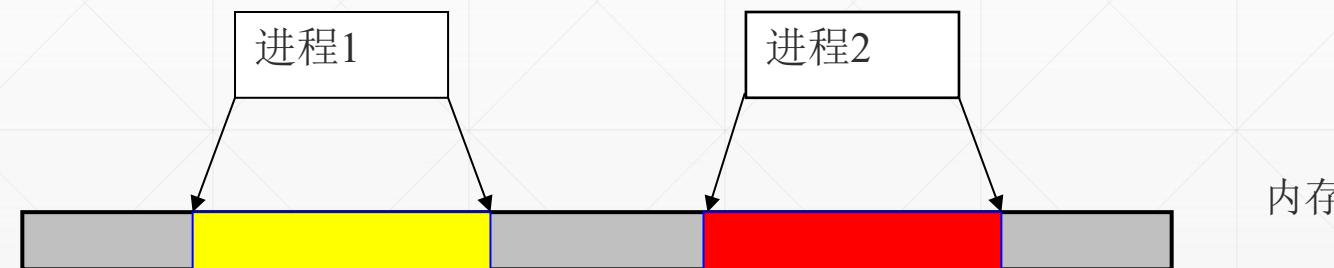
- 对称多处理 (Symmetrical Multi-Processing) 简称SMP，是指在一个计算机上汇集了一组处理器(多CPU),各CPU之间**共享内存子系统**以及总线结构。



注意互联架构图中，CCX对应的CPU所在部分，以及RAM内存和MC所在部分，通过内部IF总线相互连接从而实现了每个CPU核心都能访问到整个系统的内存数据

# 进程&线程

- 单个进程 (process)
  - 进程与程序相联，程序一旦在操作系统中运行即成为进程。进程拥有独立的执行环境（内存、寄存器、程序计数器等），是操作系统中独立存在的可执行的基本程序单位
  - 串行应用程序编译形成的可执行代码，分为“指令”和“数据”两个部分，并在程序执行时“独立地申请和占有”内存空间，且所有计算均局限于该内存空间。



# 进程与消息传递

- 单机内多个进程
  - 多个进程可同时存在于单机内同一操作系统。操作系统负责调度分时共享处理机资源(CPU、内存、存储、外设等)
  - 进程间相互独立(内存空间不相交)。在操作系统调度下各自独立地运行，例如多个串行应用程序在同一台计算机运行
  - 进程间可以相互交换信息。例如数据交换、同步等待，消息是这些交换信息的基本单位，消息传递是指这些信息在进程间的相互交换，是实现进程间通信的唯一方式

# 进程&线程

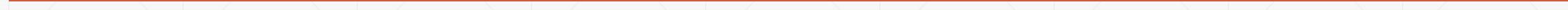
- 线程是什么？

线程是CPU调度和分派的基本单位，它可以和同一进程下的其他线程共享全部资源

- 联系：

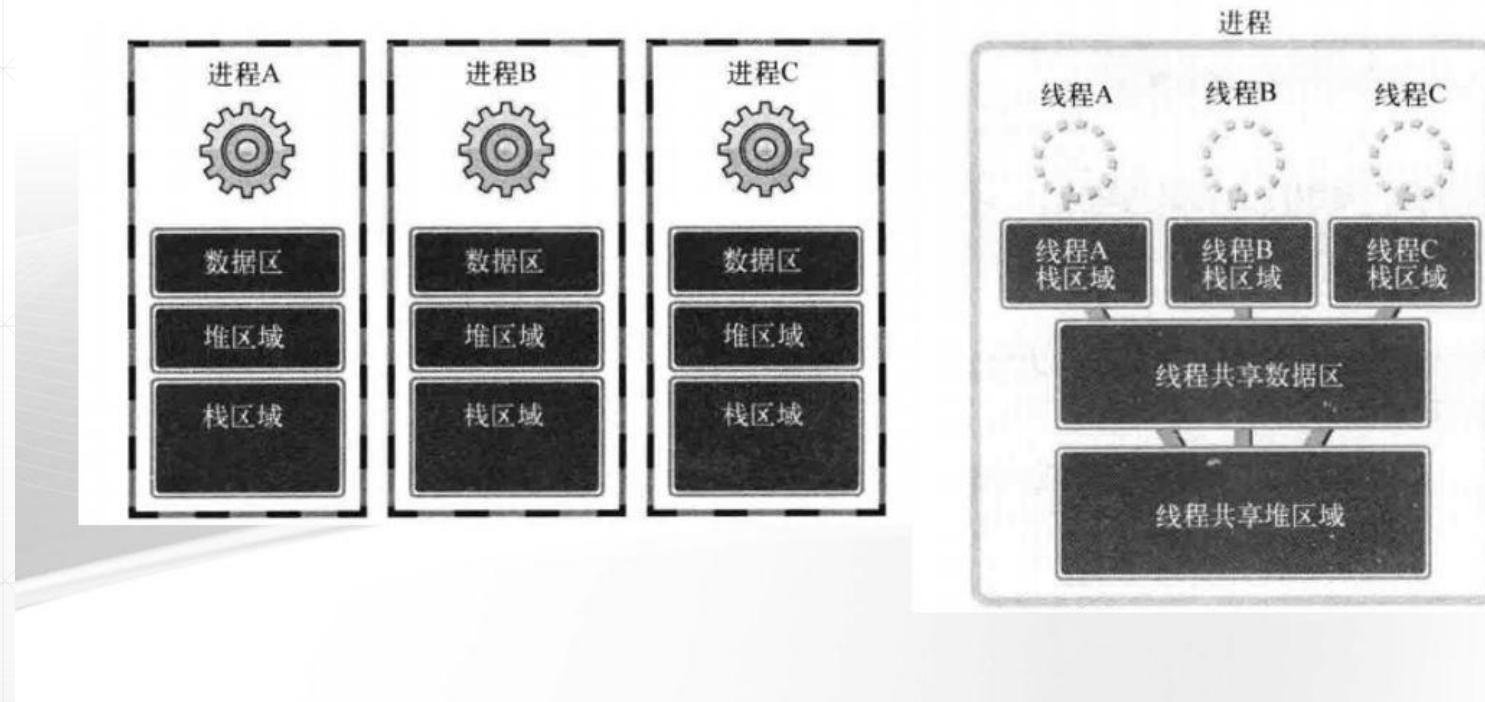
线程是进程中的一部分，一个进程可以有多个线程，但线程只能存在于一个进程中。

进程拥有各自独立的地址空间、资源，所以共享复杂，需要用IPC（Inter-Process Communication，进程间通信），而线程不但拥有私有的资源空间，也共享来自父进程的数据资源，因此共享简单，但是同步复杂，需要用加锁等措施。



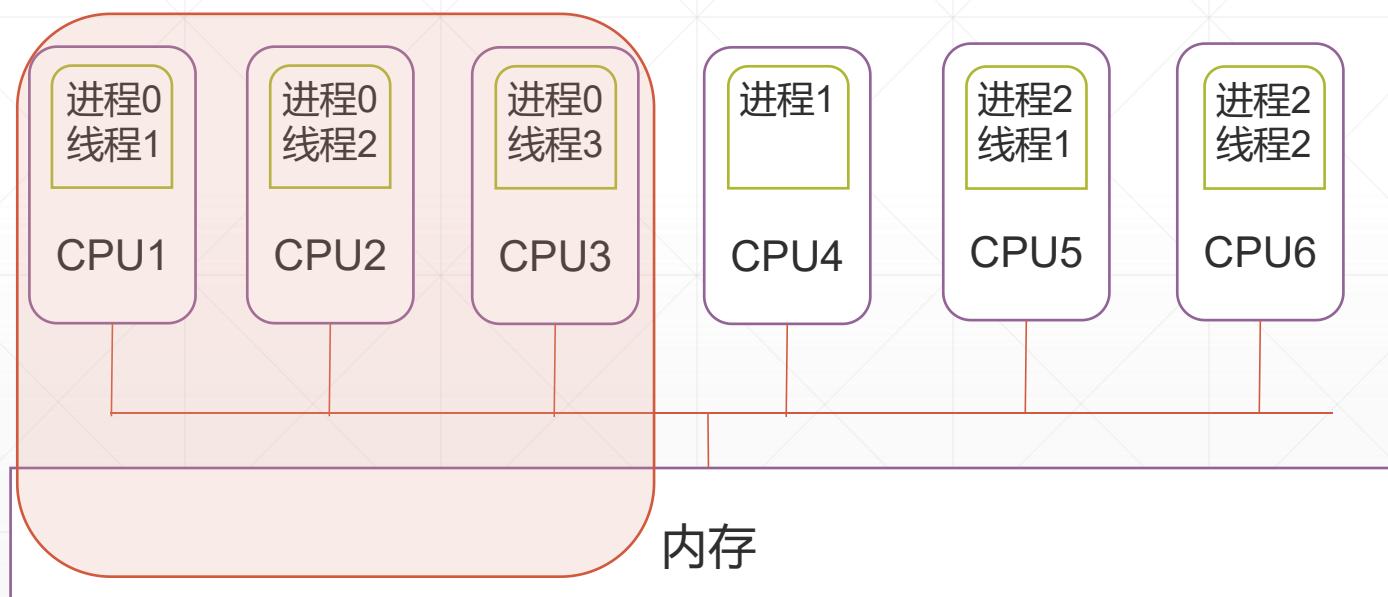
# 进程&线程

## 进程空间和线程空间



# 多线程

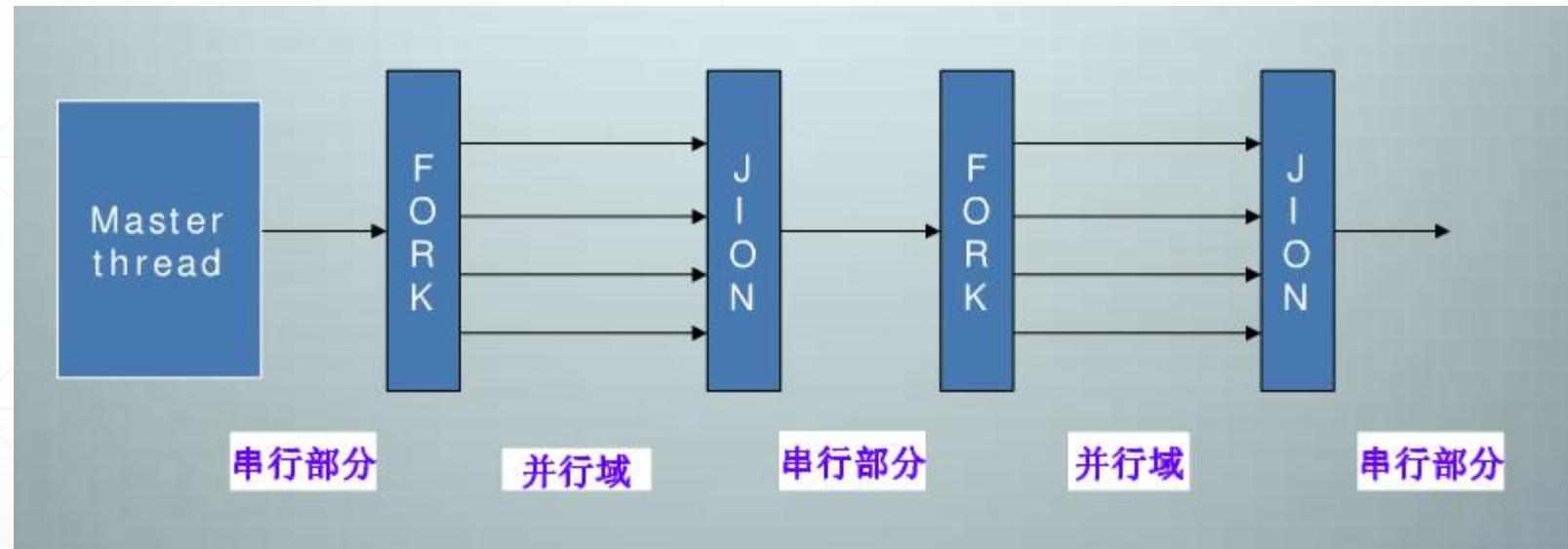
- 同一个进程下的多个线程可以同时在一个或多个CPU核心上并行执行，而这种程序也就被称作多线程程序。



## 1.2 OpenMP并行编程模式

- OpenMP是基于线程的并行编程模型
- 相比pthread这种线程标准库，OpenMP隐藏了底层细节，交给编译器和运行时系统决定线程的一些行为
- OpenMP采用Fork-Join并行执行方式
- OpenMP程序开始于一个单独的主线程（Master Thread），然后主线程一直串行执行，直到遇见第一个并行域（Parallel Region），然后开始并行执行并行域。其过程如下：
  - Fork：主线程创建一个并行线程队列，然后，并行域中的代码在不同的线程上并行执行；
  - Join：当并行域执行完之后，它们或被同步或被中断，最后只有主线程在执行

## 1.3 OpenMP程序并行框架



## 1.4 Sample - omp\_1

```
/*用OpenMP/C编写Hello World代码段*/
#include<omp.h>
#include<stdio.h>
int main(int argc,char *argv[])
{
    int tid;
    /*Fork a team of threads*/
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num(); //获取线程号
        printf("Hello World from OpenMP thread %d\n",tid);
    }
}
```

## 1.4 Sample - omp\_1

```
/*用OpenMP/C编写Hello World代码段*/
#include<omp.h>
#include<stdio.h>
int main(int argc,char *argv[])
{
    int tid;
    /*Fork a team of threads*/
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num(); //获取线程号
        printf("Hello World from OpenMP thread %d\n",tid);
    }
}
```

## 1.4 Sample - omp\_1

编译:

```
gcc -fopenmp -o helloworld helloworld.c
```

或 icc -qopenmp -o helloworld helloworld.c

运行:

```
export OMP_NUM_THREADS=4
```

./helloworld

输出:

Hello World from OpenMP thread 2

Hello World from OpenMP thread 0

Hello World from OpenMP thread 1

Hello World from OpenMP thread 3

# OpenMP编程

- 1. OpenMP编程简介
  - 2. OpenMP编程制导
  - 3. OpenMP库函数
  - 4. OpenMP环境变量
-

## 2.1 OpenMP编译制导介绍

```
#include<omp.h>
#include<stdio.h>
int main(int argc,char *argv[])
{
    int tid;
    /*Fork a team of threads*/
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num(); //获取线程号
        printf("Hello World from OpenMP thread %d\n",tid);
    }
}
```

## 2.1 OpenMP编译制导介绍

- OpenMP的并行化是通过使用嵌入到C/C++或Fortran源代码中的编译制导语句实现的。
- 编译制导是对程序设计语言的扩展。
- 支持并行区域、工作共享、同步等。
- 支持数据的共享和私有化
- 通过对串行程序添加制导语句实现并行化



## 2.2 OpenMP制导语句格式

- 编译制导语句由下列几部分组成
  - 制导标识符 (!\$OMP、#pragma omp)
  - 制导名称 (parallel、DO/for、section等)
  - 子句 (privated、shared、reduction等)
- 格式：制导标识符 制导名称【Cluse】
  - 制导标识符 (!\$OMP、#pragma omp)
  - 制导名称 (parallel、DO/for、section等)
  - 子句 (privated、shared、reduction等)

## 2.3 OpenMP编译制导标识

- 制导是特殊的、仅用于特定编译器的源代码
- 制导由一个位于行首的标识加以区分
- OpenMP制导标识：
  - Fortran:  
!\$OMP
  - c/c++  
#pragma omp

## 2.4 并行域制导

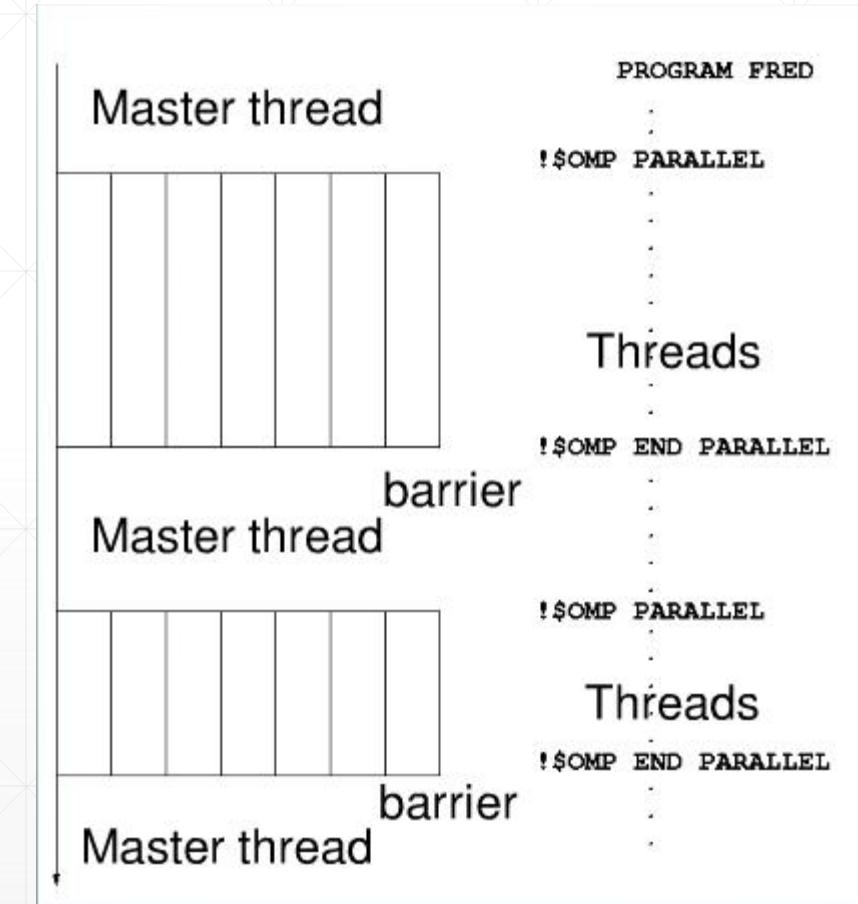
一个并行域就是一个能被多个线程并行执行的程序段

C/C++:

```
#pragma omp parallel [clauses]
{
    block
}
```

## 2.4 并行域制导

- 在并行域结尾有一个隐式同步 (barrier) .
- 子句(clause)用来说明并行域的附加信息。
- 在Fortran语言中，子句间用逗号或空格分隔；  
C/C++子句间用空格分开



## 2.6 并行域制导

shared 和private子句

- 并行域内的变量，可以通过子句说明为公有和私有
- 在编写多线程程序时，确定哪些数据的公有或私有非常重要：影响程序的性能和正确性
- Fortran：

SHARED(list)

PRIVATE(list)

DEFAULT(SHARED|PRIVATE|NONE)

- C/C++:

shared(list)

private(list)

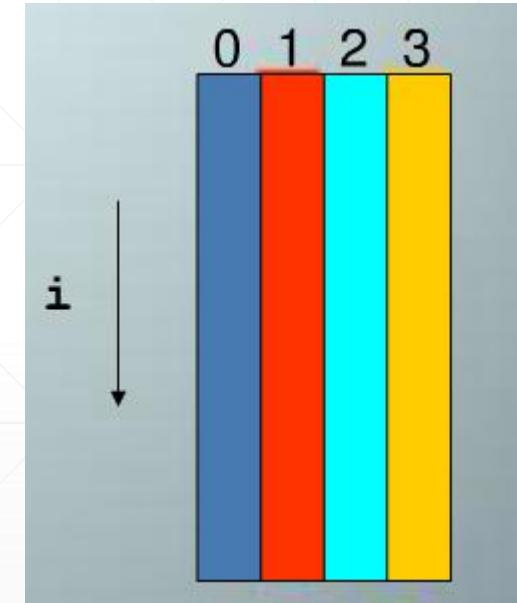
default(shared|private|none)



## 2.6 并行域制导

例：每个线程初始共享数组的一列

```
#pragma omp parallel default(None) private(i,myid) shared(a,n)
{
    myid=omp_get_thread_num()+1
    for(int i=0; i<n; i++) {
        a[i][myid]=i
    }
}
```



说明：如何决定哪些变量是共享哪些是私有？

- 通常循环变量、临时变量、写变量一般是私有的
- 数组变量、仅用于读的变量通常是共享的，默认为公有

## 2.6 并行域制导

- 任务划分并行制导
- 制导可以出现在并行域内部，并表明任务如何在多个线程间分配，OpenMP任务划分制导包括：
  - 并行do/for循环制导
  - 并行SECTIONS制导
  - SINGLE和MASTER制导
  - 其他制导



## 2.6 并行域制导

### ■ 并行DO/for循环制导

并行DO/for循环制导用来将循环划分成多个块，并分配给各线程并行执行。

Fortran:

```
!$OMP DO[clause]  
    DO 循环  
!$OMP END DO
```

C/C++:

```
#pragma omp for[clauses]  
    for循环
```

说明：

- 并行DO/for循环有时需要PRIVATE和FIRSTPRIVATE子句
- 循环变量是私有的

## 2.6 并行域制导

- 可以将并行域和DO/for制导结合成单一的简单形式

```
#pragma omp parallel [clauses]
{
    #pragma omp for [clauses]
    {
        循环体
    }
}
```

合并后形式：

```
#pragma omp parallel for [clauses]
{
    循环体
}
```

## 2.6 并行域制导

- 并行DO/for循环制导：调度子句 SCHEDULE

该子句给出迭代循环划分后的块大小和线程执行的块范围

Fortran:

```
SCHEDULE(kind[,chunksize])
```

C/C++:

```
schedule(kind[,chunksize])
```

其中： kind为STATIC,DYNAMIC或RUNTIME

chunksize是一个整数表达式

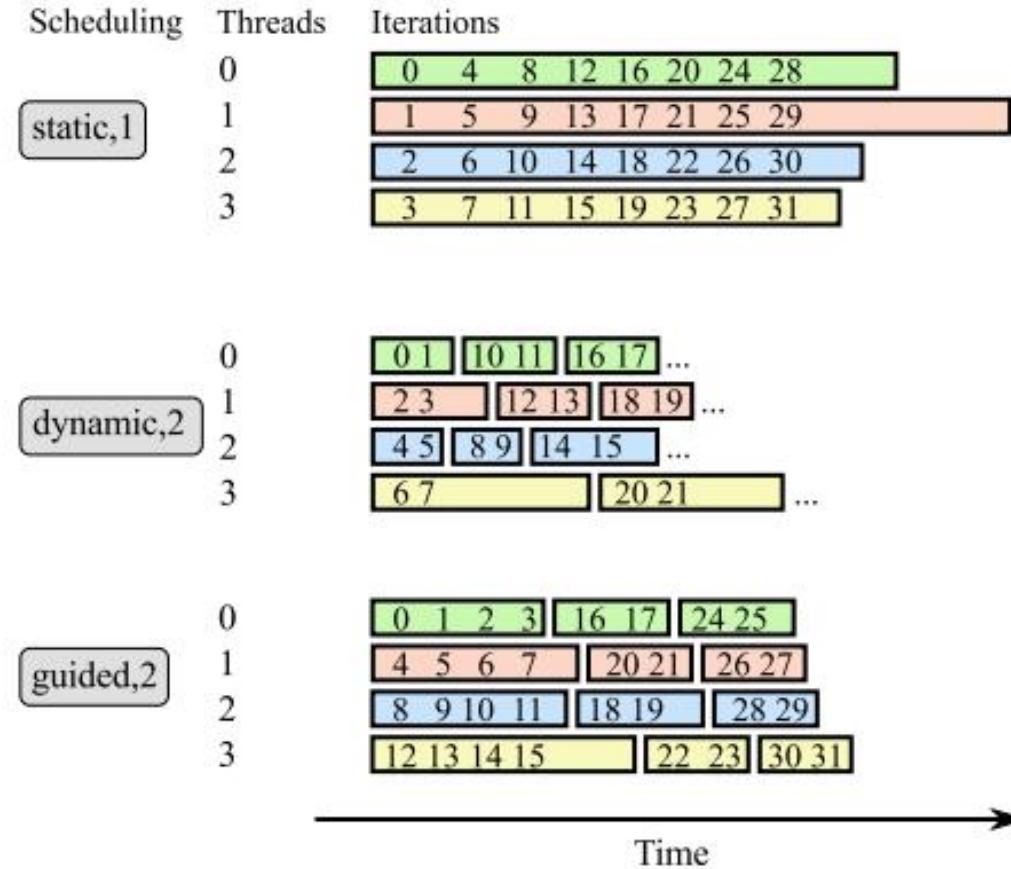
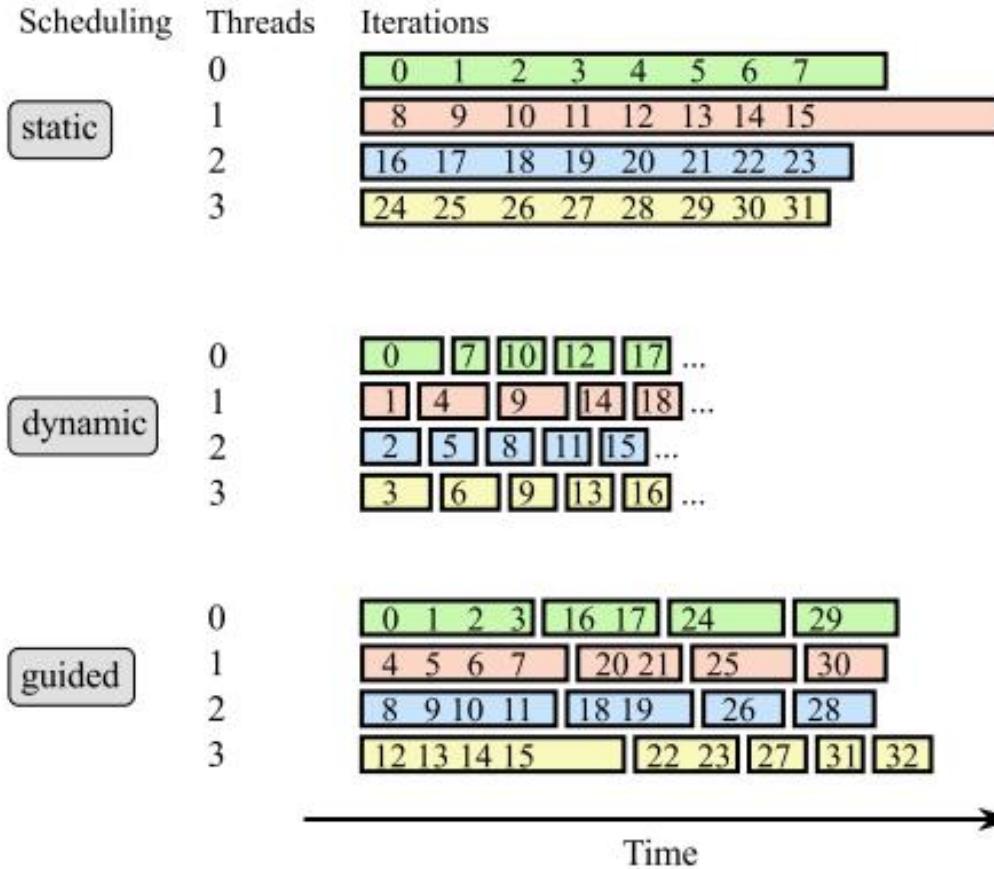
例如：

```
!$OMP DO SCHEDULE(DYNAMIC,4)
```

    循环体

```
!$OMP END DO
```

## 2.6 并行域制导



## 2.7 数据竞争问题

- 下面的循环无法正确执行
- ```
#pragma omp parallel for
for(k=0;k<100;k++)
{ x=array[k];
  array[k]=do_work(x);
}
```

- 正确的方式
- 直接声明为私有变量

```
#pragma omp parallel for private(x)
for(k=0;k<100;k++)
{ x=array[k];
  array[k]=do_work(x);
}
```

- 在parallel 结构中声明变量，这样的变量是私有的

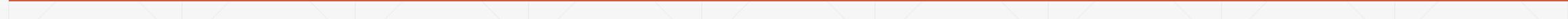
```
#pragma omp parallel for
for(k=0;k<100;k++)
{
  int x;
  x=array[k];
  array[k]=do_work(x);
}
```

## 2.8 SECTIONS制导：任务分配区

- 任务分配区(work-sharing sections)可以使OpenMP编译器和运行时库将应用程序中标出的结构化块 (block) 分配到并行区域的一组线程上

- C/C++:

```
$pragma sections[clauses]
{ $pragma section
    block
$pragma section
    block
}
```



```
#include<omp.h>
#include<stdio.h>
int main(int argc,char *argv[])
{ int tid;
#pragma omp parallel sections private(tid) {
#pragma omp section{
    tid = omp_get_thread_num();/*获取线程号*/
    printf("here is section 1 ,num %d is here!\n",tid);
}
#pragma omp section{
    tid = omp_get_thread_num();/*获取线程号*/
    printf("here is section 2 ,num %d is here!\n",tid);
}
#pragma omp section{
    tid = omp_get_thread_num();/*获取线程号*/
    printf("here is section 3 ,num %d is here!\n",tid);
}
}
return 0;
}
```

```
[root@localhost openmp]# ./sections
here is section 1 ,num 0 is here!
here is section 2 ,num 1 is here!
here is section 3 ,num 2 is here!
```

## 2.9 SINGLE制导

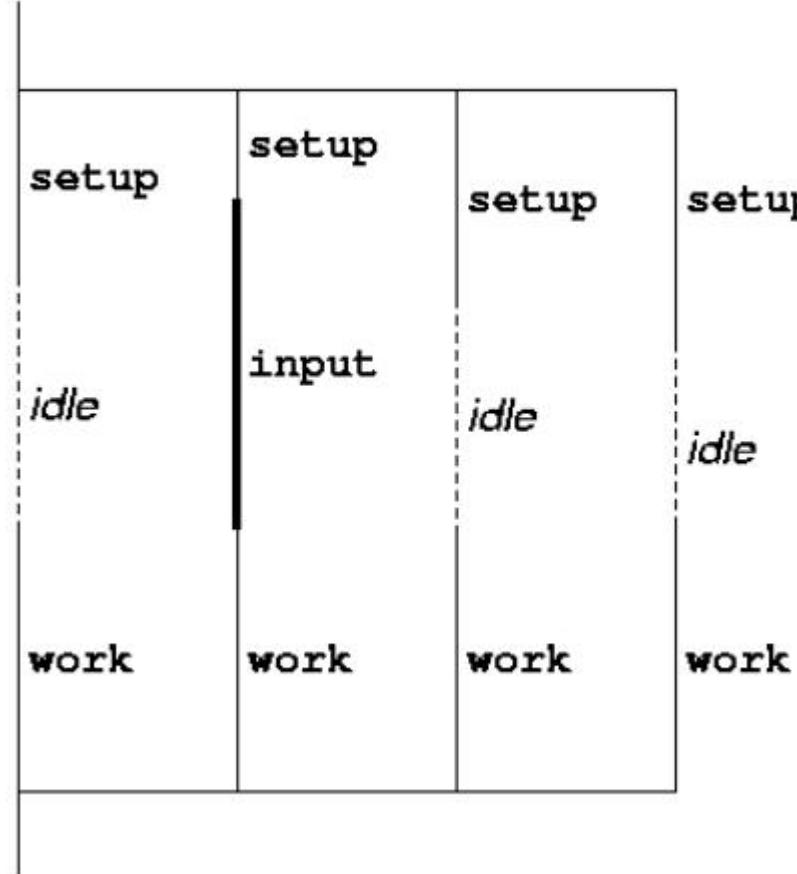
- SINGLE制导:

C/C++:

```
#pragma omp single [clauses]
{
    structure block
}
```

说明:

结构体代码仅由一个线程执行;  
并由首先执行到该代码的线程执行;  
其它线程等待直至该结构块被执行完



例子

```
#pragma omp parallel
{
    setup(x);
    #pragma omp single
    {
        input(y);
        work(x,y);
    }
}
```

```
#include<omp.h>
#include<stdio.h>
int main(int argc,char *argv[])
{
    int tid;
#pragma omp parallel private(tid)
    {
#pragma omp single{
        tid = omp_get_thread_num();/*获取线程号*/
        printf("here is single 1 ,num %d is here!\n",tid);
    }
#pragma omp single{
        tid = omp_get_thread_num();/*获取线程号*/
        printf("here is single 2 ,num %d is here!\n",tid);
    }
#pragma omp single{
        tid = omp_get_thread_num();/*获取线程号*/
        printf("here is single 3 ,num %d is here!\n",tid);
    }
}
return 0;
}
```

```
[root@localhost openmp]# ./single
here is single 1 ,num 7 is here!
here is single 2 ,num 0 is here!
here is single 3 ,num 3 is here!
```

## 2.10 MASTER制导

- MASTER制导

C/C++:

```
#pragma omp master [clauses]  
structure block
```

说明：

- 结构体代码仅由主线程执行；
- 其它线程跳过并继续执行；
- 通常用于I/O；



```
#include<omp.h>
#include<stdio.h>
int main(int argc,char *argv[])
{
    int tid;
    #pragma omp parallel private(tid)
    {
        #pragma omp master
        {
            tid = omp_get_thread_num();/*获取线程号*/
            printf("here is master relion, num %d is here!\n",tid);
        }
        tid = omp_get_thread_num();/*获取线程号*/
        printf("Hello World from OpenMP thread %d\n",tid);
    }
}
```

```
[root@localhost openmp]# ./master
Hello World from OpenMP thread 4
Hello World from OpenMP thread 7
Hello World from OpenMP thread 2
Hello World from OpenMP thread 1
Hello World from OpenMP thread 6
Hello World from OpenMP thread 3
Hello World from OpenMP thread 5
here is master relion, num 0 is here!
Hello World from OpenMP thread 0
```

## 2.11 BARRIER制导

- BARRIER是OpenMP用于线程同步的一种方法

C/C++:

```
#pragma omp barrier
```

说明:

- 在所有的线程到达之前，没有线程可以提前通过一个barrier；
- 在DO/FOR、SECTIONS和SINGLE制导后，有一个隐式barrier 存在；
- 要么所有线程遇到barrier；要么没有线程遇到barrier，否则会出现死锁。



```
#include<omp.h>
#include<stdio.h>
int main(int argc,char *argv[])
{
    int tid;
#pragma omp parallel private(tid)
    {
#pragma omp master
    {
        tid = omp_get_thread_num();/*获取线程号*/
        printf("here is master relion, num %d is here!\n",tid);
    }
#pragma omp barrier
        tid = omp_get_thread_num();/*获取线程号*/
        printf("Hello World from OpenMP thread %d\n",tid);
    }
}
```

```
[root@localhost openmp]# ./barrier
here is master relion, num 0 is here!
Hello World from OpenMP thread 0
Hello World from OpenMP thread 4
Hello World from OpenMP thread 2
Hello World from OpenMP thread 5
Hello World from OpenMP thread 1
Hello World from OpenMP thread 6
Hello World from OpenMP thread 3
Hello World from OpenMP thread 7
```

## 2.12 reduction子句

- Reduction子句使用指定的操作对其列表中出现的变量进行规约
- 初始时，每个线程都保留一份私有拷贝
- 在结构尾部根据指定的操作对线程中的相应变量进行规约，并更新该变量的全局值

```
#include <omp.h>
#include <stdio>

int main() {
    const int n = 1000;
    int total = 0;
#pragma omp parallel for reduction(+: total)
    for (int i = 0; i < n; i++) {
        total = total + i;
    }
    printf("total=%d (must be %d)\n", total, ((n-1)*n)/2);
}
```

## 2.12 reduction子句

- reduction中的op操作必须满足算术结合律和交换律
- reduction子句允许的操作符: + , \* , - , & , | , ^ , && , || , max或min等

# Sample - omp\_2

- 计算Pi值

```
/* Seriel Code */
#include<stdio.h>
#include<time.h>
static long num_steps =100000;
double step;
void main()
{
    int i;
    double x,pi,sum=0.0,start_time,end_time;
    step = 1.0/(double)num_steps;
    start_time=clock();
    for(i=1;i<=num_steps;i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
    end_time=clock();
    printf("Pi=%f Running time=%f\n",pi,end_time-start_time);
}
```

$$4 \int_0^1 \left( \frac{1}{1+x^2} \right) = 4 \arctan x \Big|_0^1 = 4 * \frac{\pi}{4}$$

$$4 \int_0^1 \left( \frac{1}{1+x^2} \right) = \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{0 < i \leq n} \frac{4}{1 + \left( \frac{i}{n} \right)^2}$$

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000;
double step;
void main ()
{
    int i,id=0;double x, pi, sum, start_time, end_time;
    step = 1.0/(double) num_steps;
    start_time=omp_get_wtime();
    #pragma omp parallel private(i,x,id)
    {
        id = omp_get_thread_num();
        sum = 0.0;
    #pragma omp for private(x,i) reduction(+:sum)
        for (i=0;i< num_steps; i++)
        {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
    }
    end_time=omp_get_wtime();
    pi=sum/num_steps;
    printf("Pi=%lf\n Running time %lf\n",pi,end_time-start_time);
}
```

## Sample - omp\_2ps

Pi=3.141593 Running time=0.343016

Using 4 threads:

Pi=3.141593 Running time 0.086781

Using 64 threads:

Pi=3.141593 Running time 0.006556

## 2.13 ATOMIC制导

- ATOMIC编译制导表明一个特殊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
- 主要用来保证操作被安全的执行。

Fortran:

```
!$OMP ATOMIC
statement
```

C/C++:

```
#pragma omp atomic
statement
```

说明

- 在fortran中，statement必须是下列形式之一：

$x=x \text{ op } expr$ 、 $x=expr \text{ op } x$ 、 $x=\text{intr}(x, expr)$ 或 $x=\text{intr}(expr, x)$ 。

其中：op是+、-、\*、/、.and.、.or.、.eqv.、或.neqv.之一；intr是MAX、min、IAND、IOR或IEOR之一

- 在C/C++中，statement必须是下列形式之一：

$x \text{ binop}=expr$ 、 $x++$ 、 $x--$ 、 $++x$ 、或 $--xx$ 。

其中：binop是二元操作符：+、-、\*、/、&、^、<< 或>>之一。

- ATOMIC编译指导的好处是允许并行的更新数组内的不同元素；而使用临界值时数组元素的更新是串行的；
- 无论何时，当需要在更新共享存储单元的语句中避免数据竞争，应该先使用atomic，然后再使用临界段。

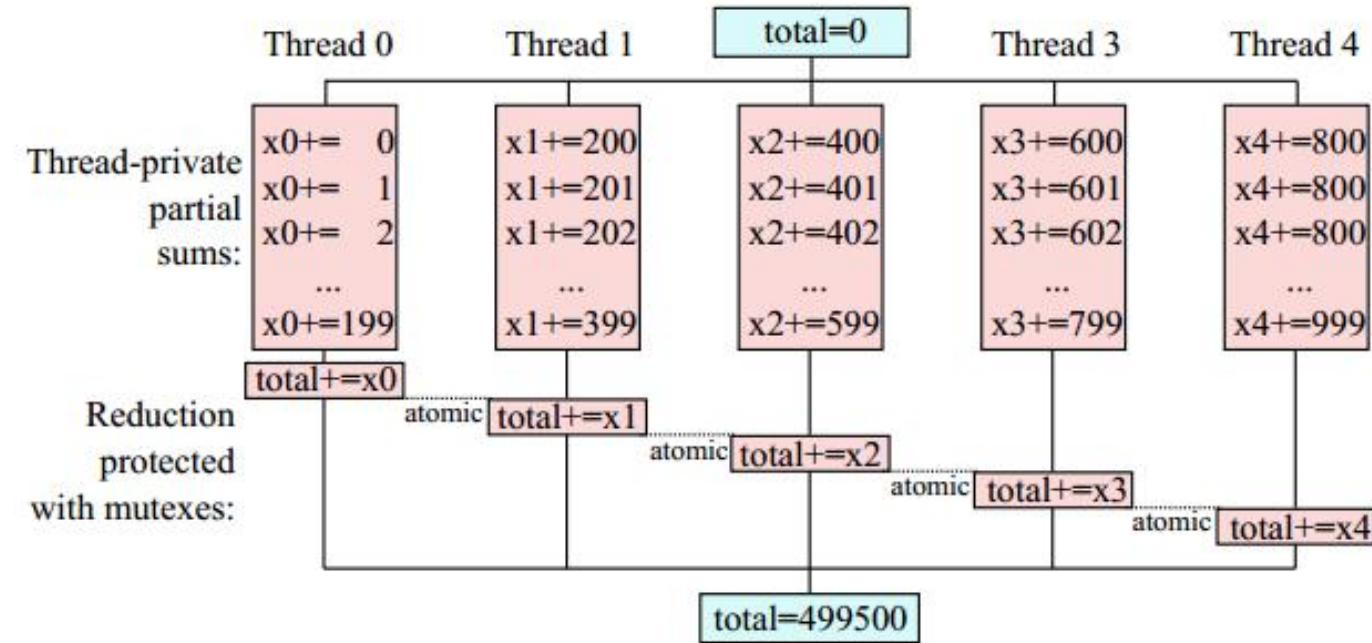
## 2.13 ATOMIC制导

```

int total = 0;
#pragma omp parallel
{
    int total_thr = 0;
#pragma omp for
    for (int i=0; i<n; i++)
        total_thr += i;

#pragma omp atomic
    total += total_thr;
}

```



## 2.14 SIMD

对循环进行向量化操作：

This approach often works:

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++) // Thread parallelism in outer loop
3 #pragma simd
4 for (int j = 0; j < m; j++) // Vectorization in inner loop
5     DoSomeWork(A[i][j]);
```

That works as well:

```
1 #pragma omp parallel for simd
2 for (int i = 0; i < n; i++) // If the problem is all data-parallel
3     DoSomeWork(A[i]);
```

# OpenMP编程

- 1. OpenMP编程简介
  - 2. OpenMP编程制导
  - 3. OpenMP库函数
  - 4. OpenMP环境变量
-

### 3. OpenMP库函数

- OpenMP标准定义了一个应用程序编程接口来调用库中的多个函数。有时需要得到线程数和线程号，这在控制不同线程执行不同的功能代码时特别有用。

- 得到线程队列中的线程数

Fortran:

```
integer function OMP_GET_NUM_THREADS ()
```

C/C++:

```
#include<omp.h>
```

```
int omp_get_num_threads()
```

- 得到执行线程的线程号：

Fortran:

```
integer function OMP_GET_THREAD_NUM ()
```

C/C++:

```
#include<omp.h>
```

```
int omp_get_thread_num()
```

---

### 3. OpenMP库函数

- 设定执行线程的数量
- 在制导语句中通过OMP\_NUM\_THREADS设定。

Fortran:

```
routine OMP_SET_NUM_THREADS( )
```

C/C++:

```
#include<omp.h>
omp_set_num_threads()
```

- 通过环境变量OMP\_NUM\_THREADS 设定。

```
export OMP_NUM_THREADS = 4
```

---

### 3. OpenMP库函数

#### 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\_WTIME() : 获取每个时钟周期的秒数, 即omp\_get\_wtime的精度
- omp\_get\_wtime
  - Fortran: double precision function omp\_get\_wtime()
  - C/C++ : double omp\_get\_wtime(void);
- omp\_get\_wtick
  - Fortran: double precision function omp\_get\_wtick()
  - C/C++ : double omp\_get\_wtick();

### 3. OpenMP库函数

#### 时间函数

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

Fortran

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

C/C++

# OpenMP编程

- 1. OpenMP编程简介
  - 2. OpenMP编程制导
  - 3. OpenMP库函数
  - 4. OpenMP环境变量
  - 5. OpenMP计算实例
-

## 4. OpenMP环境变量

- OpenMP提供了4个环境变量用来控制并行代码的执行设定线程数环境变量：

例如：

1. OMP\_NUM\_THREADS: 设定最大线程数。

export OMP\_NUM\_THREADS=42.

2. OMP\_SCHEDULE: 设定DO/for循环调度方式环境变量。

export OMP\_SCHEDULE = "DYNAMIC, 4"

3. OMP\_DYNAMIC: 确定是否动态设定并行域执行的线程数，其值为FALSE或TRUE。

export OMP\_DYNAMIC = TRUE4.

4. OMP\_NESTED:确定是否可以并行嵌套。

export OMP\_NESTED = TURE

## 4. OpenMP环境变量

- NUM\_THREADS子句

- 在OpenMP 2.0 (Fortran、C/C++) 中提供了NUM\_THREADS 子句设定线程数。

```
#pragma omp parallel for num_threads(4)  
for (i=0;i<n;i++)  
    a[i]=b[i]*c[i];
```

说明:

- 在NUM\_THREADS中提供的值将取代环境变量OMP\_NUM\_THREADS 的值或由omp\_set\_num\_threads()设定的值



# OpenMP编程

- 1. OpenMP编程简介
  - 2. OpenMP编程制导
  - 3. OpenMP库函数
  - 4. OpenMP环境变量
  - 5. OpenMP并行注意的问题
-

## 5. OpenMP总结

- OpenMP并行注意的问题
  - 数据竞争问题；
  - 线程间同步；
  - 并行执行的程序比例及其可扩展性；
  - 共享内存或伪共享内存引起的访存冲突；
  - 在for循环中插入OpenMP指导前，首先要解决的问题是检查并重构热点循环，确保没有循环迭代相关；(tricks:颠倒循环顺序可以方便检查是否有数据依赖性)
  - 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；
  - 可以借助一些工具，例 Vtune™性能分析工具，其提供了一个线程监测器。

# 线程间同步

- 使用barrier 显示同步执行位置状态（相对应的nowait能消除隐式同步）
- 使用ordered 定序区段，线程逐个顺序执行
- 使用critical 临界区段或 atomic 临界语句，保证同一个时刻只有一个线程执行
- 使用openmp互斥锁相关函数 omp\_init\_lock/omp\_set\_lock/omp\_unset\_lock等
- 使用C++ mutex互斥量



## 5. OpenMP总结

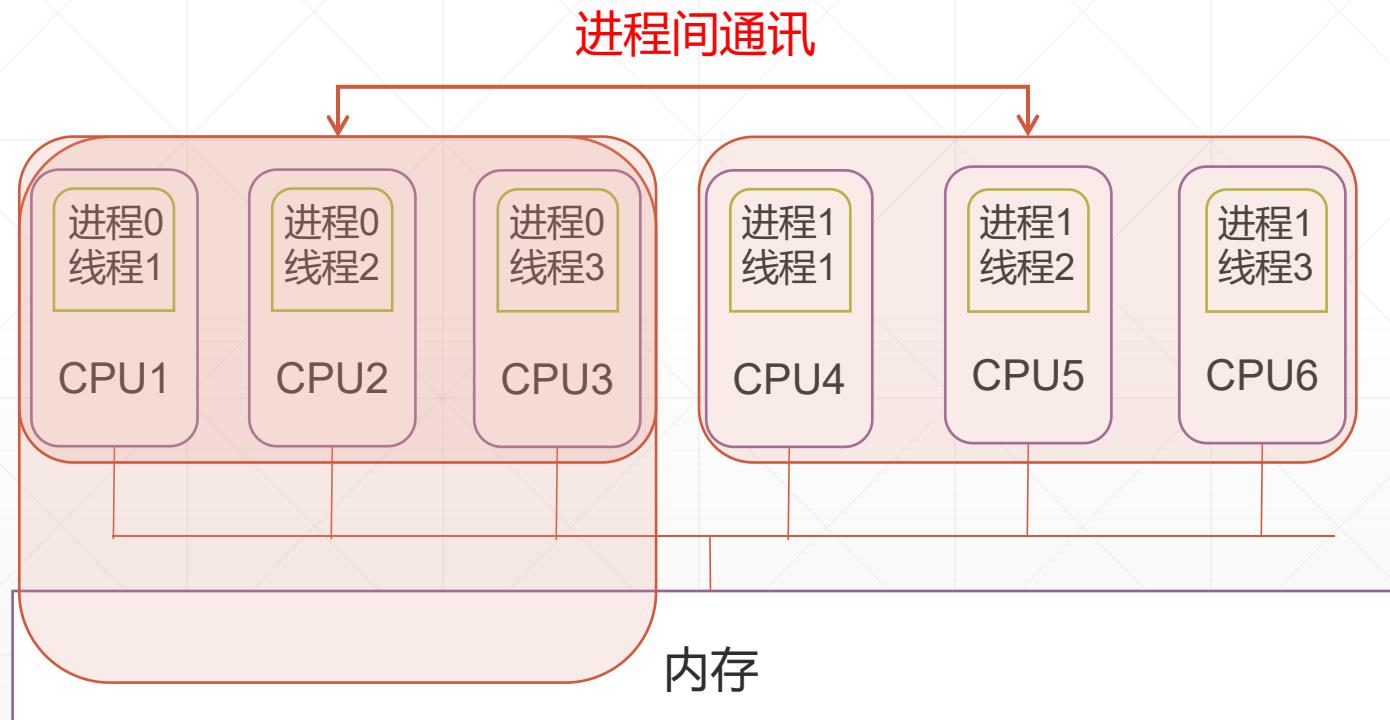
- 答疑



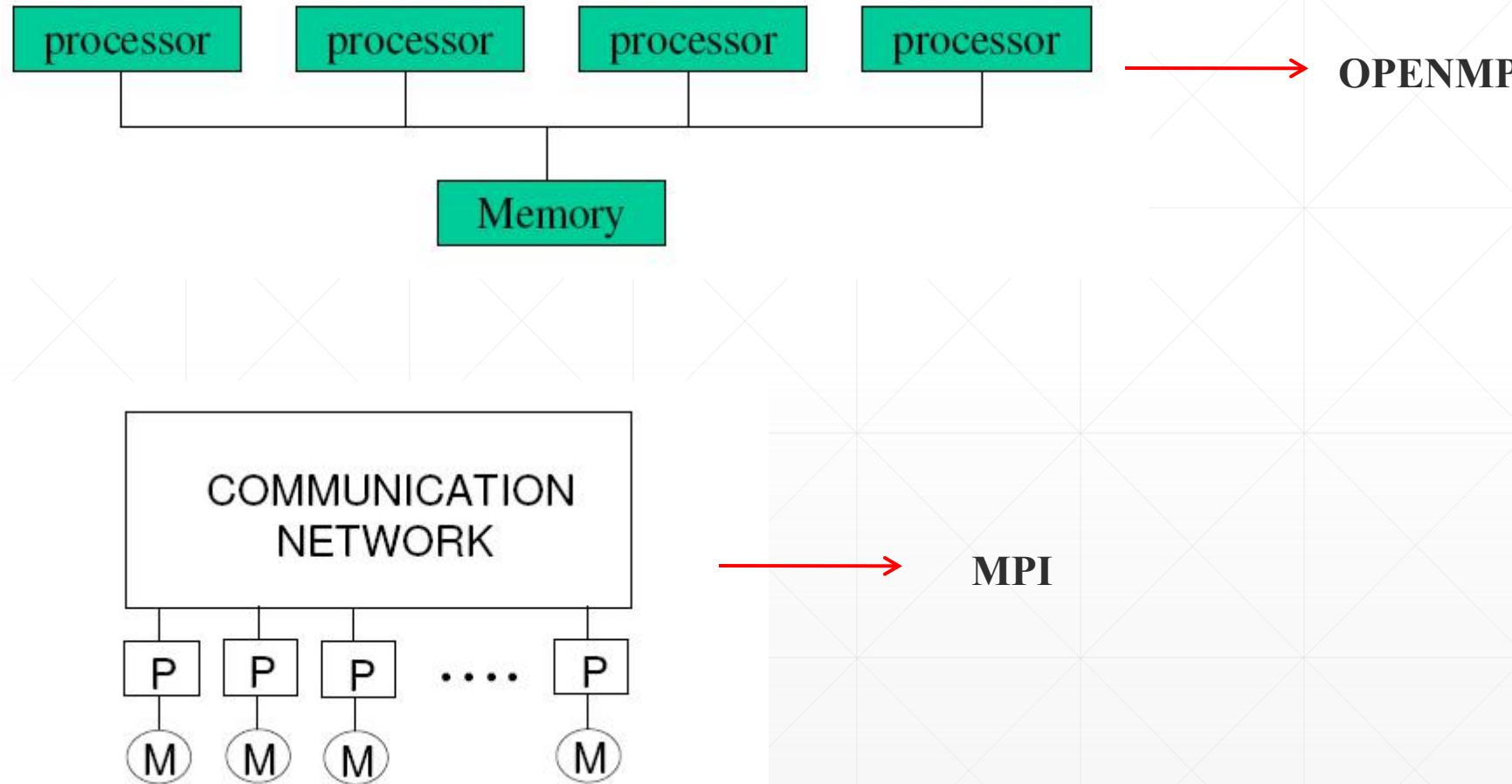
# MPI编程

---

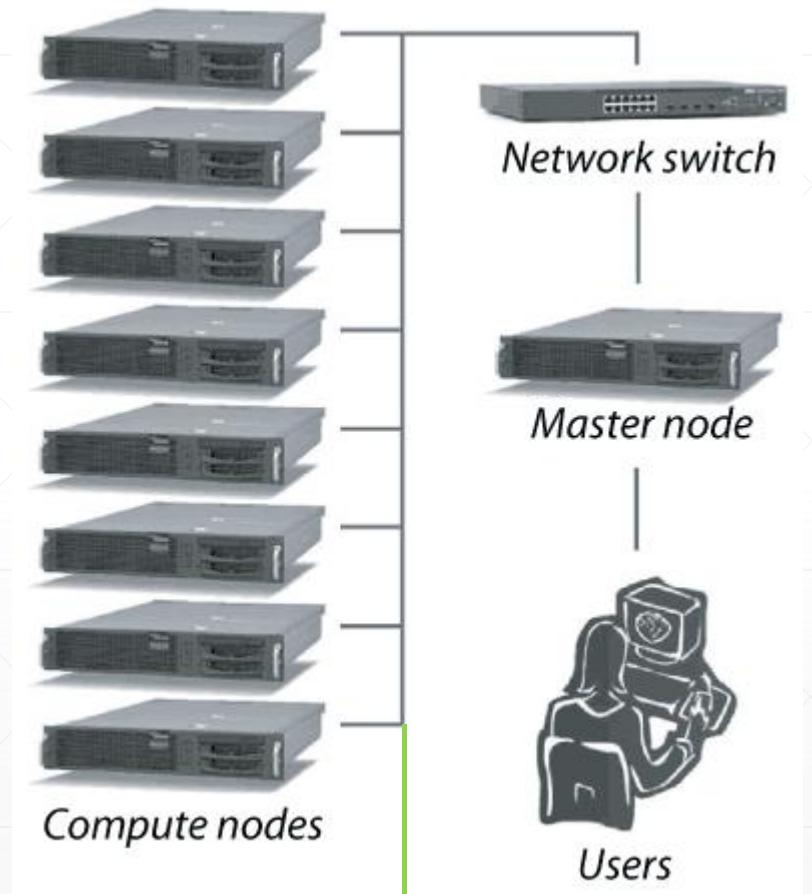
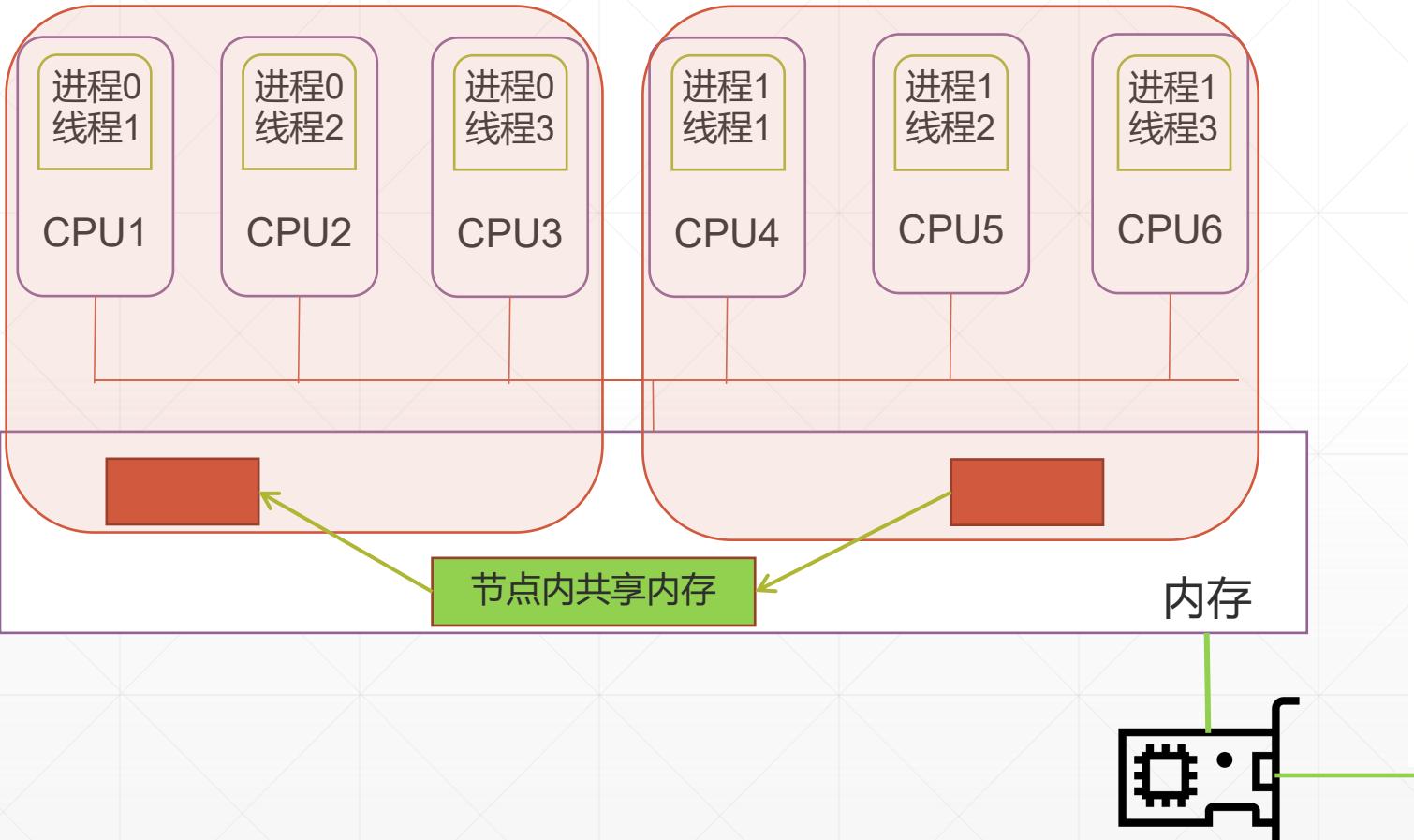
# 回顾SMP共享内存系统内的并行



# 并行计算



# MPI



# 消息传递平台MPI

- 什么是MPI (Message Passing Interface)
  - 是函数库规范，而不是并行语言；操作如同库函数调用
  - 是一种标准和规范，而非某个对它的具体实现（MPICH等），与编程语言无关
  - 是一种消息传递编程模型，并成为这类编程模型的代表
- What is the message?

DATA+ENVELOPE

- MPI的目标
  - 较高的通信性能
  - 较好的程序可移植性
  - 强大的功能

# 消息传递平台MPI

- MPI的产生
  - 1992-1994年，MPI 1.1版本问世
  - 1995-1997年，MPI 2.0版本出现
    - 扩充并行I/O
    - 远程存储访问
    - 动态进程管理等
- MPI的语言绑定
  - Fortran（科学与工程计算）
  - C（系统和应用程序开发）
- 主要的MPI实现
  - 并行机厂商提供 IntelMPI
  - 高校、科研部门

# mpi命令

- **mpirun** 运行常见参数：
  - -n node\_num 需要启动的节点数量
  - -np number\_of\_processes 参与执行的进程个数
  - -f hostsfile 节点列表，与-n一起使用表示启动节点列表中的nodenum个节点
  - 更多参数mpirun --help查看
  - mpirun -n 4 hostname # 4节点输出hostname
- **mpicc mpicxx** C/C++对应的MPI编译器
- **mpiexec**

# 消息传递平台MPI

- MPI程序编译与运行
  - 程序编译

C: %mpicc -o mpiprog mpisrc.c

Fortran 77: %mpif77 -o mpiprog mpisrc.f

- 程序运行

%mpirun -np 4 mpiprog

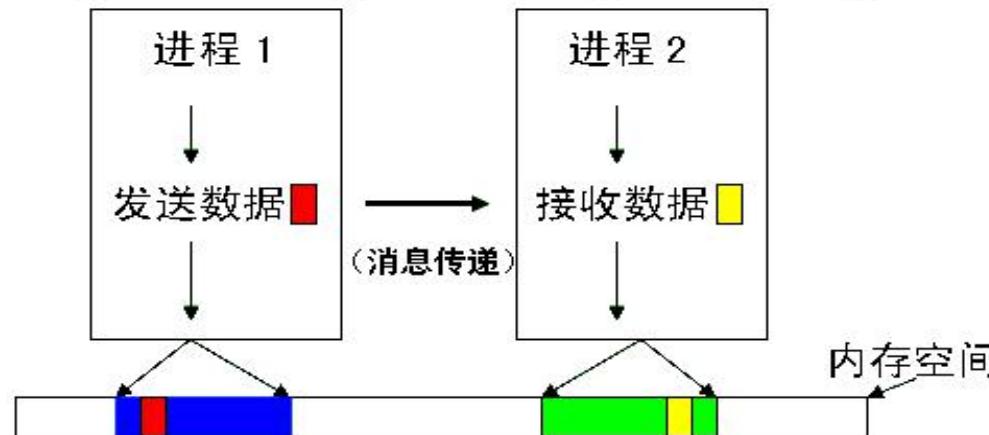
- 程序执行过程中**不能动态改变**进程的个数
- 申请的进程数np与实际处理器个数**无关**

# MPI基础知识

- 进程与消息传递
  - MPI重要概念
  - MPI函数一般形式
  - MPI原始数据类型
  - MPI程序基本结构
  - MPI几个基本函数
  - 并行编程模式
-

# 进程与消息传递

- 包含于通过网络联接的不同处理器的多个进程
  - 进程独立存在，并位于不同的处理器，由各自独立的操作系统调度，享有独立的CPU和内存资源
  - 进程间相互信息交换，可依靠**消息传递**
  - 传递的是数据和控制参数
  - 最基本的消息传递操作包括发送消息send、接受消息receive、进程同步barrier、归约reduction等



# MPI重要概念

- 进程组 (process group) 指MPI 程序的全部进程集合的一个有序子集且进程组中每个进程被赋予一个在该组中唯一的序号(rank), 用于在该组中标识该进程。序号的取值范围是[0,进程数- 1]
- 通信器 (communicator)
  - 理解为一类进程的集合即一个进程组, 且在该进程组, 进程间可以相互通信
  - 任何MPI通信函数均必须在某个通信器内发生
  - MPI系统提供省缺的通信器MPI\_COMM\_WORLD, 所有启动的MPI进程通过调用函数MPI\_Init()包含在该通信器内; 各进程通过函数MPI\_Comm\_size()获取通信器包含的(初始启动)的MPI进程个数
  - 组内通信器和组间通信器

- 进程序号 (rank) 用来在一个进程组或通信器中标识一个进程
  - MPI 程序中的进程由进程组或通信器序号唯一确定，序号相对于进程组或通信器而言（假设np个处理器，标号0...np-1）
  - 同一个进程在不同的进程组或通信器中可以有不同的序号，进程的序号是在进程组或通信器被创建时赋予的
  - MPI 系统提供了一个特殊的进程序号MPI\_PROC\_NULL，它代表空进程(不存在的进程)，与MPI\_PROC\_NULL 间的通信实际上没有任何作用
- 消息 (message)
  - 分为数据 (data) 和包装 (envelope) 两个部分
  - 包装由接收进程序号/发送进程序号、消息标号和通信器三部分组成；数据包含用户将要传递的内容

# MPI函数一般形式

C: error = MPI\_Xxxxx(parameter,...);

MPI\_Xxxxx(parameter,...);

- 整型错误码由函数值返回
- 除MPI\_Wtime() 和MPI\_Wtick()外, 所有MPI 的C 函数均返回一个整型错误码。成功时返回MPI\_SUCCESS, 其他错误代码依赖于执行



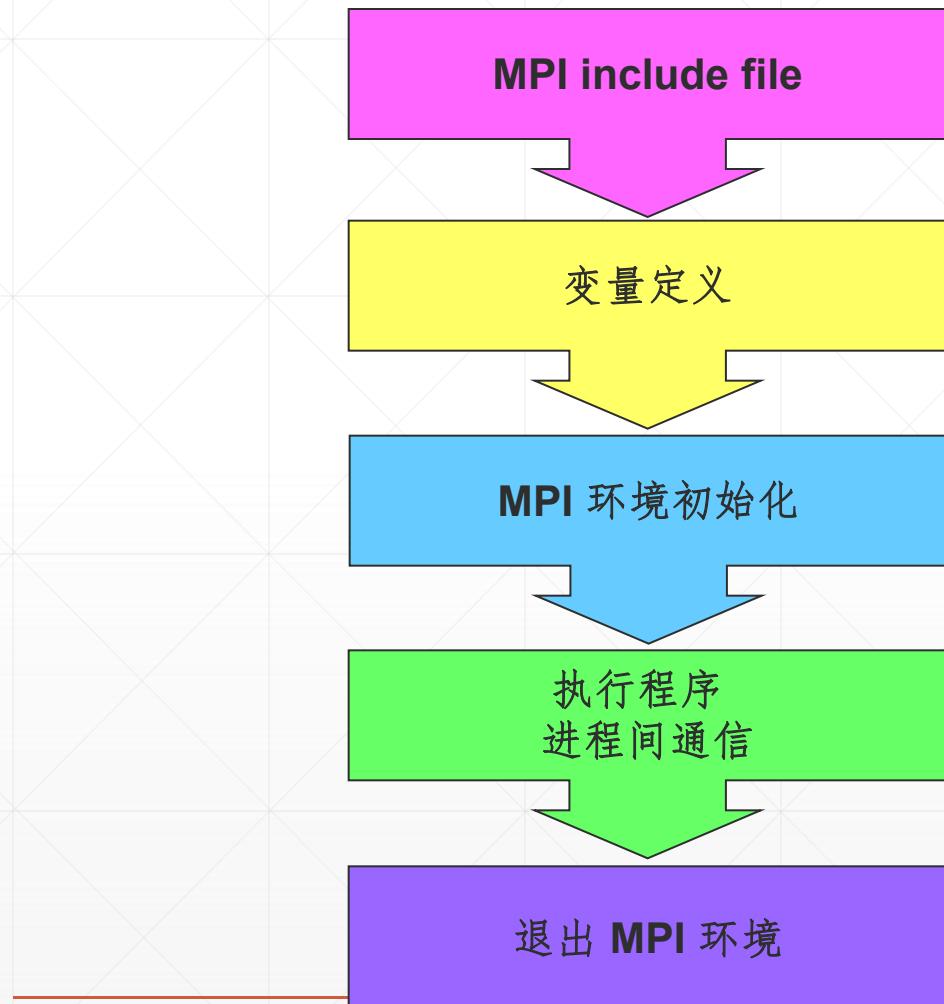
# MPI原始数据类型

| <b>MPI Datatype</b> | <b>C Datatype</b>  |
|---------------------|--------------------|
| MPI_CHAR            | Signed char        |
| MPI_SHORT           | Signed short int   |
| MPI_INT             | Signed int         |
| MPI_LONG            | Signed long int    |
| MPI_UNSIGNED_CHAR   | Unsigned char      |
| MPI_UNSIGNED_SHORT  | Unsigned short int |
| MPI_UNSIGNED        | Unsigned int       |
| MPI_UNSIGNED_LONG   | Unsigned long int  |
| MPI_FLOAT           | Float              |
| MPI_DOUBLE          | Double             |
| MPI_LONG_DOUBLE     | Long double        |
| MPI_BYTE            |                    |
| MPI_PACKED          |                    |

**MPI\_BYTE** 一个字节

**MPI\_PACKED** 打包数据

# MPI程序基本结构



```
#include <mpi.h>
void main (int argc, char *argv[])
{
    int np, rank, ierr;
    ierr = MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&np);
    /*      Do Some Works           */
    ierr = MPI_Finalize();
}
```

# 例子

```
#include "mpi.h"
```

头文件

```
int main(int argc, char ** argv)
```

```
{
```

```
    int myid, numprocs;
```

相关变量声明

```
    int namelen;
```

```
    char processor_name[MPI_MAX_PROCESSOR_NAME];
```

```
    MPI_Init(&argc,&argv);
```

程序开始

```
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

```
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
```

```
    MPI_Get_processor_name(processor_name,&namelen);
```

程序体计算与通信

```
    fprintf(stderr,"Hello World! Process %d of %d on %s\n",
```

```
            myid, numprocs, processor_name);
```

```
    MPI_Finalize();
```

程序结束

```
}
```

# MPI几个基本函数

- Index
  - MPI\_Init      **初始化**
  - MPI\_Initialized
  - MPI\_Comm\_size      **进程数**
  - MPI\_Comm\_rank      **进程号**
  - MPI\_Finalize      **结束**
  - MPI\_Abort
  - MPI\_Get\_processor\_name
  - MPI\_Get\_version

---

- MPI\_Wtime      **时间**

# MPI几个基本函数

- 初始化 MPI 系统

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

- 通常为第一个调用的MPI函数，除 MPI\_Initialized 外
- 在C接口中，MPI系统通过argc和argv得到命令行参数，并且会把MPI系统专用的参数删除，留下用户的解释参数

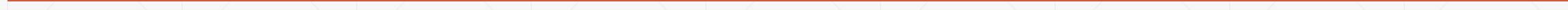


# MPI几个基本函数

- 检测 MPI 系统是否已经初始化

`int MPI_Initialized(int *flag)`

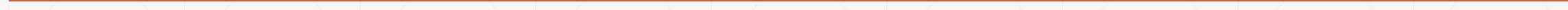
- 唯一可在 MPI\_Init 前使用的函数
- 已经调用MPI\_Init, 返回flag = true, 否则flag = false



# MPI几个基本函数

- 得到通信器的进程数和进程在通信器中的标号

```
int MPI_Comm_size(MPI_Comm comm, int *size)  
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```



# MPI几个基本函数

- 退出 MPI 系统

`int MPI_Finalize(void)`

- 每个进程都必须调用，使用后不准许调用任何MPI函数
- 若不执行MPI退出函数，进程可能被悬挂
- 用户在调用该函数前，应确保非阻塞通讯结束



# MPI几个基本函数

- 异常终止MPI程序

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

- 在出现了致命错误而希望异常终止MPI程序时执行
- MPI系统会设法终止comm通信器中所有进程
- 输入整型参数errorcode，将被作为进程的退出码返回给系统



# MPI几个基本函数

- 获取墙上时间

`double MPI_Wtime(void)`

- 返回调用时刻的墙上时间，用浮点数表示秒数
- 经常用来计算程序运行时间



# Sample :mpi\_1

C+MPI

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
void main(int argc, char *argv[ ])
{
    int myid, numprocs, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Get_processor_name(processor_name,&namelen);
    printf("Hello World! Process %d of %d on %s\n",myid, numprocs,
processor_name);
    MPI_Finalize();
}
```

# Sample :mpi\_1

C+MPI

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
void main(int argc, char *argv[ ])
{
    int myid, numprocs, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Get_processor_name(processor_name,&namelen);
    printf("Hello World! Process %d of %d on %s\n",myid, numprocs,
processor_name);
    MPI_Finalize();
}
```

# Sample :Hello World

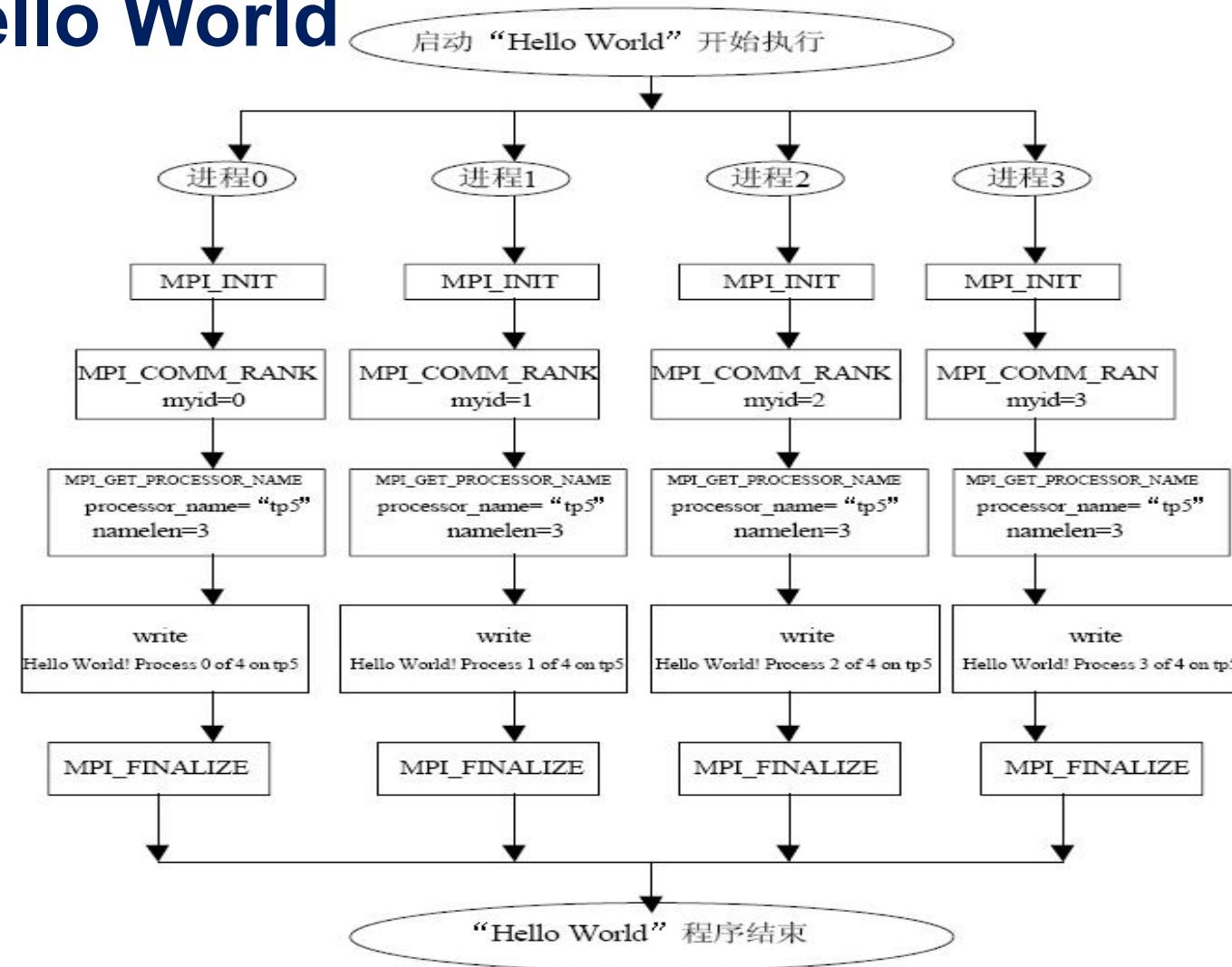
单处理器(tp5)运行4个进程

```
[sc92571@ln112%bscc-a3 ~]$ cat slurm-401091.out
Hello World! Process 1 of 4 on ed0203.para.bscc
Hello World! Process 0 of 4 on ed0203.para.bscc
Hello World! Process 2 of 4 on ed0203.para.bscc
Hello World! Process 3 of 4 on ed0203.para.bscc
```

2 个处理器(tp1,tp2,tp3,tp4)分别运行4个进程

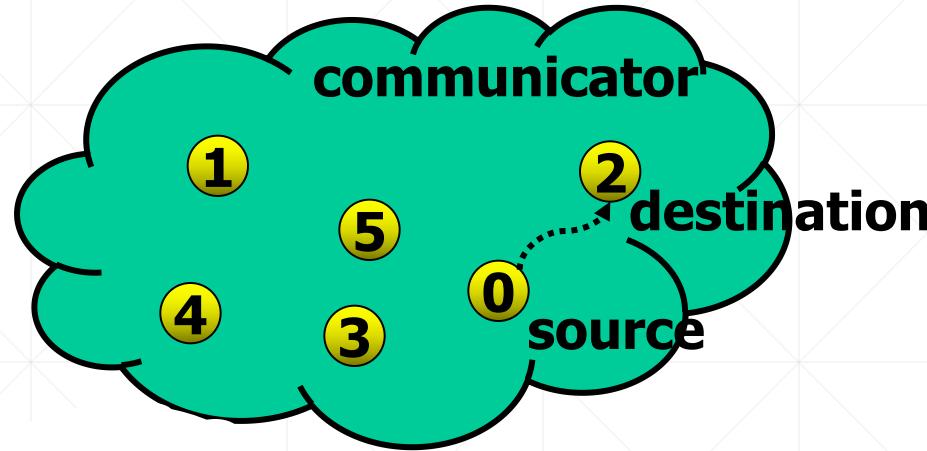
```
[sc92571@ln112%bscc-a3 ~]$ cat slurm-401094.out
Hello World! Process 4 of 8 on ed0203.para.bscc
Hello World! Process 0 of 8 on eb1210.para.bscc
Hello World! Process 6 of 8 on ed0203.para.bscc
Hello World! Process 7 of 8 on ed0203.para.bscc
Hello World! Process 5 of 8 on ed0203.para.bscc
Hello World! Process 2 of 8 on eb1210.para.bscc
Hello World! Process 3 of 8 on eb1210.para.bscc
Hello World! Process 1 of 8 on eb1210.para.bscc
```

# Sample :Hello World



# 点对点通信

- 定义
  - 阻塞式点对点通信
  - 编写安全的MPI程序
  - 其他阻塞式点对点通信函数
  - 阻塞式消息发送模式
  - 非阻塞式点对点通信
-



- 两个进程之间的通信
- 源进程发送消息到目标进程
- 目标进程接受消息
- 通信发生在同一个通信器内
- 进程通过其在通信器内的标号表示

MPI系统的所有通信方式都建立在点对点通信之上

# 阻塞式点对点通信

- Index
  - MPI\_Send
  - MPI\_Recv
  - MPI\_Get\_count
  - MPI\_Sendrecv



# 阻塞式点对点通信

## ■ 阻塞式消息发送

C:

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

- count 不是字节数，而是指定数据类型的个数
- 例如发送int a[10]这个数组的全部数据，COUNT为10，datatype为MPI\_INT
- 总计发送字节数为sizeof(MPI\_INT)\*COUNT
- datatype可是原始数据类型MPI\_INT MPI\_FLOAT等，或为用户自定义类型
- dest 是目标进程号，取值范围是 0 ~ np - 1，或MPI\_PROC\_NULL  
(np是comm中的进程总数)
- tag 取值范围是 0 ~ MPI\_TAG\_UB，用来区分消息

# 阻塞式点对点通信

- 阻塞式消息接收

C:

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

- count是接受缓存区的大小，表示接受上界，具体接受长度可用MPI\_Get\_count 获得
- source 取值范围是 0 ~ np – 1, 或MPI\_PROC\_NULL和MPI\_ANY\_SOURCE
- tag 取值范围是 0 ~ MPI\_TAG\_UB, 或MPI\_ANY\_TAG



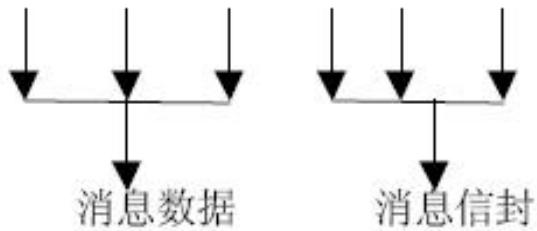
# 阻塞式点对点通信

- 消息 (message)

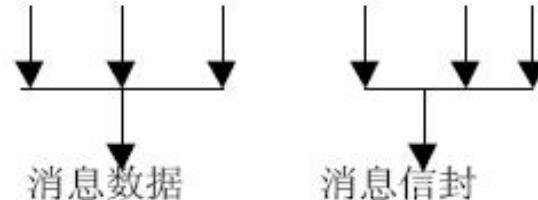
信封: <源/目, 标识, 通信域>

数据: <起始地址, 数据个数, 数据类型>

MPI\_SEND( buf, count,datatype,dest,tag,comm)



MPI\_RECV(buf,count,datatype,source,tag,comm,status)



# 阻塞式点对点通信

- status的内容
  - C中是一个数据结构为MPI\_Status的参数， 用户可以直接访问的三个域（共5个域）

```
typedef struct {  
... ...int MPI_SOURCE;  
    int MPI_TAG;  
    int MPI_ERROR; ... ...  
} MPI_Status;
```

消息源地址  
消息标号  
接收操作的错误码

- 使用前需要用户为其申请存储空间 (MPI\_Status status;)
- C中引用时为 status.MPI\_SOURCE ...

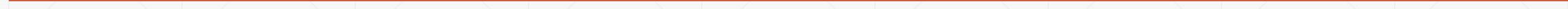
# 阻塞式点对点通信

- 查询接收到的消息长度

C:

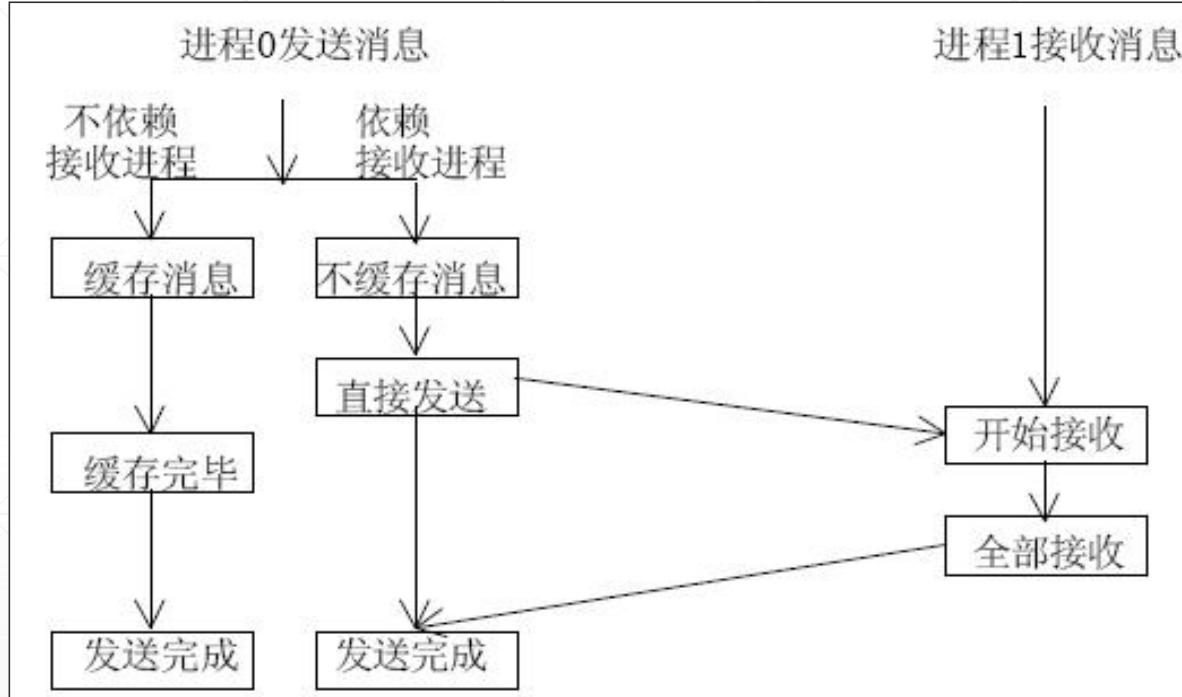
```
int MPI_Get_count(MPI_Status status, MPI_Datatype datatype, int *count)
```

- 该函数在count中返回数据类型的个数，即消息的长度
- count属于MPI\_Status结构的一个域，但不能被用户直接访问



# 定义

## ■ 标准阻塞式通信



- 是否对发送数据进行缓存，由MPI系统决定，而非程序员
- 阻塞：发送成功，意味（1）消息成功发送；（2）或者消息被缓存  
接收成功，意味消息已被成功接收

# 阻塞式点对点通信

- 消息传递成功
  - 发送进程需指定一个有效的目标接收进程
  - 接收进程需指定一个有效的源发送进程
  - 接收和发送消息的进程要在同一个通信器内
  - 接收和发送消息的 tag 要相同
  - 接收缓存区要足够大



# 阻塞式点对点通信

- 任意源进程（接收操作可以接受任意进程的消息）

MPI\_ANY\_SOURCE

- 任意标号（接收操作可以接受任意标号的消息）

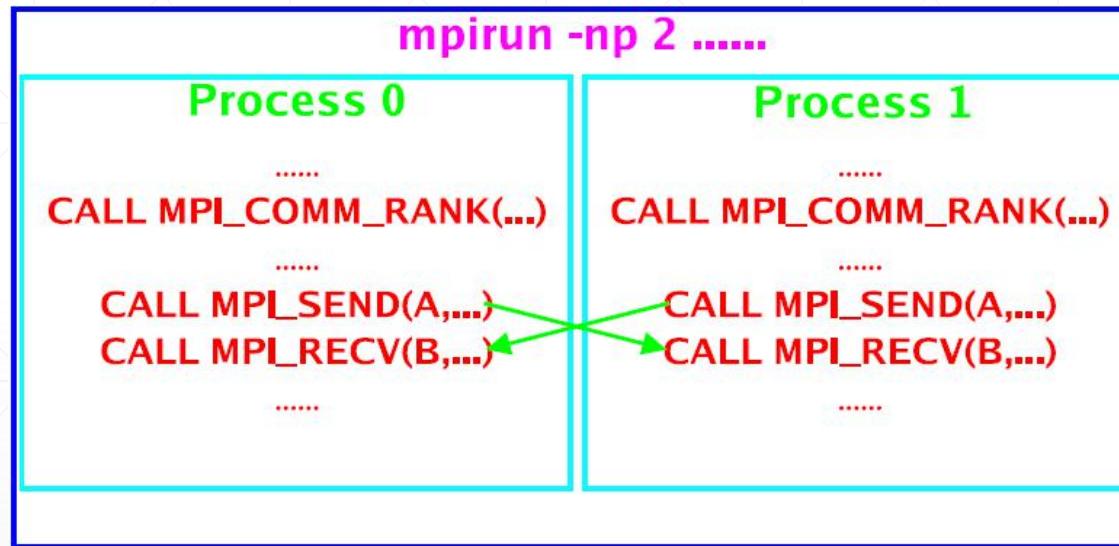
MPI\_ANY\_TAG

- 真实的源进程与消息标号可以访问接受函数中的status参数获得



# 例子

- 阻塞型消息传递:示例



# Sample - mpi\_2

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
void main(int argc, char* argv[]){
    int numprocs, myid;
    MPI_Status status;
    char message_sned[100];
    char message_recieve[100];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

    sprintf(message_sned, "Hello World! This is process %d", myid);
    MPI_Send(message_sned, 100, MPI_CHAR, (myid + 1) % 2, // 0号进程向1发送, 同时1号进程向0号发送
            99, MPI_COMM_WORLD);
    MPI_Recv(message_recieve, 100, MPI_CHAR, (myid + 1) % 2, // 接受其他进程发送的消息
            99, MPI_COMM_WORLD, &status);
    printf("接收到第%d号进程发送的消息: %s\n", (myid + 1) % 2, message_recieve);

    MPI_Finalize();
}
```

# 其他阻塞式点对点通信函数

- 捆绑发送和接收

MMPI\_Sendrecv (sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

OUT sendbuf 发送缓冲区起始地址 OUT sendcount 发送数据的个数

OUT sendtype 发送数据的数据类型 OUT dest 目标进程的标识号

OUT sendtag 发送消息标签

IN recvbuf 接收缓冲区初始地址 IN recvcount 最大接收数据个数

IN recvtype 接收数据的数据类型 IN source 源进程标识

IN recvtag 接收消息标签

comm 通信器 status 返回的状态

语义上等同于一个发送和一个接收操作结合，但此函数可以有效避免在单独发送和接收操作过程中，由于调用次序不当而造成的死锁。MPI系统会优化通信次序，从而最大限度避免错误发生。

# Sample - mpi\_2s

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
void main(int argc, char* argv[]){
    int numprocs, myid;
    MPI_Status status;
    char message_sned[100];
    char message_recieve[100];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    sprintf(message_sned, "Hello World! This is process %d", myid);

    MPI_Sendrecv(message_sned, 100, MPI_CHAR, (myid + 1) % 2, 99, // 0号进程向1发送, 同时1号进程向0号发送
                message_recieve, 100, MPI_CHAR, (myid + 1) % 2, 99, MPI_COMM_WORLD, &status); // 接受其他进程发送的消息

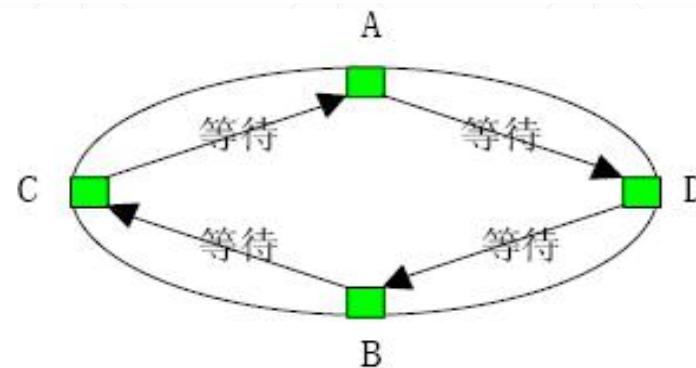
    printf("接收到第%d号进程发送的消息: %s\n", (myid + 1) % 2, message_recieve);
    MPI_Finalize();
}
```

# 编写安全的MPI程序

- Pro0发送消息到Pro1，同时，Pro1发送消息到Pro0

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF( rank .EQ. 1)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

A  
C  
B  
D



死锁

# Sample - mpi\_3

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
void main(int argc, char* argv[]){
    int numprocs, myid, source;
    MPI_Status status;
    char message[100];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    strcpy(message, "Hello World!");

    MPI_Recv(message, 100, MPI_CHAR, (numprocs+myid-1)%numprocs, 99, // 接受进程号-1的进程的消息
             MPI_COMM_WORLD, &status);
    MPI_Send(message, 100, MPI_CHAR, (myid+1)%numprocs , 99, // 向进程号+1的进程发送消息
             MPI_COMM_WORLD);

    printf("接收到第%d号进程发送的消息: %s\n", numprocs+myid-1 , message);
    MPI_Finalize();
}
```

# Sample - mpi\_3

注意此时程序一直卡住，没有结束

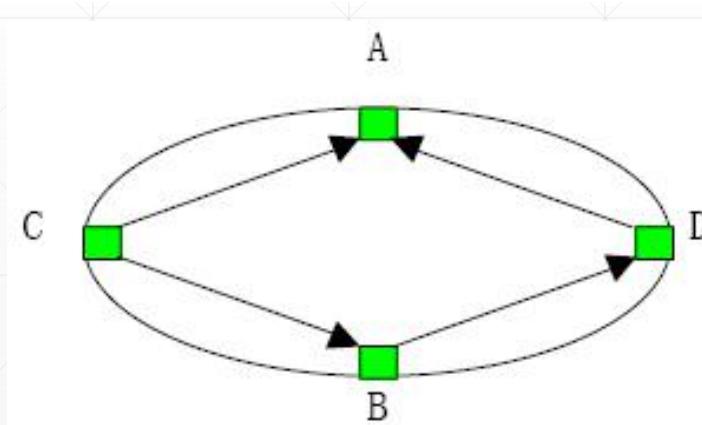
```
[sc92571@ln112%bscc-a3 ~]$ sbatch mpi_3.slurm
Submitted batch job 401616
[sc92571@ln112%bscc-a3 ~]$ cat slurm-401616.out
[sc92571@ln112%bscc-a3 ~]$ squeue
      JOBID PARTITION      NAME      USER ST      TIME  NODES NODELIST (REASON)
        401616    amd_256    ipcc  sc92571 R      0:05      2 eb1202,ec0910
[sc92571@ln112%bscc-a3 ~]$ squeue
      JOBID PARTITION      NAME      USER ST      TIME  NODES NODELIST (REASON)
        401616    amd_256    ipcc  sc92571 R      0:06      2 eb1202,ec0910
[sc92571@ln112%bscc-a3 ~]$ squeue
      JOBID PARTITION      NAME      USER ST      TIME  NODES NODELIST (REASON)
        401616    amd_256    ipcc  sc92571 R      1:30      2 eb1202,ec0910
[sc92571@ln112%bscc-a3 ~]$ cat slurm-401616.out
[sc92571@ln112%bscc-a3 ~]$ squeue
      JOBID PARTITION      NAME      USER ST      TIME  NODES NODELIST (REASON)
        401616    amd_256    ipcc  sc92571 R      2:11      2 eb1202,ec0910
[sc92571@ln112%bscc-a3 ~]$ []
```

手动 scancel 401616 终止任务

```
[sc92571@ln112%bscc-a3 ~]$ squeue
      JOBID PARTITION      NAME      USER ST      TIME  NODES NODELIST (REASON)
        401616    amd_256    ipcc  sc92571 R      2:11      2 eb1202,ec0910
[sc92571@ln112%bscc-a3 ~]$ scancel 401616
[sc92571@ln112%bscc-a3 ~]$ squeue
      JOBID PARTITION      NAME      USER ST      TIME  NODES NODELIST (REASON)
[sc92571@ln112%bscc-a3 ~]$ []
```

# 编写安全的MPI程序

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr) A
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr) C
ELSE (rank .EQ. 1)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr) B
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr) D
ENDIF
```



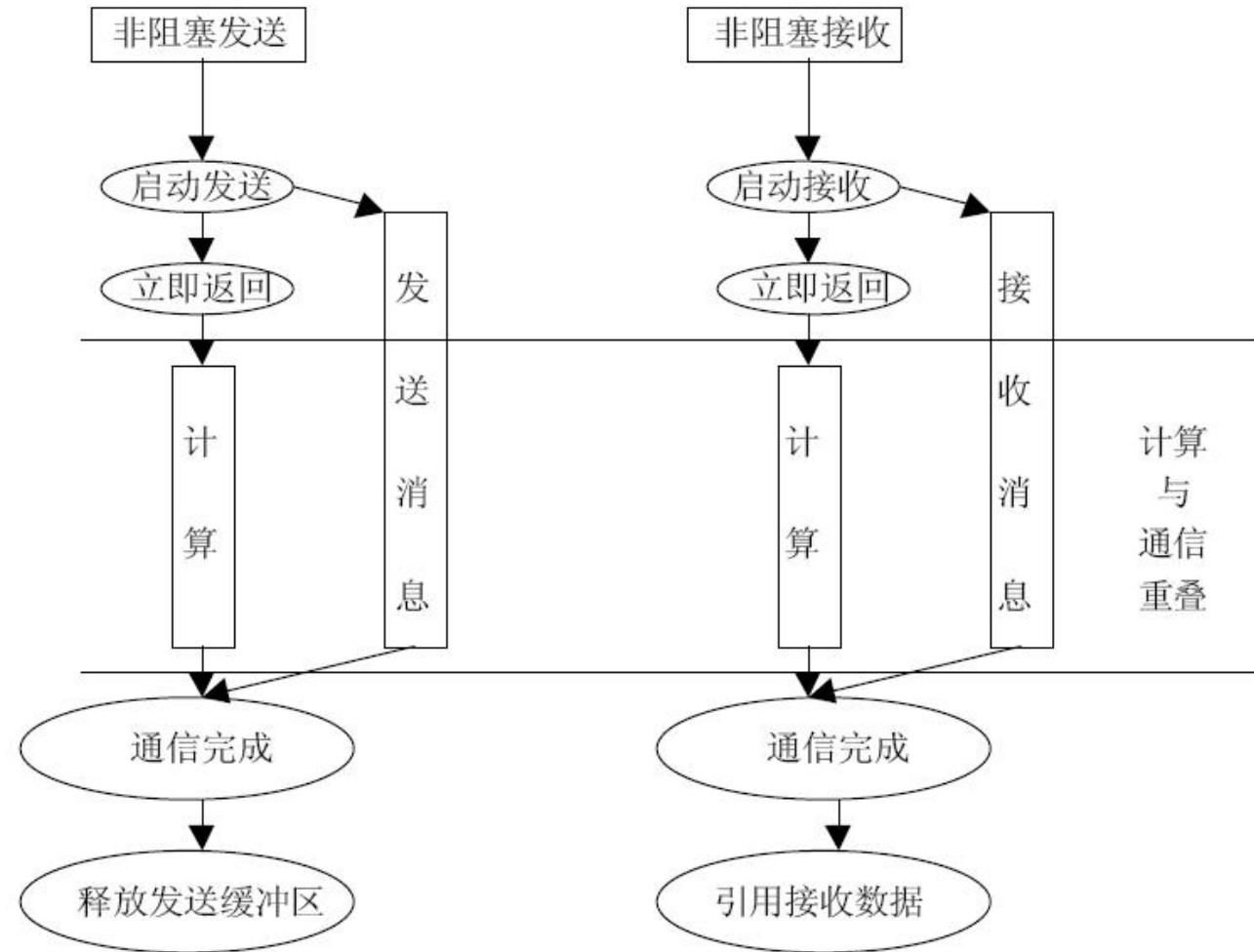
正确

# 非阻塞式点对点通信

- 阻塞式通信与非阻塞式通信

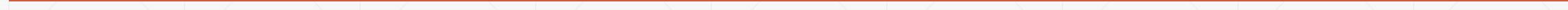
| 通信类型   | 函数返回                                                                                                             | 对数据区操作                        | 特性                                                                                  |
|--------|------------------------------------------------------------------------------------------------------------------|-------------------------------|-------------------------------------------------------------------------------------|
| 阻塞式通信  | <ol style="list-style-type: none"><li>1. 阻塞型函数需要等待指定操作完成返回</li><li>2. 或所涉及操作的数据要被<b>MPI</b>系统缓存安全备份后返回</li></ol> | 函数返回后，对数据区操作是安全的              | <ol style="list-style-type: none"><li>1. 程序设计相对简单</li><li>2. 使用不当容易造成死锁</li></ol>   |
| 非阻塞式通信 | <ol style="list-style-type: none"><li>1. 调用后立刻返回，实际操作在<b>MPI</b>后台执行</li><li>2. 需调用函数等待或查询操作的完成情况</li></ol>      | 函数返回后，即操作数据区不安全。可能与后台正进行的操作冲突 | <ol style="list-style-type: none"><li>1. 可以实现计算与通信的重叠</li><li>2. 程序设计相对复杂</li></ol> |
|        |                                                                                                                  |                               |                                                                                     |

# 非阻塞式点对点通信



# 非阻塞式点对点通信

- Index
  - MPI\_Isend/MPI\_Irecv
  - MPI\_Wait/MPI\_Waitany/MPI\_Waitall/MPI\_Waitsome
  - MPI\_Test/MPI\_Testany/MPI\_Testall/MPI\_Testsome
  - MPI\_Request\_free
  - MPI\_Cancel
  - MPI\_Test\_cancelled
  - MPI\_Probe/MPI\_Iprobe



# 非阻塞式点对点通信

## ■ 非阻塞式发送

C

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

- 该函数仅提交了一个消息发送请求，并立即返回
- MPI系统会在后台完成消息发送
- 函数为该发送操作创建了一个请求，通过**request**变量返回
- **request**可供之后（查询和等待）函数使用

# 非阻塞式点对点通信

## ■ 非阻塞式接收

C

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

- 该函数仅提交了一个消息接收请求，并立即返回
- MPI系统会在后台完成消息接收
- 函数为该接收操作创建了一个请求，通过**request**变量返回
- **request**可供之后查询和等待函数使用

# 非阻塞式点对点通信

- 等待、检测一个通信请求的完成

C

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)

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

- MPI\_Wait 阻塞等待通信函数完成后返回；MPI\_Test检测某通信，不论其是否完成，都立刻返回。如果通信完成，则flag=true
- 当等待或检测的通信完成时，通信请求request被设置成MPI\_REQUEST\_NULL
- 考察接收请求，status返回与MPI\_Recv一样；发送请求，则不确定
- MPI\_Test返回时，当flag=false, status不被赋值

# 非阻塞式点对点通信

- 等待、检测一组通信请求中某一个的完成

C

```
int MPI_Waitany(int count,  
                 MPI_Request *array_of_requests,  
                 int *index, MPI_Status *status)  
  
int MPI_Testany(int count,  
                MPI_Request *array_of_requests,  
                int *index, int *flag, MPI_Status *status)
```

- `count`表示通信请求的个数
- `array_of_requests`是一组非阻塞通信的请求
- `index`存储一个成功完成的通信在`array_of_requests`中的位置
- `flag`表示是否有任意一个通信请求完成，若有`flag=true`
- 完成的通信请求`request`被自动赋值`MPI_REQUEST_NULL`
- `MPI_Testany`返回时，当`flag=false`, `status`不被赋值

# 非阻塞式点对点通信

- 等待、检测一组通信请求的全部完成

C

```
int MPI_Waitall(int count,  
                 MPI_Request *array_of_requests,  
                 MPI_Status *array_of_statuses)  
  
int MPI_Testall(int count,  
                MPI_Request *array_of_requests,  
                int *flag, MPI_Status *array_of_statuses)
```

- **count**表示通信请求的个数
- **array\_of\_requests**是一组非阻塞通信的请求
- **array\_of\_statuses**返回该组通信完成的状态
- **flag**表示全部通信是否完成，若完成**flag=true**
- **MPI\_Testall**返回时，当**flag=false**, **array\_of\_statuses**不被赋值

# 非阻塞式点对点通信

- 等待、检测一组通信请求的部分完成

C

```
int MPI_Waitsome(int incount,
                  MPI_Request *array_of_requests,
                  int outcount, int *array_of_indices,
                  MPI_Status *array_of_statuses)

int MPI_Testsome(int incount,
                  MPI_Request *array_of_requests,
                  int outcount, int *array_of_indices,
                  MPI_Status *array_of_statuses)
```

- **MPI\_Waitsome**等待至少一个通信完成才返回
- **outcount**表示通信成功完成的个数
- **array\_of\_indices**存储完成的通信在**array\_of\_requests**中的位置
- **array\_of\_statuses**返回完成通信的状态，其他不被赋值
- **MPI\_Testsome**返回时若没有一个通信完成，则**outcount=0**
- **MPI\_Testsome**返回时，当**flag=false**, **array\_of\_statuses**不被赋值

# 阻塞型与非阻塞型通信函数

| 函数类型    | 通信模式 | 阻塞型              | 非阻塞型               |
|---------|------|------------------|--------------------|
| 消息发送函数  | 标准模式 | MPI_Send         | MPI_Isend          |
|         | 缓冲模式 | MPI_Bsend        | MPI_Ibsend         |
|         | 同步模式 | MPI_Ssend        | MPI_Issend         |
|         | 就绪模式 | MPI_Rsend        | MPI_Irsend         |
| 消息接收函数  |      | MPI_Recv         | MPI_Irecv          |
| 消息检测函数  |      | MPI_Probe        | MPI_Iprobe         |
| 等待/查询函数 |      | MPI_Wait         | MPI_Test           |
|         |      | MPI_Waitall      | MPI_Testall        |
|         |      | MPI_Waitany      | MPI_Testany        |
|         |      | MPI_Waitsome     | MPI_Testsome       |
| 释放通信请求  |      | MPI_Request_free |                    |
| 取消通信    |      |                  | MPI_Cancel         |
|         |      |                  | MPI_Test_cancelled |

# 集合通讯

## MPI 集合通信函数

| 类型   | 函数                 | 功能                      |
|------|--------------------|-------------------------|
| 数据移动 | MPI_Bcast          | 一到多，数据广播                |
|      | MPI_Gather         | 多到一，数据汇合                |
|      | MPI_Gatherv        | MPI_Gather的一般形式         |
|      | MPI_Allgather      | MPI_Gather的一般形式         |
|      | MPI_Allgatherv     | MPI_Allgather的一般形式      |
|      | MPI_Scatter        | 一到多，数据分散                |
|      | MPI_Scatterv       | MPI_Scatter的一般形式        |
|      | MPI_Alltoall       | 多到多，置换数据（全交换）           |
|      | MPI_Alltoallv      | MPI_Alltoall的一般形式       |
| 数据聚集 | MPI_Reduce         | 多到一，数据归约                |
|      | MPI_Allreduce      | MPI_Reduce的一般形式，结果在所有进程 |
|      | MPI_Reduce_scatter | 结果scatter到每个进程          |
|      | MPI_Scan           | 前缀操作                    |
| 同步   | MPI_Barrier        | 同步操作                    |

# 总结&答疑

本讲内容：

- 1、参赛平台使用
- 2、OpenMP多线程编程
- 3、MPI点对点通讯

下讲预告：

现代处理器优化技术——邵奇  
讲述如何针对CPU特性进行优化，  
尤其是Cache访存优化、AVX向量化等



↑↑扫码参加IPCC      ↓↓学习交流群



IPCC系列活动意见征集调查~

# IPCC

## 谢谢观看

时间：2021年6月

主讲人：张力越