# 并行计算

Tuesday, May 18, 2010

## 并行计算

- 并行计算
- 任务分解
- 并行程序
- EFEP90程序并行化

## 并行计算

- 什么是并行计算

  多个处理器共同完成一件单个处理器无法完成或单个处理器在有限时间内无法完成的任务

例：10张CD压缩MP3

| | 方法一 | 方法二 |
|---|---|---|
| 描述 | 一台计算机<br>一张接一张压缩 | 十台计算机<br>同时压缩 |
| 每台计算机压一张CD耗时 | $t_0$ | $t_0$ |
| 总共完成耗时 | $10t_0$ | $t_0$ |

## 并行计算

- 为什么要用并行计算

  **To pull a bigger wagon, it is easier to add more oxen than to grow a gigantic ox**

- 并行计算主要应用领域
  - 物理　天文
  - 化学　化工
  - 生物　医学
  - 力学　机械

## 并行计算 — 常用衡量参数

- **加速比**

  串行计算所用的时间与并行计算所用的时间之比

$$S(n,p)=\frac{T_s(n)}{T_p(n,p)}\quad (0<S(n,p)\le p)$$

其中

- $n$　计算规模
- $p$　并行计算中，所用处理器的个数
- $T_s$　串行计算所用时间
- $T_p$　并行计算所用时间

若$S(n,p)=p$为**线性加速比**

## 并行计算 — 常用衡量参数

- **并行效率**

  并行计算中处理器的利用率与串行计算处理器的利用率相比

$$E(n,p)=\frac{S(n,p)}{p}=\frac{T_s(n)}{pT_p(n,p)}\quad (0<E(n,p)\le 1)$$

$$\text{if } E(n,p)<\frac{1}{p},\ \text{then } T_s(n)<T_p(n,p)$$

用并行计算不如串行计算快？

## 并行计算 — 影响并行效率因素

影响并行效率的主要因素：

- 求解问题本身 — 影响效率最重要的因素
- 程序代码水平
  由程序员决定
- 并行机
  通信延迟小，内存带宽高的并行机效率高

## 任务分解

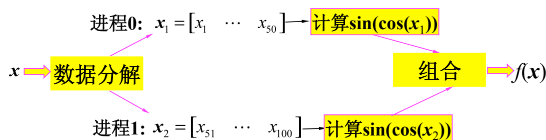
- 并行计算任务分解
  将计算任务分解，给予不同的处理器处理
- 分解的方法
  - 数据并行
    将要处理的数据分成小块，不同进程处理不同的数据
  - 过程并行
    将要处理数据的过程进行分解，不同的处理器完成不同的过程，不同处理器执行的指令是不同的

## 任务分解实例 — 数据并行

已知：$\boldsymbol{x} = [x_1 \quad x_2 \quad \cdots \quad x_{99} \quad x_{100}]$

求 $f(\boldsymbol{x})$=sin(cos($\boldsymbol{x}$))

数据并行处理过程

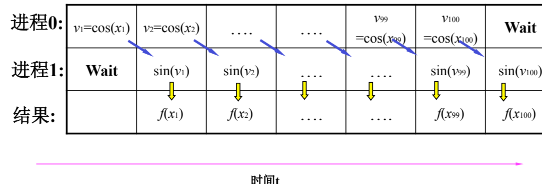



## 任务分解实例 — 过程并行

已知：$\boldsymbol{x} = [x_1 \quad x_2 \quad \cdots \quad x_{99} \quad x_{100}]$

求 $f(\boldsymbol{x})$=sin(cos($\boldsymbol{x}$))

过程并行处理过程




## 任务分解实例 — 数据并行

Jacobi迭代求解线性方程组 $\boldsymbol{Ax} = \boldsymbol{b}$

Jacobi迭代过程：

1.选取初始解：$\boldsymbol{x}_0$

2.迭代：$\boldsymbol{x}^{(k)} \Rightarrow \boldsymbol{x}^{(k+1)}$

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)})$$

3.判断是否满足条件：

$$\left\| x^{(k+1)} - x^{(k)} \right\| \le \varepsilon$$

## 任务分解实例 — 数据并行

| 进程号： | 负责计算 |
|---|---|
| 0 | $x_1 \quad x_{p+1} \quad \ldots$ |
| 1 | $x_2 \quad x_{p+2} \quad \ldots$ |
| … | … |
| $p$-1 | $x_p \quad x_{2p} \quad \ldots$ |

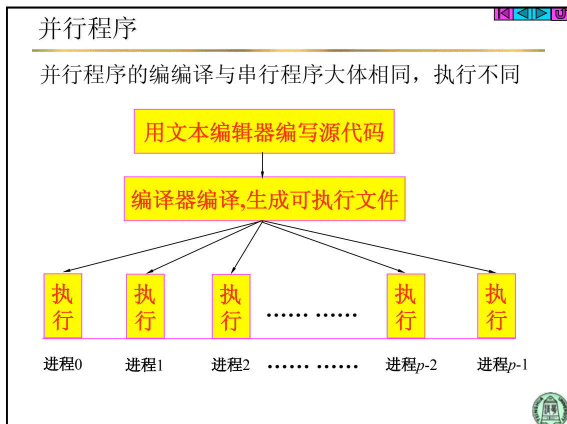
第$k$步 $x_1$ $x_2$ $x_3$ … $x_{p-1}$ $x_p$ $x_{p+1}$ $x_{p+2}$ $x_{p+3}$ … $x_{2p-1}$ $x_{2p}$ $x_{2p+1}$ $x_{2p+2}$ $x_{2p+3}$ … $x_{3p-1}$ $x_{3p}$ … … …

第$k$＋1步 $x_1$ $x_2$ $x_3$ … $x_{p-1}$ $x_p$ $x_{p+1}$ $x_{p+2}$ $x_{p+3}$ … $x_{2p-1}$ $x_{2p}$ $x_{2p+1}$ $x_{2p+2}$ $x_{2p+3}$ … $x_{3p-1}$ $x_{3p}$ … … …

需要在不同的处理器间通信！

## 并行程序

并行程序的编编译与串行程序大体相同，执行不同




## 并行程序

- 实现途径
  - 共享存储
    - OpenMP编程
    - 多线程编程
  - 消息传递
    - MPI (Message Passing Interface)编程
- 编程语言
  - C
  - C++
  - Fortran

## 并行程序 — 消息传递库MPI

- MPI的全称是Message Passing Interface
  - 使用方便，用户不必关心底层通信
- MPI是一个库，不是一个专门的语言
  - 遵守库函数/过程的调用规则
- MPI是一个标准，不是一个具体的实现
  - 具体实现由并行机厂家提供，也有免费下载
  - 常见的实现方式MPICH、LAM
- 用MPI写的并行程序优点
  - 通信性能高
  - 程序的可移植性好
  - 广泛的软硬件支持

## 并行程序 — MPI并行程序实例

并行打印“Hello!”，并打印进程号及总进程数

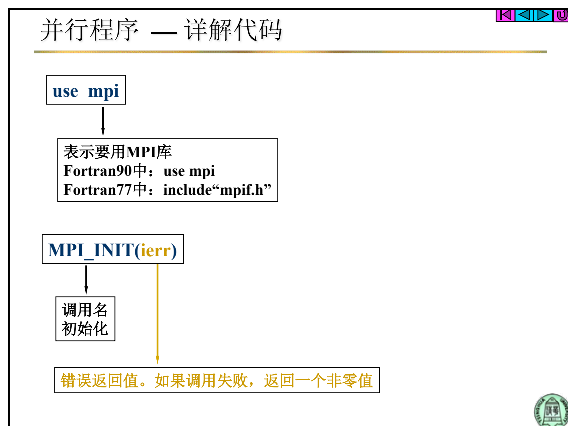
```
program hello
use mpi
implicit none

integer myid,nproces,ierr

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)

write(*,*)"Hello!","Total Proces is:",nproc,"My process is:", myid

call MPI_FINALIZE(ierr)

end
```

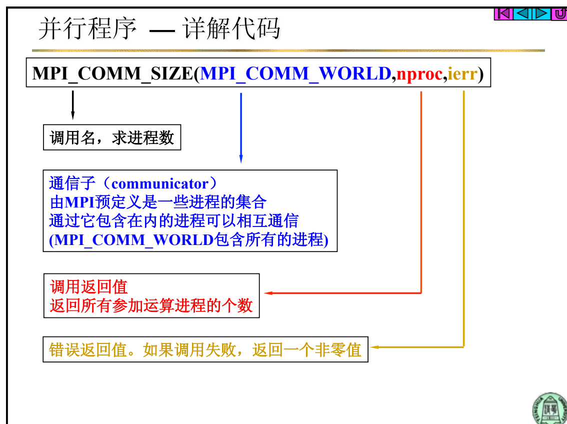
## 并行程序 — 详解代码

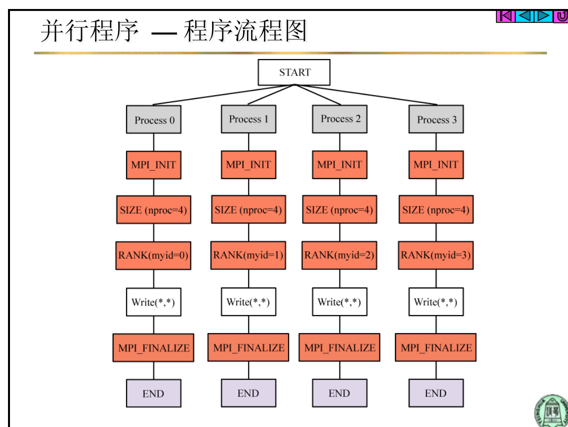
```
program hello
use mpi

implicit none
……
call MPI_INIT

call MPI_COMM_SIZE…

call MPI_COMM_RANK…

……
 call MPI_FINALIZE
……
end
```

- use mpi → FORTRAN90中并行程序必不可少的调用，在定义变量之前，表示要使用MPI库
- call MPI_INIT → 初始化调用，程序的第一个MPI调用，所有的其它MPI调用必须在初始化调用之后
- call MPI_COMM_SIZE… → 返回所有参加运算进程的个数
- call MPI_COMM_RANK… → 返回进程的进程号
- call MPI_FINALIZE → 结束MPI调用。程序当中最后一个MPI调用，所有的MPI调用必须在这个调用之前

## 并行程序 — 详解代码





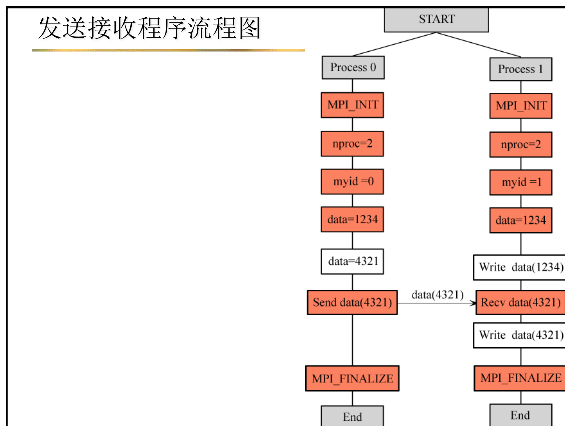


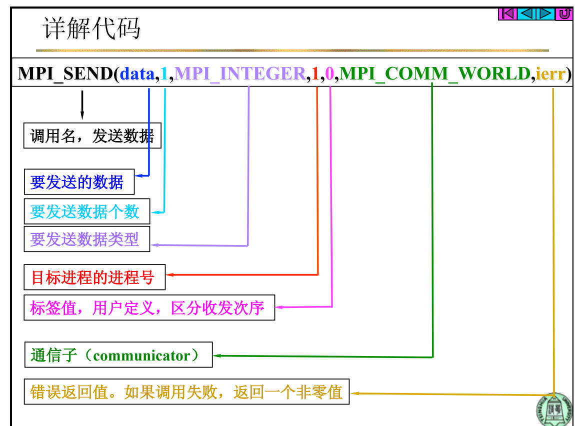

## 并行程序 — MPI程序模版

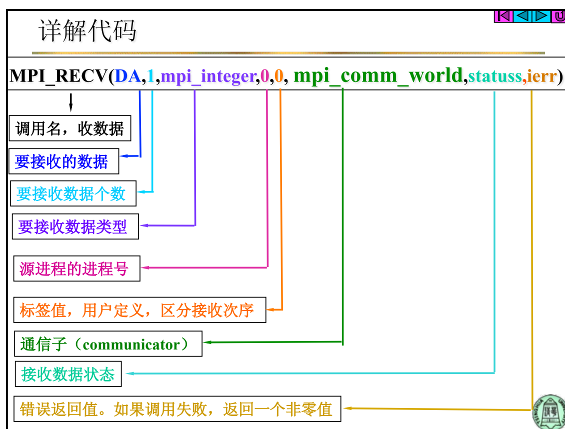
```
program programname
use mpi

[定义变量]

call MPI_INIT(ierr)     !初始化MPI调用
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproces,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)

[程序代码(可以是MPI调用，也可以不是MPI调用)]

call MPI_FINALIZE(ierr)   !结束MPI调用

[程序代码(不可以是MPI调用)]
end program
```

## MPI程序 — 发送和接收数据

```
program main
use mpi
……
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
data=1234
If (myid == 0) then
   data=4321
   call MPI_SEND(data,1,MPI_INTEGER,1,0,MPI_COMM_WORLD,ierr)
else if (myid == 1) then
   write(*,*)"Before recv:  Myid=",myid,"DATA=", data
   call MPI_RECV(data,1,MPI_INTEGER,0,0,MPI_COMM_WORLD,  &
                  status,ierr)
   write(*,*)"After  recv:  Myid=", myid, "DATA=", data
end if
call MPI_FINALIZE(ierr)
end
```

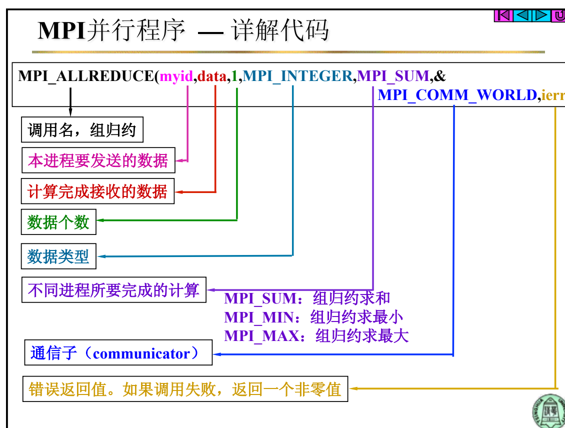


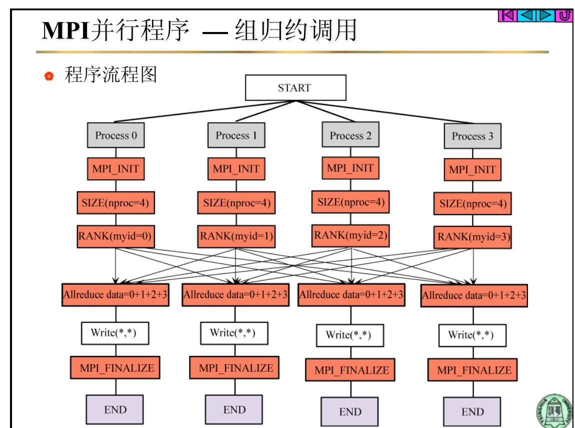



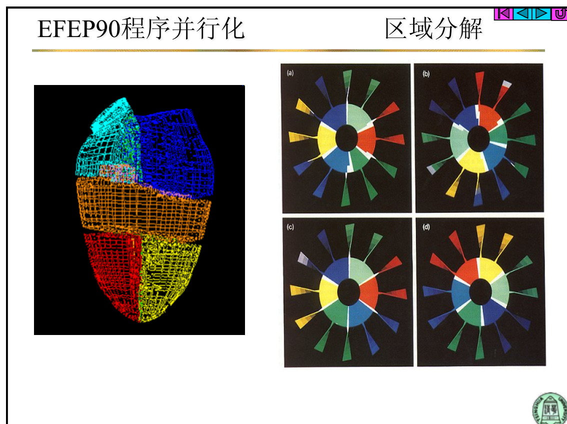




## MPI并行程序 —组归约调用

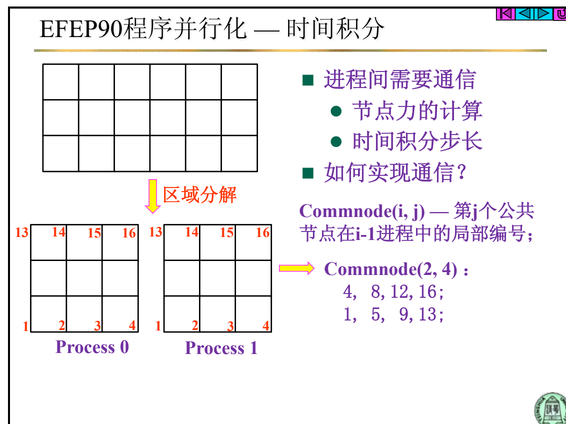
```
program main
use mpi
implicit none
integer myid,nprocs,ierr
integer data

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)

call MPI_ALLREDUCE(myid,data,1,MPI_INTEGER,&
                   MPI_SUM,MPI_COMM_WORLD,ierr)

write(*,*)"Myid=",myid,"data=",data
call MPI_FINALIZE(ierr)
end
```



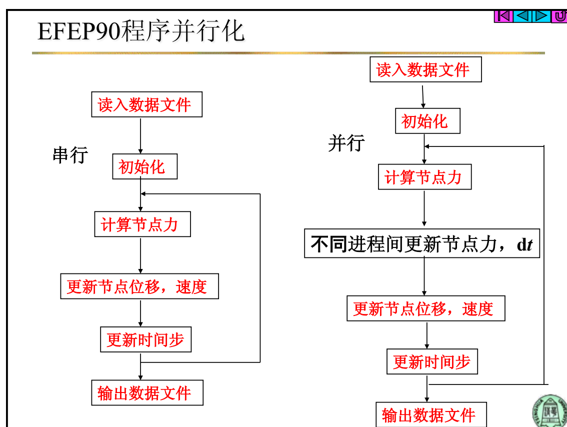





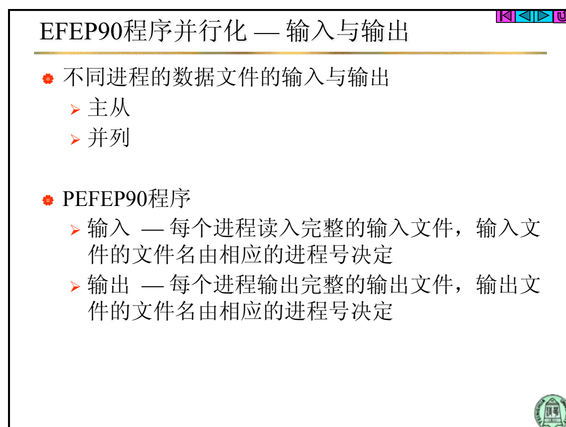
EFEP90程序并行化
区域分解

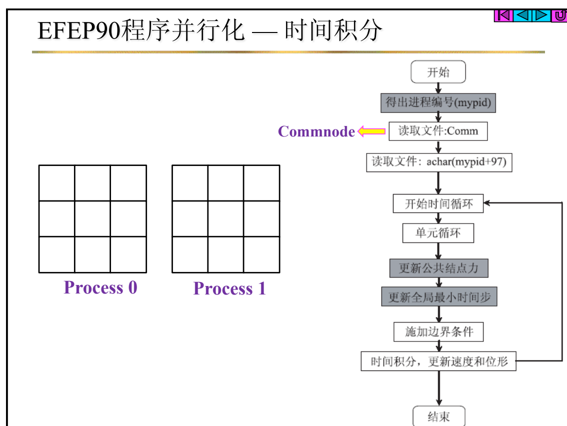
EFEP90程序并行化 — 时间积分
区域分解
Process 0
Process 1
进程间需要通信
节点力的计算
时间积分步长
如何实现通信？
Commnode(i, j) — 第j个公共节点在i-1进程中的局部编号；
Commnode(2, 4)：
4, 8, 12, 16；
1, 5, 9, 13；

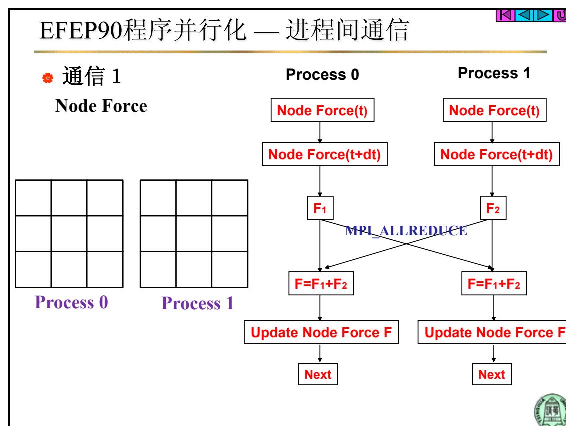
EFEP90程序并行化
串行
读入数据文件
初始化
计算节点力
更新节点位移，速度
更新时间步
输出数据文件
并行
读入数据文件
初始化
计算节点力
不同进程间更新节点力，dt
更新节点位移，速度
更新时间步
输出数据文件

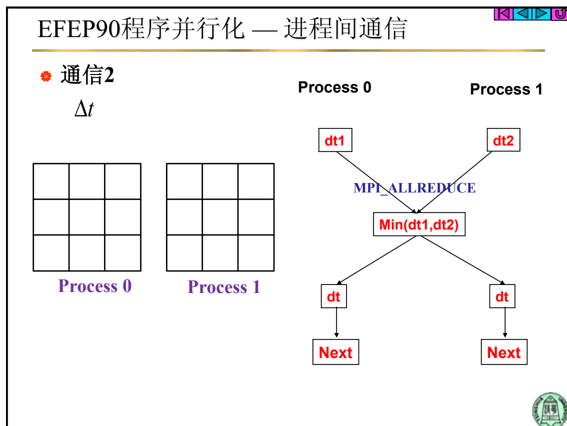
EFEP90程序并行化 — 输入与输出
不同进程的数据文件的输入与输出
主从
并列
PEFEP90程序
输入 — 每个进程读入完整的输入文件，输入文件的文件名由相应的进程号决定
输出 — 每个进程输出完整的输出文件，输出文件的文件名由相应的进程号决定

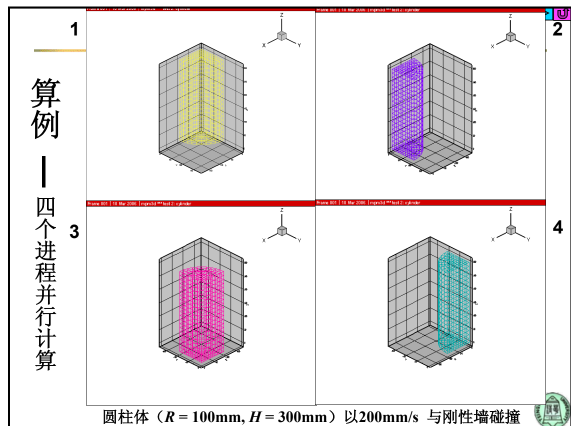



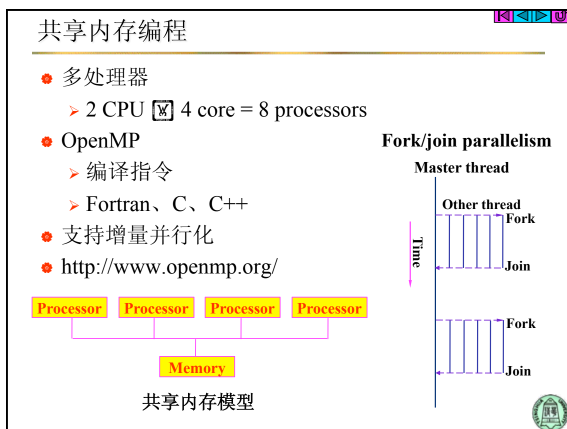
EFEP90程序并行化 — 时间积分
Process 0
Process 1
Commnode
开始
得出进程编号(mypid)
读取文件:Comm
读取文件：achar(mypid+97)
开始时间循环
单元循环
更新公共结点力
更新全局最小时间步
施加边界条件
时间积分，更新速度和位形
结束

EFEP90程序并行化 — 进程间通信
通信 1
Node Force
Process 0
Process 1
Node Force(t)
Node Force(t+dt)
F1
F2
MPI_ALLREDUCE
F=F1+F2
Update Node Force F
Next




## 共享内存编程

**基于C/C++语言的OpenMP程序的结构**

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

## 共享内存编程

- **FORTRAN** 指令格式
  - 固定形式　**!$OMP**
  - 自由形式　**!$OMP, *$OMP, C$OMP**
- **Parallel Region Construct**

```
!$OMP  Parallel  [clause[[,] clause] . . .]
  Do I = 1, 20
    A(I) = A(I) + B(I)
!$OMP End Parallel  (隐含BARRIER操作)
```

其中Clause可以为:**PRIVATE(list), SHARED(list),COPYIN(list), FIRSTPRIVATE(list), DEFAULT(PRIVATE|SHARED|NONE), REDUCTION({operation|intrinsic}:list), IF(logical_expression)**

## 共享内存编程

- 线程总数
  - 环境变量：OMP_NUM_THREADS
  - int omp_get_num_procs(void)
  - void omp_set_num_threads(int t)

```
int t;
…
t = omp_get_num_procs();
Omp_set_num_threads(t);
```

## 共享内存编程

- #pragma omp parallel for

```
#pragma omp parallel for
for (i=first; i<size; i+=prime) marked[i]=1;
```

private　shared

- private clause

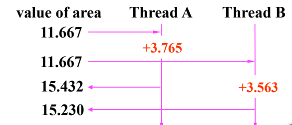
```
#pragma omp parallel for  private(j)
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[i][j] = min(a[i][j], a[i][k]+tmp[j]);
```

## 共享内存编程

- critical pragma

Protect a block of code that should be executed by a single thread at a time

```
double area, pi, x;
int i,n;
…
area = 0.0;
#pragma omp parallel for  private(x)
for (i=0; i<=n; i++) {
    x = (i+0.5)/n;
#pragma omp critical
    area += 4.0/(1+x*x);
}
pi = area/n;
```



## 共享内存编程

- reduction

```
double area, pi, x;
int i,n;
…
area = 0.0;
#pragma omp parallel for  private(x)  reduction(+:area)
for (i=0; i<=n; i++) {
    x = (i+0.5)/n;
#pragma omp critical
    area += 4.0/(1+x*x);
}
pi = area/n;
```

n = 100,000

| Threads | Execution time of program (sec) | |
|---|---|---|
| | critical pragma | reduction clause |
| 1 | 0.0780 | 0.0273 |
| 2 | 0.1510 | 0.0146 |
| 3 | 0.3400 | 0.0105 |
| 4 | 0.3608 | 0.0086 |
| 5 | 0.4710 | 0.0076 |

## 共享内存编程

- parallel pragma

```
#include <omp.h>

int main(int argc, char* argv[])
{
    int nthreads, tid;
    int nprocs;
    char buf[32];

#pragma omp parallel private(nthreads, tid)
    {
/*    Obtain and print thread id  */
      tid = omp_get_thread_num();
      printf("Hello World from OMP thread %d\n", tid);

/*    Only master thread does this  */
      if (tid==0) {
        nthreads = omp_get_num_threads();
        printf("Number of threads %d\n", nthreads);
      }
    }
    return 0;
}
```

```
Hello World from OMP thread 2
Hello World from OMP thread 0
Number of threads 4
Hello World from OMP thread 3
Hello World from OMP thread 1
```