# OpenMP并行编程

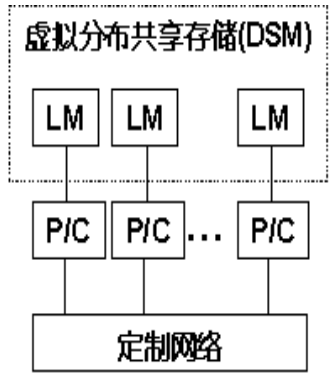
中科院计算机网络信息中心
超级计算中心

# 内容提要

- 并行机体系结构
- OpenMP编程简介
- OpenMP编程制导
- OpenMP库函数
- OpenMP环境变量
- OpenMP计算实例

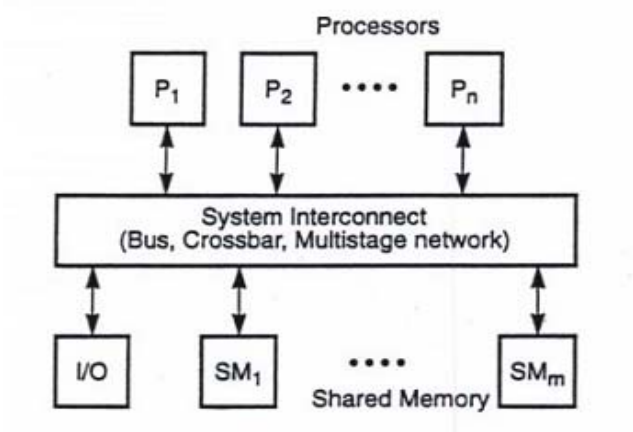
# 并行机体系结构及通信机制

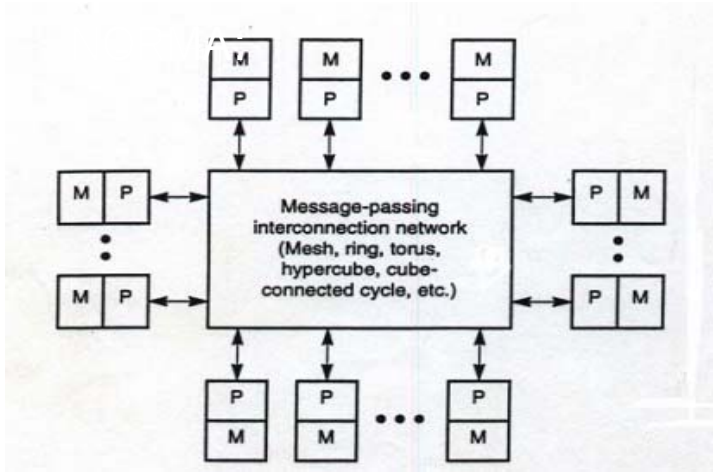
- SMP：共享内存并行机（ Shared Memory Processors ）。多个处理器通过交叉开关（Crossbar）或总线与共享内存互连。
  - 任意处理器可直接访问任意内存地址,且访问延迟、带宽、几率都是等价的;
  - 系统是对称的;
  - 单地址空间 、共享存储、UMA;
  - 并行编程方式: 通常采用OpenMP, 也可使用消息传递(MPI/PVM) 。
- DSM：分布共享存贮并行机（Distributed Shared Memory），由结点（一般是SMP系统）通过高速消息传递网络互连而成。存贮系统在物理上分布、逻辑上共享。各结点有自己独立的寻址空间。
  - 单地址空间 、分布共享
  - NUMA（ Nonuniform Memory Access ）
  - 与SMP的主要区别：DSM在物理上有分布在各个节点的局部内存从而形成一个共享的存储器;
  - 代表: SGI Origin 2000, Cray T3D

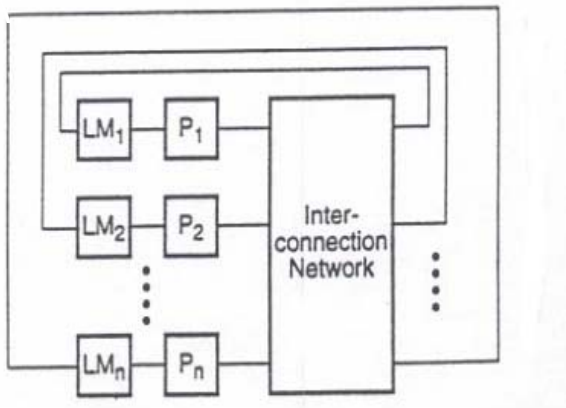


(d) DSM，逻辑上单一地址空间

# 访存模型




**多处理机（单地址空间共享存储器）**

UMA: Uniform Memory Access

NUMA: Non uniform Memory Access

**多计算机（多地址空间非共享存储器）**

NORMA:No-Remote Memory Access

# 最新的TOP500计算机

| Rank | Site | Computer |
|---|---|---|
| 1 | DOE/NNSA/LLNL<br>United States | **Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom<br>IBM |
| 2 | RIKEN Advanced Institute for Computational Science (AICS)<br>Japan | K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect<br>Fujitsu |
| 3 | DOE/SC/Argonne National Laboratory<br>United States | **Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom<br>IBM |
| 4 | Leibniz Rechenzentrum<br>Germany | **SuperMUC** - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR<br>IBM |
| 5 | National Supercomputing Center in Tianjin<br>China | **Tianhe-1A** - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050<br>NUDT |
| 6 | DOE/SC/Oak Ridge National Laboratory<br>United States | **Jaguar** - Cray XK6, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA 2090<br>Cray Inc. |
| 7 | CINECA<br>Italy | **Fermi** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom<br>IBM |
| 8 | Forschungszentrum Juelich (FZJ)<br>Germany | **JuQUEEN** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom<br>IBM |
| 9 | CEA/TGCC-GENCI<br>France | **Curie thin nodes** - Bullx B510, Xeon E5-2680 8C 2.700GHz, Infiniband QDR<br>Bull |
| 10 | National Supercomputing Centre in Shenzhen (NSCS)<br>China | **Nebulae** - Dawning TC3600 Blade System, Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050<br>Dawning |

- 国家超级计算天津中心的“天河”系统以理论峰值4701TFlops，Linpack实测2566TFlops，名列第五
  - 自主研发的FT-1000：1024个结点，含2048个CPU
  - Intel Xeon + NVIDIA GPU: 7168个，含14336个CPU、7168个GPU
  - 内存：229 TB 外存：2 PB
- 国家超级计算深圳中心的“星云”系统以理论峰值2984.3TFlops，Linpack实测1271TFlops，名列第十
  - 采用了自主设计的HPP体系结构、高效异构协同计算技术
  - 处理器是32nm工艺的六核至强X5650，并且采用了Nvidia Tesla C2050 GPU做协处理的用户编程环境

# 并行程序设计方法

- **隐式并行程序设计：**
  - 常用传统的语言编程成顺序源编码，把“并行”交给编译器实现自动并行
  - 程序的自动并行化是一个理想目标，存在难以克服的困难
  - 语言容易，编译器难
- **显式并行程序设计：**
  - 在用户程序中出现“并行”的调度语句
  - 显式并行是目前有效的并行程序设计方法。例如通过消息传递方式或多线程等
  - 语言难，编译器容易

# 并行程序设计模型

- 隐式并行（Implicit Parallel）
- 数据并行（Data Parallel）
- 共享变量（Shared Variable）
- 消息传递（Message Passing）

# 隐式并行(Implicit Parallel)

- **概况：**
  - 程序员用熟悉的串行语言编写相应的串行程序
  - 通过编译器和运行支持系统将串行程序自动转化为并行代码
- **特点：**
  - 语义简单
  - 可移植性好
  - 单线程，易于调试和验证正确性
  - 细粒度并行
  - 效率很低

# 数据并行（Data Parallel）

- **概况：**
  - SIMD的自然模型
- **特点：**
  - 并行操作于聚合数据结构（数组）
  - 松同步
  - 单一地址空间
  - 隐式交互作用
- **优点**:编程相对简单,串并行程序一致.
- **缺点**:程序的性能在很大程度上依赖于所用的编译系统及用户对编译系统的了解. 并行粒度局限于数据级并行,粒度较小.

# 共享变量(Shared Variable)

- 概况：
  - SMP，DSM的自然模型
- 特点：
  - 多线程：SPMD，MPMD
  - 松同步
  - 单一地址空间
  - 显式同步
  - 隐式数据分布
  - 隐式通信
- 典型代表：
  - OpenMP

# 消息传递(Message Passing)

- 概况：
  - MPP、COW的自然模型
- 特点：
  - 多进程异步并行
  - 多地址空间
  - 显式同步
  - 显式数据映射和负载分配
  - 显式通信
- 典型代表
  - MPI、PVM

# 并行编程模型标准

- 所有并行编程模型标准可分为以下三类:
  - 数据并行
    - HPF, Fortran90
    - 用于SMP, DSM
  - 共享编程
    - OpenMP
    - 用于SMP, DSM
  - 消息传递
    - MPI, PVM
    - 用于所有并行计算机
- 三者可混合使用:
  - 如对以SMP为节点的Cluster来说, 可以在节点间进行消息传递,在节点内进行共享变量编程.

# 基本并行化方法

- 相并行（Phase Parallel）
- 流水线并行（Pipeline Parallel）
- 主从并行（Master-Slave Parallel）
- 分治并行（Divide and Conquer Parallel）
- 工作池并行（Work Pool Parallel）

# 可扩展性

- 可扩展性(Scalability)：确定的应用背景下，计算系统(或算法或编程等)的性能随着处理器的数目的增加而按比例的提高的能力
- 总是将并行算法和体系结构一并考虑
  - 算法的可扩放性：该算法针对某一特定机器的可扩展性
  - 体系结构的可扩放性：该体系结构的机器的某一并行算法的可扩展性
- 一般情况下，增加处理器数，会增加额外开销和降低处理器利用率；所以对于一个特定的并行系统、并行算法或并行程序，它们能否有效的利用不断增加的处理器的能力应是受限的
- 目的
  - 确定某类问题用哪种并行算法与哪种并行体系结构结合。
  - 根据在小规模机器上的运行性能，预测在大规模机器上的性能
  - 对固定的问题规模，确定最有效的处理器数和加速比
  - 指导改进算法、体系结构，以利用可扩充的大量处理器

# 可扩放性评测标准

- 等效率度量标准:
  - 若问题规模w不变，随着处理器数P的增加会导致开销To随之增加，效率E下降。为了保持E不变，则在增加p的同时相应的增加问题规模W，以抵消由于p增加而导致的To的增加，从而保持效率不变
  - 随着系统规模的增加(处理器数目的增加)，测量增加多少运算量会保持效率不变
  - 增加越少表明可扩放性越好
  - E＝1/(1＋To/W)　To：额外开销时间之和

# 可扩放性评测标准

- 等速度度量标准
  - 系统规模增加时，若保持平均速度(每个处理器的速度)不变，每个处理器增加浮点操作的量
  - 速度常以每秒多少次浮点运算(Flops)来表示
- 等计算时间/通信开销比率度量标准
  - 系统规模增加时，保持计/通比不变所需要增加的问题规模
- 计算时间/通信开销比率
  - 并行计算时间与系统开销之比


OpenMP编程简介

# OpenMP简介

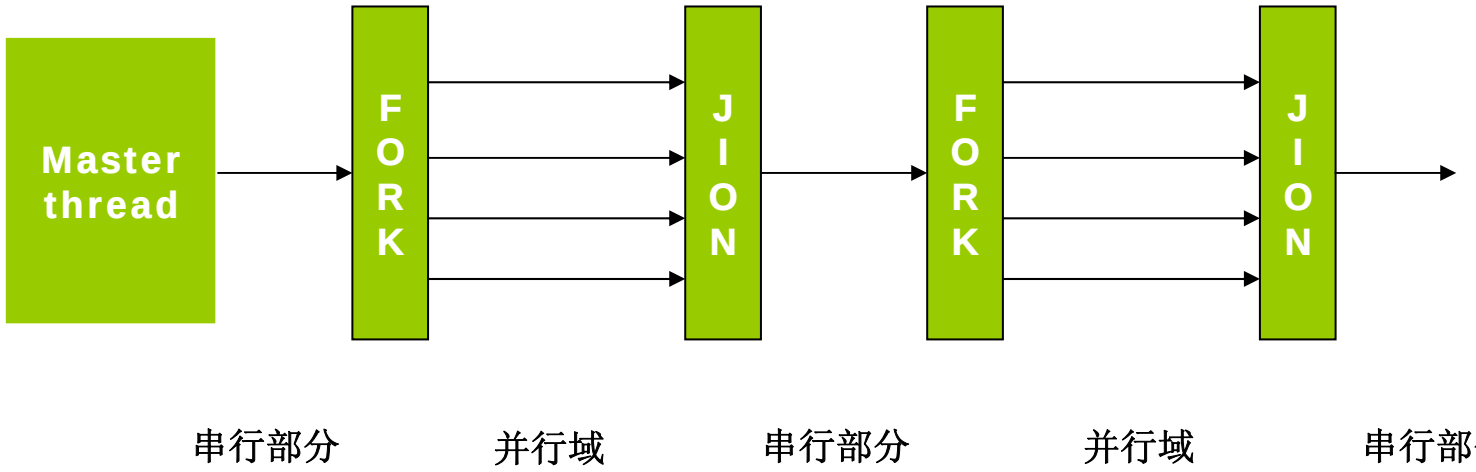
OpenMP是共享存储体系结构上的一个并行编程模型。适合于SMP共享内存多处理系统和多核处理器体系结构。

- 起源于ANSI X3H5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（Run-Time routines）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Windows等多种操作系统平台
- http://www.openmp.org/

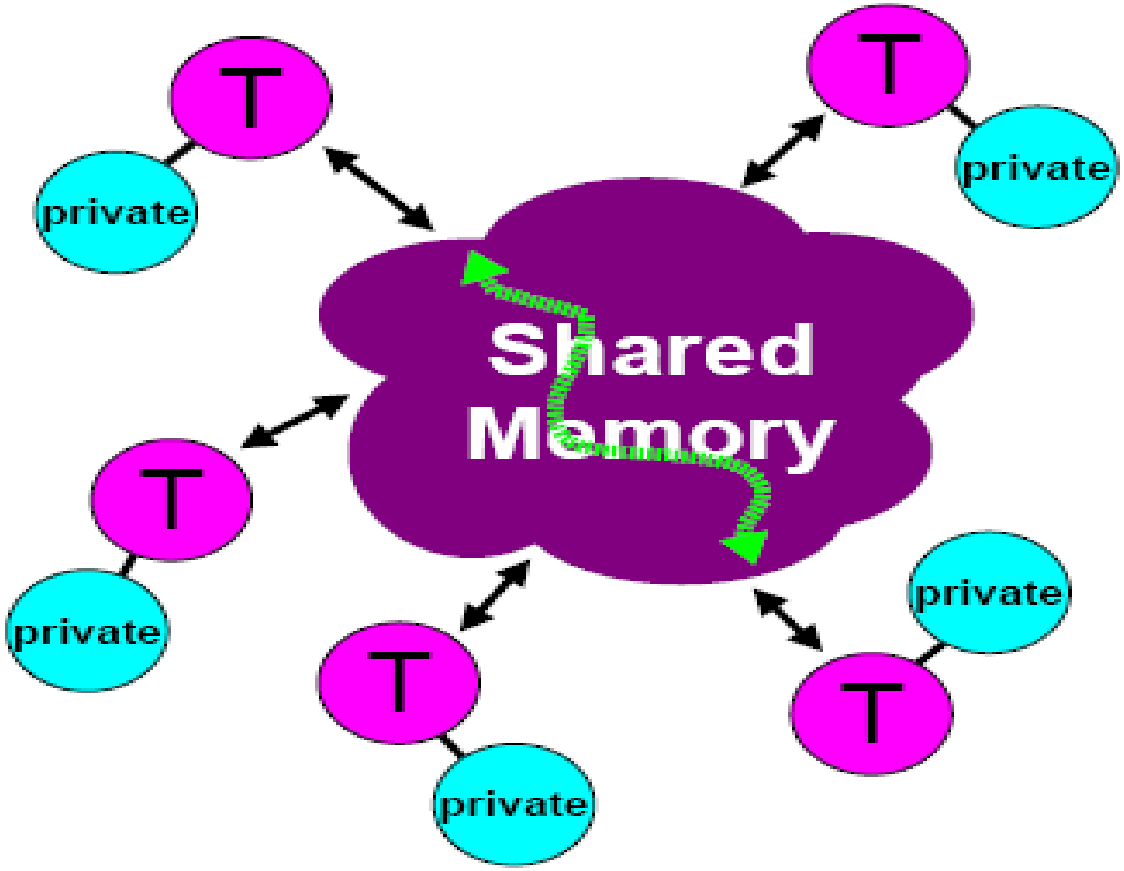
# OpenMP并行编程模式

- OpenMP是基于线程的并行编程模型。
- 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区域。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。

# OpenMP程序并行框架



# OpenMP 存储模型


T
private
T
private
Shared
Memory
T
private
private
T
T
private

# OpenMP存储模型

```
x = 2;
#pragma omp parallel num_threads(2) shared(x)
{
if (omp_get_thread_num() == 0) {
   x = 5;
} else {
   printf("1: Thread# %d: x = %d\n", omp_get_thread_num(),x );
}

#pragma omp barrier
if (omp_get_thread_num() == 0) {
  printf("2: Thread# %d: x = %d\n", omp_get_thread_num(),x );
} else {
  printf("3: Thread# %d: x = %d\n", omp_get_thread_num(),x );
}
}
```

# 支持条件编译

```
int main()
{
# ifdef _OPENMP
  printf("Compiled by an OpenMP-
  compliant implementation.\n");
# endif
  return 0;
}
```

# 简单的”Hello, world”OpenMP并行程序

```
/* 用OpenMP/C编写Hello World代码段 */
  #include <omp.h>
 int main(int argc, char *argv[])
 {
    int nthreads,tid;
    char buf[32];
    /*  Fork a team of threads      */
    #pragma omp parallel private(nthreads,tid)
    {
       tid = omp_get_thread_num(); /* Obtain and print thread id  */
       printf("Hello, world from OpenMP thread %d\n", tid);
        if (tid == 0) /*Only master thread does this  */
       {
        nthreads = omp_get_num_threads();
        printf(" Number of threads %d\n",nthreads);
       }
    }
    return 0;
 }
```

- 编译

icc -openmp – o HelloWorld HelloWorld.c

- 执行

./HelloWorld

- 运行结果：

Hello World from OpenMP thread 2

Hello World from OpenMP thread 0

Number of threads 4

Hello World from OpenMP thread 3

Hello World from OpenMP thread 1

# OpenMP程序结构

- 基于Fortran语言的OpenMP程序结构

```
PROGRAM  PROG_NAME
   INTEGER VAR1, VAR2 ,VAR3
    …….
   !$OMP PARALLEL PRIVATE(VAR1, VAR2)
SHARED(VAR3)
    …….
  !$OMP END PARALLEL
   ……
END
```

- 基于C/C++语言的OpenMP程序结构

```
#include<omp.h>
main(){
   int var1, var2, var3;
   ……..
   #pragma omp parallel private(var1, var2) shared(var 3)
   {
      ……….
    }
    ……….
}
```

OpenMP的制导指令有以下一些：

parallel 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

for 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。

parallel for parallel 和 for语句的结合，也是用在一个for循环之前，表示for循环的代码将被多个线程并行执行。

sections 用在可能会被并行执行的代码段之前

parallel sections parallel和sections两个语句的结合

critical 用在一段代码临界区之前

single 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

barrier，用于并行区内代码的线程同步，所有线程执行到barrier时要停止直到所有线程都执行到barrier时才继续往下执行。

Atomic 用于指定一块内存区域被制动更新

Master 用于指定一段代码块由主线程执行

Ordered 用于指定并行区域的循环按顺序执行

threadprivate 用于指定一个变量是线程私有

OpenMP除上述指令外，还有一些库函数，下面列出几个常用的库函数：

omp_get_num_procs, 返回运行本线程的多处理机的处理器个数。
omp_get_num_threads, 返回当前并行区域中的活动线程个数。
omp_get_thread_num, 返回线程号
omp_set_num_threads, 设置并行执行代码的线程个数
omp_init_lock, 初始化一个简单锁
omp_set_lock， 上锁操作
omp_unset_lock， 解锁操作，要和omp_set_lock函数配对使用。
omp_destroy_lock， omp_init_lock函数的配对操作函数，关闭一个锁

OpenMP的子句有以下一些：

private, 指定每个线程都有它自己的变量私有副本。

firstprivate，指定每个线程都有它自己的变量私有副本，并且变量要被继承主线程中的初值。

lastprivate，主要是用来指定将线程中的私有变量的值在并行处理结束后复制回主线程中的对应变量。

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运算。

nowait，忽略指定中暗含的等待

num_threads，指定线程的个数

schedule，指定如何调度for循环迭代

shared，指定一个或多个变量为多个线程间的共享变量

ordered，用来指定for循环的执行要按顺序执行

copyprivate，用于single指导中的指定变量广播到并行区中其它线程

copyin，用来指定一个threadprivate的变量的值要用主线程的值进行初始化。

default，用来指定并行处理区域内的变量的使用方式，缺省是shared


OpenMP编译制导

# 编译制导

- 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来实现。
- 编译制导是对程序设计语言的扩展。
- 通过对串行程序添加制导语句实现并行化。
- 支持并行区域、工作共享、同步等。
- 支持数据的共享和私有化。
- 支持增量并行。

# 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（private, shared, reduction, copyin等）

- 格式：制导标识符 制导名称 [Clause，]

# 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:

    **!$OMP** (or **C$OMP** or ***$OMP**)
  - C/C++:

    **#pragma omp**

# 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

**Fortran：**

```
！$OMP PARALLEL [clauses]
        BLOCK
！$OMP END PARALLEL
```

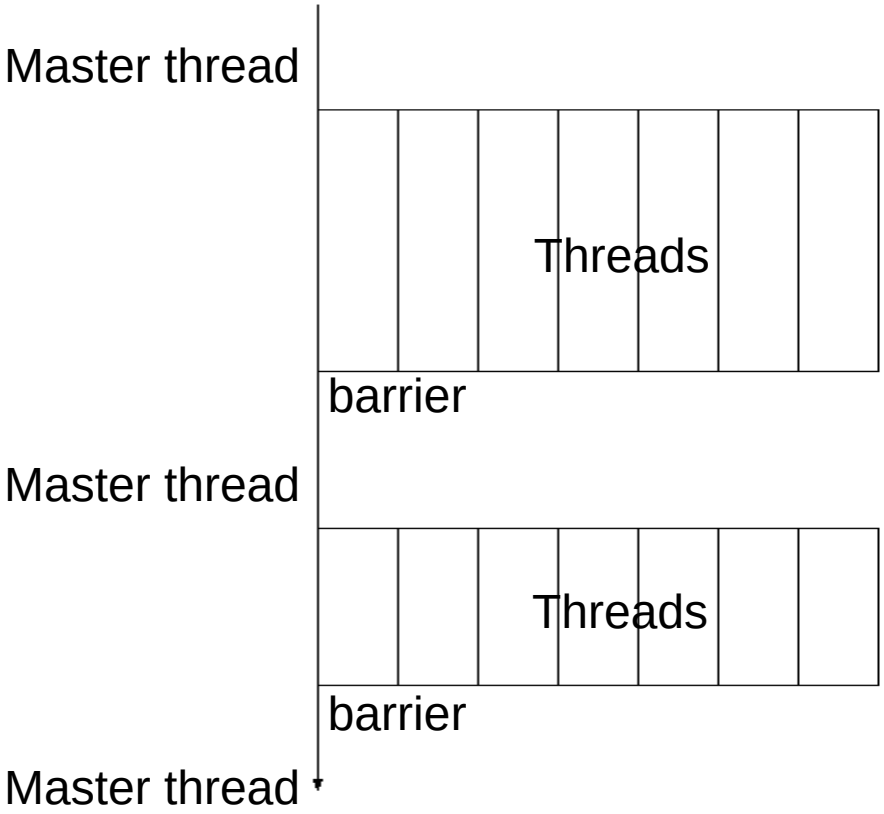
**C/C++：**

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

# 说明

- 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。

# 并行域结构：例图




```
      PROGRAM FRED
        .
        .
!$OMP  PARALLEL
        .
        .
        .
        .
        .
        .
        .
        .
!$OMP  END PARALLEL
        .
        .
        .
        .
!$OMP  PARALLEL
        .
        .
        .
        .
!$OMP  END PARALLEL
        .
        .
        .
```

# if子句

```
#pragma omp parallel if (n > threshold) \
        shared(n,x,y) private(i)
  {
    #pragma omp for
     for (i=0; i<n; i++)
        x[i] += y[i];
  } /*-- End of parallel region --*/
```

# shared 和privated子句

- 并行域内的变量，可以通过子句说明为公有或私有;
- 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++:**

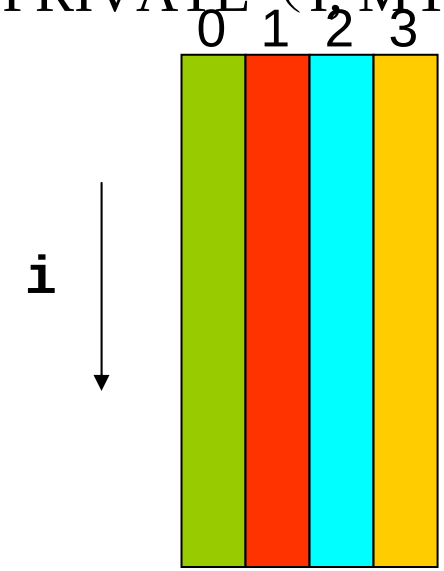
shared(list)

private(list)

default(shared|private|none)

- 例：每个线程初始共享数组的一列

```
！$OMP PARALLEL DEFAULT(NONE）, PRIVATE（I, MYID),
！$OMP & SHARED(a, n)
    myid=omp_get_thread_num()+1
     do i=1, n
        a(i, myid)=1.0
       end do
！$OMP END PARALLEL
```




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。默认时为公有。

# 并行域结构：reduction子句

- 归约用来从相关的操作（+,*,max或min等）中产生一个单一值；
- OpenMP提供了reduction子句。

Fortran：REDUCTION(op:list)

C/C++： reduction(op:list)

- 例子：将一组数值归约求和

```
sum=0;
$OMP PARALLEL REDUCTION(+: sum), PRIVATE(i,myid)
  myid=omp_get_thread_num()+1
  do i= 1, n
    sum=sum+a(i, myid)
  end do
$OMP END PARALLEL
```

说明：

- 在reduction子句中，编译器为每个线程创建变量sum的私有副本。当循环完成后，将这些值加在一起并把结果放到原始的变量sum中；
- Reduction中的op操作必须满足算术结合律和交换律。

# 计算Pi值

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \approx \frac{1}{n}\sum_{i=1}^{n} \frac{4}{i + \left(\frac{i+0.5}{n}\right)^2}$$

```
/* Seriel Code */
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\n Running time \n", pi, end_time-start_time);

}
```

# 并行域并行（SPMD并行模式）

```
include <omp.h>
 static long num_steps = 100000;   double step;
 #define NUM_THREADS     4
 void main ()
 {    int i ;  double pi, sum[NUM_THREADS] , start_time, end_time ;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS)
     start_time=omp_get_wtime();
    #pragma omp parallel
    {  int id; double x;
        id = omp_get_thread_num();
       for (i=id, sum[id]=0.0;i< num_steps; i=i+NUM_THREADS){
          x = (i+0.5)*step;   sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0;i<NUM_THREADS;i++) pi += sum[i] * step;
    end_time=omp_get_wtime();
    printf("Pi=%f\n Running time \n", pi, end_time-start_time);
 }
```

# 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
- 其它制导

# 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：**

```
！$OMP DO[clauses]
      DO 循环
！$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FIRSTPRIVARE子句；
- 循环变量是私有的。

- 可以将并行域和DO/for制导结合成单一的简单形式

**Fortran：**

```
！$OMP PARALLEL [clauses]
  ！$OMP  DO[clauses]
        循环体
  ！$OMP END DO
！$OMP END  PARALLEL
```

**合并后形式：**

```
！$OMP PARALLEL DO[clauses]
        循环体
！$OMP END  PARALLEL DO
```

**同样地，C/C++：合并后形式**

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

# 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 DO

- 子句说明
  - schedule (STATIC [, chunksize]）：

省略chunksize，迭代空间被划分成（近似）相同大小的区域，每个线程被分配一个区域；

如果chunksize被指明，迭代空间被划分为chunksize大小，然后被轮转的分配给各个线程

  - ✓ 例如：假如线程数为4

schudule(static)

T0 T1 T2 T3

1 40

schudule(static, 4)

T0 T1 T2 T3 T0 T1 T2 T3 T0 T1

1 40

# 并行DO/for循环制导：调度子句SCHEDULE

- schedule (**DYNAMIC** [, chunksize]）：
  - 划分迭代空间为chunksize大小的区间，然后基于先来先服务方式分配给各线程；
  - 当省略chunksize时，其默认值为1。
- schedule (**GUIDED** [, chunksize]）
  - 类似于DYNAMIC调度，但区间开始大，然后迭代区间越来越少，循环区间的划分是基于类似下列公式完成的（不同的编译系统可能不同）：

$$S_k = \left\lceil \frac{R_k}{2N} \right\rceil$$

其中$N$是线程个数，$S_k$表示第k块的大小，$R_k$是剩余下未被调度的循环迭代次数。

  - chunksize说明最小的区间大小。当省略chunksize时，其默认值为1。
- schedule (**RUNTIME**）
  - 调度选择延迟到运行时，调度方式取决于环境变量OMP_SCHEDULE的值，例如：

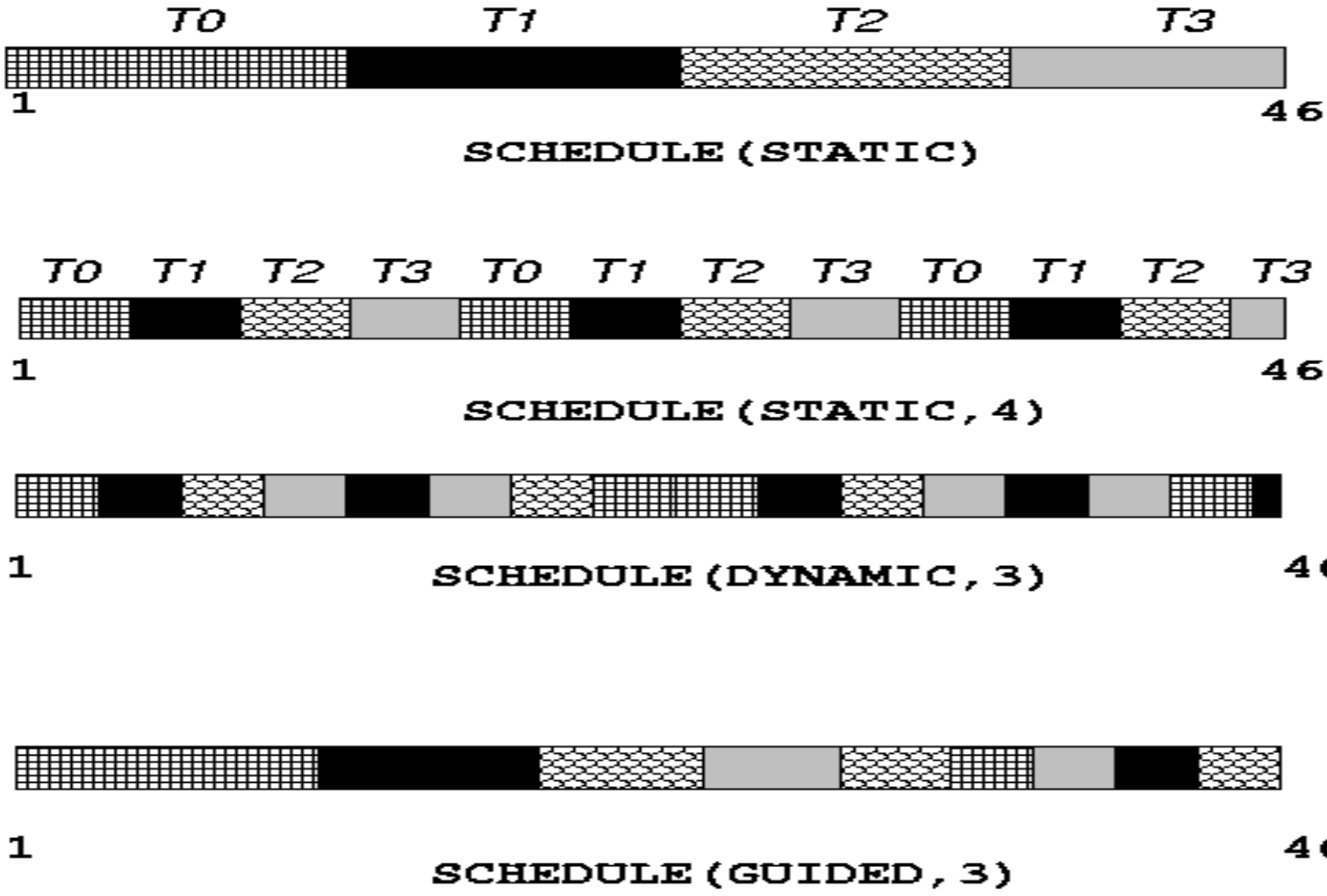
```
export OMP_SCHEDULE="DYNAMIC, 4";
```

  - 使用 RUNTIME时，指明chunksize是非法的；

# 使用for循环制导计算pi值

```
#include <omp.h>
#define NUM_THREADS 4
static long num_steps = 100000;  double step;
int main ()
{      int i,id; double x, pi, sum[NUM_THREADS];
       step = 1.0/(double) num_steps;
       omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private(x, id)
    {
       id = omp_get_thread_num();  sum[id] = 0;
    #pragma omp for
       for (i=0;i< num_steps; i++){
              x = (i+0.5)*step;
              sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0;i<NUM_THREADS; i++) pi += sum[i] * step;
    return 0;
}
```

# 调度子句SCHEDULE例图



# 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);
 }
```

# 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
- Fortran:

```
！$OMP SECTIONS[clauses]
  [！$OMP SECTION]
     block
  [！$OMP SECTION
          block ]
     ……
！$OMP END SECTIONS
```

- C/C++:

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

- 说明：
  - 各结构化块在各线程间并行执行：
    结构化块的数量少于线程个数？？；
    结构化块的数量大于线程个数？？。
  - sections制导可以带有PRIVATE、 FIRSTPRIVATE和其它子句；
  - 每个section必须包含一个结构体。
- 将并行域和SECTIONS制导结合成单一的简单形式：

**Fortran:**

```
$OMP PARALLEL SECTIONS[clauses]
        …….
$OMP END PARALLEL SECTIONS
```

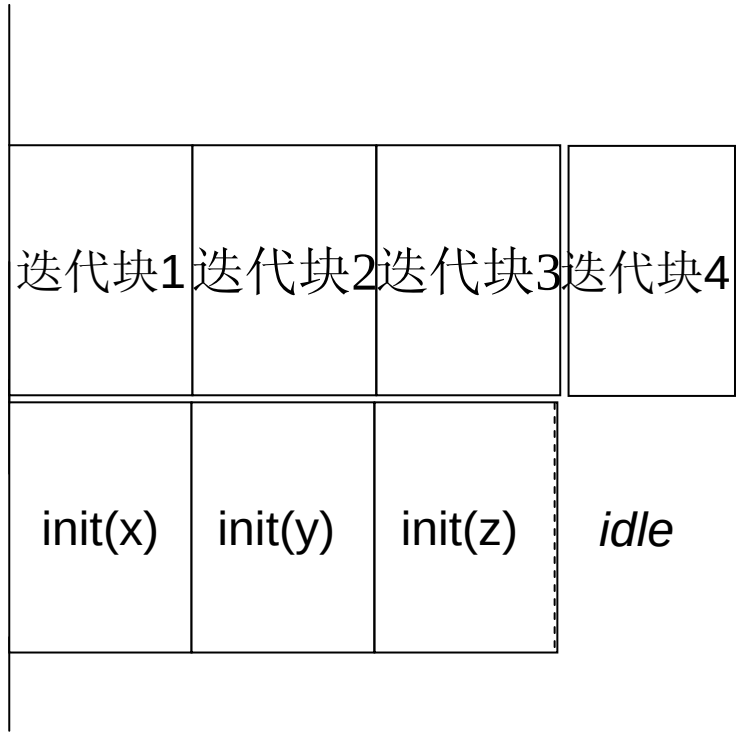
**C/C++:**

```
$pragma parallel sections[clauses]
        …….
$pragma end parallel sections
```

# 并行SECTIONS制导:例句

```
! $OMP PARALLEL
  ! $OMP DO
    循环体
  ! $OMP END DO
  ! $OMP SECTIONS
    ! $OMP SECTION
       call init(x)
    ! $OMP SECTION
       call init(y)
    ! $OMP SECTION
       call init(z)
  ! $OMP END SECTIONS
! $OMP END PARALLEL
```

- 假如有4个线程



# SINGLE制导

**SINGLE制导：**

- **Fortran:**

```
!OMP SINGLE [clauses]
    block
!OMP END SINGLE
```

- **C/C++:**

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

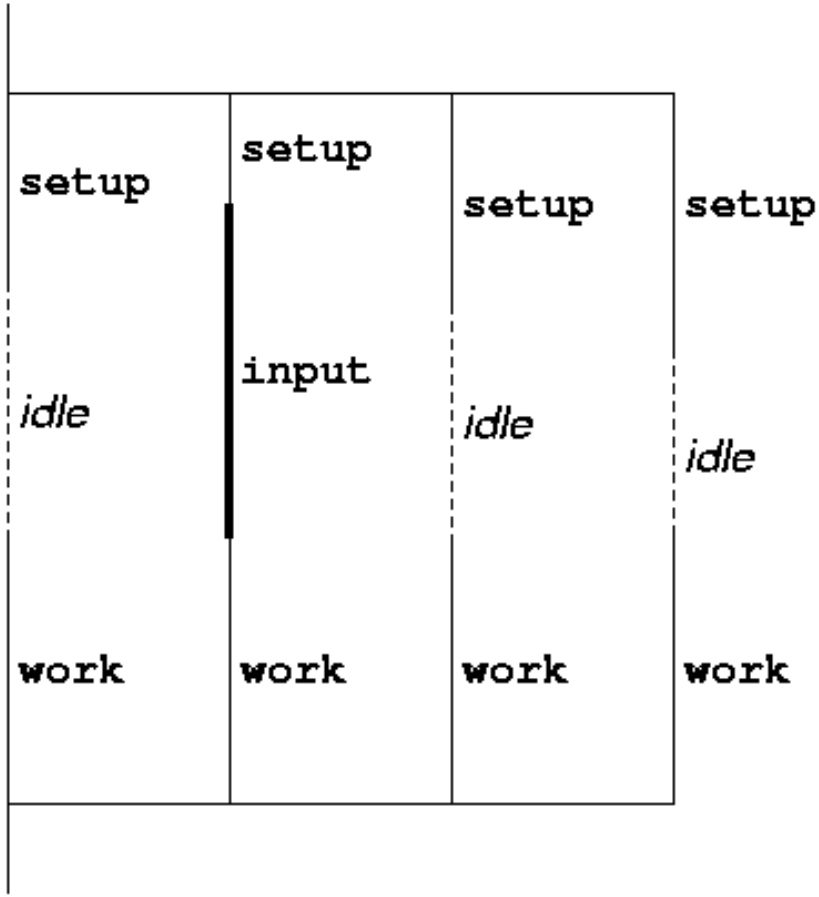
- 说明：
  - 结构体代码仅由一个线程执行；并由首先执行到该代码的线程执行；其它线程等待直至该结构块被执行完

- 例子

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

# SINGLE制导：例图

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



# ordered制导

- 指定循环按照迭代顺序执行
- 一个迭代只能执行一个ordered制导区域

```
void work(int k){
  #pragma omp ordered
    printf(" %d\n", k);
}

void a24(int lb, int ub, int stride){
int i;
#pragma omp parallel for ordered schedule(dynamic)
for (i=lb; i<ub; i+=stride)
  work(i);
}

int main(){
  a24(0, 100, 5);
  return 0;
}
```

# MASTER制导

- MASTER制导
- Fortran:

```
!OMP MASTER [clauses]
    block
!OMP END MASTER
```

- C/C++:

```
#pragma omp master [clauses]
    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；

# BARRIER制导

- BARRIER是OpenMP用于线程同步的一种方法
- **Fortran:**

!$ OMP BARRIER

- **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。

- 例子

```
! $OMP PARALLEL PRIVATE(i, myid, neighb)
   myid=omp_get_thread_num()
   neighb=myid-1
  if (myid .eq. 0) neighb=omp_get_num_threads()-1
    ……
  a(myid)=a(myid)*3.5
  ! $ OMP BARRIER
    b(myid)=a(neighb)+c
    ……
 ! $ OMP END PARALLEL
```

# 使用带reduction子句的for循环制导

```
#include <omp.h>
#define NUM_THREADS 4
static long num_steps = 100000;        double step;
int main ()
{     int i,id; double x, pi, sum, start_time, end_time;
      step = 1.0/(double) num_steps;
      omp_set_num_threads(NUM_THREADS) ;
      start_time=omp_get_wtime();
     #pragma omp parallel private(x, id)
     {  id = omp_get_thread_num();//       sum[id] = 0;
     #pragma omp for private(x) 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*step;
   return 0;
}
```

# NOWAIT 子句

- Nowait子句可以除去隐藏在循环、SECTIONS或并行区后的同步
- **Fortran：**

```
！OMP DO
      do loop
！ OMP END DO NOWAIT
```

- **C/C++：**

```
#pragma omp for nowait
    for loop
```

✓ SECTIONS制导和SINGLE制导有类似形式

- **说明：**

➢ 使用NOWAIT时要特别小心，有可能导致不可确定的bug；

- 在有些地方使用NOWAIT可能是好的代码形式，并且显式的使用BARRIERS

- 例子：两个循环间没有依赖性

```
！ $OMP PARALLEL
    ！ $ OMP DO
          do j=1, n
              …….
    ! $ OMP END DO NOWAIT

    ！ $ OMP DO
          do j=1, n
              …….
    ! $ OMP END DO NOWAIT
！ OMP END PARALLEL
```

```
int i;
#pragma omp parallel
{
#pragma omp for schedule(static) nowait
for (i=0; i<n; i++)
   c[i] = (a[i] + b[i]) / 2.0;
#pragma omp for schedule(static) nowait
for (i=0; i<n; i++)
   z[i] = sqrt(c[i]);
#pragma omp for schedule(static) nowait
for (i=1; i<=n; i++)
   y[i] = z[i-1] + a[i];
}
```

# 保存共享变量：CRITICAL制导

- CRITICAL（临界段）可以保护共享变量的更新，避免数据竞争，制导内的代码段仅能有一个线程执行
- Fortran：

!$OMP CRITICAL *[*(name)*]*

*block*

!$OMP END CRITICAL *[*(name)*]*

- C/C++:

#pragma omp critical *[*(name)*]*

*structure block*

- 说明
- Critical制导在某一时刻仅能被一个线程执行；

- Critical制导可用来保护对共享变量的修改；
- 在Fortran中，前后两个name必须一致；
- 如果name被省略,一个空（*null*）的name被假定。
- 例：下面使用了一个未命名的临界段。

```
#pragma omp critical
{
  if(max<new_value) max=new_value
}
```

- 下面使用了一个命名的临界段。

```
#pragma omp critical (maxvalue)
{
  if(max<new_value) max=new_value
}
```

- 使用命名临界段时，应用程序可以有多个临界段
- 线程将会在critical 临界区入口等待,直到没有其它线程执行相同名字的临界区

# 通过private子句和critical 制导计算pi值

```
#include <omp.h>
static long num_steps = 100000;  double step;
#define NUM_THREADS 4
void main ()
{     int i, id;   double  x, sum, pi=0.0,start_time, end_time;
      step = 1.0/(double) num_steps;
      omp_set_num_threads(NUM_THREADS)
      start_time=omp_get_wtime();
   #pragma omp parallel private (x, sum)
   {    id = omp_get_thread_num();
        for (i=id,sum=0.0;i< num_steps;i=i+NUM_THREADS){
          x = (i+0.5)*step; sum += 4.0/(1.0+x*x);
        }
   #pragma omp critical
       pi += sum;
   }
    end_time=omp_get_wtime();
}
```

# 共享变量：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 op expr、 x=expr op x 、x=intr(x, expr)或x=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 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；而使用临界制导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。

```
#pragma omp parallel for shared(x, y,
  index, n)
for (i=0; i<n; i++) {
#pragma omp atomic
    x[index[i]] += work1(i);
   y[i] += work2(i);
}
```

# LOCK 例程

- 一个锁是一个特殊的变量，它被一个线程设定，而别的线程仅能在设定锁的线程解除锁后才能设定锁
- **Fortran:**

Subroutine OMP_INIT_LOCK(VAR)
Subroutine OMP_SET_LOCK(VAR)
LOGICAL FUNCTION OMP_TEST_LOCK(VAR)
Subroutine OMP_UNSET_LOCK(VAR)
Subroutine OMP_DESTROY_LOCK(VAR)
其中变量是一个作为地址的整数

- **C/C++：**

```
#include<OMP.h>
void omp_init_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
void omp_detroy_lock(omp_lock_t *lock);
```

- 例子

```
 call omp_init_lock(ilock)
!$OMP PARALLEL SHARED(ilock)
    ...
    do while ( .not. omp_test_lock(ilock))
      call do_something_else()
    end do
    call work()
    call omp_unset_lock(ilock)
    ...
!$OMP END PARALLEL
call omp_detroy_lock(ilock)
```

说明：

- 锁在使用前需要进行初始化；不再使用时要解锁。

# FLUSH制导

- FLUSH语句是用来确保执行中存储器中的数据一致的同步点。保证一个变量从内存中的读/写
- Fortran:

!$OPM FLUSH*[*(list)*]*

- C/C++:

#prgma omp flush *[*(list)*]*

# FLUSH制导

- Barrier
- Parallel、critical、ordered的入口和退出
- omp_set_lock和omp_unset_lock
- omp_test_lock, omp_set_nest_lock, omp_unset_nest_lock和 omp_test_nest_lock
- Atomic中的变量

```
flag=0;
#pragma omp parallel sections
{
   #pragma omp section
   {
      fill_rand(N,A);
   #pragma omp flush
      flag=1;
   #pragma omp flush(flag)
   }

   #pragma omp section
   {
   #pragma omp flush(flag)
     while(flag!=1){
        #prgma omp flush(flag)
     }
   #pragma omp flush
     sum = Sum_array(N,A);
   }
}
```

# firstprivate子句

- 该子句使并行域内私有变量的初始值通过master线程的值初始化。

格式：

firstprivate(变量列表)

例：

```
x[0]=10;
for(i=0;i<n;i++){
   for(j=1;i<4;j++)
    x[j]=i+x[j-1];
  y[i]=x[1]-x[3];
}
```

```
x[0]=10;
#pragma omp parallel for private(j) firstprivate(x)
for(i=0;i<n;i++){
   for(j=1;i<4;j++)
    x[j]=i+x[j-1];
  y[i]=x[1]-x[3];
}
```

# lastprivate子句

- 该子句将使得DO/for循环制导内私有变量的最后值赋值给master线程的变量。

```
for(i=0;i<n;i++){
  x[0]=1.0;
  for(j=1;i<4;j++)
    x[j]=x[j-1]*(i-1);
  sum_of_power=x[0]+x[1]+X[2]+X[3];
}
n_cube=x[3];
```

```
#pragma omp parallel for private(j) lastprivate(x)
for(i=0;i<n;i++){
  x[0]=1.0;
  for(j=1;i<4;j++)
    x[j]=x[j-1]*(i-1);
  sum_of_power=x[0]+x[1]+X[2]+X[3];
}
n_cube=x[3];
```

# copyin子句

将主线程的threadprivate变量广播给其它线程的threadprivate变量

```
#include <omp.h>
float* work;
int size;
float tol;
#pragma omp threadprivate(work,size,tol)
void build()
{
   int i;
   work = (float*)malloc( sizeof(float)*size );
   for( i = 0; i < size; ++i ) work[i] = tol;
}

void a36( float t, int n )
{
   tol = t;
   size = n;
 #pragma omp parallel copyin(tol,size)
{
   build();
}
}
```

# copyprivate子句

```
#include <omp.h>
  float x, y;
#pragma omp threadprivate(x, y)
void init(float a, float b ) {
#pragma omp single copyprivate(a,b,x,y)
 {
   scanf("%f %f %f %f", &a, &b, &x, &y);
 }
}
```

# collapse

```
!$omp do private(j,k) collapse(2) lastprivate(jlast,
   klast)
do k = 1,2
  do j = 1,3
    jlast=j
    klast=k
  enddo
enddo
!$omp end do

!$omp single
   print *, klast, jlast
!$omp end single
```

```
!$omp parallel num_threads(2)
!$omp do collapse(2) ordered private(j,k)
   schedule(static,3)
do k = 1,3
  do j = 1,2
!$omp ordered
      print *, omp_get_thread_num(), k, j
!$omp end ordered
      call work(a,j,k)
  enddo
enddo
!$omp end do
!$omp end parallel
```

0 1 1

0 1 2

0 2 1

1 2 2

1 3 1

1 3 2

必须有
collapse(2)

# default子句

- c/c++
  - default(shared | none)
- fortran
  - default(private | firstprivate | shared | none)

```
#include <omp.h>
int x, y, z[1000];
#pragma omp threadprivate(x)

void a28(int a) {
  const int c = 1;
  int i = 0;

  #pragma omp parallel default(none) private(a) shared(z)
  {
     int j = omp_get_num_threads();
          /* O.K.   - j is declared within parallel region */
     a = z[j];    /* O.K.   - a is listed in private clause */
                  /*        - z is listed in shared clause */
     x = c;       /* O.K.   - x is threadprivate */
                  /*        - c has const-qualified type */
     z[i] = y;    /* Error - cannot reference i or y here */

     #pragma omp for firstprivate(y)
     for (i=0; i<10 ; i++) {
       z[i] = y; /* O.K. - i is the loop iteration variable */
                 /* - y is listed in firstprivate clause */
     }

     z[i] = y;    /* Error - cannot reference i or y here */
  }
}
```


Run-Time routines

# 运行库函数

- 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:**

```
interger 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:**

```
Interger function OMP_GET_THREAD_NUM ()
```

- **C/C++:**

```
#include<omp.h>
  int omp_get_thread_num()
```

# 设定执行线程的数量

- 使用运行库函数：

Fortran:

```
routine  OMP_SET_NUM_THREADS ( )
```

C/C++:

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

- 在制导语句中通过 NUM_THREADS设定。

- 通过环境变量OMP_NUM_THREADS 设定。

# 时间函数

- return current wall clock time (relative to arbitrary origin)

**Fortran:**

**DOUBLE PRECISION FUNCTION OMP_GET_WTIME()**

**C/C++:**

**double omp_get_wtime(void);**

- return clock precision

**Fortran:**

**DOUBLE PRECISION FUNCTION OMP_GET_WTICK()**

**C/C++:**

**double omp_get_wtick(void);**


OpenMP环境变量

# 环境变量

- OpenMP提供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=4

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=TRUE

# NUM_THREADS子句

- 在 OpenMP (Fortran 、 C/C++) 提供了 NUM_THREADS 子句设定线程数。
- 例子

```
!$OMP PARALLEL DO NUM_THREADS(4)
      DO J = 1,N
        A(I,J) = B(I,J)
 !$OMP END DO
```

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 )

# 设定线程个数子句:num_threads

- num_threads子句用来指定并行域内使用线程的个数，随后的其它并行域不受此影响。

例：

```
#include"omp.h"
#include"stdio.h“
main()
{
  omp_set_num_threads(4);

   #pragma omp parallel  num_threads(2)

   printf(“my thead number is %d\n",omp_get_thread_num());
}
```

- num_threads子句的优先权高于库例程omp_set_num_threads和环境变量NMP_NUM_THREADS。

# 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DO/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；
- 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多处理器上出色运行的程序是不同的；
- 可以借助一些工具，例Intel Vtune™性能分析工具，其提供了一个Intel线程监测器。

……

# 实例1：蒙特卡罗算法

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
int main()
{
  long long max=10000000;
   long long i,count=0;
  double x,y,z,bulk,start_time,end_time;
  time_t t;
  start_time=clock();
   // 产生以当前时间开始的随机种子
  srand((unsigned) time(&t));
  for(i=0;i<max;i++)
  {  x=rand();    x=x/32767;
     y=rand();    y=y/32767;
     z=rand();    z=z/32767;
     if((x*x+y*y+z*z)<=1)
            count++;
   }
  bulk=8*(double(count)/max);
  end_time= clock();
  printf(”Sphere bulk is %f \n“, bulk);
  printf(“Running time is %f \n“,
          end_time-start_time);
  return 0;
}
```

# OpenMP Parallel for with a reduction

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
int main()
{
  long long max=10000000;
  long long i,count=0;
 double x,y,z,bulk,start_time,end_time;
 time_t t;
start_time=omp_get_wtime();
//产生以当前时间开始的随机种子
srand((unsigned) time(&t));
#pragma omp parallel for private(x,y,z)
   reduction(+:count)
for(i=0;i<max;i++)
{x=rand();       x=x/32767;
  y=rand();       y=y/32767;
  z=rand();       z=z/32767;
  if((x*x+y*y+z*z)<=1)
          count++;
}
bulk=8*(double(count)/max);
end_time=omp_get_wtime();
printf(”Sphere bulk is %f \n“, bulk);
printf(“Running time is %f \n“,
                end_time-start_time);
return 0;
}
```

# 循环依赖（Loop Dependency）及其删除方法

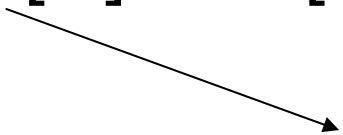
- 了解循环依赖的好处
  - OpenMP并行
  - SIMD: Vectorization(MMX, SSE, SSE2)
  - ILP: Instruction level parallelism
- 循环依赖包括
  - 流依赖（Flow Dependency）
  - 反依赖（Anti-Dependency）
  - 写依赖（Output Dependency）
  - 迭代内依赖（Intra-Iteration Dependency）

# 流依赖（Flow Dependency）

- 跨迭代的写后读

```
for(j=1; j<MAX; j++){
    A[j] = A[j-1];
}
```

```
A[1] = A[0];
```



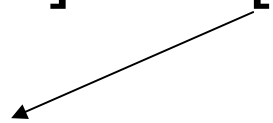
```
A[2] = A[1];
```

# 反依赖（Anti-Dependency）

- 跨迭代的读后写

```
for (j=1; j<MAX; j++){
    A[j] = A[j+1];
}
```

```
A[1] = A[2];
```



```
A[2] = A[3];
```

# 写依赖（Output Dependency）

- 跨迭代的写相关

```
for (j=1; j<MAX; j++){
    A[j] = B[j];
    A[j+1] = C[j];
}
```

```
A[1] = B[1];
A[2] = C[1];
```



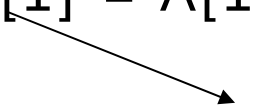
```
A[2] = B[2];
A[3] = C[2];
```

# 迭代内依赖（Intra-Iteration Dependency）

- 一个迭代内的相关
- 会破坏ILP
- 可能被编译器自动删除

```
k = 1;
for (j=1 ; j<MAX; j++){
    A[j] = A[j] + 1;
    B[k] = A[k] + 1;
    k = k + 2;
}
```

```
A[1] = A[1] + 1;
```

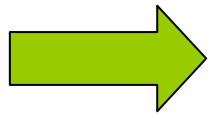


```
B[1] = A[1] + 1;
```

# 消除依赖

- 最好的选择
- OpenMP并行化的先决条件
- 并不是所有的依赖都能被消除的

```
for (j=1; j<MAX; j++){
    A[j] = A[j-1] + 1;
}
```

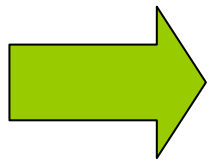


```
for (j=1; j<MAX; j++){
    A[j] = A[0] + j;
}
```

# 归纳变量删除

- 一般是循环中，其后续值形成一个算术级数的变量.除了循环控制变量以外的一些变量,也遵循与循环控制变量类似的模式.
- 用循环控制变量(j)来替换归纳变量

```
i1 = 0;
i2 = 0;
for (j=0; j<MAX; j++){
    i1 = i1 + 1;
    B[i1] = ...;
    i2 = i2 + j;
    A[i2] = ...;
}
```



```
for (j=0; j<MAX; j++){
    B[j] = ...;
    A[(j*j + j)/2] = ...;
}
```

# Reduction 变量

- 通过结合操作来收集数组的数据并存入标量

```
for(j=0; j<MAX; j++)
    sum = sum + c[j];
```

- 1.利用结合操作来计算部分和,或者局部最大值到私有空间
- 2.合并得到的部分结果到共享空间,此时需要注意同步

# 针对Reduction的优化

```
//Block Sum
for(p=0; p<NP; p++){
    begin=p*MAX/NP;
    end=((p+1)*MAX/NP)-1;
    for(j=begin;j<end;j++)
        prv_sum[p] += c[j];
}

//加锁
for(p=0;p<NP;p++)
    sum += prv_sum[p];
```

```
//Vertical Sum
for(j=0; j<MAX; j+=VS){
    sum[0] += c[j];
    sum[1] += c[j+1];
    ...
}

for(v=0; v<VS; v++)
    sum += sum[v];
```

OpenMP和SIMD支持自动reduction操作(例如累加等),但是对于求标准差等复杂操作需要上述手工方法进行.

# Recurrence

```
do j=1,n
   a(j) = a(j-1) + b(j)
enddo
```

- 类似循环，当前迭代需要上一个迭代生成的数据
- 很难并行化

# Data ambiguity

```
Void func(int *a, int *b){
    for(j=0;j<MAX;j++){
        a[j] = b[j];
    }
}
```

编译器假设数组a和b是有重叠的,不进行simd优化.

需要加#pragma ivdep

# 函数调用

```
for (j=0; j<MAX; j++){
    compute(a[j], b[j]);
    a[j][1] = sin(b[j]);
}
```

- 函数调用会阻止ILP优化
- 许多库函数调用可能不是线程安全的,需要查手册进行确认,比如
  - 内存分配
  - 随机数生成
  - I/O函数

  ……

# Loop相关的简单测试

- 反转循环的顺序,如果结果是没有变化,那么该循环是loop independence
- 需要注意归纳变量

```
for (j=0; j<MAX; j++){
    ……
}

For(j=MAX;j>=0;j--) {
    ……
}
```

# 向量化

- Linux
  - -vec-report*n*
- Intel诊断用编译器选项
  - n=0: No diagnostic information
  - n=1: (Default) loop successfully vectorized
  - n=2: Loop not vectorized – and the reason why not
  - n=3: Adds dependency information
  - n=4: Reports only non-vectorized loops
  - n=5: Reports only non-vectorized loops and add dependency info

# 作业

1. 循环是否可以直接并行

```
do i=2,n
  a(i)=2*a(i-1)
end do

ix = base
do i=1,n
  a(ix) = a(ix)*b(i)
  ix  = ix + stride
end do

do i=1,n
   b(i)= (a(i)-a(i-1))*0.5
end do
```

2.上机调试培训内容中提供的求解pi的各OpenMP并行程序

3. mc的例子并行化

4.编写一个矩阵-向量相乘的OpenMP并行程序

5.分析critical、atomic、lock、flush的用法

```
void parallel_product(Matrix & m, Vector & v, Vector & r){
   #pragma omp parallel for num_threads(2)
      for(int i = 0; i < m.size1(); ++i){
            r(i) = 0.0;
            for(int j = 0; j < m.size2(); ++j){
                  r(i) += m(i,j)*v(j);
             }
      }
}
```

# 谢谢大家！