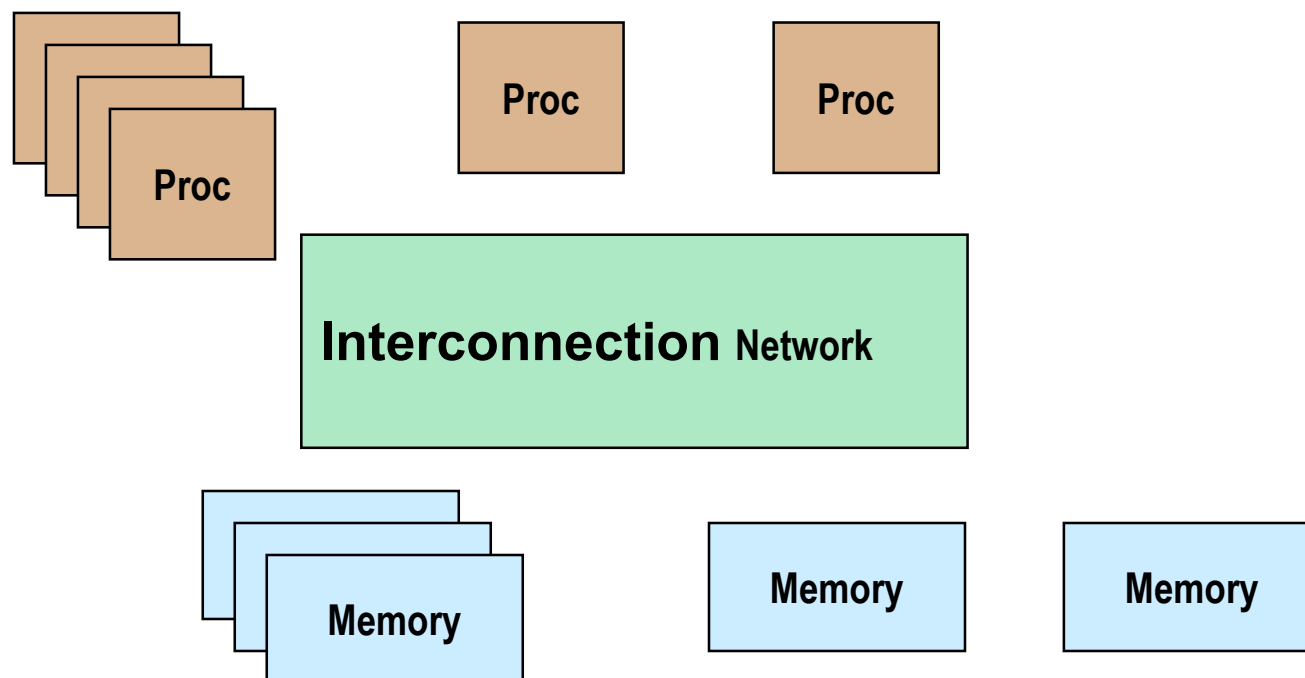

Lecture 07: 共享内存编程-OpenMP

肖俊敏

中国科学院计算技术研究所

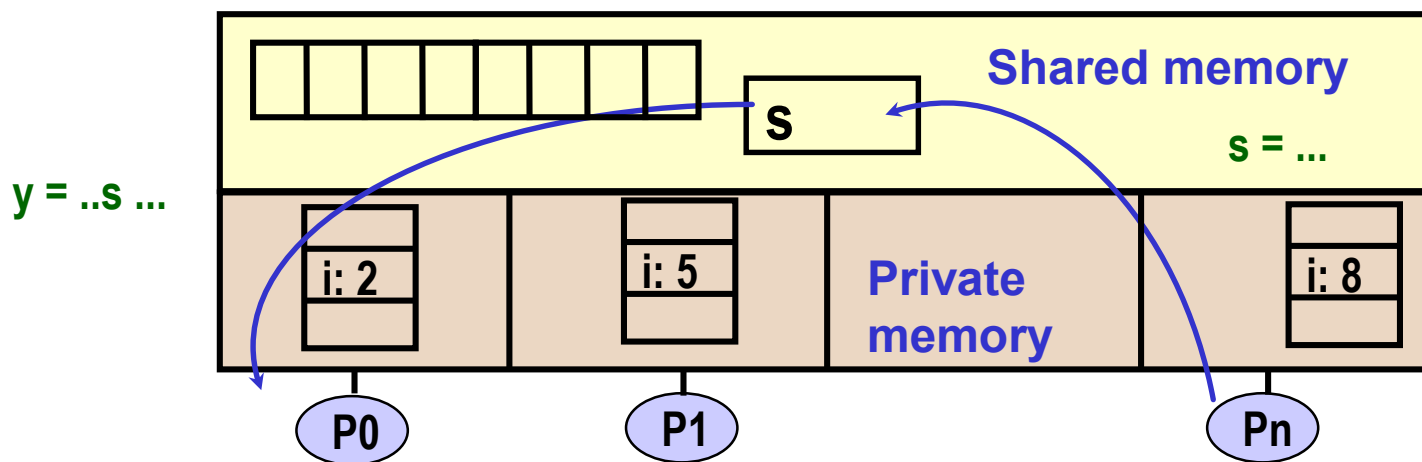
通用并行架构示例



- 内存物理上位于什么地方？
- 内存是否会直接连接到处理器上？

共享内存的编程模型

- 程序是控制线程的集合
 - 可以在某些语言中在执行过程中动态创建
- 每个线程都有一组**私有变量**，例如本地的栈变量
- 还有一组**共享变量**，例如静态变量、共享公共块或全局堆
 - 线程通过写入和读取共享变量进行**隐式通信**
 - 线程通过在共享变量上同步进行协调



目录

- 基于共享内存的多线程并行
- 什么是OpenMP&&为什么使用OpenMP
- 使用OpenMP的并行编程
- OpenMP介绍
 - 创建并行机制
 - 并行循环
 - 同步
 - 数据共享
- OpenMP背后的机制
 - 共享内存硬件
- 总结

基于共享内存的 多线程并行

POSIX线程概览

- **POSIX: P**ortable **O**perating **S**ystem **I**nterface of **U**NIX
 - 可移植操作系统接口
- **Pthreads: POSIX线程接口**
 - 创建和同步线程的系统调用
 - 在类似UNIX的操作系统平台上相对统一
- **PThreads支持**
 - 为程序创建并行机制
 - 同步机制
 - 没有显式通信，通过共享内存来隐式通信
 - 指向共享数据的指针传递给线程

POSIX线程的拷贝

Signature:

```
int pthread_create(pthread_t *,
                  const pthread_attr_t *,
                  void * (*) (void *),
                  void *);
```

Example call:

```
errcode = pthread_create(&thread_id; &thread_attribute
                        &thread_fun; &fun_arg);
```

- **thread_id 是线程id或句柄**
- **thread_attribute 各种属性**
 - 传递NULL指针会使用标准默认值
 - 示例属性：最小堆栈大小、优先级
- **thread_fun 要运行的函数（获取并返回void*）**
- **fun_arg启动时可以将参数传递给thread_fun**
- **errorcode 如果创建操作失败，将设置为非零**

简单的多线程程序示例

```
void* SayHello(void *foo) {  
    printf( "Hello, world!\n" );  
    return NULL;  
}
```

Compile using gcc -lpthread

```
int main() {  
    pthread_t threads[16];  
    int tn;  
    for(tn=0; tn<16; tn++) {  
        pthread_create(&threads[tn], NULL, SayHello,  
        NULL);  
    }  
    for(tn=0; tn<16 ; tn++) {  
        pthread_join(threads[tn], NULL);  
    }  
    return 0;  
}
```


循环层面的并行性

■ 许多科学应用程序在循环中具有并行性

■ 例如:

```
... my_stuff [n][n];  
for (int i = 0; i < n; i++)  
  for (int j = 0; j < n; j++)  
    ... pthread_create (update_cell[i][j], ...,  
                        my_stuff[i][j]);
```

■ 但是线程创建的开销比较大

- update_cell应该是主要的工作负载
- 如果可能, 占总工作量的 $1/p$ (p 个线程)
- my_stuff是共享数据, 不同线程都可以访问

数据竞争(Data Race)的例子

```
static int s = 0;
```

Thread 1

```
for i = 0, n/2-1  
  s = s + f(A[i])
```

Thread 2

```
for i = n/2, n-1  
  s = s + f(A[i])
```

- 问题是程序中变量s的竞赛条件
- 当出现以下情况时，将出现数据竞争：
 - 两个处理器（或两个线程）访问同一个变量，并且至少有一个处理器进行写入。
 - 访问是并发的（不同步），因此它们可以同时发生

同步的基本数据类型：Mutexes

■ Mutexes——互斥(mutual exclusion), 也叫锁(locks)

- 线程基本上是独立工作的
- 需要访问通用数据结构

```
lock *l = alloc_and_init();  /* shared */
acquire(l);
access data;
release(l);
```
- 锁仅影响使用它们的处理器：
 - 如果一个线程访问数据而不进行释放, 那么其他线程的锁也将失去作用
- Java和其他语言具有词法范围的同步, 即同步的方法/块
 - 切记不要忘记 “release”
- 信号量(Semaphores)是一种泛化锁, 允许k个线程同时访问; 有利于有限的资源

POSIX线程中的Mutexes

■ 创建mutex:

```
#include <pthread.h>
```

```
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
```

```
// or pthread_mutex_init(&amutex, NULL);
```

■ 使用:

```
int pthread_mutex_lock(amutex);
```

```
int pthread_mutex_unlock(amutex);
```

■ 释放:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

■ 同时使用多个锁会导致问题:

thread1

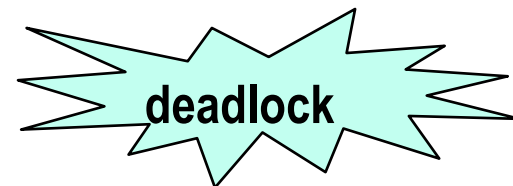
thread2

lock(a)

lock(b)

lock(b)

lock(a)



- 如果两个线程都获取了其中一个锁，那么就会导致死锁，因此两个线程均无法获取第二个锁

使用PThreads编程的总结

- **POSIX线程基于操作系统功能实现**
 - 可以在多种语言中使用（需要引入适当的头文件）
 - 大多需要熟悉程序实现
 - 共享内存的能力很方便
- **缺陷**
 - 线程创建的开销很高（单层循环迭代可能更多）
 - 数据竞争漏洞很难发现
 - 死锁通常更容易发现
- **研究人员通常使用事务性内存(transactional memory)作为一种方案**
- **OpenMP作为一种替代方案在现在更常用**
 - 但有些案例仍然会使用PThreads

什么是OpenMP

为什么使用OpenMP

什么是OpenMP

- **OpenMP=Open specification for Multi-Processing**
 - openmp.org 提供讨论、示例、论坛等
 - 由ARB控制的规范
- **动机：表达常见并行用法并简化编程**
- **OpenMP Architecture Review Board (ARB)**
 - 控制OpenMP规范的非营利组织
 - 最新规范：OpenMP 5.2，即将发布6.0
- **用于C/C++和Fortran编程的高级API**
 - 预处理器（编译器）指令（~80%）
 - `#pragma omp construct[clause [clause...]]`
 - 库的调用（约19%）
 - `#include <omp.h>`
 - 环境变量（~1%）
 - 程序运行前设置线程数目、运行方式（例如访存方式）、等等

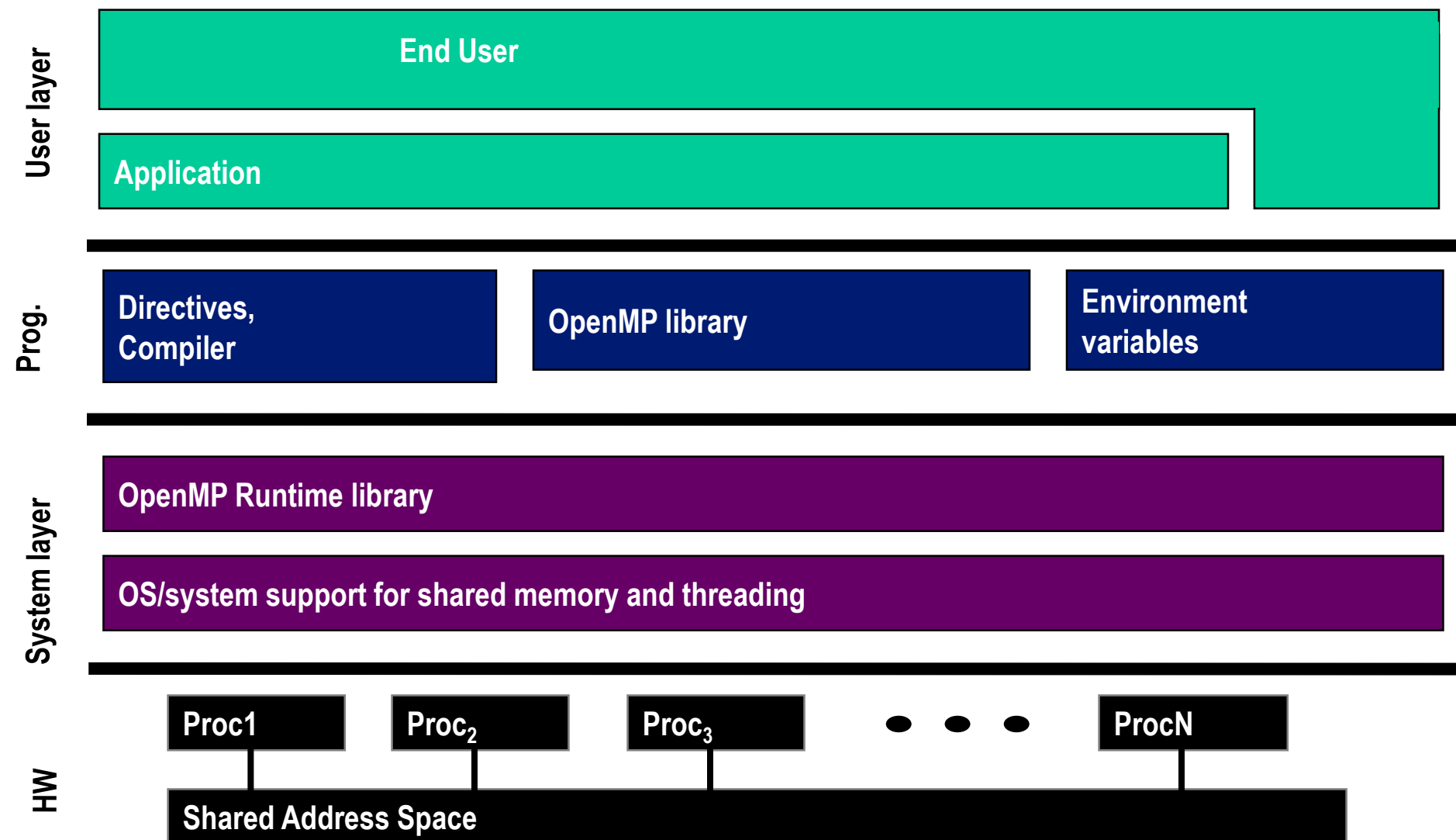
开发者视角下的OpenMP

- **OpenMP是一种可移植、线程化、共享内存编程且语法轻便的规范**
 - 需要编译器支持 (C、C++或Fortran)
- **OpenMP可以:**
 - 允许程序员将程序分为串行区域和并行区域
 - 隐藏堆栈管理
 - 提供同步语句
- **OpenMP不可以:**
 - 自动并行
 - 保证加速
 - 自动防止数据竞争(data races)

OpenMP通用核心：大多数OpenMP程序使用的语句

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads
int omp_get_thread_num() int omp_get_num_threads()	Create threads with a parallel region and split up the work using the number of threads and thread ID
double omp_get_wtime()	Speedup and Amdahl's law. False Sharing and other performance issues
setenv OMP_NUM_THREADS N	Internal control variables. Setting the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies
reduction(op:list)	Reductions of values across a team of threads
schedule(dynamic [,chunk]) schedule (static [,chunk])	Loop schedules, loop overheads and load balance
private(list), firstprivate(list), shared(list)	Data environment
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive)
#pragma omp single	Workshare with a single thread
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

OpenMP基本定义: basic solution stack



OpenMP基本语法

■ OpenMP中的大多数构造过程都是编译器完成的

C and C++	Fortran
Compiler directives	
<i>#pragma omp construct [clause [clause]...]</i>	<i>!\$OMP construct [clause [clause] ...]</i>
Example	
<i>#pragma omp parallel private(x)</i> { }	<i>!\$OMP PARALLEL</i> <i>!\$OMP END PARALLEL</i>
Function prototypes and types:	
<i>#include <omp.h></i>	<i>use OMP_LIB</i>

- 大多数OpenMP*结构适用于 “structured block” 。
- 结构化块：一个或多个语句的块，顶部有一个入口点，底部有一个出口点
 - 在结构化块中有一个exit () 是可以的

Hello world in OpenMP

- 写一个简单的hello world程序

```
#include<stdio.h>
int main()
{
    printf( " hello " );
    printf( " world \n" );
}
```

Hello world in OpenMP

■ 写一个多线程的hello world程序

```
#include <omp.h>
#include <stdio.h>
int main(){
    #pragma omp parallel
    {
        printf(" hello ");
        printf(" world \n");
    }
}
```

Switches for compiling and linking

gcc -fopenmp Gnu (Linux, OSX)

pgcc -mp pgi PGI (Linux)

icl /Qopenmp Intel (windows)

icc -fopenmp Intel (Linux, OSX)

Hello world in OpenMP

■ 写一个多线程的hello world程序

```
#include <omp.h>
#include <stdio.h>
int main()
{
    #pragma omp parallel
    {
        printf(" hello ");
        printf(" world \n");
    }
}
```

OpenMP include file

默认线程数目的并行区域

并行区域结束

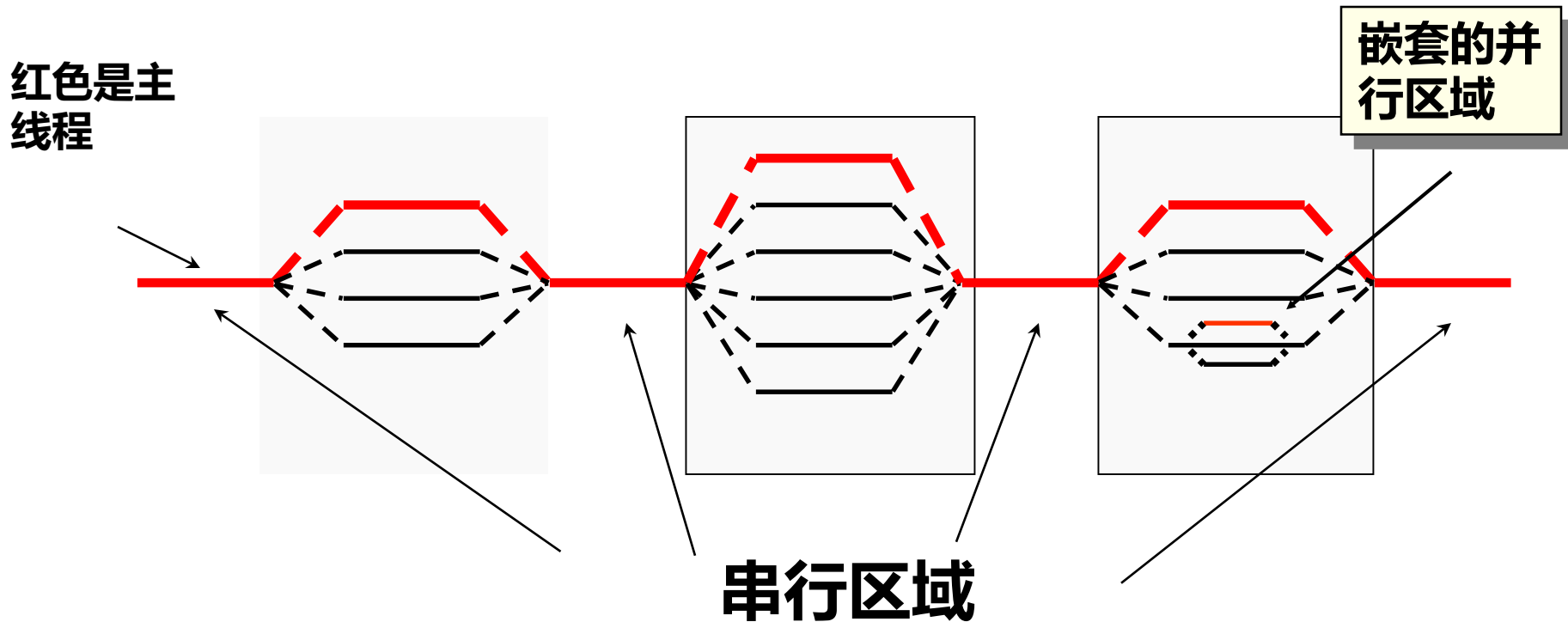
Sample Output:

```
hello hello world
world
hello hello world
world
```

OpenMP编程模型

■ Fork-Join 并行机制

- 主线程根据需要生成一组线程
- 并行度逐渐增加，直到达到性能目标，即串行程序演变为并行程序



创建线程：Parallel regions

- 使用并行结构体(parallel construct)在OpenMP中创建线程
- 例如，要创建4线程并行区域：

每个线程在
结构化块中
执行代码的
副本

```
double A[1000];  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    int ID = omp_get_thread_num();  
    pooh(ID,A);  
}
```

请求一定数量线程
的运行函数

运行时函数返回线程
ID

- 当ID = 0~3时，每个线程调用pooh (ID, A)

创建线程：Parallel regions

■ 每个线程都冗余地执行相同的代码

double A[1000];

omp_set_num_threads(4)

A的单个副本在所有线程之间共享

pooh(0,A)

pooh(1,A)

pooh(2,A)

pooh(3,A)

printf("all done\n");

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```

线程在这里等待所有线程完成后再继续
(即存在一个屏障barrier)

线程创建：实际创建了多少个线程？

- 可以使用并行结构体在OpenMP中创建一组线程
- 可以使用`omp_set_num_threads()`请求多个线程
- 但是，请求的线程数就是实际得到的线程数吗？
 - 否！底层实现默认地创建一个线程数较小的组，但是一旦建立了一个一组线程，系统就不会减少其规模。

每个线程
在结构化
块中执行
代码的副
本

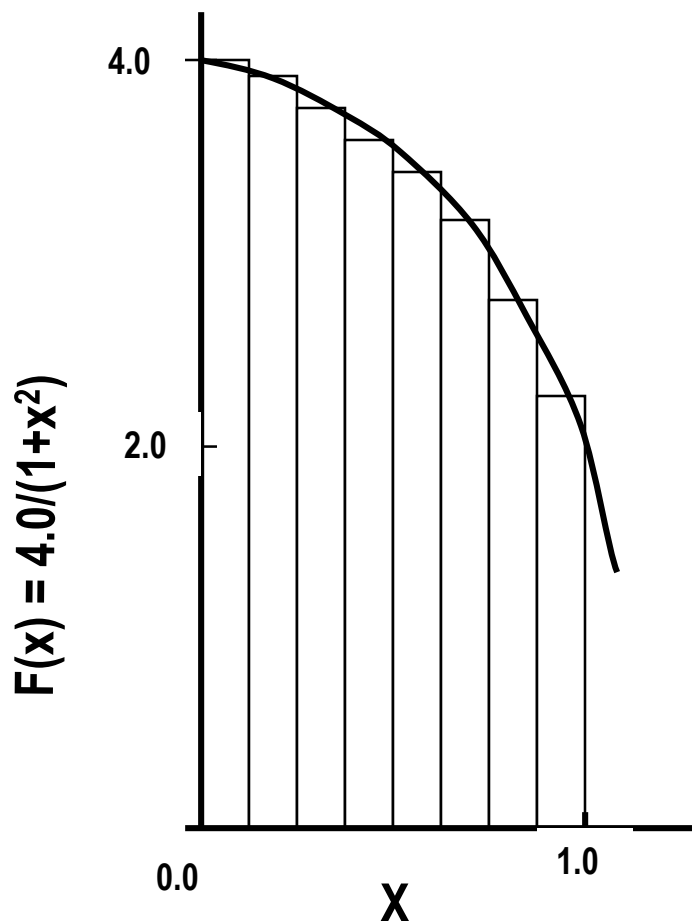
```
double A[1000];  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    int ID    = omp_get_thread_num();  
  
    int nthrds = omp_get_num_threads();  
    pooh(ID,A);  
}
```

请求一定数量线程
的运行时函数

用于返回实际线程
数的运行时函数

当ID = 0~3时，每个线程调用pooh (ID, A)

数值积分的一个有趣问题



数学上有:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

我们可以将积分近似为矩形的总和:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

每个矩形的宽为 Δx 高为 $F(x_i)$

串行的 π 程序

```
static long num_steps = 100000;
double step;
int main ()
{
    int i;    double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

串行的 π 程序

```
#include <omp.h>
static long num_steps = 100000;
double step;
int main ()
{
    int i;    double x, pi, sum = 0.0, tdata;

    step = 1.0/(double) num_steps;
    double tdata = omp_get_wtime();
    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
    tdata = omp_get_wtime() - tdata;
    printf(" pi = %f in %f secs\n",pi, tdata);
}
```

get_omp_wtime ()
用于查找代码块经过的
“墙上时间(wall
time)”

并行的 π 程序

■ 对比：串行pi程序在1.83秒内运行了100000步

```
#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 2
void main ()
{
    int i, nthreads; double pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthrds = nthrds;
        for (i=id, sum[id]=0.0; i< num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0; i<nthreads; i++) pi += sum[i] * step;
}
```

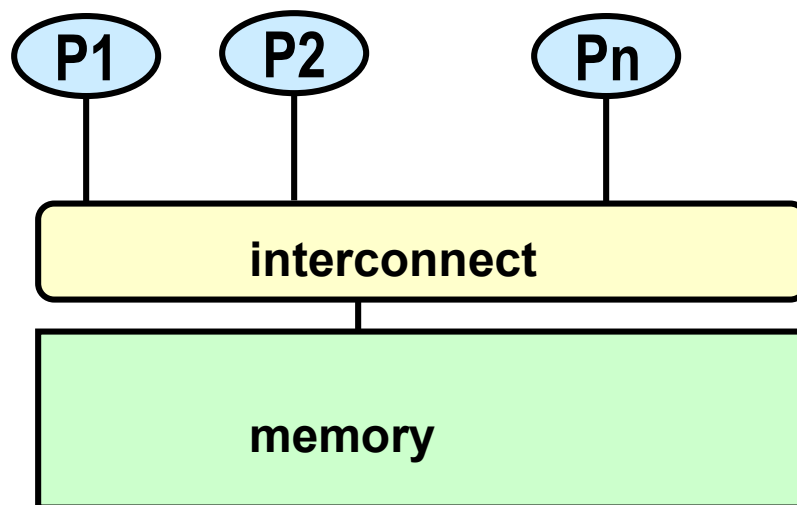
threads	1 st SPMD*
1	1.86
2	1.03
3	1.08
4	0.97

* 测试环境：Apple OS X 10.7.3上优化级别（O2）的英特尔编译器（icpc），双核（四硬件线程）英特尔®Core™ i5处理器，1.7 Ghz，4 GB DDR3内存，1.333 Ghz。

共享内存硬件与 内存一致性

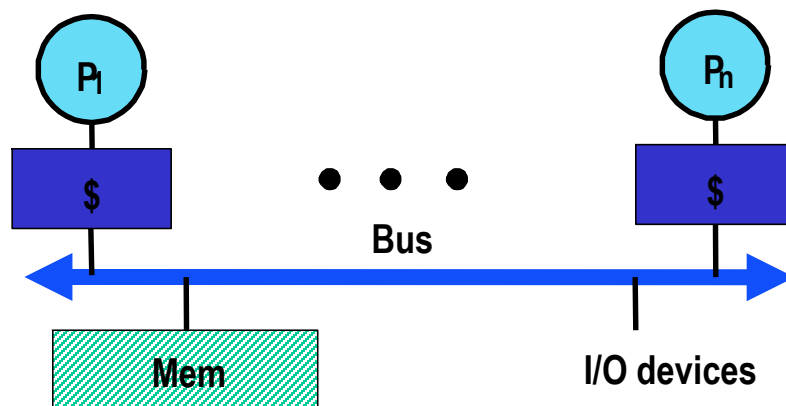
基本共享内存架构

- 所有处理器都连接到一个大型共享内存
 - 缓存在哪里？



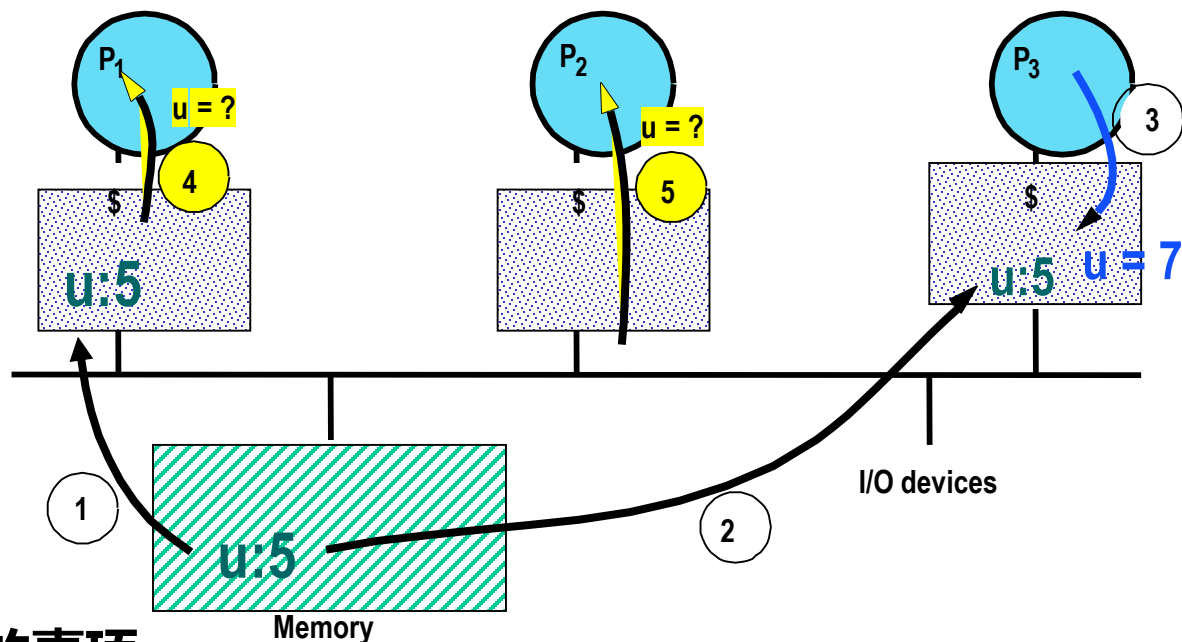
- 需要进一步去看结构、开销、限制、程序

共享内存中的Caching



- **实现共享内存的高性能需要使用缓存**
 - 每个处理器都有自己的缓存（或多个缓存）
 - 将数据从内存放入缓存
 - 写回缓存：不要通过总线将所有写操作发送到内存
- **缓存减少了平均延迟**
 - 缓存有自动复制机制，且相较于主存更接近处理器
 - 延迟对多处理器来说比单处理器更重要
- **单处理器的访问数据机制**
 - 读取和写回使用非常低开销的通信原语
- **问题：缓存一致性！**

缓存一致性问题示例



■ 需要注意的事项:

- 事件3之后, 处理器可能会看到不同的 u 值
- 缓存写回内存的值取决于缓存刷新的时机或写回值的时机

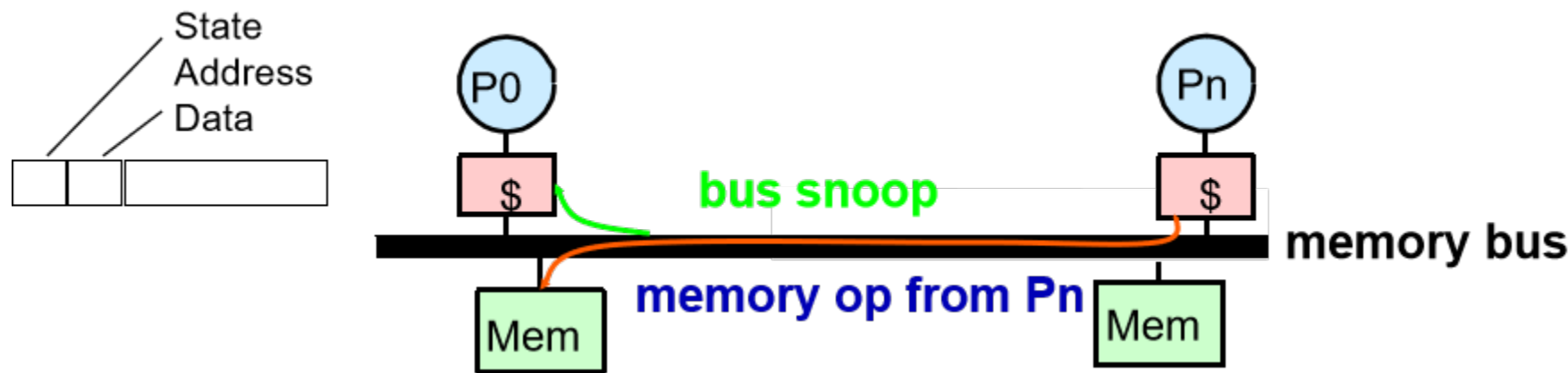
■ 如何使用总线进行修复: 一致性协议

- 使用总线广播写入或无效
- 简单的协议依赖于广播媒体的存在

■ 总线无法扩展到超过约100个处理器 (最大)

- 容量、带宽限制

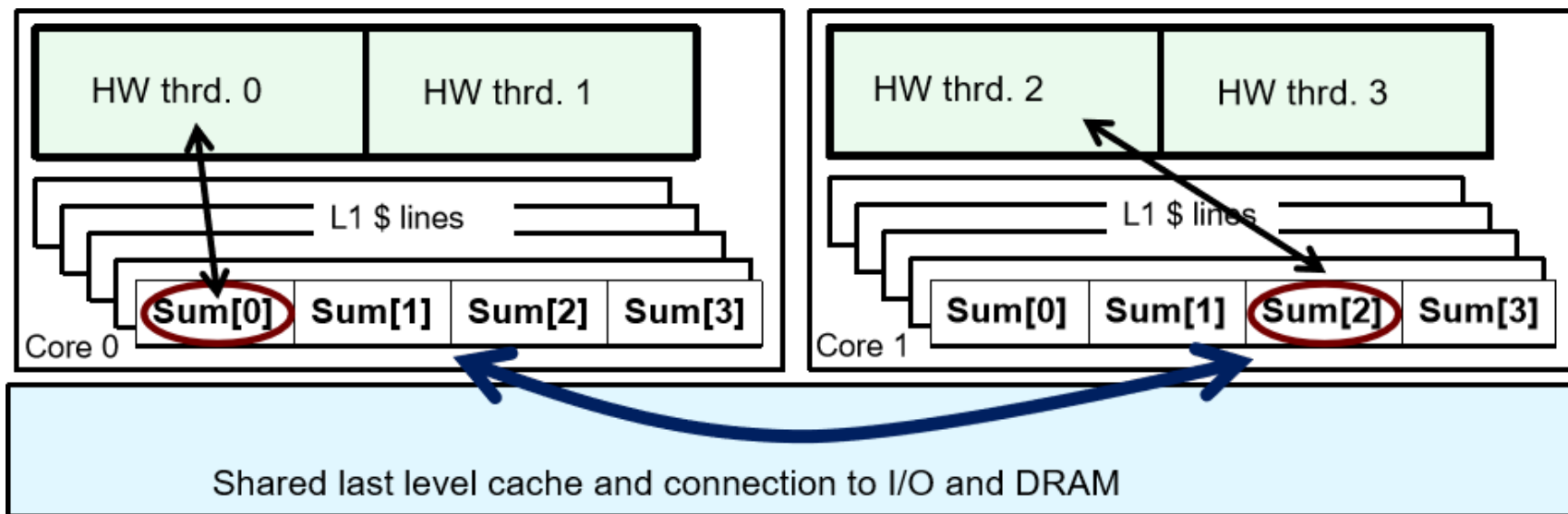
Snoopy缓存一致性协议



- 内存总线是一种广播媒介
- 缓存包含存储地址的信息
- 缓存控制器“窥探”总线上的所有事务
 - 如果事务涉及当前包含在该缓存中的缓存块，则该事务就是**相关事务**
 - 采取行动确保一致性
 - 使值无效、更新或提供值
 - 还有许多其他可能的设计

为什么缩放问题规模(scaling)的效果这么差? **false sharing**

- 如果独立的数据元素恰好位于同一个缓存行(cache line)上, 每次更新都会导致缓存行在线程之间“来回捣腾”, 这被称为“**false sharing**”




- 如果将标量提升到数组以支持SPMD程序的创建, 则数组元素在内存中是连续的, 因此共享缓存行会导致较差的可扩展性。
- 解决方案: 填充数组(padding), 使您使用的元素位于不同的缓存行上。

示例：通过填充sum数组来消除false sharing

```
#include <omp.h>
static long num_steps = 100000;    double step;
#define PAD 8                      // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{
    int i, nthreads; double pi, sum[NUM_THREADS][PAD];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id, sum[id][0]=0.0; i< num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum[id][0] += 4.0/(1.0+x*x);
        }
    }

    for(i=0, pi=0.0; i<nthreads; i++) pi += sum[i][0] * step;
}
```



填充数组，使每个和值
位于不同的缓存行中

π程序使用了padding后

■ 对比：串行pi程序在1.83秒内运行了100000步

```
#include <omp.h>
static long num_steps = 100000;      double step;
#define PAD 8                        // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{
    int i, nthreads; double pi, sum[NUM_THREADS][PAD];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id, sum[id]=0.0; i< num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum[id][0] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0; i<nthreads; i++)pi += sum[i][0] * step;
}
```

threads	1 st SPMD	1 st SPMD padded
1	1.86	1.86
2	1.03	1.01
3	1.08	0.69
4	0.97	0.53

* 测试环境：Apple OS X 10.7.3上优化级别（O2）的英特尔编译器（icpc），双核（四硬件线程）英特尔®Core™ i5处理器，1.7 Ghz，4 GB DDR3内存，1.333 Ghz。

同步

- **公共核心中包含的高级同步（完整的OpenMP规范还有很多）：**
 - critical region – 临界区
 - barrier – 同步屏障

**同步用于施加顺序约束，
并保护对共享数据的访问**

同步: critical

- 互斥: 一次只能有一个线程进入临界区(critical region)

线程等待轮到它们——一次只有一个线程调用
consumer ()

```
float res;  
  
#pragma omp parallel  
{   float B;  int i, id, nthrds;  
    id = omp_get_thread_num();  
    nthrds = omp_get_num_threads();  
    for(i=id;i<niters;i+=nthrds){  
        B = big_job(i);  
        #pragma omp critical  
        res += consume (B);  
    }  
}
```


同步屏障：barrier

- 同步屏障：程序中所有线程在允许任何线程继续之前都要到达的一个点。
- 它是一个“独立”的编译指示，意味着它与用户代码无关.....它是一条可执行语句。

```
double Arr[8], Brr[8];      int numthrds;

omp_set_num_threads(8)

#pragma omp parallel
{  int id, nthrds;

    id = omp_get_thread_num();


    nthrds = omp_get_num_threads();

    if (id==0) numthrds = nthrds;

    Arr[id] = big_ugly_calc(id, nthrds);

    #pragma omp barrier
    Brr[id] = really_big_and_ugly(id, nthrds, Arr);
}
```

线程等待，
直到所有线程都到达屏障，然后才可以继续



示例：使用临界区来消除false sharing的影响

```
#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 2
void main ()
{
    int nthreads; double pi=0.0;    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
    int i, id, nthrds; double x, sum;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if (id == 0) nthreads = nthrds;
    for (i=id, sum=0.0; i< num_steps; i=i+nthrds) {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    #pragma omp critical
        pi += sum * step;
}
}
```

为每个线程创建一个局部标量以累积部分和

没有数组，所以没有false sharing

求和超出了并行区域单一线程的“范围”，所以必须在临界区求和，维护总和pi，这样更新就不会发生冲突

使用关键区的 π 程序

■ 对比：串行pi程序在1.83秒内运行了100000步

```
#include <omp.h>
static long num_steps = 100000;      double step;
#define NUM_THREADS 2
void main ()
{
    int nthreads; double pi=0.0;      step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int i, id, nthrds;  double x, sum;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthrds = nthrds;
        for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
        #pragma omp critical
            pi += sum * step;
    }
}
```

threads	1 st SPMD	1 st SPMD padded	SPMD critical
1	1.86	1.86	1.87
2	1.03	1.01	1.00
3	1.08	0.69	0.68
4	0.97	0.53	0.53

* 测试环境：Apple OS X 10.7.3上优化级别（O2）的英特尔编译器（icpc），双核（四硬件线程）英特尔®CoreTM i5处理器，1.7 Ghz，4 GB DDR3内存，1.333 Ghz。

loop worksharing的结构体

■ loop worksharing在一组线程之间划分循环迭代

```
#pragma omp parallel
```

```
{  
#pragma omp for  
    for (l=0;l<N;l++){  
        NEAT_STUFF(l);  
    }  
}
```

Loop construct name:

- C/C++: for
- Fortran: do

默认情况下，循环控制索引l对每个线程都是“私有的”

线程在这里等待，直到所有线程都完成了并行循环，然后再继续进行循环结束

loop worksharing的例子

串行代码

```
for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

OpenMP parallel region

```
#pragma omp parallel
```

```
{
```

```
    int id, i, Nthrds, istart, iend;
```

```
    id = omp_get_thread_num();
```

```
    Nthrds = omp_get_num_threads();
```

```
    istart = id * N / Nthrds;
```

```
    iend = (id+1) * N / Nthrds;
```

```
    if (id == Nthrds-1)iend = N;
```

```
    for(i=istart;i<iend;i++) { a[i] = a[i] + b[i];}
```

```
}
```

OpenMP parallel region and loop worksharing

```
#pragma omp parallel
```

```
#pragma omp for
```

```
    for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

loop worksharing: schedule子句

■ schedule子句影响循环迭代映射到线程的方式

- `schedule (static[, chunk])`
 - 每个线程处理大小为“chunk”的迭代块。
- `schedule (dynamic[, chunk])`
 - 每个线程从队列中抓取大小为“chunk”的迭代块，直到所有迭代都得到处理

Schedule Clause	When To Use
STATIC	程序员预先确定并可预测每次迭代的工作
DYNAMIC	每次迭代的工作不可预测，变化很大

运行时工作量少：在编译时完成调度

大多数工作在运行时进行：运行时使用的复杂调度逻辑

组合parallel/worksharing结构体

- OpenMP的快捷方式：将“parallel”与worksharing指令放在同一行

```
double res[MAX]; int i;  
#pragma omp parallel  
{  
    #pragma omp for  
    for (i=0; i< MAX; i++) {  
        res[i] = huge();  
    }  
}
```

```
double res[MAX]; int i;  
#pragma omp parallel for  
for (i=0; i< MAX; i++) {  
    res[i] = huge();  
}
```



两个程序是等价的

使用组合结构体来编写程序

■ 基本方法

- 查找计算密集型循环
- 使循环的每次迭代独立，这样它们可以安全地按任何顺序执行，而无需循环携带依赖项
- 放置适当的OpenMP指令并进行测试

```
int i, j, A[MAX];  
j = 5;  
for (i=0; i< MAX; i++) {  
    j += 2;  
    A[i] = big(j);  
}
```

注意：默认情况下，循环索引“i”是私有的

Remove loop carried dependence

```
int i, A[MAX];  
#pragma omp parallel for  
for (i=0; i< MAX; i++) {  
    int j = 5 + 2*(i+1);  
    A[i] = big(j);  
}
```


归约

- 如何处理下面的例子？

```
double ave=0.0, A[MAX];  int i;
for (i=0;i< MAX; i++) {
    ave += A[i];
}
ave = ave/MAX;
```

- 我们将值组合成一个单独的累积变量 (ave) , 循环迭代之间存在着真正的相关性
- 这是一种非常常见的归约情况，类似行为统称为“归约” (reduction)
- 在大多数并行编程环境中都包含对归约操作的支持

归约

■ OpenMP归约语句:

- reduction(op:list)

■ 在并行结构或工作共享结构内部:

- 每个列表变量的本地副本(local copy)都是根据 “op” (例如0表示 “+”) 生成和初始化的
- 更新发生在本地副本上
- 本地副本被归约为单个值, 并与原始全局值组合。

■ “list” 中的变量必须在封闭的并行区域中共享。

```
double ave=0.0, A[MAX];  int i;  
#pragma omp parallel for reduction (+:ave)  
for (i=0;i< MAX; i++) {  
    ave += A[i];  
}  
ave = ave/MAX;
```

OpenMP: 归约操作数/初始值

- 可以使用许多不同的关联操作数进行减少:
- 初始值在数学上是有意义的

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

C/C++ only	
Operator	Initial value
&	~0
	0
^	0
&&	1
	0

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.

示例： π 程序，使用循环和归约

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

```
static long num_steps = 100000;    double step;
```

```
void main ()
```

```
{  int i;          double x, pi, sum = 0.0;
```

```
  step = 1.0/(double) num_steps;
```

```
  #pragma omp parallel
```

```
  {
```

```
    double x;
```

```
    #pragma omp for reduction(+:sum)
```

```
      for (i=0;i< num_steps; i++){
```

```
        x = (i+0.5)*step;
```

```
        sum = sum + 4.0/(1.0+x*x);
```

```
      }
```

```
  }
```

```
    pi = step * sum;
```

```
}
```

创建一组线程，没有这行语句是无法创建多线程环境的

创建每个线程的局部标量，以在每次迭代维护x值

分解循环迭代并将其分配给线程...将归约值设置为总和。注意...默认情况下，循环索引是线程的本地索引。

结果：π程序，使用循环和归约

■ 对比：串行pi程序在1.83秒内运行了100000步


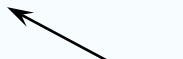
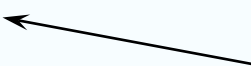
```
#include <omp.h>
static long num_steps = 100000
void main ()
{  int i;      double x, pi, sum
   step = 1.0/(double) num_steps
   #pragma omp parallel
   {
       double x;
       #pragma omp for reduction
       for (i=0;i< num_steps;
           x = (i+0.5)*step;
           sum = sum + 4.0/(1.0+x*x);
       }
   }
   pi = step * sum;
}
```

threads	1 st SPMD	1 st SPMD padded	SPMD critical	PI Loop and reduction
1	1.86	1.86	1.87	1.91
2	1.03	1.01	1.00	1.02
3	1.08	0.69	0.68	0.80
4	0.97	0.53	0.53	0.68

* 测试环境：Apple OS X 10.7.3上优化级别（O2）的英特尔编译器（icpc），双核（四硬件线程）英特尔®Core™ i5处理器，1.7 Ghz，4 GB DDR3内存，1.333 Ghz。

nowait子句

- 同步屏障开销很大
- 需要明确何时出现隐式屏障，以及如何在安全的情况下跳过同步屏障。

```
double A[big], B[big], C[big];  
  
#pragma omp parallel  
{  
    int id=omp_get_thread_num();  
    A[id] = big_calc1(id);  
  
#pragma omp barrier  
#pragma omp for  
    for(i=0;i<N;i++){C[i]=big_calc3(i,A);}   
  
#pragma omp for nowait  
    for(i=0;i<N;i++){ B[i]=big_calc2(C, i); }   
    A[id] = big_calc4(id);  
} 
```

for worksharing结构体末尾的
隐式屏障

并行区域末端的隐式屏障

由于nowait不产生隐式屏障

数据环境：默认存储属性

■ 共享内存编程模型：

- 默认情况下，大多数变量都是共享的

■ 全局变量在线程之间共享

- Fortran: COMMON块、SAVE变量、MODULE变量
- C: 文件范围变量，静态
- Fortran&C: 动态分配的内存 (ALLOCATE、malloc、new)

■ 但并不是所有的东西都是共享的

- 从并行区域调用的子程序 (Fortran) 或函数 (C) 中的栈变量为PRIVATE
- 语句块中的自动产生的变量是PRIVATE。

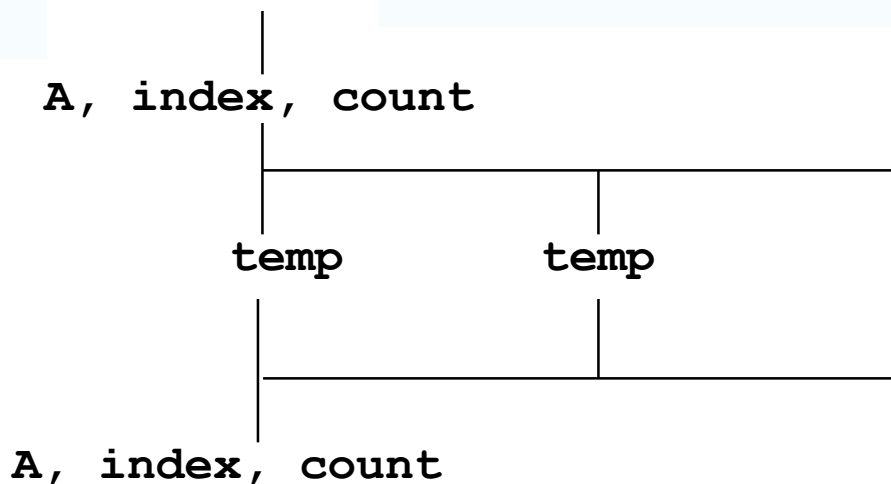
数据共享示例

```
double A[10];
int main() {
    int index[10];
    #pragma omp parallel
        work(index);
    printf( "%d\n" , index[0]);
}
```

```
extern double A[10];
void work(int *index) {
    double temp[10];
    static int count;
    ...
}
```

**A、index和count由
所有线程共享。**

**temp是每个线程的本地
变量**



数据共享：更改存储属性

- 可以使用以下子句选择性地更改结构体中list的存储属性*（注意：列表是以逗号分隔的变量列表）

- shared(list)
- private(list)
- firstprivate(list)

这些子句适用于OpenMP的结构体语句，而不适用于整个区域

- 这些属性设置可以用于parallel和for结构体，而不是只能用于parallel结构体

- 强制程序员显式定义存储属性

- default(none)

default () 可以用于并行结构体

数据共享：private子句

- **private(var)** 为每个线程创建一个新的var本地副本。
 - private副本的值是未初始化的
 - 原始变量的值在并行区域之后保持不变

```
void wrong() {  
    int tmp = 0;  
    #pragma omp parallel for private(tmp)  
    for (int j = 0; j < 1000; ++j)  
        tmp += j;  
    printf( "%d\n" , tmp);  
}
```

当您需要引用在结构体之前存在的变量tmp时，我们将其称为**原始变量**。

tmp is 0 here

tmp was not initialized

数据共享：private子句的原始变量何时有效？

- 在OpenMP结构体语句中引用原始变量，则其初始值是未定义的
 - 程序实现可能会引用原始变量或副本，这是危险的
 - 例如，考虑一下如果编译器内联work () 会发生什么？

```
int tmp;  
void danger() {  
    tmp = 0;  
#pragma omp parallel private(tmp)  
    work();  
    printf( "%d\n" , tmp);  
}
```

这里tmp值是？

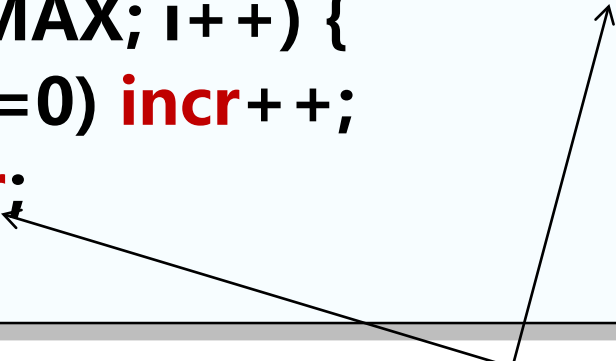
```
extern int tmp;  
void work() {  
    int ID = omp_get_thread_num();  
    tmp = ID;  
}
```

对tmp的原始变量赋值

firstprivate子句

- 从共享变量初始化的变量
- C++对象是复制构造的

```
incr = 0;  
#pragma omp parallel for firstprivate(incr)  
for (i = 0; i <= MAX; i++) {  
    if ((i%2)==0) incr++;  
    A[i] = incr;  
}
```



每个线程都有自己的incr副本，初始值为0

数据共享：数据环境测试

■ 考虑PRIVATE和FIRSTPRIVATE的示例

```
variables: A = 1, B = 1, C = 1  
#pragma omp parallel private(B) firstprivate(C)
```

- A、B、C是每个线程专用的还是在并行区域内共享的？
- 它们在并行区域内的初始值和并行区域后的初始值是多少？

在并行区域内

- A被所有线程共享，值为1
- B、C是每个线程私有的。
 - B是未初始化的
 - C初始化为1

在并行区域后

- B、C恢复为初始值1
- A为1或在并行区域内设置的值

数据共享：default子句

- **default (none)** : 强制为出现在并行结构体的静态范围内的变量定义存储属性, 如果失败, 编译器将报错, 是一种推荐的编程习惯
- 可以将default子句放在parallel以及parallel+worksharing的并行结构体上

静态范围是包含该结构体的编译单元中的代码

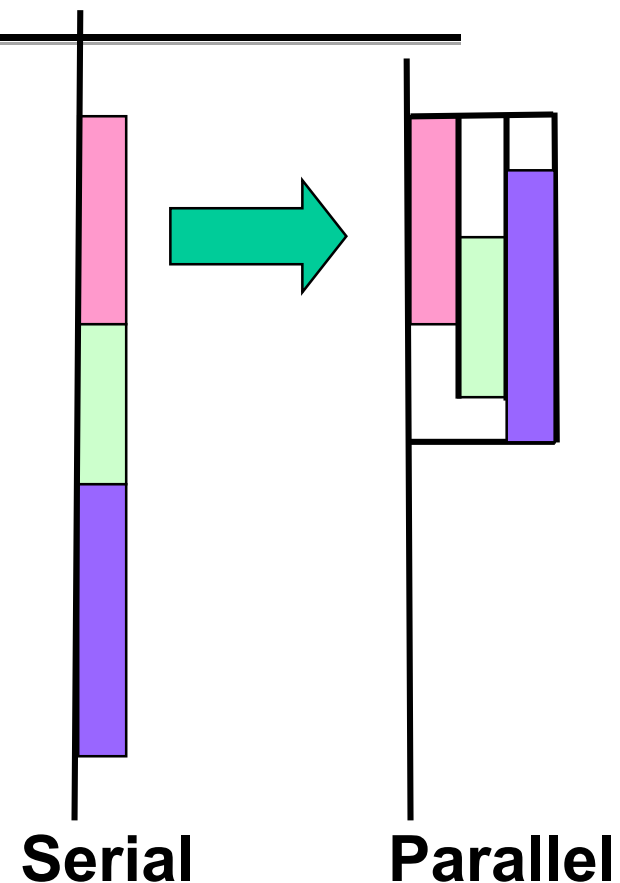
```
#include <omp.h>
int main()
{
    int i, j=5;    double x=1.0, y=42.0;
    #pragma omp parallel for default(none) reduction(*:x)
    for (i=0;i<N;i++){
        for(j=0; j<3; j++)
            x *= foobar(i, j, y);
    }
    printf( " x is %f\n" ,(float)x);
}
```

编译器会报错j, y, N没有属性, 在这里不希望j是被共享的

完整的OpenMP规范有其他版本的default子句, 但它们不经常使用
所以我们在公共核心中跳过它们

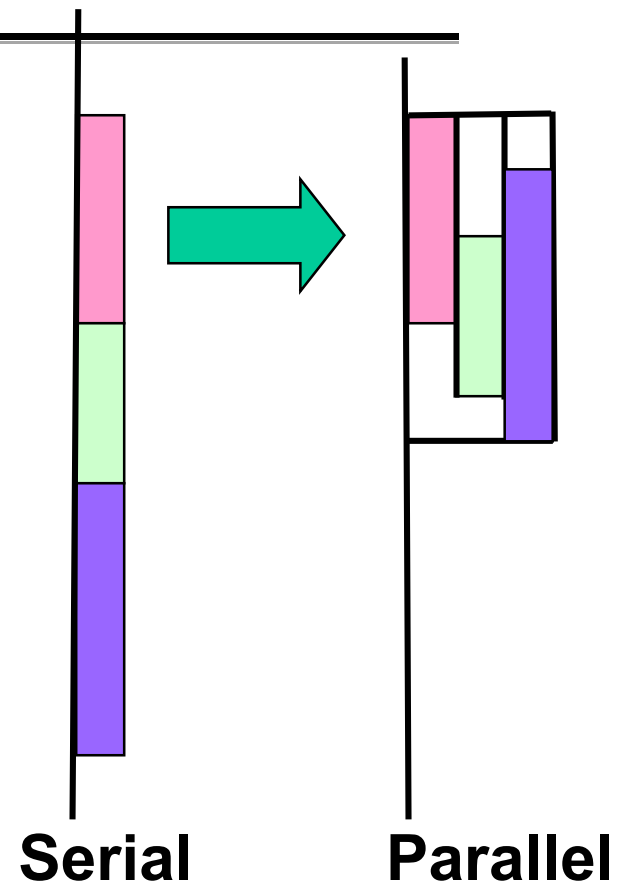
什么是tasks?

- **tasks是独立的工作单元**
- **tasks包括:**
 - 要执行的代码
 - 要计算的数据
- **线程被分配来执行每个task/任务**
 - 分配了task的线程可以立即执行
 - 线程可能会延迟执行



什么是tasks?

- task结构包括一个结构化的代码块
- 在并行区域内，分配了任务的线程将打包代码块及其数据以供执行
- task可以嵌套：即task本身可以生成task。



一种常见的模式是让一个线程创建任务
而其他线程在同步屏障处等待并执行任务

single worksharing结构体

- single表示仅由一个线程（不一定是主线程）执行的代码块。
- 屏障隐含在single block的末尾（可以通过nowait子句来消除障碍）。

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp single
    {    exchange_boundaries();  }
    do_many_other_things();
}
```

任务命令

`#pragma omp task [clauses]`
structured-block

```
#pragma omp parallel
```

创建线程

```
{
```

```
    #pragma omp single
```

单线程任务创建

```
{
```

```
    #pragma omp task  
        fred();
```

```
    #pragma omp task  
        daisy();
```

```
    #pragma omp task  
        billy();
```

某个线程按某种顺序
执行的任务

```
}
```

```
}
```

此屏障释放前完成所有任务

task何时/何地完成?

■ 线程屏障处（显式或隐式）

- 应用于当前并行区域中生成的所有任务，直到抵达同步屏障


■ taskwait命令

- 即：等待，直到当前task中定义的所有任务都已完成
 - `#pragma omp taskwait`
- 注意：仅适用于当前任务中生成的任务

示例

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        fred();
        #pragma omp task
        daisy();
        #pragma taskwait
        #pragma omp task
        billy();
    }
}
```

**fred () 和daisy ()
必须在billy () 启动之
前完成**



数据作用域的默认值

- 任务所需的行为通常是firstprivate, 因为任务可能不会立即执行 (变量此时可能已超出范围)
 - 任务构造时私有的变量默认情况下为firstprivate
- 默认情况下, 从最里面的封闭并行结构体开始的所有结构体中共享的变量都是共享的

```
#pragma omp parallel shared(A) private(B)
{
    ...
#pragma omp task
{
    int C;
    compute(A, B, C);
}
}
```

A is shared
B is firstprivate
C is private



示例：斐波那契数列

```
int fib (int n)
{
    int x,y;
    if (n < 2) return n;
```

- $F_n = F_{n-1} + F_{n-2}$
- $O(n^2)$ 时间复杂度的低效递归实现

```
    x = fib(n-1);
    y = fib (n-2);
    return (x+y);
}
```

```
Int main()
{
    int NW = 5000;
    fib(NW);
}
```

并行计算斐波那契数列

```
int fib (int n)
{  int x,y;
   if (n < 2) return n;

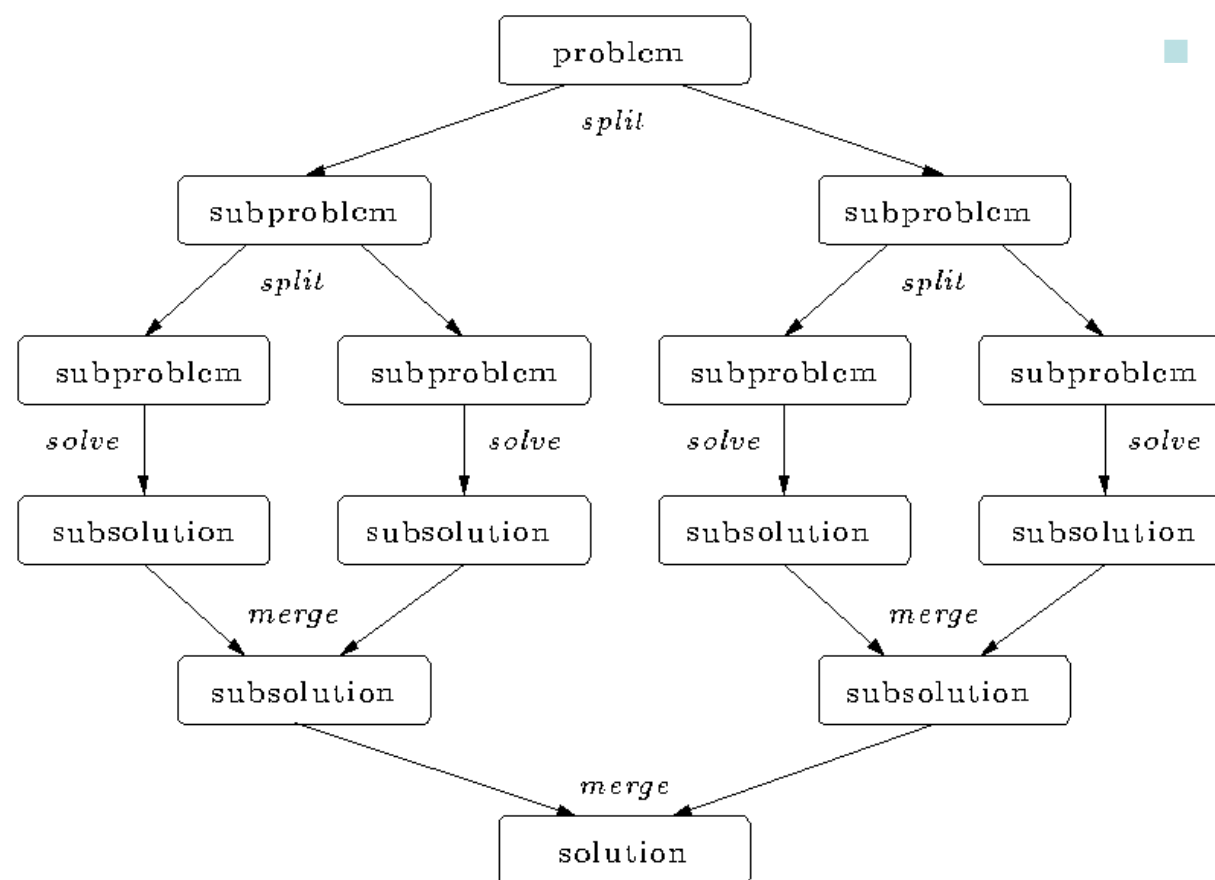
#pragma omp task shared(x)
   x = fib(n-1);
#pragma omp task shared(y)
   y = fib (n-2);
#pragma omp taskwait
   return (x+y);
}

Int main()
{  int NW = 5000;
   #pragma omp parallel
   {
       #pragma omp single
       fib(NW);
   }
}
```

- 二叉树形式组织task
- 使用递归函数遍历
- 在树中子节点(逻辑上的)的所有任务都完成之前，任务无法完成（taskwait带来的作用）
- **x、y**是本地的，因此默认情况下它们对当前任务是私有的
- 但它们必须在子任务上共享，这样就不会在这个级别创建自己的fristprivate副本！

分治法 Divide and conquer

- 将问题分解为更小的子问题；继续，直到可以直接解决子问题



- 执行时机3种选择：
 - 在分解成子问题时进行工作
 - 只在叶子节点处工作
 - 在结合的同时工作

同步

同步用于施加顺序约束并保护对共享数据的访问

■ 高级同步:

- **critical**

- **barrier**

前面介绍过

- atomic

- ordered

■ 低级同步:

- flush

- locks (both simple and nested)

同步: **atomic**

- **Atomic提供互斥，但仅适用于制定内存位置的更新
(以下示例中X的更新)**

```
#pragma omp parallel
```

```
{
```

```
    double B;
```

```
    B = DOIT();
```

```
#pragma omp atomic
```

```
    X += big_ugly(B);
```

```
}
```

同步: `atomic`

- **Atomic提供互斥，但仅适用于制定内存位置的更新（以下示例中X的更新）**

```
#pragma omp parallel
{
    double B, tmp;
    B = DOIT();
    tmp = big_ugly(B);
    #pragma omp atomic
        X += tmp;
}
```

原子操作仅保护X的读写

刷新操作(Flush operation)

- 定义一个序列点，线程在该序列点强制执行内存一致化。
- 其他线程可见并且与刷新操作相关的变量(flush-set)
 - 编译器无法改变flush操作对flush set中变量的读取/写入指令顺序
 - 刷新操作执行过程中，flush set中的变量会被从临时存储器移动到共享存储器
 - 刷新操作完成之后，flush set中变量的读取将从共享内存加载的
- 刷新操作**会让线程的临时“视野”与共享内存中的“视野”相匹配**，而刷新本身不会强制同步

内存一致性：flush的例子

- 刷新强制更新内存中的数据，以便其他线程看到最新的值

```
double A;
```

```
A = compute();
```

```
#pragma omp flush(A)
```

```
// flush to memory to make sure other  
// threads can pick up the right value
```

不带列表的刷新：刷新集包含所有线程可见的变量

使用列表刷新：刷新集是变量的列表
(list of variables)

```
T1: x=1; y=1;
```

```
T2: int r1=y; int r2=x;
```

“如果T2在读取时看到y的值为1，那么接下来读取的x也应该返回值1”，但如果T1的指令是有序，那么会发生什么呢？

注意：OpenMP的flush类似于其他共享内存API中的fence

Flush同步

- **OpenMP同步操作隐式地带有flush操作**
 - 在并行区域的入口/出口处
 - 隐式和显式同步屏障
 - 在临界区的入口/出口
 - 设置或取消设置lock
 - ...
- **但不在worksharing区域的入口或master区域(只有主线程执行)的入口/出口处**

总结

■ 共享内存编程

- 可以在共享区域中分配数据，而不必关心其位置
- 了解内存层次结构对性能调优很重要
 - 由于一致性机制，多处理器下的内存层次结构更重要(相比单处理器)
- 性能调优需要关注共享数据（正常的和false sharing）

■ 语法层面

- 需要对共享变量的访问加锁以进行安全的读写
- 顺序一致性是自然语义
 - 需要编写无数据竞争的程序
- 架构师们为实现OpenMP做出的一些工作
 - 缓存(cache)与总线(bus)或目录(directories)一致
 - 不在共享地址空间的计算机上缓存远程数据(非本地数据)
- 但是编译器和处理器可能仍然会妨碍顺序一致性
 - 非阻塞写入、读取预取、代码移动的存在
 - 避免数据竞争或谨慎使用机器特定的fence操作应对数据竞争

总结

■ 一些CPU多线程库/系统

- PThreads是POSIX Thread的标准库
 - Portable but; relatively heavyweight; low level
- OpenMP适用于应用程序级的并行计算
 - 基于共享内存支持了许多科学计算
- TBB: Thread Building Blocks
 - 英特尔C++模板库，用于并行/多核计算
- CILK: Language of the C “ilk”
 - 集成进C语言的轻量级多线程
- Java threads
 - 建立在POSIX线程之上；Java语言中的对象

THANKS