基于共享内存的并行计算

汤善江

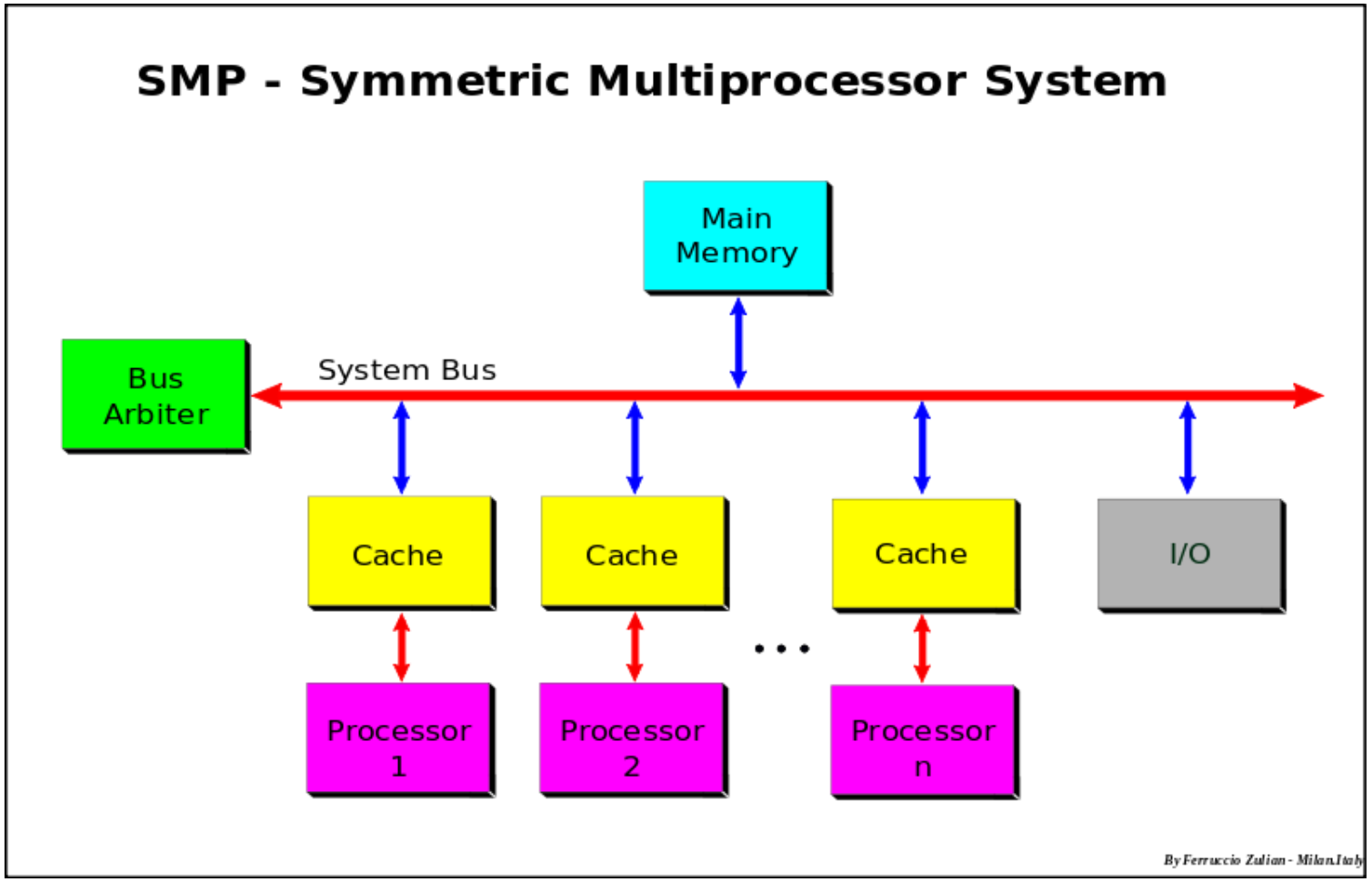
Outline

- 存储访问
- Pthead多线程
- OpenMP

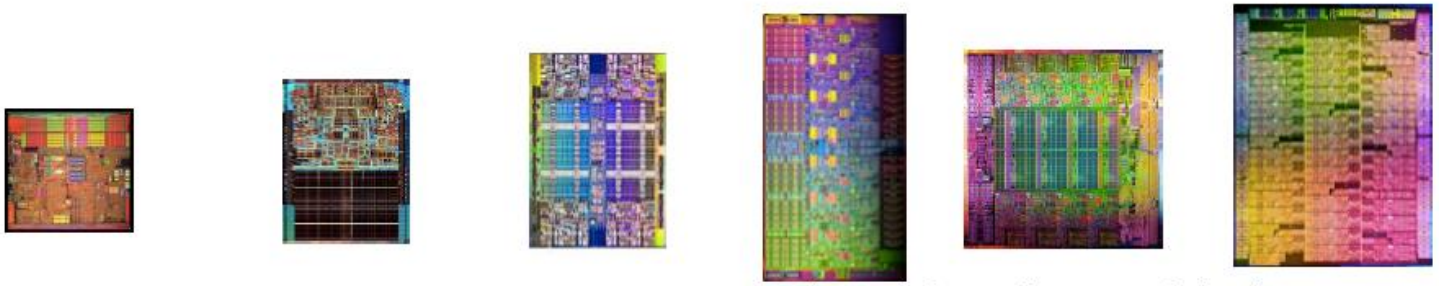
Outline

- **存储访问**
- Pthead多线程
- OpenMP

SMP


SMP - Symmetric Multiprocessor System
Main Memory
Bus Arbiter
System Bus
Cache
Cache
Cache
I/O
Processor 1
Processor 2
Processor n
By Ferruccio Zulian - Milan,Italy

Intel多核与众核

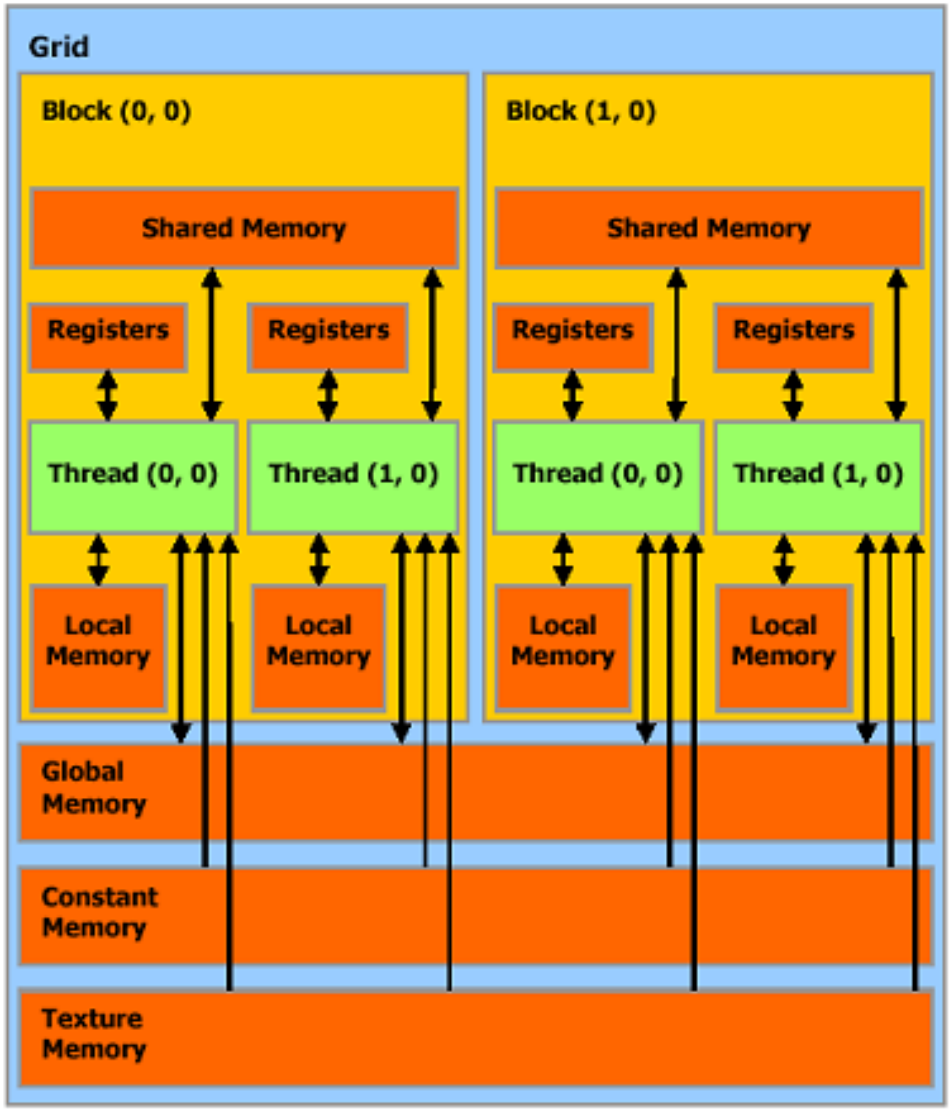


Images not intended to reflect actual die sizes

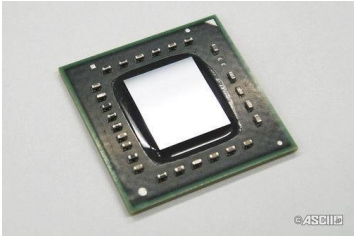
	64-bit Intel® Xeon® processor	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor E5-2600 series	Intel® Xeon Phi™ Co-processor 5110P
Frequency	3.6GHz	3.0GHz	3.2GHz	3.3GHz	2.7GHz	1053MHz
Core(s)	1	2	4	6	8	60
Thread(s)	2	2	8	12	16	240
SIMD width	128 (2 clock)	128 (1 clock)	128 (1 clock)	128 (1 clock)	256 (1 clock)	512 (1 clock)

GPU (Nvidia Kepler)

GPU存储层次


Grid
Block (0, 0)
Block (1, 0)
Shared Memory
Shared Memory
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Local Memory
Local Memory
Local Memory
Local Memory
Global Memory
Constant Memory
Texture Memory

AMD APU



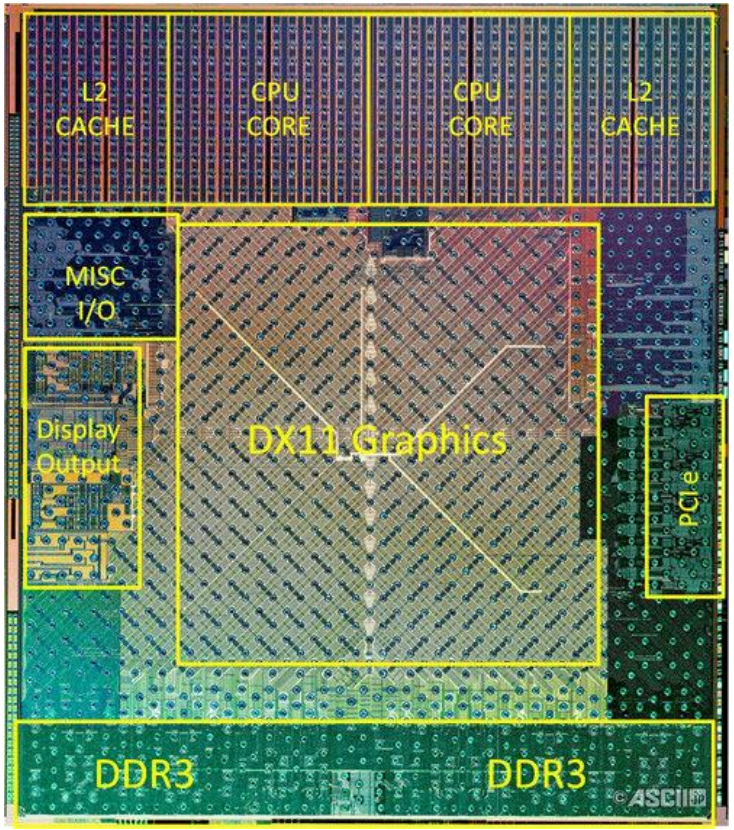
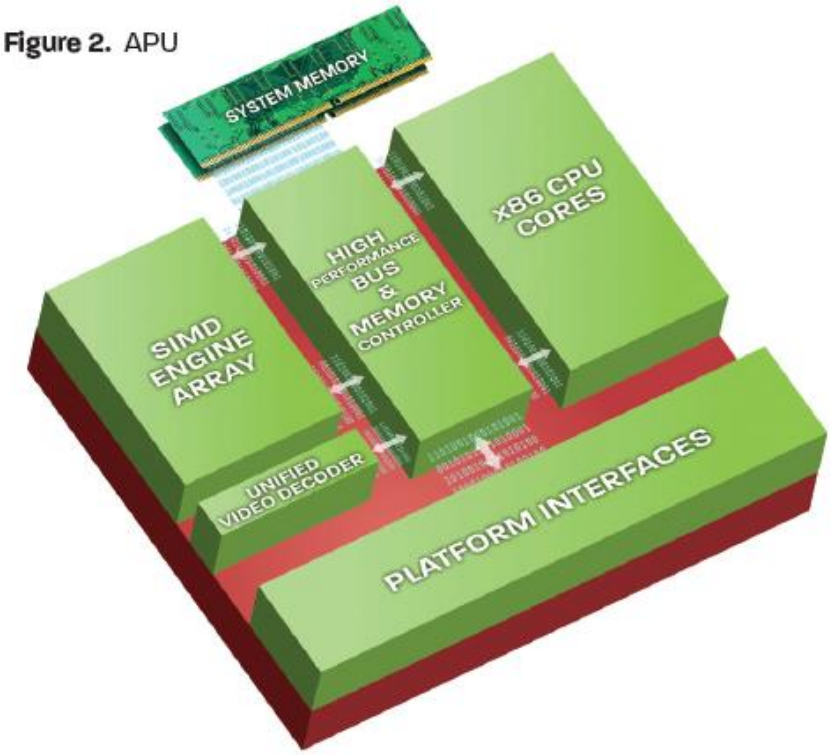
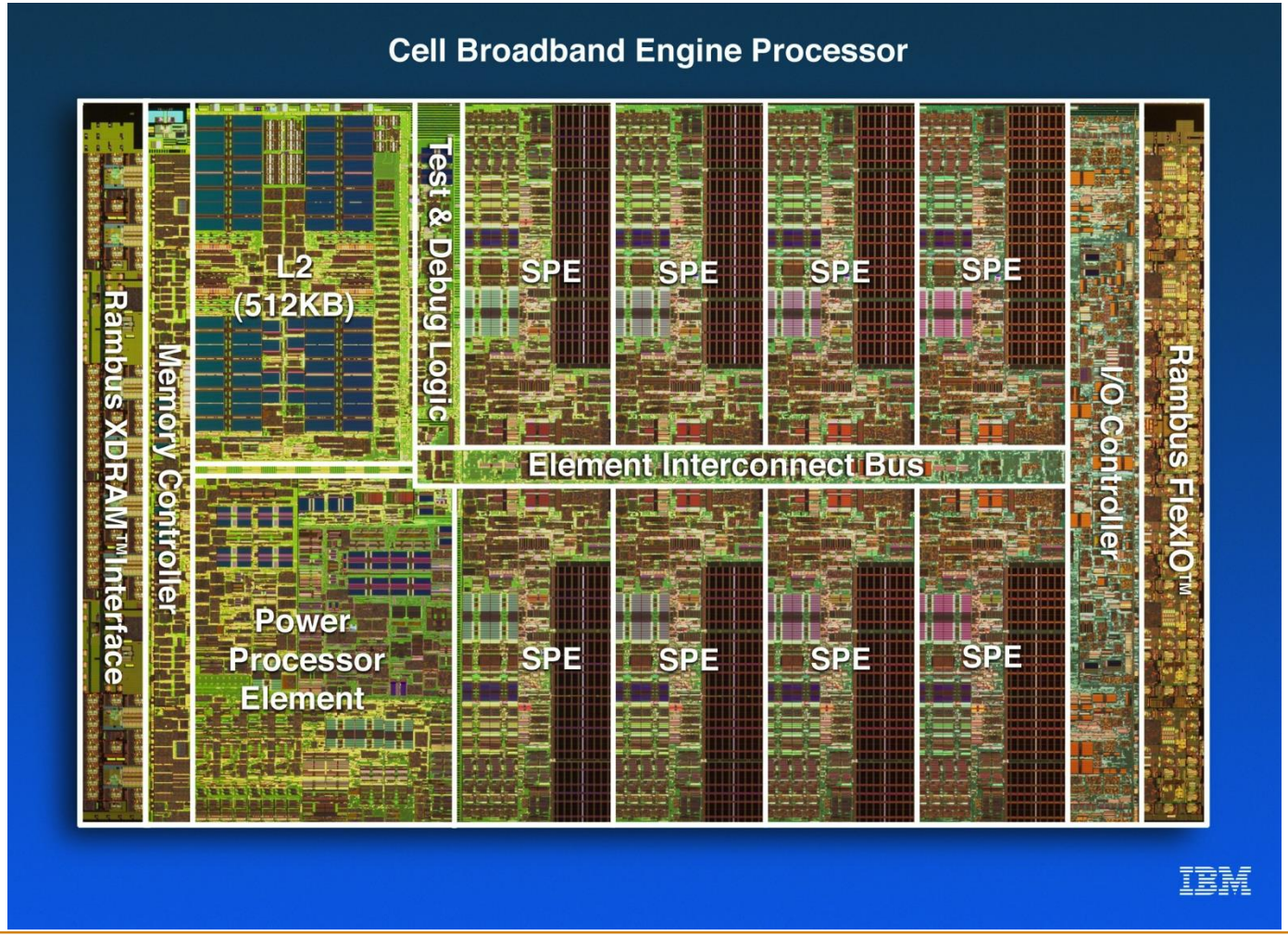


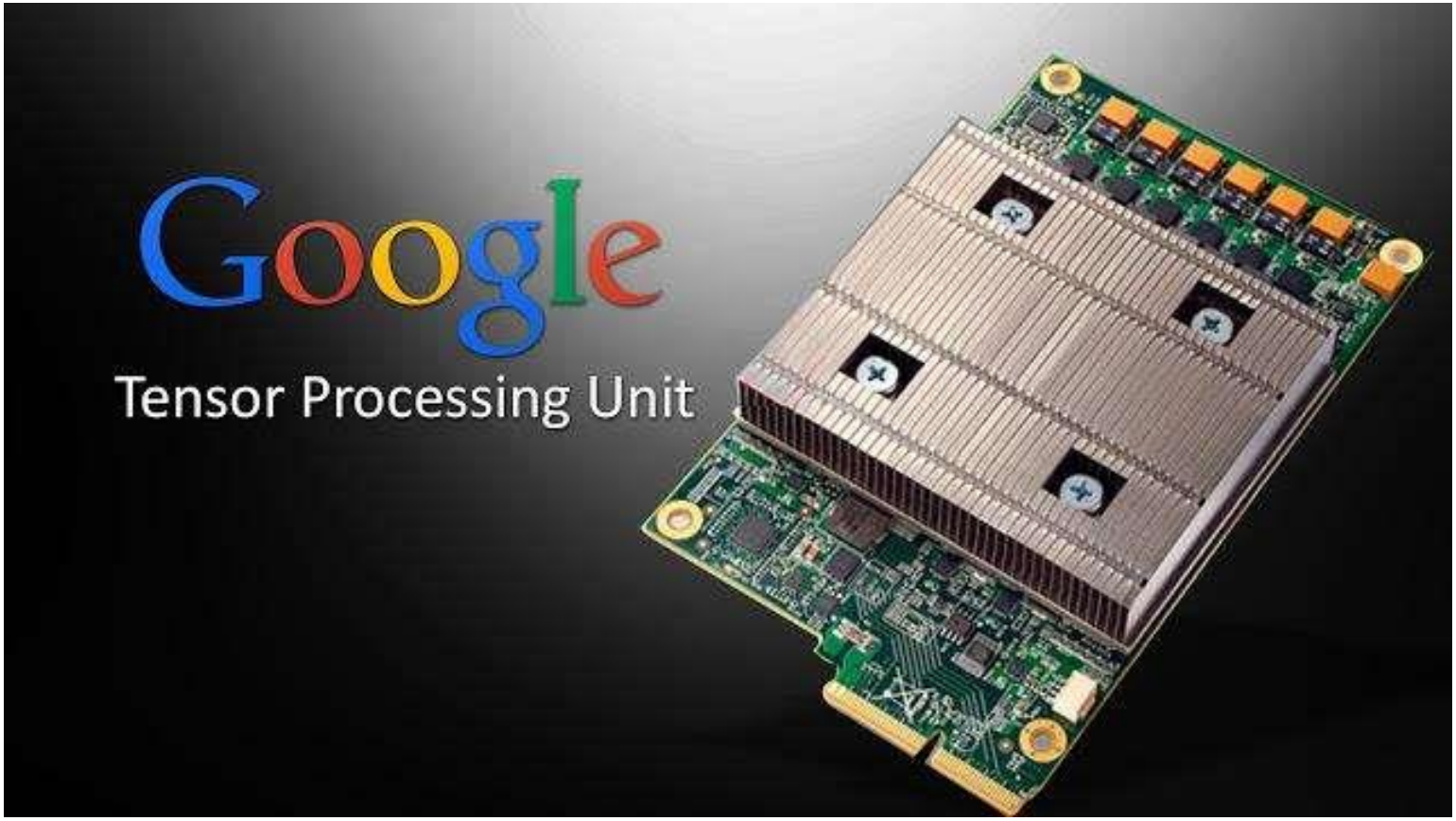
Figure 2. APU



IBM Cell BE


Cell Broadband Engine Processor
Rambus XDRAM™ Interface
Memory Controller
L2
(512KB)
Test & Debug Logic
SPE
SPE
SPE
SPE
I/O controller
Rambus FlexIO™
Element Interconnect Bus
Power
Processor
Element
SPE
SPE
SPE
SPE
IBM

Google TPU



内存系统对性能的影响

- 对于很多应用而言，瓶颈在于内存系统，而不是CPU
- 内存系统的性能包括两个方面：延迟和带宽
  - 延迟：处理器向内存发起访问直至获取数据所需要的时间
  - 带宽：内存系统向处理器传输数据的速率

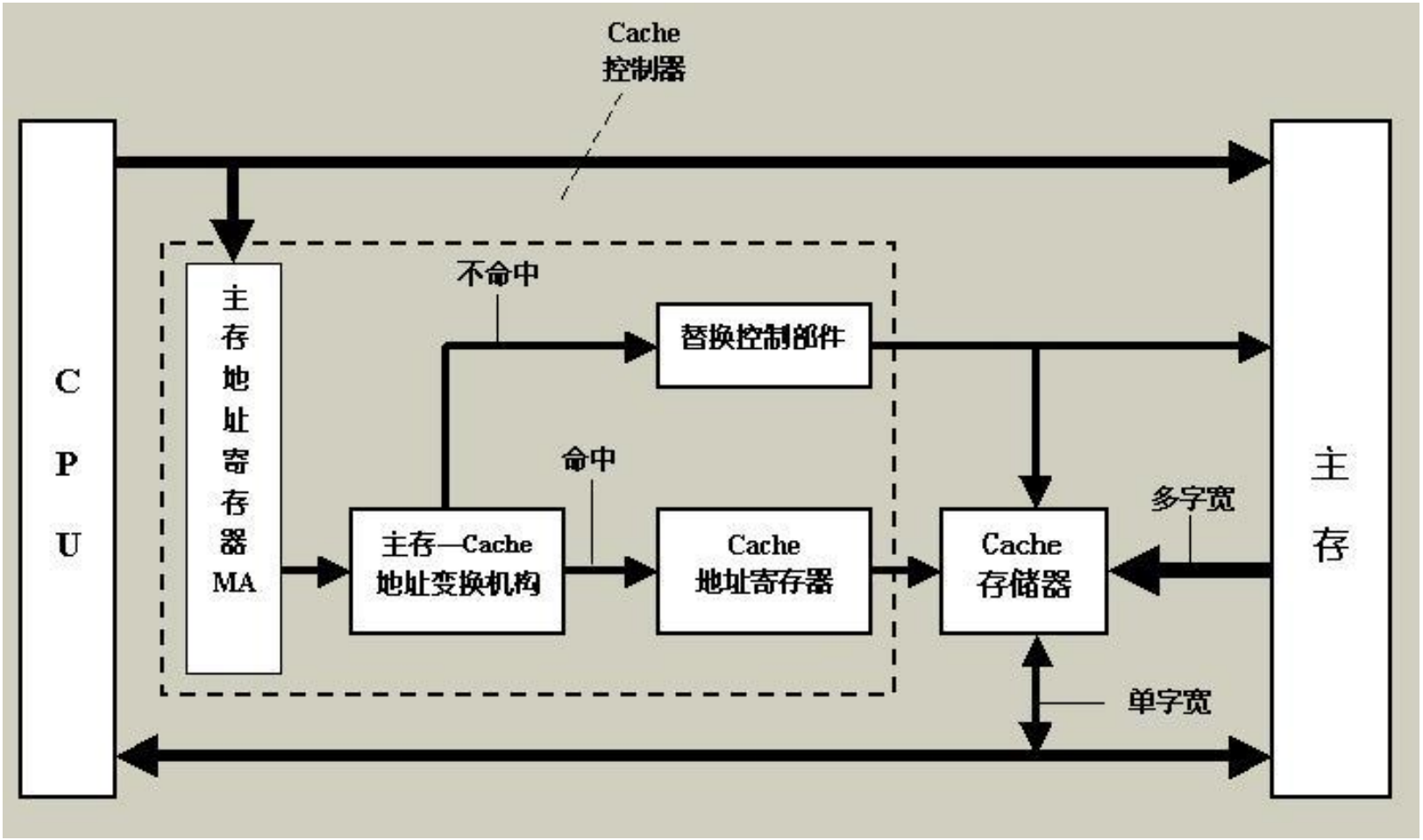
延迟和带宽的区别

- 理解延迟与带宽的区别非常重要。
- 考虑消防龙头的情形。如果打开消防龙头后2秒水才从消防水管的尽头流出，那么这个系统的延迟就是2秒。
- 当水开始流出后，如果水管1秒钟能流出5加仑的水，那么这个水管的“带宽”就是5加仑/秒。
- 如果想立刻扑灭火灾，那么更重要是减少延迟的时间。
- 如果是希望扑灭更大的火，那么需要更高的带宽。

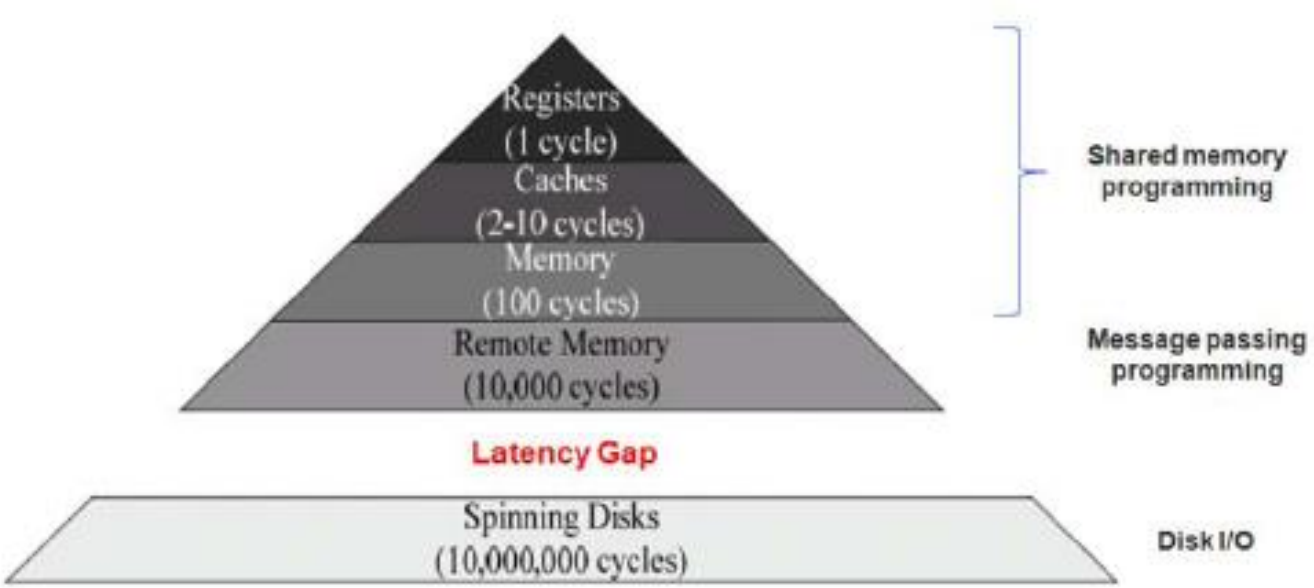
使用高速缓存改善延迟

- 高速缓存是处理器与DRAM之间的更小但更快的内存单元。
- 这种内存是低延迟高带宽的存储器。
- 如果某块数据被重复使用，高速缓存就能减少内存系统的有效延迟
- 由高速缓存提供的数据份额称为高速缓存*命中率*(*hit ratio*)
- 高速缓存命中率严重影响内存受限程序的性能。

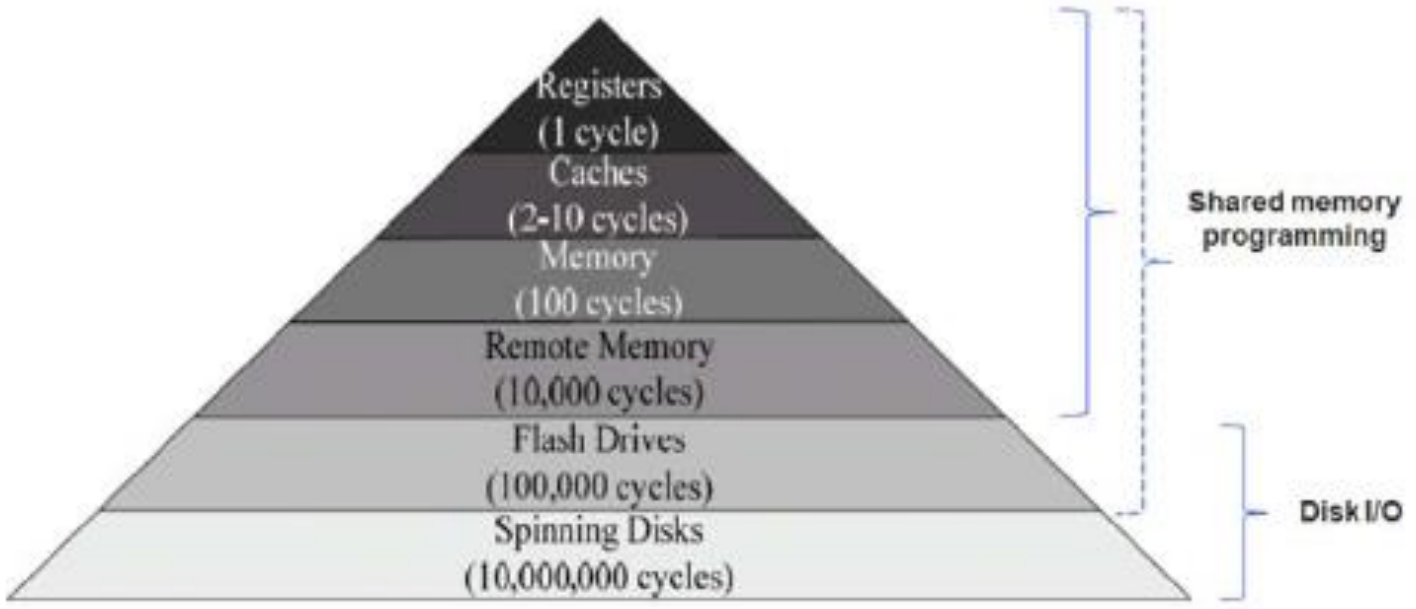
高速缓存


Cache
控制器
C
P
U
主
存
地
址
寄
存
器
MA
不命中
替换控制部件
命中
主存—Cache
地址变换机构
Cache
地址寄存器
Cache
存储器
多字宽
主
存
单字宽

传统计算系统中存储访问层次



增加Flash SSD层

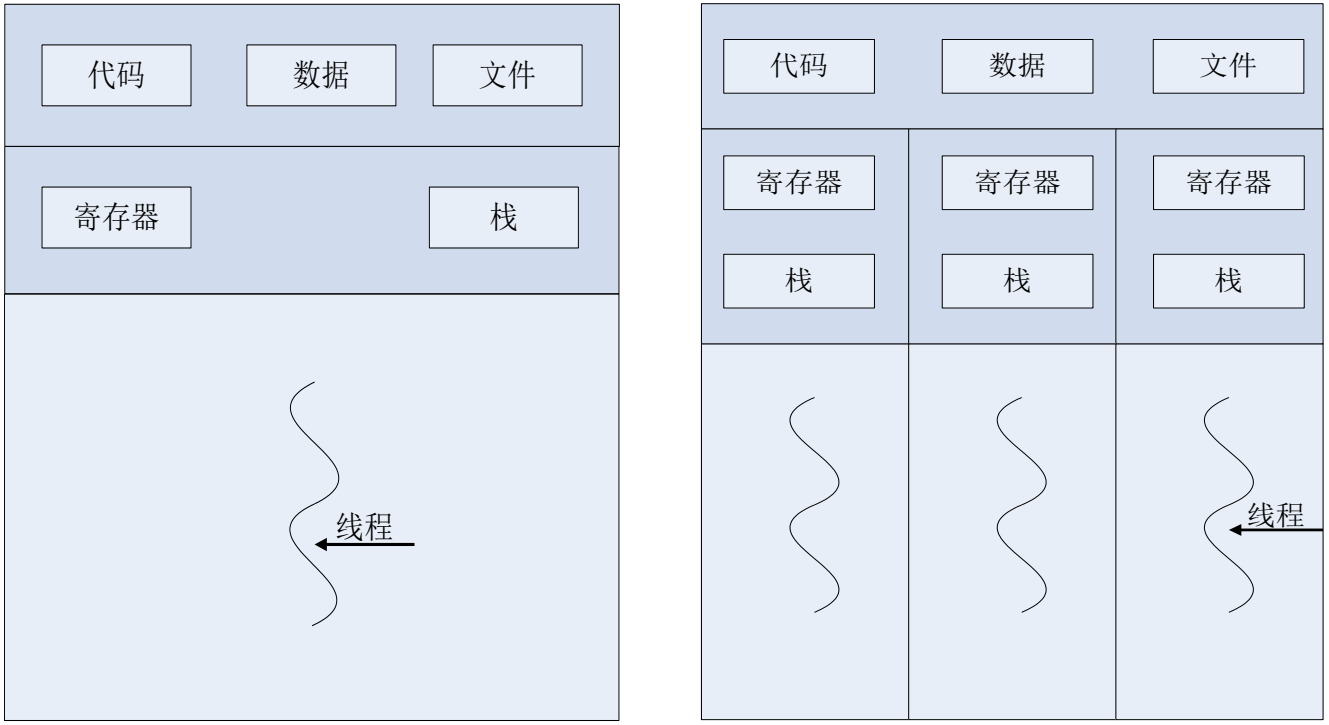


Outline

- 存储访问
- **Pthead多线程**
- OpenMP

多线程概念

- 线程（thread）是进程上下文（context）中执行的代码序列，又被称为轻量级进程（light weight process）
- 在支持多线程的系统中，进程是资源分配的实体，而线程是被调度执行的基本单元。



线程与进程的区别

- 调度
- 并发性
- 拥有资源
- 系统开销

调度

- 在传统的操作系统中，CPU调度和分派的基本单位是进程。
- 在引入线程的操作系统中，则把线程作为CPU 调度和分派的基本单位，进程则作为资源拥有的基本单位，从而使传统进程的两个属性分开，线程便能轻装运行，这样可以显著地提高系统的并发性。
- 同一进程中线程的切换不会引起进程切换，从而避免了昂贵的系统调用。
  - 但是在由一个进程中的线程切换到另一进程中的线程时，依然会引起进程切换。

并发性

- 在引入线程的操作系统中，不仅进程之间可以并发执行，而且在一个进程中的多个线程之间也可以并发执行，因而使操作系统具有更好的并发性，从而能更有效地使用系统资源和提高系统的吞吐量。
  - 例如，在一个未引入线程的单CPU操作系统中，若仅设置一个文件服务进程，当它由于某种原因被封锁时，便没有其他的文件服务进程来提供服务。
- 在引入了线程的操作系统中，可以在一个文件服务进程中设置多个服务线程。
  - 当第一个线程等待时，文件服务进程中的第二个线程可以继续运行；当第二个线程封锁时，第三个线程可以继续执行，从而显著地提高了文件服务的质量以及系统的吞吐量。

拥有资源

- 进程
  - 不论是引入了线程的操作系统，还是传统的操作系统，进程都是拥有系统资源的一个独立单位，它可以拥有自己的资源。
- 线程
  - 线程自己不拥有系统资源（除部分必不可少的资源，如栈和寄存器），但它可以访问其隶属进程的资源。亦即一个进程的代码段、数据段以及系统资源（如已打开的文件、I/O设备等），可供同一进程的其他所有线程共享。

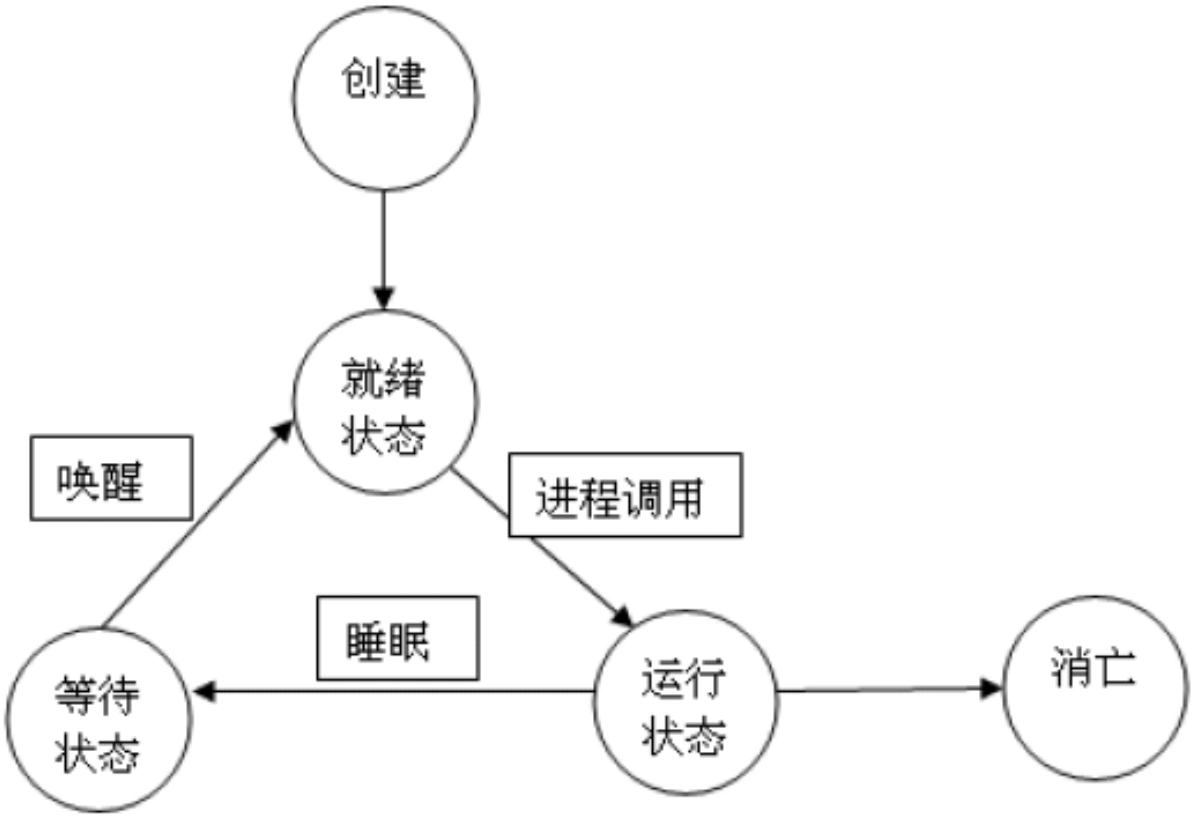
系统开销

- 进程
  - 创建或撤消进程时，系统都要为之分配或回收资源，如内存空间、I/O 设备等。
  - 在进行进程切换时，涉及到整个当前进程CPU 环境的保存环境的设置以及新被调度运行的进程的CPU 环境的设置。
- 线程
  - 切换只需保存和设置少量寄存器的内容，并不涉及存储器管理方面的操作。
  - 此外，由于同一进程中的多个线程具有相同的地址空间，致使它们之间的同步和通信的实现也变得比较容易。在有的系统中，线程的切换、同步和通信都无需操作系统内核的干预。

线程层次

- **用户级线程**在用户层通过线程库来实现。对它的创建、撤销和切换都不利用系统的调用。
- **核心级线程**由操作系统直接支持，即无论是在用户进程中的线程，还是系统进程中的线程，它们的创建、撤消和切换都由核心实现。
- **硬件线程**就是线程在硬件执行资源上的表现形式。
- 单个线程一般都包括上述三个层次的表现：用户级线程通过操作系统被作为核心级线程实现，再通过硬件相应的接口作为硬件线程来执行。

线程的生命周期



POSIX Thread API

- POSIX ： Portable Operating System Interface
- POSIX 是基于UNIX 的，这一标准意在期望获得源代码级的软件可移植性。为一个POSIX 兼容的操作系统编写的程序，应该可以在任何其它的POSIX 操作系统（即使是来自另一个厂商）上编译执行。
- POSIX 标准定义了操作系统应该为应用程序提供的接口：系统调用集。
- POSIX是由IEEE（Institute of Electrical and Electronic Engineering）开发的，并由ANSI（American National Standards Institute）和ISO（International Standards Organization）标准化。

程序示例

```c
#include <pthread.h>

/*
 * The function to be executed by the thread should take a
 * void* parameter and return a void* exit status code.
 */
void *thread_function(void *arg)
{
  // Cast the parameter into what is needed.
  int *incoming = (int *)arg;

  // Do whatever is necessary using *incoming as the argument.

  // The thread terminates when this function returns.
  return NULL;
}

int main(void)
{
  pthread_t thread_ID;
  void      *exit_status;
  int        value;

  // Put something meaningful into value.
  value = 42;

  // Create the thread, passing &value for the argument.
  pthread_create(&thread_ID, NULL, thread_function, &value);

  // The main program continues while the thread executes.

  // Wait for the thread to terminate.
  pthread_join(thread_ID, &exit_status);

  // Only the main thread is running now.
  return 0;
}
```

算法示例：积分法求π

- 公式：

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right)$$

- 串行代码：

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
    sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

线程函数

```
1  void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8
9     if (my_first_i % 2 == 0)  /* my_first_i is even */
10       factor = 1.0;
11    else  /* my_first_i is odd */
12       factor = -1.0;
13
14    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15       sum += factor/(2*i+1);
16    }
17
18    return NULL;
19 }  /* Thread_sum */
```

- 可能的结果：

	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

使用Busy-waiting的线程

```
1   void* Thread_sum(void* rank) {
2      long my_rank = (long) rank;
3      double factor;
4      long long i;
5      long long my_n = n/thread_count;
6      long long my_first_i = my_n*my_rank;
7      long long my_last_i = my_first_i + my_n;
8
9      if (my_first_i % 2 == 0)
10        factor = 1.0;
11     else
12        factor = -1.0;
13
14     for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15        while (flag != my_rank);
16        sum += factor/(2*i+1);
17        flag = (flag+1) % thread_count;
18     }
19
20     return NULL;
21  }  /* Thread_sum */
```

Busy-waiting改进

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor, my_sum = 0.0;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor)
      my_sum += factor/(2*i+1);

   while (flag != my_rank);
   sum += my_sum;
   flag = (flag+1) % thread_count;

   return NULL;
}  /* Thread_sum */
```

Mutex

```
1  void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8     double my_sum = 0.0;
9
10    if (my_first_i % 2 == 0)
11       factor = 1.0;
12    else
13       factor = -1.0;
14
15    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
16       my_sum += factor/(2*i+1);
17    }
18    pthread_mutex_lock(&mutex);
19    sum += my_sum;
20    pthread_mutex_unlock(&mutex);
21
22    return NULL;
23 }  /* Thread_sum */
```

Mutex与Busy-waiting效率比较

Table 4.1 Run-Times (in Seconds) of π Programs Using $n = 10^8$ Terms on a System with Two Four-Core Processors

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

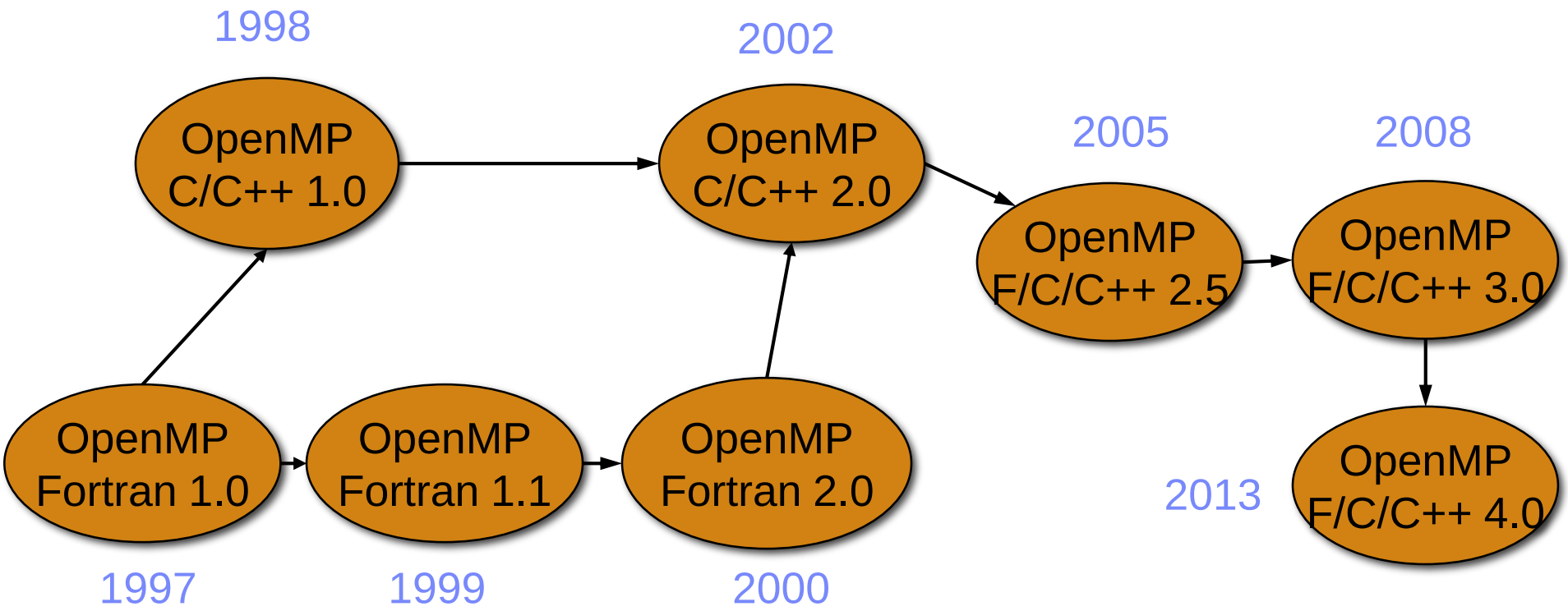
Outline

- 存储访问
- Pthead多线程
- **OpenMP**

OpenMP概述

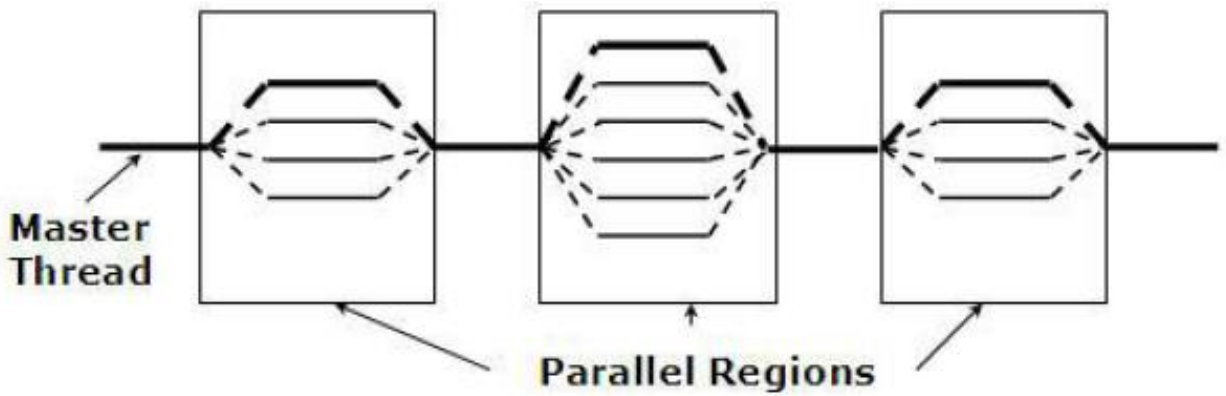
- OpenMP 是一种面向共享内存以及分布式共享内存的多处理器多线程并行编程语言。
- OpenMP是一种能够被用于显式制导多线程、共享内存并行的应用程序编程接口（API）。
- OpenMP标准诞生于1997 年，目前其结构审议委员会（Architecture Review Board, ARB）已经制定并发布OpenMP 4.0 版本。
- www.openmp.org

OpenMP发展历程


1998
2002
2005
2008
OpenMP C/C++ 1.0
OpenMP C/C++ 2.0
OpenMP F/C/C++ 2.5
OpenMP F/C/C++ 3.0
OpenMP Fortran 1.0
OpenMP Fortran 1.1
OpenMP Fortran 2.0
2013
OpenMP F/C/C++ 4.0
1997
1999
2000

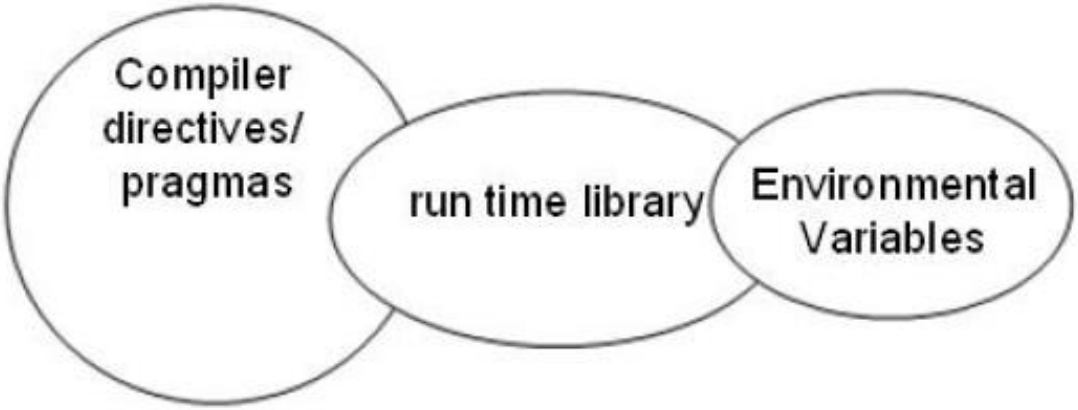
OpenMP编程模型：Fork-Join

- Fork-Join 执行模式在开始执行的时候，只有主线程程存在。主线程在运行过程中，当遇到需要进行并行计算的时候，派生出（Fork）线程来执行并行任务。在并行执行的时候，主线程和派生线程共同工作。在并行代码结束执行后，派生线程退出或者挂起，不再工作，控制流程回到单独的主线程中（Join）。



OpenMP的实现

- 编译制导语句
- 运行时库函数
- 环境变量



编译制导语句（Compiler Directive）

- 并行域
- 共享任务
- 同步

编译制导语句（Compiler Directive）

- 编译制导语句的含义是在编译器编译程序的时候，会识别特定的注释，而这些特定的注释就包含着OpenMP 程序的一些语义。
  - 在C/C++程序中，用#pragma omp parallel 来标识一段并行程序块。在一个无法识别OpenMP 语义的普通编译器中，这些特定的注释会被当作普通的注释而被忽略。

```
#pragma omp <directive> [clause[ [,] clause]…]
```

编译制导语句（Compiler Directive）

将循环拆分到多个线程执行

```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

串行代码

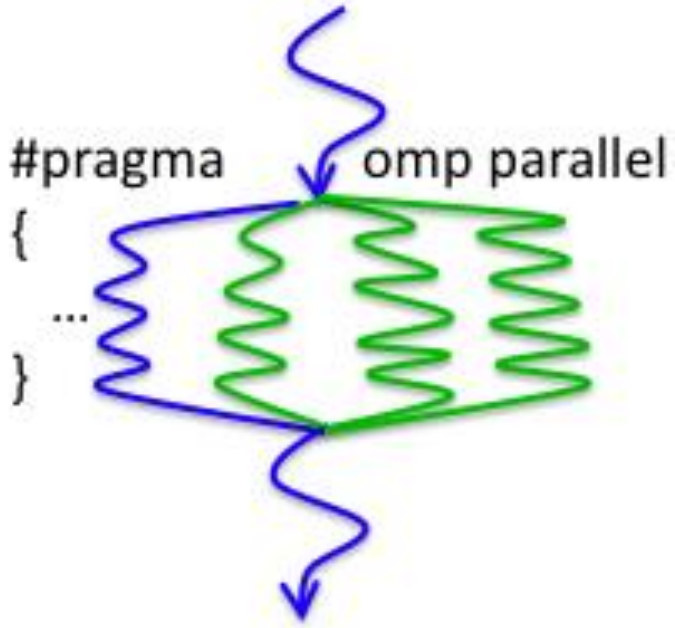
```
#include “omp.h”
void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

并行代码

编译制导语句（Compiler Directive）

- **并行域**
- 共享任务
- 同步

并行域（parallel region）



并行域

- 并行域中的代码被所有的线程执行
- 具体格式
  - #pragma omp parallel [clause[[,]clause]…]newline
  - clause=
    - **if(***scalar-expression***)**
    - **private(***list***)**
    - **firstprivate(***list***)**
    - **default(shared | none)**
    - **shared(***list***)**
    - **copyin(***list***)**
    - **reduction(***operator***: *list*)**
    - **num_threads(***integer-expression***)**

并行域示例

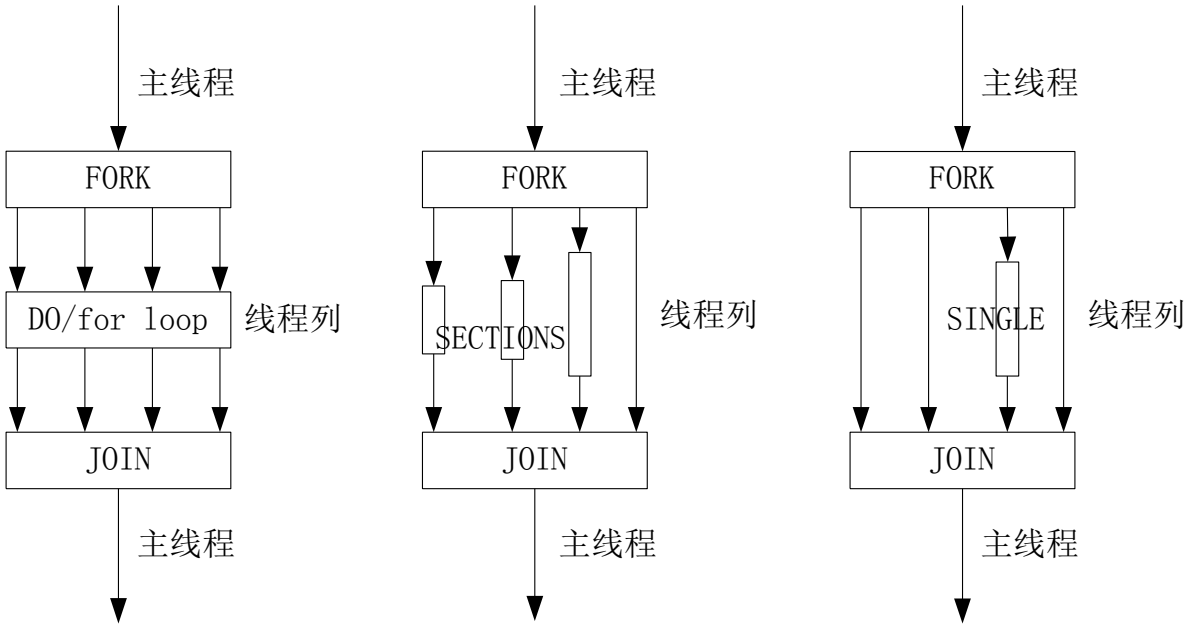
```
#include <omp.h>

main ()  {
int nthreads, tid;

/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel private(tid)  {

  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

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

  }  /* All threads join master thread and terminate */
}
```

编译制导语句

- 并行域
- **共享任务**
- 同步

共享任务

- 共享任务结构将它所包含的代码划分给线程组的各成员来执行
  - 并行for循环
  - 并行sections
  - 串行执行



for编译制导语句

- for语句指定紧随它的循环语句必须由线程组并行执行；
- 语句格式
  - #pragma omp for [clause[[,]clause]…] newline
  - [clause]=
    - Schedule(type [,chunk])
    - ordered
    - private (list)
    - firstprivate (list)
    - lastprivate (list)
    - shared (list)
    - reduction (operator: list)
    - nowait

for编译制导语句

- schedule子句描述如何将循环的迭代划分给线程组中的线程
- 如果没有指定chunk大小，迭代会尽可能的平均分配给每个线程
- type为static，循环被分成大小为 chunk的块，静态分配给线程
- type为dynamic,循环被动态划分为大小为chunk的块，动态分配给线程

for示例

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000

main ()  {
int i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
  {
  #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
       c[i] = a[i] + b[i];
  }  /* end of parallel section */
}
```

Sections编译制导语句

- sections编译制导语句指定内部的代码被划分给线程组中的各线程
- 不同的section由不同的线程执行
- Section语句格式：

```
#pragma omp sections [ clause[[,]clause]…] newline
{
[#pragma omp section newline]
        …
[#pragma omp section newline]
        …
}
```

Sections编译制导语句

- clause=
  - private (list)
  - firstprivate (list)
  - lastprivate (list)
  - reduction (operator: list)
  - nowait

- 在sections语句结束处有一个隐含的路障，使用了nowait子句除外

Sections编译制导语句

```
#include <omp.h>
#define N     1000
main (){
int i;
float a[N], b[N], c[N], d[N];
/* Some initializations */
for (i=0; i < N; i++)
  a[i] = i * 1.5;
  b[i] = i + 22.35;
#pragma omp parallel shared(a,b,c,d) private(i) {
      #pragma omp sections nowait {
            #pragma omp section
                  for (i=0; i < N; i++)
                        c[i] = a[i] + b[i];
            #pragma omp section
                  for (i=0; i < N; i++)
                        d[i] = a[i] * b[i];
    }  /* end of sections */
  }  /* end of parallel section */
}
```

single编译制导语句

- single编译制导语句指定内部代码只有线程组中的一个线程执行。
- 线程组中没有执行single语句的线程会一直等待代码块的结束，使用nowait子句除外
- 语句格式：
  - #pragma omp single [clause[[,]clause]…] newline
  - clause=
    - private(list)
    - firstprivate(list)
    - nowait

single示例

```
#include <stdio.h>

void work1() {}
void work2() {}

void a12()
{
  #pragma omp parallel
  {
    #pragma omp single
      printf("Beginning work1.\n");

    work1();

    #pragma omp single
      printf("Finishing work1.\n");

    #pragma omp single nowait
      printf("Finished work1 and beginning work2.\n");

    work2();
  }
}
```

parallel for编译制导语句

- Parallel for编译制导语句表明一个并行域包含一个独立的for语句
- 语句格式
  - #pragma omp parallel for [clause…] newline
  - clause=
    - if (scalar_logical_expression)
    - default (shared | none)
    - schedule (type [,chunk])
    - shared (list)
    - private (list)
    - firstprivate (list)
    - lastprivate (list)
    - reduction (operator: list)
    - copyin (list)

parallel for编译制导语句

```
#include <omp.h>
#define N       1000
#define CHUNKSIZE   100
int main ()
{
  int i, chunk;
  float a[N], b[N], c[N];
  /* Some initializations */
  for (i=0; i < N; i++)
            a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;
  #pragma omp parallel for shared(a,b,c,chunk) private(i)
  schedule(static,chunk)
            for (i=0; i < n; i++)
                        c[i] = a[i] + b[i];
}
```

parallel sections编译制导语句

- parallel sections编译制导语句表明一个并行域包含单独的一个sections语句
- 语句格式
  - #pragma omp parallel sections [clause…] newline
  - clause=
    - default (shared | none)
    - shared (list)
    - private (list)
    - firstprivate (list)
    - lastprivate (list)
    - reduction (operator: list)
    - copyin (list)
    - ordered

parallel sections 示例

```
void XAXIS();
void YAXIS();
void ZAXIS();

void a11()
{
  #pragma omp parallel sections
  {
    #pragma omp section
      XAXIS();

    #pragma omp section
      YAXIS();

    #pragma omp section
      ZAXIS();
  }
}
```

编译制导语句

- 并行域
- 共享任务
- **同步**

同步

- master 制导语句
- critical制导语句
- barrier制导语句

master 制导语句

- master制导语句指定代码段只有主线程执行
- 语句格式
  - #pragma omp master newline

critical制导语句

- critical制导语句表明域中的代码一次只能执行一个线程
- 其他线程被阻塞在临界区
- 语句格式：
  - #pragma omp critical [name] newline

critical制导语句

```
int dequeue(float *a);
void work(int i, float *a);

void a16(float *x, float *y)
{
  int ix_next, iy_next;

  #pragma omp parallel shared(x, y) private(ix_next, iy_next)
  {
    #pragma omp critical (xaxis)
      ix_next = dequeue(x);
    work(ix_next, x);

    #pragma omp critical (yaxis)
      iy_next = dequeue(y);
    work(iy_next, y);
  }

}
```

barrier制导语句

- barrier制导语句用来同步一个线程组中所有的线程
- 先到达的线程在此阻塞，等待其他线程
- barrier语句最小代码必须是一个结构化的块
- 语句格式
  - #pragma omp barrier newline

运行库例程与环境变量

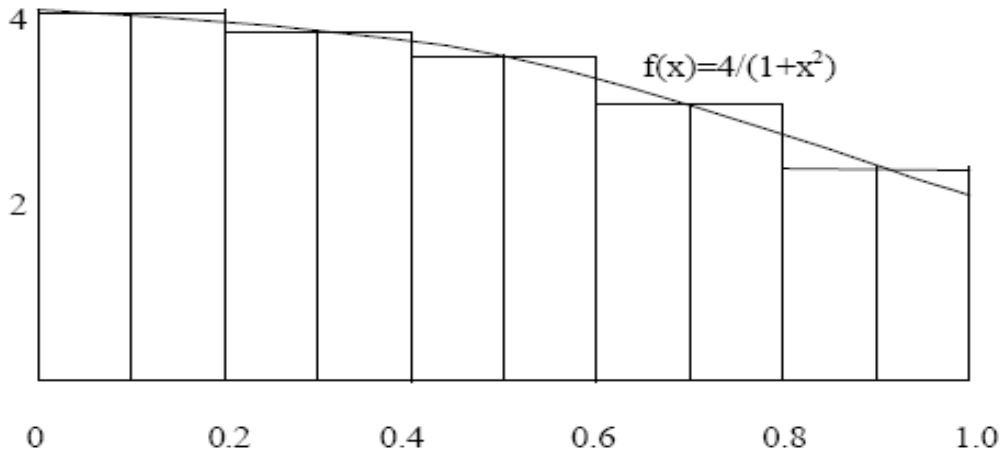
- 运行库例程
  - OpenMP标准定义了一个应用编程接口来调用库中的多种函数
  - 对于C/C++，在程序开头需要引用文件“omp.h”

- 环境变量
  - OMP_SCHEDULE：线程调度类型，只能用到for, parallel for中
  - OMP_NUM_THREADS：定义执行中最大的线程数
  - OMP_DYNAMIC：通过设定变量值TRUE或FALSE,来确定是否动态设定并行域执行的线程数
  - OMP_NESTED：确定是否可以并行嵌套

OpenMP计算实例

- 矩形法则的数值积分方法估算Pi的值

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



OpenMP计算实例

```
//串行代码
static long num_steps = 100000;
double step;
void 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;
}
```

```
//使用并行域并行化的程序
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{   int i;
    double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS); //
   #pragma omp parallel
    {
            double x;
            int id;
            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;
}
```

```
//使用共享任务结构并行化的程序
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
    int i;
    double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS) ;//******
    #pragma omp parallel //******
    {
            double x;
            int 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;
}
```

```
//使用private子句和critical部分并行化的程序
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
  int i;
  double x, sum, pi=0.0;
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS)
  #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
  }
}
```

```
//使用并行归约得出的并行程序
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{   int i;
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS)
    #pragma omp parallel for reduction(+:sum) private(x)
    for (i=0;i<num_steps; i++){
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```