基于共享内存的并行计算

汤善江

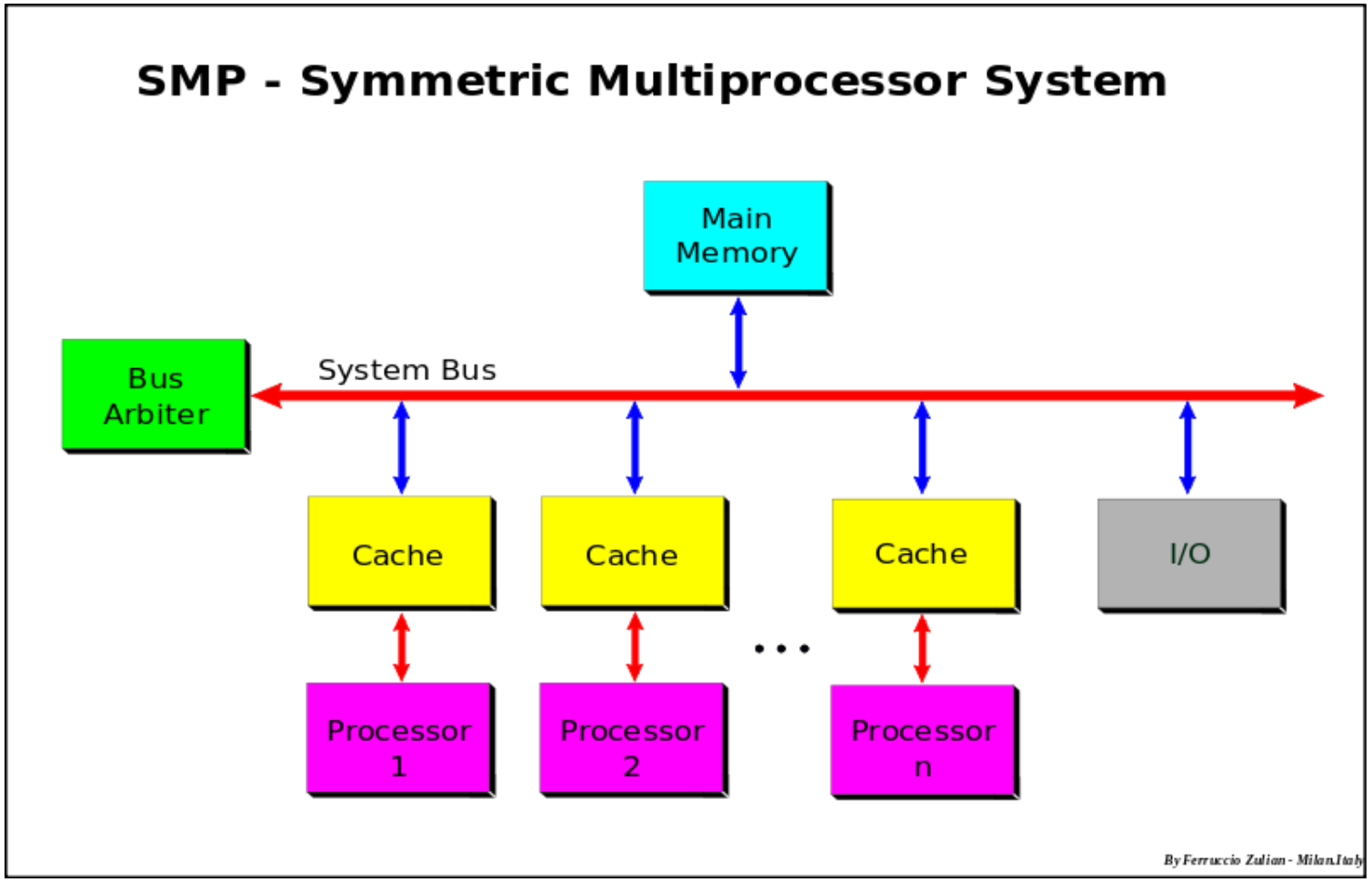
Outline

- 存储访问
- Pthead多线程
- OpenMP

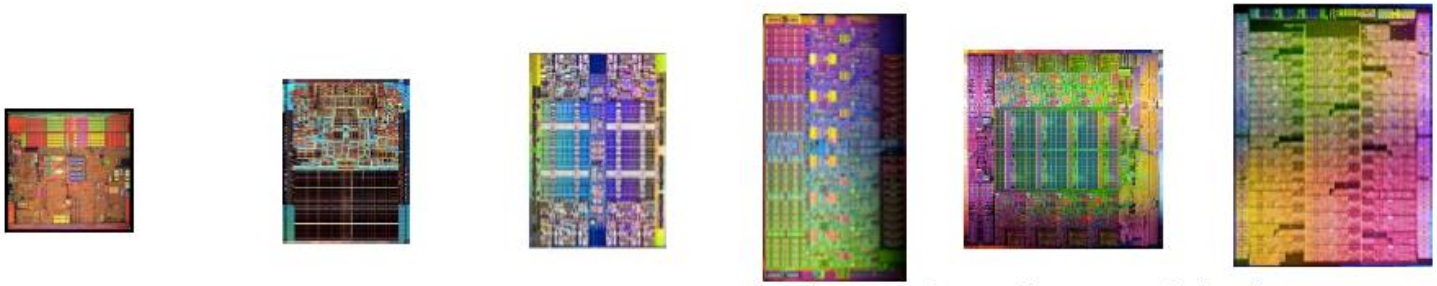
Outline

- **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

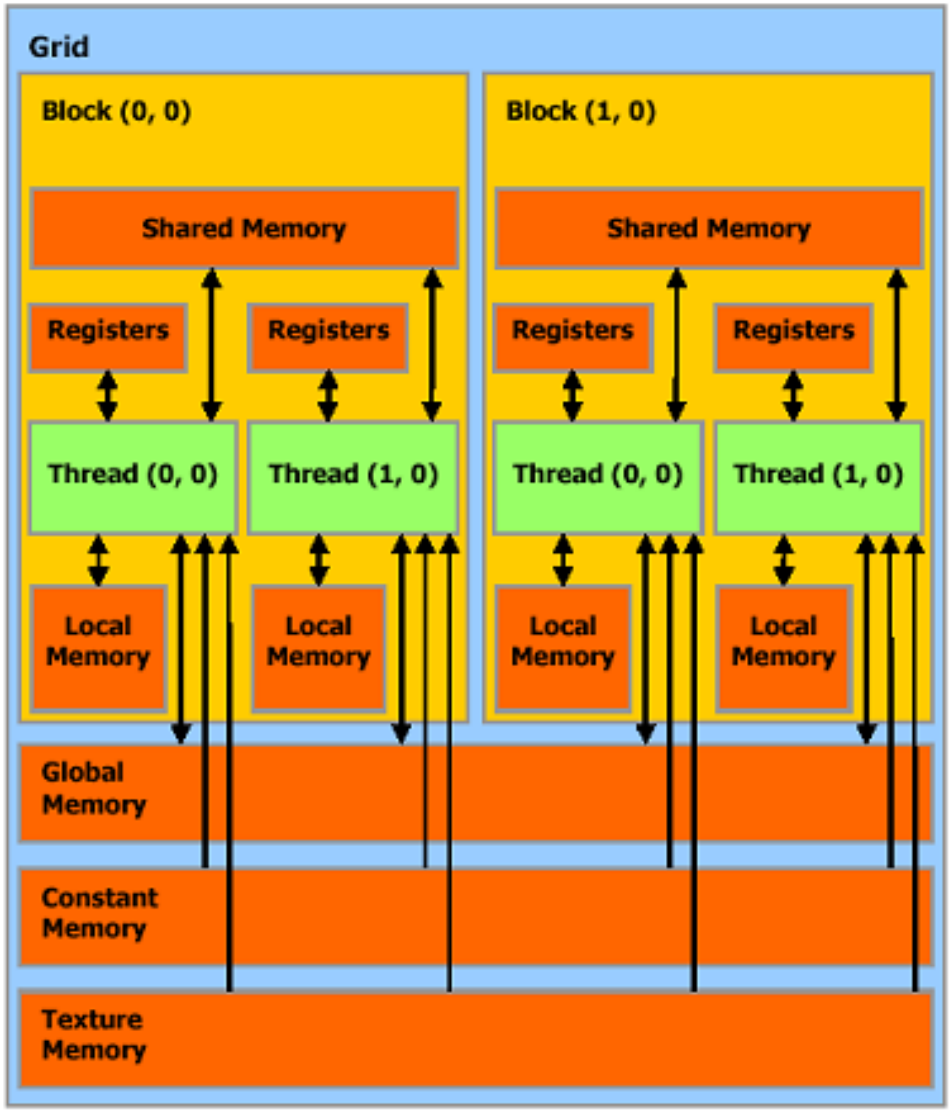


Images not intended to reflect actual die sizes

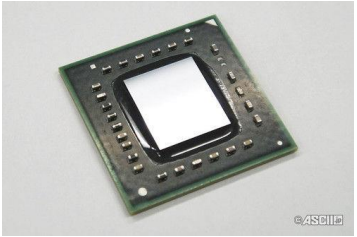
	64-bit Intel® Xeon® processor	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor E5-2600 series	Intel® Xeon Phi™ Co-processor 5110P
Frequency	3.6GHz	3.0GHz	3.2GHz	3.3GHz	2.7GHz	1053MHz
Core(s)	1	2	4	6	8	60
Thread(s)	2	2	8	12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



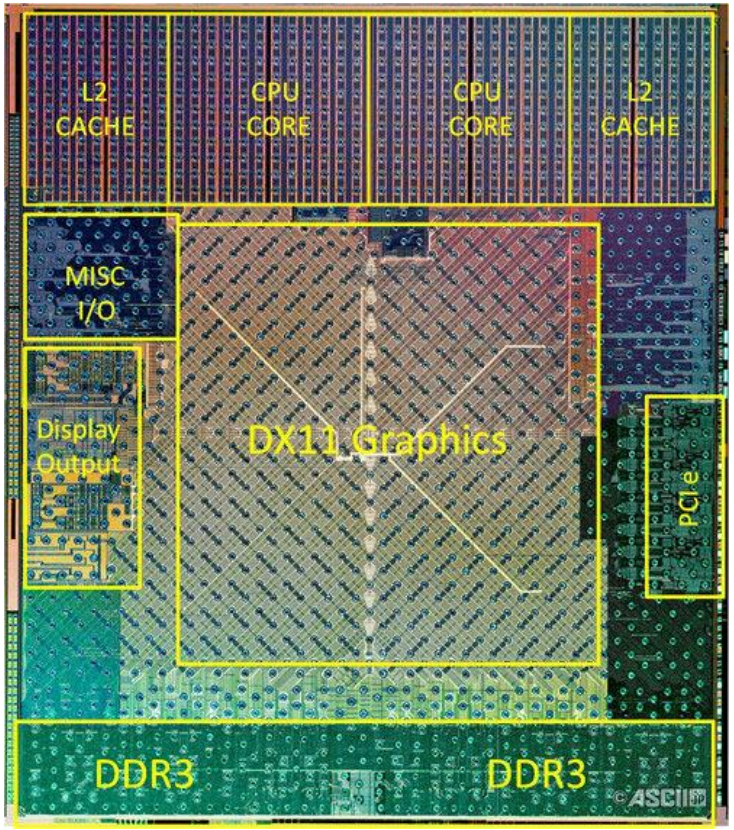
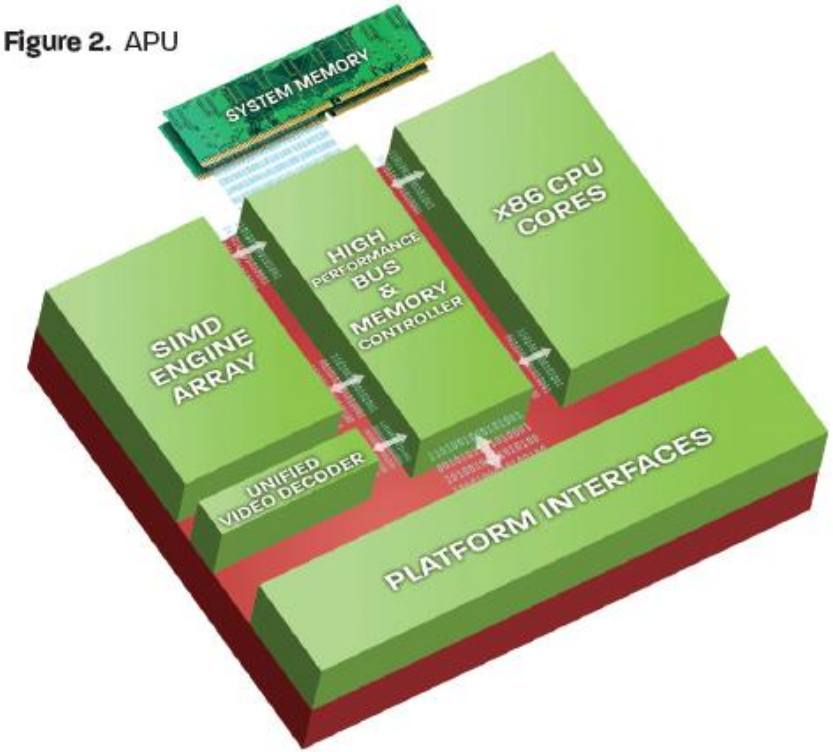
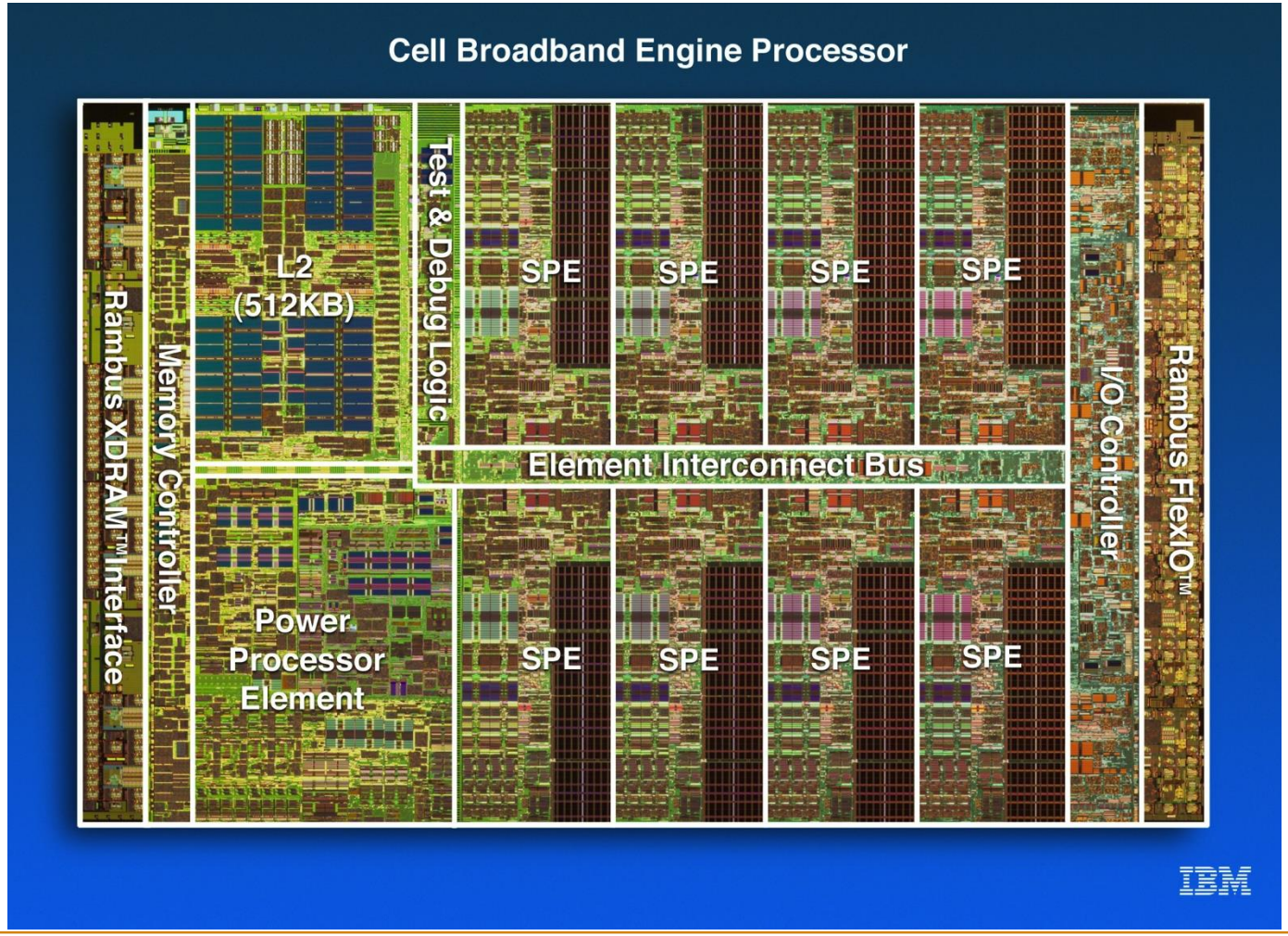


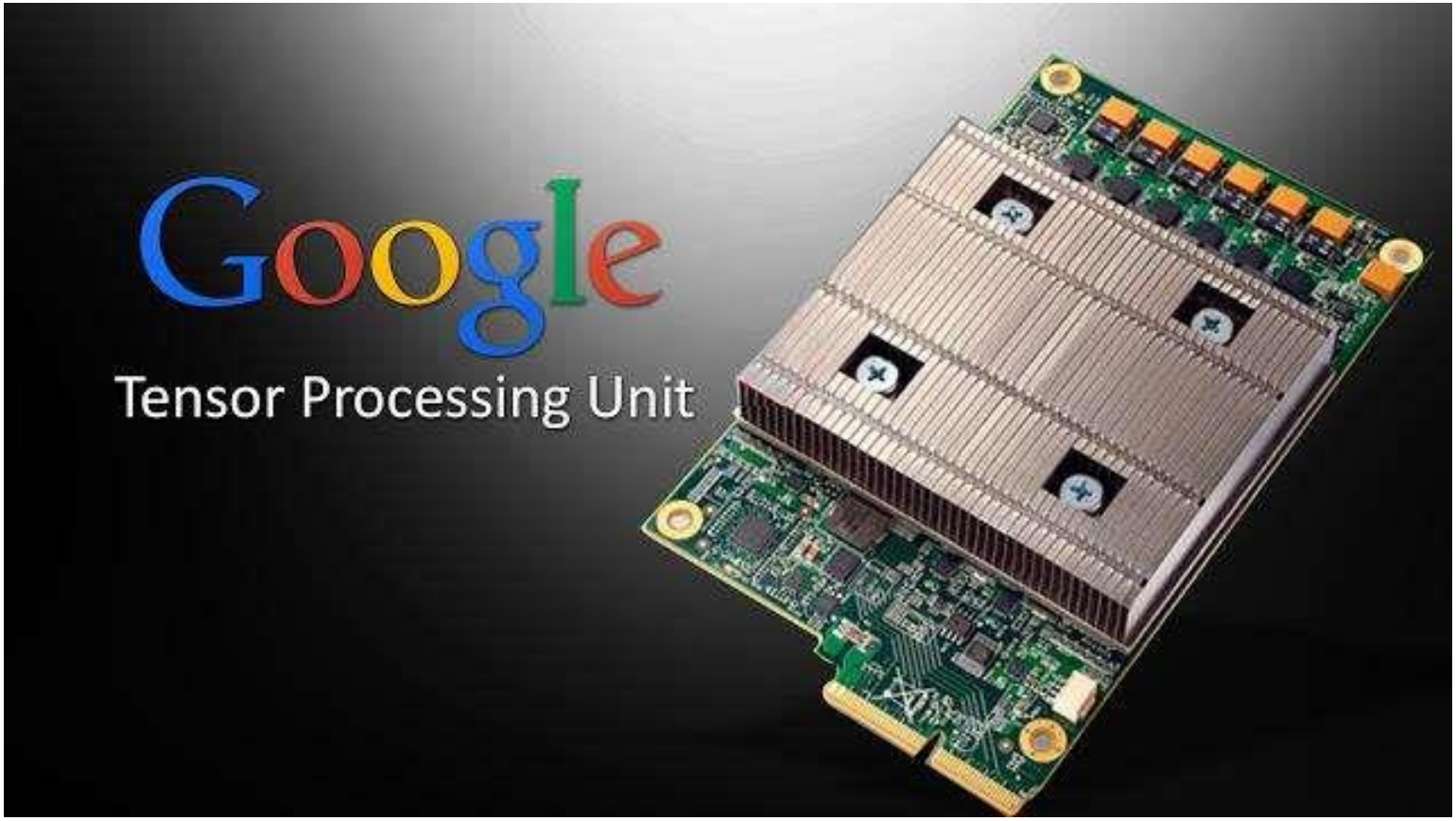
Figure 2. APU



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。

内存延迟示例

- 考虑某一处理器以1GHz（1纳秒时钟）运行，与之相连的DRAM有100纳秒的延迟（没有高速缓存）。假设处理器有两个multiply-add部件，在每1纳秒的周期内能执行4条指令。
  - 处理器的峰值是4GFLOPS。
  - 由于内存延迟是100个周期，并且块大小为一个字（word），每次处理内存访问请求时，处理器必须要等待100个周期，才能够获得数据。

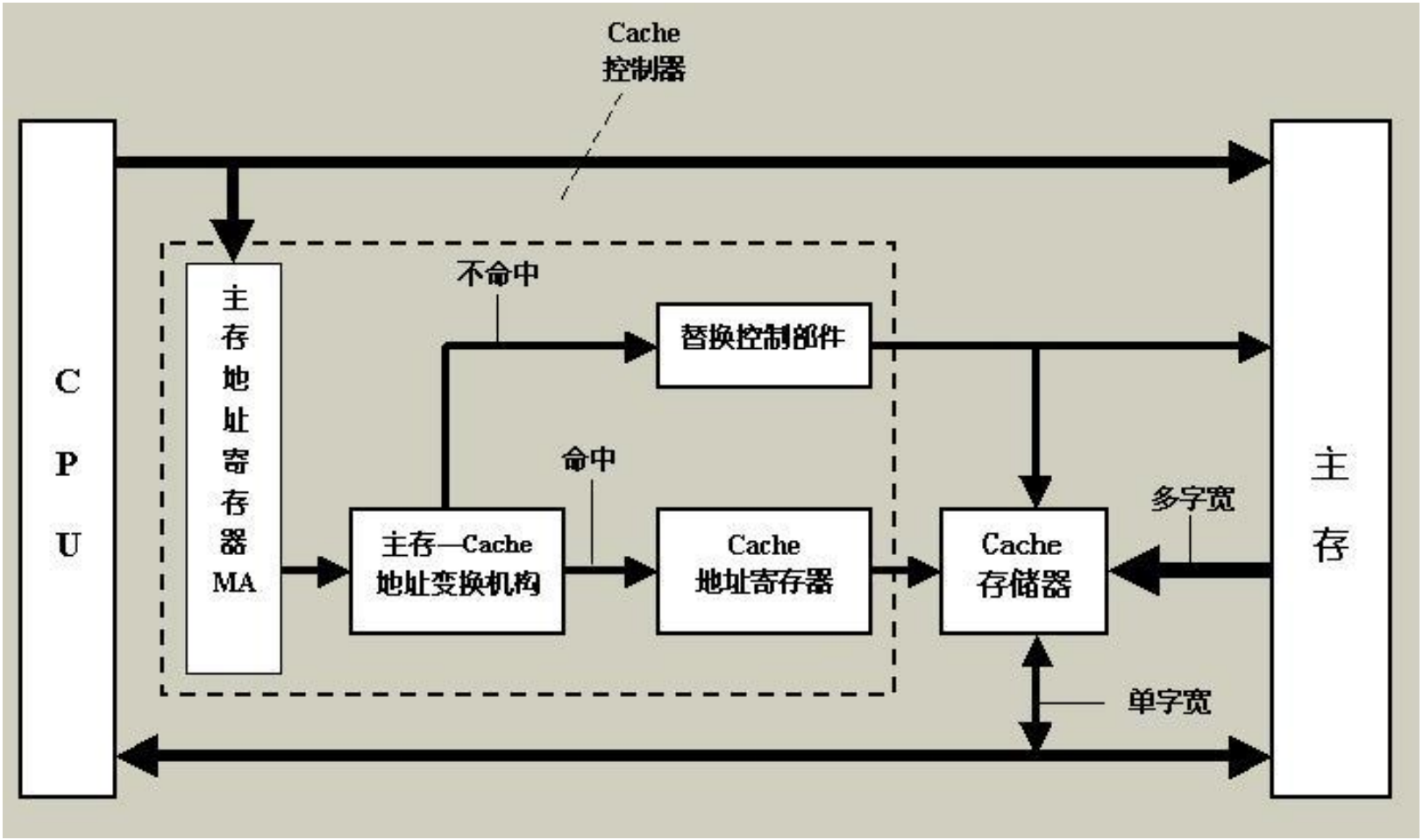
内存延迟示例

- 在以上平台上，考虑计算两个向量点积的问题。
  - 计算点积对每对向量元素进行一次乘法-加法运算，即每一次浮点运算需要取一次数据。
  - 此计算的峰值速度的限制是，每100纳秒才能够进行一次浮点计算，速度为10MFLOPS，只是处理器峰值速度的很小一部分。

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
CPU
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

缓存效果示例

- 继续考虑前一示例。
- 在其中加入一个大小为32KB，延迟时间为1纳秒(或1个周期)的高速缓存。
- 使用此系统来计算矩阵乘法，两个矩阵A和B的维数为 32×32。
  - 之所以选择这个大小，是为了能够将A、B两个矩阵都放入高速缓存中。

缓存效果示例

- 结果如下
  - 将两个矩阵取到高速缓存中等同于取2K个字，需要大约200 μs。
  - 两个$n \times n$ 的矩阵乘需要$2n^3$ 步计算。在本例中，需要64K步计算，如果每个周期执行4条指令，则需要16K个周期，即 16 μs。
  - 总计算时间大约是加载存储时间以及计算时间之和，即200 + 16 μs。
  - 峰值计算速度为64K/216 =303 MFLOPS。

缓存的效果

- 对相同数据项的重复引用相当于“时间本地性(temporal locality)”
- 对于高速缓存的性能来说，数据的重复使用至关重要。

内存带宽的影响

- 内存带宽由内存总线的带宽和内存部件决定。
  - 可以通过增加内存块的大小来提高带宽。
- 底层系统在 L时间单位内(L为系统的延迟)存取B单位的数据(B为块大小)

内存带宽的影响示例

- 继续上一示例，将块大小由1个字改为4个字。同样考虑点积计算：
  - 假定向量数据在内存中线性排列，则在200个周期内能够执行8FLOPs(4次乘法-加法)
  - 这是因为每一次内存访问取出向量中4个连续的字
  - 因此，两次连续访问能够取出每个向量中的4个元素。
  - 这就相当于每25ns执行一次FLOP，即峰值速度为40MFLOPS。

内存带宽的影响

- 增加块的大小，并不能改变系统的延迟。
- 物理上讲，本例中的情形可以认为是与多个存储区相连接的宽的数据总线(4个字，或者128位)
- 实际上，构建这样的宽总线的代价昂贵。
- 在更切实可行的系统中，得到第一个字后，连续的字在紧接着的总线周期里被送到内存总线。

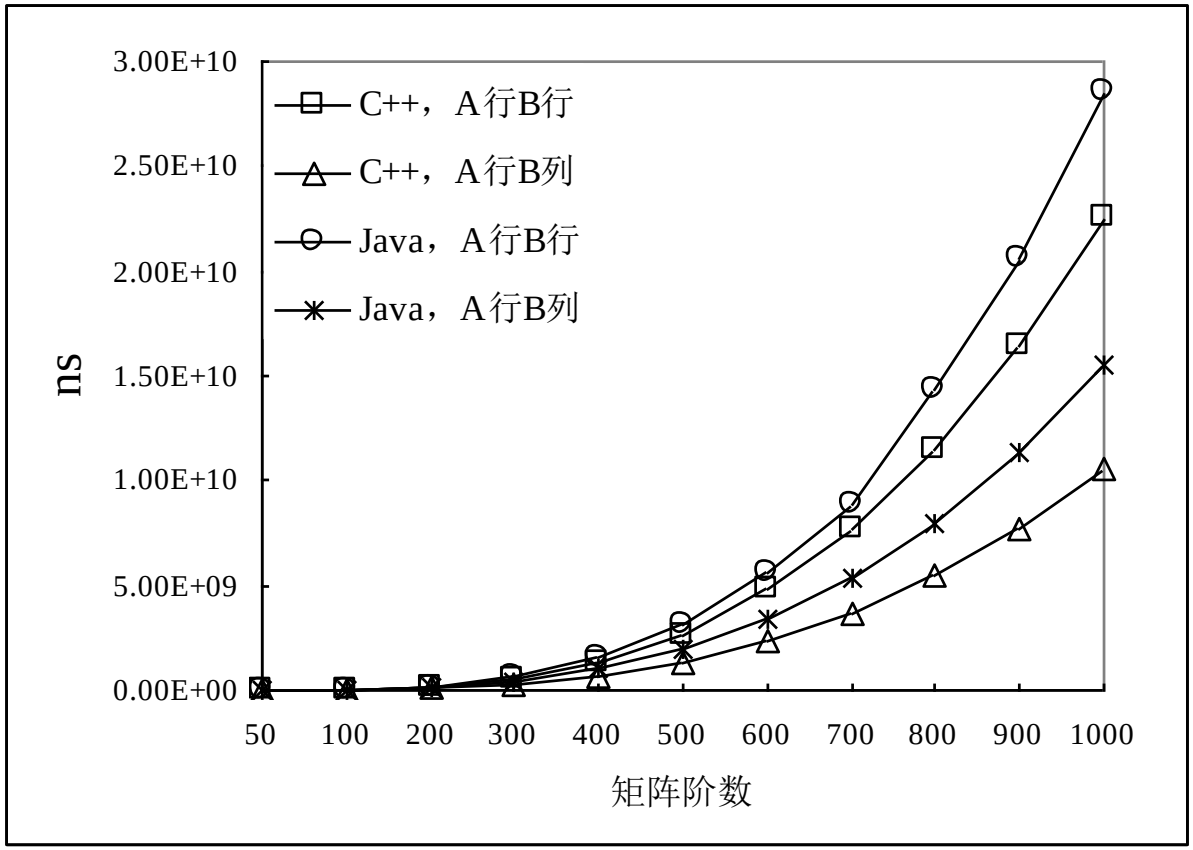
内存带宽的影响示例

- 增加带宽能够提高峰值计算速度。
- 对数据布局的假设是，连续的数据字被连续的指令所使用(空间本地性，spatial locality)
- 如果以数据布局为中心，那么计算的步骤应该确保连接的计算使用连接的数据

小结

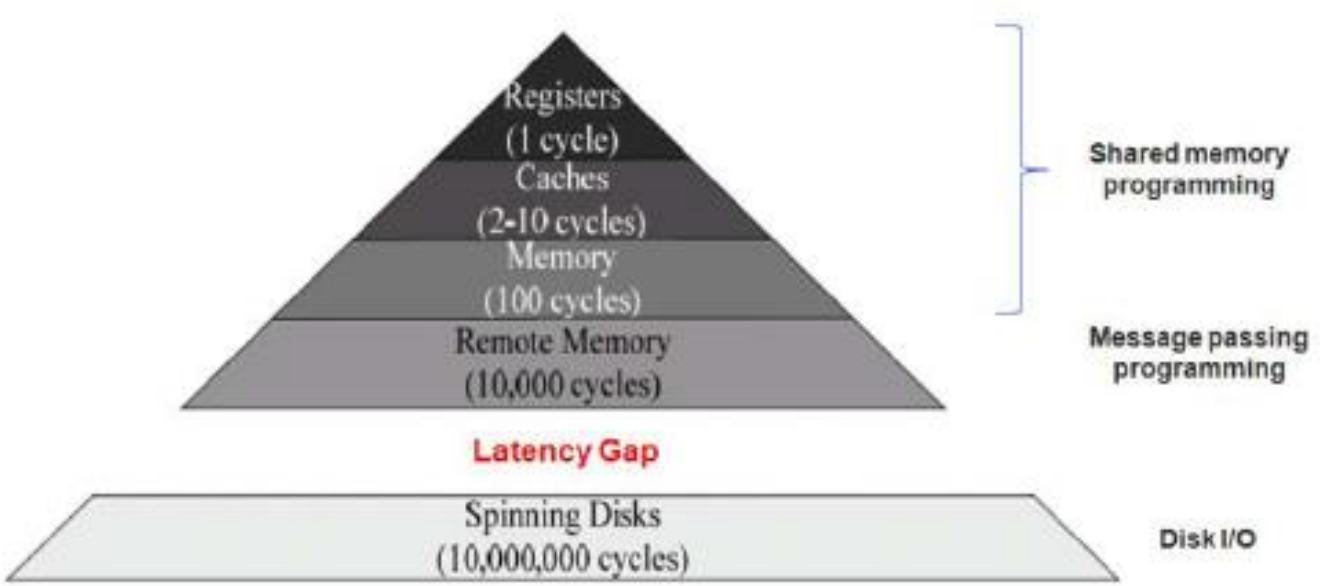
- 利用应用程序的空间本地性与时间本地性对于减少内存延迟及提高有效内存带宽非常重要。
- 计算次数与内存访问次数的比是一个很好的预测内存带宽的承受程序的指标。
- 内存的布局以及合理组织计算次序能对空间本地性和时间本地性产生重大影响。

实际测试

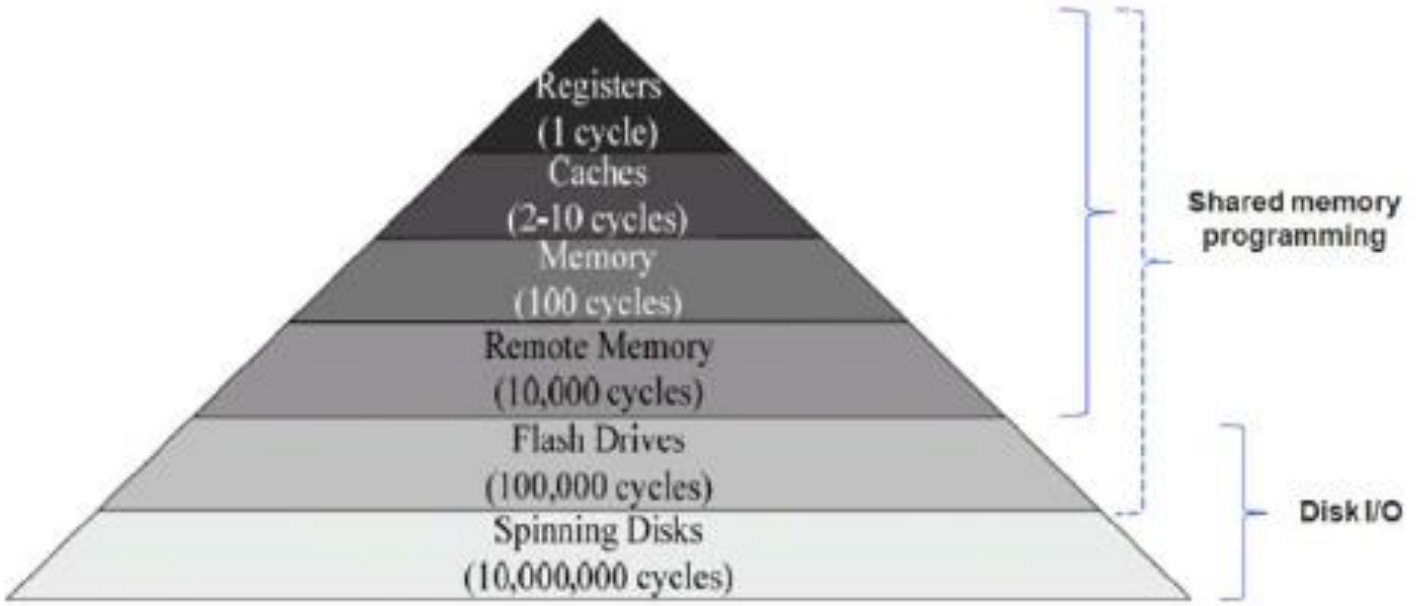


2.93GHz Intel 处理器，1M高速缓存，512M主存（533MHz）

传统计算系统中存储访问层次



增加Flash SSD层

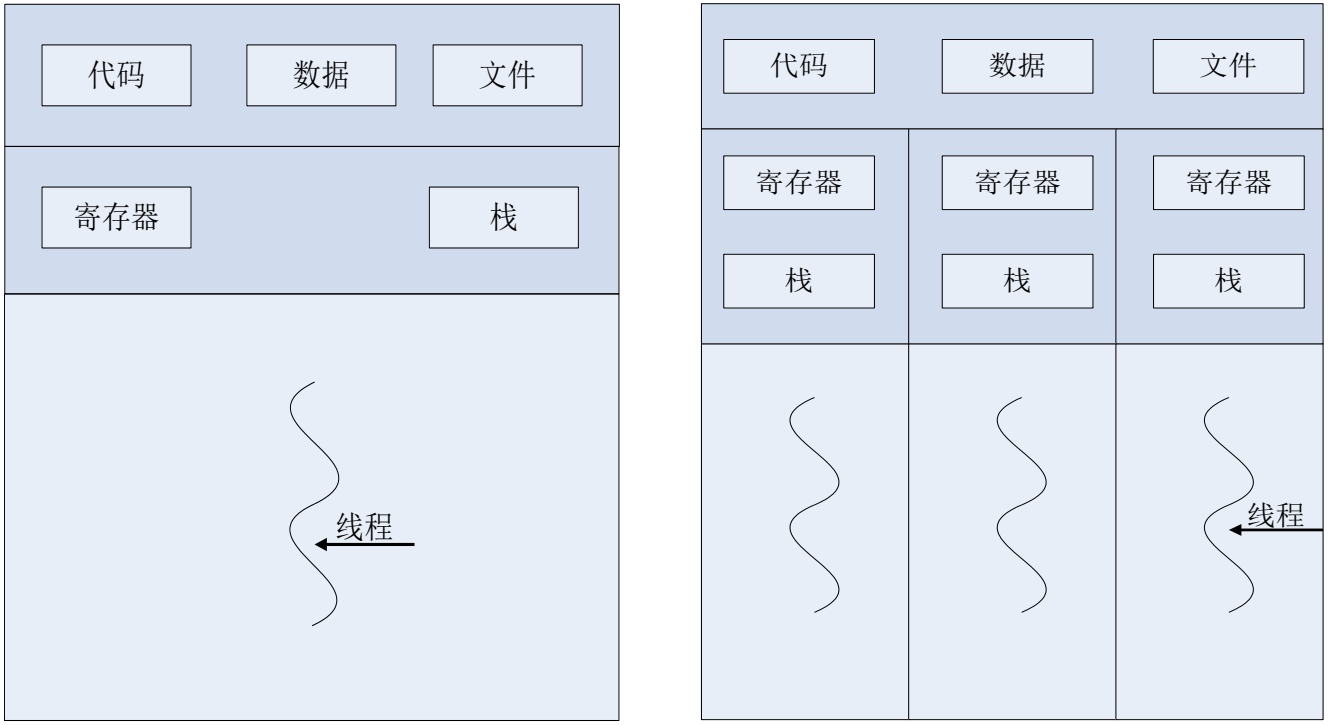


Outline

- 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。

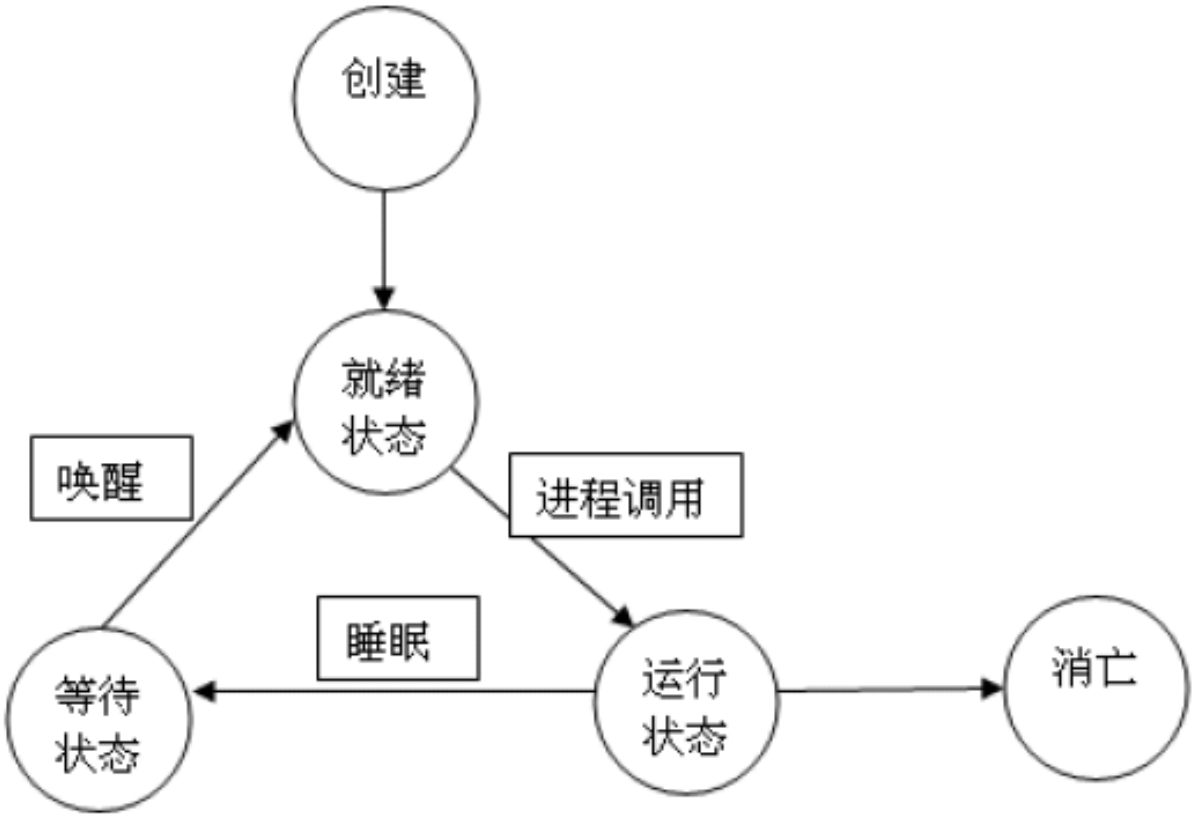
系统开销

- 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

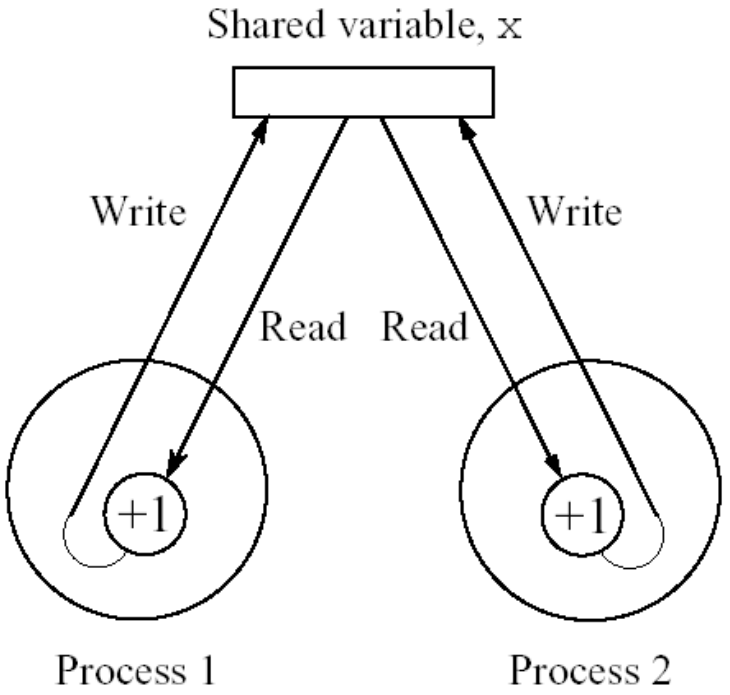
访问共享变量时产生的冲突

Instruction

```
x = x + 1;
```

Time

Process 1	Process 2
read x	read x
compute x + 1	compute x + 1
write to x	write to x



Busy-waiting

- 示例：

```
1   y = Compute(my_rank);
2   while (flag != my_rank);
3   x = x + y;
4   flag++;
```

- 关闭编译器自动优化，否则可能被编译为

```
y = Compute(my_rank);
x = x + y;
while (flag != my_rank);
flag++;
```

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

信号量

- 信号量是E. W. Dijkstra 在1965 年提出的一种方法，可以用一个整数变量sem 来表示，对信号量有两个基本的原子操作：P（wait, 减量操作）和V（signal,增量操作）

对信号量的解释

- 以一个停车场的运作为例。
  - 假设停车场只有三个车位，一开始三个车位都是空的。这时如果同时来了五辆车，看门人允许其中三辆直接进入，然后放下车拦，剩下的车则必须在入口等待，此后来的车也都不得不在入口处等待。这时，有一辆车离开停车场，看门人得知后，打开车拦，放入外面的一辆进去，如果又离开两辆，则又可以放入两辆，如此往复。
- 在这个停车场系统中，车位是公共资源，每辆车好比一个线程，看门人相当于信号量。

Pthread中的信号量

- 信号量的数据类型为结构sem_t，长整型数。
- sem_post(sem_t *sem)
  - 增加信号量的值。当有线程阻塞在这个信号量上时，调用这个函数会使其中的一个线程开始运行，选择机制由线程的调度策略决定。
- em_wait(sem_t *sem)
  - 阻塞当前线程直到信号量sem的值大于0，解除阻塞后将sem的值减一，表明公共资源经使用后减少。

条件变量

- 条件变量(Condition variable)是用来通知共享数据状态信息的。当特定条件满足时，线程等待或者唤醒其他合作线程。
- 条件变量不提供互斥，需要一个互斥锁来同步对共享数据的访问。

条件变量主要操作（pthread）

- pthread_cond_signal 使在条件变量上等待的线程中的一个线程重新开始。如果没有等待的线程，则什么也不做。如果有多个线程在等待该条件，只有一个能重启动，但不能指定哪一个。
- pthread_cond_broadcast 重启动等待该条件变量的所有线程。如果没有等待的线程，则什么也不做。

条件变量主要操作

- pthread_cond_wait 自动解锁互斥锁(如同执行了 pthread_unlock_mutex)，并等待条件变量触发。这时线程挂起，不占用 CPU 时间，直到条件变量被触发。在调用 pthread_cond_wait 之前，应用程序必须加锁互斥锁。pthread_cond_wait 函数返回前，自动重新对互斥锁加锁(如同执行了 pthread_lock_mutex)。
- 互斥锁的解锁和在条件变量上挂起都是自动进行的。因此，在条件变量被触发前，如果所有的线程都要对互斥锁加锁，这种机制可保证在线程加锁互斥锁和进入等待条件变量期间，条件变量不被触发。

条件变量示例

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;/*初始化互斥锁*/
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;/*初始化条件变量*/

void *thread1(void *);
void *thread2(void *);

int i=1; //全局变量
int main(void)
{
    pthread_t t_a;
    pthread_t t_b;

    pthread_create(&t_a,NULL,thread2,(void *)NULL);/*创建进程t_a*/
    pthread_create(&t_b,NULL,thread1,(void *)NULL); /*创建进程t_b*/
    pthread_join(t_b, NULL);/*等待进程t_b结束*/
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);
    exit(0);
}
```

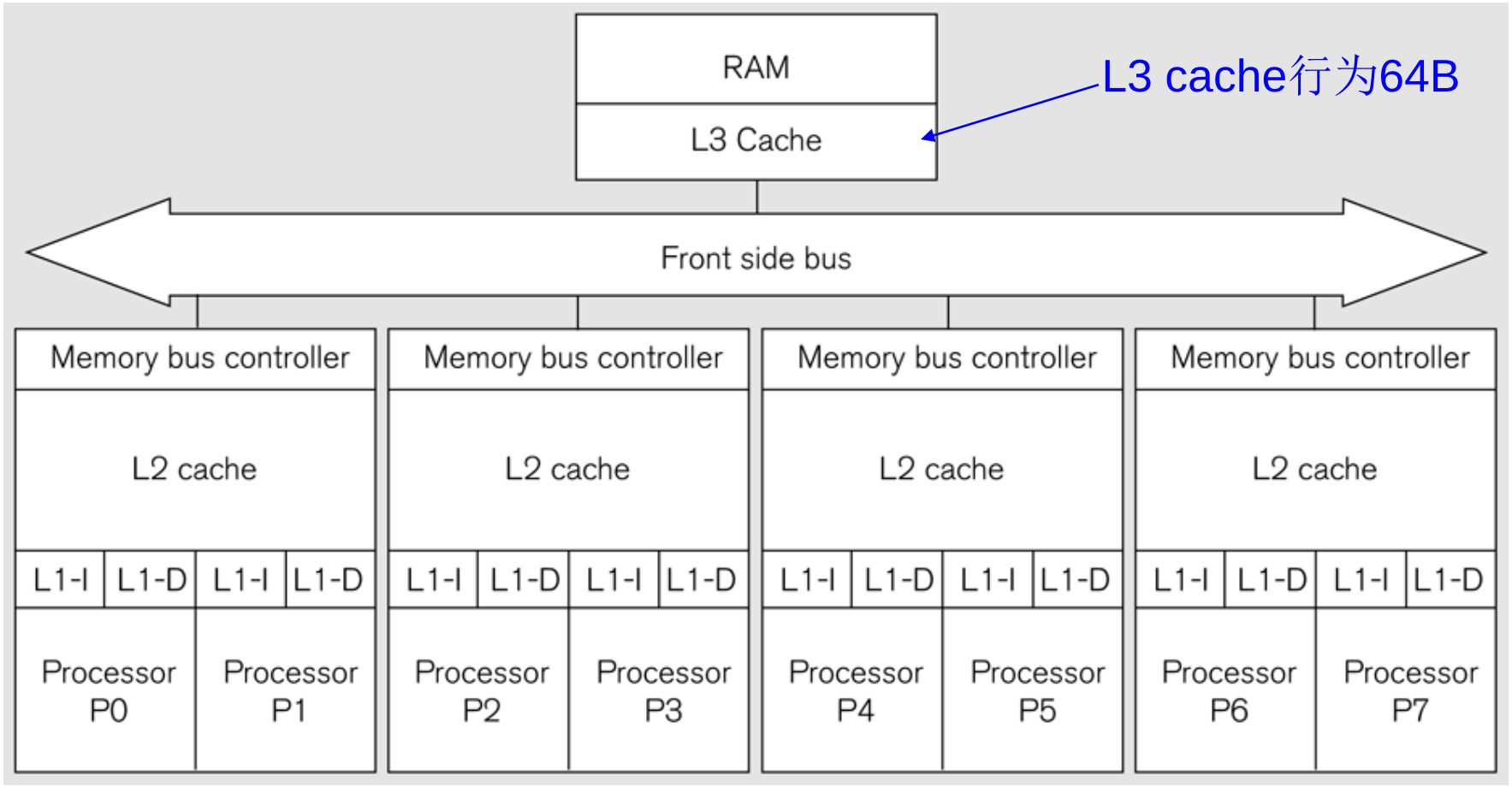
条件变量示例

```
void *thread1(void *junk)
{
    for(i=1;i<=9;i++)
    {
        pthread_mutex_lock(&mutex);/*锁住互斥锁*/
        if(i%3==0)
            pthread_cond_signal(&cond);/*条件改变，发送信号，通知t_b进程*/
        else
            printf("thead1:%d\n",i);
        pthread_mutex_unlock(&mutex);/*解锁互斥锁*/
        sleep(1);
}}

void *thread2(void *junk)
{
    while(i<9)
    {
        pthread_mutex_lock(&mutex);
        if(i%3!=0)
            pthread_cond_wait(&cond,&mutex);/*等待*/
        printf("thread2:%d\n",i);
        pthread_mutex_unlock(&mutex);
        sleep(1);
}}
```

输出结果：

```
thread1:1
thread1:2
thread2:3
thread1:4
thread1:5
thread2:6
thread1:7
thread1:8
thread2:9
```

实验环境中多核计算机系统结构

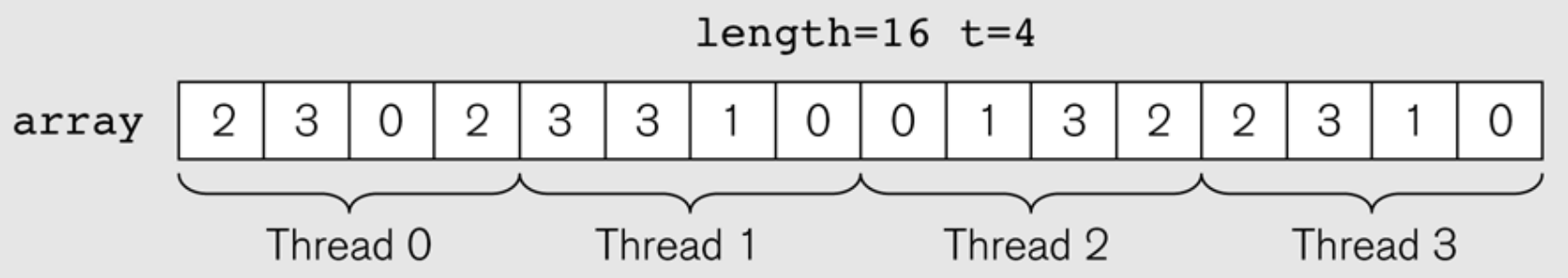


统计3的个数：串行代码（伪代码）

```
int *array;
int length;
int count;
int count3s()
{
   int i;
   count=0;
   for (i=0; i<length; i++)
   {
       if(array[i]==3)
       {
         count++;
       }
   }
   return count;
}
```

对数组的划分

- 长度为16，线程数为4



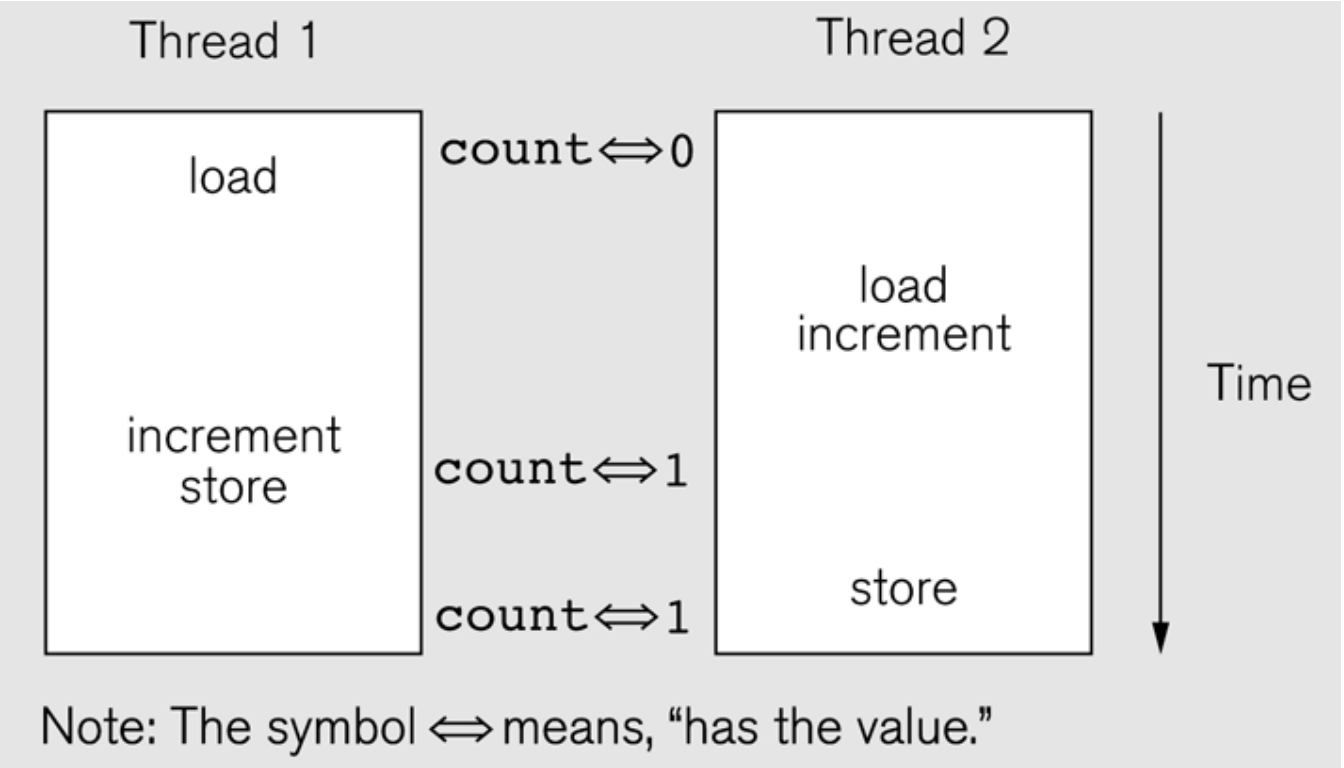
- 实际实验中数组规模为50M，随机分布30%的数值3。实验结果是1000次运行的均值。

并行算法1

此算法不能获得正确的结果

```
1  int t;                    /* number of threads */
2  int *array;
3  int length;
4  int count;
5
6  void count3s()
7  {
8    int i;
9    count = 0;
10   /* Create t threads */
11   for(i=0; i<t; i++)
12   {
13     thread_create(count3s_thread, i);
14   }
15
16   return count;
17 }
18
19 void count3s_thread(int id)
20 {
21   /* Compute portion of the array that this thread
        should work on */
22      int length_per_thread=length/t;
23      int start=id*length_per_thread;
24
25   for(i=start; i<start+length_per_thread; i++)
26   {
27     if(array[i]==3)
28     {
29       count++;
30     }
31   }
32 }
```

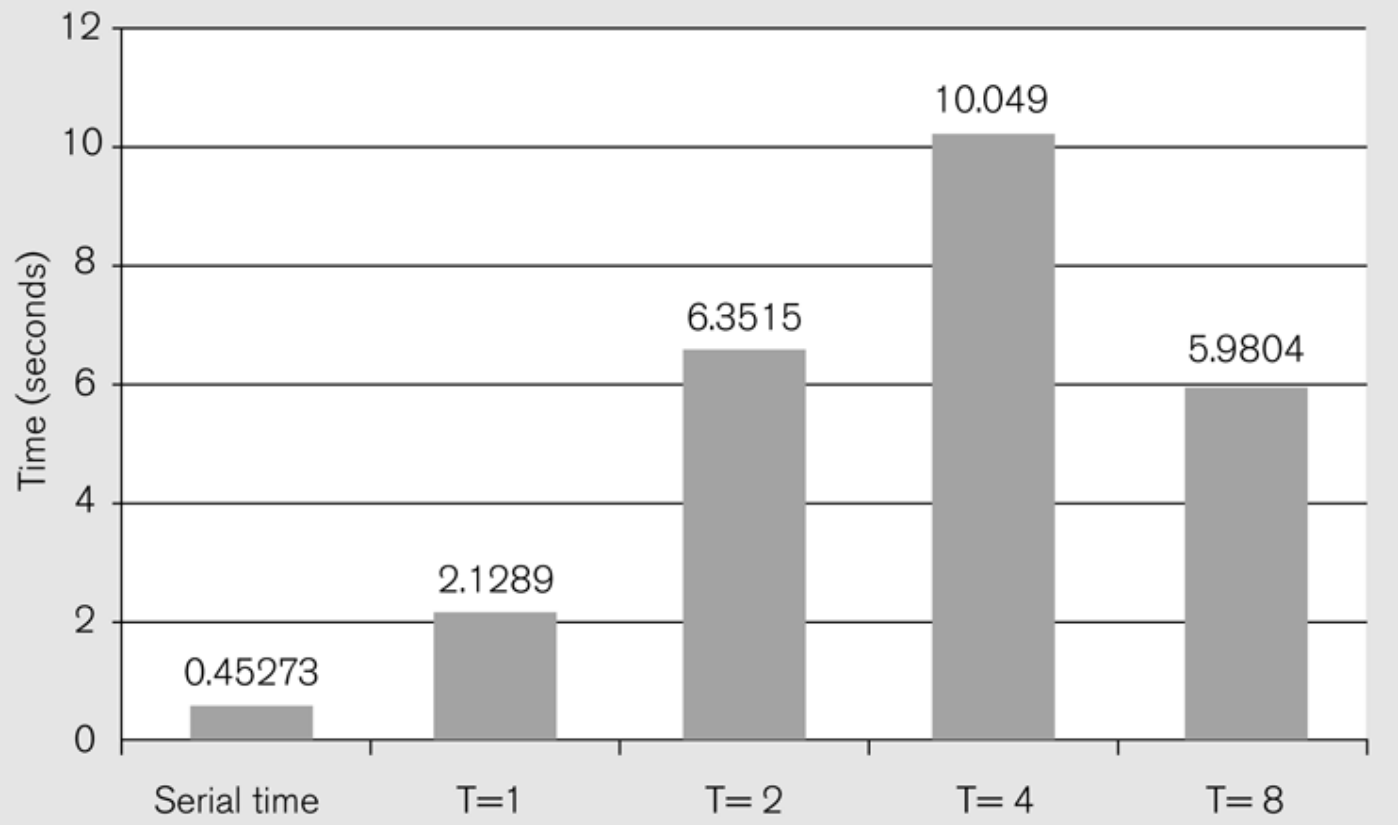
竞态条件


Thread 1
Thread 2
load
increment
store
load
increment
store
count⇔0
count⇔1
count⇔1
Time
Note: The symbol ⇔ means, “has the value.”

并行算法2：加互斥锁

```
 1  mutex m;
 2
 3  void count3s_thread(int id)
 4  {
 5    /* Compute portion of the array that this thread
        should work on */
 6    int length_per_thread=length/t;
 7    int start=id*length_per_thread;
 8
 9    for(i=start; i<start+length_per_thread; i++)
10    {
11      if(array[i]==3)
12      {
13        mutex_lock(m);
14        count++;
15        mutex_unlock(m);
16      }
17    }
18  }
```

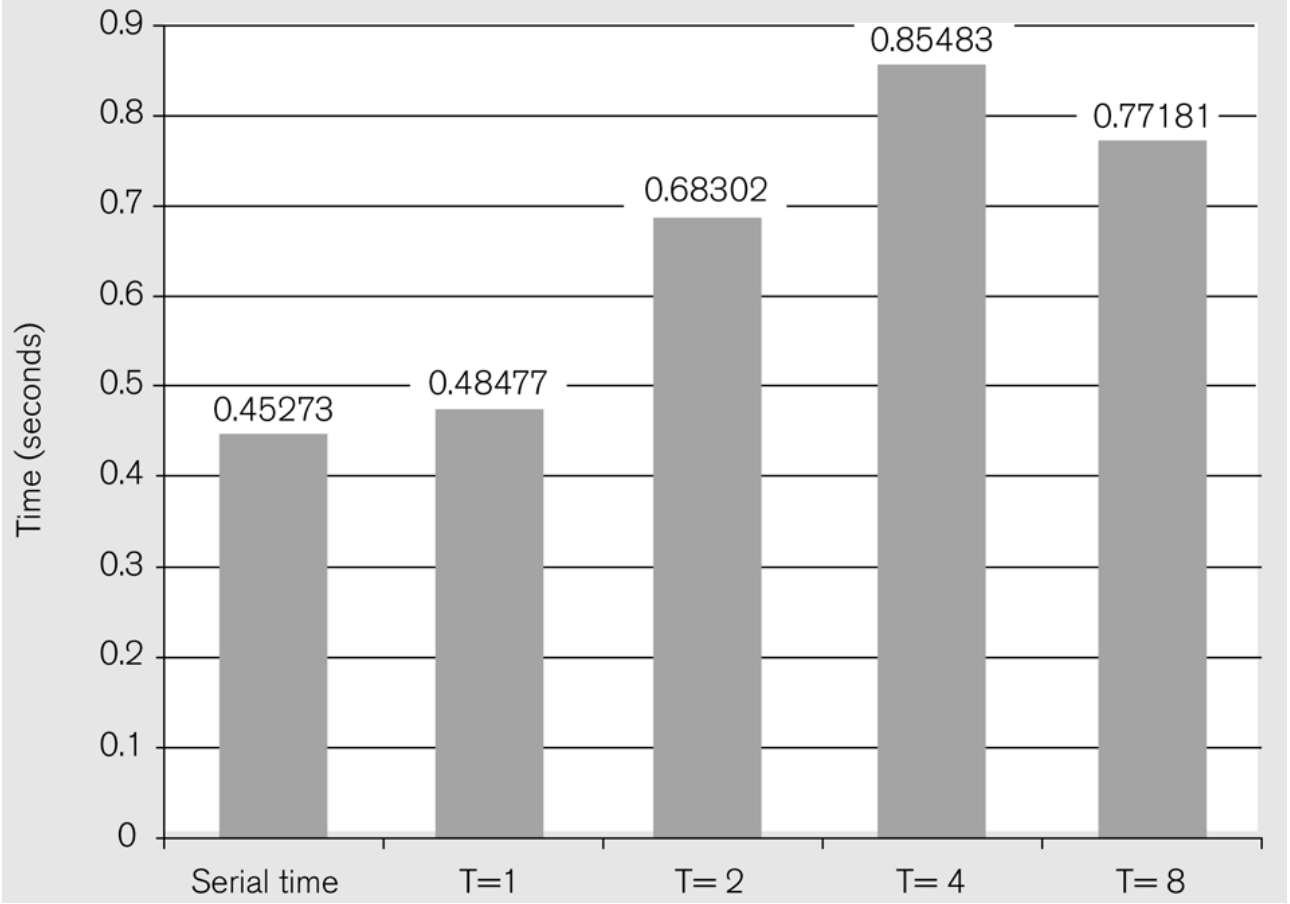
并行算法2的性能


12
10
8
6
4
2
0
Time (seconds)
0.45273
2.1289
6.3515
10.049
5.9804
Serial time
T=1
T= 2
T= 4
T= 8

并行算法3

```
 1  private_count[MaxThreads];
 2  mutex m;
 3
 4  void count3s_thread(int id)
 5  {
 6    /* Compute portion of array for this thread to
        work on */
 7    int length_per_thread=length/t;
 8    int start=id*length_per_thread;
 9
10    for(i=start; i<start+length_per_thread; i++)
11    {
12      if(array[i] == 3)
13      {
14        private_count[id]++;
15      }
16    }
17    mutex_lock(m);
18    count+=private_count[id];
19    mutex_unlock(m);
20  }
```

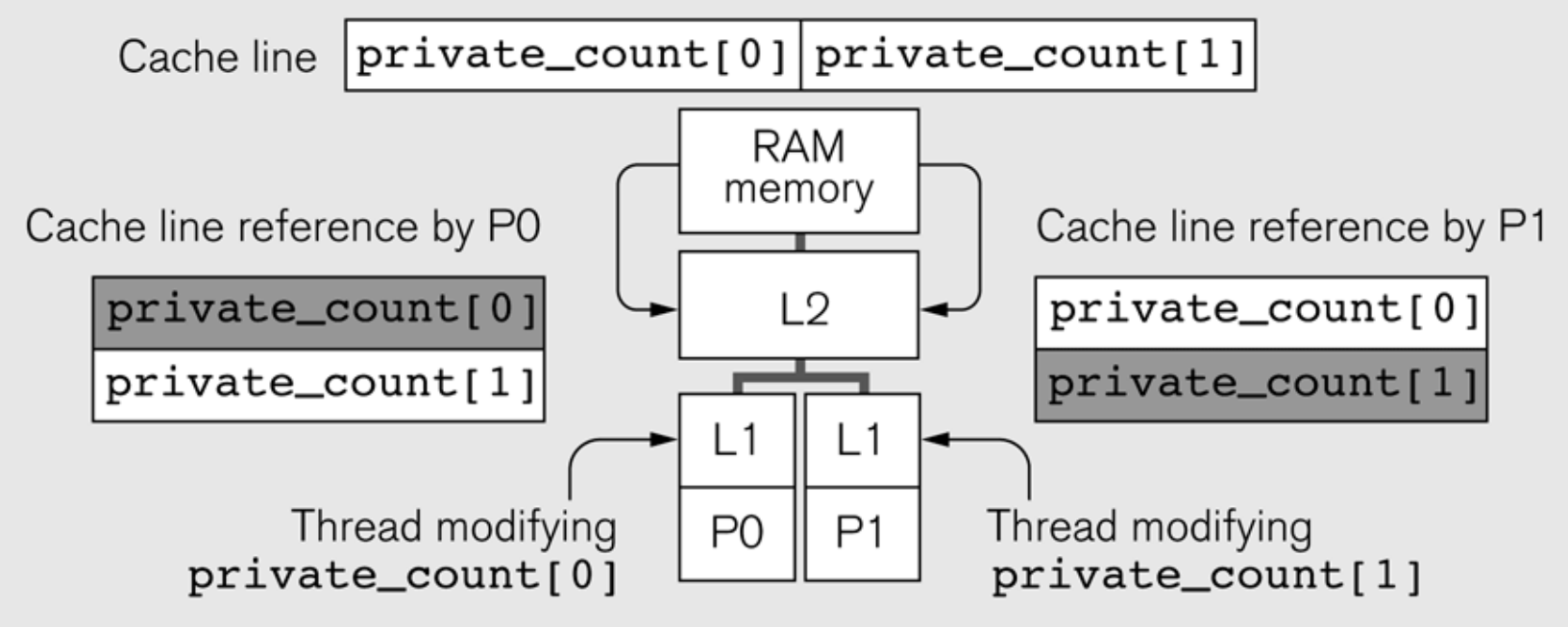
并行算法3的性能


0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Time (seconds)
0.45273
0.48477
0.68302
0.85483
0.77181
Serial time
T=1
T= 2
T= 4
T= 8

cache一致性

- cache一致性的单位是行（本例中一行为64B）
- 对cache行中的任意部分的修改等同于对整个行的修改
- L3 cache行修改后将触发L2、L1缓存的更新
- 处理器P0和P1上的线程对private_count[0]或private_count[1]进行互斥访问，但底层系统将它们置于同一个64B的cache行中。

假共享（false sharing）

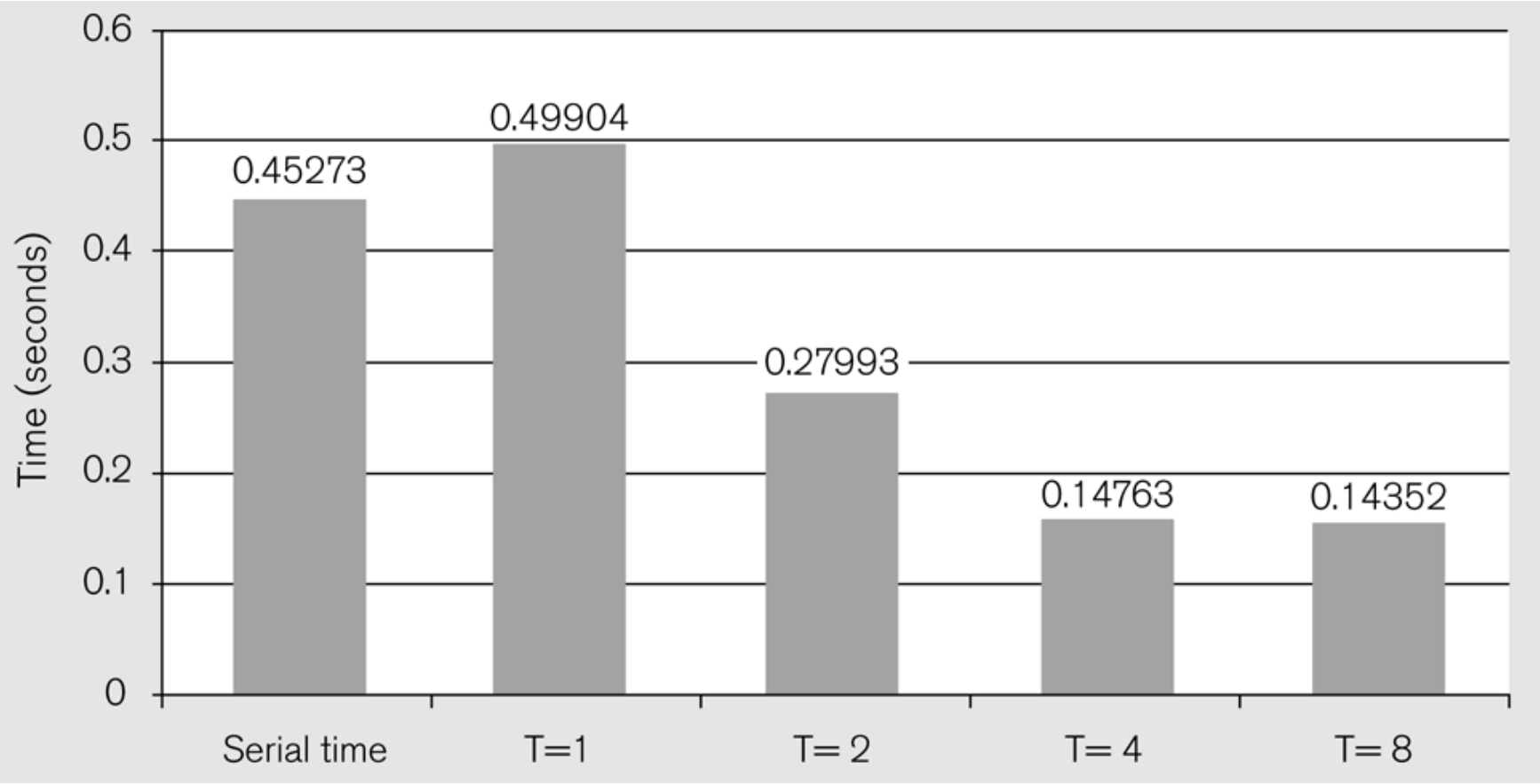


并行算法4：避免false sharing

```
 1  struct padded_int
 2  {
 3    int value;
 4    char padding[60];
 5  } private_count[MaxThreads];
 6
 7  void count3s_thread(int id)
 8  {
 9    /* Compute portion of the array this thread should
         work on */
10    int length_per_thread=length/t;
11    int start=id*length_per_thread;
12
13    for(i=start; i<start+length_per_thread; i++)
14    {
15      if(array[i] == 3)
16      {
17        private_count[id]++; (private_count[id].value++;)
18      }
19    }
20    mutex_lock(m);
21    count+=private_count[id].value;
22    mutex_unlock(m);
23  }
```

cache行大小为64B

并行算法4的性能


0.6
0.5
0.4
0.3
0.2
0.1
0
Time (seconds)
0.45273
0.49904
0.27993
0.14763
0.14352
Serial time
T=1
T= 2
T= 4
T= 8

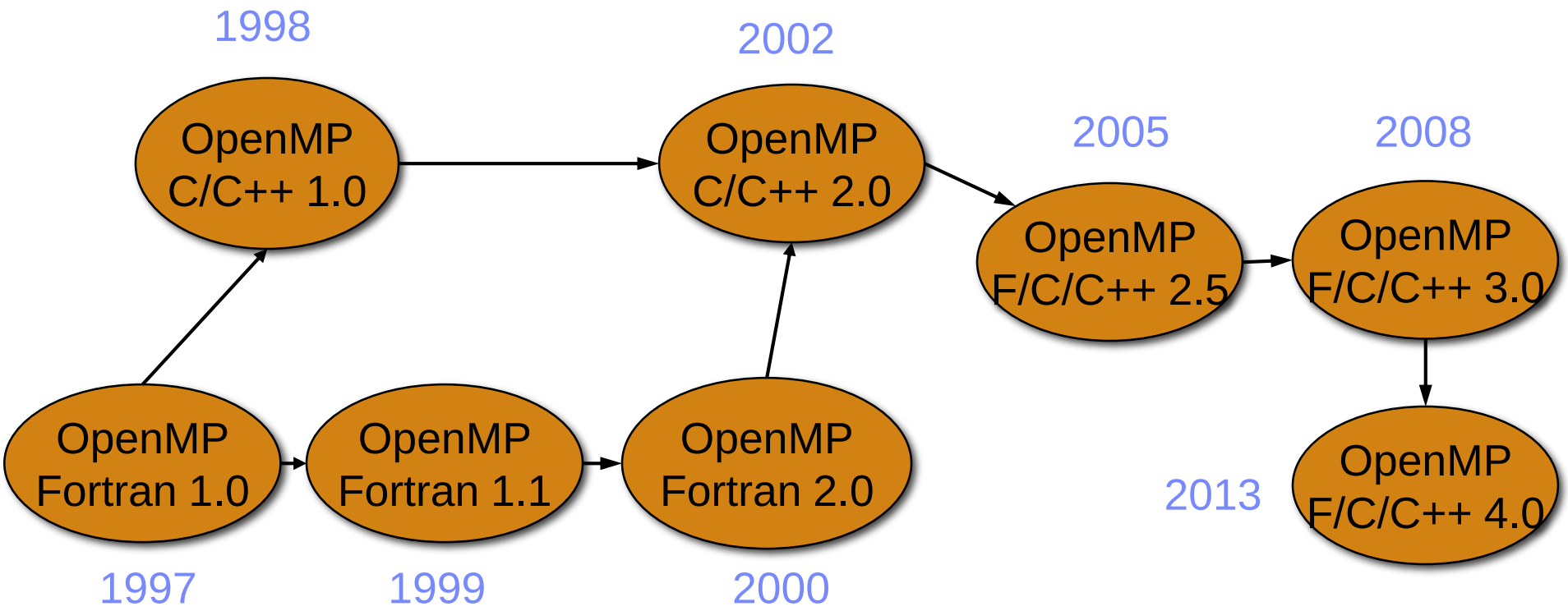
Outline

- 存储访问
- Pthead多线程
- **OpenMP**

OpenMP概述

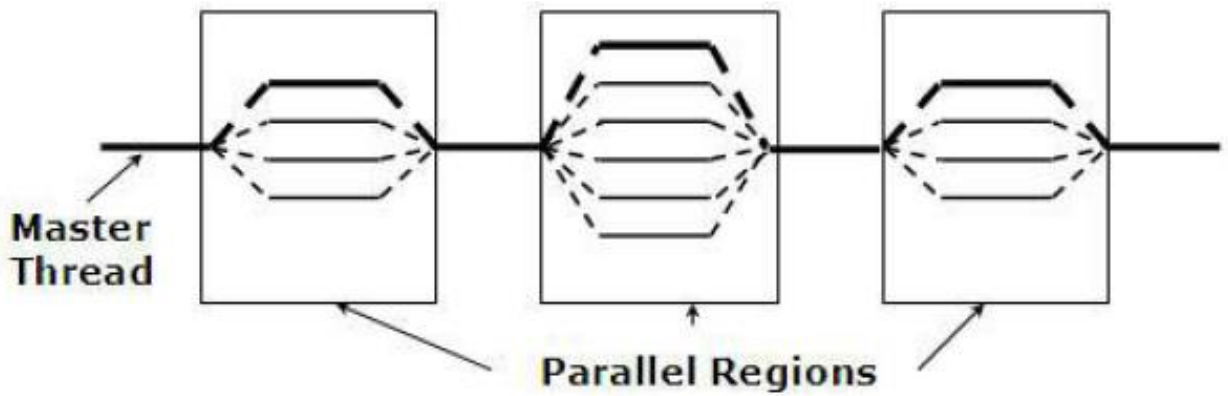
- OpenMP 是一种面向共享内存以及分布式共享内存的多处理器多线程并行编程语言。
- OpenMP是一种能够被用于显式制导多线程、共享内存并行的应用程序编程接口（API）。
- OpenMP标准诞生于1997 年，目前其结构审议委员会（Architecture Review Board, ARB）已经制定并发布OpenMP 4.0 版本。
- www.openmp.org

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

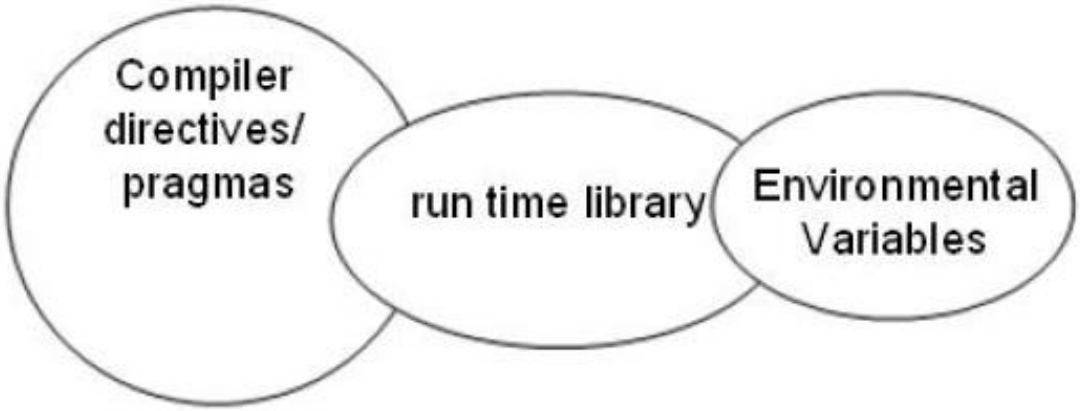
OpenMP编程模型：Fork-Join

- 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
- 数据域
  - 数据共享属性子句
  - threadprivate子句
  - 数据拷贝子句

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

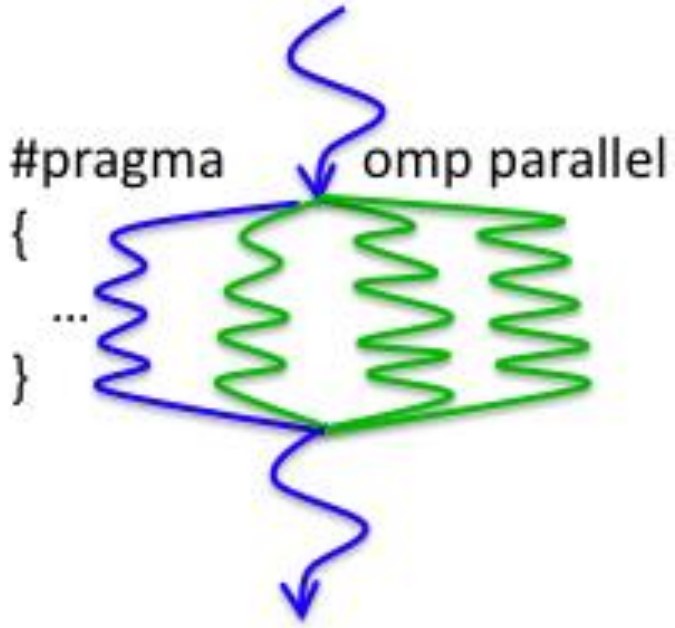
```
#include “omp.h”
void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

并行代码

编译制导语句（Compiler Directive）

- **并行域**
- 共享任务
- 同步
- 数据域
  - threadprivate
  - 数据域属性子句

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

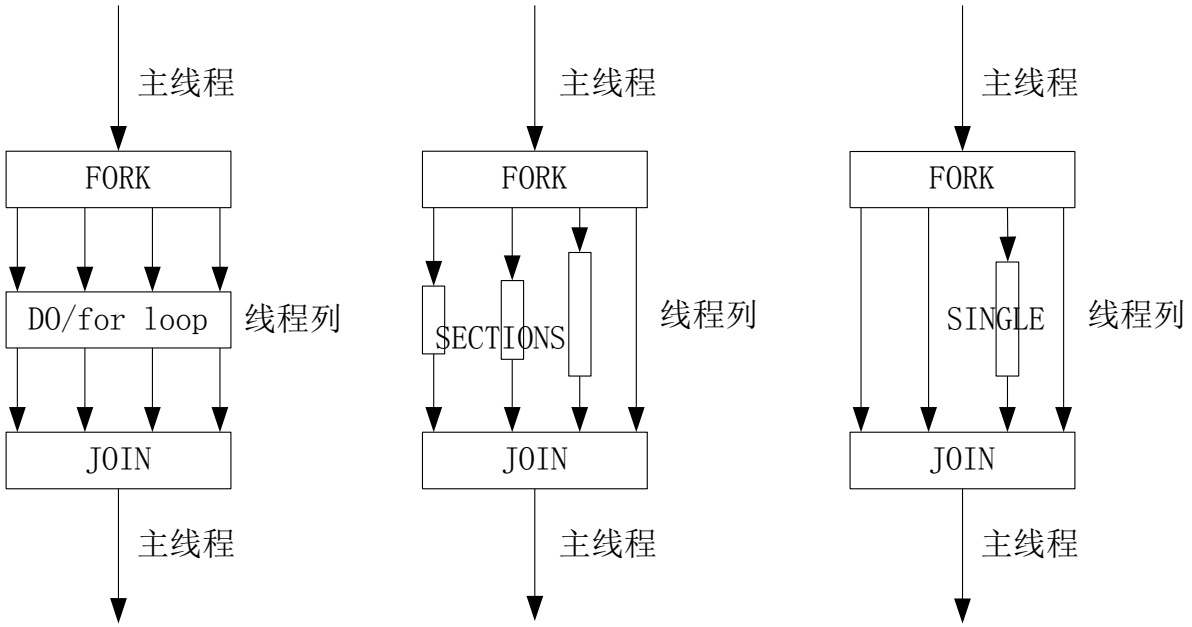
```
#include <omp.h>

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
- 数据域
  - 数据共享属性子句
  - threadprivate子句
  - 数据拷贝子句

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**
- 数据域
  - 数据共享属性子句
  - threadprivate子句
  - 数据拷贝子句

同步

- master 制导语句
- critical制导语句
- barrier制导语句
- atomic制导语句
- flush制导语句
- ordered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

atomic制导语句

- atomic制导语句指定特定的存储单元将被原子更新
- 语句格式
  - #pragma omp atomic newline
- atomic使用的格式

```
x binop = expr
x++
++x
x--
--x
```

x是一个标量
expr是一个不含对x引用的标量表达式，且不被重载
binop是+,*,-,/,&,^,|,>>,or<<之一，且不被重载

atomic示例

```
#include <iostream>
#include <omp.h>
int main()
{
    int sum = 0;
    std::cout << "Before: " << sum << std::endl;
#pragma omp parallel for
    for (int i = 0; i < 20000; ++i)
    {
#pragma omp atomic
        sum++;
    }
    std::cout << "After: " << sum << std::endl;
    return 0;
}
```

输出：

Before: 0
After:20000

无atomic，则输出结果会不确定。

flush制导语句

- flush制导语句用以标识一个同步点，用以确保所有的线程看到一致的存储器视图
- 语句格式
  - #pragma omp flush (list) newline
- flush将在下面几种情形下隐含运行，nowait子句除外

barrier
critical:进入与退出部分
ordered:进入与退出部分
parallel:退出部分
for:退出部分
sections:退出部分
single:退出部分

ordered制导语句

- ordered制导语句指出其所包含循环的执行按循环次序进行
- 任何时候只能有一个线程执行被ordered所限定部分
- 只能出现在for或者parallel for语句的动态范围中
- 语句格式：
  - #pragma omp ordered newline

ordered示例

```
void work(int i) {}
void a24_good(int n)
{
  int i;

#pragma omp for ordered
  for (i=0; i<n; i++) {
    if (i <= 10) {
      #pragma omp ordered
         work(i);
    }

    if (i > 10) {
      #pragma omp ordered
        work(i+1);
    }
  }
}
```

编译制导语句

- 并行域
- 共享任务
- 同步
- **数据域**
  - **数据共享属性子句**
  - **threadprivate子句**
  - **数据拷贝子句**

数据共享属性子句

- 变量作用域范围
- 数据域属性子句
  - private子句
  - shared子句
  - default子句
  - firstprivate子句
  - lastprivate子句
  - reduction子句

private子句

- private子句表示它列出的变量对于每个线程是局部的 。
- 语句格式
  - private(list)

private()

```
#include <stdio.h>

int main()
{
    int i, x = 100;
    #pragma omp parallel for private(x)
    for (i=0; i<8; i++)
        {
            x += i;
            printf("x = %d\n", x);
        }
    printf("global x = %d\n", x);
    return 1;
}
```

4线程

```
x = 0
x = 1
x = 2
x = 5
x = 6
x = 3
x = 4
x = 7
global x = 100
```

shared子句

- shared子句表示它所列出的变量被线程组中所有的线程共享
- 所有线程都能对它进行读写访问
- 语句格式
  - shared (list)

default子句

- default子句让用户自行规定在一个并行域的静态范围中所定义的变量的缺省作用范围
- 语句格式
  - default (shared | none)

firstprivate子句

- firstprivate子句是private子句的超集
- 对变量做原子初始化
- 语句格式：
  - firstprivate (list)

firstprivate()

```
#include <stdio.h>

int main()
{
   int i, x = 100;
   #pragma omp parallel for firstprivate(x)
   for (i=0; i<8; i++)
         {
            x += i;
            printf("x = %d\n", x);
         }
         printf("global x = %d\n", x);
         return 1;
}
```

4线程

```
x = 100
x = 101
x = 102
x = 105
x = 106
x = 107
x = 104
x = 103
global x = 100
```

lastprivate子句

- lastprivate子句是private子句的超集
- 将变量从最后的循环迭代或段复制给原始的变量
- 语句格式
  - lastprivate (list)

lastprivate()

```
#include <stdio.h>

int main()
{
   int i, x = 100;
   #pragma omp parallel for firstprivate(x) lastprivate(x)
   for (i=0; i<8; i++)
   {
         x += i;
         printf("x = %d\n", x);
   }
   printf("global x = %d\n", x);
   return 1;
}
```

4线程：

```
x = 100
x = 101
x = 102
x = 105
x = 106
x = 107
x = 104
x = 103
global x = 103
```

reduction子句

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

reduction子句

```
#include <omp.h>
int main ()
{
   int   i, n, chunk;
   float a[100], b[100], result;
   /* Some initializations */
   n = 100;
   chunk = 10;
   result = 0.0;
   for (i=0; i < n; i++)
   {
            a[i] = i * 1.0;
            b[i] = i * 2.0;
   }
   #pragma omp parallel for default(shared) private(i)\
            schedule(static,chunk) reduction(+:result)
   for (i=0; i < n; i++)
            result = result + (a[i] * b[i]);
   printf("Final result= %f\n",result);
}
```

threadprivate编译制导语句

- threadprivate语句使一个全局文件作用域的变量在并行域内变成每个线程私有
- 每个线程对该变量复制一份私有拷贝
- 语句格式:
  - #pragma omp threadprivate (list) newline

threadprivate编译制导语句

```
#include<omp.h>
int counter = 0;
#pragma omp threadprivate(counter)
void inc_counter(){counter++;}

int main(int argc, char * argv[]){
  int i;
  #pragma omp parallel private (i)
  {
          for(i=0; i<1000; i++)
                  inc_counter();
           printf("counter=%d\n",counter);
  }
  printf("counter=%d\n",counter);
}
```

8线程：

```
counter=1000
counter=1000
counter=1000
counter=1000
counter=1000
counter=1000
counter=1000
counter=1000
counter=1000
```

threadprivate编译制导语句

```
int alpha[10], beta[10], i;//eg3
#pragma omp threadprivate(alpha)
int main ()
{
  /* First parallel region */
  #pragma omp parallel private(i,beta)
  for (i=0; i < 10; i++)
        alpha[i] = beta[i] = i;
  /* Second parallel region */
  #pragma omp parallel
        printf("alpha[3]= %d and beta[3]=%d\n",alpha[3],beta[3]);
}
```

8线程：

```
alpha[3]= 3 and beta[3]=0
alpha[3]= 3 and beta[3]=0
alpha[3]= 3 and beta[3]=0
alpha[3]= 3 and beta[3]=0
alpha[3]= 3 and beta[3]=0
alpha[3]= 3 and beta[3]=0
alpha[3]= 3 and beta[3]=0
alpha[3]= 3 and beta[3]=0
```

private和threadprivate区别

	PRIVATE	THREADPRIVATE
数据类型	变量	变量
位置	在域的开始或共享任务单元	在块或整个文件区域的例程的定义上
持久性	否	是
扩充性	只是词法的- 除非作为子程序的参数而传递	动态的
初始化	使用 FIRSTPRIVATE	使用 COPYIN

copyin子句

- copyin子句用来为线程组中所有线程的threadprivate变量赋相同的值
- 主线程该变量的值作为初始值
- 语句格式
  - copyin(list)

copyin示例

```
#include<omp.h>
int global=0;
#pragma omp threadprivate(global)
int main(int argc, char * argv[])
{
   global=1000;
   #pragma omp parallel copyin(global)
   {
            printf("global=%d\n",global);
            global=omp_get_thread_num();
   }
   printf("global=%d\n",global);
   printf("parallel again\n");
   #pragma omp parallel
            printf("global=%d\n",global);
}
```

```
global=1000
global=1000
global=1000
global=1000
global=1000
global=1000
global=1000
global=1000
global=0
parallel again
global=0
global=2
global=1
global=4
global=3
global=5
global=6
global=7
```

copyprivate 子句

- copyprivate子句提供了一种机制用一个私有变量将一个值从一个线程广播到执行同一并行区域的其他线程。
- 语句格式：

  copyprivate(list)

- copyprivate子句可以关联single构造，在single构造的barrier到达之前就完成了广播工作。

copyprivate子句

```
int counter = 0;
#pragma omp threadprivate(counter)
int increment_counter()
{
    counter++;
    return(counter);
}
#pragma omp parallel
{
     int    count;
   #pragma omp single copyprivate(counter)
    {
          counter = 50;
    }
    count = increment_counter();
    printf("ThreadId: %ld, count = %ld\n", omp_get_thread_num(), count);
}
```

```
ThreadId: 2, count = 51
ThreadId: 0, count = 51
ThreadId: 3, count = 51
ThreadId: 1, count = 51
```

Outline

- OpenMP概述
- 编译制导语句
- **运行时库函数**
- **环境变量**
- 实例

运行库例程与环境变量

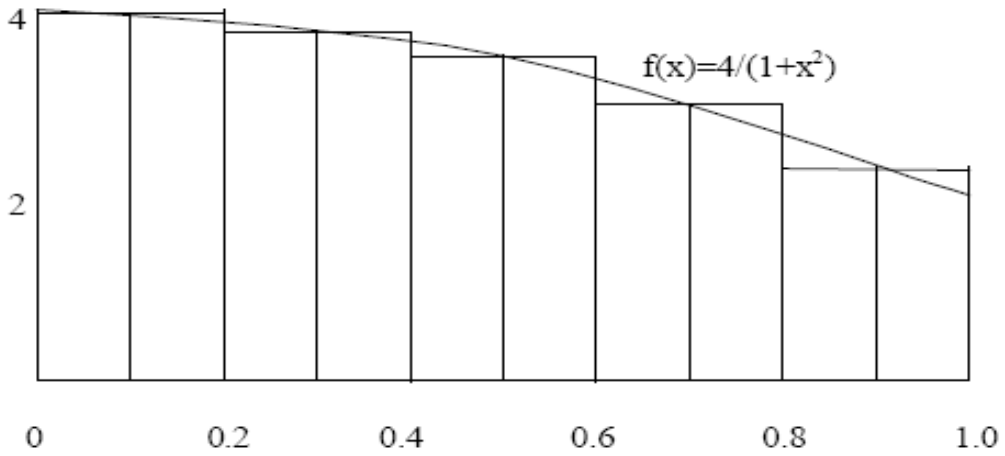
- 运行库例程
  - OpenMP标准定义了一个应用编程接口来调用库中的多种函数
  - 对于C/C++，在程序开头需要引用文件“omp.h”

- 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;
}
```

共享内存环境中并行程序设计要点

- 分析算法，并行化，确保计算结果正确（前提）
  - 共享数据保护
  - 同步，一致性
- 性能调优（重点）
  - 数据结构（适合并行程序的数据结构）
  - 任务调度与负载平衡
  - 运行时性能数据收集与分析