高性能计算期末复习

第一章

并行数组求和 线性规约$(n-1)$ vs. 树形规约$(\lceil \log_2 n \rceil)$

- 任务并行
- 数据并行: 同任务不同数据

coordination

- 通信
- 负载均衡：不同的核承担的任务量相近
- 同步

一些术语

- 并发(concurrent)
- 并行(parallel)，**单核计算机无法并行。**
- 分布式(distributed)

  https://pediaa.com/what-is-the-difference-between-parallel-and-distributed-computing/

第二章

冯诺依曼架构

CPU

- ALU
- Control Unit
- 寄存器
- 程序计数器：存储下一个被执行指令的地址

BUS

Main Memory: Local/Remote

瓶颈：BUS的信息传递速度

进程和线程

进程：一个程序实例，包括

- 可执行机器代码
- 一块内存及其描述符：调用栈、堆
- 安全信息，指定该程序能够访问的软硬件
- 程序执行状态信息，如程序计数器

线程：包含在进程中

- 派生和合并
- 线程切换比进程更快捷
- 除**程序计数器和调用栈**外，共享其他资源

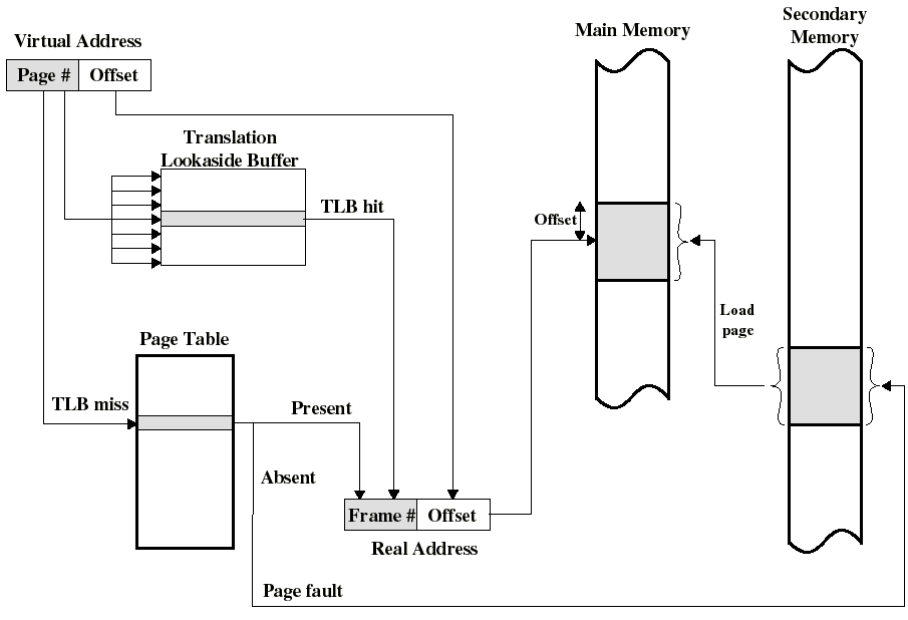
解决冯诺依曼瓶颈

Cache

- 时空局部性
- 保持缓存和内存数据一致的方法
    - 写直达（write through），缓存数据被修改后，立刻更新内存中的数据
    - 写回（write back），Lazy mark，缓存行被取代后，才更新内存中数据，常用
- 将内存映射到缓存的策略
    - 全相联（full associate），缓存行映射到缓存的任何位置
    - 直接映射（direct mapped），映射到唯一的缓存位置
    - n路组相连（n-way set associate），映射到n个缓存位置之一（最少使用原则）
- 受**硬件控制**

Virtual memory

- 使**主存作为次级存储（硬盘）的缓存**
- 基于时空局部性，只保存活动程序，闲置部分在swap space中
- 数据交换单位，**Pages，包含若干缓存行**
- page table，将虚拟地址转化为物理地址。
    - page table的一部分存储在内存中，称Translation-lookaside buffer (TLB)
    - 页面失效（page fault）attempting to access a page that's a not in memory (invalid physical address the page table).
- 为了保持一致性，必须使用写回。
- 受硬件和操作系统控制。



指令级并行 ILP

流水线（pipelining），多条指令重叠操作。

- 并行加法案例
- （补充）奔腾处理器整数流水线四级流水：指令预取、译码、执行、写回结果。

多发射（multiple issus），指令被静态或动态分配给多个核

- **在一个基本时钟周期内同时**从指令Cache中读出多条指令，**同时对多条指令进行译码。**
- 支持dynamic multiple issue的处理器是超标量处理器
- Speculation(前瞻), the complier or the processor makes guesses about instructions that can be executed simultaneously. https://zhuanlan.zhihu.com/p/33145828

硬件多线程

当前任务阻塞时的策略。

细粒度：当前线程阻塞时切换

- 能减少因阻塞浪费的时间
- Cons: a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction.

粗粒度：只在当前阻塞线程执行time-consuming operation时切换

- Pros: switching threads doesn't need to be nearly instantaneous.
- Cons: the processor can be idled on shorter stalls, and thread switching will also cause delays.

Simultaneous multithreading：允许多线程利用超标量处理器的多个功能块。

- If we designate "preferred" threads, that have many instructions ready to execute, we can somewhat reduce the issue of thread slowdown, need OS support.

并行硬件

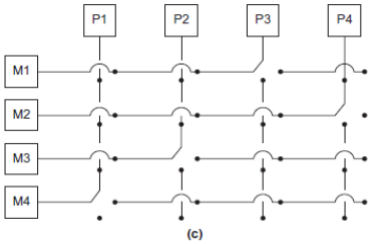
Flynn's Taxonomy

- SISD，经典冯诺依曼结构
- SIMD，数据并行，所有ALUs做一样的事或者同时闲置。
    - 向量处理器 Ch2.1 Pg 57-61
    - GPUs
- MIMD
    - 共享内存系统：一致内存访问和**非一致内存访问**，前者可扩展性有限
    - 分布式内存系统：集群(clusters)，网格(grid)，其中每个节点通常是共享内存系统.
- MISD

Interconnection networks

共享内存系统

- Bus interconnect，设备增多性能降低
- Crossbar(交叉开关矩阵)，更快，更复杂/贵



分布式内存系统

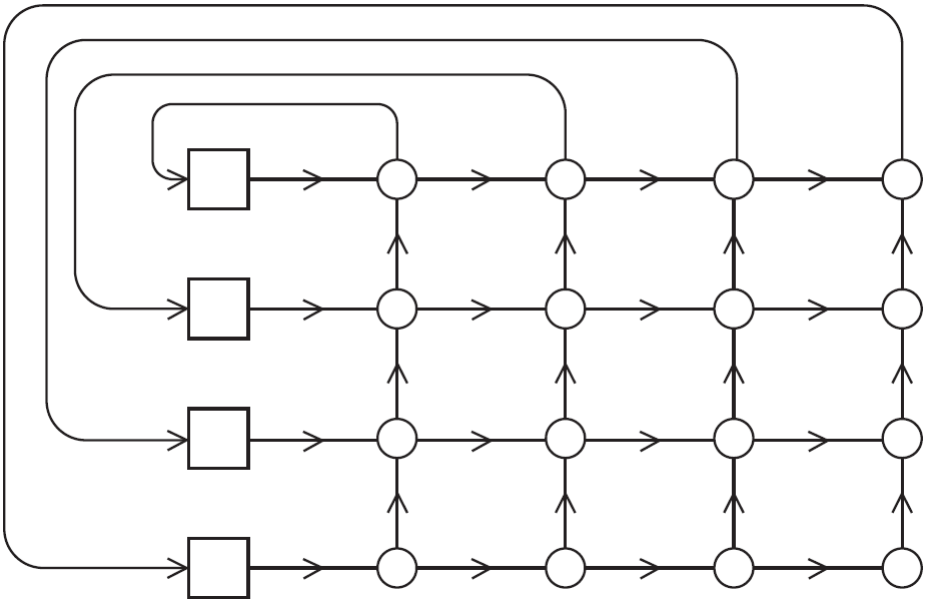
- 处理器数量p
- Bisection width, 将网络图切成两半最少断开的边数，$\leq p/2$
- Bandwidth, 链接传递数据的速度
- Bisection bandwidth，sums the bandwidth of the links.
- Latency, 从发送结束到开始接收的延迟时间
- 信息传递时间，$\text{Latency} + \frac{\text{data}}{\text{Bandwidth}}$.

Direct interconnect

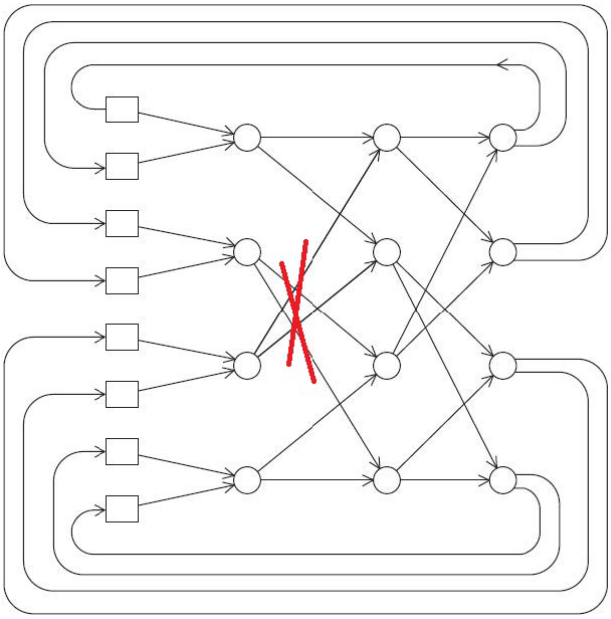
- ring，$\text{link} = p, \text{BW} = 2$
- toroidal mesh (环面网络)，$\text{link} = 2p, \text{BW} = 2\sqrt{p}$
- Fully connected network，$\text{link} = (p^2 - p)/2, \text{BW} = p/2$
- Hypercube, $\text{link} = p \log_2 p/2, \text{BW} = p/2$

Indirect interconnect

- Crossbar, $\text{link} = 2p^2$, 交换器间的链路数$2p^2 - 2p$. 只要没有两个核与同一个核通信，处理器就能同步通信



- Omega network，$\text{switch} = 2p \log_2 p$，交换器间链路数$plog_2p - p$ 交换器有$\log_2 p$层，每层$p/2$个，每个引出两条链路（每个交换器节点内有四个交换器）



缓存一致性

监听(Snooping)缓存一致性

当核A缓存中的x被修改后，沿着总线向其它核广播； 核B监听总线，将其缓存中的x标记为invalid.

- 写直达
- 写回：额外通信

更快，但对带宽需求高，可扩展性有限

基于字典的缓存一致性

https://pop0726.github.io/bxjs/text/ch02/se03/r2_3_6_4.htm

当核A缓存中的x被修改后，将目录其他核缓存的x标记为invalid.

需要额外存储，延迟更长，但对带宽需求小。

False Sharing

- 重复使用的变量被不同进程/线程反复读写，需要不断从内存更新。
- 影响效率，不影响正确性
- 解决办法：使用私有变量

并行软件

SPMD：单程序多数据

在共享内存线程调度中，有两种策略：

- 动态
- 静态，线程池，空间换时间

线程执行顺序存在非确定性。

Race condition

- 临界区(Critical section)，强迫串行以规避竞争条件。

输入输出

只有主线程/进程能访问stdin，所有线程/进程都能访问stdout/stderr，但因非确定性，后者通常只用于调试。

性能

$T_{parallel} = T_{serial}/p + T_{overhead}$

并行加速比$S = \dfrac{T_{serial}}{T_{parallel}}$

并行效率$E = S/p$

增大p, 若存在一个更大的数据规模n使得E不变，则并行程序具有扩展性。

- 若增大p，E减小很少，则具有强可扩展性；
- 若增大p，同时以相同比例增大n，E不变，则具有弱可扩展性

Amdahl's Law

如果不能并行的部分占比为α，可能的最高加速比不超过$1/\alpha$.

$$S = \frac{1}{\alpha + \frac{1-\alpha}{p}} < \frac{1}{\alpha}$$

计时

从第一个线程/进程开始执行到最后一个线程结束。

对串行程序来说，运行时间约等于CPU计算时间，但是对并行程序来说，应该用**墙上时间**来计时。墙上时间包括

- CPU计算、通信时间、同步开销时间、进程空闲时间.

并行程序设计

(Foster's methodology) 一般而言分为四步

- 任务划分
- 通讯
- 聚合前两步的成果
- 映射，将任务分配给进程/线程

第三章 MPI

SPMD的分布式内存编程。

基本函数

```
// MPI_Init定义了全局通信子MPI_COMM_WORLD
int MPI_Init(int * argc_p, char** argv_p); // 参数通常为NULL，返回Error code
int MPI_Comm_size(MPI_Comm comm, int* comm_sz_p); // 获取进程数
int MPI_Comm_rank(MPI_Comm comm, int* my_rank_p); // 获取当前进程编号

int MPI_Send(void* msg_buf_p, int msg_size, MPI_Datatype msg_type,
             int dst, int tag, MPI_Comm comm);
int MPI_Recv(void* msg_buf_p, int buf_size, MPI_Datatype buf_type,
             int source, int tag, MPI_Comm comm, MPI_Status* status_p); // 最后一个参数通常
MPI_STATUS_IGNORE

int MPI_Finalize();
```

通过MPI_Status可以获取接受数据的大小，数据发送者和tag.

```
int MPI_Get_count(MPI_Status* status_p, MPI_Datatype type, int* count_p);
status.MPI_SOURCE;
status.MPI_TAG;
```

发送和接受是保序的。接收会阻塞直到收到信息，发送有立即返回和阻塞直到接收两种结果，取决于库函数的实现。

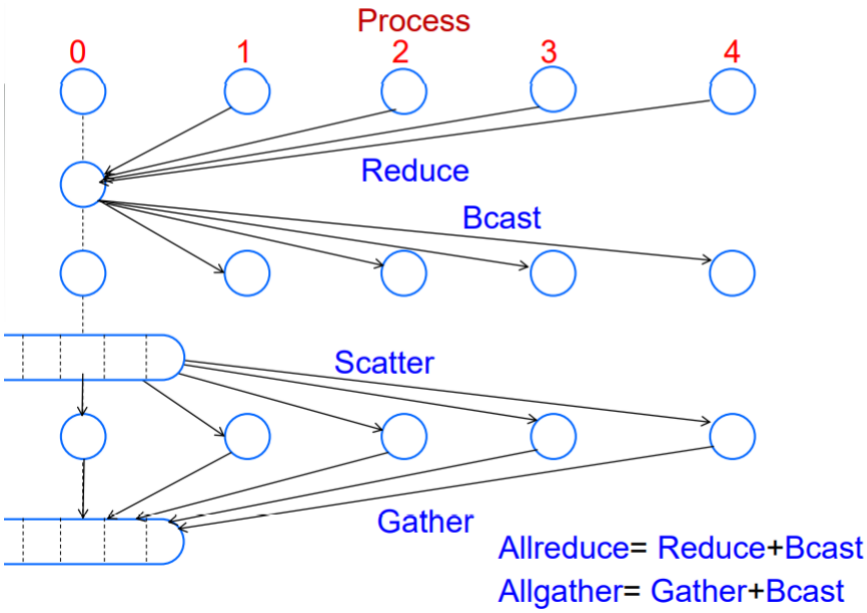
集合通讯

通信子中所有进程都需要调用同一个集合通讯函数，函数的参数要一致。

```
int MPI_Reduce(void* intput_p, void* output_p, int count,
               MPI_Datatype type, MPI_Op op,
               int dst_process, MPI_Comm comm); // 归约到单一进程

int MPI_Bcast(void* data_p, int count, MPI_Datatype type,
              int source_pro, MPI_Comm comm); // 广播变量

// 只用于count被进程数整除时
int MPI_Scatter(...);
int MPI_Gather(...);
```



派生数据类型

降低通讯开销，一次发送整组数据。

```
int MPI_Type_create_struct(int count, int blocklengths[],
                           int displacements[], MPI_Datatype types[],
                           MPI_Datatype* new_type_p);
int MPI_Get_address(void* loc_p, MPI_AInts* add_p);
// 必要动作commit/free
int MPI_Type_commit(MPI_Datatype* new_t_p);
int MPI_Type_free(MPI_Datatype* old_t_p);
```

性能分析

多次运行取最好结果。

```
int MPI_Barrier(MPI_Comm comm); // 也是集合通讯函数
double MPI_Wtime(); // MPI并行程序计时
GET_TIME(double now); // 串行程序计时，宏定义于timer.h
```

安全性

发送-接受顺序不当，形成死锁。

- 使用MPI_Ssend检查，换用MPI_Sencrecv，MPI_Sendrecv_replace.

- 重新设计通讯流程。

第四章 Pthreads

全局变量共享，线程函数局部变量私有。

```
int pthread_create(pthread_t* thread_p,
                   const pthread_attr_t* attr_p,
                   void* thread_func,
                   void* arg_p);
// void* thread_func(void* args_p);

int pthread_join(pthread_t thread, void** ret_val_p);
```

解决竞争条件

- Busy-waiting：关闭编译优化
- Mutexs（互斥锁），不能控制临界区执行的顺序。

  ```
  int pthread_mutex_init(pthread_mutex_t* mutex_p,
                         const pthread_mutexattr_t* attr_p); // attr = NULL
  int pthread_mutex_destroy(pthread_mutex_t* mutex_p);

  int pthread_mutex_lock(pthread_mutex_t* mutex_p);
  int pthread_mutex_unlock(pthread_mutex_t* mutex_p);
  ```

  在矩阵乘法等顺序相关的问题中不够方便。
- Semaphores（信号量），特殊的`unsigned`
  - 0对应于加锁；
  - $\geq$ 1对应于解锁

  ```
  int sem_init(sem_t* semaphore_p, int shared, int initial_val);
  // 后两个参数传0即可

  int sem_destroy(sem_t* semaphore_p);

  int sem_post(sem_t* sem_p); // a++
  int sem_wait(sem_t* sem_p); // wait until a>0; a--;
  ```

Barriers

```
int pthread_barrier_init(pthread_barrier_t* barr_p,
                         const pthread_mutexattr_t* attr_p,
                         unsigned count); // 解除barrier需要的线程数

int pthread_barrier_wait(pthread_barrier_t* barr_p);
int pthread_barrier_destroy(pthread_barrier_t* barr_p);
```

实现方法：ch4.2 page 5~

- Busy-waiting + a Mutex
- semaphore + a Mutex
- semaphores

- condition variables

Condition Variables

```
int pthread_cond_wait(pthread_cond_t* cond_var_p,
                      pthread_mutex_t* mutex_p);
// pthread_mutex_unlock(&mutex_p);
// wait on signal of cond_var_p
// pthread_mutex_lock(&mutex_p);

int pthread_cond_signal(pthread_cond_t* cond_var_p);
int pthread_cond_broadcast(pthread_cond_t* cond_var_p;);
```

读写锁

```
int pthread_rwlock_rdlock(...);
int pthread_rwlock_rwlock(...);
int pthread_rwlock_unlock(...);
```

线程安全

使用线程安全的C库函数。Pg 43

```
rand -> rand_r;
strtok -> strtok_r;
```

第五章 OpenMP

共享内存系统，每个核都能访问所有的内存。

基本组件：

- Complier Directives
- Run-Time Library Routines
- Environment Variables

```
#pragma omp parallel private(list) shared(list) \
    default(shared | none) \
    reduction([operator]:[var list])\
    num_threads(integer-expression)
```

一个主线程和`thread_count-1`个从线程。

线程数设置优先级 ch5 pg 22

```
int omp_get_thread_num();  // get tid
int omp_get_num_threads();  // get the number of threads

# pragma omp parallel for ..
for // ...

# pragma omp parallel
{
 # pargma omp for  // 不创建线程,而是将循环分配给线程
```

```
    for // ...
}

double omp_get_wtime();
#pargma omp barrier
```

使用`parallel for`, 则循环中不能有`break`, `return`.

循环体必须是明确的. pg 44

注意循环体之间不能有**数据依赖**.

Schedule

- static: 在编译器分配循环, 每次分配`chunksize`
- dynamic: 在运行时分配循环, 每次分配`chunksize`,默认`chunksize`为1.
- guided: 在运行时分配循环, 每次分配剩余循环/线程数, 最少分配`chunksize`, 最后一次分配可能少于`chunksize`
- auto/run-time: 由编译器/运行时系统决定.

默认策略: `schedule(static, for_loop_cnt/thread_cnt)`.

- 相比`static`, 优先尝试`dynamic`
- 如果循环的计算需求线性增加(减少), 使用小`chunksize`的`static`.

解决竞争条件

```
# pragma omp critical(name) // 临界区
// ...

# pargma omp atomic
x <op> = <expression>;
// x += y++; // unpredictable
```

OpenMP提供两种锁, **simple**/nested

```
void omp_init_lock(omp_lock_t* lock_p);
void omp_destroy_lock(omp_lock_t* lock_p);

void omp_set_lock(omp_lock_t* lock_p);
void omp_unset_lock(omp_lock_t* lock_p);
```

与Pthreads一样存在误共享和线程安全问题.

第六章 cuda

基本函数

```
__global__ void kernel(...) {  // 必须 return void
    __shared__ int arr[];  // 同一个block内共享

    __syncthreads();  // block内线程同步
}
```

```
int main(){
    //...
    cudaMalloc(...);
    cudaMemcpy(...);
    // ...
    kernel<<<griddim, blockdim>>>(...);
    // ...
    cudaFree(...);
    return 0;
}
```

- kernel的开启是异步的
- `cudaMemcpyAsync`异步拷贝内存
- `cudaDeviceSynchronize`阻塞直到所有cuda调用结束
- `cudaGetLastError`获取最后一个错误。

线程执行模型

- grid -> block -> threads, GPU -> SM(multicore processor) -> core
- block之间不同步，多个block可以放在同一个SM上

wrap

在SM内部，线程以warp为单位，32个一组进行发射。

- warp内线程并行执行，只能同时执行相同语句。
- divergence：代码分支导致warp内的不同执行路径被串行化。

原子操作

更新共享内存或global内存可能存在竞争条件。

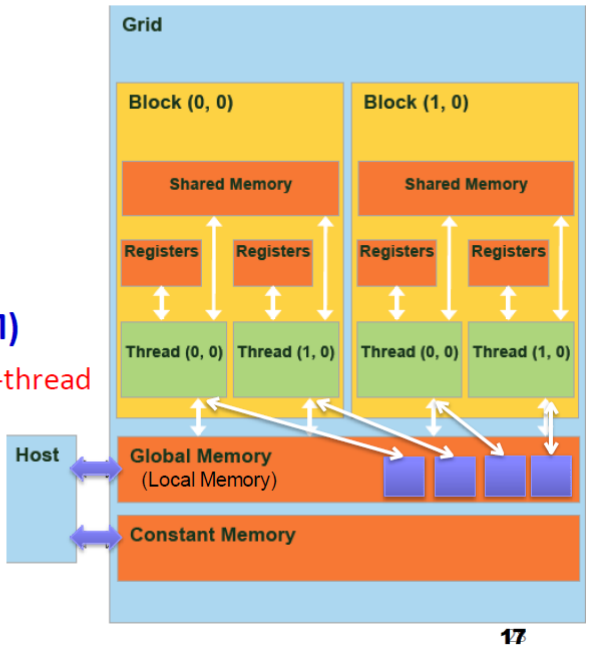
```
atomicAdd(...);
```

GPU体系结构

- on-chip vs. off-chip

GPU Memory Hierarchy

- **Registers**
  - Read/write per-thread
  - Low latency & High BW
- **Shared memory**
  - Read/write per-block
  - Similar to register performance
- **Global/Local memory (DRAM)**
  - Global is per-grid & Local is per-thread
  - High latency & Low BW
  - Not cached
- **Constant memory**
  - Read only per-grid
  - Cached



- local memory由各线程私有，store操作缓存于L1
- 编译器决定变量的存放位置
- 寄存器不可编号索引

性能优化

Memory Coalescing

off-chip memory (local/global/constant) 以chunk为单位读取

- 如果不使用整个chunk，内存带宽就被浪费了
- 使用数组结构体，而不是结构体数组
- 步长为1的访问模式

Shared Memory Bank Conflicts

共享内存以bank的方式进行存储

- 不同线程可以访问不同的bank，同一个wrap内的线程可以访问同一个地址
- 如果多个线程访问同一个bank的不同内容，将产生bank conflict

解决办法

- 改变共享内存访问模式
- memory padding

control flow divergence

同一个warp内的线程执行不同代码需要串行；

不同warp可以串行执行不同的代码。

- iteration divergence

Occupancy

任何时间，每个warp scheduler只能处理一个warp，而warp切换几乎没有开销。

- 大量线程隐藏访问延迟，细粒度并行

第七章

负载分析

对程序的特征信息(负载特征)进行测量和分析的过程.

- 明确程序瓶颈, 帮助程序优化
- 辅助设计
- 选择测试程序集

SPEC CPU: CPU性能的标准测试程序集

负载特征:

- 描述程序的行为, 负载分析的基础
- 计算、访存、通信、I/O

程序剖面(Program Profiling)

如串行程序中各个函数的执行时间，Page fault次数

并行程序中计算和通信时间比例，通信类型

- 提供统计信息，快速明确程序主要特征
- 缺少事件执行的先后顺序和详细信息

事件驱动：统计基本块个数

- 精确统计关注事件信息，收集信息具有重复性
- 开销大，影响程序执行
- 适合收集低频事件

采样技术：在固定的时间间隔对关注的事件进行测试

- 开销低，影响小
- 只适合收集高频事件
- 保证采样数足够大，可以多次运行累计采样
- 例子 Pg 48

工具：gprof, mpiP

事件轨迹(Trace)

按照程序执行顺序记录每条事件，获得详细事件序列。

- 产生的Trace量巨大，需要压缩。对程序干扰大。

硬件计数器

用于监视性能事件的寄存器。

- 低开销，非侵入，production-run

流水线功能单元：执行固定功能的电路

指标类别

- Out of Order Execution: instructions issued, instructions completed
- Speculative Execution: branch prediction hits and misses
- Instruction Counts and Functional Unit Status: Floating point/load/store ...

Calculations Pg 19-21:

$$IPC = \frac{\text{Total Instructions Exec}}{\text{Total Cycles}}$$

$$\text{L1 hit rate} = 1 - \frac{\text{L1 Misses}}{\text{Loads+Stores}}$$

$$\text{L2 hit rate} = 1 - \frac{\text{L2 Misses}}{\text{L1 Misses}}$$

工具：PAPI