并行复习2 并行硬件和并行软件

Cache相关工作原理及概念

冯诺伊曼瓶颈的概念，缓解方式有哪些（cache可以缓解、最底层的指令集并行和硬件级并行都可以缓解

冯诺伊曼结构

- 五部分: 控制器, 运算器, 存储器, 输入设备, 输出设备
- 控制器和运算器继承在cpu中, 使用不同的寄存器实现对应功能, ALU(算数逻辑单元)和程序计数器

冯·诺伊曼瓶颈

冯诺依曼架构不区分数据与指令，将两者放在同一内存中。

由于指令与数据放在同一内存带来的CPU利用率（吞吐率）限制就是冯诺依曼瓶颈。

缓解方式

- 最底层的指令级并行和硬件级并行
- 在CPU和主存之间提供一个缓存

Cache高速缓冲存储器（缓存）

为什么需要cache?CPU处理数据的速度非常快，虽然内存的读写速度也不慢，但是相对于CPU它的速度就显得太慢了，所以如果单纯地让CPU对内存进行读写，所消耗的时间绝**大部分是在内存对数据的处理**上，而这时候**CPU就在空等**，浪费了资源，因此就需要在CPU与内存之间连接一个Cache来作为缓冲。

cache的特点：速度快，容量小

cpu对缓存中的数据进行存取花费时间更短

缓存可以集成在cpu上，也可以在cpu外

缓存存储依照准则：

- **空间局部性，时间局部性**

1.空间局部性（Spatial Locality）：

2.定义： 空间局部性指的是程序访问内存时，如果一段内存地址被访问，那么附近的内存地址很可能也会被访问。

3.例子： 在数组遍历中，如果你访问了数组的一个元素，由于数组元素通常在内存中是相邻存储的，因此下一个或前一个元素也很可能被访问。这就是空间局部性的体现。

4.时间局部性（Temporal Locality）：

5.定义： 时间局部性指的是程序在一段时间内多次访问相同的内存地址。

6.例子： 在循环结构中，如果在一个循环迭代中访问了某个变量，下一次迭代中很可能再次访问同一个变量。这就是时间局部性，因为相同的内存地址在短时间内被多次访问。

综合例子：

考虑以下代码片段，其中有一个简单的循环遍历数组：

```
int array[1000];

for (int i = 0; i < 1000; ++i) {
array[i] = array[i] * 2;
}
```

7.空间局部性： 当访问array[i]时，附近的array[i-1]和array[i+1]也很可能被访问，因为数组元素在内存中是连续存储的。

8.时间局部性： 在每次循环迭代中，都会访问相同的array[i]，表现出时间局部性，因为相同的内存地址在短时间内被多次访问。

- 多级缓存（一般三级）

离cpu越近，速度越快，容量越小

低层Cache（更快、更小）通常是高层Cache的Cache。也有低层不复制高层Cache的设计。

- 缓存命中、缺失：

命中 : 在多级缓存中找到了相应的数据　缺失 : 在多级缓存中未找到

- 往缓存中存放数据时, 并不是一次放一个, 而是放一个数据块(一个高速缓存行)

第一次访问, 缓存中没有, 需要在主存中找到需要使用的数据, 然后把这个数据附近的一块数据都放入缓存中.

- 缓存和主存中数据不一致(缓存一致性)

写直达(write-through) : 一旦发生不一致, 立刻相应更改主存中数据

写回(write-back) : 将缓存中不一致的数据标记为脏数据, 待整个高速缓存行里的数据都执行完毕后, 再一次性写入主存中.

- 行主序, 列主序对访问数据效率的影响

并行 多线程相关概念

线程、进程的概念

1. 进程

进程是运行着的程序的一个实例。

一个进程包括如下实体：

☒ 可执行的机器语言程序

☒ 一块内存空间

☒ 操作系统分配给进程的资源描述符

☒ 安全信息

☒ 进程状态信息

2. 多任务

单个处理器同时运行多个任务。

3. 线程

线程包含在进程中。

一个进程被划分为一些相互独立的任务，每个运行着的独立任务称做线程。

线程为程序员提供了一种机制，将程序划分为多个大致独立的任务。

当某个任务阻塞时能执行其他任务，此外线程间的切换比进程间切换更快。

- fork 派生线程, join合并线程, master线程控制整个程序逻辑, 比如线程的派生和合并

硬件级别的单核多线程

- 细粒度 : 分时间片平均轮换执行各个线程

优点 : 一个线程遇到阻塞时, 别的线程仍然可以继续执行

缺点 : 当一个线程阻塞时, 并不能立刻切换另外一个线程, 而是要等待它所分配的时间片时间结束. 造成了一定的资源浪费

细粒度是相对的, 比之前的指令集还是要粗的

- 粗粒度 : 不再分时间片, 只有当某个线程遇到较长时间阻塞时才切换另一个线程

优点 : 线程之间切换少了, 切换浪费的时间少了

缺点 : 但是线程切换仍然需要浪费时间, 对于阻塞时间的判断也需要浪费一定时间

- 同步多线程(SMT, 超线程) : 是细粒度多线程的变种。模拟多核, 让一个核同时(真正意义上的同时)运行多个线程. 通过分配计算资源, 让不同的计算资源运行不同的线程

一个核上不会同时运行太多线程

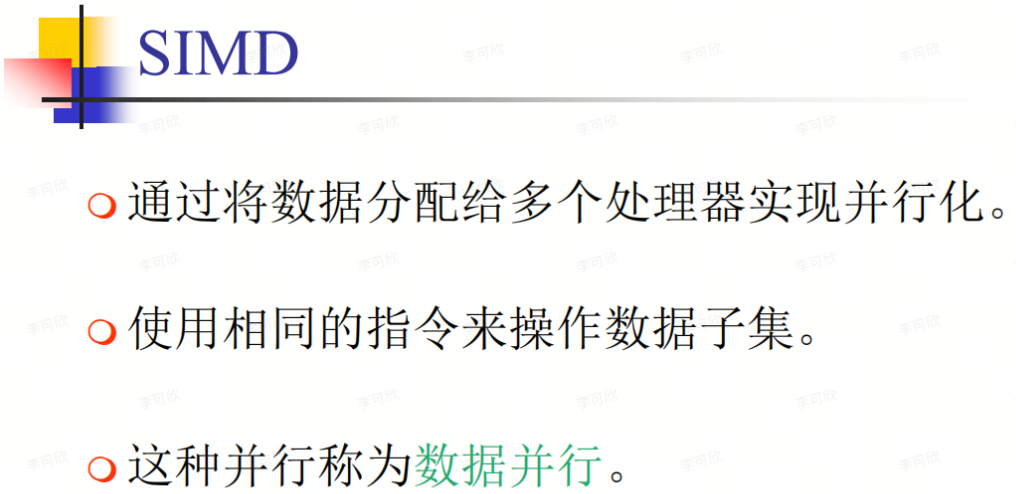
Flynn分类法相关概念

SIMD/SISD/MIMD/MISD S：single M:multiple

SIMD:单指令多数据流

MIMD：多指令多数据流

SIMD



MIMD

ALU：逻辑计算单元

SIMD同步执行，MIMD每一个处理器上都有自己的时钟，可以独立执行，并不同步。

MIMD

- 支持同时多个指令流在多个数据流上操作。
- 通常包括一组完全独立的处理单元或者核，每个处理单元或者核都有自己的控制单元和ALU。

SIMD、MIMD(共享内存\分布式内存 网络连接等)

共享式

一种总线、一种Crossbar（交叉开关矩阵）

带宽、等分带宽的概念的几个概念

共享内存系统

- 一组自治的处理器通过一个互联网络与内存系统进行连接
- 每一个处理器可以访问内存中的每一块空间
- 处理器之间通过共享数据进行隐式通信
- 包括一个或多个多核处理器

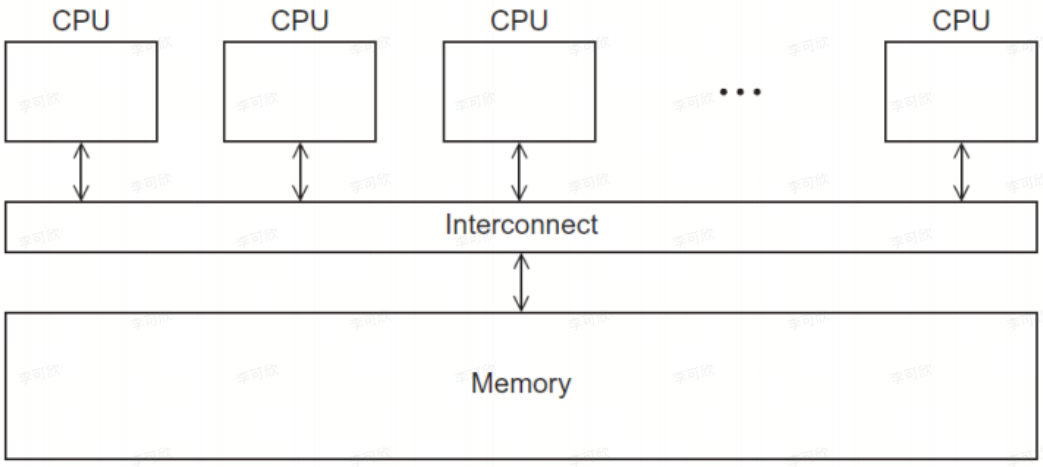
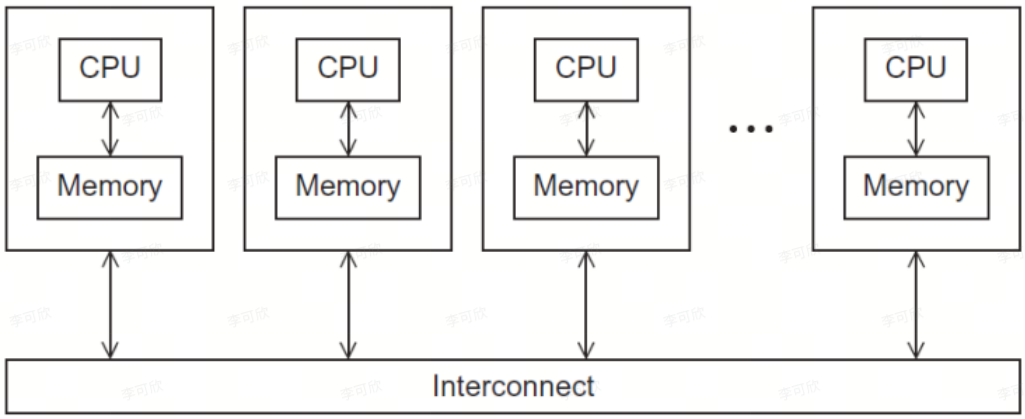


图 2.3

分布式内存系统

- 集群
- 每一个CPU单独对应一个内存空间, 构成一节点, 节点之间通过一个互联网络相互连接



互联网络

在分布式内存系统和共享内存系统中都扮演了一个决定性的角色

共享内存互连网络

- 总线

一条总线, 连接各个设备, **同一时刻总线只能被一个设备使用**

当设备数量增多时, 会设计总线争夺, 效率变低

设计简单，灵活，成本低。

- 交叉开关矩阵crossbar

多个cpu可以同时访问内存

设计复杂，灵活度低，成本高

分布式内存互连网络

1. 直接互连

每个交换器与一个处理器-内存对直接相连，交换器之间也互相连接。

2. 间接互连

交换器不一定与处理器直接连接

等分宽度

用来衡量一个互联网络连通性的标准(同时通信的链路数目)。

带宽，等分带宽

- 带宽 : 链路上传输数据的速度, MB/s, Mb/s
- 等分带宽 : 带宽 * 等分宽度, 用来衡量一个网络的数据传输速度

并行算法设计(竞争条件\数据依赖\同步)

比串行要多考虑哪些方面（竞争条件\数据依赖\同步\原子计算）

（竞争条件\数据依赖\同步\原子计算）的概念和词要记住（很可能出现在简答

概念定义

- 原子性：一组操作要么全部执行，要么全部不执行。
- 互斥：任何时刻都只有一个线程在执行。
- 竞争条件：执行结果依赖于两个或更多事件的时序。多个线程同时访问共享资源，且最终的执行结果取决于线程执行的顺序。
- 数据依赖：两个内存操作的序, 为了保证结果的正确性, 必须保持这个序。一个操作的执行依赖于另一个操作的结果。
- 同步：在时间上, 强制使各个进程/线程在某一点相互等待, 确保各进程/线程的正常顺序和对共享可写数据的正确访问。
- 障碍：阻塞线程继续执行, 在此程序点等待, 直到所有参与线程都到达障碍点才继续执行。

并行算法性能分析(加速比\效率\可扩展性\阿姆达尔定律)

- 串行算法的评价 : 算法时间复杂度标示为输入规模的函数
- 并行算法的评价 : 除了输入规模, 还要考虑处**理器数目**, 处理器相对运算速度, 通信速度.
- 评价标准
  - 运行时间
  - 加速比 : 与最优串行算法做比较

性能评价标准

- 运行时间

串行时间 : Ts, 算法开始到结束的时间

并行时间 : Tp, 并行算法开始到最后一个进程结束所经历的时间

- 并行算法总额外开销

T0 = pTp- Ts

- **加速比**

S = Ts/Tp

p为线程数

S=最优串行算法时间/并行算法时间

一般S≤p

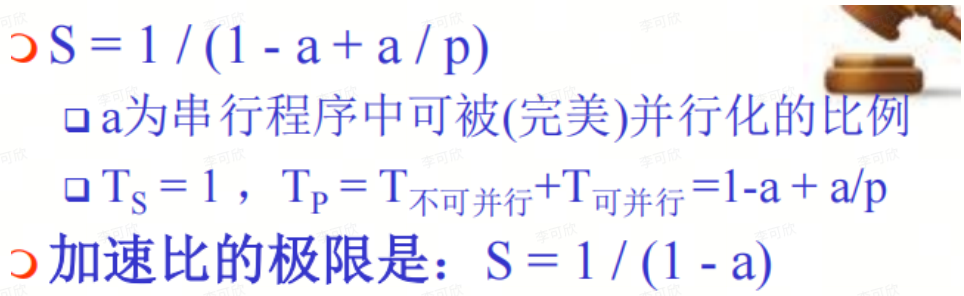
S=p : **线性加速比**

S>p : 超线性加速比, 在实践中可能出现

- **阿姆达尔定律**

除非一个串行程序的执行几乎全部都并行化, 否则不论多少可利用的核, 通过并行化产生的加速比都是会受限的。

加速比只和a,p有关，与串行时间无关。



- **效率**

度量有效计算时间

把核数引入计量指标。

$$E = S / p = T_S / (p*T_P)$$

理想情况=1，正常0～1。

因为理想情况 S = p

- **可扩展性**

若某并行程序核数(线程数/进程数)固定，并且输入规模也是固定的，其效率值为E。现增加程序核数(线程数/进程数)，如果在输入规模也以相应增长率增加的情况下，该程序的效率一直是E(不降)，则称该程序是**可扩展的**。

- 强可扩展：我们希望，保持问题规模不变时， 效率不随着线程数的增大而降低， 则称程序是可扩展的（称为**强可扩展的**）。但这往往是难达到的。
- 弱可扩展：退求其次：问题规模以一定速率增大，效率不随着线程数的增大而降低，则认为程序是可扩展的（称为**弱可扩展的**）

某程序串行版本运行时间为$T_S = n$秒, 这里n也为问题规模, 该程序某一并行版本运行时间为$T_P = n / p + T_O$。该并行程序是否可扩展？（假设T_O为常数，不随p变化而变化）

- $E = T_S / (p*T_P) = \frac{n}{p*(n/p + TO)} = \frac{1}{1+\frac{p}{n}*\frac{T_o}{1}}$
- 要保持E不变，当 p' = kp 时，只需问题规模等比例增大即可，即 n' = kn。
- 因此，该并行程序是可扩展的。

- 可扩展性是高性能并行算法追求的主要目标
  - 度量并行系统性能的方法之一
  - 度量并行体系结构在不同系统规模下的并行处理能力
  - 度量并行算法内在的并行性
  - 利用系统规模和问题规模已知的并行系统性能来预测规模增大后的性能