Multiprocessor

Parallel Computing Background

Parallel Computing

并行本质：同一事件做多样事情

并行最主要的目的就是提升性能（提升吞吐率），其余目的还有

- 降低能耗，4N 个工作在 F/4 频率的单元能耗小于 N 个工作在 F 频率的单元
- 提升可扩展性，涉及单个高性能单元的难度是较高的
- 提升可靠性，有冗余

并行的种类有

- ILP：如流水线，OoO，VLIW 等
- DLP：如 SIMD
- TLP：多线程，多处理器

有两种 TLP 的方式

- 将一个单独的任务划分成多个 task，如并行编程，如果问题本身有并行性是很容易划分的，或是线程级的划分
- 同时处理多个无关的任务，如云计算上不同用户提交的计算任务，提升系统整体 throughput 但是不影响单独任务的性能

多处理器也可以按核之间的联系划分为两种

- 联系较松
  - 没有共享的全局地址空间
  - 网络形式组织通信
  - 通常通过 message passing 模型编程
- 联系紧密
  - 共享全局地址空间
  - 传统的多处理：SMP (symmetric multiprocessing)，即多核/多线程处理器
  - 编程类似单处理器，但是对于共享数据的操作要同步

Tightly-Coupled MP

设计中主要的问题有

- 共享内存的同步，如锁以及一些原子操作
- 私有 cache 一致性
- 内存的一致性
- 共享资源管理
- 处理器间的通信：interconnect

编程时也要考虑如何划分任务和任务间的同步问题

最基本的是要确保正确性，然后提升性能

一般加速不会超过线性，即 P 个处理单元难以加速到 P 以上

Utilization & Redundancy & Efficiency

考虑所有处理器都在处理并行计算

utilization：处理器利用率

$$U = \frac{\#Operation}{\#processors \times time}$$

redundancy：处理器为了并行做了多少额外工作

$$R = \frac{\#Operation_p}{\#Operation_s}$$

一般都大于 1

efficiency：效率

$$E = \frac{\#Operation_s}{\#processors \times time_p} = \frac{U}{R}$$

并行加速的瓶颈由串行部分决定：Amdahl's Law，现实中的并行一般不是完美的并行，因为有同步和资源共享等各种问题

Bottlenecks in Parallel Portion

同步

- 争抢：操作共享资源的指令不能并行，需要锁等同步原语，操作共享资源的部分称为临界区
- 协作：一个任务可能需要其他任务提供的资源，如生产者-消费者模型，或是 barrier，必须等所有任务到达 barrier 才能继续推进

不平衡的任务分配：并行任务可能长度不同

资源争抢：cache，内存等硬件资源

pipelining：如果将循环中的计算划分成不同的任务，最费时的会成为瓶颈

Difficulty in Parallel Programming

在程序没有显性的并发特点时编程是比较有挑战的

除此之外还要保证并发的版本不出错

一般来说对于代码的不同部分有不同的硬件要求

- 串行部分需要一个强力的大核
- 并行部分需要一系列小核

这两者是冲突的，需要 tradeoff

有堆叠少数大核（IBM Power）或者大量小核（Sun Niagara）的方法

可以结合这两种方法：ACMP (Asymmetric Chip Multiprocessor)，即提供一个大核和众多小核，分别运行程序的串行部分和并行部分，神威太湖之光就是这种策略

Heterogeneity

即所谓异构（非对称），是系统设计比较常见的思想

比起有多种同类资源，提供不同类的资源效果更好，适应性强，且可以定制化

为什么要非对称设计

- 系统中不同的 workload 有不同的特性，甚至同一应用的不同阶段/同一应用的不同输入，包括
  - 局部性
  - BP
  - 数据依赖
  - 串行部分
- 系统需要满足各种情况的要求。设计永远不会只为了一个目标，要考虑多种情况，包括性能，能耗，公平性，可靠性……

symmetric design 的一招鲜吃遍天是很难实现的，而非对称设计可以根据需求选择最适合的资源

现在的非对称设计有

- GPU 集群：一个高性能 CPU 与多块 GPU
- FPGA：使用 FPGA 在数据中心加速
- TPU

非对称设计连接了完全通用和完全特化之间的 gap

tradeoff

- 优点
  - 能在多种指标上优化
  - 能更好适应不同的 workload
  - 能提供特化的性能优势以及泛化的灵活性
- 缺点
  - 设计和验证更复杂
  - 更难管理，要调度各个部分
  - 需要在各部分间交换数据