并行复习要点

By Shelly Tang 2020/8/17~2020/9/3

CH1：并行计算概览

1. 什么是并行计算（P14）

并行计算可以简单定义为同时利用多个计算资源解决一个计算问题程序运行在多个CPU上；

- 一个问题被分解成离散可并发解决的小部分；
- 每一小部分被进一步分解成一组指令序列；
- 每一部分的指令在不同的CPU上同时执行；
- 需要一个全局的控制和协调机制；

2. 并行计算的优势：（P18）

（1）节省时间和花费

① 理论上，给一个任务投入更多的资源将缩短任务的完成时间，减少潜在的代价

② 并行计算机可以由多个便宜、通用计算资源构成；

（2）解决更大、更复杂问题

很多问题很复杂，不实际也不可能在单台计算机上解决

（3）实现并发处理

① 单台计算机只能做一件事情，而多台计算机却可以同时做几件事情；

② 例如协作网络，来自世界各地的人可以同时工作；

（4）利用非本地资源

当本地计算资源稀缺或者不充足时，可以利用甚至是来自互联网的计算资源。

（5）更好的发挥底层并行硬件

1).现代计算机甚至笔记本都具有多个处理器或者核心；

2).并行软件就是为了针对并行硬件架构出现的；

3).串行程序运行在现代计算机上会浪费计算资源；

3. 并行计算的主要用途（P22-24）

科学和工程计算、工业和商业应用、全球范围的应用

4. 并行计算的主要推动力

（1）**应用发展趋势**

- 在硬件可达到的性能与应用对性能的需求之间存在正反馈（P26）
- 大量设备、用户、内容涌现（P27）
- 大数据（P28）

- 云计算的兴起；廉价的硬件，应用弹性的扩展；应用种类繁多，负载异构性增加；（P30）

（2）**架构发展趋势**

- 大规模集成电路（VLSI）（P31）
  - VLSI最的特色是在于对并行化的利用，不同的VLSI时代具有不同的并行粒度：bit并行-> 指令水平的并行 -> 线程水平的并行（P31-32）
- Moore定律（P36-39）
  - CPU主频增长越来越缓慢，Moore定律会失效：发展趋势不再是高速的CPU主频，而是"多核"
  - 如何提高CPU的处理速度（P39）
  - 未来属于异构、多核的SOC架构，SOC = System On Chip
  - Moore's law新解（P55）

5. 并行计算的粒度

函数级并行、线程级并行还是进程级并行

6. 并行计算的难点（P63）

（1）找到尽可能多的并行点（Amdahl's Law）

（2）粒度：函数级并行、线程级并行还是进程级并行

（3）局部性：并行化后是否能够利用局部数据等

（4）负载均衡：不同线程、不同core之间的敷在分布

（5）协作与同步

（6）性能模型

7. Amdahl's law

Speedup加速比 = 1线程执行时间/n线程执行时间

= 1/ [(1-p) + p/n]，p为并行的分数

CH2：并行架构

1. Flynn's 并行架构分类（P10）

- SISD：单指令单数据流（顺序执行）（P11）
- MISD：多指令单数据流（e.g. OpenMP)(P13)
- **SIMD**：单指令多数据流（e.g. CUDA/AVX 向量计算）(P12)
- **MIMD**：多指令多数据流（多节点并行计算）

2. 什么是pipeline?

- 流水线（P19)：洗衣服
- overlapping individual parts of instructions重叠指令的各个部分（P17)
- limits (P22), 数据冒险（P23）

3. 有哪些形式的指令级并行？（P17)

- 流水线pipelining：重叠指令的各个部分
- 超标量执行superscalar execution：同一时间做多个事情
- 超长指令级架构VLIW：让编译器指定哪些操作可以并行运行

- 向量处理Vector processing：指定类似（独立）的操作组
- 乱序执行OOO：允许长操作的发生（即可提前结束，不用等）（P24）
- 现代ILP：动态规划，乱序执行（P25-26）

4. 什么是 Pthreads？（P46）

- the POSIX线程接口
    - 系统调用以创建和同步线程
    - 在类似UNIX的OS平台上应该相对统一
- Pthreads支持：创建并行、同步、（隐式支持）通信（只是堆适用、栈不适用）

#补充：

- **线程调度（P50）**
- **多线程执行（P51）：硬件多线程**
- **结合ILP&TLP：面向ILP的处理器能否从利用TLP中受益？（P52）**
    - 由于停顿或代码依赖性，功能单元通常在为ILP设计的数据路径中处于空闲状态
    - TLP用作独立指令的来源，可能使处理器在停顿期间保持繁忙
    - TLP用于占用功能单元，否则当没有足够的ILP时它们将处于空闲状态
    - 这叫做"同时多线程"，或者intel中的**超线程**
    - 线程数增多，则计算部件的利用率增大。（但冲突也会增多）（P54）

5. 内存局部性原则有哪些？（P58）

- **时间局部性**temporal locality：如果引用了某个项目，则往往会很快再次引用该项目（例如，循环，重用）
- **空间局部性**spatial locality：如果引用了某个项目，则地址附近的那些项目往往会很快被引用（例如，直线代码，数组访问）

#补充：评价内存系统性能的指标：（P56）

- **延迟latency**：指从发出内存请求到处理器中数据可用的时间。
- **带宽bandwidth**：存储系统将数据送到处理器的速率。

6. 内存分层？（P60,61）

- 分了这些层：寄存器、高速缓存、主存、固态存储器、备份硬盘存储器
- 随着与CPU的距离越来越大（存储器的层次结构从上到下），速度越来越小，存储容量越来越大，价格越来越低廉，访问频率递减（局部性原理）。

7. Caches 在内存分层结构中的重要作用(P66)

- 高速缓存是处理器与DRAM之间的小型快速存储元素。
    - 充当低延迟高带宽的存储。
    - 如果重复使用一条数据，则可以通过cache减少此存储系统的有效延迟。
- 高速缓存满足的数据引用比例称为**高速缓存命中率**。
- 存储系统上的代码所达到的高速缓存命中率通常决定其性能。
- **注：cache计算例题（P67）**

8. 新型存储系统的构成?

- 磁盘&内存相结合（P73）
- 存储类型内存 Storage Class Memory(SCC)
- 在DRAM基础上，增加了MRAM，FeRAM，PCM等部件进行优化，使得获取数据的时间减少。（P77）

9. 什么是并行架构？（P87）

- 并行计算机是处理元素的**集合**，这些**元素**可以协作快速解决大型问题。
- 资源分配：
  - 集合多大？元素有多强大？内存需要多少？
- 数据访问，通信和同步
  - 这些元素如何合作和沟通？处理器之间如何传输数据？合作的抽象和原语是什么？
- 性能和可扩展性
  - 如何将其转化为性能（即 性能如何？）？ 它如何扩展？
- **补充**：并行体系结构（并行架构）是分布式计算的子类，其中的进程都在工作以解决相同的问题。并行体系结构是指许多指令能同时进行的体系结构 。

10.MIMD 的并行架构包括哪些实现类型?

- 共享内存：通过内存通信(P90)（同节点）
  - 有“硬件全局缓存一致性” 和 “非硬件全局缓存一致性”两种
  - 例子1：对称多处理器 Symmetric Multiprocessor(P91)
  - 例子2：非统一性共享内存 Non-uniform shared-memory with separate I/O through host
- 消息传递message passing(P90)：通过消息传递来通信（跨节点）
  - 应用程序在节点之间发送显式消息，以便通信
  - 例子：cluster集群
- 对于大多数计算机，共享内存基于消息传递网络构建。

11. MPP 架构的典型例子及主要构成？（P93)

- 主要构成是大点，下面的小点是典型例子

CH3： 并行编程模型

1. 什么是并行编程模型？ （P17)

- 程序员在编写应用程序中使用的，可并行
- **指定通信和同步**
- **补充**： 并行编程模型是并行计算机体系架构的一种抽象，它便于编程人员在程序中编写算法及其组合。

2. 并行编程模型的主要包括哪些类型？主要特点是什么？ （P19)

- **消息传递message passing**
  - 封装本地数据的独立任务
  - 任务通过交换消息进行交互
- **共享内存shared memory**
  - 任务共享一个公共地址空间

    - 任务通过异步读写该空间进行交互（读写共享变量）
- **数据并行data parallelization**
    - 任务执行一系列独立的操作
    - 数据通常跨任务平均分配（均匀划分）
    - 也称为“尴尬（并行）”

3. 并行编程模型主要包括哪几部分?

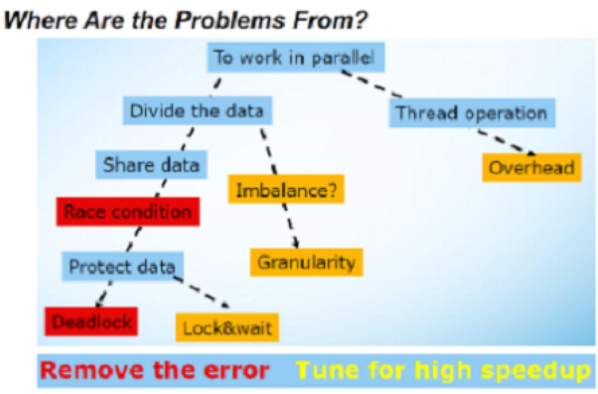
- 控制Control
    - 如何创建并行性？应该以什么顺序进行操作？不同的控制线程如何同步？
- 命名Naming
    - 什么数据是私有数据还是共享数据？如何访问共享数据？
- 操作operations
    - 什么是原子操作?
- 代价Cost
    - 如何计算运营成本?

注：并行编程模型parallel programming model位于并行应用之下，在OS，compiler和库之上。

4.共享内存模型有哪些实现？ （P32）

- 本机编译器和/或硬件映射用户程序变量到实际的内存地址，这是全局的
    - 在独立的SMP机器上，这很简单
- 在分布式共享内存机器（例如SGI Origin）上，内存物理分布在机器网络中，但是通过专用硬件和软件实现了全局性。
- **补充：共享内存模型的好处（P29）&缺点（P30）**
    - **好处**：从程序员的角度来看，此模型的优点是缺少数据“所有权”的概念，因此无需显式指定任务之间的数据通信。程序开发通常可以简化。
    - **缺点**：难以理解和管理数据局部性
        - 将数据保持在使用该处理器的本地，可以节省内存访问，高速缓存刷新以及在多个处理器使用同一数据时发生的总线流量。
        - 不幸的是，控制数据局部性很难理解，并且超出了普通用户的控制范围

5. 造成并行编程模型不能达到理想加速比的原因？ （P43）

- 有串行的部分，不能加速，所以达不到“理想加速比”
- 性能的提高无法抵消掉管理开销（线程创建开销）：特别是在问题规模较小时。
- 

    - 线程开销，数据划分不均衡，共享数据带来的竞争问题，保护带来的死锁问题

6. 任务（Task）和线程（Thread）之间的关系？ （P47&48)

- 任务由数据及其处理组成，任务调度器会将其附加到要执行的线程上
- 任务操作比线程操作便宜很多（好处）
- 通过窃取轻松平衡线程之间的工作负载，线程均衡度高。（好处）
- 适用于集合类型的数据结构，如list, map, tree.
- 注：**进程**process=内存+线程 (P26)
- 注意事项：（P48）
  - 任务多于线程：更灵活地安排任务，轻松平衡工作量
  - 任务中的计算量必须足够大以抵消管理任务和线程的开销
  - 静态调度：任务是单独的独立函数调用的集合，或者是循环迭代
  - 动态调度：任务执行长度是可变的并且是不可预测的；可能需要一个额外的线程（额外调度器）来管理一个共享结构以承担所有任务（task并不由某个线程固定调度）

7. 什么是线程竞争？如何解决？ （P49）

- **线程相互竞争以获取资源**
  - 假定执行顺序，但不能保证
- 存储冲突是最常见的
  - 多线程并发访问同一内存位置，至少有一个线程正在写入（确定性race）
- 分为：确定性竞争和数据竞争
  - 确定性竞争：不管有没有锁（加锁也无法控制which进程先到）
  - 数据竞争：没锁的情况下的确定性竞争
- 有时可能并不明显
- 注意事项：**（解决方法）**
  - 控制临界区的共享访问：互斥和同步，临界区，原子操作
  - 变量作用域在线程本地：具有共享数据的本地副本；在线程堆栈上分配变量

CH4：并行编程方法论

1.什么是增量式并行化(incremental parallelization？ （P12)

- 研究串行程序（或代码段）
- 寻找瓶颈和并行的机会
- 尽量让所有处理器忙于做有用的工作
- 注：不断查看系统中有无可并行化的部分，有则并行化。当找不到可并行化的点/多次并行化之后性能无明显提升，则停止。

2.Culler 并行设计流程?

- 图（P16）
- **分解/解耦 decomposition：（P17-18)**
  - 将问题分解为可以并行执行的任务
  - 重点：分解成足够的tasks，使底层的执行单元处于忙碌状态（即硬件得到充分利用
  - 关键：识别依赖（或 减少依赖）
  - 负责分解的是：程序员

    - 自动分解串行程序是很难的
        - compiler必须分析程序，识别依赖
            - 但依赖包括控制依赖和数据依赖（后者在执行时才会出现）
        - 循环级别的自动化解耦做的比较好
- **分派 assignment（P19-20)**
    - 将任务分配给线程
    - 目标：负载均衡，减少通信代价
    - 可以静态发生或动态发生
    - 负责分配的是：languages 或 runtimes
- **编排orchestration（P22)**
    - 包括：
        - 实现通信
        - 必要时增加同步来保持依赖性
        - 优化内存数据结构（加速）
        - 调度任务
    - 目标：减少通信/同步开销，保持数据局部性，减少开销
- **映射mapping（P24)**
    - 将线程映射到硬件执行单元
    - 可以由“OS，编译器，硬件”来完成映射
    - mapping decisions：
        - 将**相关**的线程映射到同一个处理器中：
            - 增大局部性，数据共享，减少了同步/通信的开销
        - 将**不相关**的线程映射到同一个处理器中：
            - 利用线程之间行为的差异：使得资源争用概率减小

3.Foster 并行设计流程?

- 图（P26）：划分，通信，归并，映射
- **分解partitioning：（P27-28)**
    - 将计算和数据分成多个部分
    - 利用**数据并行性**
        - 数据/域分解，将数据分成几部分，确定如何将计算与数据关联
        - domain/data partitioning 按域/数据分解
            - **步骤**：首先，决定如何在处理器之间分配数据元素；其次，确定每个处理器应该执行的任务
            - **注意**：此时，任务有可能相同，也有可能不同
    - 利用**任务并行性**
        - 任务/功能分解，将计算分为几部分，确定如何将数据与计算相关联
        - functional/task decomposition 按任务划分
            - **步骤**：首先，在处理器之间划分任务；其次，确定哪个处理器将访问（读取和/或写入）哪些数据元素
    - 利用**流水线并行性**
        - 优化循环
        - 适用于：特殊的任务分解（应用在拥有明显流程的任务，有顺序的）

        - “组装线”并行
    - **注意事项：（P69）（详见第六点）**

- **通信communication（P70-71）**
    - 确定任务之间传递的值
    - 本地通信：与相邻节点
    - 全局通信：线程间通信，跨网络
    - **注意事项：(P71)**
        - 通信是并行算法的开销，我们需要将其最小化
        - 任务间的通信要均衡
        - 每个任务仅与一小部分邻居通信（尽量做到在节点内/CPU内通信，减少全局通信）
        - 任务可以并发进行通信和并发计算
- **整合/归并 agglomeration(P72)**
    - 将任务分组为更大的任务
    - 目标：提高性能，保持程序的可扩展性，简化编程（降低软件工程成本）
    - 在消息传递编程中，通常目标是每个处理器创建一个聚合任务
    - 意义：见下面的第6点
    - **注意事项：（P76)**
- **映射mapping（P77)**
    - 向处理器分配任务的过程
        - 集中式多处理器：由OS完成映射
        - 分布式内存系统：由用户完成映射（有调度器可以帮忙）
    - **矛盾**：最大化CPU利用率 & 最小化CPU间通信
        - 如：调度到不同的CPU中执行，可能会导致跨网络通信
        - **注意**：如果按照资源来做映射，即提高CPU利用率，则会更简单，也更常用；如果按照减少通信来做映射，则实现难度高，因为要考虑通信&通信量
    - **决策树**：P79-80（详见第7点）

4.按数据分解和按任务分解的特点?

- 详见上面第3点

5.并行任务分解过程中应该注意的问题有哪些?

- **注意事项：（P69)**
    - 任务书>=10*CPU数量
    - 最小化冗余计算和冗余数据存储
    - 基本任务的大小大致相同
    - 任务数量是问题规模的递增函数

6.整合的意义是什么？（P73-75)

- 可以提高性能：消除了合并为合并任务的原始任务之间的通信；组合发送和接收任务
- 可以保持可扩展性：要考虑随着数据增多，归并会否产生影响
    - 原因：局部性提高了，映射到硬件时的开销减小
- 可以降低工程成本：更多地利用现有的串行代码，减少开发并行程序的时间和开销

7.Mapping（映射）如何决策?

- 图（P81）
- **静态任务数**（一开始任务已经划分好，# tasks固定）
  - 结构化通信
    - 每个任务的计算时间恒定：汇总任务以最大程度地减少沟通，每个处理器创建一个任务
    - 每个任务的计算时间可变：周期性地将任务映射到处理器（循环往下映射，以维持CPU负载均衡
  - 非结构化通信
    - 用静态负载平衡算法
- **动态任务数**
  - 任务之间的频繁通信：使用动态负载平衡算法
  - 有许多短暂的任务：使用运行时任务调度算法
- **注意事项**：（P82）

8.熟悉一些并行设计的例子。

- 找数组中的最大数（P30）：“分解”中的按数据分解
- GUI的事件处理程序(P43)：“分解”中的按任务分解（任务用函数表示）（？）
- 3D渲染（P51)：分时复用task

补充：依赖图dependency graph

- 边表示数据/控件依赖性（P90)
  - 数据流：变量的新值取决于另一个值
  - 控制流：在条件出现之前，无法计算变量的新值

Chapter 5 OpenMP并行编程模型

1.什么是 OpenMP?

- 缩写：Open Multi-Processing （开放多处理过程）
- 用于编写多线程应用程序的API
  - 一组用于并行应用程序程序员的编译器指令和库例程
  - 大大简化了用Fortran，C和C ++编写多线程程序的过程
- OpenMP是一个多线程共享地址模型
  - 线程通过共享变量进行通信

2.OpenMP的主要特点是？（P11）

- OpenMP是一个多线程共享地址模型
  - 线程通过共享变量进行通信
- 意外的数据共享会导致竞争
  - 程序的结果因线程调度不同而改变
- 控制竞争问题
  - 使用同步来保护数据冲突
- 同步很昂贵
  - 更改访问数据的方式以减少同步（能独立并行就独立，尽可能减少通信）

3.熟悉OpenMP的关键指令

- 设置线程数：（P16）
  - omp_set_num_threads(5)：放在#pragma外面
  - #pragma omp parallel **num_threads(5)** private(nthreads,tid)
- 区分（P16、P20）
  - 获得线程id：int omp_get_thread_num()
  - 获得线程数：omp_get_num_threads()
  - 获取机器数：int omp_get_num_procs()（P20）
- **#pragma omp parallel for**
  - 声明以下的for循环可并行化
    - 条件：for循环次数在执行前是可数的；循环中没有break\return\exit；也没有goto语句
  - 例子（P23-24)：在嵌套循环中，如何选择位置来声明parallel for
- **#pragma omp parralel for private(j)**
  - 例子：P28-30
  - considerations：P31
- **#pragma omp parallell for firstprivate(a)**
  - 在循环开始的时候，继承共享变量的值作为初值
  - 可以声明成firstprivate的有：（P34）
    - 基础数据类型、数组、指针、类实例
- **#pragma omp parallel for lastprivate(x)**
  - 当负责串行最后一次循环的线程退出循环时，其私有变量的副本将被复制回共享变量
  - 例子：P37
- **#pragma omp for & #pragma omp parallel**
  - 例子：P41
- **#pragma omp single**
  - 说明**只允许一个线程**串行执行以下代码：比如有IO的时候
  - 不执行这一部分代码的线程需要等待，除非说明nowait
  - 例子：P43
- **#pragma omp sections / section**
  - 1个section由一个线程执行一次（P44)
  - 例子：P45
- **#pragma omp ... reduction(op:list) 归并操作（P46)**

4.熟悉OpenMP关键指令的执行过程

- 详见补充材料

补充：OpenMP的优点和缺点：（P48）

- **优点:**
  - 增量化并行和串行等效
  - 适用于数据/域分解
  - 在* nix和Windows上可用
- **缺点:**

- 不适用于按照任务分解
- 编译器无法检查诸如死锁之类的错误和竞争问题
- *甚至连语法错误都无法检测
  - 当并行指令有问题时，compiler直接忽略

Chapter 6 OpenMP中的竞争和同步

1.OpenMP中为了保证程序正确性而采用哪些机制？（P3）

- barriers 屏障，障碍
- mutual exclusion 互斥
- memory fence 内存屏障

2.什么是同步，同步的方式有哪些?

- **同步**：不管线程的调度方式如何，管理共享资源以使读取和写入以正确的顺序进行的过程
- **方式**：
  - **barriers 屏障，障碍**
  - **Mutual Exclusion （互斥）** e.g. pthread_mutex_lock
  - **critical section临界区（高级别的同步）**
  - **lock（低级别的同步）**

3.OpenMP Barrier 的执行原理。

- **barrier（P5）**
  - 线程组中的每个成员必须到达，任何成员才能继续的同步点
  - **#pragma omp barrier**：插入在并行结构的最后
    - for pragma , single pragma等都有隐式的barrier存在
    - 可以用nowait语句来禁用
- **例子**：如何使得主线程输出语句在最后显示：加barrier（P7）
- **nowait语句（P9）**
  - **#pragma omp for nowait**
  - 例子：P10

补充：互斥、临界区、锁的原理（P11）

- **互斥mutual exclusion**
  - 一次仅允许一个线程或进程访问共享资源
  - 使用锁来实现
- **临界区critical section（高级别的同步）**
  - 一次只有一个线程将执行临界区中的结构化块
  - #pragma omp critical（P32）
  - 好处：可以消除RC问题
  - 缺点：临界区是串行执行的，如果临界区过大，则并行加速比会大大下降。且需自己识别临界区
- **锁lock（低级别的同步）**

3.OpenMP Barrier的执行原理

4.OpenMP中竞争的例子

- 竞争：非确定性的行为，它是由两个或多个线程访问共享变量的时间引起的
- 例子1：计算面积& π（P12-15）
- 例子2：链表插入节点（P16-20)

5.OpenMP中避免数据竞争的方式有哪些？（P22）

- **作用域变量变成线程私有的**
    - 使用openmp private语句
    - 在线程函数内定义变量
    - 在线程栈上分配（作为参数传递）
- **用临界区控制共享访问**
    - 互斥（P23）和同步
- **注：flag（自己设置的锁）不能保证互斥**
    - （P25-30）由于check和assign flag是分开的，不是原子操作
    - 分析：P31

6.OpenMP Critical 与 Atomic的主要区别是？（P37)

- critical(P32)，atomic（P36）
    - 临界区的特殊情况，以确保原子更新到内存位置
- 两者都只对下面紧挨着的语句操作
- atomic限制写操作，不限制读操作；critical整条语句都只能单线程
- atomic的效率更高！（workone(i)和index[i]的读操作都可并行。只不过"+="变成了原子、单线程的）

chapter 7 OpenMP性能优化

1.什么是计算效率？（P5）

- 硬件利用率的测量指标
- **定义**：加速比/核心数，其中加速比=串行执行时间/并行执行时间
    - 注：加速比达不到理想情况的原因是：并行有额外开销（通信开销），甚至最后加速比会下降。

2.调整后的 Amdahl 定律如何理解?

- $$加速比 = \frac{串行时间 + 并行时间}{串行时间 + 并行时间/核心数 + 并行开销时间}$$（$P13$）
- amdahl定律本身太乐观：忽略了并行处理的开销，包括线程创建和销毁的时间（线程管理时间）；此开销随着核心数的增加而增加
- 且没考虑负载不均衡的问题：会带来等待时间，导致执行时间增加
- 当n趋向于无穷的时候，串行时间可忽略，加速比趋向于p（p为核心数）

- 问题规模n增大，则可并行部分的占比增大

3.OpenMP 中 Loop 调度的几种方式，执行过程

- loop schedule调度：loop 循环如何分配给线程
- **static schedule静态调度（P24-25)**
  - 迭代在执行loop之前分配给线程（P23)
  - **schedule (static [, chunk])**
    - 一次分配chunk个迭代给线程
    - 轮询分配
    - 低开销，会引起负载不均衡
  - 最适合迭代次数可预测和任务较均衡
  - 在无schedule时，默认为static方式
- **dynamic schedule动态调度（P26-27)**
  - 迭代在执行loop过程中分配给线程（P23）
  - **schedule (dynamic [, chunk])**
    - 线程抓住迭代"块"
    - 完成迭代后，线程请求下一个迭代集合
    - 更高的线程开销，可以减少负载不均衡
  - 最适合不可预测或高度可变的工作（快的多请求的）
- **guided schedule（P28-29)**
  - **schedule (guided [, chunk])**
    - 从大块开始的动态调度（特殊的dynanmic调度，看谁执行的快就调度给谁）
    - 块的大小缩小； 不小于"chunk"阈值（最少分配给线程的循环次数）
    - 最初的块的大小正比于"迭代次数/线程数"（分配给线程0的）
    - 后续的块的大小正比于"剩余迭代次数/线程次数（上取整）
  - 是动态调度的特殊情况，以在计算逐渐消耗更多时间时减少调度开销

4.OpenMP 中 Loop 转换的方式包括哪几种？熟练掌握。

- **loop fission 循环裂变（P31）**
  - 从具有循环承载依赖性的单循环开始
  - 将循环分为两个或多个循环
  - 新循环可以并行执行
  - 例子：（P32-33）将有数据依赖的循环拆成两个，分别并行
- **loop fusion 循环聚变（P34)**
  - 组合循环来增加粒度（也是为了加速）
  - 例子1：（P35-36）将独立语句拿出来，合并循环
    - 这样就只要一个barrier，而不用2个
  - 可和**复制copy**结合起来（P37）
    - copy变量，可变得更快（减少同步开销）
    - 例子2：（P38-40）将变量x的计算放在最开始（cause这是相对独立的）
  - 例子3：（P41-42）迭代次数是素数的时候，通常会负载不均衡
    - 此时，可以合并循环，较大的迭代次数为负载均衡和隐藏开销提供了更好的机会
- **loop exchange 循环交换（P43)**

- 嵌套的for循环可能具有数据依赖性，从而阻止并行化
- 交换for循环可以：
  - 使loops变得并行
  - 增加并行粒度（每个线程执行的工作量大小增加，减少线程数）
  - 提高线程局部性
- 例子：（P45-52）交换后，可以在第一层循环外声明并行，每个线程的计算力度增加（任务量增加），效果更好

chapter 8 MPI编程模型

1.什么是 MPI 编程模型?

- 消息传递模型，通过网络连接的节点的并行问题（P5）

2.消息传递性并行编程模型的主要原则是什么？ （P6-7)

- 支持消息传递范例的机器的逻辑视图由p个进程组成，每个进程都有自己的专有地址空间
- 限制：
  - 每个数据元素必须属于该空间的一个分区； 因此，必须对数据进行显式划分和放置
  - 所有交互（只读或读/写）都需要两个进程的合作-具有数据的进程和想要访问数据的进程
  - 这两个约束虽然繁重（繁重），但对于程序员而言，基本**成本**非常明显
- 消息传递程序通常使用异步或松散同步的范式编写
  - 在异步范例中，所有并发任务均异步执行（进程间完全独立：异步）
  - 在松散同步模型中，任务或任务子集进行同步以执行交互。 在这些交互之间，任务完全异步执行（在某些点，需同步：松散同步）
- 大多数消息传递程序都是使用**单程序多数据流（SPMD）**模型编写的

3.MPI 中的几种 Send 和 Receive 操作包括原理和应用场景。

- buffer：有没有data缓存？blocking：发送时，有无终止自己的计算任务
- **Non-buffered blocking 无缓存阻塞式（P10)**
  - 使send操作仅在安全的情况下才返回
  - 在非缓冲阻塞发送中，在接收过程中遇到匹配的接收之前，操作不会返回
  - 劣：空转和死锁是非缓冲阻塞发送的主要问题
    - 优：保证正确
  - 应用场景：P11
- **buffered blocking 缓存阻塞式（P12)**
  - 在缓冲阻塞式发送中，发送方只需将数据复制到指定的缓冲区中，并在复制操作完成后返回。
    - 不用等待找到接受进程的匹配
  - 数据也被复制到接收端的缓冲区。
  - 缓冲减轻了空转，但代价是复制开销
  - 应用场景：P14,P17-P18
    - 有界的缓冲区大小可能会对性能产生重大影响（P17）

        - 大小要匹配！
    - 由于接收操作阻塞，使用缓冲仍然可能发生死锁
        - 仍会产生死锁，但概率减小了！
- **Non-blocking 非阻塞式：（P20）**
    - 程序员必须确保发送和接收的语义。（复杂度增大）
    - 此类非阻塞协议在语义上是安全的之前从发送或接收操作中返回。
    - 非阻塞操作通常伴随着检查状态操作。（程序员确保安全）
    - 如果正确使用这些原语，它们可以通过有用的计算来重叠通信开销。（并发通信&计算）
    - 消息传递库通常提供阻塞和非阻塞原语。
    - 应用场景（P21）：硬件/软件支持

4.MPI 中的关键编程接口（P26）

- **MPI_Init**(&argc, &argv)：初始化MPI（一定要有！）(P27)
- **MPI_Finalize**()：终止化MPI（一定要有！）(P27)
- int **MPI_Comm_size**(MPI_Comm comm, int *size)：确定进程数(P31)
- int **MPI_Comm_rank**(MPI_Comm comm, int *rank)：确定标识符（从0开始）(P31)
- int **MPI_Send**(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)：发送信息(p35)
- int **MPI_Recv**(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)：接受信息(P35)
- 各种类型的send（阻塞P36 和非阻塞P37 都有）

5.什么是通信子？（P28-P30）

- 进程可以分组(groups)
    - 抽象成可以进行相互通信的进程的集合
- 每条消息都是在上下文(context)中发送的，必须在相同的上下文中接收
- 进程集合和上下文共同构成**通信子(communicator)**
- 一个进程通过其在一个通信子内部的**标号**来标识
    - 标号并非全局的！！从0开始的！
- **MPI_COMM_WORLD**：默认的通信子，也是最大的通信子，其进程集合包含**所有初始的进程**
- 通信子的函数：P29
- 通信子定义了通信域：允许彼此通信的一组过程(P30)
- 有关通信域的信息存储在MPI_Comm类型的变量中
- 一个进程可以**属于许多不同**的（可能是重叠的）**通信子**（通信子可交叉）

6.MPI 中解决死锁的方式有哪些?

- **例子1**：进程1两个send，进程2两个recv，两者的次序相反（P41,P44）
- **例子2**：进程1&2都是先send后recv，互相等（P42-43）

7.MPI 中的集群通信操作子有哪些？原理是什么？（P50）

- int **MPI_Barrier**(MPI_Comm comm)（阻塞到所有进程完成调用）（P50）
    - 在一个通信组中生效，相当于同步点
- 一对所有广播：int **MPI_Bcast**(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)(P50)

- 所有对一归并：int **MPI_Reduce**(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int target, MPI_Comm comm)（P50）
  - 归并操作MPI_Op：P51
- 所有对所有归并：int **MPI_Allreduce**(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) （P54）
  - all-to-all通信（实际是先归并到1个，然后broadcast）
- 计算前缀和：int **MPI_Scan**(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)（P54）
  - 可以看成是特殊的**MPI_Reduce**
- gather操作：int **MPI_Gather**(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int target, MPI_Comm comm)（P55）
  - 汇聚成一个行向量，放到某个进程中
- 所有进程都有gather之后的数据：int **MPI_Allgather**(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, MPI_Comm comm)（P55，P58）
- 把数据分散到其他进程中：int **MPI_Scatter**(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, MPI_Comm comm)（P55,P57）
  - **与MPI_Bcast()的区别**：（P55)
    - Bcast的所有进程得到的data是一样的
    - 而Scatter收到的data可能不同
- MPI_Gather和MPI_Scatter的区别：P56

chapter 9 MPI与OpenMP联合编程

1.如何利用 MPI 实现 Matrix-vector 乘积？不同实现的特点是什么？

- **按行划分(Rowwise Block Striped Matrix)（P5)**
  - 通过域分解进行分区
  - 原始任务与矩阵行和整个向量有关
  - 每一行进行内积（进程内部），然后再将每个进程的结果通过all-gather通信组合在一起（P6）
  - **allgatherv**：允许每个进程的数据量不同（P7-P10)
  - 归并&映射：（P10)
    - 任务数固定（行数固定），规则通信方式（all-gather)，每个任务计算时间相同（task划分均匀），策略：按行归并+每个进程创建一个任务
- **按列划分(Columnwise Block Striped Matrix)（P11)**
  - 通过域分解进行分区
  - 任务与矩阵列和向量的元素有关
  - 每个进程完成矩阵列和对应向量元素的相乘，然后在进行“+”的归并操作
    - 这个归并是作相互的数据分发，**开销很大**！“all-to-all exchange"
  - 过程：P12-16
  - 归并&映射：（P17)

        - 任务数固定（列数固定），规则通信方式（all-to-all），每个任务计算时间相同（task划分均匀），策略：按列归并+每个进程创建一个任务
- **按棋盘划分(checkerboard block decomposition)（P18）**
    - 原始任务与矩阵A的每个元素相关联；每个原始任务执行一次乘法；将原始任务聚合为矩形块；进程形成二维网格；向量b在网格的第一列中按块分布在进程之间（向量b也要拆分P19）
    - 具体流程：P20-21
        - 重新分配b -> 矩阵-向量点乘 -> 归并 -> 把b从欧诺个第1列弄到第1行：分类讨论：当前进程数量是否是平方数？（P22）
            - 如果是平方数：则第一列发送b的各部分，到第一行接收
            - 如果不是平方数：将b聚集到进程(0,0)，然后进程(0,0)广播到第一行
        - -> 第一行的进程在各列中分散
    - 随着进程数增加，期盼化的优点就体现出来了（P35）
        - 位数增多，邻居增多，通信时更方便

2.MPI 和 OpenMP 结合的优势是什么？（P39）

- 更快！！
- 降低通信开销
    - 消息通过mk进程传递 & p消息通过m个进程传递，每个进程有k个线程
- 程序的更多部分可以并行（程序并行度增加，加速比提高）
- 允许通信与计算产生更多重叠，重合度提高
- 具体例子：P40
- **纯MPI和MPIP+OpenMP的图像比较（P46）**
    - 分析：在CPU数量为2,4时，由于MPI+OpenMP模型的线程和进程共享带宽，而带宽一定，所以两者争夺带宽引发冲突，导致结果可能不如MPI快；然而，当CPU数量上升到6,8时，MPI+OpenMP模型的优势通信代价较低，优势体现出来。

3.如何利用 MPI+OpenMP 实现高斯消元?

-

chapter 10 GPGPU、CUDA和OpenCL编程模型

1.CUDA 的含义是什么？(P13)

- CUDA：计算统一设备架构
    - CUDA是一种新的操作GPU计算的硬件和软件架构，它将GPU视作一个数据并行计算设备。
- 为跨多种处理器的数据并行编程提供固有的**可扩展环境**（Nvidia仅制造GPU）
- 使通用编程人员可以访问SIMD硬件。 否则，将浪费大量可用执行硬件！

2.CUDA 的设计目标是什么？与传统的多线程设计有什么不同?

- **可扩展性（P14）**

    - Cuda表示许多独立的计算块，可以按任何顺序运行。（独立模块，独立执行）
    - Cuda编程模型固有的**可伸缩性**大部分源于“线程块”的批量执行；
    - 在同一代GPU之间，许多程序在具有**更多“核心”**的GPU上实现了线性加速
- **SIMD编程（P15）：做向量计算**
    - 硬件架构师喜欢SIMD，因为它允许非常节省空间和能源的实现
    - 但是，CPU上的标准SIMD指令不灵活，难以使用，编译器也难以定位
    - Cuda Thread抽象将提供可编程性，但需要额外的硬件
- **不同点：**
    - CPU遵循的是**冯诺依曼架构**，其核心就是：**存储程序，顺序执行。** 因为CPU的架构中需要大量的空间去放置存储单元和控制单元，相比之下计算单元只占据了很小的一部分，所以它在大规模并行计算能力上极受限制，而更擅长于逻辑控制。 对更大规模与更快处理速度需求效果不好。
    - GPU就是用很多简单的计算单元去完成大量的计算任务这种策略基于一个前提，就是线程间的工作没有什么依赖性，是互相独立的。

3.什么是 CUDA kernel？（P17）

- kernel函数：可由CPU装载到GPU，可由CPU访问的（主机级）， 在GPU上执行的核心程序，这个kernel函数是运行在某个Grid上的。
- 理解kernel，必须要对kernel的线程层次结构有一个清晰的认识。首先GPU上很多并行化的轻量级线程。kernel在device上执行时实际上是启动很多线程，一个kernel所启动的所有线程称为一个网格（grid），同一个网格上的线程共享相同的全局内存空间，grid是线程结构的第一层次，而网格又可以分为很多线程块（block），一个线程块里面包含很多线程，这是第二个层次。线程两层组织结构如上图所示，这是一个gird和block均为2-dim的线程组织。grid和block都是定义为dim3类型的变量，dim3可以看成是包含三个无符号整数（x，y，z）成员的结构体变量，在定义时，缺省值初始化为1。因此grid和block可以灵活地定义为1-dim，2-dim以及3-dim结构，kernel调用时也必须通过执行配置<<<grid, block>>>来指定kernel所使用的网格维度和线程块维度。举个例子，
- device函数：只能由GPU访问（设备级）

4.CUDA 的编程样例

- 向量加法：P19-20（使用global函数）

5.CUDA 的线程分层结构（P24）

- Cuda编程模型中的并行性表示为4级层次结构
- **流(Stream)/一串指令序列**（或是kernel函数）是一系列按顺序执行的Grid，Fermi GPU并行执行多个流
- **网格(grid)**是一组最多2^32的线程块，它们执行同一内核（包含多个SM）
- **线程块(thread block)**(=流多处理器SM)是一组最多1024个Cuda线程的集合
- **Warp**：由32个线程组成，它们在锁步SIMD中执行（SIMD中"M"的宽度）
- **CUDA线程(CUDA THREAD)**是一个独立的，轻量级的标量执行上下文
    - 最小的计算单位，且只能执行标量计算

补充：各部分介绍

- **CUDA 线程：（P25）**
    - 从逻辑上讲，每个CUDA线程：具有自己的控制流和PC，寄存器文件，调用堆栈；可随时访问任何GPU全局内存地址；可通过五个整数在网格内唯一地标识：threadIdx{x，y，z}(三维),blockIdx{x，y}（二维）
    - 非常细粒度：不要期望任何单个线程都能完成昂贵的计算工作

        - 全部占用时，每个线程有21个32位reg；存储很小；GPU没有操作数旁路网络：解码、装载有花销，需通过多线程或ILP来抵消等待时间
- **Warp：（P26）**
    - CUDA处理器的SIMD执行宽度
    - 一组32个同时执行的CUDA线程
        - 当warp中的所有线程从同一台PC执行指令时，执行硬件的使用效率最高；否则，如果线程发散（执行不同的PC），则某些执行管道未使用
        - 如果warp中线程访问的是对齐连续的DRAM块，则访问合并为单个高带宽访问（传输带宽增大）
    - 技术上说，warp size在未来会变化
- **Thread block线程块：（P27）**
    - 是虚拟化多线程核心
    - 执行中等粒度的数据并行任务（所有线程共享指令cache）
    - 块中的线程通过屏障内部机制进行**同步**，并通过快速的片上**共享存储**器进行**通信**（**同步发生在这一级别**，且所有线程都要同步！）
    - 以1维、2维或3维组织
- **Grid网格：（P28）**
    - 是执行相关计算的线程块的集合
    - 性能可移植性/可扩展性要求每个grid需要有许多blocks
    - 1个内核调用的线程块必须是并行的子任务（消费者生产者模型：冒险！）
    - 以1维、2维组织，共享全局内存
- **stream流：P29**
    - 是依次执行的一系列命令（内核调用，内存传输）
    - 为了同时执行多个内核调用或内存传输，应用程序必须指定多个流。
        - 常用于有多个用户（即程序）在同个GPU上跑的时候

6.CUDA 的内存分层结构

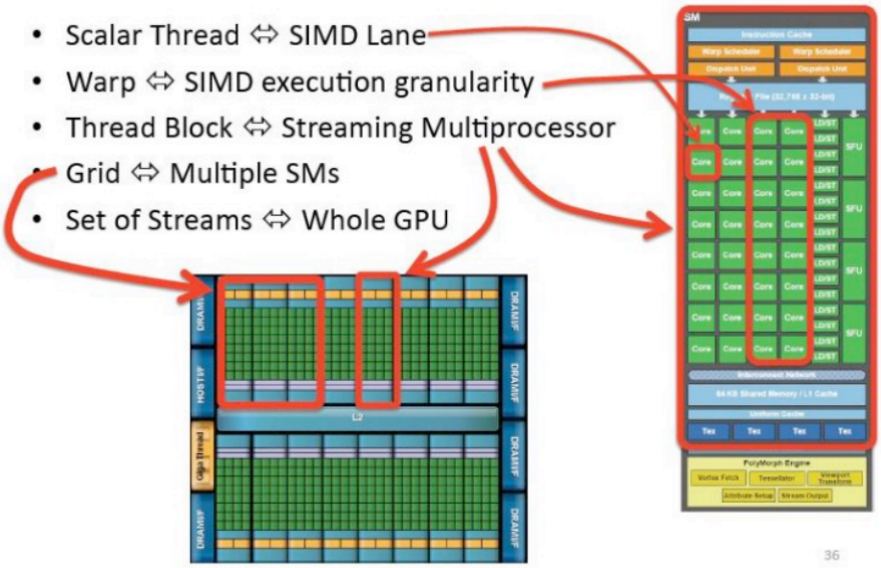
- 每个CUDA线程都可以私有访问可配置数量的寄存器(P31)
- 每个线程块都可以私有访问可配置数量的暂存器(P31)
- 所有网格中的线程块均共享对大型"全局"内存池的访问，而该池与主机CPU的内存分开（P32）
- 由于Cuda的图形遗产，内存层次结构中还有其他只读组件（P33）
- 64 KB Cuda常量内存与全局内存位于同一DRAM中，但可以通过每个SM高速缓存特殊的只读8 KB访问；Texture Memory也位于DRAM中，可以通过每个SM的小型只读缓存进行访问，但还包括插值硬件（P33）
- 此硬件对于图形性能至关重要，但仅对偶尔使用的通用工作负载有用（P33）
- 系统中的每个CUDA设备都有自己的全局内存，与主机CPU内存分开(P34)
- 主机<->设备内存的传输是通过PCI-E上的cudaMemcpy（）进行的，非常昂贵(P34)
- 通过多个CPU线程管理多个设备(P34)

7.CUDA 中的内存访问冲突（P38)

- 共享内存已存储：由32个可独立寻址的4字节宽的内存组成（32*4 bytes）
- 每个存储体每个周期可以满足一个4字节的访问
    - 当两个线程（在同一个warp中）尝试在给定的周期中访问同一存储库时，发生存储库**冲突**
    - GPU硬件将串行执行两次访问，使得warp的指令将花费**额外的周期**来执行
- 内存冲突是次级的性能影响：**对片上共享内存的串行访问也比对片外DRAM的访问要快**
- 3种无冲突的访问情况（P40）：

- 可以在32浮点的块中进行重新排列
- 多个线程读取相同的内存地址（不冲突，全读）
- 所有读取相同内存地址的线程是广播

补充：CUDA如何映射到GPU中？(P45)

- 

- GPU处理器层次结构的每个级别都与一个内存资源相关联（P52）
  - 标量线程/warp：寄存器文件的子集
  - 线程块/SM：共享内存（L1 cache）
  - 多个SMs/整个GPU：全局DRAM

8.OpenCL 运行时编译过程（P75）

- 1通过提供源代码或二进制文件并选择要定位的设备来创建程序对象
- 2编译程序：确定运行在什么设备，传递编译参数，检查编译错误
- 3建立程序，4创建内核
- 编译程序和创建内核的开销很大
  - 每个操作只需执行一次（在程序开始时），通过设置不同的参数可以多次重用内核对象
- **命令**：gcc -o vecadd vecadd.c –I/opt/AMDAPP/include – L/opt/AMDAPP/lib/x86_64 -lOpenCL（P82）

Chapter 11 MapReduce并行编程模型

1.为什么会产生 MapReduce 并行编程模型?

- 大规模数据处理
  - 全社会数据产生的速度非常快；大数据的呈现出指数增长速度；
  - 想要处理大量数据（> 1TB）；想要在成百上千个CPU中并行化，并简化过程
  - 而MapReduce可以解决大数据带来的问题。它是一个简单而强大的界面，可实现大规模计算的自动并行化和分配，并结合该界面的实现，可在大型商用PC集群上实现高性能

2.MapReduce 与其他并行编程模型如 MPI 等的主要区别是什么?

- 两者都是消息通信模型，都是**SPMD**模型（单程序多数据流）
- **MPI**：

    - 计算编程：没对计算做抽象，暴露通信接口
    - 环境：运行在高性能计算环境，需要强大的CPU计算资源、内存、网络传输能力
    - 性能：底下硬件非常可靠
    - 上层应用：支持高密度计算型应用，如天机预报（解方程）
- **MapReduce：**
    - 计算编程：抽象计算接口，抽象（封装）通信接口；计算只剩下两种简陋形式：map & reduce
    - 环境：运行在商业PC上，简单环境
    - 性能：简单硬件环境（底下硬件不大可靠，容易损坏），程序可靠性/容错性高
    - 上层应用：支持数据密集型应用（数据量大，计算可能简单）

3.MapReduce的主要流程是什么？（P11-13)

- 使用特殊的map() 和 reduce()函数处理数据（P11）
    - 在输入中的每个项目上调用map()函数，并发出一系列中间键/值对
    - 与给定键关联的所有值都组合在一起
    - 在每个唯一键及其值列表上调用reduce()函数，并发出添加到输出中的值
- **map：（P12)**
    - 数据源中的记录（文件中的行，数据库的行等）作为键*值对输入到map函数中。例如：（文件名，行）
    - map() 产生一个或多个中间值以及来自输入的输出键
- **reduce：P13**
    - 映射阶段结束后，给定输出键的所有中间值将合并到一个列表中
    - reduce()将这些中间值合并为同一输出键的一个或多个最终值

补充：MapReduce执行？（P39-P51）

4.MapReduce的简单实现。如Hello World 例子。

- P16 统计词典单词个数（具体：P33-36）

5.MapReduce具有哪些容错措施？（P22)

- 调度器发现进程出错：
    - 重新执行已完成和正在进行的map() 任务
    - 重新执行进行中的reduce() 任务
- 主机注意到特定的输入键/值导致map() 崩溃，并在重新执行时跳过这些值
    - 效果：可以解决第三方库中的错误！（允许丢数据，cause数据本身就是冗余的）
- **主进程master**定期ping**工作进程worker**（P51）
    - map-task错误：重新执行：所有输出都在本地储存
    - reduce-task错误：只重新执行部分任务：所有输出存在全局文件系统

6.MapReduce存在哪些优化点？（P23）

- 映射完成之前无法开始归并
    - 单个慢速磁盘控制器可以限制整个过程的速率
- 用额外资源去冗余执行慢速任务；谁先执行完就取它的作为结果
- “合并”的函数(map之后的步骤）可以与映射器在同一台机器上运行
- 在真正的还原阶段之前发生一个小型还原阶段，以节省带宽

7.MapReduce可以解决的问题有哪些?

- 图：P29
    - 主节点运行JobTracker（ResourceManager）实例，该实例接受客户的工作要求；TaskTracker（NodeManager）实例在从属节点上运行；TaskTracker为任务实例分叉单独的Java流程
- 例子1：统计单词频率：P33-36

Chapter 12 基于Spark的分布式计算

1.Spark与 Hadoop的区别和联系

- 联系：都是采用mapreduce
- 区别：spark把基于磁盘的计算转成了基于内存的计算
    - 但不是所有的数据都在内存，主要在内存，还有比较少的在磁盘
    - 导致了计算速度非常快，变成了实时计算；hadoop是批处理大规模数据，离线计算

2.传统MapReduce的主要缺点是什么？（P12）

- mapreduce在单遍计算方面表现出色，但对于多遍算法（迭代计算）效率低下
    - cause迭代间的data sharing已经写到磁盘中
- 缺少数据共享的高效原语
    - 步骤之间的状态进入分布式文件系统（即磁盘）
    - 低效原因在于复制和磁盘存储

3.Spark中的RDD如何理解?

- **RDD**：resilient distributed datasets弹性分布式数据集
    - 对象的不可变集合（不可写&只可读-> 容错，恢复），分布在整个集群中
    - 静态类型：RDD[T]有T类型的对象
- 整个集群中的具有用户控制的划分和存储功能（内存，磁盘）的对象集合
- 通过并行转换来构建（map, filter，...）
- 可以再创建失效时自动恢复，**容错能力高**（**可靠性**！！）
    - 可以跟踪线性信息来重建丢失数据

4.Spark样例程序

- **逻辑回归LR**：P25-28
    - P26 hadoop&spark速度比较
    - P27 内存成比例增加，速度也成比例增加

Chapter 13 离散搜索与负载均衡

1.深度优先搜索的主要流程（P6）

- 使用DFS考虑组合搜索问题的可行的解决方案
- 递归算法
- 当一个节点没有孩子或者其所有的孩子都被遍历完时，回溯

- 注：这不是最简单的算法，写起来简单而已

2.深度优先搜索的复杂度（P18）

- 假设状态空间树中的平均分枝数（每个节点的平均孩子数）是b
- 时间复杂度：搜索一个高为k的树需要最多检查$\theta(b^k)$个节点（在最差情况下：指数时间）。
- 空间复杂度：$\Theta(k)$。

3.并行深度优先搜索的主要设计思想

- **第一种策略：给每一个进程分配一个分支**
  - 假设p=b^k，（p为进程数，b为平均分枝数）（P19）
    - 一个进程搜索所有的节点直到深度为k的层
    - 然后它只探索k层的1棵子树
    - 如果d（d为搜索的深度）> 2k，则每个进程探索之前的k层的状态空间树（时间）可忽略不计
  - 假设 p $\neq b^k$，（p为进程数，b为平均分枝数）（P21）
    - 进程可以执行串行搜索到状态空间树的m层
    - 每个进程都探索以m层节点为根的子树中的份额（？）
      - 如果m增大则有更多的子树可以在进程间划分（分的多，分的更细），使得负载更均衡
      - 提高m，也会提高冗余计算
    - 加速比和m的图像关系（P22）：后面加速比降低（由于前面都是串行，冗余计算提高）
  - 缺点：（P23）
    - 在大多数情况，状态空间树是不均衡的
    - 替代方案：使串行搜索更深入，以便每个进程处理许多子树（循环分配），这样会相对均衡一点
- **动机：P25**
- 动态负载均衡
  - 产生条件：P26
  - 整个空间分配给一个处理器去开始
  - 当一个处理器做完了工作，他会向其他处理器请求任务执行
    - 消息传递机器：工作请求和答复
    - 共享地址空间机器：锁和提取工作
  - 没有探索的状态可以方便的储存在处理器的本地栈
  - 当一个处理器到达最后的状态，则所有处理器终止

4.动态负载均衡的三种方式，以及每种方式的额外开销复杂度

- **异步轮询（ ARR: Asynchronous round robin）**
  - 每个进程都维护一个计数器（指针），并以循环方式发出请求
  - W= O($p^2log(p)$)（P42)
  - 异步轮询性能差原因：有很多的工作请求（P45）
- **全局轮询（GRR: Global round robin）**
  - 系统维护一个全局计数器（指针），并以循环方式发出请求
  - W =O(p log(p)) 或 O($p^2log(p)$)，(P43)

    - 性能差的原因：共享counter的竞争（尽管他的请求数是最小的）（P45）
- **随机轮询（RP：random polling)**
    - 请求随机选择的进程来工作（**效能最好**！！）
    - W=O($plog^2(p)$)．(P44)

5.最优搜索的处理过程

- 以华容道为例：（P53- 60)

6.并行最优搜索的主要思想和实现方式

- 适用范围：适合在多计算机或分布式多处理器上实现（P61）
- 矛盾目标：（P61）
    - 最大化本地内存和非本地内存引用的比值（减少开销）
    - 希望确保处理器搜索状态空间树的有价值部分（尽快找到最优解）
- 单优先队列：不好（P62）
    - 通信开销太大（访问冲突，竞争）；访问队列是性能瓶颈；不允许问题大小随处理器数量扩展
- 多优先队列：好（P63）
    - 每个进程维护未经检查的子问题的单独优先队列
    - 每个进程检索下界最小的子问题以继续搜索
    - 偶尔，进程发送未经检查的子问题给其他进程
- 具体实现：（P64）
    - 核心数据结构是由优先队列实现的开放列表；每个处理器锁定此队列，提取最佳节点，然后将其解锁；生成节点的后继节点，估计其启发式函数，并在适当锁定后根据需要将节点插入到开放列表中；当我们发现解决方案的成本比公开清单中的最佳启发式值高时，终止；由于我们一次扩展不止一个节点，会扩展那些不会在串行算法中扩展的节点。
- **集中策略Centralized strategy（共享内存式）**：用锁来序列化各种处理器的队列访问（P65）
- 避免多队列间的冲突；处理器可并行访问队列；（P67）
- **MPI式**
    - 可能会导致队列间的不均衡（有些队列被访问多，则任务少；相反，多），解决办法：（P67）
    - 均衡策略：随机，轮询（ring communication strategy P68），黑板通信（blackboard comm. strategy P69）

7.什么是加速比异常？主要分为哪几类？（P70）

- 由于处理器探索的搜索空间是在运行时动态确定的，因此实际工作可能会有很大不同
- 分为两类：
    - **加速异常**：通过使用p个处理器产生大于p的加速比的执行
        - 由于分配不均，所以可能很快就能找到最优解，然后通知别的p终止，导致加速异常
    - **减速异常**：通过使用p个处理器产生小于p的加速比的执行（实际是正常的）
- 加速异常也存在于BFS中，如果启发式函数很好，那么并行BFS所完成的工作通常要比其串行对应项多。

Chapter 14 性能优化：任务分发&调度

1.负载均衡主要有哪些方式？分别有什么特点？

- **静态调度 static assignment (P7)**
  - 线程的工作分配是预先确定的
    - 不一定在编译时确定（分配算法可能取决于运行时参数，例如输入数据大小，线程数等）
  - 例子：调用求解器示例：为每个线程（工作者）分配相等数量的网格单元（工作）
  - **优势**：简单，运行时开销基本为零；**劣势**：不平衡概率增大
  - **适用范围：**当工作的成本（执行时间）和工作量是可预测时（这样程序员就可以提前制定好任务）；当工作是可预测的，但并非所有工作的成本都相同（平均成本相同，整体平衡）（P8-9）
- **半静态调度 semi-static assignment（P10)**
  - 短时间可预测工作成本
  - 应用程序会定期自我描述并重新调整分配，重新调整之间的时间间隔分配为“静态”
- **动态调度 dynamic assignment（P11）**
  - 程序会在运行时动态确定分配，以确保负载分配合理。
    - **适用范围：**任务的执行时间或任务总数是不可预测的。
  - 例子：验证素数（P11）（事先无法预测x是否为素数）；图示P12
    - 使用共享变量“锁”，同步开销非常大（P13）
  - **优点**：可自适应；**缺点**：非常复杂

2.静态、动态负载均衡适用的场景是什么？

- 详见上一问

3.如何选择任务的粒度？（P15，P64）

- 设置比处理器数量更多的任务是有用的（许多小任务可以通过动态分配实现良好的工作负载平衡）——激励小粒度任务
  - 如果粒度太粗（任务数太少），则不会足够并行，更倾向于负载不均衡
- 但是要尽可能少的任务以最小化管理任务的开销——激发大粒度任务
  - 如果粒度太细（任务数太多），则调度开销非常大
- 理想的粒度取决于许多因素（本课程的共同主题：必须了解您的工作量和您的计算机）（编程者自己的观察&经验）
- 例子：P13-14：粗粒度和细粒度的选择

4.Clik_spawn的原理是什么？（P26）

- **cilk_spawn**：如foo(arg)：创建新的逻辑控制线程
  - 调用foo，但与标准函数调用不同，调用者可以继续与foo执行异步执行
    - 原线程继续执行，新线程执行foo
  - **注意**：cilk_spawn的抽象并没有指明spawn调用是如何和何时被调度执行的（P28）
    - 只是它们可以与调用方caller同时运行，也可以和其他被spawned调用的同时运行（即如果有两个调用，则两者无谁先谁后，谁先被调度就先执行）

5.Clik_sync 的原理是什么？（P26）

- **click_sync**：当 当前函数产生的所有调用完成时返回

- **注意1：**每个包含cilk_spawn的函数的末尾都有一个隐含的cilk_sync（含义：当Cilk函数返回时，与该函数相关的所有工作都已完成）
- **注意2：**cilk_sync的确对调度有限制，一定要所有调度完成之后，cilk_sync才可以返回（P28）

6.Clik_spawn 的调度方式有哪些？各自有什么特点?

- **child first孩子优先：（P39,-41）**
  - 记录后续执行体以供以后执行（P39）
    - continuation可以被其他线程窃取——**continuation stealing**
  - 调用者线程仅创建一个要窃取的项目（代表所有剩余迭代的后续）（P41）
  - 执行顺序和移除spawn时一样
  - DFS深度优先搜索遍历调度图
  - 例子：迭代循环的child first（P41-42)
  - 可以证明具有T线程的系统的工作队列存储空间不超过单线程执行的堆栈存储空间的T倍（P42）
  - 例子：P43 **快排**
- **continuation first后续执行体优先：（P39-40）**
  - 记录孩子以供以后执行（P39）
    - child可以被其他线程窃取（先创建出来）——**child stealing**
  - BFS广度优先搜索遍历调度图，复杂度为O(N)（P40）
  - 如果没有窃取，执行顺序和有cilk_spawn时的顺序非常不同（P40）
  - 例子：迭代循环的child first（P40)

7.Clik_spawn 中任务在不同线程之间 steal 的过程。（P44-46）

- 用**双端队列deque**来实现（P44）
  - 这种两端都可读写的双端队列是不需要锁的，避免了同步
  - 本地线程从**底部**推入/弹出工作；远程线程从**顶部**窃取
- 空闲的线程**随机**选取一个线程来窃取工作（P46）
- 从顶部窃取工作：（P46）
  - **减少**了与本地线程的**竞争**：本地线程和窃取线程访问的地方不同
  - 窃取是在**调用树的开头**进行的：这是一项"较大"的工作，因此，进行窃取的成本将在较长的未来计算中分摊
  - -最大化本地性：（结合运行孩子优先策略）本地线程在调用树的本地部分上工作
- **注**：还可以steal多一点，避免多次steal带来的同步开销

8.Clik_sync 的几种实现方式。

- **no stealing （P49）**
  - 没有发生窃取，同步无意义（和串行执行的控制流相同）
- **stalling join 拖延join? （P50-56）**
  - 初始化fork的线程要完成sync操作（也就是spawn线程的线程，通常是thread 0）
- **greedy policy 贪心算法（P57-63）**
  - 当创建fork的线程空闲，它会尝试窃取新的工作
  - 最后一个到达同步点的线程继续往下执行（不用通知thread 0）

- cilk使用这种方法调度
  - 当是空闲的时候，所有线程都会尝试窃取（线程只会在没有工作可以窃取的时候变得空闲）
  - 创建spawn的工作线程**可以不是**执行cilk_syn后面逻辑的线程