并行大家一起复习鸭~ 用于开卷

可以参考往届学姐的笔记呀，但是我们自己编写的过程中可以添加多一些自己的理解，也当是一个复习的过程啦

并行复习笔记.pdf
1.84MB

第一讲：并行计算概览

1. 什么是并行计算?

(1)并行计算是什么

早期就是一个核上的并发。

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)可并行的计算问题应该满足

- 可分解成同时计算的几个离散片段；
- 在任意时刻可同时执行多条指令；
- 多计算资源所花的时间少于单计算资源；

2. 并行计算有哪些优势?

(1)节省时间和花费

- 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
- 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

- 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；

- 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

- 现代计算机甚至笔记本都具有多个处理器或者核心；
- 并行软件就是为了针对并行硬件架构出现的；
- 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?

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是什么?

(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，需求与性能相互促进(科学计算：生物、化学、物理；通用计算：视频、图像、CAD、数据库)
- 大量设备、用户、内容涌现
- 大数据的发展
- 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；

(2)架构发展趋势

- CPU架构技术经历了四代即：电子管（Tube）、晶体管（Transistor）、集成电路（IC）和大规模集成电路（VLSI），这里只关注**VLSI**

  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

  如何提高CPU的处理速度?

  利用额外的额外的晶体管在芯片上构建更多／更简单的处理器

  为什么不靠增加CPU芯片的面积来增加主频?

  增大核心面积可以简单粗暴的增加CPU的核心数量并加入更多的功能和指令集，然而这也提升了产品成本，因为晶圆的尺寸是固定的，切割出来的芯片越小成本就越低，而且在单位面积出错率固定的情况下，面积越大的芯片蚀刻时上面有坏点变成不良片的可能性就越大，实际上大型芯片的不良率远比小芯片高。

- Moore定律
  - “芯片上的集成晶体管数量每18个月增加一倍”；
  - 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（2000年之后的解决方式：时钟频率很难增加；SS触到天花板出现“收益下降”；利用额外的晶体管在芯片上构建更多／更简单的处理器）
- 未来属于异构、多核的SOC架构，SOC = System On Chip
- Moore’s law新解（每两年芯片上的核心数目会翻倍；时钟频率不再增加，甚至是降低；需要处理具有很多并发线程的系统；需要处理芯片内并行和芯片之间的并行;需要处理异构和各种规范（不是所有的核都相同））

5. 并行计算的粒度?

- 函数级并行
- 线程级并行
- 进程级并行

6. 并行计算的难点?

- **找到尽可能多的并行点**（Amdahl’s Law）
- **粒度**：函数级并行、线程级并行还是进程级并行
- **局部性**：并行化后是否能够利用局部数据等
- **负载均衡**：不同线程、不同core之间的负载分布
- **协作与同步**
- **性能模型**：所有这些难点让并行编程要比串行编程复杂得多

7. Amdahl’s law?

加速比用于度量并行程序的加速效果。

理想的加速比：$speedup = \frac{T_{串行}}{T_{并行}}$

实际的加速比：$speedup = \frac{1}{1-p+p/n}$，（$p$为并行部分的比例，$n$为线程数）

补充：阿姆达尔定律提供了一个并行程序可以得到的加速比的上界：如果并行部分为1-p，那么无论有多少线程/进程，加速比无法超越1/(1-p)。

8. 要点补充

（1）简单理解一下**数据并行**与**任务并行**。

- 数据并行：将可以解决问题的数据进行分割，将分割好的数据放在一个或者多个核上进行执行；每一个核对这些数据都进行类似的操作。
- 任务并行：将许多可以解决问题的任务分割，然后分布在一个或者多个核上进行程序的执行。
- 例：

  有一个professor P，他在给本科生上一门课，一共有300个本科生，考试的试卷上有15到题目。有三个助教来帮助Professor P修改试卷。

数据并行：每个人平均看100份试卷，然后分别对自己的试卷的15道题目打分。

任务并行：每个人分别负责5个题目。

（2）顺便把效率写一下吧！

$$E = \frac{T_{串行}}{p * T_{并行}}$$

当问题规模不变，随着核数增加，效率等比例增加的，称为**强可扩展性**

当问题规模变大，随着核数增加，效率等比例增加的，称为**弱可扩展性**

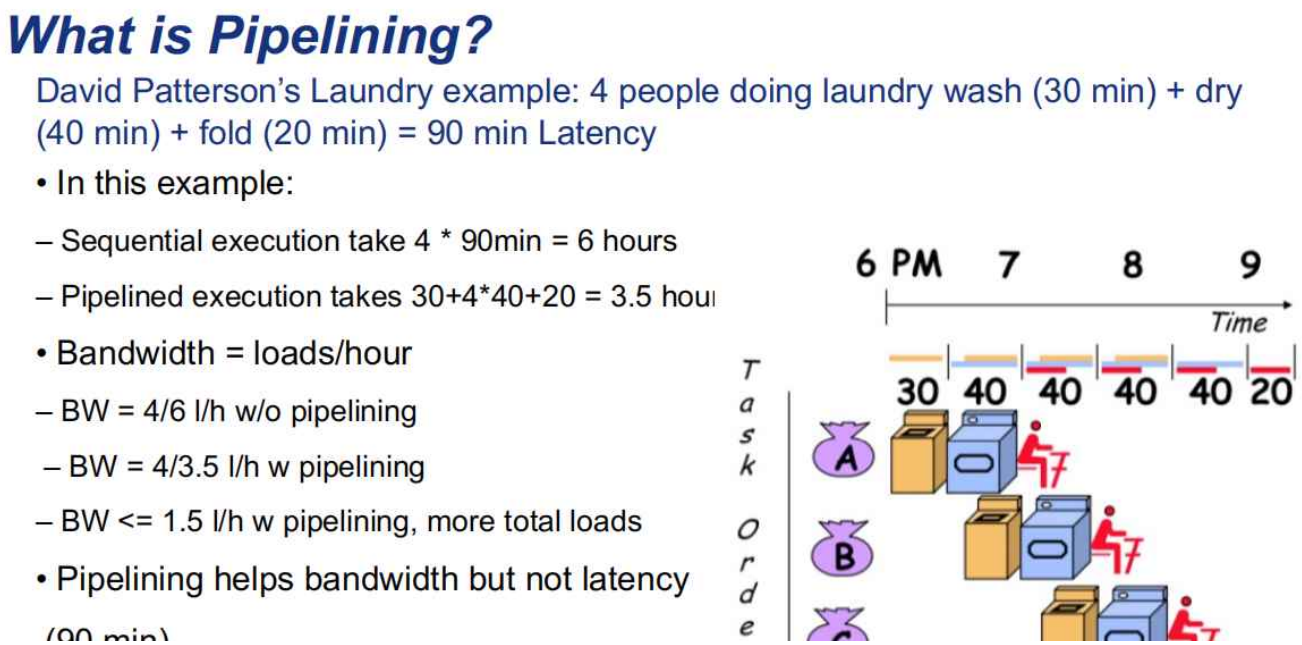
第二讲：并行架构

1. Flynn’s并行架构分类?

- SISD：单指令单数据流，一般的单处理器计算机,一条指令处理一个数据，所有的冯诺依曼体系结构的“单处理器计算机”都是这类，基本上07年前的PC都是这类。
- MISD：多指令单数据流，没有系统是按照这个结构设计的
- SIMD：单指令多数据流，MMX（多媒体扩展），SSE（流式SIMD），AVX（高级向量扩展），GPU（SIMT），指的是一个指令广播到多个处理器上，但每个处理器都有自己的数据。这种系统主要应用在“专有应用”系统上，例如数字信号处理、向量运算处理，其并行功能一般都是由编译器完成的。
- MIMD：多指令多数据流，每个处理单元有独立指令和数据。共享内存（Pthread，OpenMP），分布式内存（MPI，cluster）

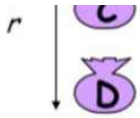
2. 什么是pipeline?

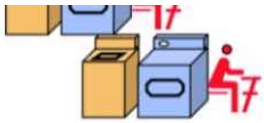
流水线是指在程序执行时，多条指令重叠进行操作的一种准并行处理的实现技术。各种部件同时处理是针对不同指令而言的，它们同时为多条指令的不同部分进行工作，以提高各部件的利用率和指令的平均执行速度。



(90 min)

- Bandwidth limited by slowest pipeline stage
- Potential speedup = Number of pipe stages

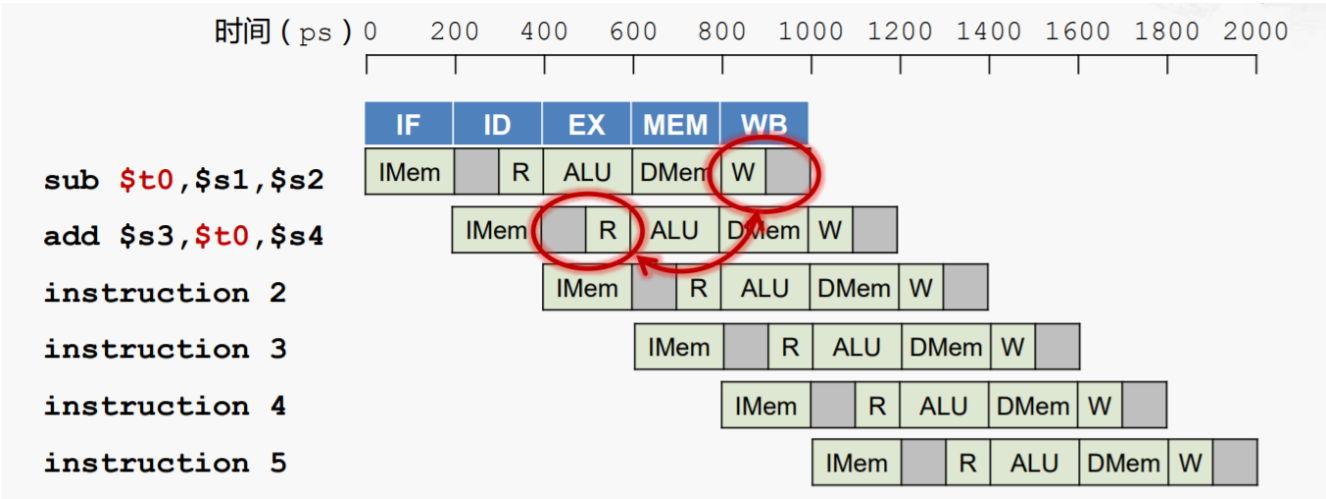




- 指令执行5个阶段：取指令->译码->执行->访存->写回
- 流水线的缺点：
- **冒险（Hazard）**：在流水线中我们希望当前每个时钟周期都有一条指令进入流水线可以执行。但**在某些情况下，下一条指令无法按照预期开始执行，这种情况就被称为冒险**。
- 结构冒险：如果一条指令需要的硬件部件还在为之前的指令工作，而无法为这条指令提供服务，那就导致了结构冒险。（这里结构是指硬件当中的某个部件）

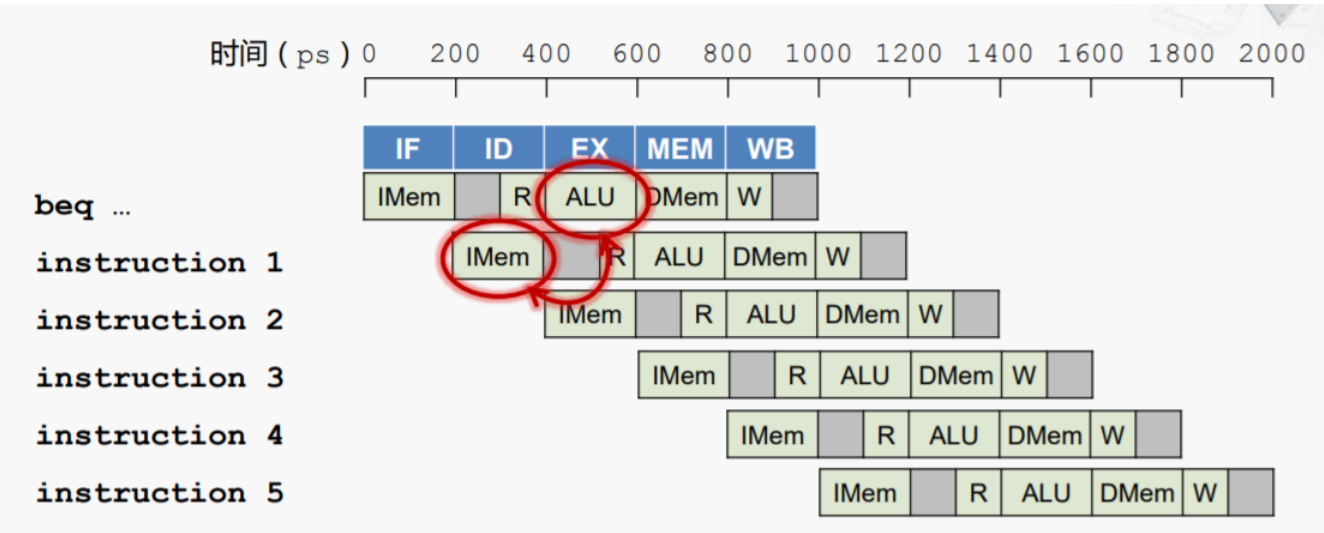
  比如如果两条指令读寄存器和写寄存器同时发生，如何处理？或者是同时读写存储器

  可以通过流水线停顿，产生bubble空泡的方式避免。但结构冒险在设计处理器时就考虑并解决好了，我们在使用时就不必考虑。
- 数据冒险：一条指令需要使用之前指令的结果，但是结果还没有写回。



解决方法：1.空泡 2.数据前递：t0在EX阶段就被计算出，所以可将它送到下一条指令ALU的输入，而不需要添加气泡。它还有个名称叫作旁路

- 控制冒险：尚未确定是否发生分支，如何进行下一次取指



解决：1.停顿bubble

3. 有哪些形式的指令级并行?

- 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，所谓的超标量CPU，就是只集成了多个ALU、多个FPU、多个译码器和多条流水线的CPU，以并行处理的方式来提高性能。
- 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，面向向量型并行计算，以流水线结构为主的并行处理计算机。 采用先行控制和重叠操作技术、运算流水线、交叉访问的并行存储器等并行处理结构，对提高运算速度有重要作用。
- 乱序执行OOO(Out-of-Order)：允许长操作的发生（即可提前结束，不用等）允许后面的指令继续进行（P24）
- 现代ILP：动态规划的乱序执行

4. 什么是Pthreads?

- 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）

5. 内存局部性原则有哪些?

- 时间局部性temporal locality：程序最近访问的地址在不久的将来很可能再次被访问（例如循环，重用）

- 空间局部性spatial locality：程序最近访问的地址其附近的地址将来很可能再次被访问（例如，顺序排列的代码，数组访问）

6. 内存分层?

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

- 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。

8. 新型存储系统的构成?

- 磁盘&内存相结合
- 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。

9. 什么是并行架构?

- 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

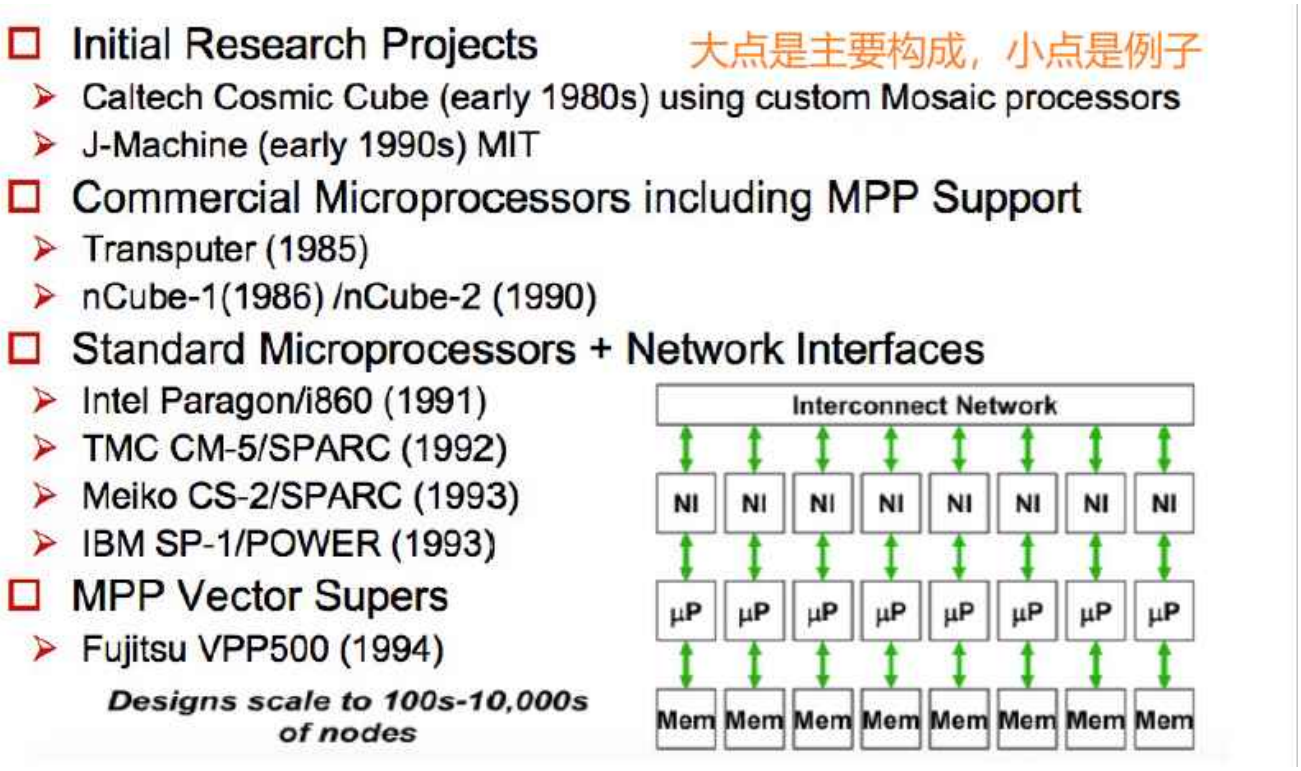
消息传递message passing(P90)：通过消息传递来通信（跨节点）

- 应用程序在节点之间发送显式的消息，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？

Massively Parallel Processors大规模并行处理机



第三讲：并行编程模型

1. 什么是并行编程模型？

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。

现在，我们将编程模型与底层并行机体系结构分离。过去主要使用指令集架构，现在主要是通信架构

并行编程模型位于并行应用之下，在操作系统，编译器和库之上。

2. 并行编程模型的主要包括哪些类型？主要特点是什么？

- **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?
- 变量命名Naming：哪些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?

4. 共享内存模型有哪些实现?

补充：首先说一下什么是共享内存模型?

（1）每个处理器都可以直接访问任何内存地址，在load和store的过程中，通信隐式地发生了。

（2） 单处理器上与分时系统类似，多道程序负载上具有较好的吞吐量

（3）通常被称为共享内存机器或模型：Ambiguous: memory may be physically distributed among processors

线程通过读/写共享变量进行通信。共享变量就像一个大公告板（同步原语也是共享变量）

- 在这个编程模型中，任务共享一个公共地址空间，它们异步读写
- 可以使用诸如锁/信号量之类的各种机制来控制对共享内存的访问

共享内存模型有哪些实现?

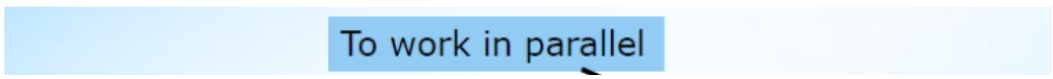
- 本机编译器和/或硬件映射用户程序变量到实际的内存地址，这是全局的
  - 在独立的SMP（对称多处理器）机器上，这是非常直接的
- 在分布式共享内存机器（例如SGI Origin）上，内存物理分布在机器网络中，但是通过专用硬件和软件实现了全局性。

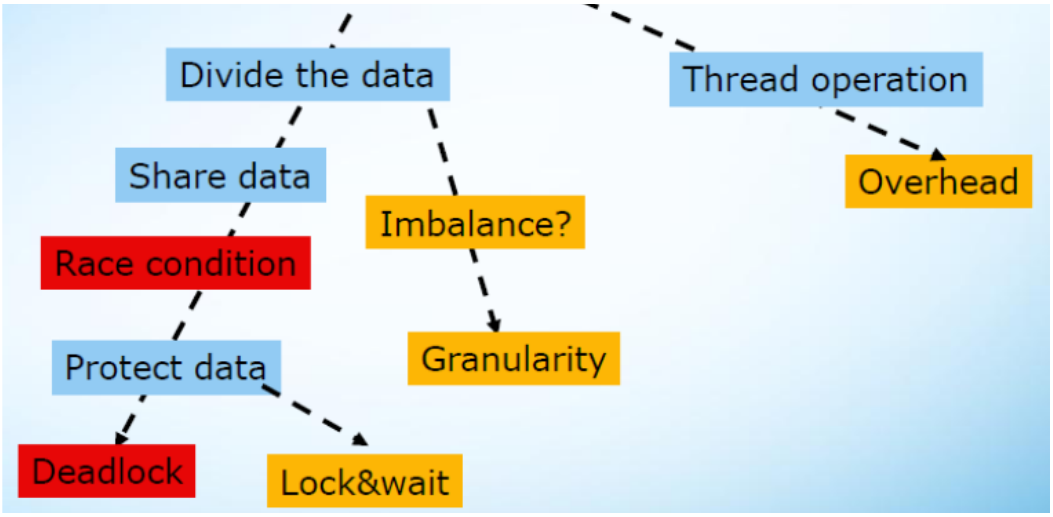
补充：共享内存模型的优缺点：

- 优点：从程序员的角度来看，该模型的一个优点是缺少数据“所有权”的概念，因此不需要明确指定任务之间的数据通信
- 缺点：性能方面的一个重要缺点是，理解和管理数据局部性变得更加困难
  - 将数据保持在其上工作的处理器的本地，可以节省多个处理器使用相同数据时发生的内存访问、缓存刷新和总线通信
  - 不幸的是，控制数据位置很难理解，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?

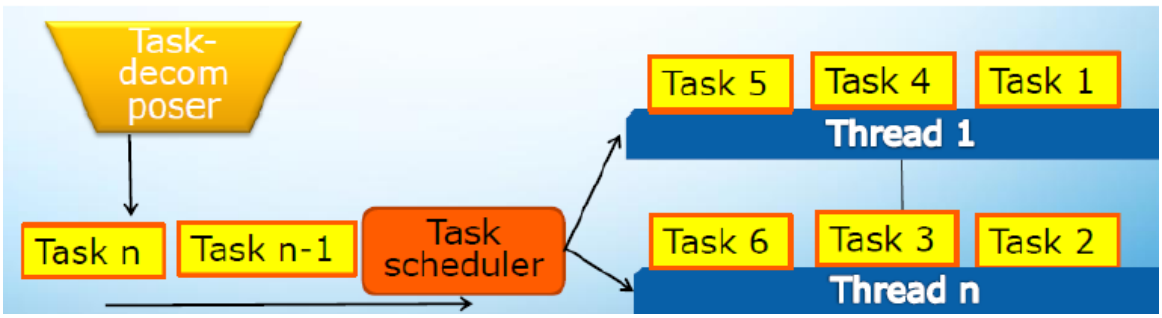
答：1.存在必须要串行的部分，这部分不能并行加速，达不到理想加速比；2.性能的提高无法抵消掉管理开销（线程的创建、管理与回收），特别是在问题规模较小的时候；3.竞态问题，为避免死锁，可能会带来锁的开销和等待问题；4.负载不均衡，数据划分达不到理想的并行加速。





6. 任务（Task）和线程（Thread）之间的关系？

- 任务由**数据**及其**处理过程**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开销小很多
- 通过偷取来轻松平衡线程之间的工作负载Ease to balance workload among threads by stealing？？？
- 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？

定义：由于两个或者多个进程竞争使用不能被同时访问的资源，使得这些进程有可能因为时间上推进的先后原因而出现问题，称为竞争条件Race condition

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）

竞争，又分为**确定性竞争**和**数据竞争**：

- 确定性竞争：有锁无锁都有可能

> 当两条并行链访问同一内存位置，且至少一条链执行写入操作时，就会发生确定性竞争。程序结果取决于哪个串“赢得竞争”并首先访问内存。

- 数据竞争：确定性竞争的特例，没有锁

> 数据竞争是确定性竞争的**特例**。数据竞争是当两个并行链（没有公共锁）访问同一内存位置且至少一个链执行写入操作时发生的竞争条件。程序结果取决于哪个串“赢得竞争”并首先访问内存。
>
> **如果并行访问受锁保护，则不存在数据竞争**。但是，使用锁的程序可能不会产生确定性结果。锁可以通过保护数据结构在更新期间在中间状态下不可见来确保一致性，但不能保证确定性结果。

解决办法如下：

- 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

8. 要点补充

补充：什么是死锁？

2个或更多线程等待彼此释放资源，线程等待一个永远不会发生的事件，比如挂起的锁。最常见的原因是锁层次的结构。

解决方法：始终以相同的顺序锁定和取消锁定；使用原子性操作

补充：什么是线程安全？

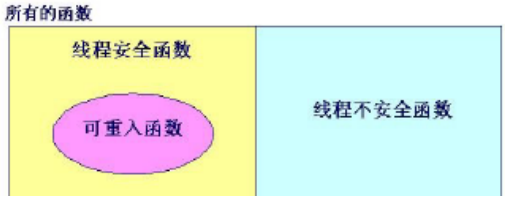
多个线程同时执行的时候能够正常工作。

可能出现非线程安全的时候：访问全局/静态变量或堆；分配/重新分配/释放具有全局作用域的资源（文件）；通过句柄和指针进行间接访问

解决方法：任何更改的变量必须是每个线程的本地变量；例程可以使用互斥来避免与其他线程的冲突

补充：什么是可重入函数？

- 可重入：多个执行流反复执行一个代码，其结果不会发生改变，通常访问的都是各自的私有栈资源；
- 可重入函数：当一个执行流因为异常或者被内核切换而中断正在执行的函数而转为另外一个执行流时，当后者的执行流对同一个函数的操作并不影响前一个执行流恢复后执行函数产生的结果；
- 可重入函数满足的条件：
  - 不使用全局变量或静态变量；
  - 不使用malloc或者new开辟出的空间；
  - 不调用不可重入函数；
  - 不返回静态或全局数据，所有数据都由函数的



调用者提供；

- 使用本地数据，或者通过制作全局数据的本地拷贝来保护全局数据；
- 不调用标准I/O；

53

Not thread-safe, not re-entrant

```
int tmp;
int add10(int a) {
  tmp = a;
  return tmp + 10; // <--- interrupt here
}
```

Thread-safe but not re-entrant

```
thread_local int tmp;
int add10(int a) {
  tmp = a;
  return tmp + 10;
}
```

Not thread-safe, but re-entrant

```
int tmp;
int add10(int a) {
  tmp = a;
  return a + 10;
}
```

Thread-safe and re-entrant

```
int add10(int a) {
  return a + 10;
}
```

补充：什么是消息传递模型？

详见后面的章节。通过显式的I/O操作来通信；仅可以直接访问私有的地址空间，通过显式的消息来通信（send/receive）

补充：什么是数据并行模型？

详见后面的章节。

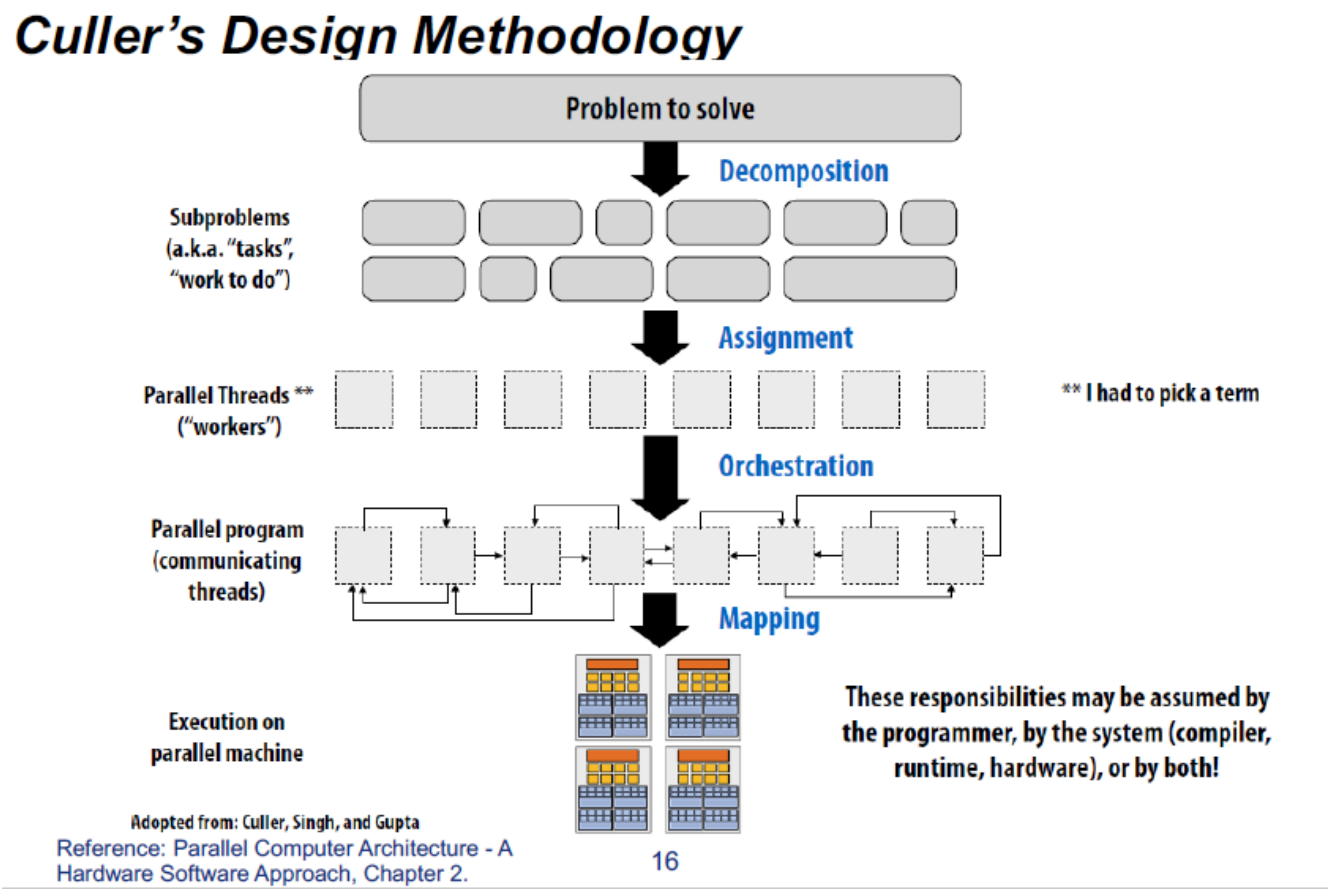
第四讲：并行编程方法论

1. 什么是增量式并行化?

- 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?



分解/解耦 decomposition

- 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

- 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

包括：

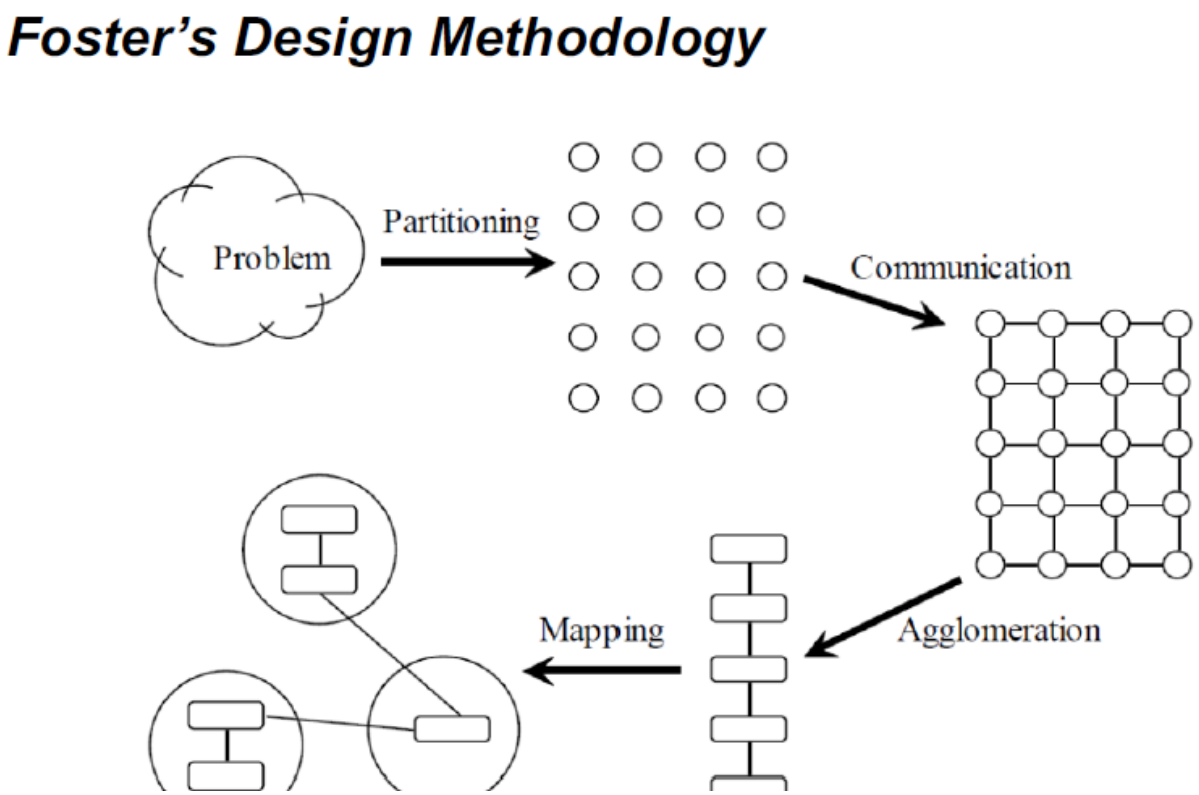
- 实现通信
- 必要时增加同步来保持依赖性
- 优化内存数据结构（加速）
- 调度任务

目标：减少通信/同步开销，保持数据局部性，减少开销

映射mapping

- 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?



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

通信communication

- 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

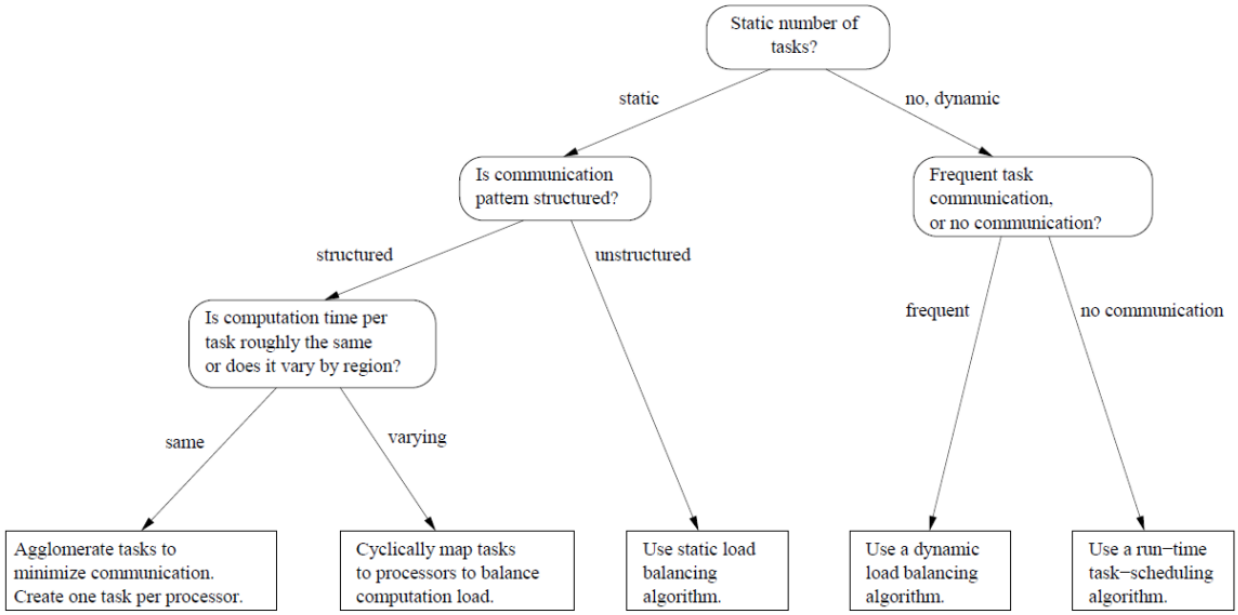
6. 整合的意义是什么?

- 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?



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

- 考虑基于每个处理器一个任务和每个处理器多个任务的设计
- 评估静态和动态任务分配
- 如果选择动态任务分配，则任务分配器不是性能瓶颈
- 如果选择静态任务分配，任务与处理器的比率至少为10:1

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

9. 要点补充

依赖图中，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

第五讲：OpenMP并行编程模型

1. 什么是OpenMP?

Open Multi-Processing （开放多处理过程）提供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

补充：什么是Fork-Join Model?

fork：主线程创建（或唤醒）一组并行线程

join：当team threads完成并行区域构造中的语句时，它们将同步并终止，只留下主线程。

2. OpenMP的主要特点是什么?

- 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
- 一些数据共享可能会导致竞争——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。

（1）设置线程数：`omp_set_num_threads(5);`在#pragma外面，直接调用的函数

（2）获取一些线程信息

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()`

（3）`#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等语句；也没有goto语句

如何确定放的位置？考虑循环依赖，考虑是否会产生大量的线程，考虑共享的变量是否会相互影响

（4）`#pragma omp parallel for private(j)`

对变量j进行了本地拷贝。定义为私有变量后，线程之间不能互相访问。

所有线程都不能使用j在该语句之前定义的值，需要重新赋值；且所有线程都不能修改先前的共享的j

如果变量未定义，需要在循环里重新赋值

（5）`#pragma omp parallel for firstprivate(a)`

在循环开始的时候，继承共享变量的值作为初值。

> private子句的私有变量不能继承同名变量的值，firstprivate则用于实现这一功能-继承并行区域额之外的变量的值，**用于在进入并行区域之前进行一次初始化**

可以声明成firstprivate的有：

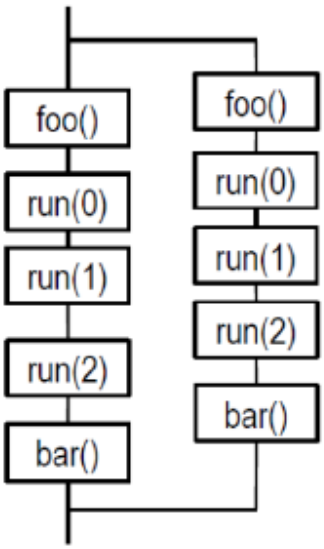
- 基础数据类型、数组、指针、类实例

（6）`#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

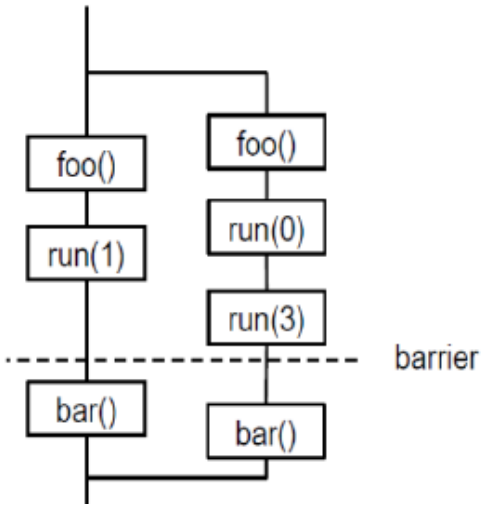
（7）`#pragma omp for 和 #pragma omp parallel`

`parallel`在已经用parallel标记的代码块中使用并将迭代分发到活动线程




```
#pragma omp parallel \
 num_threads(2)
{
    foo();
    for (int i = 0; i < 3; i++)
        run(i);
    bar();
}
```

```
#pragma omp parallel \
 num_threads(2)
{
    foo();
    #pragma omp for
    for (int i = 0; i < 3; i++)
        run(i);
    bar();
}
```



(8) `#pragma omp single`

- 说明只允许一个线程串行执行以下代码：常用于非线程安全的时候，比如有IO操作
- 不执行这一部分代码的线程需要等待，除非说明nowait

```
tid = omp_get_thread_num();
if (tid == 0) {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
}
```

=

```
#pragma omp master
{
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
}
```

由master来执行

≈

```
#pragma omp single nowait
{
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
}
```

串行执行，不一定是master

(9) `#pragma omp sections / section`

一个section由一个线程执行一次

```
#pragma omp parallel shared(a,b,c,d) private(i)
{
#pragma omp sections
   {
#pragma omp section
```

交给一个线程

```
    {
      for (i=0; i<N; i++)
        c[i] = a[i] + b[i];
    }
#pragma omp section      交给另一个线程（如果足够快，也可能是同一个）
    {
      for (i=0; i<N; i++)
        d[i] = a[i] * b[i];
    }
  } /* end of sections */
} /* end of parallel section */
```

（10）`#pragma omp ... reduction(归约操作符:归约变量)`

可以在parallel,for,sections后面添加该关键字

首先对`归约变量`初始化，初始化的值如下所示：

Operator	Initial Value
+	0
*	1
-	0
^	0

Operator	Initial Value
&	~0
|	0
&&	1
||	0

之后，OpenMP用归约变量为每个线程创建一个私有的变量，用来存储自己归约的结果，所以归约的代码不需要critical保护也不会发生冲突。最后再把各自归约后的私有变量归约到归约变量上。

比较特殊的是，在减法时私有变量的初始值也是0，最后再把这些值按加法运算归约到归约变量上。

4. 熟悉OpenMP关键指令的执行过程。

全写在上面了！

5. 要点补充

第六讲：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?

OpenMP并行编程中的正确性问题:

- Barriers（路障）
- Mutual exclusion（互斥）
- Memory fence（内存屏障）

2. 什么是同步，同步的主要方式有哪些?

同步(Synchronization)：无论线程如何调度读取和写入以正确顺序执行，管理共享资源的过程。

主要方式有:

- 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。

关键字Barriers：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`

- 会自动在某些关键句之后执行（隐式barrier）
  - parallel、single、for子句后
- for pragma , single pragma等都有隐式的barrier存在
- 可以使用 `nowait` 关键字来消除

关键字nowait：可用于消除隐式的barrier。

`nowait` 使用示例:

C++

```
#pragma omp parrallel
{
    #pragma omp for nowait
    for(i = 0; i < N; i ++)
        a[i]=alpha(i);
    #pragma omp single nowait
    if(delta<0.0)
        printf("delta<0.0\n");
    #pragma omp for
    for(i = 0; i < N; i++)
        b[i]=beta(i,delta);
}
```

4. OpenMP中竞争的例子。

(1)当线程A和线程B同时执行`area += 4.0/(1.0 + x*x);`会出错。

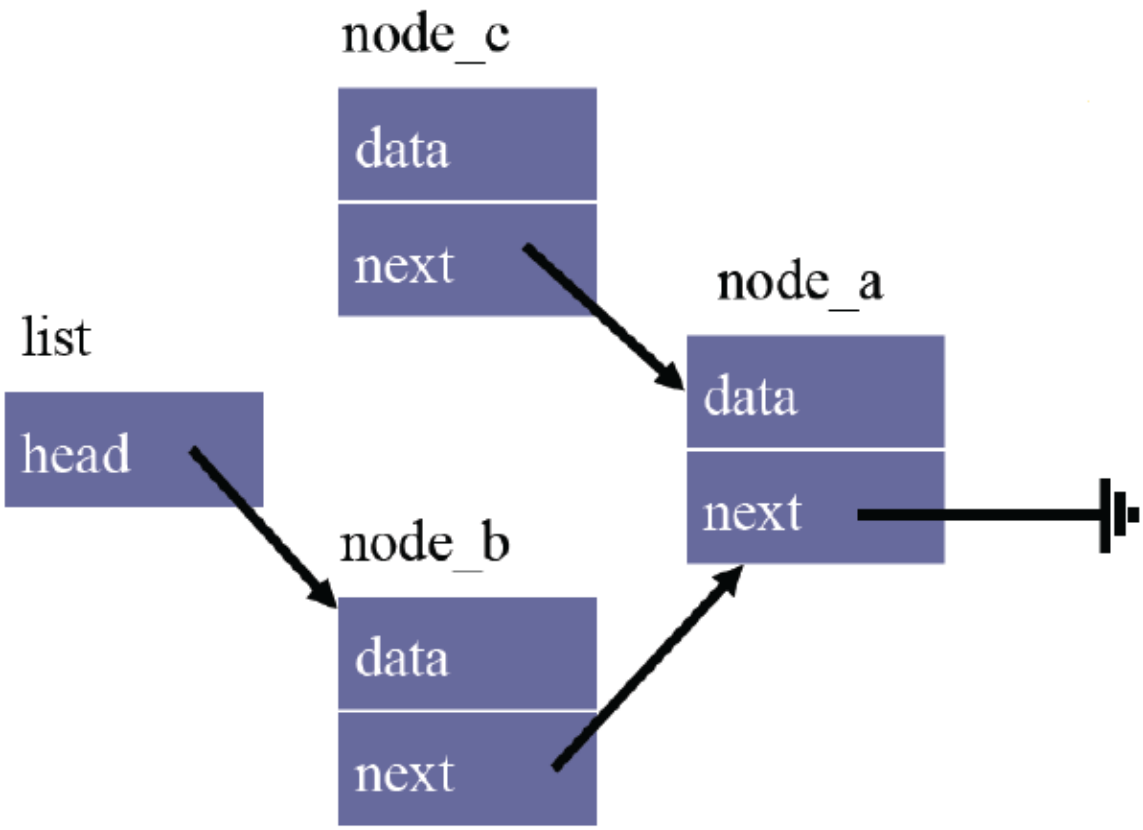
C++

```
double area, pni, x;
int i, n;
...
area = 0.0;
for (i = 0; i <n; i++) {
    x = (i + 0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

(2)一个链表，线程A和线程B同时添加头节点时会出错。内存泄漏，node_c永远不会被回收

Rust

```
void AddHead (struct List* list, struct Node* node) {
    node->next = list->head;
    list->head = node;
}
```



5. OpenMP中避免数据竞争的方式有哪些?

解决方法：

- 使变量对于线程来说是私有的
  - 使用`private`子句
  - 在线程函数中声明变量
  - 在线程栈里分配变量（传参）
- 使用临界区来控制共享资源的访问
  - 互斥和同步

6. OpenMP Critical 与Atomic的主要区别是什么?

关键字critical

- 临界区可以消除race condition；
- 但是该区域执行的过程会变成串行的；
- 我们必须对所有**读取和写入**共享资源的操作都进行保护

C++

```
void AddHead(struct List* list, struct Node* node){
    #pragma omp critical
    {
        node->next = list->head;
        list->head = node;
    }

}
```

关键字atomic

- 仅仅作用域简单的运算符：
  - Pre- or post-increment(++)
  - Pre- or post-decrement(--)
  - Assignment with binary operator(of scalar types) 二元运算赋值`+=`、`-=`
- 且仅作用于一条语句

对比critical和atomic

C++

```
#pragma omp parallel for
{
    for(i = 0; i < n; i ++){
        #pragma omp critical
        x[index[i]]+=WorkOne(i);
        y[i]+=WorkTwo(i);
    }
}
```

critical保护了的语句有：

- 对`WorkOne()`的调用
- 寻找数组`index[i]`的值
- `+=`中的加法操作
- `+=`中的赋值操作

C++

```
#pragma omp parallel for
{
    for(i = 0; i < n; i ++){
        #pragma omp atomic
        x[index[i]]+=WorkOne(i);
        y[i]+=WorkTwo(i);
    }
}
```

atomic保护了的语句有：

- `+=`中的加法操作
- `+=`中的赋值操作

没有对`index[i]`和`WorkOnde(i)`进行互斥操作，所以最后结果可能会出错。

第七讲：OpenMP性能优化

1. 什么是计算效率?

$$E = \frac{T_{串行}}{p * T_{并行}}$$

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?

$$加速比 = \frac{T_{串行} + T_{并行}}{T_{串行} + \frac{T_{并行}}{p} + T_{并行开销}}$$

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）

3. OpenMP中Loop调度的几种方式，执行过程

C++

```C++
#include<iostream>
#include<cmath>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>
#include<unistd.h>
#include <sys/time.h>
using namespace std;

#define GET_TIME(now) { \
   struct timeval t; \
   gettimeofday(&t, NULL); \
   now = t.tv_sec + t.tv_usec/1000000.0; \
}

void dummy(){
    sleep(1);
}
int main(int argc, char* argv[]) {
    int thread_count = strtol(argv[1], NULL, 10);
    int n = 5;
    int i = 0;
    double start, stop;
    #  pragma omp parallel num_threads(thread_count)  \
     default(none) private(i,start,stop) shared(n)
    {
        GET_TIME(start);
    #  pragma omp for //这里决定调度方式

         for(i = 0;i < n ;i++) {
            printf("this is iteration %d from thread
%d\n",i,omp_get_thread_num());
            dummy();
         }

```

```
36          GET_TIME(stop);
37
38          printf("run time of thread%d  : %e\n\n",omp_get_thread_num(), stop-
    start);
39
40      }
41
42       return 0;
43  }
```

`schedule (<type> [, chunk])`

- static：按照指定块大小chunk，轮询分配。无schedule时，默认这种方式 `#  pragma omp for`

  默认即为static，假设有n次循环迭代，t个线程，此时给每个线程静态分配大约n/t次迭代计算。并且因为n/t不一定是整数，因此实际分配的迭代次数可能存在差1的情况

  `#  pragma omp for schedule(static,1)`

  在这种情况下，依次给不同线程分配1个迭代，这样5个迭代0号线程分到0，2，4，1号线程分到1，3，两个线程都延迟三秒

- dynamic：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

  `#  pragma omp for schedule(dynamic,2)`

- guided：每次都要重新计算比例，再动态分配
- auto：编译器和运行时系统决定调度方式
- runtime：调度在运行时决定。根据环境变量来决定使用哪种调度方式

4. OpenMP中Loop转换的方式包括哪几种？熟练掌握

- Loop fission 裂变
- Loop fusion 聚变、合并
- Loop exchange 循环交换

Loop fission（裂变、分离）

> 从具有循环携带依赖性的单循环开始，将循环拆分为两个或多个循环，使新的循环可以并行执行

举例：

C++

```
float *a, *b;
for(int i = 1; i < N; i ++){
    //perfectly parallel
    if(b[i] > 0.0)a[i] = 2.0;
    else a[i]=2.0*fabs(b[i]);
    //loop-carried dependence
    b[i] = a[i-1];
}
```

修改为：

C++

```
float *a, *b;
#pragma omp parallel
{
    #pragma omp for
    for(int i = 1; i < N; i ++){
        if(b[i] > 0.0)a[i] = 2.0;
        else a[i]=2.0*fabs(b[i]);
    }
    #pragma omp for
    for(int i = 1; i < N; i ++)b[i] = a[i-1];
}
```

Loop fusion（聚变、合并）

> 与loop fission相反，我们将多个循环合并在一起

举例一：如果对下面的程序进行并行，需要进行两次的隐式barrier

C++

```
float *a, *b, x, y;
...
for(int i = 0; i < N; i ++)a[i] = foo(i);
x = a[N-1] - a[0];
for(int i = 0; i < N; i ++)b[i] = bar(a[i]);
y = x * b[0] / b[N-1];
```

修改为：只需要进行一次的隐式barrier

C++

```
#pragma omp parallel for
for(int i = 0; i < N; i ++){
    a[i] = foo(i);
    b[i] = bar(a[i]);
}
x = a[N-1] - a[0];
y = x*b[0]/b[N-1];
```

举例二：

C++

```
for(int k = 0; k < N; k ++)
    for(int j = 0; j < M; j ++)
        w_new[k][j] = DoSomeWork(w[k][j],k,j);
```

修改为：

C++

```
for(int kj = 0; kj < N*M; kj ++){
    k = kj / M;
    j = kj % M;
    w_new[k][j] = DoSomeWork(w[k][j],k,j);
}
```

Loop exchange（交换）

> 增大颗粒大小，提升并行程序的局部性

举例：

C++

```
for (int j =1; j < n; j++)
    #pragma omp parallel for
    for (int i = 0; i < m; i++)
        a[i][j] = 2 * a[i][j-1];
```

修改为：

C++

```
#pragma omp parallel for
for (int i =0; i < m; i++)
    for (int j= 1; j < n; j++)
        a[i][j] = 2 * a[i][j-1];
```

第八讲：MPI编程模型

什么库函数连接到对象文件中，包装脚本可以简化编译命令的运行。

程序 3-1　打印来自进程问候语句的 MPI 程序

```
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank,
          comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

1. 什么是MPI编程模型?

MPI是一个跨语言的通讯协议，用于编写并行计算机。支持点对点和广播。MPI是一个信息传递应用程序接口，包括协议和和语义说明，他们指明其如何在各种实现中发挥其特性。MPI的目标是高性能，大规模性，和可移植性。

与OpenMP并行程序不同，MPI是一种基于信息传递的并行编程技术。消息传递接口是一种编程接口标准，而不是一种具体的编程语言。简而言之，MPI标准定义了一组具有可移植性的编程接口。

2. 消息传递性并行编程模型的主要原则是什么?

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。

- 有无buffer指的是有没有缓冲区，blocking指的是，发送时会不会终止自己的计算任务。
- **无缓冲阻塞式**

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

缺点：空转和死锁是非缓冲阻塞发送的主要问题

优点：保证正确

- **缓冲阻塞式**

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，但代价是产生了拷贝的开销

有界的缓冲区大小可能会对性能产生重大影响

由于接收操作阻塞，使用缓冲仍然可能发生死锁

- **非阻塞式**

程序员必须确保发送和接收的语义。（复杂度增大）

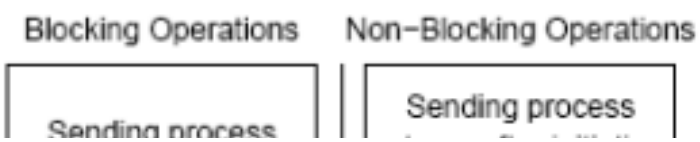
此类非阻塞协议在语义上是安全的之前从发送或接收操作中返回。

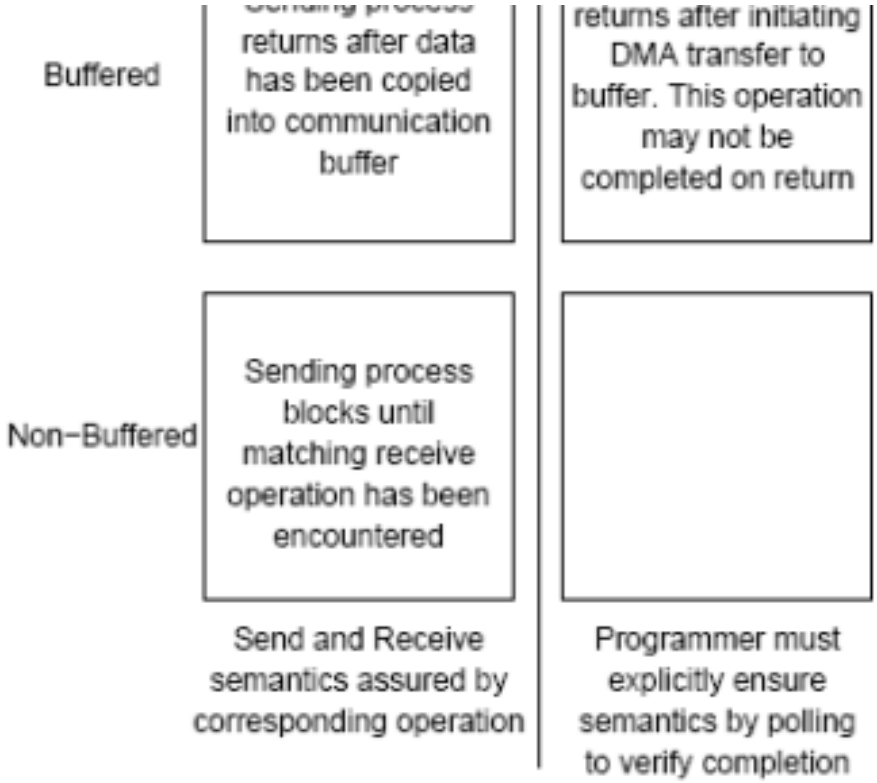
非阻塞操作通常伴随着检查状态操作。（程序员确保安全）

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





> 补充：
>
> （1） `MPI_Send` 即标准发送。在标准模式下，消息可能会被缓冲，也可能不被缓冲。如果消息被缓冲，那么send函数可以立即返回；如果没有被缓冲，那么要直到recv函数接收到数据后才能返回。
>
> （2） `MPI_Ssend` ，指同步发送。无论是否有recv，都可以开始进行发送。会先将数据存储到一个缓冲区中，但是必须等到一个 `MPI_Recv` 接收之后，该函数才能够返回。可以节省缓冲区大小。
>
> （3） `MPI_Bsend` ，指缓冲发送。无论是否有recv，都可以开始进行发送。数据会存储到一个缓冲区中，且函数会立即返回。但是这个函数对缓冲区是有要求的，可能会挤爆缓冲区。
>
> （4） `MPI_Rsend` ，指就绪发送。需要通过通信的方式，匹配到recv之后，才开始发送。需要消耗的时间增加。
>
> （5） `MPI_Isend` 和 `MPI_Irecv` 。他们都多了一个 `MPI_Request` 变量，用来储存通信的信息。因为在调用这个函数的时候，通信并没有结束，因此我们给这次通信一个身份，以便之后确认本次通信的状态。 `MPI_Irecv` 没有 `MPI_Status` 变量，因为 `MPI_Irecv` 是在通信结束之前建立，因而我们不可能知道当前的通信状态。
>
> （6） `MPI_Wait` 和 `MPI_Waitall`
>
> 虽然在等待通信的过程做做一些别的计算是一件很有用的事情，然而我们最终还是会碰到需要等待所有通信都被完成的时刻。这就是 `MPI_Wait` 和 `MPI_Waitall` 所做的事。
>
> - `MPI_Wait` 用来等待某一个通信的完成；

- `MPI_Waitall`用来等待一组通信的完成；
- 调用时，他们只会等待你需要他们等待完成的通信，而其他处理器的通信则会继续发生。

（7） `MPI_Test`和`MPI_Testall`

在非阻塞通信中，我们有时需要检查一个通信是否完成。譬如，你可能会有一个条件循环，当通信完成时会停止；如果你需要重复不停地给一个处理器发送数据，你也可以检查上一个通信是否完成以便立即建立下一个通信，这样数据就会源源不断被发送，提高程序效率。

4. MPI中的关键编程接口。

（1） `MPI_Init(&argc, &argv)`：初始化MPI（一定要有！）

（2） `MPI_Finalize()`：终止化MPI（一定要有！）

（3） `int MPI_Comm_size(MPI_Comm comm, int *size)`：确定进程数

（4） `int MPI_Comm_rank(MPI_Comm comm, int *rank)`：确定我的进程号（从0开始）

（5） `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`：发送信息

（6） `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`：接受信息

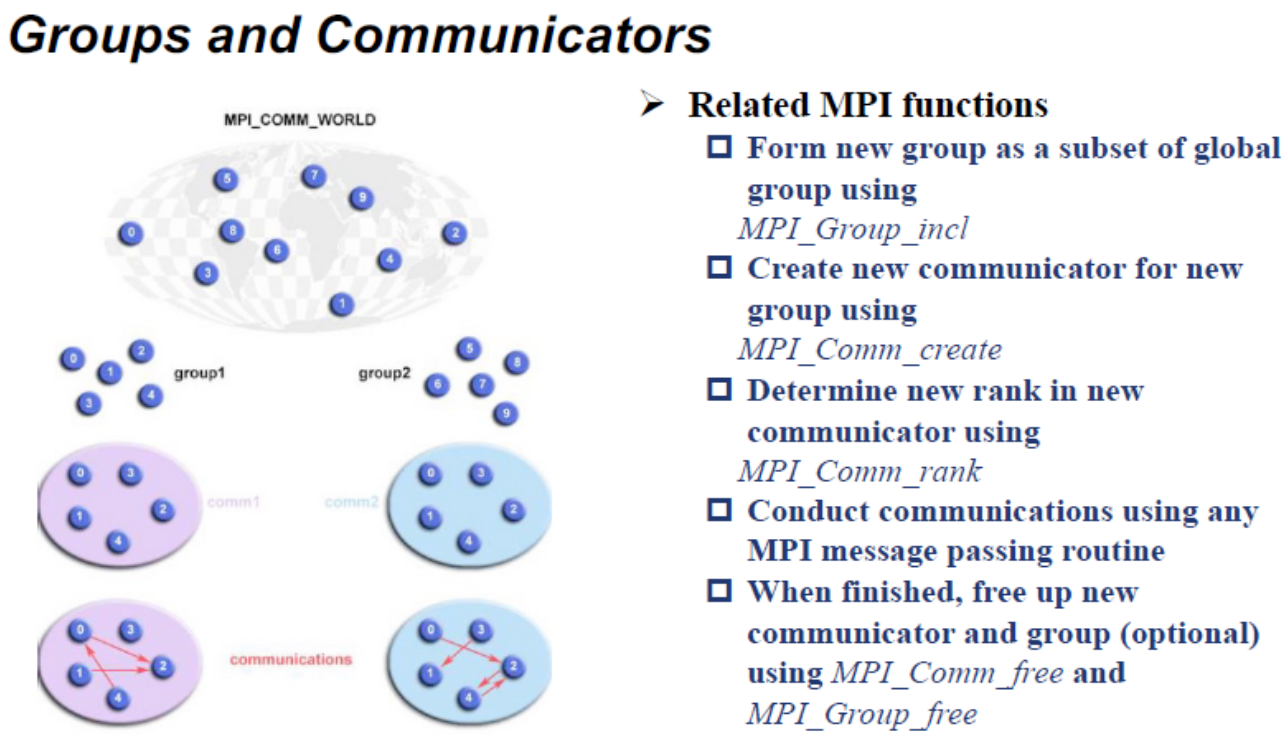
（7）各种类型的send（阻塞P36 和非阻塞P37 都有）

```
int MPI_Sendrecv(
      void*          send_buf_p      /* in  */,
      int            send_buf_size   /* in  */,
      MPI_Datatype   send_buf_type   /* in  */,
      int            dest            /* in  */,
      int            send_tag        /* in  */,
void*          recv_buf_p      /* out */,
int            recv_buf_size   /* in  */,
MPI_Datatype   recv_buf_type   /* in  */,
int            source          /* in  */,
int            recv_tag        /* in  */,
MPI_Comm       communicator    /* in  */,
MPI_Status*    status_p        /* in  */);
```

5. 什么是通信子?

- 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（感觉不是重点）：



- 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?

- 调节send、recv的顺序；
- 使用`MPI_Sendrecv`

7. MPI中的集群通信操作子有哪些？原理是什么?

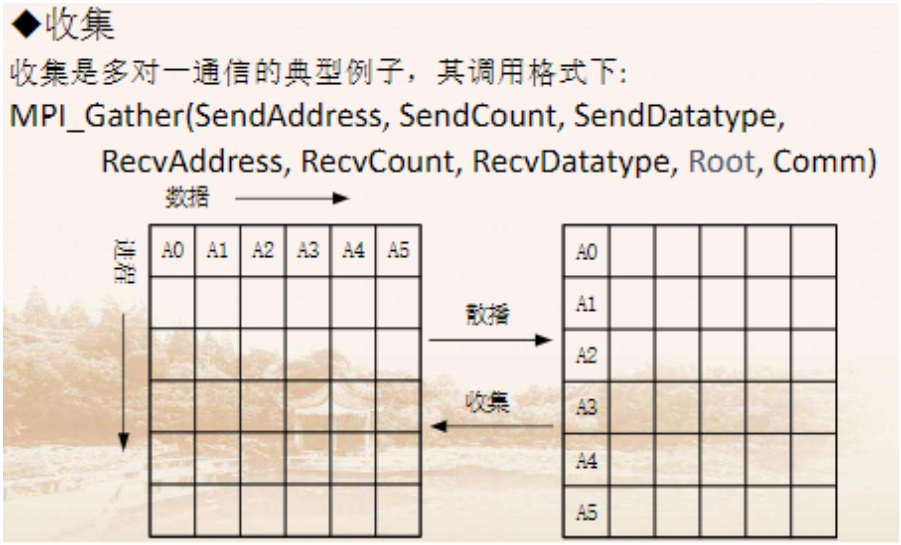
(1)`int MPI_Barrier(MPI_Comm comm)`

(2)`int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source,MPI_Comm comm)`

(3)`int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int target, MPI_Comm comm)`

(4)`int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`

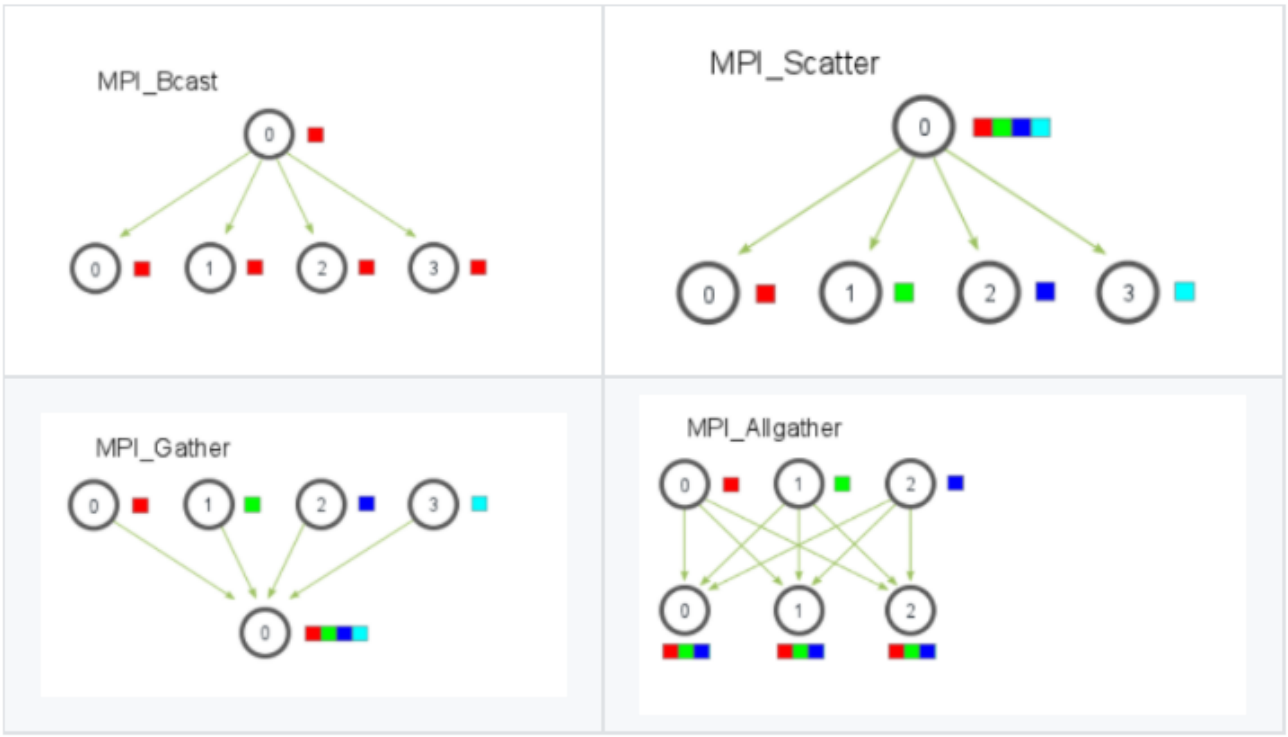
gather



(5)计算前缀和(可以看成特殊的MPI_Reduce)：`int MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`

(6) `int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, MPI_Comm comm)`

(7) `int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, MPI_Comm comm)`



8. 要点补充

第九讲：MPI与OpenMP联合编程

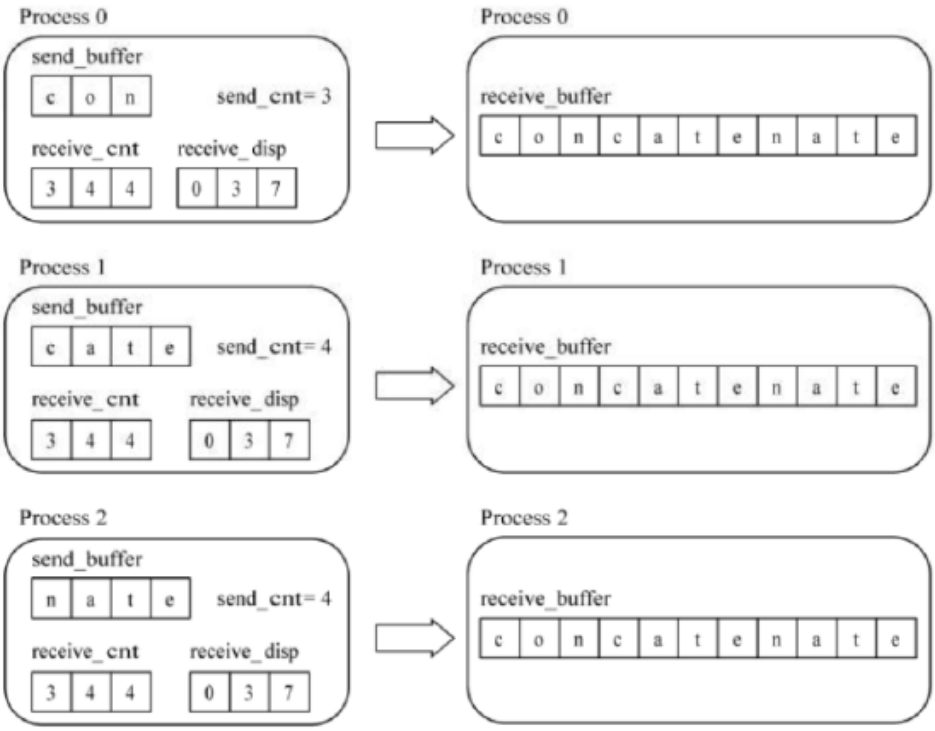
1. 如何利用MPI实现Matrix-vector乘积？不同实现的特点是什么？

- 按行划分

通过域分解进行分区，原始任务与矩阵行和整个向量有关

流程：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

可以使用allgatherv函数：允许每个进程的数据量不同

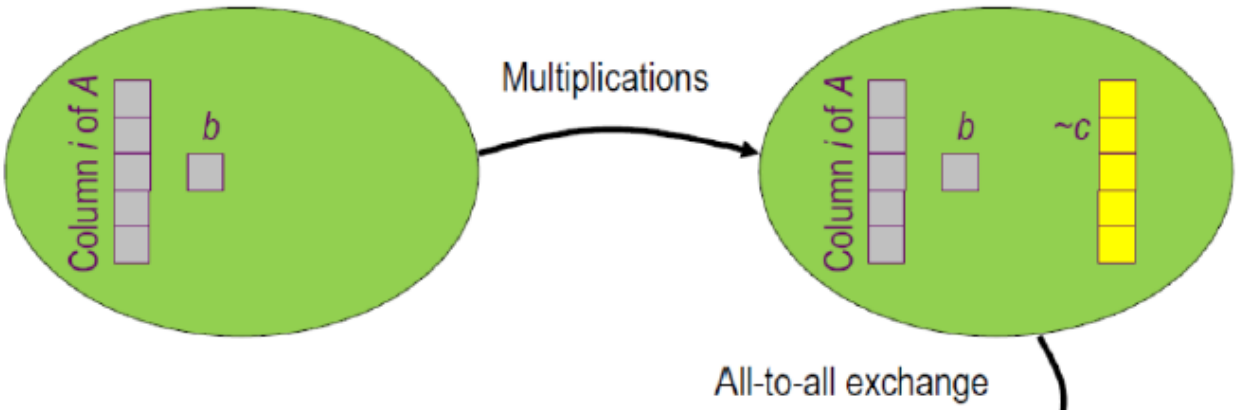


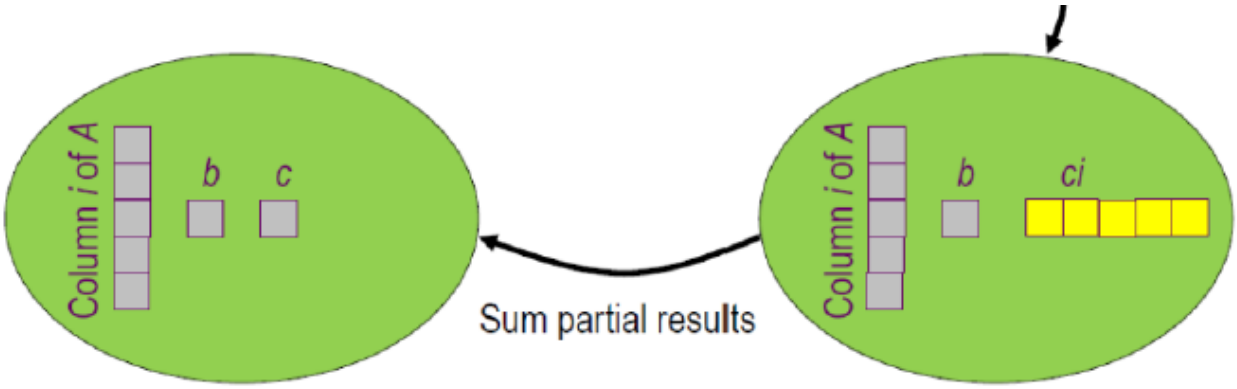
特点：**任务数固定**（行数固定）；**规则通信方式**（all-gather)；**每个任务计算时间相同**（task划分均匀），策略：按行归并+每个进程创建一个任务

- 按列划分

通过域分解进行分区，任务与矩阵列和向量的元素有关

流程：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）





特点：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

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

流程：

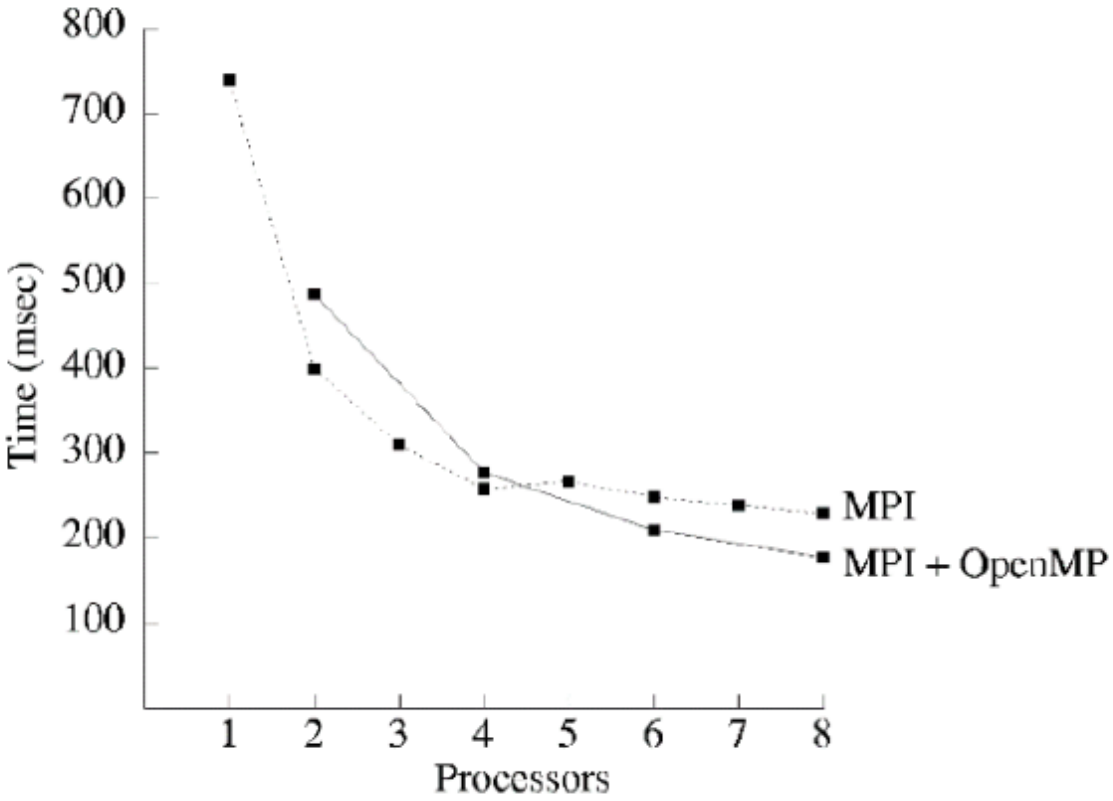
- 重新分配b -> 矩阵-向量点乘 -> 归并 -> 把b从欧诺个第1列弄到第1行：分类讨论：当前
- 进程数量是否是平方数?
  - 如果是平方数：则第一列发送b的各部分，到第一行接收
  - 如果不是平方数：将b聚集到进程(0,0)，然后进程(0,0)广播到第一行

-> 第一行的进程在各列中分散

特点：随着进程数增加，棋盘化的优点就体现出来了：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?

- 更快
- 降低通信开销
  - 消息通过m*k个进程传递
  - 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



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?

第十讲：GPGPU、CUDA和OpenCL编程模型

1. CUDA的含义是什么?

- CUDA（Compute Unified Device Architecture）：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**
  - 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**：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

- **与传统的多线程设计的不同点**
  - 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?

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。

> 要深刻理解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下图所示，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，对于图中结构（主要水平方向为x轴），定义的grid和block如下所示，kernel在调用时也必须通过执行配置`<<<grid, block>>>`来指定kernel所使用的线程数及结构。

C++

```
dim3 grid(3, 2);
dim3 block(5, 3);
kernel_fun<<< grid, block >>>(prams...);
```

device函数：只能由GPU访问（设备级）

4. CUDA的编程样例。

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}
```

```
int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays, fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

5. CUDA的线程分层结构

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

CUDA 线程:

- 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:

- 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：

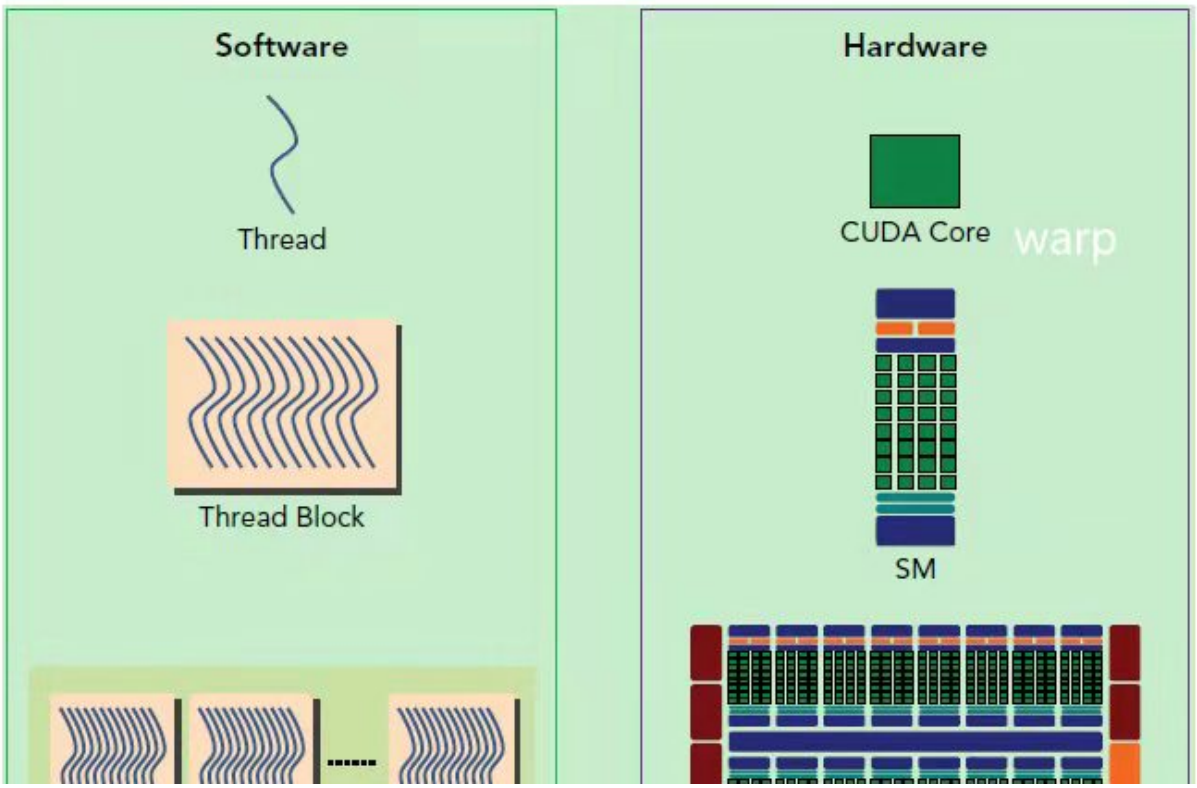
- 是虚拟化多线程核心
- 执行中等粒度的数据并行任务（所有线程共享指令cache）
- 块中的线程通过屏障内部机制进行同步，并通过快速的片上共享存储器进行通信（同步发生在这一级别，且所有线程都要同步！）
- 以1维、2维或3维组织

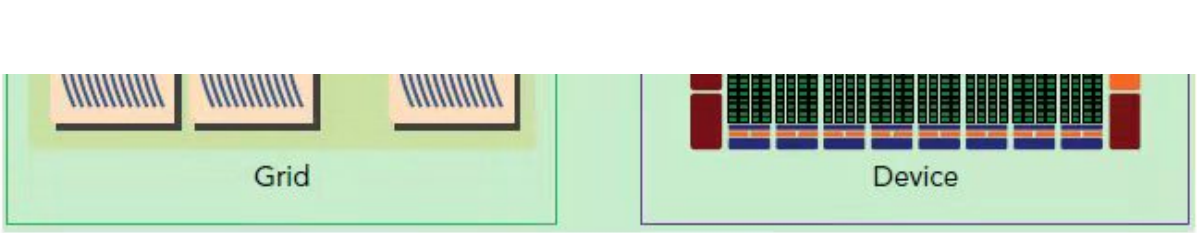
Grid网格：

- 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：

- 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





CUDA编程的逻辑层和物理层

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
- 每个线程块都可以私有访问可配置数量的暂存器
- 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
- 由于Cuda的图形遗产，内存层次结构中还有其他只读组件
- 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
- 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
- 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
- 主机<->设备内存的传输是通过PCI-E上的cudaMemcpy（）进行的，非常昂贵
- 通过多个CPU线程管理多个设备

7. CUDA中的内存访问冲突

- 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

8. OpenCL运行时编译过程。

（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`

第十一讲：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）

对于MPI而言：

- **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）

对于MapReduce而言：

- **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？

使用特殊的map() 和 reduce()函数处理数据

- 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：

- 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

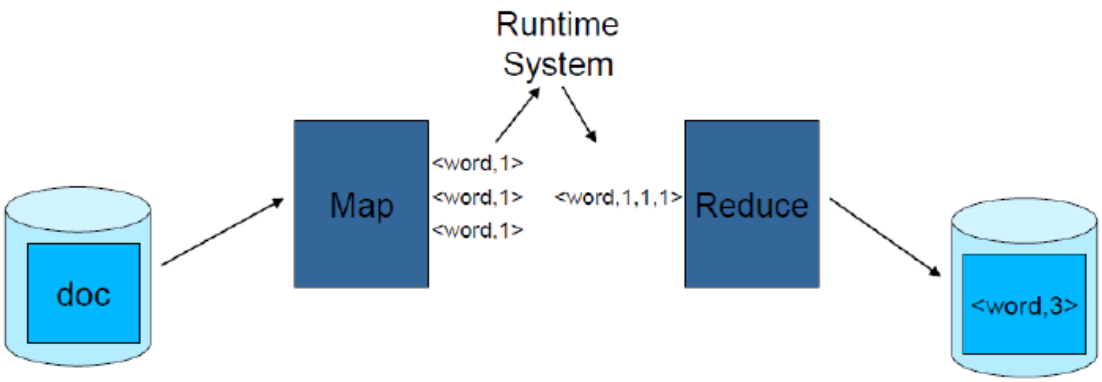
reduce：

- 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

4. MapReduce的简单实现。如Hello World例子。

MapReduce Examples

- Word frequency

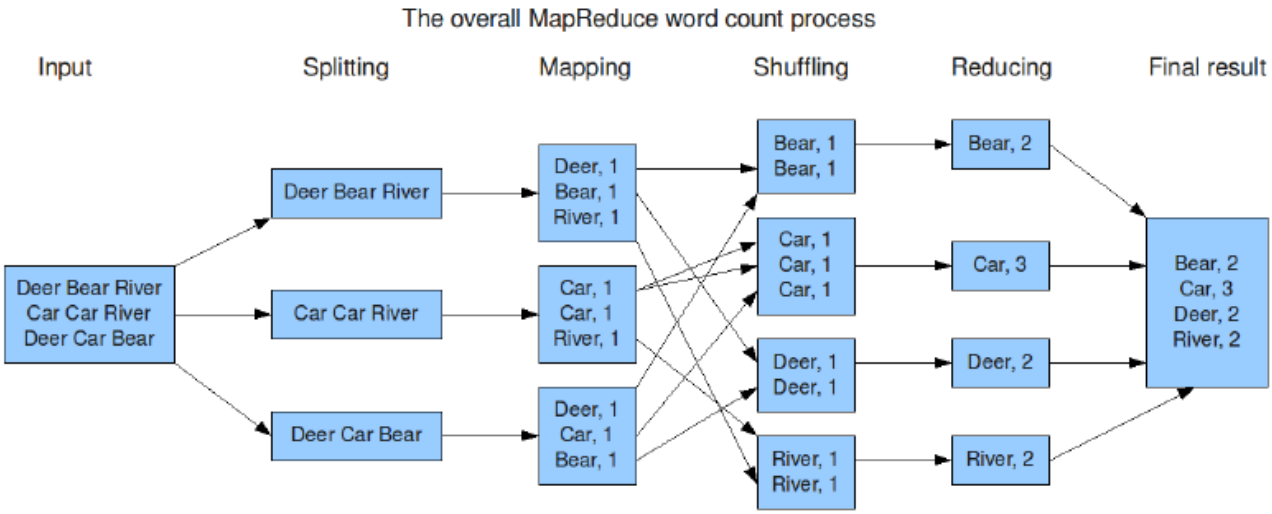


Example: Count Word Occurrences

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
```

```
    EmitIntermediate(w, "1");

reduce(String output_key, Iterator
  intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
 Emit(AsString(result));
```



5. MapReduce具有哪些容错措施?

- 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?

- 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?

- 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
- 统计单词频率

第十二讲：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

> hadoop框架侧重于离线大批量计算，而spark则侧重内存以及实时计算。
>
> 一、应用场景不同
>
> Hadoop和Spark两者都是大数据框架，但是各自应用场景是不同的。Hadoop是一个分布式数据存储架构，它将巨大的数据集分派到一个由普通计算机组成的集群中的多个节点进行存储，降低了硬件的成本。Spark是那么一个专门用来对那些分布式存储的大数据进行处理的工具，它要借助hdfs的数据存储。
>
> 二、处理速度不同
>
> hadoop的MapReduce是分步对数据进行处理的，从磁盘中读取数据，进行一次处理，将结果写到磁盘，然后在从磁盘中读取更新后的数据，再次进行的处理，最后再将结果存入磁盘，这存取磁盘的过程会影响处理速度。spark从磁盘中读取数据，把中间数据放到内存中，，完成所有必须的分析处理，将结果写回集群，所以spark更快。
>
> 三、容错性不同
>
> Hadoop将每次处理后的数据都写入到磁盘上，基本谈不上断电或者出错数据丢失的情况。Spark的数据对象存储在弹性分布式数据集 RDD，RDD是分布在一组节点中的只读对象集合，如果数据集一部分丢失，则可以根据于数据衍生过程对它们进行重建。而且RDD 计算时可以通过 CheckPoint 来实现容错。

> 四、Hadoop是一个由Apache基金会所开发的分布式系统基础架构。 Hadoop实现了一个分布式文件系统HDFS。HDFS有高容错性的特点，并且设计用来部署在低廉的硬件上；而且它提供高吞吐量来访问应用程序的数据，适合那些有着超大数据集的应用程序。Hadoop的框架最核心的设计就是：HDFS和MapReduce。HDFS为海量的数据提供了存储，而MapReduce则为海量的数据提供了计算

2. 传统MapReduce的主要缺点是什么?

- 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，...)
- 可以在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

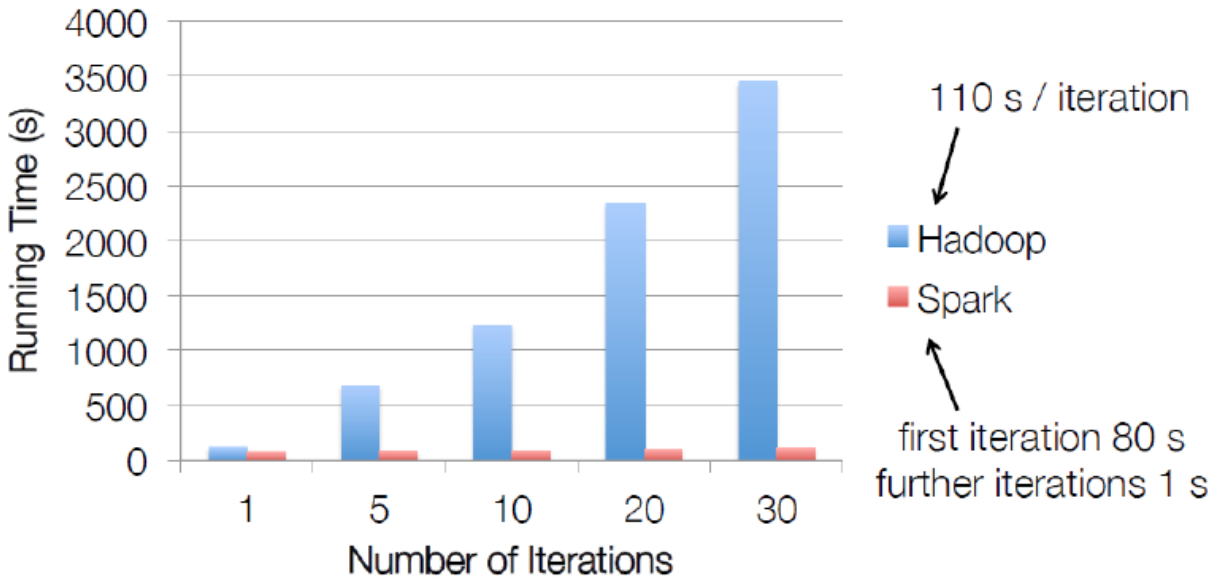
4. Spark样例程序。

Logistic Regression

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

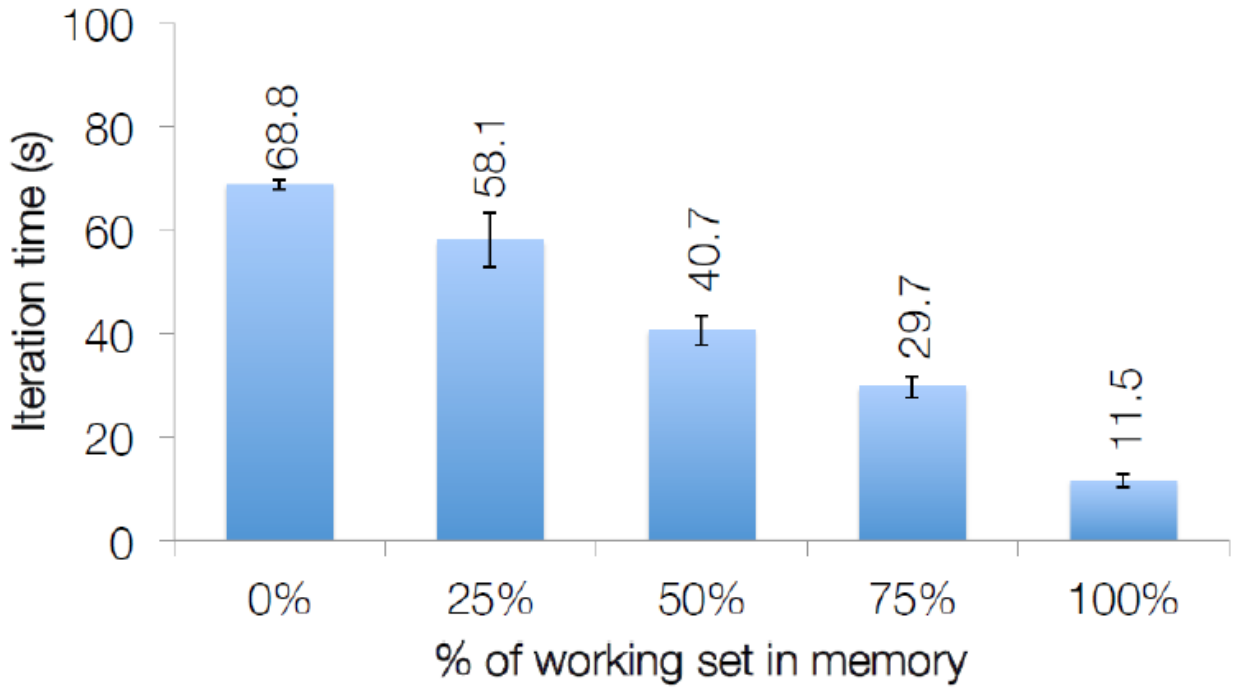
```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= alpha * gradient
}
```

P26 hadoop&spark速度比较



100 GB of data on 50 m1.xlarge EC2 machines

P27 内存成比例增加，速度也成比例增加



第十三讲：离散搜索与负载均衡

1. 深度优先搜索的主要流程。

- 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。

- 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。

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($p^2 log(p)$) (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($p^2 log(p)$), (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。

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？

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。

第十四讲：并行图算法

1. 最小生成树的串行和并行算法原理；

串行算法

Prim的MST算法是一种贪婪算法。

- 从选择任意顶点开始，将其包含到当前MST中。
- 通过插入与当前MST中已存在的一个顶点最近的顶点来增长当前MST。

Minimum Spanning Tree: Prim's Algorithm

```
1.    procedure PRIM_MST(V, E, w, r)
2.    begin
3.        V_T := {r};
4.        d[r] := 0;
5.        for all v ∈ (V − V_T) do
6.            if edge (r, v) exists set d[v] := w(r, v);
7.            else set d[v] := ∞;
8.        while V_T ≠ V do
9.        begin
10.           find a vertex u such that d[u] := min{d[v]|v ∈ (V − V_T)};
11.           V_T := V_T ∪ {u};
12.           for all v ∈ (V − V_T) do
13.               d[v] := min{d[v], w(u, v)};
14.       endwhile
15.   end PRIM_MST
```

Prim's sequential minimum spanning tree algorithm

并行算法

该算法在n个外部迭代中工作-很难同时执行这些迭代。

- **内层循环相对容易并行化（重点，就是对最里面的循环做并行）**

设p为进程数，n为顶点数。

- The adjacency matrix is partitioned in a 1-D block fashion, with distance vector d partitioned accordingly.邻接矩阵以1-D块方式进行划分，距离向量D相应地进行划分。
- In each step, a processor selects the locally closest node, followed by a global reduction to select globally closest node.在每个步骤中，处理器选择本地最近的节点，然后进行全局缩减以选择全局最近的节点。
- This node is inserted into MST, and the choice is broadcast to all processors. Each processor updates its part of the d vector locally. 此节点插入MST，选择广播到所有处理器。每个处理器在本地更新其d向量的一部分

2. Prim并行算法的复杂度；

The parallel time per iteration is $O(n/p + \log p)$.

- **The cost of a broadcast is $O(\log p)$.**
- **The cost of local update of the d vector is $O(n/p)$.**

The total parallel time is given by $O(n^2/p + n \log p)$.

3. 单源最短路径算法的原理；

与prim算法类似，对最内层的循环做并行

4. 单源最短路径算法的并行算法的复杂度；

和 Prim 算法相同。

5. 基于Dijkstra的并行All-pair最短路径算法的复杂度；p24,p25

6. Floyd并行算法的复杂度；

7. 要点补充

第十五讲：性能优化之一（任务分派和调度）

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

- 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）

半静态调度 semi-static assignment

- 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

- 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（事先无法预测x是否为素数）；
  - 使用共享变量“锁”，同步开销非常大
- 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？

同上一问

3. 如何选择任务的粒度？

- 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）

4. Cilk_spawn的原理是什么？

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）
  - 并行快排例子：P28

5. Cilk_sync的原理是什么？

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. Cilk_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）

潜在性能问题：

- 重量级派生操作
- 并发运行的线程比内核多得多
- 上下文切换开销
- 工作集比需要的工作集大，缓存位置少

7. Cilk_spawn中任务在不同线程之间steal的过程。

- 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. Cilk_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）

- 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

第十六讲：性能优化之一（局部性、通行和竞争）

1. 吞吐量和延迟的定义；

Bandwidth: The rate at which operations are performed. Memory can provide data to the processor at 25 GB/sec.（带宽：执行操作的速率。内存可以以25 GB/秒的速度向处理器提供数据。单位时间完成的任务数量，高通量计算）

Latency: The amount of time needed for an operation to complete. A memory load that misses the cache has a latency of 200 cycles（延迟：操作完成所需的时间量。未命中缓存的内存负载的延迟为200个周期）

2. 提高程序吞吐有哪些方法?

相当于提升开车的速度，即降低 Latency；增加车道；流水线：提高了 Throughput！但没有提高 Latency。

3. 通信时间和通信代价的定义；

Total communication time = overhead(开销) + occupancy(占用) + network delay(网络延迟)

Total communication cost = communication time - overlap

Overlap: portion of communication performed concurrently with other work, "Other work" can be computation or other communication(重叠,与其他工作同时执行的通信部分，“其他工作”可以是计算或其他通信)

4. 什么是人为通信？什么是天然通信？结合具体的例子说明

Inherent communication(天然通信)：必须在并行算法中进行的通信。通信是算法的基础。**在p25的例子中**，每个进程获取了三行元素，但是计算边界元素时必须获取相邻行的元素，这个时候产生的就是天然通信。

从根本上必须在处理器之间移动以执行给定指定分配的算法的信息（假设无限容量缓存、最小粒度传输等）（运算过程中产生的天然通信，即使有足够的cache，天然通信都不可避免）

Artifactual communication(人为通信)：all other communication (artifactual communication results from practical details of system implementation). 所有其他通信(人为通信来自系统实现的实际细节)（很大可能可以避免）

人为通信例子：

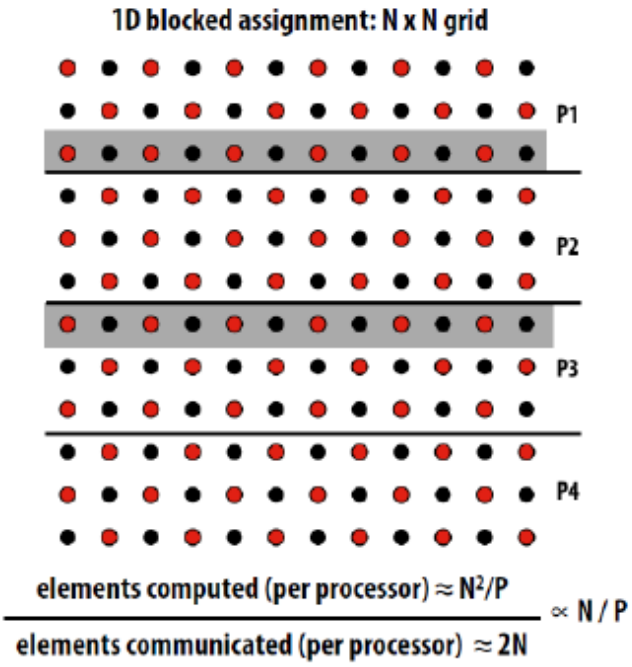
- 系统可能具有最小的传输粒度（结果：系统必须传输比所需更多的数据）
- 程序加载一个4字节浮点值，但必须从内存传输整个64字节缓存线（通信量比需要多16倍）
- 数据存储时，连续存储16个4byte的浮点数；必须先load再store。其中load是不必要的
- 数据在分布式内存中的位置不佳（数据不在访问最多的处理器附近）数据存放的不合理，破坏了空间局部性
- 有限的复制容量（同一数据多次传输到处理器，因为本地存储（例如缓存）太小，无法在访问之间保留）（有限的存储容量，cache不够）

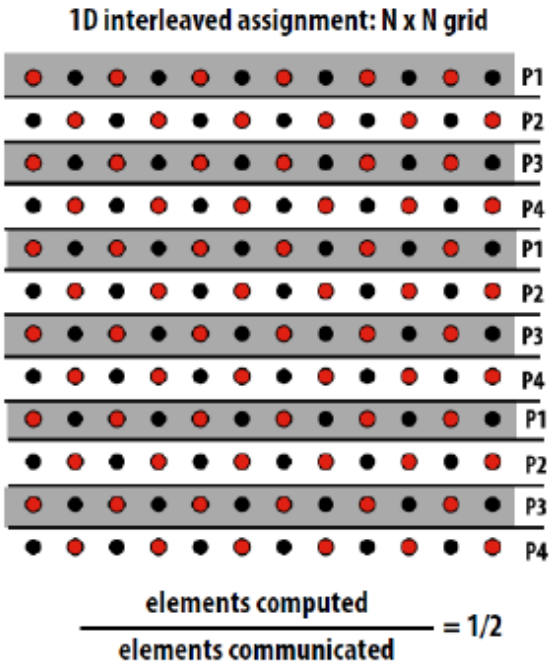
5. 减少通信的方法有哪些?

（1）好的划分可以减少天然通信量。

$$\text{运算密度} = \frac{\text{amount of computation (e.g., instructions)}}{\text{amount of communication (e.g., bytes)}}$$

运算次数 / 通信的字节数

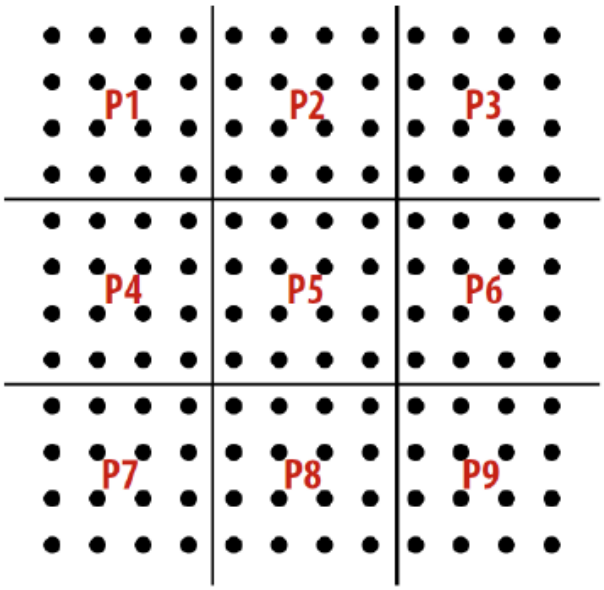




每计算一行N，需要进行2N次通信

Reducing inherent communication

2D blocked assignment: N x N grid



N^2 **elements**

P **processors**

elements computed: (per processor) $\frac{N^2}{P}$

elements communicated: (per processor) $\propto \frac{N}{\sqrt{P}}$

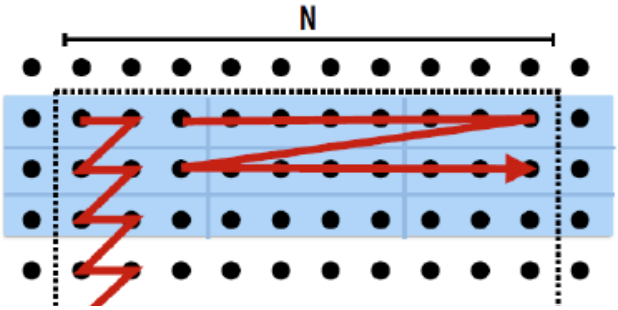
arithmetic intensity: $\frac{N}{\sqrt{P}}$

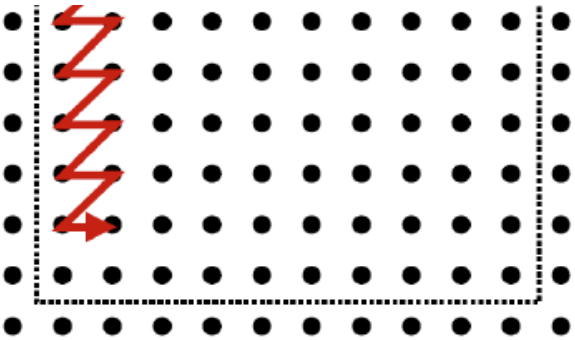
Asymptotically better communication scaling than 1D blocked assignment

Communication costs increase sub-linearly with P

Assignment captures 2D locality of algorithm

（2）通过改变网格遍历顺序改善时间局部性(p37)





（3）通过**合并循环**改进时间局部性

（4）通过**共享数据**提高计算密度

6. 减少竞争的方法有哪些?

以n body问题为例

- (1)一个可能的答案是按单元分解工作：对于每个单元，独立计算其中的粒子（消除争用，因为不需要同步）
  - 并行性不足：只有16个并行任务，但需要数千个独立任务才能有效利用GPU）
  - 工作效率低下：在单元中执行16倍以上的粒子计算量比顺序算法大
- (2)为每个CUDA线程指定一个粒子。线程计算包含粒子的单元，然后原子地更新列表。
  - 大规模争用：数千个线程争用更新单个共享数据结构的权限

C++

```
lock(cell_list_lock)
append p to cell_list[c]
unlock(cell_list_lock)
```

- (3)通过使用**每单元锁**缓解**单个全局锁**的争用-假设粒子在二维空间中均匀分布~比解决方案2少16倍的争用

C++

```
lock(cell_list_lock[c])
append p to cell_list[c]
unlock(cell_list_lock[c])
```

- (4)计算部分结果+合并

并行生成M个“部分”网格，然后合并

-示例：创建M个线程块（至少与SMX内核的线程块数量相同）

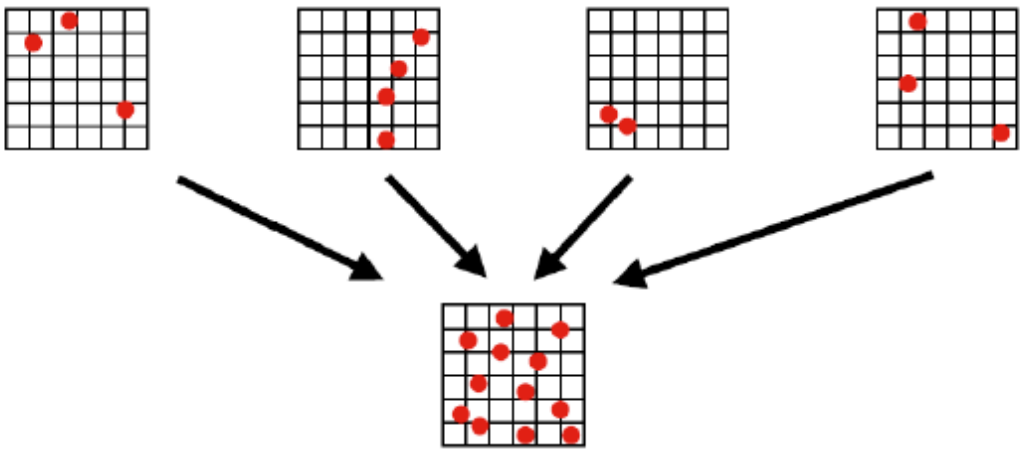
-每个块指定N/M个粒子

-线程块中的所有线程更新相同的网格

-支持更快的同步：争用减少了M倍，而且同步成本更低，因为它是在块本地变量上执行的（在CUDA共享内存中）

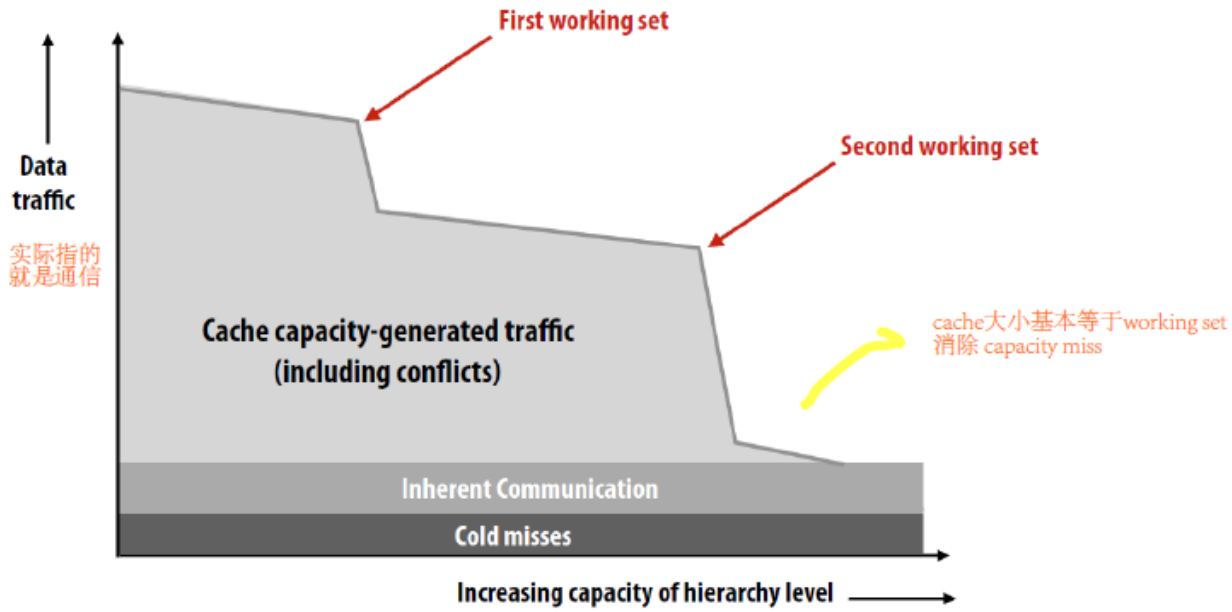
-需要额外的工作：在计算结束时合并M个网格

-需要额外的内存占用：存储M个列表网格，而不是1个



7. 要点补充

- Cold miss 首次数据访问，不可避免
- Capacity miss 工作集太大，只能到内存去取(工作集是程序执行过程中，访问过的页框集合。)
- Conflict miss 冲突导致cache miss
- Communication miss 通信导致的miss



cache例题

问题2

2. Consider a memory system with a level 1 cache of 32 KB and DRAM of 512 MB with the processor operating at 1 GHz. The latency to L1 cache is one cycle and the latency to DRAM is 100 cycles. In each memory cycle, the processor fetches four words (cache line size is four words). What is the peak achievable performance of a dot product of two vectors? Note: Where necessary, assume an optimal cache placement policy.

```
/* dot product loop */
for (i = 0; i < dim; i++)
        dot_prod += a[i] * b[i];
```

由题目可知一个cache line = 4 words,

如果是在32位系统中，一行存储4个int类型，所以每当遇到一次cache-miss时，就会会从内存 DRAM 中装入 4个int值，而一次循环可能出现两次cache-miss，则一次循环后cache中将装入$a[i]$~$a[i+3]$和$b[i]$~$b[i+3]$,接下来三次循环都会hit，所以每四次循环构成一个周期，在这个周期内，每次循环有1次乘法和1次加法，所以4个循环有8次运算，即8次浮点运算操作，又频率为1GHz，可得一个cycle的时间为$10^{-9}s$，所以四个循环所花的时间为：第 1 次循环中从 DRAM 中取数的延迟+第 2、3、4 次循环中从 cache 中取数的延迟=$2 \times 100 + 3 \times 2 \times 1 = 206cycles = 206 \times 10^{-9}s$

所以peak achievable performance of a dot product of two vector为
$8 \div (206 \times 10^{-9}) = 38.8MFLOPS$

同理若是在64位系统，cache一行可以存储8个int类型，所以将 8 个循环作为一个单位，则总时间是第 1 次循环中从 DRAM 中取数的延迟+第 2~8 次循 环中从 cache 中取数的延迟=
$2 \times 100 + 7 \times 2 = 214cycles$。而以 8 个循环作为一个单位，每 8 次 循环中进行了 8 次乘法操作和 8 次加法操作，所以一共有 16 次浮点操作。所以 peak achievable performance of a dot product of two vectors 是

$16 \div (214 \times 10^{-9}) = 74.8MFLOPS$

问题3

3. Now consider the problem of multiplying a dense matrix with a vector using a two-loop dot-product formulation. The matrix is of dimension 4*K* x 4*K*. (Each row of the matrix takes 16 KB of storage.) What is the peak achievable performance of this technique using a two- loop dot-product based matrix-vector product?

```
/* matrix-vector product loop */
for (i = 0; i < dim; i++)
        for (j = 0; i < dim; j++)
                c[i] += a[i][j] * b[j];
```

一个cache line = 4 words,

在32位系统中，一个cache line存放4个int，因为要计算peak performance，那么在内层循环的过程中,c[i]保持不变，这个延迟可以忽略不计，b向量总共占据空间刚好是16KB，为cache容量的一半，所以cache也能完全缓存好b向量，因此要考虑的只有a矩阵的cache-miss，因此，可以将 4 次内层循环看成一个单位，总时间是：第 1 次循环中 a 矩阵从 DRAM 中取数造成的延迟+第 2、3、4 次循环中 a 矩阵从 cache 中取数造成的延迟+第 1、2、3、4 次 循环中 b 向量从 cache 中取数造成的延迟 $= 100 + 3 + 4 = 107cycles$。而 4 次内层循环中，共有 4 次乘法操作和 4 次加法操作，共 8 次浮点数操作。所以peak achievable performance of this technique using a two

-loop dot-product based matrix-vector product是$8 \div (107 \times 10^{-9}) = 74.77MFLOPS$

在64位系统中，将 8 次内层循环看成一个单位，

总时间是：第 1 次循环中 a 矩阵从 DRAM 中取数造成的延迟+第 2~8 次循环中 a 矩阵从 cache 中取数造成的延迟+第1~8次循环中b向量从cache中取数造成的延迟$= 100 + 7 + 8 = 115cycles$。 而 8 次内层循环中，共有 8 次乘法操作和 8 次加法操作，共 16 次浮点数操作。所以 peak achievable performance of this technique using a two-loop dot-product based matrix-vector product 是

$16 \div (115 \times 10^{-9}) = 139.13MFLOPS$