CPU代码原理

1. CPU图计算背景

1.1 现有CPU图计算方案难题

- 可编程性差：许多时候图非常大，为了加速图计算可以使用并行编程。但是要实现高效的并行程序是非常困难的，尤其是对于共享内存机器。
- 效率低：图形分析的特定领域语言 (domain-specific languages DSL)作用是简化程序编写。程序表示为顶点运算符的迭代应用，其中顶点运算符是读取和写入节点及其直接邻居的函数。通过以批量同步方式将运算符同时应用于图的多个节点来利用并行性；同步调度插入必要的同步以确保一轮中的所有操作在下一轮开始之前完成。虽然DSL解决了易用性的问题，但是这种编程模型不足以实现高性能、通用的图形分析，在这种情况下，通用是指算法和被分析的输入图形类型的多样性。

1.2 同步调度和异步调度

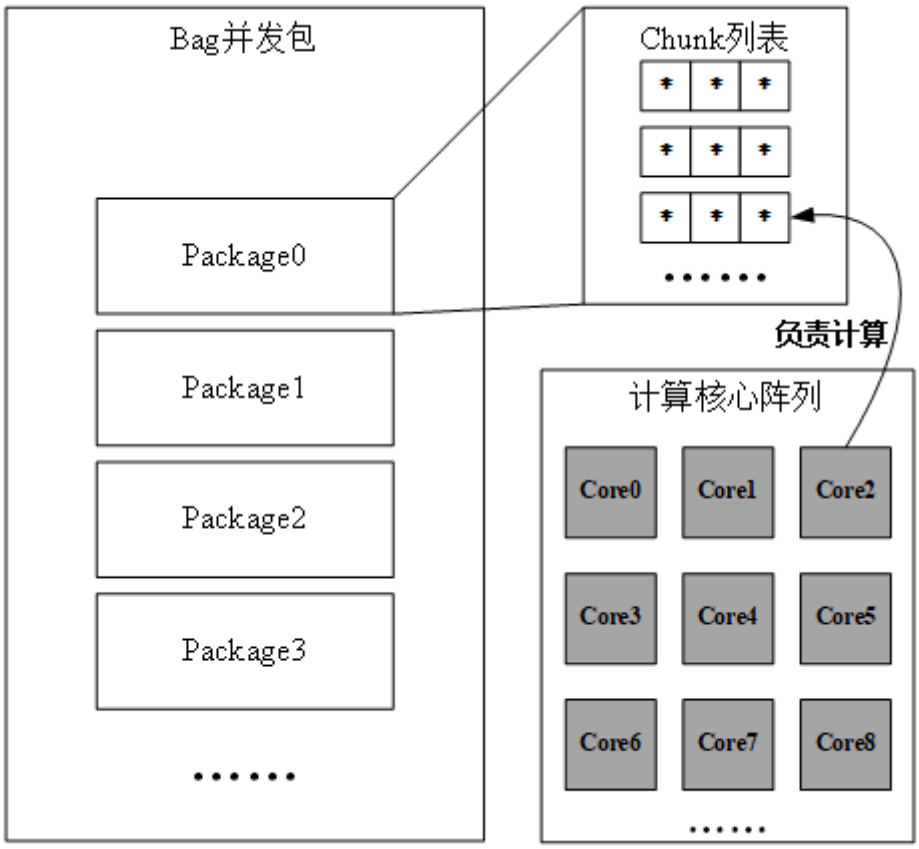
- 同步调度将活动的调度限制为执行轮次，如批量同步并行 (BSP) 模型 。整个程序的执行分为一系列由屏障同步分隔的超步。在每个超级步骤中，选择并执行活动节点的子集。在共享内存实现中写入共享内存，或在分布式内存实现中写入消息，被认为是从一个超步到下一个超步的通信。因此，每个超步都包括根据前一个超步的通信更新内存、执行计算，然后向下一个超步发出通信。对同一位置的多次更新以不同的方式解决，就像各种 PRAM 模型所做的那样，例如通过使用归约操作 。
- 在异步调度中，活动是用事务语义执行的，因此它们的执行看起来是原子的和孤立的。并行活动是可序列化的，因此整个程序的输出与一些顺序交错的活动相同。线程从工作列表中检索活动节点并执行相应的活动，仅在需要时与其他线程同步以确保事务语义。这种细粒度同步可以通过对图元素使用逻辑锁或无锁操作的推测执行来实现。当活动提交时，活动的副作用在外部变得可见。

DepGraph指出，传统的以点为中心的计算模型通常采用**同步调度**策略，虽然能够满足处理一般性的图分析任务的要求，但在执行高性能图分析任务时效率不高，而基于数据驱动的**异步调度**，应用程序在其数据可用时被安排执行，所以能够实现更好的性能（不是所有算法都能采用异步调度方式实现）。为了提高效率，异步调度需要特定于应用程序的**优先级函数**，这些函数必须由程序员提供，因此必须得到编程模型的支持。为了实现异步调度，DepGraph相比之前的图计算系统做出了如下改进：**拓扑感知工作窃取调度程序、优先级调度程序和可扩展数据结构库。**

2. 拓扑感知的任务调度

DepGraph调度器的核心是基于机器拓扑感知的任务调度，当应用没有设置特定优先级时，它会使用并发包装载任务，这些任务是无序可并行执行的，可以插入新任务也可以请求内部任务。具体过程如下图所示，每个计算核心都有一个chunk结构，它是一个环状缓冲区，可以容纳8-64（由用户在编译时确定）个任务，计算核心可以用它插入新任务或获取任务工作。DepGraph的调度分级如下，一个Bag并发包中有多个package列表，每个package列表记录一组chunk数据结构，如果某个计算核心的chunk已

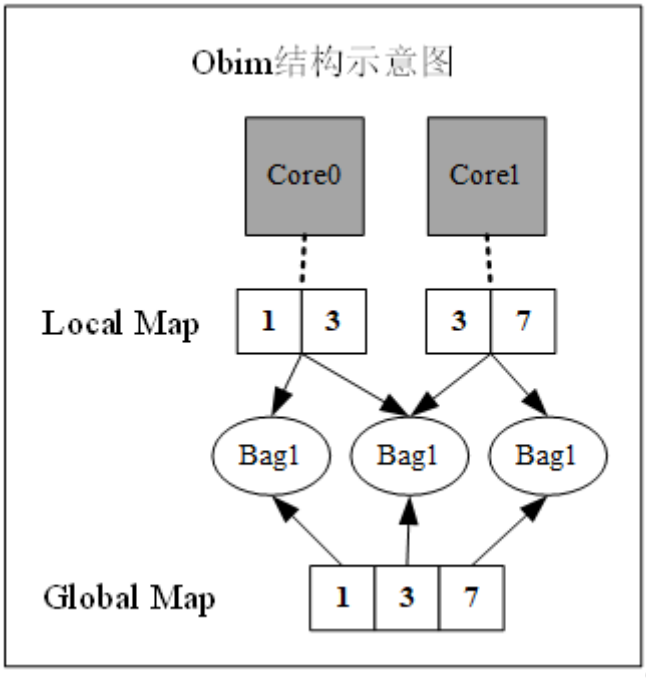
满，就会从package中插入任务到其他chunk，如果chunk为空，就会从package中查找其他chunk的任务，如果package也为空，就去另一个package查找。



图：DepGraph并发包结构示意图

3. 优先级调度

DepGraph提出了obim软优先级调度器，将机器拓扑感知任务调度策略与任务具有优先级的情况结合。obim包含一系列并发包，每个包中放置同等优先级的任务，可以任意顺序并发执行，不同包之间的任务按优先级执行。如下图所示，图中包含3个包，分别存放任务1、3、7，全局维护一个操作日志记录全局包活跃；每个线程维护一个本地包映射，缓存全局包结构，记录日志点。当线程需要插入和获取任务时，首先到本地查找特定优先级的包，若无，从全局获取最新结构，若仍无，则创建新优先级的包，同步全局和本地包和日志。使得所有线程共享当前处理任务的优先级，提高任务获取效率，保证任务的执行顺序。



图：DepGraph obim优先级调度器结构示意图

4. 可扩展数据结构库

DepGraph还提供了一套扩展库和运行时系统来支持其任务调度策略，包括内存分配机制、拓扑感知的同步机制和代码优化机制。用户能够通过DepGraph灵活丰富的编程接口尽可能简单地实现复杂的功能。DepGraph作为一个一般性的图计算编程框架除了自身提供的基于任务调度策略的执行模式之外，还可以支持其他图计算系统的API，例如GraphLab的点程序执行模式，PowerGraph的GAS执行模式，GraphChi用外存计算的手段，Ligra的push和pull切换策略，以及将核外计算和push/pull结合执行的模式。

GPU代码原理

> 参考论文：《Scalable GPU Graph Traversal》

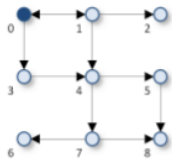
1. GPU图计算背景

现代处理器架构提供了越来越多的并行性，以便在保持能源效率的同时提供更高的吞吐量。现代 GPU 处于这一趋势的前沿，提供数以万计的数据并行线程。图计算具有高并发的特点，以顶点为中心和以边为中心的图计算编程模型都隐藏着大量的并行语义，因此都能够有效地通过GPU来进行并行加速。同时，图计算属于数据密集型应用，经常需要处理数十GB的数据，所以GPU的高带宽也是一个明显的优势。尽管有这些优势，但是用GPU来加速大规模图计算仍然面临着诸多挑战：

- 图计算是典型的不规则应用，它的内存访问模式具有间接性、数据依赖等特点，所以无法充分利用GPU的高带宽以及高并行性。

- GPU内存大小的限制，使得经常需要将数据从主存中向GPU内存中移动，而这会导致额外的开销，同时使GPU扩展到大图上面临着内存空间不足等问题。
- 现实中的图大多是幂律分布（power-law）的，各个顶点的度数分布不均匀，所以在图算法中，负载均衡问题也需要格外注意。
- 由于GPU的硬件架构特点，它并不适合处理图计算中的条件分支语义（if-else等），所以有时无法充分利用GPU的SIMT（Single Instruction Multiple Threads）带来的高并行性，所以分支语义会极大降低GPU性能。

图遍历算法一直是图计算中的一个重要算法，同时也是很多其他图算法的基础。Breath-first search（BFS）是图遍历的经典算法，也是一种典型的内存访问和工作负载既不均衡又依赖于数据的并行算法。Merrill等人提出了一个基于GPU的并行BFS算法。并行的BFS大体上分为两步：1) 遍历顶点的边，将正在遍历的顶点边界（vertex-frontier）扩展为边边界（edge-frontier）；2）通过状态检查和过滤等方法，将已经访问过以及重复添加的顶点剔除，收缩成新的顶点边界，供下次迭代使用。



C: 1,3 | 0,2,4 | 4 | 5,7 | 8 | 6,8

R: 0 | 2 | 5 | 5 | 6 | 8 | 9 | 9 | 11 | 11

Traversal from source vertex v_0		
BFS Iteration	**Vertex frontier**	**Edge frontier**
1	{0}	{1,3}
2	{1,3}	{0,2,4,4}
3	{2,4}	{5,7}
4	{5,7}	{6,8,8}
5	{6,8}	{}

Fig. 1. Example sparse graph, corresponding CSR representation, and frontier evolution for a BFS beginning at source vertex v_0.

BFS 工作负载主要由两个部分组成：与顶点前沿处理相关的 O(n) 工作，以及与边缘前沿处理相关的 O(m) 工作。其中边缘前沿处理占主导地位，因此我们将注意力集中在边缘前沿操作的两个基本方面：邻居收集和状态查找。

2. 邻居顶点收集（neighbor-gathering）

为解决在GPU并行计算过程中的负载不均衡问题，他们探究了使用不同的任务处理粒度对负载均衡的影响。

图： 顺序收集

顺序收集：顺序收集的方法为每个线程分配一个顶点，各个线程获取其处理顶点对应的邻居。如图4-4-5所示，线程t_0、t_1、t_2、t_3所处理的顶点分别有2、1、0、3个邻居。显然，在各个顶点的度分布不均匀的情况下，这种方法会导致同一个Warp中不同线程的负载不均衡。

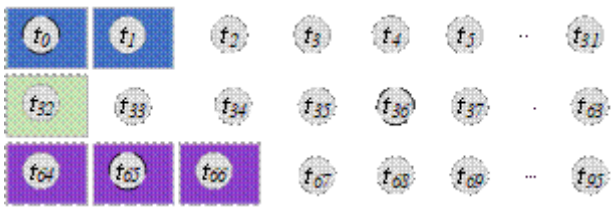
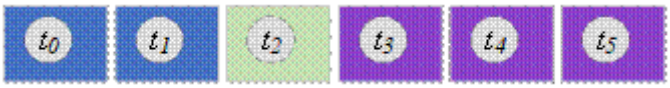


图: 基于warp的粗粒度收集

基于warp的粗粒度收集：基于warp的粗粒度收集方法允许线程对其所在warp的控制权进行竞争，竞争成功的线程可以使用整个warp的资源来处理其所分配的顶点，处理完以后其他线程继续竞争。这种方法一定程度上可以减少负载不均衡问题，但是很多时候一个顶点的邻居数目是少于一个warp中的线程数目的，这样就会产生性能未充分利用的问题。同时，有时候不同warp中线程被分配顶点的总度数相差较

大，又会产生warp之间的负载不均衡。



图：基于扫描的细粒度收集

基于扫描的细粒度收集：基于扫描的细粒度收集方法允许一个CTA中的线程共享一个列索引偏移量数组，并生成相应的共享收集向量（shared gather vector）。这个向量的内容对应于分配给该CTA的顶点的邻接表，然后利用整个CTA对于相应的顶点进行处理：每个线程从共享向量中取出一条边进行处理。这样一来，线程间的负载不均衡就不会被昂贵的全局内存访问放大。但是，这种方法也可能出现共享数组无法充分利用的问题，例如某个顶点的邻居数据过大，几乎占满了整个共享向量，其他线程对应的邻接表就无法被处理。

Scan + Warp + CTA的混合收集方法：将基于扫描的细粒度以及基于CTA和warp的粗粒度的任务分配策略结合起来。基于CTA的策略与warp类似，只是线程将会争夺整个CTA的控制权。首先将顶点分配个线程，那些顶点的邻接表大于CTA（中的线程个数）的线程，会竞争CTA，使用整个CTA来处理其顶点的邻接表；那些邻接表比CTA小，但是比warp大的，会竞争warp；而对于那些比warp还要小的邻接表，则使用scan，将线程对应的邻接表整合到共享内存中来共同处理。这种混合策略能有效地克服单独一种方法的不足，从而在多种图算法中都达到较好的负载均衡效果。

3. 状态查找（status-lookup）

在产生下次迭代所需的顶点集合时，为了避免之前已经访问过的顶点重复访问，需要一个状态码来标识每个顶点是否已经被访问。论文中采取位掩码的方法，将状态码的大小从每个顶点32bits减小到了1bits，有效提高了缓存效率，减少内存访问量，从而提升性能。

4. 并发发现（concurrent discovery）

在并行遍历边时，有可能不同顶点的边连接到同一个目的顶点，因为GPU并行机制的原因，所有访问同一个顶点状态的线程都会得到相同的值，即有某个线程已经对该顶点进行过处理，但是因为是在同一次迭代中，其他线程访问该顶点的状态时仍然得到未访问的结果，从而导致对同一个顶点的多次重复计算，浪费资源和性能。为了解决这种false-negative问题，文章中提出了两种方案：

1. Warp筛选（warp culling）。Warp中的线程利用哈希函数将顶点与共享内存中的位置关联起来，每一个顶点对应一个位置。当一个线程要检查其邻居顶点时，首先尝试将其线程id放到对应的共享内存中，如果该位置已经有其他线程id，则代表该顶点已经被其他线程处理，可以安全的剔除出去。
2. 历史筛选（history culling）。这种方法是warp筛选的一种补充。该方法在共享内存中维护最近检查的顶点标识符的缓冲，如果一个线程发现其所处理顶点的邻居之前已经被记录时，那么它就认为这个邻居已经被处理过了，可以剔除。
   通过同时使用这两种方法，能够将冗余计算降低到5%，大大提高算法性能。

5. 多GPU并行：

该算法还支持多个GPU的系统。通过将图分割为大小相等的、互不相交的子集，在主程序的协调下，分别对不同GPU调用相应处理函数，并且设置跨GPU的屏障（barrier），来保证GPU之间的同步。

6. 总结

该算法利用混合的任务负载方式，细粒度地在线程之间分配任务，达到较好的负载均衡效果。利用前缀和来代替原子的读-修改-写（read-modify-write）操作，来执行多线程并行写入。并通过一些其他的优化方法如位掩码等来实现基于GPU的高效并行BFS算法。