Note of “Graphene”

论文 Paper

> (*Cheng Zhao:*) 这篇文章的切入点为目前的图计算系统如果把图全部读入内存中计算太浪费，只在硬盘上又会造成很大的IO开销，并且很多图计算的算法不需要对整张图进行计算，相反可能只需要计算其一小部分。所以是否可以在计算前找准要计算的点，将其读入内存中进行计算。
> 创新点：
>
> - 整篇文章还是着重在IO开销上做优化，比如使用小的IO块（512B而不是4KB）使得读取粒度更细，使用Bitmap管理IO块优化对节点数据的操作等；
> - 图划分算法是在GridGraph上做的改进，针对GridGraph对边划分不均的问题，保证在Grid的过程中使得每个分片中边的个数相同；
> - 最后考虑到TLB丢失严重的问题，此系统支持HugePage（介绍见下文）。

1. Motivation & Contribution

Motivation

- In-memory processing: 开销大，扩展困难；
- External memory processing: 编程困难，IO管理复杂，IO是整个系统的瓶颈。

Contribution

- IO (request) centric graph processing
- Bitmap based, asynchronous IO
- Direct hugepage support
- Balanced data and workload partition

IO Request Centric Graph Processing

IoIterator API

Type	Name	Return Value	Description
System provided	`Iterator->Next()`	`io_block_t`	获取内存中下一个数据块的指针
System provided	`Iterator->HasMore()`	`bool`	从IO中检测是否有其他available节点
System	`Iterator->Current()`	`vertex`	获取下一个available节点

provided			
System provided	`Iterator->GetNeighbors(vertex v)`	`vertex array`	获取节点`v`的相邻节点
User defined	`IsActive(vertex v)`	`bool`	检测节点`v`是否为available
User defined	`Compute(vertex v)`	`void`	具体定义计算流程

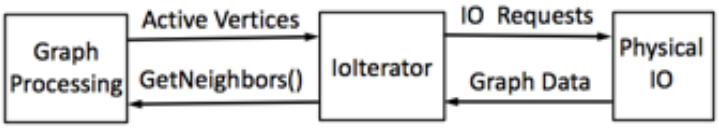


Figure 2: IoIterator programming model.

Bitmap Based, Asynchronous IO

Bitmap-Based IO Management

512B IO blocks

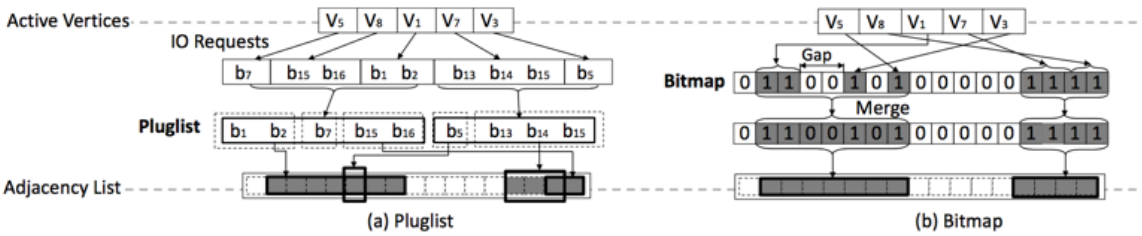


Figure 5: Pluglist vs. bitmap IO management, (a) Pluglist where sorting and merging are limited to IO requests in the pluglist. (b) Bitmap where sorting and merging are applied to all IO requests.

Pre-difined maximum gap
Merge 小于这个gap的文件块
maximum gap is set to 16 blocks (8KB)

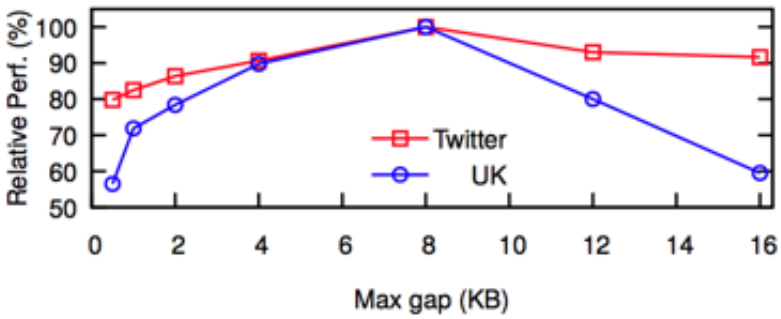
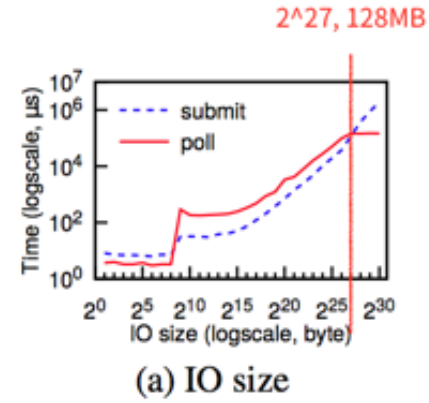


Figure 6: Graphene BFS Performance of maximum gap.

Asynchronous IO

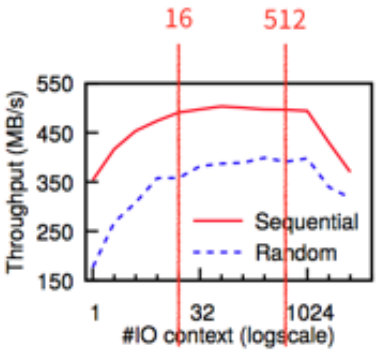
典型的异步IO分为两步：1. 将IO请求提交到IO context；2. 轮询这个context.



(a) IO size

当请求大小超过128MB时，提交请求的时间超过轮询时间，从而请求会被block住。本文设置默认请求上届为16KB。

异步IO中，每个IO context顺序地读取IO请求，Graphene使用多个IO context处理并发请求问题。例如，当一个县城处理一个IO context发出的请求时，另外一个IO context可以被用来处理相同的SSD发出的其他请求。



(b) IO context

过多的IO context会降低磁盘吞吐量，默认为512个context。

> 个人理解IO context就是专门为IO服务的线程，这里默认创建512个线程为整个系统的IO请求服务。

Balancing Data and Workload

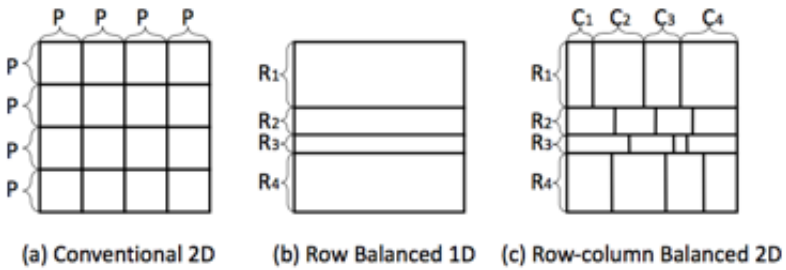


Figure 8: Graphene Balanced 2D partition.

相比较GridGraph，Graphene的目标是使每一个分片的边数相等，于是Graphene首先对行进行等分（图b），然后在每行内对列进行等分（图 c）。

在内存中维护每个分片中点的信息（metadata）。

Balancing IO and Processing

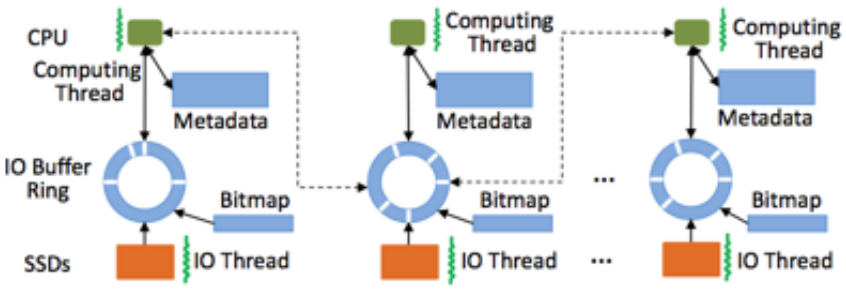


Figure 10: Graphene scheduling management.

Graphene 使用了work stealing技术：一旦一个计算线程完成了其数据的IO请求，即检查IO buffer中其他计算线程是否完成；只要有未完成任务的线程，就去帮助该线程。

> 文中没有细讲IO Buffer Ring的设计作用，这块不是特别明白。

Hugepage Support

随着计算需求规模的不断增大，应用程序对内存的需求也越来越大。为了实现虚拟内存管理机制，操作系统对内存实行分页管理。自内存“分页机制”提出之始，内存页面的默认大小便被设置为 4096 字节（4KB），虽然原则上内存页面大小是可配置的，但绝大多数的操作系统实现中仍然采用默认的 4KB 页面。4KB 大小的页面在“分页机制”提出的时候是合理的，因为当时的内存大小不过几十兆字节，然而当物理内存容量增长到几 G 甚至几十 G 的时候，操作系统仍然以 4KB 大小为页面的基本单位，是否依然合理呢？

在 Linux 操作系统上运行内存需求量较大的应用程序时，由于其采用的默认页面大小为 4KB，因而将会产生较多 TLB Miss 和缺页中断，从而大大影响应用程序的性能。当操作系统以 2MB 甚至更大作为分页的单位时，将会大大减少 TLB Miss 和缺页中断的数量，显著提高应用程序的性能。这也正是 Linux 内核引入大页面支持的直接原因。好处是很明显的，假设应用程序需要 2MB 的内存，如果操作系统以 4KB 作为分页的单位，则需要 512 个页面，进而在 TLB 中需要 512 个表项，同时也需要 512 个页表项，操作系统需要经历至少 512 次 TLB Miss 和 512 次缺页中断才能将 2MB 应用程序空间全部映射到物理内存；然而，当操作系统采用 2MB 作为分页的基本单位时，只需要一次 TLB Miss 和一次缺页中断，就可以为 2MB 的应用程序空间建立虚实映射，并在运行过程中无需再经历 TLB Miss 和缺页中断（假设未发生 TLB 项替换和 Swap）。

为了能以最小的代价实现大页面支持，Linux 操作系统采用了基于 hugetlbfs 特殊文件系统 2M 字节大页面支持。这种采用特殊文件系统形式支持大页面的方式，使得应用程序可以根据需要灵活地选择虚存页面大小，而不会被强制使用 2MB 大页面。本文将针对 hugetlb 大页面的应用和内核实现两个方面进行简单的介绍，以期起到抛砖引玉的作用。

(https://www.ibm.com/developerworks/cn/linux/l-cn-hugetlb/index.html)