GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning

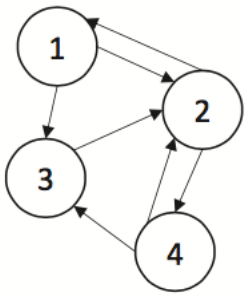
总结经验

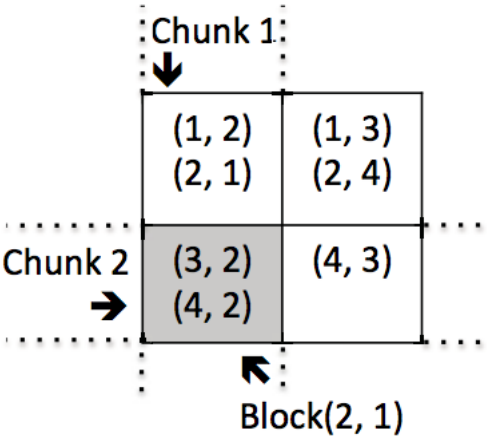
GraphChi :每次预处理和对滑动窗口更新的时候都会进行 sort 操作, 过于耗时；由于滑动窗口的读写过程，产生了大量的随机读写操作，大大浪费了外存的带宽。

X-Stream：首先，scatter 过程产生的 update 缓存过大，其大小大概等同于边的规模，造成了额外的硬盘读写负担；其次，其是一个同步模型，虽然每一步内都可以多线程并行运行，但每一步最后存在一个同步的过程，性能因此降低；最后不支持有选择的更新，这对于图计算来说是一个很致命的漏洞，因为图计算算法最后是会收敛的，很多节点是没有必要更新的，因此 X-Stream 的运算量会大大增加，收敛速度会下降。

单机图计算的核心问题在于如何减少随机读写，尽量使用序列读，这样能最大限度的利用外存的带宽；其次尽量减少写入文件的大小，因为写的性能远远小于读的性能。这也是 GridGraph 的核心所在。

创新点 1:Grid 的存储方式





如上图所示，将节点分为 P 个 Chunk（图中分为了 2 个 Chunk：{1,2}、{3,4}），画一个 P*P 的二维表，其中每一行代表 source 在该 Chunk 的边的集合，每一列代表 destination 在该 Chunk 的边的集合，这样就会将边划分到 P*P 的格子内，每一

个格子作为一个单独的文件进行保存，**所以 GridGraph 是需要预处理的。**与 GraphChi 不同的是 GridGraph 不需要进行排序操作，所以预处理的时间会比 GraphChi 快。

这样分块存储会遇到一个问题就是就是每一块的大小是不同的。经过统计，其大小和数量呈现一个指数分布的关系，即大量的小块和少量的大块，小块会浪费序列读的带宽，所以这里做了一个优化，即在存储的时候将多个小块放在一起存储。

关于 P 的取值，由于为了实现精细的有选择的更新

最后系统会记录一些全局信息(metadata)，比如分块的大小，合并存储的情况，边的总数和点的总数等等，其大小很小，所以可以一直放在内存里。

创新点 2:灵活的函数接口

GridGraph 定义了三个编程接口：

F 接口 :输入为 Vertex，输出为 boolean，用来判断一个节点是否还是 active 的。

F_v接口：运行点上的计算函数，伪代码如下图所示：

Algorithm 1 Vertex Streaming Interface

```
function StreamVertices(F_v, F)
    Sum = 0
    for each vertex do
        if F(vertex) then
            Sum += F_v(edge)
        end if
    end for
    return Sum
end function
```

F_e接口：运行边上的计算函数，伪代码如下图所示：

Algorithm 2 Edge Streaming Interface

```
function StreamEdges(F_e, F)
    Sum = 0
    for each active block do          ▷ block with active edges
        for each edge ∈ block do
            if F(edge.source) then
                Sum += F_e(edge)
            end if
        end for
    end for
    return Sum
end function
```

这两个函数借口返回的都是全局统计量，用于判断算法结束的条件。文中给了一个 PageRank 的例子，伪代码如下图所示：

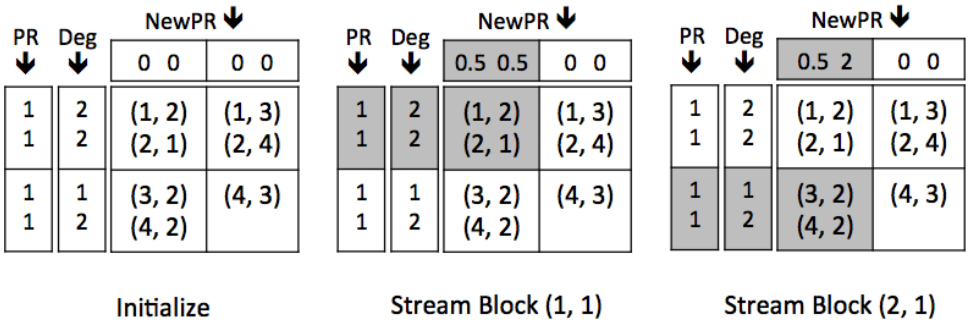
Algorithm 3 PageRank

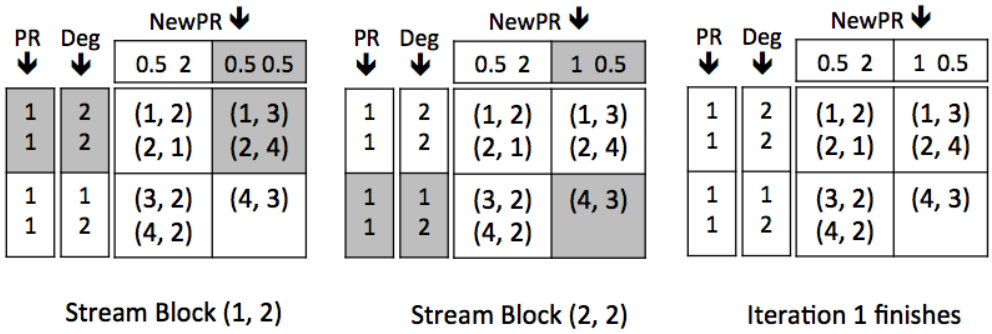
```
function CONTRIBUTE(e)
    Accum(&NewPR[e.dest], PR[e.source] / Deg[e.source])
end function
function COMPUTE(v)
    NewPR[v] = 1 − d + d × NewPR[v]
    return |NewPR[v] − PR[v]|
end function
d = 0.85
PR = {1, ..., 1}
Converged = 0
while ¬Converged do
    NewPR = {0, ..., 0}
    StreamEdges(Contribute)
    Diff = StreamVertices(Compute)
    Swap(PR, NewPR)
    Converged = Diff / V ≤ Threshold
end while
```

我们可以发现，**GridGraph 既可以在边上进行计算，又可以在点上进行计算**，这个可以根据算法的需求用户自己调节使用，十分的灵活方便。

创新点 3:Dual Sliding Windows

Gird 形式的存储格式，将大量的减少随机写的次数。根据不同的图算法，GridGraph 可以进行按行读取或按列读取，取决于图算法对 source 进行更改还是对 destination 进行更改。在一列（一行）的更改的过程中，由于节点是存在内存中的，所以可以很快的进行多次的修改，当一列（一行）读取结束之后，把最后修改的结果值直接写入外存，这样就大大减少了对外存的随机写。下图是一个示例：





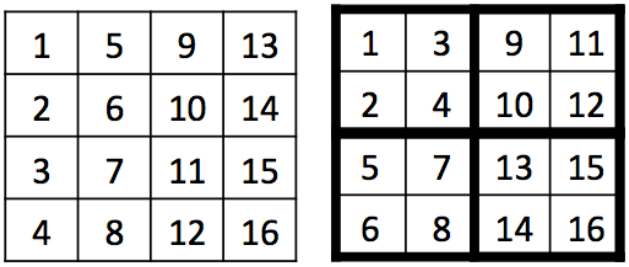
创新点 4:2-Level Hierarchical Partitioning

I/O 总量为：

$$E + P * V + 2 * V$$

其中 E 为边的个数，V 为点的个数，P 是分片的数量，我们可以看出来当 P 越小的时候，I/O 的负载越小，但是对选择性更新要求 P 越大越好，在这里就出现了冲突。

传统的做法是测试一个折衷的方案，但是在本文中找到了一个两全其美的方法——2-Level Hierarchical Partitioning。具体形式如下图所示：



逻辑上按照 4*4 的方式来使用，物理上按照右侧 2*2 的方式来进行存储和读取。这样既满足了选择性更新的细粒度要求，又减少了 I/O 总量。

测试结果

	i2.xlarge (SSD)				d2.xlarge (HDD)			
	BFS	**WCC**	**SpMV**	**PageR.**	**BFS**	**WCC**	**SpMV**	**PageR.**
LiveJournal								
GraphChi	22.81	17.60	10.12	53.97	21.22	14.93	10.69	45.97
X-Stream	6.54	14.65	6.63	18.22	6.29	13.47	6.10	18.45
GridGraph	2.97	4.39	2.21	12.86	3.36	4.67	2.30	14.21
Twitter								
GraphChi	437.7	469.8	273.1	1263	443.3	406.1	220.7	1064
X-Stream	435.9	1199	143.9	1779	408.8	1089	128.3	1634
GridGraph	204.8	286.5	50.13	538.1	196.3	276.3	42.33	482.1
UK								
GraphChi	2768	1779	412.3	2083	3203	1709	401.2	2191
X-Stream	8081	12057	383.7	4374	7301	11066	319.4	4015
GridGraph	1843	1709	116.8	1347	1730	1609	97.38	1359
Yahoo								
GraphChi	-	114162	2676	13076	-	106735	3110	18361
X-Stream	-	-	1076	9957	-	-	1007	10575
GridGraph	16815	3602	263.1	4719	30178	4077	277.6	5118

测试分为了 SSD 和 HDD 两个部分，使用了 BFS、WCC、SpMV、PageR 四个算法，比较了 GraphChi、X-Stream 和 GridGraph 三个系统的性能。可以看出 GridGraph 全面占优。

	C (S)	G (S)	C (H)	G (H) P	G (H) M	G (H) A
LiveJournal	14.73	1.99	13.06	1.64	1.02	2.66
Twitter	516.3	56.59	425.9	76.03	117.9	193.9
UK	1297	153.8	1084	167.6	329.7	497.3
Yahoo	2702	277.4	2913	352.5	2235.6	2588.1

这是 GridGraph 与 GraphChi 的预处理对比，C = GraphChi, G = GridGraph; S = SSD, H = HDD; P = time for partitioning phase, M = time for merging phase, A = overall time。可以看出预处理时间也是 GridGraph 更快一些。