分布式系统期末大作业

分布式并发图系统

数据科学与计算机学院　17大数据与人工智能

17341015　陈鸿峥

由于这一年刚好在做并行分布式大数据系统的相关工作，因此本次大作业我选择了自选题目，主要内容是**并发图系统**。去年并行课上我已经实现了单机共享内存版本，而此次则将其扩展至多机分布式系统。本文成果已以第一作者身份投稿至CCF-A类会议Annual Technical Conference (ATC'20)，目前在审，详情可见补充材料。

两个版本的并发图系统都已开源，也可以在Github上查看我的提交记录。系统核心代码采用C++进行编写，并使用OpenMP、Cilk Plus、MPI等多种并行编程框架，同时涉及模板元编程等C++11特性。单机版本[1]总代码修改量超过1万行，分布式版本[2]由于在单机版本上进行扩展，因此大约在3千行代码的修改量。

本实验报告将简要介绍本项目的背景与方法，并着重说明与本门课程的联系。本项目与课程内容章节的对照可参见附录A。

一、 简介

图结构可以对大量现实世界中的复杂关系进行建模，因此近年来在社交网络、机器学习、科学计算等领域被广泛应用。为了高效地处理这些超大规模的图结构，人们提出了很多图计算系统，从最早年谷歌由MapReduce延伸出来的Pregel [1]，到之后在Spark上搭建的的GraphX [2]，再到现在在Facebook内部应用的Giraph [3]。图计算系统经历了几个发展阶段，也一直是并行分布式计算领域一个极具挑战性的研究方向。

而随着越来越多应用在同一个大图上运行，并发图任务的需求也越来越大。第一方面来源于**客户端-服务器的场景**。类比传统的关系型数据库，多个用户可以同时对其发起请求。对于图计算系统来说也是类似，很多可能在同一时间段内有多个用户在上面进行计算。可能有些用户会关心一个大的社交网络内的人与人之间的n度关系（即子图匹配），而另一些用户关心哪些人可以构成一个具有一定共性的群体（即社群检测）。这时如果这些用户同时发起请求，则要求系统快速地对这些查询作出应答。

而第二方面则来源于**对图应用的观察**。我们发现事实上有大量的图应用都是由基本的图算法构成，而在该应用下同时执行这些算法就显得至关重要。举一个最近的例子，图嵌入(graph embedding)是目前自然语言处理、推荐系统等方向一个非常重要的技术，其中采用的DeepWalk

[1]Krill单机共享内存版本：https://github.com/chhzh123/Krill

[2]Krill分布式版本：https://github.com/chhzh123/GeminiGraph

[4]、Node2Vec [5]等随机游走的方法实际上就是在图上同时做多个n度的遍历，如何快速地对这些路径进行采样就成为了至关重要的问题。这里的每一个遍历都可以看作是一个图计算任务(task)，而这些图计算任务构成了一个完整的工作(job)，即随机游走。 SOSP'19上已经有相关工作KnightKing [6]，但他们仅仅对随机游走本身进行并行优化，而我们则着重于更加通用的并发图计算任务。

目前的图计算系统大多都是为单一图计算任务而设计 [1, 3, 7–10]，它们在任务内的并行已经发挥得非常出色，但对于这一最大粒度的并行——任务间并行——其实做得并不是很好，而主要原因来源于它们忽略了大量的共享关系，如共享的图结构、共享的访问顶点及共享的遍历模式等。

因此，在本项目中，我们提出了Krill这一并发图计算系统。通过最大程度挖掘任务与任务之间的相互关系，利用图核融合(graph kernel fusion)技术，减少冗余数据访问；同时提出了快速边界筛和提前任务筛来加速每轮迭代中的遍历。 Krill的单机共享内存版本基于Ligra [7]搭建，而分布式的版本则基于陈文光老师的Gemini [11]进行搭建。最终，Krill比基准Ligra系统减少了14倍的内存访问开销，运行时间比其快8倍；分布式版本比当前最好的并发图计算系统Seraph [12]快了4.25倍。

二、 背景与动机

目前的图计算系统基本采用以下两种引擎，一种是Push的，即从源结点到汇结点进行遍历；另一种是Pull的，即从汇结点到源节点进行遍历。两种方式各有优劣，而Krill采用了混合引擎以最大化这两者的优势。

Push Engine

```
parallel_for (vSrc : vertices) {
  if (!vSrc in frontier)
    continue;
  for (vDst : vSrc.outngh) {
    if (cond(vDst))
      atomicUpdate(vDst);
  }
}
```

Pull Engine

```
parallel_for (vDst : vertices) {
  if (!cond(vDst))
    continue;
  for (vSrc : vDst.inngh) {
    if (vSrc in frontier)
      update(vDst);
  }
}
```

在数据结构表示上，图数据可以被分割成两个部分，一部分是图结构，即图的邻接关系；另一部分为图性质，即图上每一顶点的专有数据。通常图结构采用稀疏压缩行(Compressed Sparse Row, CSR)格式存储，而图性质则采用一般的数组进行存储，如图1所示。

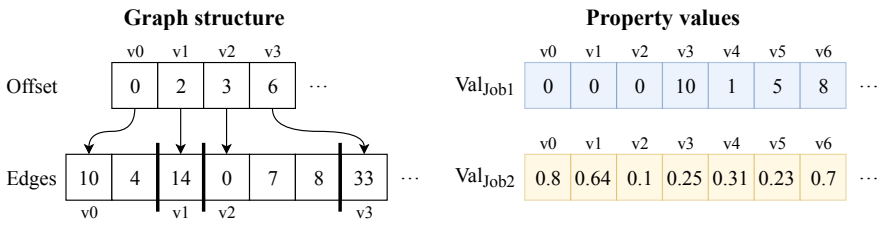


图 1: 图结构与图性质

近几年已经有北大代亚非老师组的Seraph [12]和华中科技大学金海老师组的CGraph [13]两个并发图系统被研发，但是仍存在大量的性能提升空间，包括以下三点：

- **不同任务之间的相互关系**：由于Seraph和CGraph将每一个任务都看作是独立的个体，因此丧失了对它们统一表示及统一优化的机会。Krill则提出了图核融合技术，将所有的任务看作一个整体进行处理，进而可以充分挖掘任务之间的相似性，以减少不必要的访问计算开销。
- **大量的内存冗余访问**：对于Seraph和CGraph来说，由于每个任务都是独立的个体，因此在一轮迭代中，总的访问图结构的时间为$O(c|E|)$，其中c为任务数量，$|E|$为图中边数。事实上，如果我们将这些任务看成一个整体，统一去访问图结构，那么访问开销将会被降至$O(|E|)$，这也是图核融合的核心思想。
- **未利用的大量内存带宽**：随着近年来体系结构和图处理技术的发展，IO开销被降低，伴随着就是大量的内存访问带宽被闲置。我们通过实验数据得到，单一任务下内存带宽的利用率最大仅到80%，且大多数情况远低于这个水平。但如果简单地将这些任务进行并行，那么必然会导致带宽互相侵占，进而各个任务的性能降低。因此需要有更加好的任务管理方案，来充分利用这些带宽资源。

三、 核心技术

这一节将简要叙述Krill中所采用的核心技术，详情细节请见原始论文。

1. 图核融合

所谓图核(graph kernel)，即一个计算任务，可以类比于GPU的计算核。考虑图$G(V,E)$，则图核是一个$f:V\times V\mapsto\mathbb{R}$的映射，可以标识一个图任务。当多个图任务$f_1,\ldots,f_k$在一起考虑时，我们希望找到这样的映射$g$使得

$$g=f_1\times f_2\times\cdots\times f_k \tag{1}$$

$$g:V\times V\mapsto\mathbb{R} \tag{2}$$

此即图核融合的最核心思想：用一个计算核代替所有计算核，其计算效果跟所有计算核同时计

算的效果等效，这样便能减少不必要的内存访问开销。

具体实现上则是由上文的观察，将遍历任务的循环体内移，使得原来$O(c|E|)$次的图结构访问，下降至$O(|E|)$次。对应的Push和Pull引擎可见算法1和算法2。

Algorithm 1 结合图核融合的Push引擎

```
for each vSrc in vertices do
    for vDst in vSrc.outngh do
        for each job in jobList do
            condPushUpdate(job,vSrc,vDst)
```

Algorithm 2 结合图核融合的Pull引擎

```
for each vDst in vertices do
    for vSrc in vDst.inngh do
        for each job in jobList do
            condPullUpdate(job,vSrc,vDst)
```

其中的`condPushUpdate`和`condPullUpdate`则是顶点更新规则，包括判断源节点是否是活跃结点、目标结点是否满足更新前提、目标结点是否被更新函数更新。

2. 快速边界筛

可以看到，如果仅采用前文所述的简单图核融合引擎，那么依然存在大量的冗余遍历，一方面是源节点的访问冗余，另一方面则是汇结点的访问冗余。

考虑源结点的情况，即Push引擎。其实在单任务图系统上Push的方式已经做得非常成熟，常常利用边界集(frontier)优化的方法来加速遍历 [7, 11]，因此Krill同样采用边界集来跟踪每一轮迭代中的活跃结点。

如果设k个任务的边界集分别为$\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_k$，则

$$\mathcal{D}_{\cup} = \bigcup_{i=1}^{k} \mathcal{D}_i = \mathcal{D}_1 \cup \mathcal{D}_2 \cdots \cup \mathcal{D}_k \tag{3}$$

必然是包括这些边界集中所有顶点的最小集合，这一点利用简单的集合论知识可以证明。

那么对于每一个任务i，只需维护自己的边界集$\mathcal{D}_i$；对于Krill系统，则只需在此基础上维护一个公共边界集$\mathcal{D}_{\cup}$即可做到活跃结点的快速筛选，核心伪代码如算法3所示。

Algorithm 3 添加了快速边界筛的Push引擎

```
for each vSrc in currCommonFrontier do
    for vDst in vSrc.outngh do
        for each job in jobList do
            job.condPushUpdateFFF
                (CommonFrontier,vSrc,vDst)
```

其中的`condPushUpdateFFF`与算法1的类似，只是添加了更新公共边界集的步骤。

3. 提前任务筛

对于Pull引擎来说，由于外层循环先遍历汇结点，内层循环再遍历源节点，因此无法充分利用边界集优化技术，需要考虑其他方式来减少对结点的冗余访问。

Krill选择在进入内层循环之前，提前对需要遍历源节点的任务给筛选出来，然后在内层循环才对这些任务进行遍历，核心伪代码如算法4所示。

Algorithm 4 添加了提前任务筛的Pull引擎

```
1: for each vDst in vertices do
2:     availJob.initialize()
3:     for each job in jobList do
4:         if job.cond(vDst) then
5:             availJob.append(job)
6:     if (availJob.empty()) continue
7:     for vSrc in vDst.inngh do
8:         for each job in availJob do
9:             job.condPullUpdateAJF(vSrc,vDst)
10:            if (!job.cond(vDst)) then
11:                availJob.del(job)
```

其中的`condPullUpdateAJF`与算法2的类似，同样添加了更新公共边界集的流程。

四、 系统实现

Krill的执行流程如图2所示。Krill主要包括了一个提取器(fetcher)、一个调度器(scheduler)和一个执行器(executor)。当有新的任务到来时，提取器将会抓取任务并将其添加入任务池中。在每一轮迭代开始时，调度器会将正在进行的任务和新的任务重新进行融合，然后将融合后的任务交给执行器去执行。

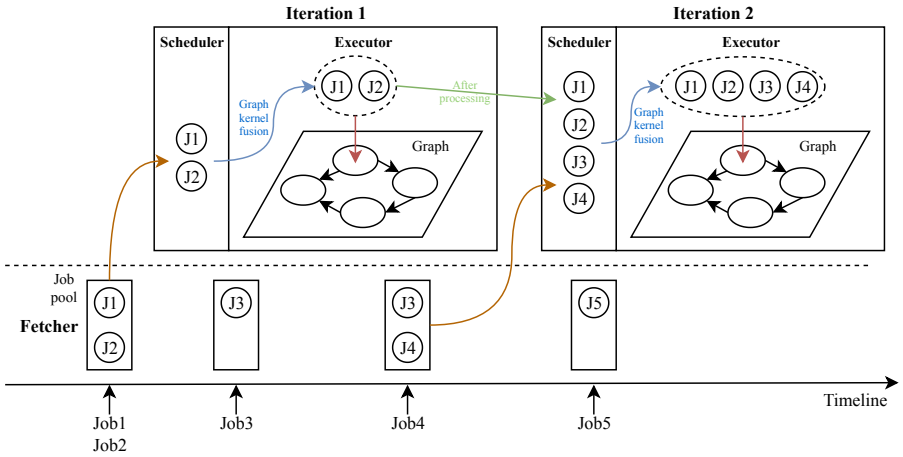


图 2: Krill的执行流程

我们采用OOP对任务进行封装，为每一个任务提供一个基类(base class)，同时提供一个容器类Kernels，可以将所有的任务封装在里面。用户只需在每个任务内定义基本的cond、update和updateAtomic函数，并将这些任务打包在Kernels类中，通过setKernels函数即可将任务传递给Krill系统进行执行。用户API样例如下：

```
void setKernels(graph& G, Kernels& K)
{
  BFS* bfs = new BFS(G.n); // pass in # of vertices
  SSSP* sssp = new SSSP(G.n);
  K.appendJob({bfs,sssp});
}
```

五、 实验

这里主要讲述Krill分布式的部分，也是本次项目的重点，其单机共享内存的实验方式与结果可以见原始论文。

我选择了陈文光老师组的Gemini [11]系统进行扩展实现。 Gemini是**分布式图计算系统**的典范，但其依然是单任务图计算系统，因此我将其扩展为支持并发图任务，同时在上面实现我的Krill系统及现有的并发图Seraph [12]系统。

1. Seraph实现

要实现并发图处理，很关键一点在于**将图结构与图性质解耦**(decouple)。原本的Gemini将关于图的所有数据都封装在一个类Graph中，包括分布式节点的通信缓存(buffer)也封装在其中，这直接导致一个图只能对应一个任务，而无法在同一个图上执行多个任务。

因此使Gemini能够支持并发图处理的第一步操作就是解耦，将图结构封装在原始的Graph类中，而图性质及其他相关数据则新建其他的类进行存储，同时通过传递参数的方式实现调用。解耦后的计算模式如图3所示，即Seraph [12]提出来的Graph-State-Exchange data (GES)计算模式。采用这种方法即可以方便地对图结构和图性质分开处理，用户只需关心图性质的处理逻辑，而关于图结构的遍历则全部交由图计算系统完成。

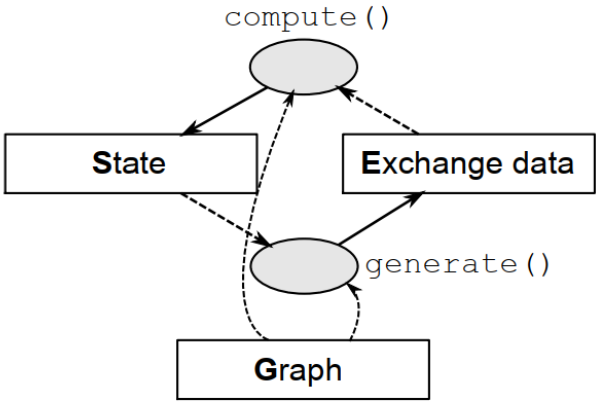


图 3: Graph-State-Exchange data (GES)计算模式

解耦完图结构和图性质后，就可以在同一个图上进行多个任务的操作。为实现Seraph，采用C++11的线程库<thread>，为每个任务分配一个线程，然后共享相同的图结构，而图性质则依不同任务而不同，它们通信缓存也相互不同，以标号的形式避免冲突。

注意这里图结构的读入与划分(partition)是由Gemini完成的，它保证了在每一个分布式节点上都存在一部分的图划分。但由于每一个节点都没有全部的图数据，因此在具体实现算法时需要确保任务的图性质数据都正常在节点之间传递，这一步需要在算法中用emit强制声明。

2. Krill实现

Gemini系统提供的用户API相对比较底层，核心为process_edges函数，其定义如下

```
template<typename R, typename M>
R process_edges(
    std::function<void(VertexId)> sparse_signal,
    std::function<R(VertexId, M, VertexAdjList<EdgeData>)> sparse_slot,
    std::function<void(VertexId, VertexAdjList<EdgeData>)> dense_signal,
    std::function<R(VertexId, M)> dense_slot,
    Bitmap * active
    )
```

其中四个用户自定义函数对应着其Signal-Slot的编程模型，如图4所示，其声明了每个图顶点的更新方式，以及在分布式环境下如何在Master和Mirror之间传递数据。

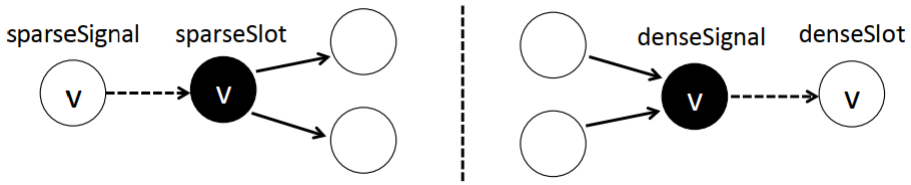


图 4: Sparse-dense signal-slot模型

为实现Krill的几个技术点，我们其实只需更改用户自定义的四个函数即可。

- 通过将任务遍历的循环移至最里层，可以实现**图核融合**。
- 通过新建`common_active_in/out`数组，实时维护公共边界集，实现**快速边界筛**。
- 通过在`dense_signal`开始部分先轮询任务数组，判断更新前提条件，并创建位掩码(bitmask)进行维护，即可实现**提前任务筛**。

由于现在是多个图任务占用一个处理单元，因此在消息传递时需要对每个活跃结点的处所有任务的消息打包进行传递。由于`process_edges`的API参数均为模板，因此可以新建一个消息传递类(message passing class)，类似gRPC的方式，提前组织好数据后再统一进行发送，这样也大大减少了分布式节点间的通信量。完整实施样例可以见附录B。

3. 实验环境

我们采用了宽度优先搜索(BFS)、连通分量(CC)、页面排序(PageRank)和单源最短路径(SSSP)四个算法作为基本的算法。然后将这些算法分别组合成如表1所示的任务集，每个任务集都由8个基本的算法组成。其中的异构性和同构性是通过内存访问带宽大小来进行分类的，详情可见原始论文或文 [14]。

表 1: 评测任务集描述

任务集	算法	Description
Homo1	$\{BFS, CC\} \times 4$	同构
Homo2	$\{PR, SSSP\} \times 4$	同构
Heter	$\{BFS, CC, PR, SSSP\} \times 2$	异构
M-BFS	$\{BFS\} \times 8$	相同任务
M-SSSP	$\{SSSP\} \times 8$	相同任务

关于单机共享内存的评测请见原始论文，这里只给出分布式环境下评测所采用的两个数据集。

表 2: 实验所采用的数据集

简写	数据集	$\|V\|$	$\|E\|$	大小
RM	rMat24 [15]	33.6 M	168 M	2.0 GB
TW	Twitter [16]	41.7 M	1.4 B	15.7 GB

由于超算机时占用严重，因此实验只在实验室的小集群上运行，环境配置请见原始论文。

运行时间实验结果如图5所示，可以清晰看到我们的Krill系统比Seraph系统最快快了4.25倍，这很大原因来源于Krill对各个图任务之间关系的挖掘。

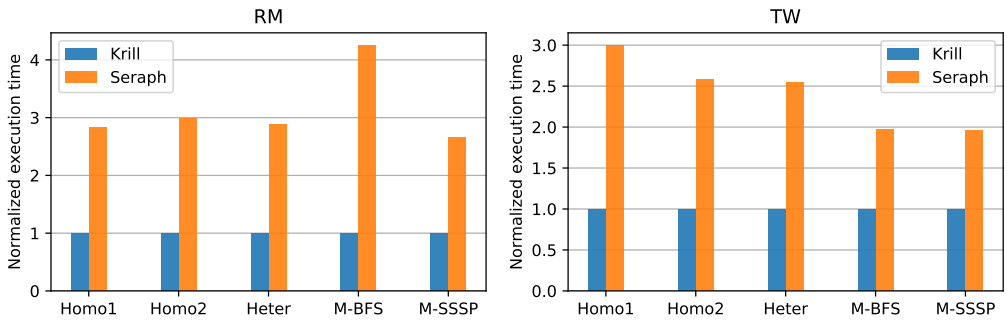


图 5: 分布式环境下Krill与Seraph运行时间比较

内存访问量比较结果如表3所示，从中也可以看出Krill减少了近1倍的内存访问开销，而且这还是在没有优化的情况下的比较，如果将Krill的所有优化技术全部结合使用上去，那么得出的内存访问减少量的差异应该更加明显。

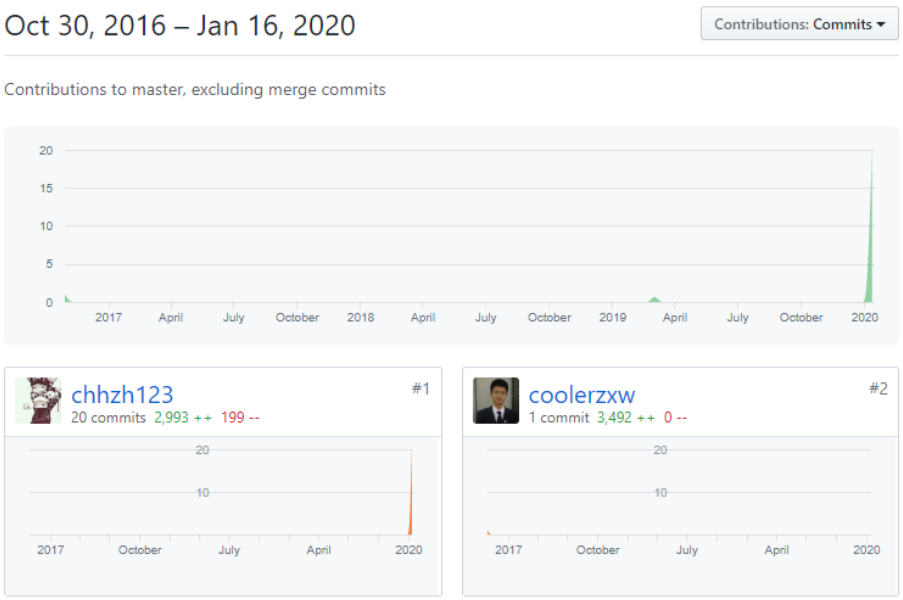
表 3: Krill和Seraph内存访问量比较(RM)

	Homo1	**Homo2**	**Heter**	**M-BFS**	**M-SSSP**
Krill	121,277,102,205	184,484,882,245	161,182,322,856	71,368,664,593	156,714,974,125
Seraph	183,416,200,102	375,861,175,531	287,505,536,960	115,908,456,530	351,852,690,249

六、 总结

本项目陈述了并发图处理的重要性，并且提出了自己的并发图系统Krill，也给出了单机共享内存和分布式环境下的实现。实验结果体现了我们提出的三个方法的有效性，大大减少了内存访问开销，进而提升了系统的整体性能。我们也相信Krill能够处理更加复杂的并发图任务，并得到更加广泛的应用。

PS：这里为了说明Krill和Seraph的分布式系统实现确实是本次课程期间完成，且具有一定工作量，附上Github提交记录截图，chhzh123为我的Github账号。



参考文献

[1] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[2] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. 2014.

[3] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. In *Proc. VLDB Endow.*, 2015.

[4] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: Online learning of social representations. 2014.

[5] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*, 2016.

[6] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. KnightKing: a fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles - SOSP '19*, 2019.

[7] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013.

[8] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. GraphMat: High performance graph analytics made productive. 2015.

[9] Keshav Pingali. High-speed graph analytics with the galois system. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, 2014.

[10] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. In *Proc. VLDB Endow.*, 2012.

[11] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[12] Jilong Xue, Zhi Yang, Zhi Qu, Shian Hou, and Yafei Dai. Seraph: An efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, 2014.

[13] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, and Haikun Liu. CGraph: A correlations-aware approach for efficient concurrent iterative graph processing. In *2018 USENIX Annual Technical Conference*, 2018.

[14] P. Pan and C. Li. Congra: Towards efficient processing of concurrent graph queries on shared-memory machines. In *2017 IEEE International Conference on Computer Design*, 2017.

[15] The Graph 500 List. Graph 500, 2017. https://graph500.org/.

[16] Jaewon Yang and Jure Leskovec. Patterns of temporal variation in online media. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, 2011.

附录 A. 章节对照

项目中所使用的技术与课程内容的关联性对照如下表所示，实际上分布式系统课程的内容是紧密渗透在我的项目之中的，毕竟图计算系统就是一个**物理分布，逻辑集中；个体独立，整体统**一的分布式系统。

项目采用技术	课程章节	课程内容
类封装	第1章	透明性
图结构与图性质解耦	第1章	策略与机制相分离
多线程技术	第3章	线程
Master-Mirror消息打包通信	第4章	RPC/gRPC
原子操作	第7章	以数据为中心的一致性模型
并行编程	第11章	大数据
VSCode + Github	补充内容	DevOps连续集成

当然还有很多细节技术这里没有一一列举出来，总的来说在分布式系统这门课上我学习到了很多东西，同时也都能够学以致用。

附录 B. Gemini+Krill编程实例(Multi-BFS)

假设前期所有数据结构都准备好了，这里只展示最核心的遍历步骤及消息传递的数据结构，考虑8个BFS的并发图任务。

```
class VecVertexId
{
public:
    VecVertexId() = default;
    VecVertexId(const VertexId val)
    {
        for (int i = 0; i < 8; ++i)
            vertex_id[i] = val;
    }
    VecVertexId(const VecVertexId &other)
    {
        for (int i = 0; i < 8; ++i)
            vertex_id[i] = other.vertex_id[i];
    }
    VertexId vertex_id[8] = {0};
};

active_vertices = graph->process_edges<VertexId, VecVertexId>(
    [&](VertexId src) {
        VecVertexId vecId(graph->vertices);
```

```
21          for (int i = 0; i < 8; ++i)
22              if (active_in[i]->get_bit(src))
23                  vecId.vertex_id[i] = src;
24          graph->emit(src, vecId);
25      },
26      [&](VertexId src, VecVertexId msg, VertexAdjList<Empty> outgoing_adj) {
27          VertexId activated = 0;
28          for (AdjUnit<Empty> *ptr = outgoing_adj.begin; ptr != outgoing_adj.end; ptr
                ↪ ++)
29          {
30              VertexId dst = ptr->neighbour;
31              bool flag = false;
32              for (int i = 0; i < 8; ++i)
33              {
34                  if (active_in[i]->get_bit(src) && parent[i][dst] == graph->vertices
                        ↪ && cas(&parent[i][dst], graph->vertices, msg.vertex_id[i]))
                        ↪ // be careful!
35                  {
36                      active_out[i]->set_bit(dst);
37                      flag = true;
38                  }
39              }
40              if (flag)
41              {
42                  activated += 1;
43                  common_active_out->set_bit(dst);
44              }
45          }
46          return activated;
47      },
48      [&](VertexId dst, VertexAdjList<Empty> incoming_adj) {
49          // advanced task filter
50          // int bit_mask = 0;
51          bool bit_mask[8] = {0};
52          int cnt = 0;
53          for (int i = 0; i < 8; ++i)
54          {
55              if (parent[i][dst] != graph->vertices) //(visited[i]->get_bit(dst))
56              {
57                  // return;
58                  // bit_mask &= 1 << i;
59                  bit_mask[i] = 1;
60                  cnt++;
61              }
62          }
```

```
63          if (cnt == 8)
64              return; // all visited
65          VecVertexId vecId(graph->vertices);
66          bool flag = false;
67          for (AdjUnit<Empty> *ptr = incoming_adj.begin; ptr != incoming_adj.end; ptr
                ↪ ++)
68          {
69              VertexId src = ptr->neighbour;
70              for (int i = 0; i < 8; ++i)
71              {
72                  if (!bit_mask[i] && active_in[i]->get_bit(src)) // dst not visited &
                        ↪  src active
73                  {
74                      vecId.vertex_id[i] = src;
75                      bit_mask[i] = 1;
76                      flag = true;
77                  }
78              }
79          }
80          if (flag)
81              graph->emit(dst, vecId);
82      },
83      [&](VertexId dst, VecVertexId msg) {
84          bool flag = false;
85          for (int i = 0; i < 8; ++i)
86          {
87              if (cas(&parent[i][dst], graph->vertices, msg.vertex_id[i]))
88              {
89                  active_out[i]->set_bit(dst);
90                  // return 1;
91                  flag = true;
92              }
93          }
94          if (flag)
95          {
96              common_active_out->set_bit(dst);
97              return 1;
98          }
99          else
100             return 0;
101     },
102     common_active_in, nullptr);active_vertices = graph->process_edges<VertexId,
            ↪ VecVertexId>(
103     [&](VertexId src) {
104         VecVertexId vecId(graph->vertices);
```

```
105         for (int i = 0; i < 8; ++i)
106             if (active_in[i]->get_bit(src))
107                 vecId.vertex_id[i] = src;
108         // VecVertexId vecId(src);
109         graph->emit(src, vecId);
110     },
111     [&](VertexId src, VecVertexId msg, VertexAdjList<Empty> outgoing_adj) {
112         VertexId activated = 0;
113         for (AdjUnit<Empty> *ptr = outgoing_adj.begin; ptr != outgoing_adj.end; ptr
                ↪ ++)
114         {
115             VertexId dst = ptr->neighbour;
116             bool flag = false;
117             for (int i = 0; i < 8; ++i)
118             {
119                 if (active_in[i]->get_bit(src) && parent[i][dst] == graph->vertices
                        ↪ && cas(&parent[i][dst], graph->vertices, msg.vertex_id[i]))
                        ↪ // be careful!
120                 {
121                     active_out[i]->set_bit(dst);
122                     flag = true;
123                 }
124             }
125             if (flag)
126             {
127                 activated += 1;
128                 common_active_out->set_bit(dst);
129             }
130         }
131         return activated;
132     },
133     [&](VertexId dst, VertexAdjList<Empty> incoming_adj) {
134         // advanced task filter
135         // int bit_mask = 0;
136         bool bit_mask[8] = {0};
137         int cnt = 0;
138         for (int i = 0; i < 8; ++i)
139         {
140             if (parent[i][dst] != graph->vertices) //(visited[i]->get_bit(dst))
141             {
142                 // return;
143                 // bit_mask &= 1 << i;
144                 bit_mask[i] = 1;
145                 cnt++;
146             }
```

```
147         }
148         if (cnt == 8)
149             return; // all visited
150         VecVertexId vecId(graph->vertices);
151         bool flag = false;
152         for (AdjUnit<Empty> *ptr = incoming_adj.begin; ptr != incoming_adj.end; ptr
                ↪ ++)
153         {
154             VertexId src = ptr->neighbour;
155             for (int i = 0; i < 8; ++i)
156             {
157                 if (!bit_mask[i] && active_in[i]->get_bit(src)) // dst not visited &
                        ↪  src active
158                 {
159                     vecId.vertex_id[i] = src;
160                     bit_mask[i] = 1;
161                     flag = true;
162                 }
163             }
164         }
165         if (flag)
166             graph->emit(dst, vecId);
167     },
168     [&](VertexId dst, VecVertexId msg) {
169         bool flag = false;
170         for (int i = 0; i < 8; ++i)
171         {
172             if (cas(&parent[i][dst], graph->vertices, msg.vertex_id[i]))
173             {
174                 active_out[i]->set_bit(dst);
175                 // return 1;
176                 flag = true;
177             }
178         }
179         if (flag)
180         {
181             common_active_out->set_bit(dst);
182             return 1;
183         }
184         else
185             return 0;
186     },
187     common_active_in, nullptr);
```