



科研训练三

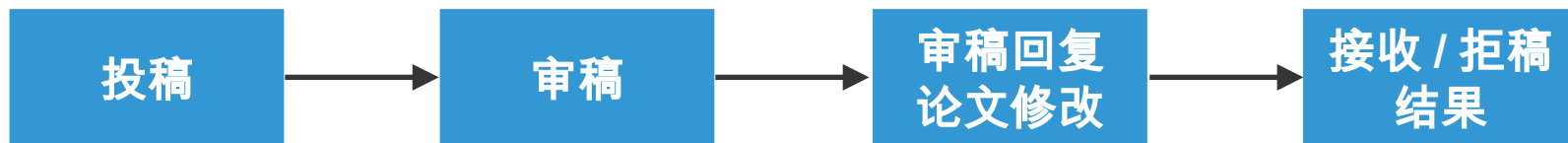
4 论文写作：防御性写作，Defensive Writing

晏潇 博士

yanxiaosunny@whu.edu.cn



论文审稿流程



- ❖ 通常有 3-5 位审稿人，可能是同一领域专家，也可能较为随机
- ❖ 负责写审稿意见和评价审稿回复
- ❖ 有 Area Chair 或 Associate Editor，但通常不读论文，只是综合审稿人意见
- ❖ 大模型和科研大通胀时代
 - 论文质量和审稿意见中性化，区分度不大
 - 写作质量，尤其是审稿回复的质量，对录用尤其关键



审稿人的反应难以预测

❖ 审稿人风格和品味各异，同一篇文章，不同审稿人可能从完全相反的角度攻击论文

- Smart vs. Dull
- Erudite vs. Ignorant
- Careful vs. Careless
- Tolerant vs. Severe



❖ 怎么防住不同审稿人的攻击，实现论文录用？



Disclaimer： 中稿的核心是研究质量，本讲主要讨论写作技巧，可能有偏颇表述



带着防守意识去写 (Defensive Writing)

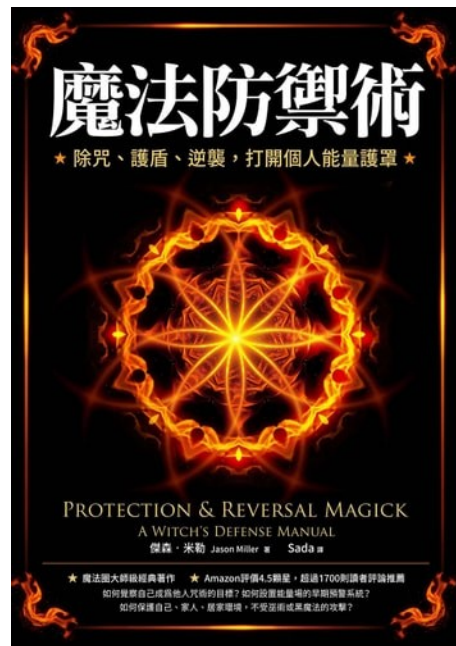
❖ 防御式写作的核心思路

- ❑ 写作时设想审稿人会怎么攻击和批评，提前把可能的问题解决掉
- ❑ 把不同审稿人的共性要求提炼出来，逐条满足
- ❑ 一套方法论，多年来实现 100+ 论文录用

❖ 针对不同审稿人的三条具体标准（3 个 S）

- ❑ Sparkling 亮眼 —— 在开头就打动人，虎头，以攻代守
- ❑ Smooth 流畅 —— 新手也能轻松阅读和理解论文
- ❑ Steady 扎实 —— 内容严谨，经得起反复严格检查

❖ 每条标准下面都有更具体且可操作的规则





Sparkling 亮眼：闪电型的审稿人 Flash

❖ 假想审稿人与攻击类型

- ❑ 截稿前一天打开邮件，手上有五篇论文要审
- ❑ 很忙，审稿意见要在三十分钟内写完
- ❑ 对这个题目没兴趣 & 不懂，但被分到了必须审
- ❑ . . .

❖ 摘要和引言需要给人留下强烈印象，让人决定接收论文

- ❑ 注意力和专注度会随时间下降，前面的内容读得更细
- ❑ 留下好或坏的第一印象，之后很难改变
- ❑ 有些审稿人会先下判断，然后找证据

❖ Let the flash rush!





规则一：长引言 Introduction

❖ 写一个长引言

- 把整篇论文的内容概述一遍，让审稿人有足够的信息做决策
- Intro 段落级提纲：问题；现有解法；我们的解法；实验；贡献
- 一条逻辑链串起来：问题 A 的核心是 B，已有方法缺点是 C，我们通过 D 机制来解决

CoroGraph: Bridging Cache Efficiency and Work Efficiency for Graph Algorithm Execution

Xiangyu Zhi
Xiao Yan*
Department of Computer Science and Engineering, Southern University of Science and Technology
zhixy202@mail.sustech.edu.cn
yansx@mail.sustech.edu.cn

Bo Tang
Ziyao Yin
Department of Computer Science and Engineering, Southern University of Science and Technology
tangb@sustech.edu.cn
yinyz202@mail.sustech.edu.cn

Yanchao Zhu
Ming Zhou
Gauss Department, Huawei Company
zhuycn@huawei.com
zhouming@huawei.com

ABSTRACT

Many systems are designed to run graph algorithms efficiently in memory but they achieve only cache efficiency or work efficiency. We tackle this fundamental trade-off in existing systems by designing CoroGraph, a system that attains both cache efficiency and work efficiency for in-memory graph processing. CoroGraph adopts a novel hybrid execution model, which generates update messages at vertex granularity to prioritize promising vertices for work efficiency, and commits updates at partition granularity to share data access for cache efficiency. To overlap the random memory access of graph algorithms with computation, CoroGraph extensively uses *coroutines*, i.e., a lightweight function in C++ that can yield and resume with low overhead, to prefetch the required data. A suite of designs are incorporated to reap the full benefits of *coroutines*, which include perfect pipeline, cache-friendly graph format, and step-free synchronization. We compare CoroGraph with the state-of-the-art graph algorithm systems via extensive experiments. The results show that CoroGraph yields shorter algorithm execution time than all baselines in 38 out of 28 cases, and its speedup over the best-performing baseline can be over 2. Detailed profiling suggests that CoroGraph achieves both cache efficiency and work efficiency with a low memory stall and a small number of processed edges.

PLDI Reference Format:

Xiangyu Zhi, Xiao Yan, Bo Tang, Ziyao Yin, Yanchao Zhu, Ming Zhou. CoroGraph: Bridging Cache Efficiency and Work Efficiency for Graph Algorithm Execution. PLDI, 17(0): XXX-XXX, 2023.

PLDI Artifact availability:

The source code, data, and other artifacts have been made available at <https://github.com/xiangyuzhi/corograph>.

1 INTRODUCTION

Graphs are ubiquitous in many domains such as social media [13, 15], e-commerce [7], finance [1, 14], and biology [22]. Algorithms are executed on these graphs to support various applications, e.g.,

*No two authors

This work is licensed under the Creative Commons BY-NC-ND 4.0 International license. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing the author(s). CoroGraph is a trademark of the author(s). Publication rights are reserved by the IEEE/ACM.

Proceedings of the 2023 ACM SIGPLAN Conference on Programming Language Design and Implementation, June 17, 2023, 4:XXX-4:XXX.

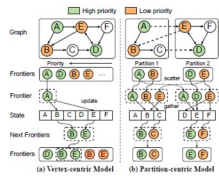


Figure 1: The partition-centric and vertex-centric execution models in existing systems (best viewed in color).

community identification [2, 45], recommendation [26], fraud detection [4], and drug discovery [20]. Examples of popular graph algorithms include PageRank [9], single source shortest path (SSSP), weakly connected component (WCC), and k-core [43]. Many frameworks are designed to execute graph algorithms efficiently, e.g., Liger [46], Galois [48], Gemini [49], GraphB [46], and GPOV [11]. In this paper, we consider in-memory graph algorithm frameworks that work on a single machine with multiple cores and focus on reducing algorithm execution time.

1.1 Motivation and Problem

Graph algorithms generally adopt the following patterns: each vertex v in the graph is associated with a vertex state $s[v]$; there are some active vertices called *frontiers*; algorithm updates the neighbors' state of each frontier, and these neighbors may become new frontiers. Existing graph algorithm frameworks optimize either the work efficiency or cache efficiency of graph algorithm execution.

Work efficiency is measured by the number of vertex state update operations conducted during graph algorithm execution [35, 46].

*This preview was created by the authors of the slides. For comments, see the notes to the slides.

Since each update is conducted along an edge (i.e., from frontier to target vertex), work efficiency is also quantified by the number of processed edges [35]. Existing systems (e.g., Galois) achieve work efficiency with a vertex-centric execution model. The observation is that frontiers make different contributions to algorithm progress, and thus work-efficient graph algorithms (e.g., SSSP, k-core, and WCC) determine a priority value for each frontier to prioritize promising frontiers that accelerate algorithm progress. As shown in Figure 1(a), the vertex-centric model uses a queue to manage the frontiers, and the highest-priority frontier (e.g., d) is obtained from the queue each time. The neighbors of the dequeued frontier (e.g., b and f) are updated, and if a neighbor meets the condition specified by the algorithm after update, it is inserted into the frontier queue, and its lower-priority instances will be pruned.

Cache efficiency is quantified by the ratio of the CPU stall time caused by cache misses over the algorithm execution time (called memory bound [10], [47, 48]). Since the neighbors of a frontier are usually not consecutive in memory, the neighbor update operation of graph algorithms essentially conducts random memory access. Random access causes both *read amplification* and *cache miss*, which result in high memory bound. Existing systems (e.g., GPOV) improve cache efficiency with a partition-centric execution model. In particular, the vertex states are organized into partitions that fit in L2 or L3 cache, and processing is conducted at partition granularity. As shown in Figure 1(b), the partition-centric model involves a scatter phase and a gather phase. In the scatter phase, the frontiers of each partition are copied to the partitions where its neighboring vertices reside. For instance, the neighbors of frontier a span two partitions, so a is copied to both partitions. In the gather phase, each partition collects the states from the scatter phase, updates the vertices within the partition, and gets the frontier for the next iteration. By ensuring that the working partition resides in cache, the partition-centric model avoids memory stall and achieves high cache efficiency.

Table 1 shows the trade-off between cache efficiency and work efficiency in existing graph frameworks. In particular, GPOV has low memory stall but poor work efficiency. This is because its partition-centric model divides computation by partition to share data access and cannot follow the priority order to process individual frontiers. Galois conducts less work but has high memory stall. This is because its vertex-centric model processes each frontier individually and cannot share data access among the frontiers as in the partition-centric model. This fundamental trade-off leads us to the following question: *is it possible to architect a graph algorithm framework that achieves both cache and work efficiency?*

1.2 Our Solution: CoroGraph

Our initial design is to reduce memory stall for the vertex-centric model with a *coroutine-based prefetch pipeline*. In particular, our prefetch pipeline allows programs to specify the data to load [9], and introduces an lightweight function that can yield and resume with low overhead [19]. In our pipeline, processing tasks are conducted

*This preliminary means that if an integer value is fetched via the cache line but which cache line happens because the automatic hardware prefetch of CPU will load consecutive data.

Table 1: Execution statistics of representative graph frameworks and our CoroGraph. The algorithm is SSSP, and the graph is LiveJournal. GPOV targets cache efficiency while Galois targets work efficiency. For both memory bound and processed edges (i.e., #Edges), smaller is better.

System	Memory Bound	#Edges (M)	Time (ms)
Liger	65.3%	184	1022
Gemini	53.2%	613	824
GraphB	60.5%	145	782
GPOV	52.9%	902	553
Galois	70.2%	89	513
CoroGraph	27.2%	68	262

by *coroutines*, which issue *prefetch instructions*, switch to the computation tasks of other *coroutines*, and resume for computation after the required data are fetched to cache. This design overlaps memory access and computation but the performance is unsatisfactory for two reasons. First, memory stall dominates execution time for vertex-centric model (over 70% for Galois in Table 1), and thus the gain is limited even if memory access and computation are perfectly overlapped (e.g., map-reduce by Galois (100/70 + 145)). Second, the threads have data conflicts when processing different frontiers in parallel because the frontiers share common neighbors, and these data conflicts invalidate prefetched data.

To tackle the problems above, CoroGraph adopts a novel hybrid execution model that combines the benefits of partition-centric and vertex-centric execution. In particular, like vertex-centric execution, the frontiers are managed by a priority queue and processed in the order of their contributions to algorithm progress, resulting in good work efficiency. Meanwhile, like partition-centric execution, updates to vertex states are committed at partition granularity. To accommodate the two mismatching patterns, the frontiers are processed in a scatter phase that generates update messages while the actual updates are conducted in a gather phase. A single thread is assigned to process all updates to a partition, which results in good cache efficiency by sharing data access among update operations.

The thread-to-partition update pattern also benefits prefetch by eliminating the data conflicts that invalidate prefetched data. Inside the hybrid execution model, we take a suite of optimizations in CoroGraph. First, we use the aforementioned coroutine-based prefetch pipeline to overlap memory access with computation. Second, we propose a lightweight but effective strategy to switch between prefetch and direct data access because prefetch becomes slower than direct access as a thread gradually warms the cache by processing a partition. Third, since the threads are blocked when waiting to access shared data structures, we adopt a flexible synchronization strategy, which allows the threads to proceed with the tasks instead of blocking. Besides, we also adjust the graph execution format to reduce cache miss and control the N/A-chance of modern processors when scheduling the tasks and configuring the prefetch pipeline.

We experiment with a popular graph algorithms (i.e., SSSP, k-core, PR, and WCC) and compare our CoroGraph with 5 state-of-the-art graph algorithm frameworks (i.e., Liger, Gemini, GPOV, Galois,



亮眼：其他规则

- ❖ 强调问题和挑战，不是方案，不要写太多我们解决方案的细节
 - ❑ 方案细节打动不了专家（知识渊博），还让新手审稿人难以理解
 - ❑ 问题为什么重要、根本困难是什么、现有方法为什么不好、我们为什么能做好
- ❖ 围绕一张主图或主表来组织关键论证（Alpha Figure or Table）
 - ❑ 闪电型的审稿人不会细读文字，具体的图和表给人印象更深，比文字论证更让人信服
 - ❑ 用让主图或主表把动机、问题、解法、效果都讲出来

Table 1: Execution statistics of representative graph frameworks and our CoroGraph. The algorithm is SSSP, and the graph is Livejournal. GPOP targets cache efficiency while Galois targets work efficiency. For both memory bound and processed edges (i.e., #Edges), smaller is better.

System	SSSP		
	Memory Bound	#Edges (M)	Time (ms)
Ligra	65.3%	184	1022
Gemini	55.2%	613	824
GraphIt	60.5%	145	782
GPOP	32.9%	902	553
Galois	70.2%	89	513
CoroGraph	27.2%	68	262

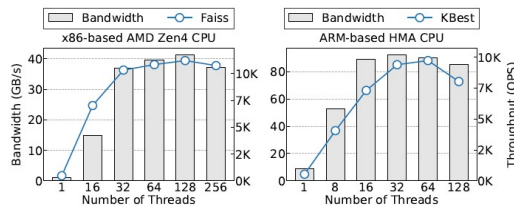


Figure 2: Memory traffic and query throughput at different thread counts. The dataset is GIST1M, and the recall target is 0.90. We utilize the SOTA proximity implementations, i.e., HNSW from FAISS [31] for x86-based AMD Zen4 CPU and SSG from KBest [41] for our ARM-based CPU.



打造一套概念和逻辑体系

❖ 起名字，搭一套概念和逻辑体系

- 清晰分类现有工作（场景，设计逻辑等），清晰描述它们的原理
- 定义关键概念，用它们来推理动机、问题和解法：考验对问题的理解

Graph algorithms generally adopt the following pattern: each vertex v in the graph is associated with a vertex state $s[v]$; there are some active vertices called frontiers; algorithm updates the neighbors¹ of each frontier, and these neighbors may become new frontiers. Existing graph algorithm frameworks optimize either the work efficiency or cache efficiency of graph algorithm execution.

efficiency with a vertex-centric execution model. The observation is that frontiers make different contributions to algorithm progress, and thus work-efficient graph algorithms (e.g., SSSP, k-core, and of graph algorithms essentially conducts random memory access. Random access causes both read amplification and cache miss², which result in high memory bound. Existing systems (e.g., GPOP) improve cache efficiency with a partition-centric execution model. In particular, the vertex states are organized into partitions that fit in



流畅 Smooth：迟钝型的审稿人 Dull

❖ 假想审稿人与攻击类型

- ❑ 不熟悉相关问题和工作
- ❑ 累了不愿思考，只想交差
- ❑ 记性和专注力差，容易走神
- ❑ 对公式和细节过敏
- ❑ 。 。 。

❖ 对领域有基础理解的学生应该能轻松读懂论文

- ❑ 吸引读者、把内容讲清楚，是作者的义务
- ❑ 读不懂的时候，多数审稿人会认为是作者的问题
- ❑ 读不懂会让人烦躁，并会为这种糟糕体验给出惩罚

❖ John knows nothing but we enlighten





顺滑：规则

❖ 每段只说一件事：段落级提纲

- ❑ 在几件事之间跳来跳去会造成混乱、分散注意力
- ❑ 可以在段首或段尾替读者做个总结；避免过长的段落，造成读者压力大

❖ 用加粗和斜体做适当的强调和归纳

- ❑ 黑体让人容易定位内容、把握整体结构
- ❑ 斜体可以强调关键概念和名词，吸引人注意

CoroGraph supports two modes to cater for the characteristics of different graph algorithms, i.e., *synchronous* and *asynchronous*. In particular, synchronous graph algorithms (e.g., Bellman Ford, PageRank, and WCC) require to process all frontiers before updating the vertex states while asynchronous algorithms (e.g., Δ -step, k-core) do not have this requirement. In asynchronous mode, CoroGraph executes the scatter phase and gather phase asynchronously and allows the threads to be in different phases as discussed in Section 4.1. In synchronous mode, CoroGraph first assigns the threads to conduct the scatter phase and only enters the gather phase after all threads finish scatter. Instead of copying the local message buffers of each thread to the global buffers of each block, the gather phase of synchronous mode directly accesses the local buffers to conduct update in order to reduce one message copy.

5.1 Experiment Settings

Datasets. In our experiments, we evaluate the systems on four representative benchmark datasets covering various data types and scales: GIST [6], OpenAI [48], Text2Image [17], and LAION [51]. These datasets span different dimensionalities, data distributions, and embedding modalities, enabling a comprehensive assessment of system performance. GIST represents classic hand-crafted vision descriptors, while the OpenAI dataset contains modern high-dimensional neural embeddings. Text2Image introduces cross-modal retrieval challenges, and LAION provides large-scale image embeddings. We summarize the dataset configurations in Table 1.

Baselines. We evaluate HetANNS against several state-of-the-art vector search systems spanning both CPU-optimized and specialized hardware implementations. In all experiments, HetANNS is configured with two NUMA groups. For the GIST and OpenAI datasets, the hybrid-precision retrieval pipeline uses 4-bit and 8-bit



顺滑：图表与隔离复杂性

❖ 一图胜千言

- 图和表比文字直观，值得下大功夫
- 画好图表之后，论述起来也省力
- 图表的标准：好看易懂，黑白能区分，字体大小合适，题目标注详实

❖ 不只是介绍自己的方法，还要讨论、解释、分析，给出洞见

- 方法本身之外，人还想知道它为什么能起作用
- 喜欢有洞见、直观、有趣的解释和例子

❖ 基于例子（Running Example）讲述比较复杂的算法和流程

❖ 隔离复杂性

- 扎实的工作通常复杂，但读者应该不看细节就理解思路框架
- 用定理、引理、证明隔离复杂部分；给出概览和路线图



Steady 扎实：挑剔型的审稿人 Picky

❖ 想审稿人与攻击类型

- 严谨认真的学者，他们让世界变好
- 记性好、逻辑强的学者，审稿人中不少见
- 毫无理由地持负面态度；心情不好时挑个人发泄
-

❖ 有知识也有耐心的读者读三遍也找不出错

- 顶尖学者之所以是顶尖，正因为他们认真，也正因如此才来做顶会和顶刊审稿人
- 小错就能让他们生气并拒稿，不管工作本身有多好
- 错误有很多形态：不一致、逻辑、数学、图表、拼写



❖ Be a Superman to Please the Batman!



扎实的规则

❖ 表述、命名、逻辑要一致，语言要精确

- ❑ 顶尖学者记性和逻辑都好，一定找得出不一致和错误
- ❑ 有些不算错误，稍加解释就能理清，但读论文没有解释的机会
- ❑ 整体逻辑要自洽，不能有冲突
- ❑ 好好设计实验，覆盖所有关键论断，提前消解可能的质疑
- ❑ 注意公式、表格、图和命名；同一个符号绝不表示两个不同的变量
- ❑ 表和图里的每样东西都要说得通：文字、标签、图例
- ❑ 出现规律或反常时要解释；「全面超过所有人」无聊，规律和反常更有趣

❖ 不要试图藏问题，把它们暴露出来

- ❑ 工作很少是完美的；藏起来专家一定会发现并攻击，主动暴露能挡掉主动攻击

❖ 打印出来，认真读 3-4 遍



风格和细节

❖ 论文不是散文，语言平实精准，不追求长句难词

- ❑ 介绍相关工作，不要堆砌技术名词，让不熟悉领域的读者看懂
- ❑ 避免长句和不常用的词，4 行以上的句子少用
- ❑ 避免长段

❖ 图表风格清新直接

- ❑ 传统标准：黑白打印能够区分，通过点型和线型区分
- ❑ 彩色作图：充分利用整个色域
- ❑ 图表中的文字：不能太小，以正文的文字和全屏 PDF 阅读为标准

❖ 避免孤悬挂行和章节

❖ 图表算法的出现位置，要和描述它的正文尽量接近，最好同一页



写审稿回复 Author Response

- ❖ 大模型时代，回复可能比论文更重要，要求也更高
 - 审稿人水平参差，为稳妥常给中间分，看别人的意见和你的回复再调整
 - 争锋相对回答审稿人问题，“对线”强度非常大
 - 回复要在 1-2 周内完成，常常需要大量补充实验
- ❖ 规则一：相信自己；你比多数审稿人更懂这个方向
- ❖ 规则二：结论先行，先给直接给答案，再给解释；适当总结问题
- ❖ 规则三：有逻辑、具体；引用图和表要指名，说明结果怎么解读
 - [W2：相对现有工作的创新] XXX 为适应流式场景做了三处主要创新：(1) 创新点一及收益。(2) 创新点二及收益.....
- ❖ 规则四：利用论文里已有的内容和其他审稿人的意见来帮助自己
- ❖ 规则五：排优先级，关键审稿人值得最多的注意力和投入



防御式写作与研究

- ❖ 写作像化妆，能让平庸的工作在某个地方被接收；要去顶会，工作本身得够好
 - 要达到 S3 的写作标准，研究得按 P3 来做
- ❖ 研究重要的问题（亮眼）：是不是重要问题、有没有指出 SOTA 的根本局限
 - 问题的关键特征是什么？它们怎么导致困难？
- ❖ 设计有原理的解法（顺滑）：核心想法是什么，它怎么和问题性质连起来
 - 和 SOTA 有什么不同？性能和复杂度能说到什么程度？
- ❖ 执行审慎的过程（扎实）：严谨地做设计、分析、编码和实验
 - 不要藏问题，出现意外时要讲清原因和机理
- ❖ 认真对待论文写作，留出时间打磨



与导师合作

- ❖ 一般学生等着学，好的学生反过来教；做你项目的主人
 - 研究在推动知识边界，导师也不知道答案，你本该比他更懂这个题目
 - 推着导师走：给他推荐论文、分享想法和实验计划、征求反馈
- ❖ 关注导师的问题和疑问，不是他的方案
 - 导师的优势是经验、批判性思维和学术标准，不是对目前题目的理解
 - 他的具体方案未必可行；用他的疑问引出你自己的解法
- ❖ 留出时间从杂事里抽身，去思考
 - 研究既要努力也要思考，但大多同学思考太少
 - 定期——一周一次——理清目标、位置和计划
- ❖ 用写作来引导项目：问题定义、方案设计、性能分析、实现计划
 - 写不出来就是没想清楚；有文档才能自查、复盘，也才能推导师



防御性写作与研究

- ❖ 写作像化妆，能够让一般的工作被接收；顶级发表，工作本身必须优秀
- ❖ 论文写作的 S3 标准，要求研究中需要做到 P3
- ❖ 研究重要的问题 **Prominent Problem: Sparking: Impress Upfront**
 - 问题是否重要？是否分析清楚了 SOTA 方法的根本缺陷？
 - 问题的核心和本质是什么？由这些核心和本质，导致了怎样的挑战？
- ❖ 设计理性的解法 **Principled Solution: Smooth: Follow with Ease**
 - 我们解法的核心是什么？它和问题的核心性质是如何关联的？
 - 我们方法和 SOTA 的不同在哪里？它的复杂度、收敛性、性能有什么理论结果？
- ❖ 执行严谨的研究流程：**Steady: Survive Rigorous Checks**
 - 设计、分享、编程和实验全程保持严谨
 - 不隐藏问题，出现异常的结果时，能够解释异常的原因



总结：防御性写作

- ❖ 满足 S3 写作标准，亮眼 (Sparkling)、顺滑 (Smooth)、扎实 (Steady)
- ❖ 照着详细规则打磨，并且要花时间
- ❖ 按 P3 的方式做研究，Prominent, Principled, and Prudent.



闪电型

Sparkling

Impress upfront

Prominent Problem



迟钝型

Smooth

Follow with ease

Principled Solution



挑剔型

Steady

Survive hostile checks

Prudent Methodology