Graph Computation with Adaptive Granularity

Ruiqi Xu
National University of Singapore
ruiqi.xu@nus.edu.sg

Yue Wang
Shenzhen Institute of Computing Sciences
yuewang@sics.ac.cn

Xiaokui Xiao
National University of Singapore
xkxiao@nus.edu.sg

***Abstract*—Despite the development of various distributed graph systems, little attention has been paid to the granularity of computation and communication, which can significantly impact overall efficiency. Moreover, users often struggle to write and optimize new parallel algorithms to fit different programming abstractions, which can be a daunting task. To address these challenges, this paper introduces** Argan**, a parallel graph system that offers efficient adaptive-grained executions and a user-friendly abstraction.** Argan **utilizes the adaptive-Grained Asynchronous Parallel (**GAP**) model, which enables runtime adjustments of granularity to enhance performance. Additionally, its programming model allows users to directly derive parallel programs from existing batch sequential algorithms. Our experiments using real-life and synthetic graphs demonstrate that for a variety of graph applications,** GAP **effectively improves the performance of** Argan**, which outperforms** Grape$^+$**,** PowerSwitch**, and** Maiter**.**

***Index Terms*—distributed computation, granularity, parallel model, programming model, graph analytics**

I. Introduction

In recent years, distributed systems have been developed to handle the increasing volume of graph data [1]–[8]. These systems partition a graph G into fragments and distribute them across a cluster of workers connected via a network to answer a query $Q(G)$. Each worker performs computations over its local fragment of G, and communicates with remote workers to collaboratively answer $Q(G)$. The granularity of the computation, *i.e.,* the amount of computation performed by a worker between its two consecutive communications with remote workers, plays a crucial role in the efficiency of these systems. However, existing systems often have a fixed granularity and ignore the need for different and dynamic levels of granularity in real-world applications.

The granularities of existing systems are often either *coarse-grained* or *fine-grained*, both of which have their own advantages and drawbacks. The former features high throughput but comes with *stragglers* and computational *staleness*. The latter incurs a greater potential of parallelism, but suffers from the increased *communication overhead*. The different levels of granularity are determined and fixed by the systems' parallel models and programming models.

Coarse granularity is often found in systems with synchronous parallel models (*e.g.,* Pregel [1]), where computations over all workers are coarsened together by the global barriers. It is also common in graph-centric systems (*e.g.,* Blogel [9]), where the computation is performed in batches over the local subgraph. For these systems, the coarse granularity often incurs skewed workload, where only few workers (*i.e.,* stragglers) in the cluster remain working. Worse still, since messages received by a worker are blocked by its ongoing computation for a relatively long period of time, the computation may become stale, *i.e.,* canceled or overridden due to the more up-to-date messages.

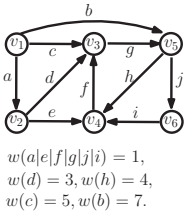


Fig. 1: Graph G

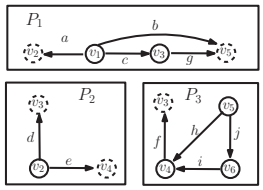


Fig. 2: Partitions of G

Models		Ticks T 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
BSP	P_1	a	b	c	g	X					X	g	X	g			X			
&	P_2					X	d	e			X		X				X			
GC	P_3					X	f	h	j	i	X	f	X	h	j	i	X	h	j	i
AAP	P_1	a	b	c	g	X				g	X	g	X	g	X		X			
&	P_2						d	e	X				X				X			
GC	P_3						f	h	j	i	X	f	X	-	-	h	j	i		
AP	P_1	a	X	b	X	c	g	X		g	X									
&	P_2		d	X	e	X														
VC	P_3				h	j	f	X	h	j	h	j	i							
GAP	P_1	a	X	b	c	X	g	X	g	X										
&	P_2		d	e	X															
ACE	P_3					f	X	h	j	h	j	i								

TABLE I: SSSP from v_1 under different models

Fine granularity, on the other hand, is common for systems that are both asynchronous and vertex-centric (*e.g.,* GraphLab$_{\text{async}}$ [10] and Maiter [5]). In these systems, the individual tasks are generally vertex programs with relatively short execution time and messages can be processed on the fly. The fine-grained systems, however, may suffer from the frequent and fragmented communications.

Several attempts have been made on intermediate granularities [7], [11], [12]. However, these approaches all have their limitations. For instance, GiraphUC [7] applies a one-size-fit-all granularity, which is not suitable for all applications with different run time characterizations.KLA [11] predefines the granularity of a graph application by fitting it into a theoretical cost model. Its granularity remains static throughout the computation, and hence, is unable to handle runtime anomalies such as stragglers. PowerSwitch [12] uses runtime information for switching granularities, but selects between only fine-grained and coarse-grained computations, both of which are suboptimal. Therefore, it is essential to investigate a

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00169

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00169

more adaptable and dynamic granularity that can adjust to the changing requirements of graph applications during runtime.

Motivation. Observe that the differences in granularity have a significant impact on the overall performances of parallel executions. We illustrate this with the following example.

Example 1. Consider the query of Single Source Shortest Paths (SSSP) over graph G with v_1 as the source (Figure 1). We run the parallelized Dijkstra's algorithm [3] using 3 workers P_1, P_2 and P_3 over the partitioned G (Figure 2).

Table I shows the execution traces of SSSP for different combinations of programming and parallel models. For simplicity, we adopt the following model of traces: A "tick" represents a unit of time, during which a worker can perform either of the two following operations. One operation involves scanning a single edge and updating the distance on its target vertex. The other operation ejects an arbitrary number of messages to remote workers, denoted as "X", which are received at no additional cost at the next tick. Next, we study how the programming and parallel models influence granularity and its impact on efficiency.

(a) For systems under the Bulk Synchronous Parallel (BSP) [13] model and the graph-centric (GC) programming model (such as Blogel [9] and Grape [3]), all workers are coarsened together by global barriers (at $T = 5, 10, 12$ and 16). As a result, it takes 19 ticks in total. The coarse granularity leads to staleness, *e.g.,* edge j at P_3 is scanned 3 times, with the first 2 passes being stale.

(b) The performance can be improved by switching to the more flexible Adaptive Asynchronous Parallel (AAP) model [8], [14]. The model removes global barriers and introduces "delay sketches" (marked as "-") to delay the ingestion of received messages. This reduces the staleness *e.g.,* edge j is scanned only twice instead of three times. However, it still suffers from the stragglers due to the coarse granularity introduced by its GC model. For instance, P_1 runs alone for 5 ticks at the start, which makes it a straggler.

(c) Fine granularity is achieved in systems that employ both Asynchronous Parallel (AP) model and vertex-centric (VC) programming model, such as Maiter [5] and $\mathsf{GraphLab}_{\mathsf{async}}$ [4], [10]. In these systems, messages are forwarded and ingested at the earliest available tick, resulting in a response time of 13 ticks and little staleness in computation. However, the frequent communication also leads to inefficiencies. For example, P_2 takes 4 ticks to complete computation, emitting messages at ticks $T = 4$ and 6. In contrast, it only takes 3 ticks under coarse GC model. □

Different parallel graph algorithms often require different granularity for their optimal performances. Moreover, certain algorithms may favor different granularity at different stages of the computations [12]. These lead us to a crucial question: How can we effectively adjust the granularity to boost parallel executions for various graph algorithms, while maintaining programming simplicity?

Contributions. To answer the question, this paper introduces Argan (Asynchronous Graph Engine with Adaptive Granularity), the first graph system that offers runtime adaptive granularity. Argan adjusts its granularity to the characterization of the applications and the runtime staleness, which effectively mitigates the drawbacks of both fine-grained and coarse-grained computations. Lying in the core of Argan is a new programming model and a new parallel model, which we explain in the following.

New programming model. Following the graph-centric paradigm, we develop a new programming abstraction named ACE. Different from conventional graph-centric models, where the computation over a local subgraph is performed in coarse-grained batches, ACE models local computation over the subgraph as fixed point iterations and allows communication between any two consecutive iterations. Such a model enables Argan to adapt to a wide range of different levels of granularity, making it more flexible.

Better still, we show that the flexibility in granularity does not complicate the programming under ACE. Instead, ACE is easy to use since it directly parallelizes a class of existing batch sequential algorithms abstracted as fixpoint iterations, and offers correctness guarantees. Thus, users are relieved from designing new platform-specific programs from scratch.

New parallel model. Although the ACE model provides the flexibility to achieve different levels of granularity, it is still unclear how to systematically tune the granularity on-the-fly for better efficiencies. This requires us to develop a new *parallel model* that defines how a parallel program is executed over a cluster of computing units and controls the timing of inter-process communications.

To this end, we propose the Adaptive-grained Asynchronous Parallel (GAP) model, which is both efficient and expressive, and offers *runtime dynamic* granularity adjustment. We approach granularity adjustment by seeking to enhance *computational effectiveness*,and introduce a feedback control mechanism to dynamically adjust the granularity based on runtime information. Such an approach strikes a balance between the staled computation and the communication overhead and boosts the overall efficiency. GAP also comes with extensive *expressive power*, and subsumes other parallel models like BSP and AP as its special cases.

We illustrate how ACE and GAP help to improve the overall efficiency with the following example.

Example 2. Following Example 1, equipped with ACE and GAP, the response time is reduced to only 12 ticks by employing an adaptive granularity (see Table I). 1) At the start , GAP detects P_1 as the straggler and immediately sends messages to invoke the idle P_2. 2) To mitigate stale computation, GAP switches to a finer granularity on P_3, allowing it to immediately ingest the message received at $T = 10$ and override the pending stale computation over edges j and i. The finer granularity is made possible via the ACE programming model (see Section II-A). 3) GAP reduces 1

TABLE II: Notations

Symbol	Notation
n	number of workers
$G = (V, E)$	graph G with vertices V and edges E
$F_i = (V_i, E_i)$	i-th fragment with vertices V_i and edges E_i
P_i	i-th worker (or the coordinator when $i = 0$), $i \in [0, n]$
M_i (resp. $M_{i,j}$)	messages received at P_i (resp. from P_i to P_j
h_{in} (resp. h_{out})	in-message (resp. out-message) handler
f_{step}, f_{term}	local step function, termination condition
T_c (resp. T_w)	communication (resp. staleness) cost
σ, Σ	a shared variable, the set of all shared variables
$\mathbb{B}_i^+$ (resp. $\mathbb{B}_{i,j}^-$)	buffer for holding messages in M_i (resp. $M_{i,j}$)
x_v, Ψ_i	the status variable of v, all status variables at P_i
f_{x_v}, Y_{x_v}	update function for x_v, input parameters of f_{x_v}
D_i	the intermediate results at P_i
ξ_i^+ (resp. $\xi_{i,j}^-$)	message passing indicator for $\mathbb{B}_i^+$ (resp. $\mathbb{B}_{i,j}^-$)
η_i, φ	granularity bounds at P_i, computational effectiveness
ρ, $\mathcal{A}$	a parallel ACE program, a sequential algorithm
χ_v	amortized staleness computation cost at v

tick of communication, as P_2 sends messages in batches. □

As proofs of concept, we parallelize algorithms including SSSP, Graph Coloring (Color), PageRank (PR), Core Decomposition (Core) and Graph Simulation (Sim). Empirically evaluating using real and synthetic graphs, our findings are summarized as follows.

(a) Compared with systems including Grape [3], Grape$^+$ [8], [14], GraphLab$_{\mathsf{sync}}$ [4], [10], GraphLab$_{\mathsf{async}}$ [4], PowerSwitch [12] and Maiter [5], Argan is at least 38%, 44%, 73% and 17%, faster for SSSP, Color, PR and Core, respectively.

(b) We compare ACE programs under GAP against their counterparts under AAP, AP and BSP, which are all special cases of GAP. For SSSP, Color, PR and Core, GAP is at least 2.1, 2.2, 1.7 and 1.3 times faster than other parallel models, respectively. This verifies the efficiency of GAP.

(c) Over the synthetic graph with size of $|G| = 6 \times 10^9$, Argan responds to queries for SSSP, Color, PR, Core and Sim in 10.0s, 14.4s, 29.9s, 49.7s and 11.3s, respectively.

The application scope of this work primarily includes 1) a class of iterative parallel ACE programs that incur staleness in parallel executions and 2) a class of sequential graph algorithms modeled as fixpoint iterations where the corresponding ACE programs can be directly derived. A detailed discussion on its application scope is presented in Section V.

II. Argan: The System

For a query class $\mathcal{Q}$, we study how to compute the answer $Q(G)$ to a query $Q \in \mathcal{Q}$ over a graph $G = (V, E)$ in parallel across a cluster of workers. The graph is partitioned according to a partition strategy $\mathcal{P}$ into n fragments $F_1, ..., F_n$, and scattered across the n workers, denoted as $P_1, ..., P_n$. To answer $Q(G)$, each worker performs computation over its local fragment while communicating with each other via messages. The messages sent from P_i to P_j are denoted by $M_{i,j}$, and the messages received at P_i are denoted by M_i.

To provide better flexibility and efficiency in graph computation, we introduce Argan, a new graph system that features an adjustable granularity. The adjustability is made possible through Argan's programming model that decomposes coarse-grained graph-centric programs into iterations of finer granularity (Section II-A) and the parallel model that automatically groups these iterations into desired granularity (Section II-B).

A. The Programming Model

The ACE programming model of Argan extends conventional graph-centric models. It provides users with high-level abstractions of parallel graph computations as follows.

Argan works over n *workers* $P_1, ..., P_n$ and an additional *coordinator* P_0. It employs the vertex partitioning as the partition strategy. For a graph G, its vertex set V is partitioned into n disjoint sets $V_1', ...V_n'$. Each fragments $F_i = (V_i, E_i)$ contains 1) vertices in V_i', 2) edges E_i that are adjacent to V_i' and 3) vertices induced by E_i.

Programming. For a class $\mathcal{Q}$ of queries, over the coordinator P_0 and workers P_i, users need to provide two functions GlobalEval, LocalEval, denoted as an ACE program ρ.

GlobalEval. Over P_0, the GlobalEval function takes as input a query $Q \in \mathcal{Q}$, and messages $M_{*,0}$ from workers. It declares a set Σ of variables, which are shared among all workers and synchronized via messages. It 1) broadcasts the query Q at the start of the computation, 2) updates globally shared variables in Σ via messages from and to workers and 3) determines whether the computation of $Q(G)$ has terminated or not according to the shared variables.

LocalEval. Upon receiving messages, each worker P_i executes one round of LocalEval over fragment F_i and updates its local intermediate results, denoted as D_i. At the end of the local computation, each P_i emits messages $M_{i,j}$ to P_j.

Following the graph-centric paradigm, the LocalEval function computes over the subgraph in batches, which is essentially coarse-grained. To enable an adjustable granularity, the function is decomposed into the following phases.

(1) In-message Handler. Denoted as h_{in}, the function is triggered at P_i at the start of LocalEval, and is responsible for ingesting messages $M_{j,i}$ from other P_j. For $M_{j,i}$ from remote worker P_j, each message is a pair of $\langle v, x_v \rangle$, where $v \in V_i$ and x_v is the status variable associated with v. The local D_i consist of two parts, *i.e.,* 1) the part that can be amortized to status variables x_v, each associated with a vertex v in F_i, denoted as Ψ_i, and 2) the part that can be seen as an aggregation of Ψ_i, denoted as S_i. To update D_i, h_{in} integrates x_v into Ψ_i using an aggregate function g_{aggr}, and updates S_i accordingly. For $M_{0,i}$ from the coordinator P_0, each message encodes an update to the local copy of a shared variable $\sigma \in \Sigma$.

(2) Local Iterations. After h_{in}, the ACE program enters a loop of local iterations. The loop iteratively executes a *local step function* f_{step}, until the *local termination condition* f_{term} is satisfied. The function f_{term} returns a boolean variable based on D_i, signalling the termination of the iterations. After each iteration, new messages $\Delta M_{i,j}$ to remote worker P_j are generated, and accumulated in a corresponding buffer, denoted as $\mathbb{B}_{i,j}^-$. Each message in $\Delta M_{i,j}$ is in the form of $\langle v', x_{v'} \rangle$,

where v' is a vertex at P_i, that is also replicated at a remote worker P_j, and $x_{v'}$ is its associated status variable.

Modelling local computations as iterations of f_{step}, the graph-centric computation is decomposed into finer granularity. This makes flexible granularity possible, as messages can alternatively be forwarded and ingested between any two consecutive f_{step}. Instead of exposing the complicated task of granularity adjustment to users, we integrate it into the system as part of the parallel model (discussed in Section III).

(3) Out-message Handler. Denoted as h_{out}, the function is triggered after $f_{\mathsf{term}}(D_i) = \mathsf{true}$. It aggregates accumulated messages in $\mathbb{B}^-_{i,j}$ as $M_{i,j}$, and generates $M_{i,0}$ to P_0.

B. The GAP Model

Next, we introduce the *Adaptive-Grained Asynchronous Parallel* (GAP) model, which allows for adjustable granularity. In contrast to the programming model, GAP defines how the user-provided components are executed in parallel, which is transparent to users.

Under GAP, each worker P_i (resp. P_0) executes LocalEval (resp. GlobalEval) upon receiving new messages in an asynchronous way. We refer to each invocation of LocalEval as one *round* of computation.

Structures. GAP employs the following structures.

Accumulative in-message buffer. In an asynchronous environment, messages $M_{j,i}$ are received by P_i in a non-blocking way and accumulated in a local buffer, denoted as $\mathbb{B}^+_i$. The buffer is reset after its messages are ingested by h_{in}.

Algorithm 1 LocalEval under GAP

Input: LocalEval with components of h_{in}, h_{out}, f_{step} and f_{term}, local data F_i, D_i, and $\mathbb{B}^+_i$
Output: Updated D_i and messages $M_{i,*}$

```
1: /* key components for granularity flexibility colored in blue */
2: D_i ← h_in(D_i, B_i^+);   clear B_i^+; set ξ_i^+ and ξ_{i,*}^- as false
3: messages are received and accumulated in B_i^+
4: while f_term(D_i) = false do
5:     (D_i, ΔM_{i,*}) ← f_step(D_i)
6:     for all j ∈ [1,n], j ≠ i do
7:         append ΔM_{i,j} to B_{i,j}^-
8:         if ξ_{i,j}^- = true then
9:             M_{i,j} ← h_out(B_{i,j}^-);  send M_{i,j} to P_j;  clear B_{i,j}^-
10:    if ξ_i^+ = true then
11:        D_i ← h_in(D_i, B_i^+);   clear B_i^+
12:    adjust ξ_{i,j}^- and ξ_i^+ according to rules R_1 ∼ R_3
13: for all j ∈ [1,n], j ≠ i do
14:    M_{i,j} ← h_out(B_{i,j}^-);  send M_{i,j} to P_j;  clear B_{i,j}^-
```

Message passing indicators. To enable flexible timing for communication, each worker P_i maintains a set of system-tuned boolean variables called *message passing indicators*. P_i declares a message-passing indicator ξ^+_i for $\mathbb{B}^+_i$, and $n-1$ indicators $\xi^-_{i,j}$, one for each $\mathbb{B}^-_{i,j} (j \in [1,n], j \neq i)$. When ξ^+_i (resp. $\xi^-_{i,j}$) is true, it indicates that the messages in the corresponding buffers should be ingested (resp. forwarded) immediately after the current f_{step} returns. After the messages in the buffer are processed, ξ^+_i (resp. $\xi^-_{i,j}$) is reset to false.

Overview. GAP computes $Q(G)$ as follows.

GlobalEval. The coordinator P_0 starts by posting the query Q to all workers. Messages for updating globally shared variables in Σ are processed by P_0 in a non-blocking way. It also decides the termination of $Q(G)$.

LocalEval. Algo 1 shows LocalEval under GAP, with key components for enabling granularity flexibility highlighted in blue, summarized as follows.

(1) Messages arrive at each worker in a non-blocking way without global barriers, and are accumulated in a local in-message buffer $\mathbb{B}^+_i$ before ingested by h_{in} (line 3).

(2) h_{in} can be invoked multiple times through one round of LocalEval, instead of only once (lines 10-11). That is, after each f_{step}, P_i checks ξ^+_i and ingests the messages accumulated in $\mathbb{B}^+_i$ by calling h_{in}, if $\xi^+_i = \mathsf{true}$.

(3) Messages in $\mathbb{B}^-_{i,j}$ are also forwarded with flexible timing based on the corresponding indicator of $\xi^-_{i,j}$, rather than only at the end of LocalEval (lines 8-9).

(4) The indicators of ξ^+_i and $\xi^-_{i,j}$ are dynamically adjusted at worker P_i (line 12), which are transparent to users.

Simulations and Convergence. BSP and AP are both special cases of GAP. These can be carried out by GAP by specifying the configuration of ξ^+_i and $\xi^-_{i,j}$ as follows.

BSP. ξ^+_i and $\xi^-_{i,j}$ are initially false and are set as true, if and only if $f_{\mathsf{term}}(D_i) = \mathsf{true}$ for all $i \in [1,n]$. After ingesting (resp. forwarding) messages, ξ^+_i (resp. $\xi^-_{i,j}$ for each j) is reset to false. Here, $f_{\mathsf{term}}(D_i)$ is seen as a global variable in Σ shared across all P_i via messages.

AP. We consider the AP under vertex-centric (resp. graph-centric) model, which is fine-grained (resp. coarse-grained).

(1) AP under vertex-centric paradigm can be achieved, when (a) each f_{step} executes vertex program over exactly one single vertex, and (b) ξ^+_i and $\xi^-_{i,j}$ are constantly true.

(2) AP under graph-centric paradigm, on the other hand, is a special case of GAP, when ξ^+_i and $\xi^-_{i,j}$ fixed as false. This way, messages are ingested (resp. forwarded) at the beginning (resp. end) of a round of LocalEval.

We show that GAP can optimally simulate AAP, PRAM and MapReduce, and provide conditions under which GAP is guaranteed to asynchronously converge (see [15] for details). The convergence conditions require monotonicity of LocalEval with respect to the local partial results.

The Parallel Model. We complete the GAP model by describing how the system controls the indicators ξ^+_i and $\xi^-_{i,j}$ based on a set of rules and a few system-provided parameters.

Parameters for Message Passing. Our goal is to dynamically adjust the indicators to achieve the following objectives: 1) reduce the stragglers, 2) mitigate computation staleness and

3) avoid excessive communication overheads. To this end, we introduce the following parameters:

(1) *Worker status*. For each worker P_i, we define its status as idle, if it has finished the current LocalEval and $\mathbb{B}_i^+ = \emptyset$. Otherwise, its status is working. The worker status of P_i is a shared variable in Σ that only P_i itself can modify.

(2) *Granularity bounds*. Denoted as η_i, the granularity bound is an upper bound of timespan during which P_i can execute LocalEval locally without handling messages. The bound η_i is automatically tuned by the system (see Section III).

We next present the set of rules for setting indicators.

Setting the Indicators. The indicators ξ_i^+ and $\xi_{i,j}^-$ are initially false. To set the indicators, we follow the rules below:

$\mathcal{R}_1$: Set $\xi_{i,j}^-$ as true, if the status of P_j is idle.

$\mathcal{R}_2$: Set ξ_i^+ as true, all workers except P_j are idle.

$\mathcal{R}_3$: Set ξ_i^+ and $\xi_{i,j}^-(j \in [1,p])$ as true, if $T(\mathbb{B}_i^+) \geq \eta_i$.

Rules $\mathcal{R}_1$ and $\mathcal{R}_2$ set $\xi_{i,j}^-$ and ξ_i^+ based on worker status. They help mitigate the impact of stragglers by forwarding and ingesting messages in advance, instead of passively waiting for the straggler to catch up.

Rule $\mathcal{R}_3$ controls the granularity of both computation and communication. Here $T(\mathbb{B}_i^+)$ denotes the timespan for which the buffer $\mathbb{B}_i^+$ has been holding messages. We study in Section III on how to adjust η_i to boost the performance.

We demonstrate these rules with the following example.

Example 3. In the same settings as Examples 1 and 2, consider evaluating $Q(G)$ with Argan (*i.e.,*GAP & ACE):

(a) The system immediately detects P_1 as the "straggler" after the first tick, since both P_2 and P_3 are in idle. Thus, employing rule $\mathcal{R}_1$, messages generated on the target vertex of edge a is immediately forwarded to P_2, reducing its idle time.

(b) The default granularity η_i is initialized as 2 ticks for all P_i. Consequently, we can see: 1) LocalEval at P_1 and P_3 are truncated after ticks $T = 4$ and $T = 9$ according to Rule $\mathcal{R}_3$, as the local timer exceeds the granularity bounds of $\eta_1 = \eta_3 = 2$; 2) Since LocalEval at P_3 is truncated after $T = 9$, P_3 is able to ingest the message at $T = 10$, and consequently avoids the redundant scan over edge i. In contrast, one can verify that when using $\eta_i = 1$ or $\eta_i = 3$, the response time will be at least 1 tick longer. This shows that appropriate granularity bounds η_i boost the efficiency of the parallel computation. □

III. Adaptive Granularity

In this section, we answer how to adjust the granularity η_i tailoring for better performance of the parallel programs. We formulate the problem in Section III-A as an optimization over the computation effectiveness. Then, in Section III-B, we introduce a feedback control mechanism to dynamically adjust η_i based on runtime information. In Section III-C, we categorize graph algorithms and provide estimations of computation staleness for each class. By categorizing an ACE program and utilizing the proposed framework, the granularity can be automatically and adaptively adjusted at runtime. This enables Argan to optimize its parallel performance and adapt to runtime characteristics of different applications. At last, we provide practical optimizations to enhance its efficiency.

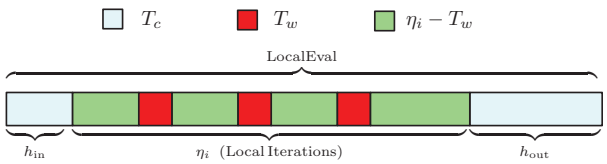


Fig. 3: Composition of costs in LocalEval

A. Problem Formulation

Our objective is to strike a balance between computation staleness and communication overheads, both of which are introduced by the parallelizations. To this end, we formulate the objective of granularity adjustment as an optimization problem over the *computation effectiveness*.

Computation Effectiveness. Denoted as φ, we define the computation effectiveness, *i.e.,* the ratio of the effective local computation against the overall cost, as a function over η_i:

$$\varphi(\eta_i) = \frac{\eta_i - T_w}{\eta_i + T_c} \tag{1}$$

Shown in Figure 3, we explain the terms as follows.

(1) The granularity η_i of local computation at P_i, is the time interval during which the local iterations last, according to $\mathcal{R}_3$.

(2) T_w denotes the sub-timespan of the stale computation in η_i, marked as red parts in Figure 3. It consists of computations that are cancelled or overridden by later computations due to more up-to-date messages.

(3) T_c denotes the communication overhead. Shown in Figure 3, it typically includes costs of the message handlers h_{in} and h_{out} for transferring messages.

Putting these together, we can see φ at P_i is the ratio of effective computation $\eta_i - T_w$ against the total cost $\eta_i + T_c$ of LocalEval including the communication. The objective of tuning η_i can be formulated as maximizing φ for each P_i. To this end, we next model T_w and T_c as functions of η_i.

Communication Overhead. For each P_i, $T_c(\eta_i)$ can be modelled as follows.

$$T_c = \sum_{j \neq i} T_{\mathbb{B}}(|M_{i,j}|) \tag{2}$$

Here, $T_{\mathbb{B}}$ denotes an end-to-end hardware-dependent function that maps buffer size to point-to-point communication costs, which can be profiled in advance offline (see Section VI).

Computation Staleness. As is shown in Figure 3, the staled computation is generally scattered across the timespan η_i of local iterations. To identify the staled parts T_w, we need to further characterize the local iterations of f_{step}, and amortize the computation cost to each individual nodes or edges.

For a timespan η_i of local computation at P_i, we amortize the cost η_i to vertices v in F_i, according to how their associated

x_v is accessed and updated. To achieve this, we extend the programming model of ACE as follows. Denoted as f_{x_v}, an *update function* is employed to update x_v and each f_{step} consists of a sequence of such update functions. Each f_{x_v} takes as input a set of status variables, denoted as Y_{x_v}, and deterministically decides the value of x_v. This extended model is elaborated in Section IV, here we focus on how to extract the amortized computation cost.

Suppose an update function $f_{x_v}(Y_{x_v})$ is called at clock t_0, and takes a timespan of T_0. We amortize the timespan T_0 to status variables in $Y_{x_v} \cup \{x_v\}$, *i.e.,* the ones that are either read or written by f_{x_v}. For a status variable x_u, if the amortized cost is T_u, we add a tuple $\langle t_0, T_u, x_u\rangle$ at u, denoted as χ_u. Here, χ_u is a timestamped sequence in the format of $\{\langle t_1, T_1, x_1\rangle, \ldots, \langle t_m, T_m, x_m\rangle\}$.

With the amortized cost of χ_v on each vertex, the staleness T_w is also amortized to vertices and can be extracted by a *staleness function* τ. The function takes the following as inputs: 1) the updated status variable x_v, 2) the final fixpoint denoted as x_v^*, 3) the sum of amortized cost of $T_v = \sum_{\langle t_j, T_j, x_v\rangle \in \chi_v} T_j$ for updating x_v. The staleness amortized to vertex v is computed as $\tau(x_v, x_v^*, T_v)$, by extracting the stale part of T_v. The overall staleness can be expressed as:

$$T_w = \sum_{v \in V_i} \tau(x_v, x_v^*, T_v) \tag{3}$$

The details of cost amortization and the definitions of function τ are discussed in details in Section III-C based on a categorization of graph applications.

B. Granularity Adjustment

We next introduce an algorithm for adapting granularity η_i. In order to properly adjust the granularity η_i, we need to acquire the effectiveness $\varphi(\eta_i')$ at an alternative η_i', and adjust the granularity to η_i' if $\varphi(\eta_i') > \varphi(\eta_i)$. However, currently based on Equations 1-3, we can only acquire $\varphi(\eta_i')$ if the program is running at η_i', rather than its current η_i. Towards this, we show that 1) when an algorithm is running at η_i, its effectiveness $\varphi(\eta_i)$ can be efficiently estimated on the fly, and 2) all $\varphi(\eta_i')$ for any $\eta_i' \leq \eta_i$ can also be easily acquired employing some auxiliary structures. The adjustment of η_i is then informed by these estimations.

Estimating Communication Cost. To estimate T_c for an alternative $\eta_i' \leq \eta_i$, we only needs to recover each $|M_{i,j}|$ at the time of $t = \eta_i'$, denoted as $|M_{i,j}^{t=\eta_i'}|$. Towards this we only need to build a lookup table mapping timestamp to message buffer sizes, such that the size $|M_{i,j}^{t=\eta_i'}|$ can be restored. Consequently, the cost T_c can be expressed as:

$$T_c(\eta_i') = \sum_{j \neq i} T_{\mathbb{B}}(|M_{i,j}^{t=\eta_i'}|) \tag{4}$$

Estimating Staleness. As for the estimation of staleness $T_w(\eta_i')$, since the overall staleness is already amortized to vertices with timestamps, it is can be expressed as:

$$T_w(\eta_i') = \sum_{v \in V_i} \tau(x_v^{t=\eta_i'}, x_v^*, T_v^{t=\eta_i'}) \tag{5}$$

Here, $x_v^{t=\eta_i'}$ denotes the value of x_v at time $t = \eta_i'$, and $T_v^{t=\eta_i'} = \sum_{\langle t_j, T_j, x_v\rangle \in \chi_v \wedge t_j \leq \eta_i'} T_j$, both of which can be readily retrieved from the amortized cost of χ_v.

Note that fixpoint of x_v^* for each status variable x_v are mostly unknown before a query $Q(G)$ is answered. Hence, the fixpoint x_v^* in Equation 5 must be substituted with an estimation for practical use. Towards this, we use $x_v^{t=2\eta_i}$ as an estimation, *i.e.,* we wait for an extra round of LocalEval, and use the intermediate results as an estimation of the fixpoints. That is,

$$T_w(\eta_i') \approx \sum_{v \in V_i} \tau(x_v^{t=\eta_i'}, x_v^{t=2\eta_i}, T_v^{t=\eta_i'}) \tag{6}$$

We show in Section VI-A via experimental study that such substitution of the fixpoint x_v^* is effective and justified.

The Algorithm. Putting these together, we now present the Granularity Adjustment (GA) algorithm. It is triggered every two rounds of LocalEval at each P_i, while running the user-defined ACE program ρ. The algorithm comprises two phases: *information collection* and *granularity adjustment*. Each phase corresponds to one round of LocalEval and lasts for a timespan of η_i. In the first phase, the algorithm collects the amortized cost χ_v for each local vertex v and stamps the accumulated messages. In the second phase, after another round of LocalEval, the algorithm estimates $\varphi(\eta_i')$ for a set of candidates in the range of $(0, \eta_i]$, and updates the local granularity η_i accordingly.

Algorithm 2 Granularity Adjustment (GA)

Input: F_i, D_i and η_i, an ACE program ρ
Output: Updated D_i and η_i
1: /* Information Collection Phase */
2: **let** $t_0 \leftarrow \mathsf{clock}()$; **let** Q_η be a queue of granularity-vertex pairs
3: **let** S_j be a lookup table mapping time t' to size of buffer $M_{i,j}$
4: **for all** $v \in F_i$ updated by ρ, till the activation of h_{out} **do**
5: $\quad t' \leftarrow \mathsf{clock}() - t_0$; $\quad x_v \leftarrow f_{x_v}(Y_{x_v})$; $\quad t \leftarrow \mathsf{clock}() - t_0$
6: $\quad$ amortize $t - t'$ to $Y_{x_v} \cup \{x_v\}$, with timestamp t
7: $\quad$ add an entry mapping t to current $|M_{i,j}|$ in each S_j
8: $\quad$ push $\langle t', v\rangle$ into Q_η
9: /* Granularity Adjustment Phase */
10: run ρ to update D_i for another period of η_i;
11: **let** $T_w \leftarrow 0$; **let** S_η be a set of granularity-effectiveness pairs
12: **for all** $\langle t, v\rangle \in Q_\eta$ **do**
13: $\quad$ incrementally update T_w by adding parts from χ_v
14: $\quad$ **let** $T_c(t) = \sum_j T_{\mathbb{B}}(S_j(t))$
15: $\quad$ **let** $\varphi(t) = (t - T_w)/(t + T_c(t))$; add $\langle t, \varphi(t)\rangle$ into S_η
16: **if** $\varphi(t)$ in S_η is monotonically increasing *w.r.t.* t **then**
17: $\quad \eta_i \leftarrow 2\eta_i$
18: **else** $\eta_i \leftarrow \arg_t \max_{\langle t, \varphi(t)\rangle \in S_\eta} \varphi(t)$

Shown in Algorithm 2, it takes as input F_i, D_i and the current η_i. It runs a given ρ while adjusting the local granularity η_i for every two consecutive LocalEval rounds.

In the first phase, the algorithm collects the amortized computation cost (lines 1-8). This is carried out by recording the local clock before and after each f_{x_v} is executed (line 5). It also constructs a lookup table S_j mapping a timestamp t' to the current $|M_{i,j}|$, in order to restore the size of $M_{i,j}^{t=t'}$

for estimations in the next phase (line 7). All the timestamps along with corresponding vertices are recorded in the queue Q_η as candidates (line 8).

The second phase estimates $\varphi(t)$ for all candidates t in Q_η. It first runs the LocalEval of ρ for another η_i to get a rough estimation of the final fixpoint x_v^* for each vertex. Then it computes $\varphi(t)$ according to Equations 1, 4 and 5 (lines 13-15). The candidate granularity with the highest estimated effectiveness is picked to update the current local η_i (line 18). If the effectiveness is monotonically increasing, then the optimal granularity could be beyond the scope of $(0, \eta_i]$, hence we enlarge the granularity by doubling it (line 17).

Cost Analysis. We analyze the extra time and space complexity introduced by GA to the original ACE program ρ.

Time Cost. The GA algorithm does *not* increase the complexity of the original ρ. In the first phase, there is $\mathcal{O}(1)$ additional cost for timestamping and recording the entries in Q_η, S_j and χ_v, whenever an update function f_{x_v} is triggered, which is at least $\mathcal{O}(1)$. The same goes for the second phase, which incurs extra cost for linear scanning of the recorded Q_η and S_η.

Space Cost. Denote by V_{η_i} the multi-set of vertices that is accessed by LocalEval during a period of η_i, then we can see the introduced space cost is in $\mathcal{O}(|V_{\eta_i}|)$. As is shown in Section VI, a query usually consists of a few dozens of LocalEval rounds, and hence one round of η_i only takes a small fraction of the total response time. Consequently, the additional space cost of $\mathcal{O}(|V_{\eta_i}|)$ is also very small compared to the space cost of the original ρ, as $|V_{\eta_i}|$ is bounded by the time cost η_i.

C. Staleness Categories

It remains to answer how to define the staleness function τ to extract the amortized T_w over each x_v. To address this question, we categorize graph algorithms and formulate τ for each of the category.

Intuitively, τ represents the additional cost incurred by parallelization that is absent in a sequential run. To determine this function, we investigate the access pattern of f_{x_v} in both sequential and parallel runs.

Access Patterns. For sequential algorithms modeled as fixpoint iterations (see Section IV) and parallel ACE programs, we categorize them based on how each status variable x_v is accessed by the program ρ. Some algorithms only read x_v after it reaches the fixpoint in x_v^*, while others may access x_v and update it for multiple times through the computation. For the former category, no staleness is incurred as change propagation is only triggered by converged values. We call these *Propagate-After-Fixpoint* (PAF) algorithms, including the Dijkstra's algorithm for SSSP [16], k-core [17], [18] and graph simulation [19], [20]. The latter category is named *Propagate-Before-Fixpoint* (PBF) algorithms, since each x_v can be accessed before its fixpoint. PBF algorithms usually update each x_v multiple times before it converges, such as PageRank [21] and h-index [22].

TABLE III: Categories Based on Access Pattern

Category	Access Pattern		Examples
	$\mathcal{A}$	$\rho_{\mathcal{A}}$	
I	PAF	PAF	Sim, Core (peeling)
II	PAF	PBF	SSSP (Dijkstra), BFS, WCC, MST (Borůvka), Color
III	PBF	PBF	PR, SimRank, Core (h-index), SSSP (Bellman-Ford)

It is important to note that the access pattern of a program ρ may also change due to parallelization. A PAF program ρ for sequential runs may become PBF when running in parallel due to the incompleteness of local information. The parallelized Dijkstra's algorithm (see Section IV) is one such example.

According to the access pattern of a sequential algorithm $\mathcal{A}$ and its parallelized ACE program $\rho_{\mathcal{A}}$, they can be categorized into the following three classes (Table III).

Category-I. This category consists of programs that are PAF for both sequential and parallel executions, such as Graph Simulation (Sim) [19], [20], and peeling-based Core Decomposition (Core) [17], [18]. These algorithms do not incur staleness over their parallel executions, and the workload of each worker is determined by the partitioning over G and the query Q. The staleness function can be formalized as:

$$\tau(x_v, x_v^*, T_v) = 0 \tag{7}$$

Category-II. This category contains algorithms that are sequentially PAF, but PBF for parallel runs, including the parallelized Dijkstra's algorithm for SSSP, Graph Coloring (Color), Breadth-First Search (BFS), Weakly Connected Components (WCC), and Borůvka's algorithm for Minimum Spanning Tree (MST) [23]. The staleness function for algorithms in this class can be formalized as:

$$\tau(x_v, x_v^*, T_v) = \begin{cases} T_v & \text{if } x_v^* \neq x_v \\ 0 & \text{if } x_v^* = x_v \end{cases} \tag{8}$$

That is, staleness is only incurred when x_v is accessed before its fixpoint in a PBF manner.

To amortize the cost T incurred by update function $f_{x_v}(Y_{x_v})$, T_u is individually recorded for each $x_u \in Y_{x_v}$, such that $\sum_u T_u = T$, and no cost is amortized to x_v.

Category-III. Algorithms that are PBF for both sequential and parallel runs fall into this category, including the Δ-based PageRank (PR) [5], SimRank [24], h-index based Core [25] and the Bellman-Ford algorithm for SSSP [26]. The corresponding staleness function can be formalized as:

$$\tau(x_v, x_v^*, T_v) = \frac{\Delta x_v^*}{\Delta x_v + \Delta x_v^*} T_v \tag{9}$$

Here, Δx_v^* denotes the changed part in x_v^*, *i.e.,* $\Delta x_v^* = |x_v^* - x_v|$. Similarly, we also keep track of the changed part in x_v in the first phase, denoted as Δx_v. The ratio of stale computation is quantified as $\Delta x_v^*/(\Delta x_v + \Delta x_v^*)$, which characterizes the staleness by comparing the residual change Δx_v^* against the overall change $\Delta x_v + \Delta x_v^*$.

To amortize the cost of the update function $f_{x_v}(Y_{x_v})$, the whole cost T_v is directly tied to x_v.

D. Optimizations

Although we have shown that the GA algorithm does not increase the complexity of a given ρ program, its frequent use of timestamps and the large size of candidate set S_Q might still slow down the overall execution of ρ. To further speedup the GA algorithm and reduce its impact on the overall efficiency and memory usage, we introduce a discretization based optimization to speed up the baseline of GA.

Discretization. The optimized algorithm is named Granularity Adjustment with Discretization (GAwD). It implements two-folds of discretization over GA, described as follows.

1) To avoid the frequent invocation of the expensive high-precision clock at line 5 of GA, we substitute the amortized cost of $t-t'$ with an estimation of the cost of f_{x_v}. Specifically, the cost of $f_{x_v}(Y_{x_v})$ is estimated as $|Y_{x_v}|+1$ when x_v is fixed-lengths primitive data-types; and as $|x_v| + \sum_{x_u \in Y_{x_v}} |x_u|$, if x_v is some advanced data-structure like sets and maps. This is because f_{x_v} typically consists of a linear scanning of Y_{x_v}.

2) Note that the candidate set size of S_η in GA is unnecessarily large. Towards this, we pre-determine the size of S_η as an integer k and discretize the candidate interval $(0, \eta_i]$ into only k candidates of $\eta^{(l)} = l \cdot \eta_i / k$, $l \in [1, k]$. This way, (a) the size of list $|\chi_v|$ for each v is also limited to k, saving more space, and (b) the size of S_η is limited to k, which saves the cost for scanning and selecting the optimal candidates in the second phase of GA. We discuss the choice of k for GAwD in Section VI-A.

IV. Parallelization

Writing correct parallel ACE programs is not an easy task for users. To assist in this process, we parallelize existing sequential algorithms as ACE programs.

Unlike Grape [27], which plugs in batch and incremental versions of an algorithm, we directly parallelize the batch ones. We accomplish this by first modeling a class of sequential batch algorithms as "fixpoint iterations". We then show how to derive parallel ACE programs from this model.

A fixpoint model. We model sequential graph algorithms as fixpoint iterations, following [28]. A sequential batch algorithm $\mathcal{A}$ tailored for a query class $\mathcal{Q}$ often works as follows.

(1) To answer $Q(G)$ for a $Q \in \mathcal{Q}$ with $\mathcal{A}$, the algorithm builds data structures $D_\mathcal{A}$, serving as auxiliary structures and intermediate results. $D_\mathcal{A}$ can be split into two parts, $\Psi_\mathcal{A}$ and $S_\mathcal{A}$. The former is a set of *status variables* x_v, *i.e.,* the structures amortized to each vertex v, while the latter is aggregations of variables in $\Psi_\mathcal{A}$.

(2) To update each $x_v \in \Psi_\mathcal{A}$, $\mathcal{A}$ applies an *update function* f_{x_v}, *i.e.,* $x_v = f_{x_v}(Y_{x_v})$, where Y_{x_v} denotes a subset of $\Psi_\mathcal{A}$ that decides the value of x_v.

(3) $\mathcal{A}$ terminates when all x_v in $\Psi_\mathcal{A}$ reaches a fixpoint *w.r.t.* f_{x_v}, *i.e.,* $f_{x_v}(Y_{x_v}) = x_v$ for all x_v in $\Psi_\mathcal{A}$. Then, the final result $Q(G)$ is extracted from $D_\mathcal{A}$.

We say $\mathcal{A}$ is *fixpoint iterations*, if it can be expressed as

$$(D_\mathcal{A}^{t+1}, H_\mathcal{A}^{t+1}) = f_\mathcal{A}(D_\mathcal{A}^t, H_\mathcal{A}^t, Q, G), \text{such that} \qquad (10)$$

(1) $D_\mathcal{A}^t$ denotes $D_\mathcal{A}$ after the t-th round of iterations.

(2) $H_\mathcal{A}^t$ is the *active set* of x_v after t-th iteration, such that for each x_v in $\Psi_\mathcal{A}$, if x_v has not reached the fixpoint, then $x_v \in H_\mathcal{A}^t$.

(3) The *step function* $f_\mathcal{A}$ serves as the immediate consequence operator of the fixpoint. It (a) selects x_v in $H_\mathcal{A}^t$ and updates their value as $f_{x_v}(Y_{x_v})$, (b) adjusts $H_\mathcal{A}^{t+1}$ by removing x_v that have reached the fixpoint and adding any new x_u that may be affected by changes in its Y_{x_u} and (c) updates the data structures in $S_\mathcal{A}$ according to changes in $\Psi_\mathcal{A}$.

Parallelization. We next show how to derive the parallel ACE program $\rho_\mathcal{A}$, from a sequential $\mathcal{A}$ modeled as fixpoint iterations. The LocalEval and GlobalEval of $\rho_\mathcal{A}$ are as follows.

LocalEval. At each worker P_i, LocalEval declares the following structures according to $\mathcal{A}$. 1) The local D_i consists of $\Psi_{\mathcal{A},i}$ and $S_{\mathcal{A},i}$, which corresponds to $\Psi_\mathcal{A}$ and $S_\mathcal{A}$ in $\mathcal{A}$. The former consists of only local x_v in F_i. The latter is aggregated from $\Psi_{A,i}$. 2) It also declares the message buffers of $\mathbb{B}_i^+$ and $\mathbb{B}_{i,j}^-$. These contain messages that are in the form of $\langle v, x_v \rangle$, *i.e.,* a key-value pair with vertex v as keys and its associated x_v as value. 3) It keeps track of a local active set, denoted as $H_{\mathcal{A},i}$, containing only vertices in local F_i.

In-message handler h_{in} ingests the messages in $\mathbb{B}_i^+$. For each message $\langle v, x_v{}' \rangle$, v is a replicated vertex in $V_i \setminus V_i'$, and $x_v{}'$ is an update to the local x_v from F_j. To process the messages, h_{in} 1) aggregates $x_v{}'$ into the local x_v , *i.e.,* $x_v \leftarrow g_{\mathsf{aggr}}(x_v{}', x_v)$ and 2) updates local $S_{\mathcal{A},i}$ and $H_{\mathcal{A},i}$ accordingly.

Local iterations repeat f_{step} locally at P_i, which corresponds to $f_\mathcal{A}$. It extends $f_\mathcal{A}$ in the following ways: 1) updates are conducted over the local D_i and $H_{\mathcal{A},i}$, and 2) whenever an updated vertex v is replicated on remote fragment F_j, a new message of $\langle v, x_v \rangle$ is appended to the buffer $\mathbb{B}_{i,j}^-$. The function f_{step} is repeated until $f_{\mathsf{term}}(D_i) = \mathsf{true}$, which is achieved when the local fixpoint of $H_{\mathcal{A},i} = \emptyset$ is reached.

Out-message handler h_{out} aggregates messages in $\mathbb{B}_{i,j}^-$ using g_{aggr} as $M_{i,j}$ and sends them to remote worker P_j, for each j. Here g_{aggr} is derived from the way x_v is updated in $\mathcal{A}$.

GlobalEval. For GlobalEval at the coordinator, it checks if all workers has reached the local fixpoint with no messages pending to ingest. If so, the computation of ρ is terminated, and each worker extracts partial results from the local D_i.

Correctness. We say the parallelization of a deterministic sequential algorithm $\mathcal{A}$ is correct, if $\rho_\mathcal{A}$ always returns the same results as $\mathcal{A}$. The correctness can be ensured, if the sequential $\mathcal{A}$ satisfies certain conditions. Here we summarize the key intuitions for the correctness of parallelization: 1) The conditions for $\mathcal{A}$ are defined such that the parallelized $\rho_\mathcal{A}$

TABLE IV: Datasets

Name	Abbr.	$\|V\|$	$\|E\|$	Directed	Network Type
Hollywood [29]	HW	1.1×10^6	5.6×10^7	no	collaboration
DBpedia [30]	DP	6.2×10^6	3.3×10^7	yes	knowledge base
Livejournal [31]	LJ	4.8×10^6	6.8×10^7	yes	social network
Twitter [32]	TW	4.2×10^7	1.5×10^9	yes	social network
Friendster [31]	FS	6.6×10^7	1.8×10^9	no	social network
UKWeb [33]	UK	1.1×10^8	3.7×10^9	yes	hyperlink

satisfies the corresponding conditions for asynchronous convergence; 2) We show that the sequential $\mathcal{A}$ can be mapped to a special case of parallel run of $\rho_{\mathcal{A}}$. See [15] for detailed conditions and proofs.

V. Discussions

Before presenting the experiment results, we discuss the application scope and limitations of Argan. The graph applications that benefit most from Argan are those that incur staled computations and can be directly parallelized from their sequential counterparts.

Argan's adaptive granularity boosts efficiency without complicating the programming. Its GAP model not only enables a wide range of granularity for graph computation, but also expressively subsumes popular models including BSP, AP and AAP as special cases. Through granularity adjustment, Argan is able to switch among these models, and therefore its performance is not inferior to systems built on these parallel models. Moreover, formulating the problem of granularity adjustment as improving the overall computation effectiveness, the process of granularity adjustment is made transparent to users. Users are only required to identify the staleness category for a graph application, instead of performing theoretical analysis over the algorithm.

However, Argan's granularity adjustment also has its own limitations. For applications in category-I, *e.g.,*Sim, there is little room for further improvement. Additionally, Argan assumes that the computation cost can be amortized to vertices, which may not always be the case. Moreover, parallel algorithms that do not fit into the ACE model, *e.g.,* recursive algorithms, are beyond the scope of Argan.

As for parallelization, Argan takes a step further than Grape and Grape$^+$by providing a way to derive ACE programs directly from a class of graph algorithms characterized as fixpoint iterations. This way, users are no longer required for developing new parallel algorithms from scratch or adapting existing incremental algorithms for parallelizations. However, this also narrows the scope of the parallelizable algorithms compared to Grape and Grape$^+$, primarily comprising fixpoint iteration algorithms with monotonicity property. Also note that the parallelization is to derive an iterative ACE program, which excludes certain algorithms such as recursive ones.

VI. Experimental Study

In this section, we conduct experiments to answer the following questions: 1) How effective are GA and GAwD, compared to baselines with fixed granularity? 2) Is Argan parallel scalable, *i.e.,* taking less time with more computation resources? 3) How does the efficiency of GAP compare to BSP, AP and AAP? 4) Can Argan scale to large graphs? 5) How does Argan compare to existing systems? We answer question 1) in Section VI-A, 2) to 4) in Section VI-B, and 5) in Section VI-C, respectively.

Datasets. We use 6 real-life datasets of various types, both directed and undirected, summarized in Table IV. Edge weights and vertex labels are randomly generated for testing SSSP and Sim, for all datasets except DP, which comes with vertex labels.We also generate synthetic datasets employing the built-in power-law graph generator of GraphLab$_{\text{sync}}$ [4], [10], fixing $\alpha = 2.5$ and varying $|V|$ to test the scalability of Argan.

Queries. We randomly generate the following queries. 1) For SSSP, we sample 10 nodes in each graph as the source. We ensure each source reaches more than 90% of vertices in V. 2) For Sim, we generate 10 patterns, fixing $|Q| = (|V_Q|, |E_Q|) = (4, 5)$.

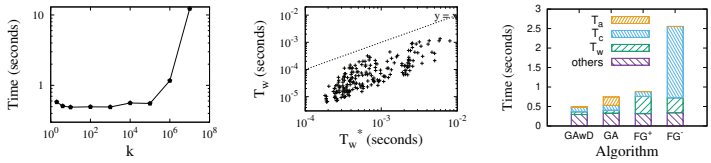
Algorithms. We implement the system of Argan with C++ and openMPI. We parallelized 1) the Dijkstra algorithm [16] for SSSP, 2) the Welsh-Powell algorithm [34] for Color, 3) the Delta-based algorithm [5] for PR, 4) the h-index algorithm [25] for Core and 5) the algorithm [19] for Sim, following Section IV. These algorithms cover all three categories identified in Section III-C, encompassing commonly used graph applications. Details for these parallelizations are discussed in [15].

Competitors. We compare Argan against other systems, including Grape [3], Grape$^+$ [8], [14], GraphLab$_{\text{sync}}$ [4], [10], GraphLab$_{\text{async}}$ [4], [10], PowerSwitch [12] and Maiter [5]. These competitors cover various combinations of different parallel models including BSP, AP, AAP and programming models including both GC and VC. Among these, Grape and Grape$^+$are most similar to Argan, as they are essentially graph-centric systems with key differences in parallel models. We implement our own codes for unavailable applications, including Sim and Core for GraphLab$_{\text{sync}}$, GraphLab$_{\text{async}}$, PowerSwitch and Maiter, as well as Color for Maiter. For systems that support loading partitioned graphs, we use XtraPuLP [35] as default partitioning.

We deploy the systems on a cluster of 22 machines, each with an Intel Xeon CPU E5-2692 v2 @ 2.20GHz (12 cores) and 64 GB memory. Each query is repeated 5 times, and the average response time is reported. The function $T_{\mathbb{B}}()$ is profiled using the Netguage [36] package offline (see [15]).

A. Effectiveness of Granularity Adjustment

We first use SSSP over LJ as an example and verify the effectiveness of GAwD and GA by comparing with fixed-grained baselines. Specifically, we examine the following aspects: 1) How to select the parameter k for the discretization employed by GAwD. 2) The estimations of T_w employed by GA and GAwD are justified. 3) GAwD optimizes GA without degrading the overall effectiveness. The results for other query classes and datasets are in accordance to the shown results.



(a) Varying k (GAwD) (b) T_w v.s. T_w^* (c) Response Time

Fig. 4: Effectiveness of Granularity Adjustment (SSSP, $n = 128$, LJ)

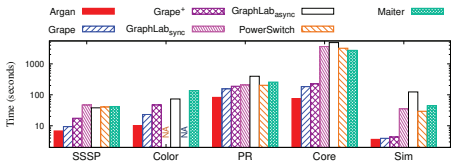
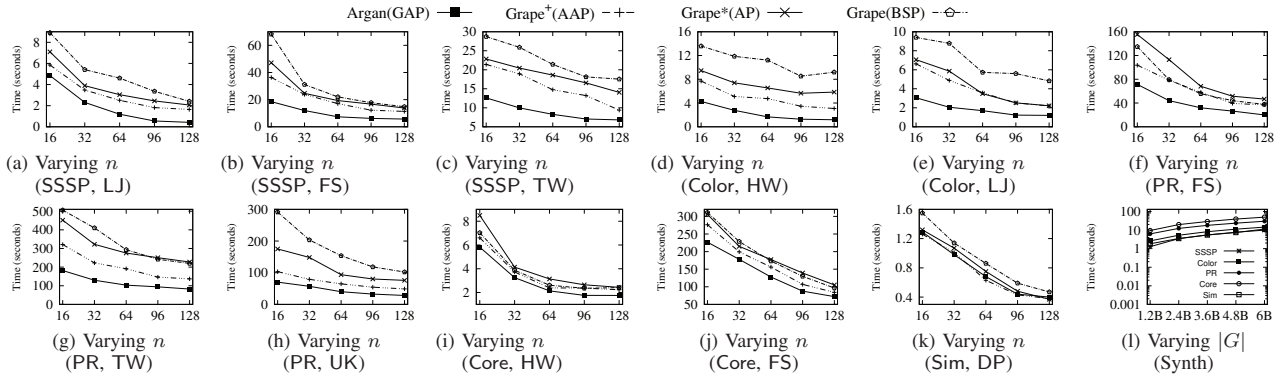


Fig. 5: Response Time on TW ($n = 128$)



(a) Varying n (SSSP, LJ) (b) Varying n (SSSP, FS) (c) Varying n (SSSP, TW) (d) Varying n (Color, HW) (e) Varying n (Color, LJ) (f) Varying n (PR, FS)

(g) Varying n (PR, TW) (h) Varying n (PR, UK) (i) Varying n (Core, HW) (j) Varying n (Core, FS) (k) Varying n (Sim, DP) (l) Varying $|G|$ (Synth)

Fig. 6: Efficiency of Argan

Parameter k in GAwD. We first investigate how to select a proper parameter k for GAwD. Varying k from 2 to 1×10^7, the overall response time of SSSP with GAwD is shown in Figure 4a. We find that: 1) When k increases from 1×10^5 to 1×10^7, the cost of GAwD grows by 120 times from 9.8×10^{-2}s to 11.8s (accounted in the response time, not individually shown) and dominates the overall response time. 2) When $4 \leq k \leq 1 \times 10^3$, the changes in response time are less than 3.7%. 3) When $k = 2$, we also see a slight increase of 14% in overall response time, which indicates 2 is too small for an effective GAwD granularity adjustment. In the rest of the section, we fix $k = 4$ for GAwD, unless otherwise stated.

Estimations of T_w. We next verify the estimation of Equation 6. Towards this, we first compute the fixpoints of x_v^* for a given query Q. Then we run the query for one more time including x_v^* as additional inputs, based on which the real $T_w(\eta_i)$ in Equation 5 can be computed. We denote such staleness as T_w^*. Figure 4b plots 200 points of the estimated T_w against its real value T_w^*, which are randomly sampled at runtime. We can see that: 1) $T_w \leq T_w^*$, as expected. This is because T_w only captures partial staleness cost. 2) The correlation coefficient of T_w and T_w^* is 0.79. That is, T_w serves as a good estimation of T_w^*.

Composition of Response Time. We compare the response time and φ of GAwD and GA to baselines with fixed granularity. We consider the following baselines: 1) Fixing $\eta_i = \infty$, all worker run at the most coarsen granularity, denoted as FG^+. 2) Fixing $\eta_i = 0$, the program ρ is executed at an extreme fine granularity, where only one f_{step} is performed in each LocalEval. We denote this fine-grained variant as FG^-.

Shown in Figure 4c, we report the total response time of the query, together with its composition *w.r.t.* 1) staleness cost T_w, 2) communication cost T_c, and 3) the cost of granularity adjustment in GAwD and GA, denoted as T_a.

The results show that: 1) GAwD outperforms FG^+ and FG^- by 1.8× and 5.2×, respectively. Comparing with fine-grained FG^-, GAwD greatly reduces communication cost T_c. GAwD takes only 57 rounds of LocalEval to finish the query, while FG^- costs high frequent communications for 14057 rounds. Comparing with coarse-grained FG^+, GAwD saves more than 85% of T_w costs. These verify the effectiveness of dynamic granularity adjustment. 2) GAwD is 1.5× faster than GA. The speedup mainly comes from T_a, which are $0.017s$ and $0.22s$ for GAwD and GA, respectively. The costs of T_c and T_w in GAwD and GA remain comparable. This shows that GAwD efficiently speeds up the GA process while ensuring its effectiveness. 3) GAwD, GA, FG^+ and FG^- has an overall effectiveness φ of 59.7%, 44.2%, 36.0% and 13.3%, which directly correspond to their overall response times. This also empirically verifies our formulation of the computational effectiveness of Equation 1.

Since the superiority of GAwD over GA, fine-grained FG^- and coarse-grained FG^+ is verified, in the rest of the section, we equip Argan with GAP model and GAwD for granularity adjustment by default, unless otherwise stated.

B. Efficiency

Varying the number of CPU cores n from 16 to 128, we next evaluate parallel scalability and scalability of Argan(Figure 6). We evaluate the parallelized algorithms including SSSP, Color, PR, Core and Sim. To verify the effectiveness of GAP, we also

test Argan against Grape and Grape$^+$, as the key differences lie in their parallel models. Specifically, we test Argan under GAP against Grape under BSP and Grape$^+$under AAP. In addition, we also compare with Grape$^+$under AP, which is a special case of AAP, denoted as Grape*.

(1) SSSP. Figures 6a-6c show the overall response time of SSSP over LJ, FS and TW, respectively.

(a) For SSSP, under Argan is on average 2.1, 2.5 and 3.2 (resp. up to 3.9, 5.0 and 6.1) times faster than Grape$^+$, Grape*and Grape, respectively. These verify the effectiveness of the GAP model. Specifically, 1) the asynchronous models including GAP, AAP and AP are faster than BSP because the stragglers caused by global barriers are gone. 2) Compared with AAP and AP, SSSP under GAP employs finer granularity in both communication and computation. This reduces the stale computation, leading to a faster response time. For instance, over TW, stale computation T_w of SSSP under GAP only accounts for less than 20% of the response time, while for AAP and AP it is more than 59% (accounted in total response time, not individually shown).

(b) SSSP under Argan runs faster with a larger n, as expected. When n grows from 16 to 128, the speedup of is 11.7, 3.2 and 1.8 over LJ, FS and TW, respectively.

(2) Color. Shown in Figures 6d-6e are results of Color over HW and LJ, respectively. 1) When n varies from 16 to 128, Color under Argan consistently outperforms its counterparts under Grape$^+$, Grape*and Grape, by 2.2, 2.9 and 2.8 times on average, respectively. GAP in Argan speeds up Color by reducing stale computation up to 12.7 times, compared with AAP in Grape$^+$. 2) When n grows from 16 to 128, Argan speeds up Color by 2.6× and 3.6× over HW and LJ.

(3) PR. Next we test PR over FS, TW and UK. The results are shown in Figures 6f-6h, respectively. We employ the same termination condition as [5], *i.e.,* iterations are terminated when $\sum_{v\in V}|x_v - x_v^*| < 0.001|V|$. Here, x_v (resp. x_v^*) denotes the PR value (resp. its fixedpoint) for vertex v. The fixpoint is obtained offline in advance. We find that 1) PR under Argan is on average 1.7, 2.5 and 2.8 times faster than its counterparts under Grape$^+$, Grape*and Grape, respectively. Recall that the staleness of PR correlates to changes accumulated at border vertices (Eq. 9). With adaptive granularity, GAP communicates whenever sufficient changes have accumulated at the border vertices, leading to less staleness in computation and a faster convergence rate. 2) When n increases from 16 to 128, PR on average speeds up by 2.8, 2.4, 2.5 and 2.9 times for Argan, Grape$^+$, Grape*and Grape, respectively.

(4) Core. The results of Core over HW and FS are shown in Figures 6i and 6j, respectively. We can see that: 1) For Core, Argan consistently beats Grape$^+$, Grape*and Grape: it is on average 40.9% and 31.7% faster than Grape*and Grape, and up to 34.9% faster than Grape$^+$. Similar to PR, GAP boosts Core by adjusting the computation and communication granularity according to changes accumulated in border vertices, thus reducing stale computation. 2) Over HW and FS, Argan speeds up Core by 3.3 and 3.1 times, respectively, when n grows from 16 to 128.

(5) Sim. We also test Sim over DP (Figure 6k). We find that 1) Sim under Argan outperforms its counterpart under Grape by 22.6% on average. 2) For Sim, the performance of Argan is comparable with those of Grape$^+$and Grape*. That is, the gap between GAP and AAP (resp. AP) is less than 6.6% (resp. 10.8%). As mentioned in Section III-C, the staleness function of Sim goes to Category-III, which is constantly zero. Consequently, GAP can not further boost Sim by reducing its stale computation, leading to a narrower gap of speedup against the baselines. 3) When n rises from 16 to 128, Argan speeds up Sim 3.2 times.

(6) Scalability. Fixing n as 128, we test the scalability of Argan over large synthetic graphs. Varying $|G| = |V| + |E|$ from 1.2×10^9 to 6×10^9, the results are shown in Figure 6l. We find the following. 1) All applications take longer when $|G|$ grows, as expected. 2) Argan scales well with increasing graph size. The response time of Argan grows by 7.8, 5.5, 4.8, 5.3 and 6.2 times, for SSSP, Color, PR, Core and Sim, respectively, when $|G|$ grows by 5 times. 3) When $|G|$ is 6 billion, Argan answers queries for SSSP, Color, PR, Core and Sim in 10.0s, 14.4s, 29.9s, 49.7s and 11.3s, respectively.

C. Comparison with Existing Systems

Fixing n as 128, we compare all five applications over Argan against existing graph systems including Grape, Grape$^+$, GraphLab$_{\text{sync}}$, GraphLab$_{\text{async}}$, PowerSwitch and Maiter over TW, FS and UK. The results over TW is shown in Figure 5 (see [15] for more results). We find the following.

(a) For applications with stale computations including SSSP, Color, PR and Core, Argan is at least 38%, 44%, 73% and 17%, faster than all tested competitors, respectively. This verifies the effectiveness and efficiency of Argan.

(b) For applications with no stale computation, namely Sim, the gap between Argan and Grape$^+$is less than 10%.

(c) Among tested graph-centric systems, Argan is on average 1.8 (resp. 2.7) times faster than Grape$^+$(resp. Grape). This verifies the superiority of GAP. In contrast, vertex-centric systems including GraphLab$_{\text{sync}}$, GraphLab$_{\text{async}}$, PowerSwitch and Maiter are generally much slower than Argan.

(d) For Color, the data for GraphLab$_{\text{sync}}$ and PowerSwitch are not available (marked as "NA" in Figure 5), as both fail to converge. For GraphLab$_{\text{sync}}$, under BSP, Color cannot converge under synchronous model due to the oscillation in colors, which is also observed in [12]. As for PowerSwitch, it also begins with BSP, but fails to switch to AP, leading to the same results as GraphLab$_{\text{sync}}$. This highlights the necessity for correctness and convergence guarantees.

Summary. We find the following. 1) Both GAwD and GA effectively speed up the overall computation in Argan by adjusting local granularity tailored for better the computational

effectiveness φ. GAwD also significantly reduces the costs for granularity adjustment by $13\times$.2) The GAP model of Argan boosts the computation significantly when the application incurs stale computation. Comparing with Grape$^+$(AAP), Grape*(AP) and Grape (BSP), Argan (GAP) is on average 2.1, 2.5 and 3.2 times faster for SSSP, 2.2, 2.9 and 4.8 for Color 1.7, 2.5 and 2.8 for PR and 1.3, 1.4 and 1.3 for Core, respectively. Its performance remains comparable with the baselines for applications with no stale computation, like Sim. 3) Argan scales well with large graphs. Over a graph with $|G| = 6$ billion, it responds to queries for SSSP, Color, PR, Core and Sim in 10.0s, 14.4s, 29.9s, 49.7s and 11.3s, respectively. 4) Compared with existing graph systems including Grape, Grape$^+$, GraphLab$_{\text{sync}}$, GraphLab$_{\text{async}}$, PowerSwitch and Maiter, Argan is at least 38%, 44%, 73% and 17%, faster for SSSP, Color, PR and Core, respectively. For Sim, it is still comparable with these systems, *i.e.,* at least 9.4% faster than the fastest competitors. 5) Asynchronous systems may actually be unable to converge in some cases Argan avoids this by providing conditions for asynchronous convergence.

VII. Related Work

The related work is summarized as follows.

Distributed graph systems and abstractions. There has been a significant amount of works on distributed graph processing frameworks (*e.g.,* [1], [3]–[5], [8], [10], [12], [14], see [37] for a survey). These systems follow either vertex-centric or graph-centric paradigms for programming abstraction.

Vertex-centric abstractions recast graph computations into vertex programs, where vertices communicate through messages sent along adjacent edges. Such systems include Pregel [1], GraphLab [4], [10], Giraph [38]. However, these abstractions can be limiting in their expressive power and efficiency. Edge-centric [39], [40] and neighborhood-centric [41], [42] variants have also been proposed, but they still impose too many restrictions. Graph-centric systems, on the other hand, offer greater flexibility, allowing existing sequential optimizations to be directly applied. Examples of such systems include Blogel [9], Grape [3] and Grape$^+$ [14]. However, they are inherently coarse-grained, which can lead to staleness when critical messages are blocked by ongoing batch processing.

To address these issues, Argan introduces the ACE model, which breaks subgraph computation into finer granularity, improving flexibility. The programming model features adjustable granularity, allowing ongoing batch computation to be interrupted by message ingestion or forwarding. Argan's approach provides greater expressive power and efficiency, while avoiding the issues of staleness commonly found in conventional graph-centric systems.

Parallel models and granularity. To boost parallel executions, many parallel models have been developed. BSP [13] model has been widely used (*e.g.,* [1], [3], [9], [38]) due to its simplicity and high throughput. However, its global barriers may cause stragglers when the workload is skewed. In response, asynchronous models such as AP [4], [5], [10] and AAP [14] have been proposed. It is important to note that parallel models have a significant impact on the granularity of a graph system, which in turn affects the overall performance. Coarse-grained parallel models like BSP have high throughput but incur staleness in computation, whereas fine-grained ones like AP suffer from frequent and fragmented communications.

Some parallel models offer more flexible granularity. For instance, GiraphUC [7] and KLA [11] are locally asynchronous but globally synchronous, whereas PowerSwitch [12] automatically switches between the BSP and AP. However, the granularities of these models are still fixed or pre-determined. In contrast, our GAP model is more adaptive and efficient. It decides the granularity based on the algorithm categorization and the runtime information collected by the system. This approach provides the flexibility to adjust the granularity according to the characterization of staleness, leading to better performance and resource utilization.

Parallelization of existing algorithms. Studies have been conducted on adapting existing algorithms for parallelization. Some of these studies focus on the instruction or operator level [43]–[45], breaking dependencies through symbolic and automata analysis. However, this approach requires experienced developers to identify and analyze the dependencies. Grape [3] takes a step further by parallelizing sequential algorithms as a whole. It requires users to plug in a sequential batch algorithm and its incremental version tailored for inter-worker updates. However, incremental graph algorithms are generally sparse and difficult to develop. In contrast, Argan allows users to directly adapt sequential batch algorithms to the ACE model while guaranteeing correctness. This provides a more straightforward and user-friendly approach to parallelizations.

VIII. Conclusion

We have proposed Argan, a parallel graph system that offers efficient adaptive-grained asynchronous executions and easy-to-use abstractions. Argan comes with a simple yet principled programming abstraction that allows users to adapt a class of existing batch sequential algorithms to the model. Additionally, its GAP model offers extended expressive power by enabling the granularity of computation and communication to be adjusted adaptively at runtime. We have formally defined the problem of granularity adjustment and proposed a solution based on runtime information and the categorization of graph algorithms. Our experimental results confirm the effectiveness of GAP and the efficiency of Argan compared to existing systems. Future work might expand the system to support graphs beyond static simple graphs.

Acknowledgement

This research / project is supported by the National Research Foundation, Singapore under its Strategic Capability Research Centres Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore. Yue Wang is partially supported by China NSFC(No.62002235).

REFERENCES

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[2] "Apache Giraph," *https://giraph.apache.org/*.

[3] W. Fan, W. Yu, J. Xu, J. Zhou, X. Luo, Q. Yin, P. Lu, Y. Cao, and R. Xu, "Parallelizing sequential graph computations," *ACM Transactions on Database Systems (TODS)*, vol. 43, no. 4, pp. 1–39, 2018.

[4] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[5] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 8, pp. 2091–2100, 2013.

[6] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy." in *CIDR*, vol. 13, 2013, pp. 3–6.

[7] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.

[8] W. Fan, P. Lu, W. Yu, J. Xu, Q. Yin, X. Luo, J. Zhou, and R. Jin, "Adaptive asynchronous parallelization of graph algorithms," *ACM Transactions on Database Systems (TODS)*, vol. 45, no. 2, pp. 1–45, 2020.

[9] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1981–1992, 2014.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "{PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs," in *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, 2012, pp. 17–30.

[11] A. Fidel, N. M. Amato, L. Rauchwerger *et al.*, "Kla: A new algorithmic paradigm for parallel graph computations," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2014, pp. 27–38.

[12] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "Sync or async: Time to fuse for distributed graph-parallel computation," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 194–204, 2015.

[13] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[14] W. Fan, P. Lu, X. Luo, J. Xu, Q. Yin, W. Yu, and R. Xu, "Adaptive asynchronous parallelization of graph algorithms," in *SIGMOD*, 2018.

[15] R. Xu, Y. Wang, and X. Xiao, "Graph computation with adaptive granularity (full version with appendix)," 2024. [Online]. Available: https://figshare.com/articles/conference_contribution/Graph_Computation_with_Adaptive_Granularity_full_version_with_appendix_/25314589

[16] E. W. Dijkstra, "A note on two problems in connexion with graphs," in *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, 2022, pp. 287–290.

[17] S. B. Seidman, "Network structure and minimum degree," *Social networks*, vol. 5, no. 3, pp. 269–287, 1983.

[18] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *IEEE Transactions on parallel and distributed systems*, vol. 24, no. 2, pp. 288–300, 2012.

[19] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke, "Computing simulations on finite and infinite graphs," in *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE, 1995, pp. 453–462.

[20] W. Fan, X. Wang, Y. Wu, and D. Deng, "Distributed graph simulation: Impossibility and possibility," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1083–1094, 2014.

[21] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[22] A. E. Sariyuce, C. Seshadhri, and A. Pinar, "Local algorithms for hierarchical dense subgraph discovery," *arXiv preprint arXiv:1704.00386*, 2017.

[23] J. Nesetril, E. Milková, and H. Nesetrilová, "Otakar boruvka on minimum spanning tree problem," *Discrete Mathematics*, vol. 233, no. 1-3, pp. 3–36, 2001.

[24] G. Jeh and J. Widom, "Simrank: a measure of structural-context similarity," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2002, pp. 538–543.

[25] L. Lü, T. Zhou, Q.-M. Zhang, and H. E. Stanley, "The h-index of a network node and its relation to degree and coreness," *Nature communications*, vol. 7, no. 1, pp. 1–7, 2016.

[26] R. Bellman, "On a routing problem," *Quarterly of applied mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[27] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, Z. Zheng, B. Zhang, Y. Cao, and C. Tian, "Parallelizing Sequential Graph Computations," in *SIGMOD*, 2017.

[28] W. Fan, C. Tian, R. Xu, Q. Yin, W. Yu, and J. Zhou, "Incrementalizing graph algorithms," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 459–471.

[29] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: https://networkrepository.com

[30] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "Dbpedia: A nucleus for a web of open data," in *international semantic web conference*. Springer, 2007, pp. 722–735.

[31] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, 2012, pp. 1–8.

[32] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 591–600.

[33] P. Boldi, M. Santini, and S. Vigna, "A large time-aware graph," *SIGIR Forum*, vol. 42, no. 2, pp. 33–38, 2008.

[34] D. J. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, pp. 85–86, 1967.

[35] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri, "Partitioning trillion-edge graphs in minutes," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 646–655.

[36] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm, "Netgauge: A network performance measurement framework," in *High Performance Computing and Communications: Third International Conference, HPCC 2007, Houston, USA, September 26-28, 2007. Proceedings 3*. Springer, 2007, pp. 659–671.

[37] H. Jin, H. Qi, J. Zhao, X. Jiang, Y. Huang, C. Gui, Q. Wang, X. Shen, Y. Zhang, A. Hu *et al.*, "Software systems implementation and domain-specific architectures towards graph analytics," *Intelligent Computing*, vol. 2022, 2022.

[38] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," *Proceedings of the Hadoop Summit. Santa Clara*, vol. 11, no. 3, pp. 5–9, 2011.

[39] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 472–488.

[40] H. Zhu, L. He, M. Leeke, and R. Mao, "Wolfgraph: The edge-centric graph processing on gpu," *Future Generation Computer Systems*, vol. 111, pp. 552–569, 2020.

[41] S. Ko and W.-S. Han, "Turbograph++ a scalable and fast graph analytics system," in *Proceedings of the 2018 international conference on management of data*, 2018, pp. 395–410.

[42] A. Quamar, A. Deshpande, and J. Lin, "Nscale: neighborhood-centric large-scale graph analytics in the cloud," *The VLDB Journal*, vol. 25, pp. 125–150, 2016.

[43] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo *et al.*, "The tao of parallelism in algorithms," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 12–25.

[44] V. Raychev, M. Musuvathi, and T. Mytkowicz, "Parallelizing user-defined aggregations using symbolic execution," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 153–167.

[45] Y. Zhuo, J. Chen, Q. Luo, Y. Wang, H. Yang, D. Qian, and X. Qian, "Symplegraph: distributed graph processing with precise loop-carried dependency guarantee," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 592–607.

[46] L. G. Valiant, "General purpose parallel architectures," in *Algorithms and Complexity*. Elsevier, 1990, pp. 943–971.

[47] D. Marx, "Graph colouring problems and their applications in scheduling," *Periodica Polytechnica Electrical Engineering (Archives)*, vol. 48, no. 1-2, pp. 11–16, 2004.

[48] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified np-complete problems," in *Proceedings of the sixth annual ACM symposium on Theory of computing*, 1974, pp. 47–63.
[49] D. S. Johnson, "The np-completeness column: an ongoing guide," *Journal of algorithms*, vol. 6, no. 3, pp. 434–451, 1985.
[50] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.
[51] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu, "Graph pattern matching: From intractable to polynomial time," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 264–275, 2010.
[52] J. A. Rico-Gallego, J. C. Díaz-Martín, R. R. Manumachu, and A. L. Lastovetsky, "A survey of communication performance models for high-performance computing," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.