Adaptive Asynchronous Parallelization of Graph Algorithms

WENFEI FAN, University of Edinburgh & Beihang University & SICS, Shenzhen University
PING LU, BDBC, Beihang University
WENYUAN YU, JINGBO XU, QIANG YIN*, XIAOJIAN LUO, and JINGREN ZHOU, Alibaba Group
RUOCHUN JIN, University of Edinburgh

This paper proposes an Adaptive Asynchronous Parallel (AAP) model for graph computations. As opposed to Bulk Synchronous Parallel (BSP) and Asynchronous Parallel (AP) models, AAP reduces both stragglers and stale computations by dynamically adjusting relative progress of workers. We show that BSP, AP and Stale Synchronous Parallel model (SSP) are special cases of AAP. Better yet, AAP optimizes parallel processing by adaptively switching among these models at different stages of a single execution. Moreover, employing the programming model of GRAPE, AAP aims to parallelize existing sequential algorithms based on simultaneous fixpoint computation with partial and incremental evaluation. Under a monotone condition, AAP guarantees to converge at correct answers if the sequential algorithms are correct. Furthermore, we show that AAP can optimally simulate MapReduce, PRAM, BSP, AP and SSP. Using real-life and synthetic graphs, we experimentally verify that AAP outperforms BSP, AP and SSP for a variety of graph computations.

CCS Concepts: • **Information systems** → **Database management system engines**; **Parallel and distributed DBMSs**;

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Graph computations; parallel graph query engines; parallelizing sequential algorithms; convergence; simulation

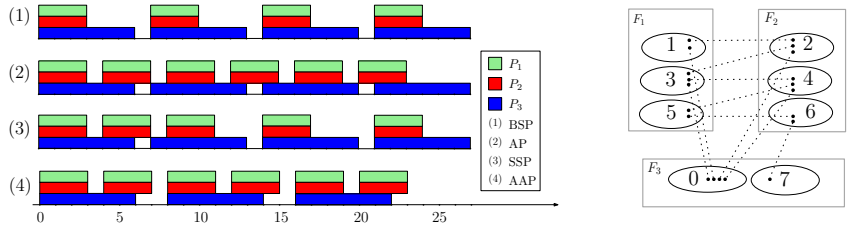
ACM Reference format:
Wenfei Fan, Ping Lu, Wenyuan Yu, Jingbo Xu, Qiang Yin, Xiaojian Luo, Jingren Zhou, and Ruochun Jin. 2020. Adaptive Asynchronous Parallelization of Graph Algorithms. *ACM Trans. Datab. Syst.* 1, 1, Article 1 (January 2020), 44 pages.
DOI: 10.1145/3397491

1 INTRODUCTION

Bulk Synchronous Parallel (BSP) model [75] has been adopted by graph systems, *e.g.,* Pregel [58] and GRAPE [38]. Under BSP, iterative computation is separated into supersteps, and messages from one superstep can only be accessible in the next one. The synchronous nature of BSP simplifies the analysis of parallel algorithms. However, its global synchronization barriers lead to stragglers, *i.e.,* some workers take substantially longer than the others. Since workers converge asymmetrically, the speed of each superstep is limited to that of the slowest worker.

*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 ACM. 0362-5915/2020/1-ART1 $15.00
DOI: 10.1145/3397491



(a) BSP, AP, SSP and AAP (b) A CC example

Fig. 1. Runs under different parallel models

To reduce stragglers, Asynchronous Parallel (AP) model has been employed by, *e.g.,* GraphLab [43, 57] and Maiter [91]. Under AP, a worker has immediate access to incoming messages. Fast workers can move ahead, without waiting for stragglers. However, AP may incur excessive stale computations, *i.e.,* processes triggered by messages that soon become stale due to more up-to-date messages. Stale computations are often redundant and increase unnecessary computation and communication (data shipment) costs. Moreover, it is observed that AP makes it hard to write, debug and analyze programs [80], and complicates the consistency analysis (see [85] for a recent survey).

A recent study shows that neither AP nor BSP consistently outperforms the other for different algorithms, input graphs and cluster scales [83]. For many graph algorithms, different stages in a single execution demand *different models* for optimal performance.

To rectify the problems, revisions of BSP and AP have been studied, notably Stale Synchronous Parallel (SSP) [47], a relaxed consistency protocol of ASPIRE [78] and a hybrid model Hsync [83]. SSP relaxes BSP by allowing fastest workers to outpace the slowest ones by a fixed number c of steps (known as bounded staleness). It reduces stragglers, but incurs redundant stale computations. Like SSP, ASPIRE adopts a uniform staleness threshold t; unlike SSP, it employs a "best effort refresh" policy to fetch the latest updates when "stale-hit" occurs. Hsync suggests to switch between AP and BSP, but it requires us to predict switching points and incurs switching costs.

Is it possible to have a simple parallel model that inherits the benefits of BSP and AP, and reduces both stragglers and stale computations, without explicitly switching between the two? Better still, can the model retain the programming simplicity of BSP, ensure consistency, and guarantee correct convergence of graph computations under a general condition?

AAP. To answer the questions, we propose a parallel model, referred to as Adaptive Asynchronous Parallel (AAP) model. Without global synchronization barriers, AAP is essentially asynchronous. As opposed to BSP and AP, each worker under AAP maintains parameters to measure (a) its progress relative to other workers, and (b) changes accumulated by messages (staleness). Each worker has immediate access to incoming messages, and decides whether to start the next round of computation based on its own parameters. In contrast to SSP, each worker dynamically adjusts its parameters based on its relative progress and message staleness, instead of using a fixed bound.

Example 1.1. Consider a computation task being conducted at three workers, where workers P_1 and P_2 take 3 time units to do one round of computation, and P_3 takes 6 units; it takes 1 unit to pass messages. This is carried out under different models as follows, as shown in Fig. 1a (it depicts runs for computing connected components shown in Fig. 1b, to be elaborated in Example 3.2).

(1) BSP. As depicted in Fig. 1a (1), worker P_3 takes twice as long as P_1 and P_2, and is a straggler. Due to its global synchronization, each superstep takes 6 time units, the speed of the slowest P_3.

(2) AP. AP allows a worker to start the next round as soon as its message buffer is not empty. However, it comes with redundant stale computation. As shown in Fig. 1a (2), at clock time 7,

the second round of P_3 can only use messages from the first round of P_1 and P_2. This round of computation at P_3 becomes stale at time 8, when the latest updates from P_1 and P_2 arrive. As will be seen later, a large part of the computations of faster P_1 and P_2 is also redundant.

(3) SSP. Consider bounded staleness of 1, *i.e.,* the fastest worker can outpace the slowest one by at most 1 round. As shown in Fig. 1a (3), P_1 and P_2 are not blocked by the straggler in the first 3 rounds. However, like AP, the second round of P_3 is stale. Moreover, P_1 and P_2 cannot start their rounds 4 and 5 until P_3 finishes its rounds 2 and 3, respectively, due to the bounded staleness condition. As a result, P_1, P_2 and P_3 behave like in BSP model after clock time 14.

(4) AAP. AAP allows a worker to accumulate changes and decide when to start the next round based on the progress of others. As shown in Fig. 1a (4), after P_3 finishes one round of computation at clock time 6, it may start the next round at time 8, at which point the latest changes from P_1 and P_2 are available. As opposed to AP, AAP reduces redundant stale computation. This also helps us mitigate the straggler problem, as P_3 can converge in less rounds by utilizing the latest updates.

Remark. Observe the following about the example.

(1) When stragglers are forced to wait and accumulate messages as in AAP, the stragglers may converge in less rounds, and the overall performance can be substantially improved.

(2) To simplify the discussion, we assume no overlap between computation and communication above. Nonetheless, in the presence of overlap, it is easy to show that the behaviors of these models actually resemble their counterparts depicted in Figure 1a. □

AAP reduces stragglers by not blocking fast workers. This is especially helpful when the computation is CPU-intensive and skewed, when an evenly partitioned graph becomes skewed due to updates, or when we cannot afford evenly partitioning a large graph due to partition cost. Moreover, AAP activates a worker only after it receives sufficient latest updates and thus reduces stale computations. This allows us to reallocate resources to useful computations via workload adjustments.

In addition, AAP differs from previous models in the following.

(1) Model switch. BSP, AP and SSP are special cases of AAP with fixed parameters. Hence AAP can naturally switch among these models at different stages of the same execution, without asking for explicit switching points or incurring the switching costs. As will be seen later, AAP is more flexible: under AAP, workers with similar speed are automatically grouped together after a few rounds of computation; it adopts BSP within a group, and AP across different groups.

(2) Programming paradigm. AAP works with the programming model of GRAPE [38]. It allows users to extend existing sequential (single-machine) graph algorithms with message declarations, and parallelizes the algorithms across a cluster of machines. It employs aggregate functions to resolve conflicts raised by updates from different workers, without worrying about race conditions or requiring extra efforts to enforce consistency by using, *e.g.,* locks [85].

(3) Convergence guarantees. AAP is modeled as a simultaneous fixpoint computation. Based on this we develop one of the first conditions under which AAP parallelization of sequential algorithms guarantees (a) convergence at correct answers, and (b) the Church-Rosser property, *i.e.,* all asynchronous runs converge at the same result, as long as the sequential algorithms are correct.

(4) Expressive power. Despite its simplicity, AAP can optimally simulate MapReduce [31], PRAM (Parallel Random Access Machine) [76], BSP, AP and SSP. That is, algorithms developed for these models can be migrated to AAP without increasing the complexity bound.

System	PageRank on Friendster		CC on traffic	
	Time(s)	Communication(GB)	Time(s)	Communication(GB)
Giraph	6117.7	767.3	4707.0	108.6
$\text{GraphLab}_{\text{sync}}$	99.5	138.0	1792.2	471.4
$\text{GraphLab}_{\text{async}}$	200.1	333.0	504.1	1024.2
GiraphUC	9991.6	3616.5	2081.1	119.8
Maiter	199.9	134.3	347.3	1.94
Husky	141.4	201.8	178.6	16.1
Galois	18.7	11.8	25.8	4.1
Pregel+	61.9	17.0	113.5	3.8
PowerSwitch	85.1	39.9	386.2	524.4
TDataflow	26.12	218.3	7.89	0.43
GRAPE+	21.2	16.2	2.8	0.03

Table 1. PageRank and CC with 192 workers

(5) Performance. AAP outperforms BSP, AP and SSP for a variety of graph computations. As an example, for PageRank [23] on social network Friendster [6] and connected components (CC) on transportation network traffic [3] with 192 workers, Table 1 shows the performance of (a) Giraph [8] (an open-source version of Pregel), GraphLab [57], Husky [88], Galois [29, 64] and Pregel+ [87] under BSP, (b) GraphLab, Maiter [91] and TDataflow (Timely-Dataflow) [9, 62] under AP, (c) GiraphUC [45] under BAP, (d) PowerSwtich [83] under Hsync, (e) GRAPE+, an extension of GRAPE by supporting AAP. We can see that GRAPE+ performs better than or at least comparably to the state-of-the-art systems in both response time and data shipment (communication cost).

Contributions & organization. This paper introduces AAP, from foundations to implementation.

(1) Programming model (Section 2). We present the programming model of GRAPE, and show that it works well with AAP, to parallelize existing sequential graph algorithms.

(2) AAP (Section 3). We propose AAP. We show that AAP subsumes BSP, AP and SSP as special cases, and reduces both stragglers and stale computations by adjusting relative progress of workers.

(3) Foundation (Section 4). We model AAP as a simultaneous fixpoint computation with partial evaluation and incremental computation. We provide a condition under which AAP guarantees convergence at correct answers, *i.e.,* termination and the Church-Rosser property. We also show that AAP can optimally simulate MapReduce, PRAM, BSP, AP and SSP.

(4) AAP *programming* (Section 5). As case studies, we show that a variety of graph computations can be easily carried out by AAP. These include single-source shortest paths (SSSP), connected components (CC), collaborative filtering (CF) and PageRank (PageRank).

(5) Runtime estimation (Section 6). To dynamically adjust the relative progress of workers under AAP, we estimate runtime of workers and message arrival rate based on machine learning techniques. In particular, we propose a runtime estimation model based on random forest regressions [49].

(6) Implementation (Section 7). As proof of concept, we develop GRAPE+ by extending GRAPE [37] from BSP to AAP. We outline the implementation of GRAPE+.

(7) Experiments (Section 8). Using real-life and synthetic graphs, we evaluate the performance of GRAPE+, compared with the systems listed in Table 1, and Petuum [84], a parameter server under SSP. Over real-life graphs and with 192 workers, we find the following. (a) GRAPE+ is at least 1.1, 1.9, 1.7 and 4.6 times faster than these systems for SSSP, CC, PageRank and CF on

real-life graphs on average, respectively, up to 4127, 1635, 446 and 7.6 times. On average, (b) AAP outperforms BSP, AP and SSP by 4.8, 1.7 and 1.8 times in response time, up to 27.4, 3.2 and 5.0 times, respectively. Over larger synthetic graphs with 10 billion edges, it is 4.3, 14.7 and 4.7 times faster, respectively. (c) GRAPE+ is on average 2.4, 2.7, 2.3 and 1.7 times faster for SSSP, CC, PageRank and CF, respectively, when the number of workers varies from 64 to 192. (d) Our prediction method estimates the runtime of the algorithms fairly well, and our dynamic adjustment of relative progress is efficient and effective.

Related work. This paper extends its conference version [36] as follows. (1) We provide detailed proofs of the results of the paper (Section 4). (2) We propose a runtime estimation model based on random forest regressions in a new section (Section 6), for AAP to dynamically adjust the relative progress of workers. (3) We conduct new experiments to verify the accuracy and efficiency of our runtime and message arrival rate estimation, and compare with more graph systems (Section 8).

Several parallel models have been studied for graphs. PRAM [76] supports parallel RAM access with shared memory, and is not for the shared-nothing architecture that is widely used nowadays. MapReduce [31] is adopted by, *e.g.,* GraphX [44]. However, it is not very efficient for iterative graph computations due to its blocking and I/O costs. BSP [75] with vertex-centric programming works better for graphs as shown by [58]. However, it suffers from stragglers due to its global synchronization. As remarked earlier, AP reduces stragglers, but it comes with redundant stale computations. It also bears with race conditions and their locking/unblocking costs, and complicates the convergence analysis (see Section 4.1) and programming [80].

SSP [47] promotes bounded staleness for machine learning. Maiter [91] reduces stragglers by accumulating updates, and supports prioritized asynchronous execution. BAP model (*barrierless asynchronous parallel*) [45] reduces global barriers and local messages by using light-weighted local barriers. As remarked earlier, Hsync proposes to switch between AP and BSP [83]. ASPIRE [78] revises SSP with a relaxed consistency protocol to cope with staleness in asynchronous computations.

Several graph systems under these models are in place, *e.g.,* Pregel [58], GPS [69], Giraph++ [74], GRAPE [38], Gemini [93] and Galois [29, 64] under BSP; GraphLab [43, 57], Maiter [91], GRACE [80] and TDataflow [9, 62] under (revised) AP; PowerLyra [25] and Husky [88] under both BSP and AP; parameter servers [47, 56, 81, 84] and Tornado [71] under SSP; GiraphUC [45] under BAP; and PowerSwitch under Hsync [83]. Blogel [86] works like AP within blocks, and in BSP across blocks. Most of these are vertex-centric. While Giraph++ and Blogel [74] process blocks [74], they inherit vertex-centric programming by treating blocks as vertices. In contrast, GRAPE parallelizes sequential graph algorithms as a whole.

AAP differs from the prior models in the following.

(1) AAP reduces (a) stragglers of BSP via asynchronous message passing, and (b) redundant stale computations of AP by imposing a bound (delay stretch), for workers to wait and accumulate updates. AAP is not vertex-centric. Based on fixpoint computation, it works well with the programming paradigm of GRAPE, which simplifies the convergence and consistency analyses of AP.

(2) SSP mainly targets machine learning, with different correctness criteria. (a) SSP promotes the idea of "letting slow workers catch up". In contrast, AAP shows that the performance can be improved when stragglers are forced to wait and accumulate messages instead of to catch up. (b) SSP adopts an "upper bound" on relative progress of workers, while AAP reduces stale computations by enforcing a "lower bound" on accumulated messages. (c) SSP uses a predefined constant as a uniform bound for all workers, while AAP allows each worker to keep track of its own relative

progress and staleness, dynamically adjust its bounds, and decide when to trigger its next round of computation. (d) Under AAP, workers with similar speed are grouped together and follow BSP within a group, and AP is adopted across the worker groups. It is unclear how SSP can achieve these. (e) Bounded staleness is not needed by SSSP, CC and PageRank as will be seen in Section 5.3.

(3) Unlike SSP, ASPIRE adopts a "best effort refresh" policy to fetch the latest updates when a "stale-hit" occurs. Like SSP, it uses a predefined uniform bounded staleness. In contrast, (a) AAP allows each worker to decide when to trigger the next round based on its own dynamic parameters, to reduce both stragglers and stale computations; (b) AAP is developed for graph-centric programming and shard-nothing architectures, while ASPIRE is for vertex-centric programming and distributed shared-memory systems; and (c) AAP employs techniques quite different from ASPIRE, *e.g.,* machine learning methods to predict parameters and aggregate functions to resolve conflicts.

(4) Similar to Maiter, AAP aggregates changes accumulated. As opposed to Maiter, it reduces redundant computations by (a) imposing a delay stretch on workers, to adjust their relative progress, (b) dynamically adjusting bounds to optimize performance, and (c) combining incremental evaluation with accumulative computation. AAP operates on graph fragments, while Maiter is vertex-centric.

(5) Both BAP and AAP reduce unnecessary messages. However, AAP achieves this by operating on fragments (blocks), and moreover, optimizes performance by adjusting relative progress of workers.

(6) Closer to AAP is Hsync, and PowerSwitch has performance close to GRAPE+. As opposed to Hsync, AAP does not demand complete switch from one mode to another. Instead, each worker may decide its own "mode" based on its relative progress. As will be seen in Sections 3 and 8, workers with similar speed are grouped together and follow BSP within a group, and AP is adopted among the worker groups; these are beyond Hsync. Moreover, the parameters are adjusted dynamically, and hence AAP does not have to predict switching points and pay the price of switching cost.

Prior work to mitigate the straggler problem includes dynamic repartitioning [18, 51, 59], work stealing [14, 21], shedding [33], LATE [89], and fine-grained partition [26]. AAP is complementary to these methods, reducing stragglers and stale computation by adjusting relative progress of workers.

2 THE PROGRAMMING MODEL

AAP adopts the programming model of [38], which we review next. As will be seen in Section 3, AAP is able to parallelize sequential graph algorithms following the programming paradigm of GRAPE. That is, the asynchronous model does not make programming harder than GRAPE.

Graph partition. AAP supports data-partitioned parallelism. It works on graphs that are partitioned into smaller fragments and distributed across a cluster of workers.

Consider graphs $G = (V, E, L)$, directed or undirected, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a set of edges; and (3) each node v in V (resp. edge $e \in E$) is labeled with $L(v)$ (resp. $L(e)$) indicating its content, as found in property graphs.

Given a natural number m, a strategy $\mathcal{P}$ partitions G into *fragments* $\mathcal{F} = (F_1, \ldots, F_m)$ such that each $F_i = (V_i, E_i, L_i)$ is a subgraph of G, $V = \bigcup_{i\in[1,m]} V_i$, and $E = \bigcup_{i\in[1,m]} E_i$. Here F_i is called a *subgraph of* G if $V_i \subseteq V$, $E_i \subseteq E$, and for each node $v \in V_i$ (resp. edge $e \in E_i$), $L_i(v) = L(v)$ (resp. $L_i(e) = L(e)$). Note that F_i is a graph itself, but is not necessarily an induced subgraph of G.

AAP allows users to pick an edge-cut [15] or vertex-cut [52] strategy $\mathcal{P}$ to partition a graph G. When $\mathcal{P}$ is edge-cut, a cut edge from F_i to F_j has a copy in both F_i and F_j. Denote by

(a) $F_i.I$ (resp. $F_i.O'$) the set of nodes $v \in V_i$ such that there exists an edge (v', v) (resp. (v, v')) with a node v' in F_j ($i \neq j$); and
(b) $F_i.O$ (resp. $F_i.I'$) the set of nodes v' in some F_j ($i \neq j$) such that there exists an edge (v, v') (resp. (v', v)) *with* $v \in V_i$.

We refer to the nodes in $F_i.I \cup F_i.O'$ as the border nodes of F_i *w.r.t.* $\mathcal{P}$.

For vertex-cut, border nodes are those that have copies in different fragments. In general, a node v is a border node if v has an adjacent edge across two fragments, or a copy in another fragment.

Programming. Using our familiar terms, we refer to a graph computation problem as a class $\mathcal{Q}$ of graph queries, and instances of the problem as queries of $\mathcal{Q}$. Following GRAPE [38], to answer queries $Q \in \mathcal{Q}$ under AAP, one only needs to specify three functions.

(1) PEval: a sequential (*i.e.,* single-machine) algorithm for $\mathcal{Q}$ that given a query $Q \in \mathcal{Q}$ and a graph G, computes the answer $Q(G)$ to Q in G.
(2) IncEval: a sequential incremental algorithm for $\mathcal{Q}$ that given Q, G, $Q(G)$ and updates ΔG to G, computes updates ΔO to the old output $Q(G)$ such that $Q(G \oplus \Delta G) = Q(G) \oplus \Delta O$, where $G \oplus \Delta G$ denotes G updated by ΔG [68]. Here IncEval only needs to deal with changes ΔG to update parameters (status variables) to be defined shortly.
(3) Assemble: a function that collects partial answers computed locally at each worker by PEval and IncEval, and assembles the partial results into a complete answer $Q(G)$.

Taken together, the three functions are referred to as *a* PIE *program for* $\mathcal{Q}$ (PEval, IncEval and Assemble). PEval and IncEval can be *existing sequential* (incremental) algorithms for $\mathcal{Q}$, which are to operate on a fragment F_i of G partitioned via a strategy $\mathcal{P}$.

The only additions are the following declarations in PEval.

(a) Update parameters. PEval declares *status variables* $\bar{x}$ for a set C_i in a fragment F_i, to store contents of F_i or partial results of a computation. Here C_i is a set of nodes and edges within d-hops of the nodes in $F_i.I \cup F_i.O'$ for an integer d. In particular, when $d = 0$, C_i is $F_i.I \cup F_i.O'$.

We denote by $C_i.\bar{x}$ the set of *update parameters* of F_i, which consists of status variables associated with the nodes and edges in C_i. As will be seen in Section 3, the variables in $C_i.\bar{x}$ are the candidates to be updated by incremental steps that are carried out by IncEval.

(b) Aggregate functions. PEval also specifies an aggregate function f_{aggr}, *e.g.,* min and max, to resolve conflicts when multiple workers attempt to assign different values to the same update parameter.

These are specified in PEval and are shared by IncEval.

Example 2.1. Consider *graph connectivity* (CC). Given an undirected graph $G = (V, E, L)$, a subgraph G_s of G is a *connected component* of G if (a) it is connected, *i.e.,* for any two nodes v and v' in G_s, there exists a path between v and v', and (b) it is maximum, *i.e.,* adding any node of G to G_s makes the induced subgraph disconnected. Note that for each G, CC has a single query Q, to compute all connected components of G, denoted by $Q(G)$. CC is in $O(|G|)$ time [17].

AAP parallelizes CC with the same PEval and IncEval of GRAPE [38]. More specifically, a PIE program ρ for CC is given as follows.

(1) As shown in Fig. 2, at each fragment F_i, PEval uses a sequential CC algorithm (Depth-First Search, DFS) to compute the local connected components and create their ids, except that it declares the following (underlined in Fig. 2): (a) for each node $v \in V_i$, an integer variable v.cid, initially its node id v.id; (b) $F_i.O$ as the candidate set C_i, and $C_i.\bar{x} = \{v.\text{cid} \mid v \in F_i.O\}$ as the update parameters;

Input: A fragment $F_i(V_i, E_i, L_i)$.
Output: A set $Q(F_i)$ consists of current v.cid for $v \in V_i$.
Message preamble: /*candidate set C_i is $F_i.O$*/
For each node $v \in V_i$, a variable v.cid;
1. $\mathbb{C} := \mathsf{DFS}(F_i)$; /* find local connective components by DFS */
2. **for each** $C \in \mathbb{C}$ **do**
3. create a new "root" node v_c;
4. v_c.cid := min$\{v.\mathsf{id} \mid v \in C\}$;
5. **for each** $v \in C$ **do**
6. link v to v_c; v.root := v_c; v.cid := v_c.cid;
7. $Q(F_i) := \{v.\mathsf{cid} \mid v \in V_i\}$;

Message segment: $M_{(i,j)} := \{v.\mathsf{cid} \mid v \in F_i.O \cap F_j.I, i \neq j\}$;
$f_{\mathsf{aggr}}(v) := \mathsf{min}(v.\mathsf{cid})$;

Fig. 2. PEval for CC under AAP

Input: A fragment $F_i(V_i, E_i, L_i)$, partial result $Q(F_i)$, and message M_i.
Output: New output $Q(F_i \oplus M_i)$
1. $\Delta := \emptyset$;
2. **for each** v^{in}.cid $\in M_i$ **do** /* use min as f_{aggr} */
3. v.cid := min$\{v.\mathsf{cid}, v^{\mathsf{in}}.\mathsf{cid}\}$;
4. $v_c := v$.root;
5. **if** v.cid $< v_c$.cid **then**
6. v_c.cid := v.cid; $\Delta := \Delta \cup \{v_c\}$;
7. **for each** $v_c \in \Delta$ **do** /* propagate the change */
8. **for each** $v \in F_i.O$ that linked to v_c **do**
9. v.cid := v_c.cid;
10. $Q(F_i) := \{v.\mathsf{cid} \mid v \in V_i\}$;

Message segment: $M_{(i,j)} := \{v.\mathsf{cid} \mid v \in F_i.O \cap F_j.I, v.\mathsf{cid}$ decreased$\}$;

Fig. 3. IncEval for CC under AAP

and (c) min as the aggregate function f_{aggr}; that is, if there exist multiple values to be assigned to the same v.cid, the smallest value is taken by the linear order on integers.

For each local connected component C, (a) PEval creates a "root" node v_c carrying the minimum node id in C as v_c.cid, and (b) links all the nodes in C to v_c, and sets their cid as v_c.cid. These can be done in one pass of the edges in fragment F_i via DFS.

(2) Given a set M_i of changed cids of border nodes, IncEval incrementally updates local components in fragment F_i, by "merging" components when possible. As shown in Fig. 3, by using min as f_{aggr}, IncEval (a) updates the cid of each border node to the minimum one; and (b) propagates the change first to its root v_c and then in turn to all border nodes linked to v_c.

(3) Assemble first updates the cid of each node to the cid of its linked root. It then merges all the nodes having the same cids in a single bucket, and returns all buckets as connected components. □

We remark the following about the programming paradigm.

(1) There are methods for incrementalizing graph algorithms, to deduce incremental algorithms from batch algorithms [13, 90]. Moreover, one can get IncEval by revising a batch algorithm in response to changes to update parameters, *e.g.,* the ones for CC (Example 3.2) and PageRank (Section 5.3).

(2) We adopt edge-cut in the sequel unless stated otherwise; but AAP works with other partition strategies. Indeed, as will be seen in Section 4, the correctness of asynchronous runs under AAP remains intact under the conditions given there, regardless of partitioning strategies used. Nonetheless, different strategies may yield partitions with various degrees of skewness and stragglers, which have an impact on the performance of AAP, as will be seen in Section 8.

(3) The programming model aims to facilitate users to develop parallel programs, especially for those who are more familiar with conventional sequential programming. There is no need to revise the logic of the existing algorithms, and hence it reduces "the total cost of ownership". This said, programming with AAP still requires users to specify update parameters and aggregate function.

3 THE AAP MODEL

We next present the adaptive asynchronous parallel model (AAP).

Setting. Adopting the programming model of GRAPE (Section 2), to answer a class $\mathcal{Q}$ of queries on a graph G, AAP takes as input a PIE program ρ (*i.e.,* PEval, IncEval, Assemble) for $\mathcal{Q}$, and a partition strategy $\mathcal{P}$. It partitions G into fragments $(F_1, \ldots, F_m)$ using $\mathcal{P}$, such that each fragment F_i resides at a virtual worker P_i ($i \in [1, m]$). It works with a master P_0 and n shared-nothing physical workers $(P_1, \ldots, P_n)$, where $n \leq m$, *i.e.,* multiple virtual workers may be mapped to the same physical worker and share memory. Graph G is partitioned once for all queries $Q \in \mathcal{Q}$ posed on G.

As remarked earlier, PEval and IncEval are (existing) sequential batch and incremental algorithms for $\mathcal{Q}$, respectively, except that PEval also declares update parameters $C_i.\bar{x}$, and defines an aggregate function f_{aggr}. At each worker P_i, (a) PEval computes $Q(F_i)$ on fragment F_i, and (b) IncEval takes F_i and updates M_i to $C_i.\bar{x}$ as input, and computes updates ΔO_i to $Q(F_i)$ such that $Q(F_i \oplus M_i)$=$Q(F_i) \oplus \Delta O_i$. We refer to each invocation of PEval or IncEval as *one round of computation* at worker P_i.

Message passing. After each round of computation at worker P_i, P_i collects update parameters of $C_i.\bar{x}$ with *changed values* in a set $\Delta C_i.\bar{x}$. It groups $\Delta C_i.\bar{x}$ into $M_{(i,j)}$ for $j \in [1, m]$ and $j \neq i$, where $M_{(i,j)}$ includes $v.x \in \Delta C_i.\bar{x}$ for $v \in C_j$, *i.e.,* v also resides in fragment F_j. That is, $M_{(i,j)}$ includes changes of $\Delta C_i.\bar{x}$ to the update parameters $C_j.\bar{x}$ of F_j. It sends $M_{(i,j)}$ as a message to worker P_j.

Messages $M_{(i,j)}$ are referred to as *designated messages* in [38]. To efficiently determine the destination of the designated messages, each worker P_i maintains the following:

(1) an index I_i that given a border node v, retrieves the set of $j \in [1, m]$ such that $v \in F_j.I' \cup F_j.O$ and $i \neq j$, *i.e.,* where v resides; it is deduced from the partition strategy $\mathcal{P}$; and
(2) a buffer $\mathbb{B}_{\bar{x}_i}$, to keep track of messages from other workers.

As opposed to GRAPE that adopts BSP, AAP is asynchronous in nature. (1) AAP adopts (a) point-to-point communication: a worker P_i can send a message $M_{(i,j)}$ directly to worker P_j, and (b) *push-based* message passing: P_i sends $M_{(i,j)}$ to worker P_j as soon as $M_{(i,j)}$ is available, regardless of the progress at other workers. A worker P_j can receive messages $M_{(i,j)}$ at any time, and saves it in its buffer $\mathbb{B}_{\bar{x}_j}$, without being blocked by supersteps. (2) Under AAP, master P_0 is only responsible for making decision for termination and assembling partial answers by Assemble (see details below). (3) Workers exchange their status to adjust relative progress (see below).

Parameters. To reduce stragglers and redundant stale computations, each (virtual) worker P_i maintains a *delay stretch* DS_i such that P_i is put on hold for DS_i time to accumulate messages passed from other workers. Intuitively, this may enable worker P_i to converge in less rounds as shown in Example 1.1. Stretch DS_i is dynamically adjusted by a function δ based on the following.

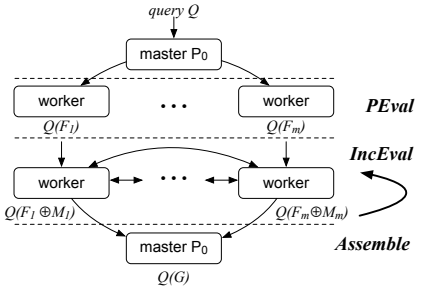


Fig. 4. Workflow of AAP

(1) Staleness η_i, measured by the number of messages in buffer $\mathbb{B}_{\bar{x}_i}$ received by worker P_i from distinct workers. Intuitively, the larger η_i is, the more messages are accumulated in $\mathbb{B}_{\bar{x}_i}$ and hence, the earlier P_i should start the next round of computation.

(2) Bounds $r_{\min}$ and $r_{\max}$, the smallest and largest rounds being executed at all workers, respectively. Each P_i also keeps track of its current round r_i. These are to control the relative speed of workers.

For example, to simulate SSP [47], when $r_i = r_{\max}$ and $r_i - r_{\min} > c$, we can set $\mathsf{DS}_i = +\infty$, to prevent P_i from moving too far ahead. Intuitively, when the fastest worker P_i are c rounds ahead of the slowest one, P_i is suspended for bounded staleness; worker P_i proceeds only after the slowest worker catches up, and at that moment DS_i is set to 0 to activate P_i immediately.

We will present an adjustment function δ for DS_i shortly.

Parallel model. Given a query $Q \in \mathcal{Q}$ and a partitioned graph G, AAP posts the same query Q to all the workers. It computes $Q(G)$ in three phases as shown in Fig. 4, described as follows.

(1) Partial evaluation. Upon receiving Q, PEval computes partial results $Q(F_i)$ at each worker P_i in parallel. After this, PEval generates a message $M_{(i,j)}$ and sends it to worker P_j for $j \in [1, m]$, $j \neq i$.

More specifically, message $M_{(i,j)}$ consists of triples (x, val, r), where (a) status variable $x \in C_i.\bar{x}$ is associated with a node v that is in $C_i \cap C_j$, and C_j is deduced from the index I_i; (b) val is the value of x, and (c) r indicates the round of P_i when val is computed. Worker P_i receives messages from other workers at any time and stores the messages in its buffer $\mathbb{B}_{\bar{x}_i}$.

(2) Incremental evaluation. In this phase, IncEval iterates until the termination condition is satisfied (see below). To reduce redundant computation, AAP adjusts (a) relative progress of workers and (b) workload assignments. More specifically, IncEval works as follows.

(1) IncEval is triggered at worker P_i to start the next round if (a) $\mathbb{B}_{\bar{x}_i}$ is nonempty, and (b) P_i has been suspended for DS_i time. Intuitively, IncEval is invoked only if changes are inflicted to $C_i.\bar{x}$, *i.e.,* $\mathbb{B}_{\bar{x}_i} \neq \emptyset$, and only if P_i has accumulated enough messages.

(2) When IncEval is triggered at worker P_i, it does the following:

- compute $M_i = f_{\mathsf{aggr}}(\mathbb{B}_{\bar{x}_i} \cup C_i.\bar{x})$, *i.e.,* IncEval applies the aggregate function f_{aggr} to $\mathbb{B}_{\bar{x}_i} \cup C_i.\bar{x}$, to deduce changes to its local update parameters; and it clears buffer $\mathbb{B}_{\bar{x}_i}$;
- incrementally compute $Q(F_i \oplus M_i)$ with IncEval, by treating M_i as updates to F_i (*i.e., $C_i.\bar{x}$*);
- derive messages $M_{(i,j)}$ that consists of *updated values* of $C_i.\bar{x}$ for border nodes that are in both C_i and C_j, for all $j \in [1, m]$, $j \neq i$; and send $M_{(i,j)}$ to worker P_j.

In the entire process, P_i keeps receiving messages from other workers and saves them in its buffer $\mathbb{B}_{\bar{x}_i}$. No synchronization barrier is imposed as opposed to BSP.

When IncEval completes its current round at worker P_i or when P_i receives a new message, delay stretch DS_i is adjusted. The next round of IncEval is triggered if the conditions (a) and (b) in (1) above are satisfied; otherwise P_i is suspended for DS_i time, and its resources are allocated to other (virtual) workers P_j to do useful computation, preferably to P_j that is assigned to the same physical worker as P_i to minimize the overhead for data transfer. When the suspension of P_i exceeds DS_i, worker P_i is activated again to start the next round of IncEval.

(3) Termination. When IncEval is done with its current round of computation, if $\mathbb{B}_{\bar{x}_i} = \emptyset$, worker P_i sends a flag *inactive* to master P_0 and becomes inactive. Upon receiving *inactive* from all workers, P_0 broadcasts a message *terminate* to all workers. Each worker P_i may respond with either *ack* if it is inactive, or *wait* if it is active or is in the queue for execution. If one of the workers replies *wait*, the iterative incremental step proceeds as described phase (2) above.

Upon receiving *ack* from all workers, P_0 pulls partial results from all workers, and applies Assemble to the partial results. The outcome of Assemble is referred to as *the result of the parallelization* of ρ under $\mathcal{P}$, denoted by $\rho(Q, G)$. AAP returns $\rho(Q, G)$ and terminates at this point.

Example 3.1. Recall the PIE program ρ for CC from Example 2.1. Under AAP, it works in three phases as follows. No changes need to be made to the PIE program.

(1) PEval computes connected components and their cids at each fragment F_i by worker P_i, in parallel using DFS. At the end of the process, the cids of border nodes are grouped as messages and sent to neighboring workers by each P_i. More specifically, for $j \in [1, m]$ and $j \neq i$, $\{v.\mathsf{cid} \mid v \in F_i.O \cap F_j.I\}$ is sent to worker P_j as message $M_{(i,j)}$ and is stored in buffer $\mathbb{B}_{\bar{x}_j}$ of P_j.

(2) IncEval first computes updates M_i by applying the aggregate function min to those changed cids in $\mathbb{B}_{\bar{x}_i} \cup C_i.\bar{x}$, when it is triggered at worker P_i as described above. It then incrementally updates local components in F_i starting from M_i. At the end of the process, the changed cid's are sent to neighboring workers as messages by each worker P_i, just like PEval does.

(3) Assemble is invoked at master when more changes can be made can be made in the phase above. It computes and returns connected components in the same way as described in Example 2.1. □

The example shows that AAP works well with the programming model of GRAPE.

Special cases. BSP, AP and SSP are special cases of AAP. Indeed, these can be carried out by AAP by specifying function δ as follows.

- BSP: function δ sets $\mathsf{DS}_i = +\infty$ if $r_i > r_{\min}$, *i.e.,* worker P_i is suspended; otherwise $\mathsf{DS}_i = 0$, *i.e.,* P_i proceeds at once; thus all workers are synchronized as no one can outpace the others.
- AP: function δ always sets $\mathsf{DS}_i = 0$, *i.e.,* worker P_i can trigger the next round of computation as soon as its buffer is nonempty.
- SSP: function δ sets $\mathsf{DS}_i = +\infty$ if $r_i > r_{\min} + c$ for a fixed bound c like in SSP, and sets $\mathsf{DS}_i = 0$ otherwise. That is, the fastest worker may move at most c rounds ahead.

AAP can also simulate Hsync [83] by using function δ to implement the switching rules of Hsync.

Dynamic adjustment. To reduce both stragglers and stale computation, AAP dynamically adjusts the delay stretch DS_i at each worker P_i. Intuitively, DS_i helps us control the following.

(a) Delay stretch DS_i puts worker P_i on hold to accumulate messages before a new round of IncEval. As shown in Example 1.1, letting stragglers wait and accumulate messages can improve the convergence in some cases, as opposed to letting stragglers catch up. It helps us strike a balance between stale-computation reduction and useful outcome from the next round of computation

(b) It helps us prevent a worker P_i from waiting indefinitely. To this end, AAP keeps track of how long a worker has waited and compares it with the expected waiting time.

(c) With DS_i, AAP is also able to prevent fast workers from outpacing the slowest ones too much, when necessary. This can help us reduce stale computation of the fast workers like SSP.

Putting these together, we give an example function δ to adjust DS_i as follows.

$$\mathsf{DS}_i = \begin{cases} +\infty & \neg S(r_i, r_{\min}, r_{\max}) \vee (\eta_i = 0) \\ T^i_{\mathsf{L}_i} - T^i_{\mathsf{idle}} & S(r_i, r_{\min}, r_{\max}) \wedge (1 \leq \eta_i < \mathsf{L}_i) \\ 0 & S(r_i, r_{\min}, r_{\max}) \wedge (\eta_i \geq \mathsf{L}_i) \end{cases} \tag{1}$$

where the parameters of function δ are described as follows.

(1) Predicate $S(r_i, r_{\min}, r_{\max})$ is to decide whether worker P_i should be suspended immediately. For example, under SSP, it is defined as false if $r_i = r_{\max}$ and $|r_{\max} - r_{\min}| > c$. When bounded staleness is not needed (see Section 5.3), $S(r_i, r_{\min}, r_{\max})$ is constantly true.

(2) Variable L_i "predicts" how many messages should be accumulated before the next round of IncEval at worker P_i. AAP adjusts L_i at each round, based on (a) predicted running time t_i of the next round, and (b) predicted arrival rate s_i of messages. When s_i is above the average rate, L_i is changed to $\max(\eta_i, \mathsf{L}_\perp) + \Delta t_i * s_i$, where Δt_i is a fraction of t_i, and $\mathsf{L}_\perp$ is adjusted with the number of "fast" workers. One can approximate t_i and s_i by aggregating statistics of consecutive rounds of IncEval. To get more precise estimate, we use a random forest model [49] (see Section 6).

(3) Variable $T^i_{\mathsf{L}_i}$ estimates how longer worker P_i should wait to accumulate L_i many messages. We approximate it as $\frac{\mathsf{L}_i - \eta_i}{s_i}$, in terms of the number of messages that remain to be received, and message arrival rate s_i. Finally, T^i_{idle} is the idle time of worker P_i after the last round of IncEval. The reason to use T^i_{idle} is to prevent worker P_i from indefinite waiting.

Example 3.2. As an instantiation of Example 1.1, recall the PIE program ρ for CC given in Example 2.1 and illustrated in Example 3.1. Consider a graph G that is partitioned into fragments F_1, F_2 and F_3 and distributed across workers P_1, P_2 and P_3, respectively. As depicted in Fig. 1b, (a) each circle represents a connected component, annotated with its cid, and (b) a dotted line indicates a cut edge between fragments. One can see that graph G has a single connected component with the minimal vertex id 0. Suppose that workers P_1, P_2 and P_3 take 3, 3 and 6 time units, respectively.

One can verify the following by referencing Figure 1a.

(a) Under BSP, Figure 1a (1) depicts part of a run of the PIE program ρ, which takes 5 rounds for the minimal cid 0 to reach connected component 7.

(b) Under AP, a run is shown in Fig. 1a (2). Note that before getting cid 0, workers P_1 and P_2 invoke 3 rounds of IncEval and exchange cid 1 among connected components 1-4, while under BSP, one round of IncEval suffices to pass cid 0 from P_3 to these components. Hence a large part of the computations of faster P_1 and P_2 is stale and redundant.

(c) Under SSP with bounded staleness 1, a run is given in Fig. 1a (3). It is almost the same as Fig. 1a (2), except that P_1 and P_2 cannot start round 4 before P_3 finishes round 2. More specifically, when minimal cids in components 5 and 6 are set to 0 and 4, respectively, P_1 and P_2 have to wait for P_3 to set the cid of component 7 to 5. These again lead to unnecessary stale computations.

(d) Under AAP, worker P_3 can suspend IncEval until it receives enough changes as shown in Fig. 1a (4). For instance, function δ starts with $\mathsf{L}_\perp = 0$. It sets $\mathsf{DS}_i = 0$ if $|\eta_i| \geq 1$ for $i \in [1, 2]$ since

no messages are predicted to arrive within the next time unit. In contrast, it sets $DS_3 = 1$ if $|\eta_3| \leq 4$ since in addition to the 2 messages accumulated, 2 more messages are expected to arrive in 1 time unit; hence δ decides to increase DS_3. These delay stretches are estimated based on the running time (3, 3 and 6 for P_1, P_2 and P_3, respectively) and message arrival rates. With these delay stretches, P_1 and P_2 may proceed as soon as they receive new messages, but P_3 starts a new round only after accumulating 4 messages. Now P_3 only takes 2 rounds of IncEval to update all the cids in F_3 to 0. Compared with Figures 1a (1)–(3), the straggler reaches fixpoint in less rounds. □

AAP reduces the costs of iterative graph computations mainly from three directions.

(1) AAP reduces both stale computations and stragglers by adjusting relative progress of workers. In particular, when the time taken by different rounds at a worker does not vary much (*e.g.,* PageRank in Section 8), fast workers are "automatically" grouped together after a few rounds and run essentially under BSP within the group, while the group and slow workers run under AP.

(2) Like GRAPE, AAP employs incremental IncEval to minimize recomputations. The speedup is particularly evident when IncEval is bounded [68], localizable or relatively bounded [34]. For instance, IncEval is *bounded* [67] if given F_i, Q, $Q(F_i)$ and M_i, it computes ΔO_i such that $Q(F_i \oplus M_i) = Q(F_i) \oplus \Delta O_i$, in cost that can be expressed as a function in $|M_i| + |\Delta O_i|$, the size of changes in the input and output; intuitively, it reduces the cost of computation on (possibly big) F_i to a function of small $|M_i| + |\Delta O_i|$. For example, IncEval for CC (Fig. 3) is a bounded incremental algorithm.

(3) Observe that algorithms PEval and IncEval are executed on fragments, which are graphs themselves. Hence AAP inherits all optimization strategies developed for the sequential algorithms.

4 CONVERGENCE AND EXPRESSIVE POWER

As observed by [85], asynchronous executions complicate convergence analysis. Nevertheless, we develop a condition to ensure AAP to converge at correct answers. Moreover, AAP is generic. We show that parallel models MapReduce, PRAM, BSP, AP and SSP can be optimally simulated by AAP.

4.1 Convergence and Correctness

Given a PIE program ρ (*i.e.,* PEval, IncEval, and Assemble) for a class $\mathcal{Q}$ of graph queries and a partition strategy $\mathcal{P}$, we want to know whether the AAP parallelization of ρ converges at correct results. That is, whether for all queries $Q \in \mathcal{Q}$ and all graphs G, ρ terminates under AAP over G partitioned via $\mathcal{P}$, and moreover, it produces correct result $\rho(Q, G) = Q(G)$.

We formalize termination and correctness as follows.

Fixpoint. Similar to GRAPE [38], AAP parallelizes a PIE program ρ based on a simultaneous fixpoint operator $\phi(R_1, \ldots, R_m)$ that starts with partial evaluation of PEval and employs incremental function IncEval as the intermediate consequence operator, as follows:

$$R_i^0 = \mathsf{PEval}(Q, F_i^0[\bar{x}_i]), \tag{2}$$

$$R_i^{r_i+1} = \mathsf{IncEval}(Q, R_i^{r_i}, F_i^{r_i}[\bar{x}_i], M_i), \tag{3}$$

where for $i \in [1, m]$, $R_i^{r_i}$ denotes partial results in round r_i at worker P_i, fragment $F_i^0 = F_i$, $F_i^{r_i}[\bar{x}_i]$ is fragment F_i at the end of round r carrying update parameters $C_i.\bar{x}$, and M_i denotes changes to $C_i.\bar{x}$ computed by aggregate $f_{\text{aggr}}(\mathbb{B}_{x_i} \cup C_i.\bar{x})$ as we have seen in Section 3.

The computation reaches a fixpoint if for all $i \in [1, m]$, there exists r_i such that $R_i^{r_i+1} = R_i^{r_i}$ and after round $r_i + 1$, IncEval cannot be invoked on $R_i^{r_i+1}$. Hence there exists r_0 such that no more

changes can be incurred to $R_i^{r_0}$ at any worker P_i ($i \in [1, m]$). At this point, Assemble is applied to $R_i^{r_0}$ for $i \in [1, m]$, and computes $\rho(Q, G)$. If so, we say that the PIE program ρ *converges at* $\rho(Q, G)$.

In contrast to synchronous execution, a PIE program ρ may have different asynchronous runs, when IncEval is triggered in different orders at multiple workers depending on, *e.g.,* partition of G, clusters and network latency. These runs may end up with different results [92]. A run of ρ can be represented as traces of PEval and IncEval at all workers (see, *e.g.,* Figure 1a).

We say that ρ *terminates under* AAP with $\mathcal{P}$ if for all queries $Q \in \mathcal{Q}$ and graphs G, all runs of ρ converge at a fixpoint. We say that ρ has *the Church-Rosser property* under AAP if all its asynchronous runs converge at the same result. We say that AAP *correctly parallelizes* ρ if ρ has the Church-Rosser property, *i.e.,* it always converges at the same $\rho(Q, G)$, and $\rho(Q, G) = Q(G)$.

Termination and correctness. We now identify a monotone condition under which a PIE program is guaranteed to converge at correct answers under AAP. We start with some notations.

(1) We assume a partial order $\preceq$ on the domain of status variables. Since the partial results R_i at fragment F_i are encoded as the collection of status variables defined on F_i, we extend $\preceq$ to partial results R_i as follows. We say that $R_i \preceq R_i'$ if $x.\mathrm{val} \preceq x.\mathrm{val}'$ for each status variable x defined on F_i, where $x.\mathrm{val}$ and $x.\mathrm{val}'$ are the values of x in partial results R_i and R_i', respectively. This contrasts with GRAPE [38], which defines partial order only on update parameters. We need this notion to analyze the Church-Rosser property of asynchronous runs under AAP.

We need another order to compare associated collections of update parameters. Denote by S_x and S_x' multi-sets of values for a parameter x. We write $S_x \trianglelefteq S_x'$ if the minimal element in S_x is no "larger" than any element in S_x' *w.r.t.* order $\preceq$. For example, if $\preceq$ is the linear order over integers, $S_x = \{1, 3, 3\}$ and $S_x' = \{7, 7, 8\}$, then $S_x \trianglelefteq S_x'$ since the minimal element 1 in S_x is smaller than each element in S_x'. We extend $\trianglelefteq$ to collections $\bar{x}_i$, *i.e.,* we write $S_{\bar{x}_i} \trianglelefteq S'_{\bar{x}_i}$ if $S_x \trianglelefteq S_x'$ for each $x \in \bar{x}_i$.

(2) We study the following properties of IncEval.

- IncEval is *contracting* if when $R_i^{r+1} = \mathsf{IncEval}(Q, R_i^r, F_i^r[\bar{x}_i], M_i)$ and $M_i = f_{\mathrm{aggr}}(\mathbb{B}_{x_i} \cup C_i^r.\bar{x})$, then (1) $R_i^{r+1} \preceq R_i^r$ and (2) $C_i^{r+1}.\bar{x} \trianglelefteq M_i \trianglelefteq \mathbb{B}_{x_i} \cup C_i^r.\bar{x}$, for queries $Q \in \mathcal{Q}$ and fragmented graphs G via $\mathcal{P}$ ($i \in [1, m]$). Intuitively, IncEval decreases status variables along the partial order $\preceq$.
- IncEval is *monotonic* if when $\bar{R}_i^s \preceq R_i^t$ and their associated collections of update parameters S and T satisfy $S \trianglelefteq T$, then (1) $\bar{R}_i^{s+1} \preceq R_i^{t+1}$; and (2) $f_{\mathrm{aggr}}(S) \trianglelefteq f_{\mathrm{aggr}}(T)$, for all queries $Q \in \mathcal{Q}$, graphs G, and all $i \in [1, m]$. Here $M_i = f_{\mathrm{aggr}}(S)$, $M_i' = f_{\mathrm{aggr}}(T)$, $\bar{R}_i^{s+1} = \mathsf{IncEval}(Q, \bar{R}_i^s, F_i^s[\bar{x}_i], M_i)$, and $R_i^{t+1} = \mathsf{IncEval}(Q, R_i^t, F_i^t[\bar{x}_i], M_i')$.

The contracting property describes how status variables are updated in the same run of IncEval, while the monotonic property concerns how status variables are updated in possibly different runs.

For instance, we will show that the PIE program ρ for CC given in Example 2.1 is contracting and monotonic in Examples 4.2 and 4.4, respectively.

(2) We want to identify a condition under which AAP correctly parallelizes a PIE program ρ as long as its sequential algorithms PEval, IncEval and Assemble are correct, regardless of the order in which PEval and IncEval are triggered. We use the following.

We say that (a) PEval is *correct* if for all queries $Q \in \mathcal{Q}$ and graphs G, $\mathsf{PEval}(Q, G)$ returns $Q(G)$; (b) IncEval is *correct* if $\mathsf{IncEval}(Q, Q(G), G, M)$ returns $Q(G \oplus M)$, where M denotes messages (updates); and (c) Assemble is *correct* if when ρ converges at round r_0 under BSP, $\mathsf{Assemble}(R_1^{r_0}, \ldots, R_m^{r_0}) = Q(G)$. We say that ρ is *correct for* $\mathcal{Q}$ if PEval, IncEval and Assemble are correct for $\mathcal{Q}$.

A monotone condition. We identify three conditions for ρ.

(T1) The values of updated parameters are from a finite domain.
(T2) IncEval is contracting.
(T3) IncEval is monotonic.

While conditions T1 and T2 are essentially the same as the monotonic conditions for the correctness of GRAPE under BSP [38], condition T3 does not find a counterpart in [38]. Note that T3 does not entail T2. For instance, suppose that $\preceq$ is the linear order $\leq$ on integers, and status variable x has value 5 and 4 in $\bar{R}_i^s$ and R_i^t, respectively. After one round of IncEval, it is possible that x gets value 7 and 6, respectively, depending on how IncEval is defined. Then IncEval may be monotonic but it is not contracting, since the values of x increase. Similarly, one can show that T2 does not entail T3.

The termination condition of GRAPE remains intact under AAP.

Theorem 4.1. *Under* AAP*, a* PIE *program ρ guarantees to terminate with any partition strategy $\mathcal{P}$ if ρ satisfies conditions T1 and T2.* □

These conditions are general. Indeed, given a graph G, the values of update parameters are often computed from the active domain of G and are finite. By the use of aggregate function f_{aggr}, IncEval is often contracting, as illustrated by the PIE program for CC above.

Example 4.2. Consider the PIE program ρ for CC from Example 2.1. Since all ids of connected components are integers, we define the partial order $\preceq$ to be the order $\leq$ of integers.

We now show that ρ satisfies T2, *i.e.,* the contraction of IncEval. Since f_{aggr} uses min to select minimal v.cid in the buffer and updated parameters, the value of v.cid in M_i is no larger than the one in R_i^r and $\mathbb{B}_{x_i}$, where M_i denotes changes to $\bar{x}_i$ computed by $f_{\text{aggr}}(\mathbb{B}_{x_i} \cup C_i^r.\bar{x})$. Recall that for any node $v \in V_i$, IncEval sets v.cid to be the minimal v'.cid over all node v' in the connected component containing v. Hence after the computation $R_i^{r+1} = \text{IncEval}(Q, R_i^r, F_i^r[\bar{x}_i], M_i)$, the value of v.cid in R_i^{r+1} is no larger than the one in R_i^r, $\mathbb{B}_{x_i}$ and M_i. That is, IncEval is contracting. □

Proof. It is easy to verify that if ρ terminates at $(N_i + 1)$-th round under AAP for each fragment F_i ($i \in [1, n]$), then in each round $r_i \leq N_i$, IncEval changes at least one update parameter. By this property, we show that under T1 and T2, ρ always terminates under AAP.

Assume by contradiction that there exist a query $Q \in \mathcal{Q}$ and a graph G such that ρ does not terminate. Denote by (a) N_x the size of the finite set consisting of assigned values for variable x, where x is an update parameter of G; and (b) $N = \sum_{x \in \bar{x}_i, i \in [1,m]} N_x$, *i.e.,* the total number of distinct values assigned to update parameters. Since ρ does not terminate, there exists a worker P_i running at least $N + 1$ rounds. By the property above, in each round of IncEval, at least one status variable is updated. Hence there exists a variable x that is updated $N_x + 1$ times. Moreover, since IncEval is contracting (T2), the assigned values to x follow a partial order. Thus x has to be assigned $N_x + 1$ distinct values, which contradicts the assumption that there exist only N_x distinct values for x. □

However, the condition of GRAPE does not suffice for the Church-Rosser property of asynchronous runs. For the correctness of a PIE program under AAP, we need condition T3 additionally.

Theorem 4.3. *With any partition strategy $\mathcal{P}$, under conditions T1, T2 and T3,* AAP *correctly parallelizes a* PIE *program ρ for a query class $\mathcal{Q}$ if ρ is correct for $\mathcal{Q}$.* □

Example 4.4. Continuing with Example 4.2, we show that the PIE program ρ for CC satisfies condition T3. Let $\bar{R}_i^s$ and R_i^t be two partial results in (possibly different) runs, and S and T be their associated collections of update parameters. Suppose that $\bar{R}_i^s \preceq R_i^t$ and $S \trianglelefteq T$. We show that (1) $f_{\text{aggr}}(S) \trianglelefteq f_{\text{aggr}}(T)$ and (2) $\bar{R}_i^{s+1} \preceq R_i^{t+1}$. Note that f_{aggr} is min, *i.e.,* for each $v \in F_i.O$, f_{aggr} updates

v.cid to the minimum one in the collection of update parameters. Since $S \trianglelefteq T$, $f_{\text{aggr}}(S) \trianglelefteq f_{\text{aggr}}(T)$. It follows that cid's of root nodes in R_i^s are no larger than their counterparts in R_i^t. Observe that (1) the cid's for nodes in $F_i.O$ are updated to the cid's of their linked roots (see Figure 3); and (2) for nodes in $V_i \setminus F_i.O$, their cid's do not change in IncEval. Hence $\tilde{R}_i^{s+1} \preceq R_i^{t+1}$ and IncEval is monotonic. □

Proof. By Theorem 4.1, any run of the PIE program ρ terminates under T1 and T2. In particular, consider the BSP run σ^* of ρ, a special case of AAP runs, and assume that all workers terminate after r^* rounds. Denote by $\tilde{R}_i^r$ the partial result on the i-th fragment in σ^* after r rounds. Then $(\tilde{R}_1^{r^*}, \ldots, \tilde{R}_m^{r^*})$ is a fixpoint of ρ under BSP. To prove Theorem 4.3 it suffices to show the Church-Rosser property, *i.e.,* we only need to show that an arbitrary run σ of ρ under AAP converges to the same fixpoint as σ^*. More specifically, assume that worker P_i terminates after r_i rounds at partial result $R_i^{r_i}$ in σ. Then $(R_1^{r_1}, \ldots, R_m^{r_m}) = (\tilde{R}_1^{r^*}, \ldots, \tilde{R}_m^{r^*})$. That is, it suffices to prove the following.

Lemma 4.5. $R_i^{r_i} \preceq \tilde{R}_i^r$ *for* $i \in [1, m]$ *and* $r \geq 0$, i.e., *partial results in the* BSP *run* σ^* *are no "smaller" than the fixpoint in the run* σ. □

Lemma 4.6. $\tilde{R}_i^{r^*} \preceq R_i^r$ *for* $i \in [1, m]$ *and* $r \geq 0$, i.e., *the partial results in the run* σ *are no "smaller" than the fixpoint in the run* σ^*. □

Proof of Lemma 4.5. The lemma follows from the following two claims.

Claim 4.7 (Fixpoint). Under T1, T2 and T3, for all $i \in [1, m]$, there exists r_i such that (a) $R_i^{r_i} = \text{IncEval}(Q, R_i^{r_i}, F^{r_i}[\bar{x}_i], M_i)$, where $M_i = f_{\text{aggr}}(\emptyset \cup C_i^{r_i}.\bar{x})$, *i.e.,* no more messages from other workers, and (b) after round r_i, IncEval cannot be invoked on partial results $R_i^{r_i}$. □

Claim 4.8 (Consistency). Under T1, T2 and T3, when all workers terminate, the values of update parameters are consistent, *i.e.,* for each $i, j \in [1, m]$ with $i \neq j$, and for each variable $x \in C_i.\bar{x} \cap C_j.\bar{x}$, x has the same value in $R_i^{r_i}$ and $R_j^{r_j}$. □

Assuming these claims, we show Lemma 4.5 by induction on r.

∘ *Base case.* The case $R_i^{r_i} \preceq R_i^0 = \tilde{R}_i^0$ follows from T2 and the fact that the first IncEval runs after the same PEval in both σ and σ^*.

∘ *Inductive step.* Suppose that $R_i^{r_i} \preceq \tilde{R}_i^r$ for all $i \in [1, m]$. We will show below that $R_1^{r_i} \preceq \tilde{R}_1^{r+1}$ by using the inductive hypothesis and the monotonicity of IncEval. Let $\tilde{R}_1^{r+1} = \text{IncEval}(Q, \tilde{R}_1^r, F_1^r[\bar{x}_1], M_1)$, where $M_1 = f_{\text{aggr}}(m_1 \cup \ldots \cup m_k \cup \tilde{C}_1^r.\bar{x})$ (updates to $\bar{x}_1$), and m_j is the message from $\tilde{R}_{i_j}^r$ for $j \in [1, k]$. By Claim 4.7, we know that $R_1^{r_1} = \text{IncEval}(Q, R_1^{r_1}, F_1^{r_1}[\bar{x}_1], M_1')$, where $M_1' = f_{\text{aggr}}(\emptyset \cup C_1^{r_1}.\bar{x})$, and P_1 terminates at round r_1. By the inductive hypothesis, we have $R_i^{r_i} \preceq \tilde{R}_i^r$ for $i \in [1, m]$.

To see that $R_1^{r_1} \preceq \tilde{R}_1^{r+1}$, it suffices to show that $C_1^{r_1}.\bar{x} \trianglelefteq m_1 \cup \ldots \cup m_k \cup \tilde{C}_1^r.\bar{x}$. For if it holds, then $R_1^{r_1} \preceq \tilde{R}_1^{r+1}$ by T3, the monotonicity of IncEval. Indeed, let $\bar{x}_1 = \{y_1, \ldots, y_\ell\}$. Then the inequality can be deduced from the following: (1) the value of y_j ($j \in [1, \ell]$) in $C_1^{r_1}.\bar{x}$ is no "larger" than the one in $\tilde{C}_1^r.\bar{x}$ since $R_1^{r_1} \preceq \tilde{R}_1^r$; and (2) for each (y, val, t) in $m_1 \cup \cdots \cup m_k$, the value of y in $C_1^{r_1}.\bar{x}$ is no "larger" than val by Claim 4.8 and $R_{i_j}^{r_{i_j}} \preceq \tilde{R}_{i_j}^r$ ($j \in [1, k]$). Here $R_{i_j}^{r_{i_j}}$ ($j \in [1, k]$) is the partial result on the i_j-th fragment after the (arbitrary) run σ terminates. □

It remains to verify Claims 4.7 and 4.8.

Proof of Claim 4.7 (Fixpoint). It suffices to show that when AAP terminates, we have that $(\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x}) \trianglelefteq C_i^{r_i}.\bar{x}$ ($i \in [1, m]$), where r_i is the last round of computation on fragment F_i, $\mathbb{B}_{\bar{x}_i}$ consists of

the messages received before the last round, and $C_i^{r_i-1}.\bar{x}$ denotes the update parameters of $F_i^{r_i-1}$. Indeed, if $(\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x}) \trianglelefteq C_i^{r_i}.\bar{x}$ for all $i \in [1,m]$, we can further show $R_i^{r_i}$=IncEval$(Q, R_i^{r_i}, F^{r_i}[\bar{x}_i], M_i)$; and since no IncEval can be invoked after the r_i-th round on fragment F_i for all $i \in [1,m]$, we know that ρ reaches a fixpoint. We can verify that $R_i^{r_i}$ = IncEval$(Q, R_i^{r_i}, F^{r_i}[\bar{x}_i], M_i)$ in two steps.

(1) We first show that $f_{\text{aggr}}(\emptyset \cup C_i^{r_i}.\bar{x}) = C_i^{r_i}.\bar{x}$. By T2 and T3, we have that

$$C_i^{r_i}.\bar{x} \trianglelefteq f_{\text{aggr}}(\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x}) \trianglelefteq f_{\text{aggr}}(\emptyset \cup C_i^{r_i}.\bar{x}) \trianglelefteq C_i^{r_i}.\bar{x}. \quad (4)$$

These follow from (a) the contraction of IncEval (T2), (b) the hypothesis $(\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x}) \trianglelefteq C_i^{r_i}.\bar{x}$ and the monotonicity of IncEval (T3), and (c) T2, respectively. Thus $f_{\text{aggr}}(\emptyset \cup C_i^{r_i}.\bar{x}) = C_i^{r_i}.\bar{x}$.

(2) Next, we show that $R_i^{r_i}$ = IncEval$(Q, R_i^{r_i}, F^{r_i}[\bar{x}_i], M_i)$. Note that IncEval$(Q, R_i^{r_i-1}, F_i^{r_i-1}[\bar{x}_i], M_i')$ computes $Q(F_i^{r_i-1} \oplus M_i')$, where $M_i' = f_{\text{aggr}}(\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x})$. Similarly, IncEval$(Q, R_i^{r_i}, F_i^{r_i}[\bar{x}_i], M_i)$ computes $Q(F_i^{r_i} \oplus M_i)$, where $M_i = f_{\text{aggr}}(\emptyset \cup C_i^{r_i}.\bar{x})$. By inequality (4), we know that $M_i = M_i'$. By the contraction of IncEval, $F_i^{r_i} \oplus M_i$ is the same as $F_i^{r_i}$. It follows that $R_i^{r_i}$ = IncEval$(Q, R_i^{r_i}, F^{r_i}[\bar{x}_i], M_i)$.

To show that $(\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x}) \trianglelefteq C_i^{r_i}.\bar{x}$, we verify that $C_i^{r_i}.\bar{x} \subseteq (\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x})$. For if it holds, then $(\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x}) \trianglelefteq C_i^{r_i}.\bar{x}$ follows from the definition of the order $\trianglelefteq$. We verify this by contradiction. Suppose that $C_i^{r_i}.\bar{x} \subseteq (\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x})$ does not hold. Then there exists an update parameter x^* in $\bar{x}_i$ such that its value in $C_i^{r_i}.\bar{x}$ is strictly "smaller" than the ones in $\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x}$, *i.e.,* x^* is updated by IncEval. Note that $C_i^{r_i}.\bar{x}$ is obtained from $\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x}$ by using IncEval, and IncEval is contracting (T2). Based on these, we show that there exists one more run of IncEval on $R_i^{r_i}$, a contradiction to the assumption that worker P_i terminates after r_i rounds.

Let val be the value of x^* in $C_i^{r_i}.\bar{x}$ and $P_{j_1}, \ldots P_{j_k}$ be the workers sharing x^*. Since x^* is updated, P_i sends a message containing (x^*, val, r_i) to $P_{j_1}, \ldots, P_{j_k}$. Let val_ℓ be the value of x^* on P_{j_ℓ} for $\ell \in [1,k]$ when P_{j_ℓ} processes the message. There are the following two cases to consider.

(i) If for some val_ℓ ($\ell \in [1,k]$), $\text{val}_\ell \preceq \text{val}$, then AAP must have sent a message containing $(x^*, \text{val}_\ell, r_\ell)$ to P_i. It contradicts the assumption that val is strictly "smaller" than the value of x^* in $\mathbb{B}_{\bar{x}_i}$.

(ii) If val is strictly less than all of $\text{val}_1, \ldots \text{val}_k$, then by the contraction of IncEval (T2), $P_{j_1}, \ldots, P_{j_k}$ update the values of x^* to $\text{val}_1', \ldots, \text{val}_k'$, respectively, such that $\text{val}_1' \preceq \text{val}, \ldots, \text{val}_k' \preceq \text{val}$. By assumption val_ℓ' is strictly "smaller" than val_ℓ for $\ell \in [1,k]$. The value of x^* on $P_{j_1}, \ldots, P_{j_k}$ is updated. Thus AAP sends these values of x^* to P_i, triggering a new round of IncEval, a contradiction.

Putting these together, we have that $(\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x}) \trianglelefteq C_i^{r_i}.\bar{x}$. □

Proof of Claim 4.8 (Consistency). We next show that if val_i and val_j are the final values of an update parameter x shared by F_i and F_j, respectively, then $\text{val}_i = \text{val}_j$.

Let val_i^0 and val_j^0 be the initial values of x in F_i and F_j, respectively; then $\text{val}_i^0 = \text{val}_j^0$. Observe the following. (a) If PEval and IncEval do not update x, then $\text{val}_i = \text{val}_j$. (b) Otherwise assume *w.l.o.g.* that $\text{val}_i \preceq \text{val}_i^0$ and $\text{val}_i \neq \text{val}_i^0$. Suppose that the value of x is updated to val_i in the r-th round of IncEval on F_i. At the end of this round, (x, val_i, r) is sent to P_j, triggering one round of computation on F_j. By the contraction of IncEval (T2), we have that $\text{val}_j \preceq \text{val}_i$. By a similar argument, we can show that $\text{val}_i \preceq \text{val}_j$. Putting these together we have that $\text{val}_i = \text{val}_j$. □

Proof of Lemma 4.6. Since the BSP run σ^* is a special run under AAP, from Claims 4.7 and 4.8, we can deduce the following: (a) $\tilde{R}_i^{r^*}$ is a fixpoint, and moreover, $\tilde{R}_i^{r^*}$ = IncEval$(Q, \tilde{R}_i^{r^*}, F_i^{r^*}[\bar{x}_i], M_i)$, where M_i= $f_{\text{aggr}}(\emptyset \cup C_i^{r^*}.\bar{x})$; and (b) the values of update parameters are consistent.

We prove that $\tilde{R}_i^{r^*} \preceq R_i^r$ in two steps. (1) We first construct a finite tree $\mathcal{T}$ to represent the computation trace of R_i^r, where the root of $\mathcal{T}$ is (i,r), and nodes of $\mathcal{T}$ are in the form of (j,t),

indicating the t-th round of IncEval on the j-th fragment. (2) We then show that $\tilde{R}_j^{r^*} \preceq R_j^t$ for each node (j,t) of $\mathcal{T}$. It follows that $\tilde{R}_i^{r^*} \preceq R_i^r$ by applying $\tilde{R}_j^{r^*} \preceq R_j^t$ to the root (i,r) of $\mathcal{T}$.

(1) Tree $\mathcal{T}$ is constructed top-down from the root (i,r). For a node (j,t) with $t \neq 0$, we define its children based on the t-th round of IncEval on the j-th fragment. Suppose that R_j^t is computed by $R_j^t = \mathsf{IncEval}(Q, R_j^{t-1}, F_j^{t-1}[\bar{x}_j], M_j)$ and M_j is the aggregation result of $m_1, \ldots, m_k$, which are k messages sent from the j_1-th, …, j_k-th worker after their t_1-th, … t_k-th round of IncEval, respectively. Then we add k+1 pairs $(j,t-1), (j_1,t_1), \ldots, (j_k,t_k)$ as the children of (j,t). Intuitively, the children of (j,t) encode the dependency of R_j^t. The construction stops when each path of $\mathcal{T}$ reaches a node (j,t) with $t = 0$. Tree $\mathcal{T}$ is finite since (i) for $(j,t_1), (j,t_2), \ldots (j,t_\ell)$ on a path from the root (i,r), $t_1 > t_2 > \cdots > t_\ell$; and (ii) $\mathcal{T}$ is finite branching since each round of IncEval only uses finitely many messages.

(2) We show that $\tilde{R}_j^{r^*} \preceq R_j^t$ for each node (j,t) of $\mathcal{T}$ by induction in a bottom-up manner.

Base case. When $t = 0$, $\tilde{R}_j^{r^*} \preceq \tilde{R}_j^0 = R_j^0$ for $j \in [1,m]$ by the contraction of IncEval (T2).

Induction step. Suppose that (j,t) has children $(j,t-1), (j_1,t_1), (j_2,t_2), \ldots, (j_k,t_k)$, and that the inductive hypothesis holds for $(j,t-1), (j_1,t_1), (j_2,t_2), \ldots, (j_k,t_k)$. We show that $\tilde{R}_j^{r^*} \preceq R_j^t$. Observe that R_j^t is computed by $R_j^t = \mathsf{IncEval}(Q, R_j^{t-1}, F_j^{t-1}[\bar{x}_j], M_j)$, where $M_j = f_{\mathsf{aggr}}(m_1 \cup \cdots \cup m_k \cup C_j^{t-1}.\bar{x})$ denotes changes to $\bar{x}_j$, and $m_1, \ldots, m_k$ are messages from workers $P_{j_1}, \ldots, P_{j_k}$ after their t_1-th, …, t_k-th round of IncEval, respectively. Meanwhile, since $\tilde{R}_j^{r^*}$ is a fixpoint, $\tilde{R}_j^{r^*} = \mathsf{IncEval}(Q, \tilde{R}_j^{r^*}, F_j^{r^*}[\bar{x}_j], M_j')$, where $M_j' = f_{\mathsf{aggr}}(\emptyset \cup C_j^{r^*}.\bar{x})$ denotes changes to $\bar{x}_j$. By the induction hypothesis, $\tilde{R}_j^{r^*} \preceq R_j^{t-1}$ and $\tilde{R}_{j_\ell}^{r^*} \preceq R_{j_\ell}^{t_\ell}$ ($\ell \in [1,k]$). By the monotonicity of IncEval (T3), to show that $\tilde{R}_j^{r^*} \preceq R_j^t$, it suffices to prove that $C_j^{r^*}.\bar{x} \trianglelefteq m_1 \cup \ldots m_k \cup C_j^{t-1}.\bar{x}$. The latter can be verified along the same line as Lemma 4.5. □

This completes the proof of Lemmas 4.5 and 4.6 and hence Theorem 4.3. □

Special cases. Recall that BSP, AP and SSP are special cases of AAP. From the proof of Theorem 4.3 we can conclude that as long as a PIE program ρ is correct for Q, ρ can be correctly parallelized

- under conditions T1 and T2 by BSP;
- under conditions T1, T2 and T3 by AP; and
- under conditions T1, T2 and T3 by SSP.

Novelty. As far as we are aware of, T1, T2 and T3 provide the first condition for asynchronous runs to converge and ensure the Church-Rosser property. To illustrate this, we examine convergence conditions for GRAPE [38], Maiter [91], BAP [45] and SSP [28, 47].

(1) As remarked earlier, the condition given for GRAPE in [38] does not ensure the Church-Rosser property, which is not an issue for BSP under which GRAPE is developed.

(2) Maiter [91] adopts vertex-centric programming and identifies four conditions for convergence, on an update function f that changes the state of a vertex based on its neighbors. The conditions require that f is distributive, associative, commutative and moreover, satisfies an equation on initial values.

As opposed to [91], we deal with block-centric programming of which the vertex-centric model is a special case, when a fragment is limited to a single node. Moreover, the last condition of [91] is quite restrictive. Further, the proof of [91] does not suffice for the Church-Rosser property. A counterexample could be conditional convergent series, for which asynchronous runs may diverge [27, 53].

(3) It is shown that BAP can simulate BSP under certain conditions on message buffers [45]. It does not consider the Church-Rosser property, and we make no assumption about message buffers.

(4) Conditions have been studied to assure the convergence of stochastic gradient descent (SGD) with high probability [28, 47]. In contrast, our conditions are deterministic: under T1, T2 and T3, all AAP runs guarantee to converge at correct answers. Moreover, AAP computations are not limited to SGD.

(5) Two global properties, namely P-MONO and P-INIT, and one local condition P-EDGE, are proposed to ensure convergence in the analysis of graph processing over evolving graphs [77]. Our conditions differ from theirs in the following. (a) P-MONO is analogous to the contraction condition T2, but it alone does not ensure the Church-Rosser property. (b) P-INIT is a property even stronger than Church-Rosser, and is not easy to verify. (c) P-EDGE requires vertex-centric models. In contrast, our conditions work for both vertex-centric and graph-centric models.

4.2 Simulation of Other Parallel Models

We next show that algorithms developed for MapReduce, PRAM, BSP, AP and SSP can be migrated to AAP without extra complexity. That is, AAP is as expressive as the other parallel models. Note that while we focus on graph computations here, AAP is not limited to graphs. It is a parallel computation model as generic as BSP and AP, and does not have to take graphs as input.

Following [76], we say that a parallel model $\mathcal{M}_1$ *optimally simulates* model $\mathcal{M}_2$ if there exists a compilation algorithm that transforms any program with cost C on $\mathcal{M}_2$ to a program with cost $O(C)$ on $\mathcal{M}_1$. The cost includes computational and communication cost. That is, the complexity bound remains the same for all computations on $\mathcal{M}_1$ and $\mathcal{M}_2$.

As remarked in Section 3, BSP, AP and SSP are special cases of AAP. Hence the following holds.

Proposition 4.9. AAP *can optimally simulate* BSP*,* AP *and* SSP. □

By Proposition 4.9, algorithms developed for, *e.g.,* Pregel [58], GraphLab [43, 57] and GRAPE [38] can be migrated to AAP. As an example, a Pregel algorithm $\mathcal{A}$ (with a function *compute()* for vertices) can be simulated by a PIE algorithm ρ as follows. (a) PEval runs *compute()* over vertices with a loop, and uses status variable to exchange local messages instead of *SendMessageTo()* of Pregel. (b) The update parameters are status variables of border nodes, and function f_{aggr} groups messages just like Pregel, following BSP. (c) IncEval also runs *compute()* over vertices in a fragment, except that it starts from active vertices (border nodes with changed values). Point-to-point message passing is supported with auxiliary structures (*e.g.,* a clique, as will seen in the proof of Theorem 4.10 shortly).

We next show that AAP can optimally simulate MapReduce and PRAM. It was shown in [38] that GRAPE can optimally simulate MapReduce and PRAM, by adopting a form of key-value messages. We show a stronger result, which simply uses the message scheme of Section 3, which is referred to as designated messages in [38], without using key-value messages of GRAPE [38].

Theorem 4.10. *MapReduce and PRAM can be optimally simulated by (a)* AAP *and (b)* GRAPE *with designated messages only.* □

Proof. Since PRAM can be simulated by MapReduce [50], and AAP can simulate GRAPE (Section 3), it suffices to show that GRAPE can optimally simulate MapReduce with designated messages.

We show that all MapReduce programs with n processors can be optimally simulated by GRAPE with n processors. A MapReduce algorithm $\mathcal{A}$ is defined as follows. Its input is a multi-set I_0 of $\langle$*key, value*$\rangle$ pairs, and operations are a sequence $(B_1, \ldots, B_k)$ of subroutines, where each subroutine B_r ($r \in [1, k]$) consists of a mapper μ_r and a reducer ρ_r. Given I_0, $\mathcal{A}$ iteratively runs B_r ($r \in [1, k]$) as follows [31, 50]. Denote by I_r the output of subroutine B_r ($r \in [1, k]$).

(1) The mapper μ_r handles each pair $\langle$*key, value*$\rangle$ in I_{r-1} one by one, and produces a multi-set I'_r of $\langle$*key, value*$\rangle$ pairs as output.

(2) Group pairs in I'_r by the *key* values, *i.e.,* two pairs are in the same group if and only if they have the same *key* value. Group I'_r by distinct *key*s. Let $G_{k_1}, \ldots, G_{k_j}$ be the obtained groups.
(3) The reducer ρ_r processes the groups G_{k_l} ($l \in [1, j]$) one by one, and generates a multi-set I_r of $\langle$*key, value*$\rangle$ pairs as output.
(4) If $r < k$, $\mathcal{A}$ runs the next subroutine B_{r+1} on I_i in the same way as steps (i)-(iii); otherwise, $\mathcal{A}$ outputs I_k and terminates.

Given a MapReduce algorithm $\mathcal{A}$ with n processors, we simulate $\mathcal{A}$ with a PIE program $\mathcal{B}$ by GRAPE with n workers. We use PEval to simulate the mapper μ_1 of B_1, and (2) IncEval simulates reducer ρ_i, mapper μ_{i+1} ($i \in [1, k-1]$), and reducer ρ_k in final round.

There are two mismatches: (a) $\mathcal{A}$ has a list $(B_1, \ldots, B_k)$ of subroutines, while IncEval of GRAPE is a single function; and (b) $\mathcal{A}$ distributes $\langle$*key, value*$\rangle$ pairs across processors, while workers of GRAPE exchange message via update parameters only.

For (a), IncEval treats subroutines $B_1, \ldots, B_k$ of $\mathcal{A}$ as program branches, and uses an index r ($r \in [1, k]$) to select branches. For (b), we construct a complete graph G_W of n nodes as an additional input of $\mathcal{B}$, such that each worker P_i is represented by a node w_i for $i \in [1, n]$. Each node w_i has a status variable x to store a multi-set of $\langle r$,*key, value*$\rangle$ tuples. By using G_W, all n nodes become border nodes, and we can hence simulate the arbitrary shipment of data in $\mathcal{A}$ by storing the data in the update parameters of the workers of GRAPE.

More specifically, consider a multi-set I_0 of $\langle$*key, value*$\rangle$ pairs as input. We distribute these pairs in I_0 in exactly the same way as $\mathcal{A}$ does; each node w_i of G_W stores the pairs assigned to worker P_i.

The PIE program $\mathcal{B}$ is specified as follows.

(1) PEval simulates the mapper μ_1 of the subroutine B_1 as follows.

(a) Each worker runs the mapper μ_1 of B_1 on its local data.
(b) It computes the output $(I_1)'$ of μ_1 and stores it in the update parameters for later supersteps.
(c) For each pair $\langle$*key, value*$\rangle$ in $(I_1)'$, it includes a tuple $\langle 1, key, value\rangle$ in an update parameter.

If worker P_i of the reducer ρ_1 is to handle the pair $\langle$*key, value*$\rangle$, PEval adds $\langle$1, *key, value*$\rangle$ to the update parameter of node w_i. The aggregation function first takes a union of the update parameters of all w_i ($i \in [1, n]$), and then groups the tuples by *key*.

(2) IncEval first extracts the index r from the $\langle r$, *key, value*$\rangle$ tuples received, and uses r to select the right subroutine, as remarked earlier. IncEval then carries out the following operations:

(a) extract a multi-set $(I_r)'$ of $\langle$*key, value*$\rangle$ from the received messages of $\langle r$, *key, value*$\rangle$ tuples;
(b) run the reducer ρ_r, which is treated as a branch program of IncEval, on $(I_i)'$; denote by I_r the output of ρ_r; and
(c) if $r = k$, then IncEval sets the updated parameter to be empty, which terminates $\mathcal{B}$; otherwise, IncEval runs the mapper μ_{r+1} on I_r, constructs tuples $\langle r + 1$, *key, value* $\rangle$ for each $\langle$*key, value*$\rangle$ pair in the output of μ_{r+1}, and distributes update parameters in the same way as PEval.

(3) Assemble takes a union of the partial results from all workers.

It is easy to verify that the PIE program $\mathcal{B}$ correctly simulates the MapReduce program $\mathcal{A}$. Moreover, if $\mathcal{A}$ runs in T time and incurs communication cost C, then $\mathcal{B}$ takes $O(T)$ time and sends $O(C)$ data. Formally, this is verified by induction on k for the number of subroutines $(B_1, \ldots, B_k)$ in $\mathcal{A}$. □

5 PROGRAMMING WITH AAP

We have seen how AAP parallelizes CC (Examples 2.1–3.2). We next develop PIE algorithms for SSSP, CF and PageRank as examples. As opposed to [38], we parallelize these algorithms under AAP (Sections 5.1–5.3). These demonstrate that AAP does not make programming harder.

5.1 Graph Traversal

We start with the *single source shortest path* problem (SSSP), a primitive graph traversal operation. Consider a directed graph $G = (V, E, L)$ in which for each edge e, $L(e)$ is a positive number. The length of a path $(v_0, \ldots, v_k)$ in G is the sum of $L(v_{i-1}, v_i)$ for $i \in [1, k]$. For a pair (s, v) of nodes, denote by $\mathsf{dist}(s, v)$ the *shortest distance* from s to v. SSSP is stated as follows.

- Input: A directed graph G as above, and a node s in G.
- Output: Distance $\mathsf{dist}(s, v)$ for all nodes v in G.

AAP parallelizes SSSP in the same way as GRAPE [38], as described below.

(1) PIE. AAP takes Dijkstra's algorithm [40] for SSSP as PEval and the sequential incremental algorithm for SSSP developed in [67] as IncEval. It declares a status variable x_v for every node v, denoting $\mathsf{dist}(s, v)$, initially ∞ (except $\mathsf{dist}(s, s) = 0$). The candidate set C_i at each fragment F_i is $F_i.I \cup F_i.O$. The status variables in the candidates set are updated by PEval and IncEval in the same way as in [38], and are aggregated by using min as f_{aggr}. When no changes can be incurred to these status variables, Assemble is invoked to take the union of all partial results.

(2) Correctness is assured by the correctness of the sequential algorithms for SSSP and Theorem 4.3. Note that the values of update parameters are from the finite set $\{d \in \mathbb{N} \mid 0 \leq d \leq \sum_{e \in E} L(e)\} \cup \{\infty\}$. Thus T1 is satisfied. Define the partial order $\preceq$ over $\mathbb{N} \cup \{\infty\}$ as the linear order on $\mathbb{N}$ with the extension that $d \preceq \infty$ for each $d \in \mathbb{N} \cup \{\infty\}$. We show that IncEval of SSSP satisfies T2 and T3.

(i) IncEval is contracting because of the following. (a) For update parameter $x \in C_i.\bar{x}$, IncEval uses min as f_{aggr} to compute the minimal value of x in $\mathbb{B}_{\bar{x}_i} \cup C_i.\bar{x}$. (b) The incremental algorithm IncEval only decreases the values of status variables following the order $\preceq$ (see more in [38]).

(ii) IncEval is monotonic. Suppose that $\bar{R}_i^s \preceq R_i^t$ and that their associated update parameter S and T satisfy $S \trianglelefteq T$. More specifically, we have that (1) $x_v.\mathsf{val} \preceq x_v.\mathsf{val}'$ for each $v \in F_i$, where $x_v.\mathsf{val}$ and $x_v.\mathsf{val}'$ are values of x_v in $\bar{R}_i^s$ and R_i^t, respectively; and (2) for each $x \in C_i.\bar{x}$, the minimal value of x in S is smaller than the one in T. Thus by taking min as f_{aggr}, we have that $M_i \trianglelefteq M_i'$, where $M_i = f_{\text{aggr}}(S)$ and $M_i' = f_{\text{aggr}}(T)$, consisting of updates to $C_i.\bar{x}$.

To show that $\bar{R}_i^{s+1} \preceq R_i^{t+1}$, we construct two auxiliary graphs G_i^s and G_i^t, by extending F_i with (1) a new source node s^*; and (2) a new edge e from s^* to v for each v in F_i. In G_i^s and G_i^t, the edge from s^* to v is labeled with the latest value of x_v in $\bar{R}_i^s \oplus M_i$ and $R_i^t \oplus M_i'$, respectively. Since $R_i^s \preceq R_i^t$ and $M_i \trianglelefteq M_i'$, we have that $L(e) \preceq L(e)'$ for each edge e, where $L(e)$ and $L(e)'$ are the label of e in G_i^s and G_i^t, respectively. Thus each path in G_i^s has a smaller length than its counterpart in G_i^t. Since IncEval essentially computes the shortest path from s^* in the auxiliary graphs, we have that $\bar{R}_i^{s+1} \preceq R_i^{t+1}$.

5.2 Collaborative Filtering

We next consider collaborative filtering (CF) [55]. It takes as input a bipartite graph G that includes two types of nodes, namely, users U and products P, and a set of weighted edges $E \subseteq U \times P$. More specifically, (1) each user $u \in U$ (resp. product $p \in P$) carries an (unknown) latent factor vector $u.f$ (resp. $p.f$). (2) Each edge $e = (u, p)$ in E carries a weight $r(e)$, estimated as $u.f^T * p.f$ (possibly $\emptyset$, *i.e.,* "unknown") that encodes a rating from user u to product p. The *training set* E_T refers to edge set $\{e \in E \mid r(e) \neq \emptyset\}$, *i.e.,* all the known ratings. The CF problem is stated as follows.

- Input: A directed bipartite graph G, and a training set E_T.
- Output: The missing factor vectors $u.f$ and $p.f$ that minimize a loss function $\epsilon(f, E_T)$, estimated as $\sum_{((u,p)\in E_T)}(r(u,p) - u.f^T * p.f)^2 + \lambda(\|u.f\|^2 + \|p.f\|^2)$.

AAP parallelizes stochastic gradient descent (SGD) [55], a popular algorithm for CF simulating SSP.

(1) PIE. PEval declares a status variable $v.x = (v.f, v.\delta, t)$ for each node v, where $v.f$ is the factor vector of v (initially $\emptyset$), $v.\delta$ records accumulative updates to $v.f$, and t bookkeeps the timestamp at which $v.f$ is lastly updated. Assuming *w.l.o.g.* that $|P| \ll |U|$, it takes $F_i.O \cup F_i.I$, *i.e.,* the shared product nodes related to F_i, as its candidate set C_i. PEval is essentially "mini-batched" SGD. It computes the descent gradients for each edge (u, p) in F_i and accumulates them in $u.\delta$ and $p.\delta$, receptively. The accumulated gradients are then used to update the factor vectors of all local nodes. At the end, PEval sends the updated values of $C_i.\bar{x}$ to neighboring workers.

IncEval first checks whether the fastest worker is ahead of the slowest one by at most c rounds, where c is a given bounded staleness. Once the test passes, IncEval aggregates the factor vector of each node p in $F_i.O$ by taking max on the timestamp for tuples $(p.f, p.\delta, t)$ in $\mathbb{B}_{\bar{x}_i} \cup C_i.\bar{x}$. For each node in $F_i.I$, it aggregates its factor vector by applying a weighted sum of gradients computed at other workers. It then runs a round of SGD, starting from nodes with the changes to their status variables and processing the affected area. It sends the updated status variables as in PEval.

Assemble simply returns the collection of the factor vectors of all nodes from all the workers.

(2) Correctness has been verified under the bounded staleness condition [47, 84]. Along the same lines, one can show that the PIE program converges and correctly infers missing CF factors.

5.3 PageRank

Finally, we study PageRank [23] for ranking Web pages. Consider a directed graph $G = (V, E)$ representing Web pages and links. For each page $v \in V$, its ranking score is denoted by P_v. The PageRank algorithm of [23] iteratively updates P_v as follows:

$$P_v = d * \Sigma_{\{u|(u,v)\in E\}} P_u/N_u + (1-d),$$

where d is damping factor and N_u is the out-degree of u. The process iterates until the sum of changes of two consecutive iterations is below a threshold. The PageRank problem is stated as follows.

- Input: A directed graph G and a threshold ϵ.
- Output: The PageRank scores of nodes in G.

AAP parallelizes PageRank along the same lines as [74, 91], as follows.

(1) PIE. PEval declares a status variable x_v for each node $v \in F_i$ to keep track of updates to v from other nodes in F_i, at each fragment F_i. It takes $F_i.O \cup F_i.I$ as its candidate set C_i. Starting from an initial score 0 and an update x_v (initially $1-d$) for each v, PEval (a) increases the score P_v by x_v, and (b) updates variable x_u for each u linked from v by an incremental change $\delta = d * x_v/N_v$. At the end of its process, it sends the updated values of status variables for nodes in $F_i.O$ to its neighboring workers.

Upon receiving messages, IncEval iteratively updates scores. It (a) first aggregates changes to each border node by using sum as f_{aggr}; (b) starting from border nodes, it then propagates the changes to update other nodes in the local fragment and processes the affected nodes as in PEval; and (c) it derives the changes to the values of $C_i.\bar{x}$ and sends them to its neighboring workers as in PEval.

Assemble collects the scores of all the nodes in G when the convergence condition is met, *i.e.,* when the sum of changes of two consecutive iterations at each worker is below ϵ.

(2) Correctness. We show that the PIE program under AAP terminates and has the Church-Rosser property, along the same lines as the proof of Theorem 4.3. The proof makes use of the following property, as also observed by [91]: for each node v in graph G, P_v can be expressed as $\Sigma_{p\in\mathcal{P}} p(v) + (1-d)$, where $\mathcal{P}$ is the set of all paths to v in G, p is a path $(v_n, v_{n-1}, \ldots v_1, v)$, $p(v) = (1-d) \cdot \prod_{j=1}^{n} \frac{d}{N_j}$, and N_j is the out-degree node v_j for $j \in [1, n]$.

Remark. Bounded staleness forbids fastest workers to outpace the slowest ones by more than c steps. It is mainly to ensure the correctness and convergence of CF [47, 84]. By Theorem 4.3, CC and SSSP are not constrained by bounded staleness; conditions T1, T2 and T3 suffice to guarantee their convergence and correctness. Hence fast workers can move ahead any number of rounds without affecting their correctness and convergence. One can show that PageRank does not need bounded staleness either, since for each path $p \in \mathcal{P}$, $p(v)$ can be added to P_v at most once (see above).

6 PARAMETERS ESTIMATION FOR DYNAMIC ADJUSTMENT OF AAP

As shown in Section 3, we use running time and message arrival rate to dynamically adjust delay sketch and relative progress of workers. We propose to use a random forest model to estimate running time and show how to estimate message arrival rate. Below we first formulate running time estimation as a regression problem in machine learning (Section 6.1). Adopting mean squared relative error as the evaluation metric, we then show how to predict running time of CC, SSSP, PageRank and CF (Section 6.2). Finally, we predict message arrival rate based on localized mean (Section 6.3).

6.1 Problem Formulation

Below we first formulate runtime prediction as a regression problem. We then identify constraints for real-life predictors, examine regression methods for runtime prediction [49] under the constraints, and conclude that random forest regression strikes a balance between efficiency and accuracy. We also present a guideline for feature selection in runtime prediction.

Formulation. Given running information X_j of worker j before triggering its next round, we want to train a predictor P that is able to estimate the running time $\tilde{t_j}$ of the coming round. Since the output of P is a real number (time in milliseconds), this problem can be formulated as a regression problem in machine learning, as shown in the following equation:

$$\tilde{t_j} = \mathsf{P}(X_j). \tag{5}$$

The prediction can be divided into two parts. (a) The first part is to collect the running information X_j (*a.k.a. feature extraction* in machine learning) of the next round. We use X_j to encode the running status of the next round, so that predictor can estimate the running time. (b) The second part aims to design and train a suitable predictor P from candidate regression models.

We focus on how to predict the running time and message arrival rate for incremental computation of IncEval, since IncEval iterates until it converges, while PEval runs only once.

Three criteria. In addition to the common concerns about a machine learning model, *e.g.,* prediction accuracy and over-fit avoidance, the following three constraints are crucial to predictors in AAP.

Computation constraint. The prediction should be highly efficient with little overhead. A solution is not practical if it takes as long as the entire incremental computation process.

Training constraint. This constraint is twofold. First, due to limited training data available, the predictor P should be able to get well-trained given a small training set. Second, the training process should be fast; in certain cases the training must complete within several IncEval rounds.

Hyper-parameter constraint. The number of hyper-parameters in P should be small, since we hope to train a general and robust predictor for various data input. Otherwise, one has to fine-tune the model for each specific dataset, which is time-consuming and may increase the risk of over-fitting.

Method selection. We compare four regression models for algorithm runtime prediction. Based on the comparison, we select the best for parameter prediction of AAP. To analyze the computational complexity of each method, we assume that there exist n training samples. The input feature vector of each sample is denoted by X, and the dimension of X is p.

Ridge Regression (RR). For the three criteria above, RR satisfies the computation constraint only, since it requires at most tens of multiply-accumulate operations during prediction. Moreover, the training complexity is $O(p^3)$, which is only related to the vector dimension [49, 63]. However, we can hardly train a robust RR model with few training samples, because its performance can easily deteriorate in the presence of noisy training samples in a small training set [49]. In addition, it is rather hard to decide the hyper-parameters of RR. For example, one has to train and test an RR model multiple times to figure out a suitable order of the RR polynomial function.

Neural Networks (NN). While NN has a strong feature extraction ability and often offers high prediction accuracy, it is not suitable for runtime prediction in AAP for the following reasons [42]. (1) A NN-based regression model incurs heavy computation, mainly due to the non-linear activation functions in each layer. These activation functions often involve exponential computations, which slows down the prediction process. (2) The training process of NN is typically time-consuming since it needs to back propagate the loss values layer by layer. In addition, a large number of training samples are required, which cannot be provided by limited query log information from the initial rounds. (3) It is tricky to design the NN architecture, and there are tens of hyper-parameters (such as learning rate in each layer) for users to decide; this hampers its applications.

Gaussian Process Regression (GPR). Although it is easy to train a GPR model with few hyper-parameters, it is computationally expensive: it takes $O(n^3)$ time due to the inversion of a $n \times n$ matrix. In addition, the complexity of one prediction is $O(n^2)$, which slows down the system [49].

Random Forest Regression (RFR). RFR [22, 41] meets the three criteria above. (1) For an RFR containing T regression trees with average tree depth d, the prediction complexity is $O(T \cdot \log d)$ in the best case (when all regression trees are balanced) and $O(T \cdot d)$ in the worst [49]. The depth d is approximately $\log n$ in the best case (balanced) and n in the worst. (2) Similarly, the training complexity of RFR is from $O(T \cdot p \cdot n^2 \log n)$ to $O(T \cdot p \cdot n \log^2 n)$ [41]. In fact, as empirically verified in [49], while most regression trees are not perfectly balanced, the training complexity is much closer to the best than to the worst. This enables RFR to be fast trained online.

For the hyper-parameter constraint, (3) RFR is robust during training, and there exists typically only one hyper-parameter to choose, *i.e.,* the number of regression trees. Other hyper-parameters take default in many application packages and perform well in prediction [20, 32]. Its robustness comes from the aggregation of outputs from each tree, which overcomes the sensitivity to small changes in the training data and avoids over-fitting [22]. Although the prediction accuracy increases as the number of regression trees grows [49], the computational cost grows linearly. Thus, we use only 10 regression trees for each RFR throughout our experiments to keep computational costs low.

Putting these together, we conclude that random forest regression offers the best model for parameter prediction in AAP. Below we show how to use random forest regression in AAP.

Guidelines of runtime prediction. Before giving the construction of RFR predictors, we present a guidance for training strategy and feature selection of our runtime prediction solution.

Query Dependency and Training Strategy. An algorithm is *query independent* if on a dataset (*e.g.,* graph G, training set and accuracy threshold), there exists a unique query. Otherwise the algorithm is *query dependent*. For example, PageRank, CC and CF are query independent, since on a given dataset there exists only one query. In contrast, SSSP is query dependent since given a graph G, there are multiple SSSP queries depending on the choices of a source vertex s to start with.

The query dependency of an algorithm decides the training strategy of its predictor P as follows. (1) For query dependent algorithms, we can first collect adequate runtime log information from previous runs, and train the prediction model offline. Then we deploy this pre-trained model for future prediction. (2) In contrast, for query independent algorithms, we have to train the prediction model immediately based on the running log information from the initial rounds. Then we utilize the newly trained model to predict the runtime of the following rounds. Thus, query-independent algorithms require fast model training given limited amount of training data.

Feature Extraction Guideline. The computation time of each fragment in each round is mainly decided by the scale of the fragment, the fragment's topological structure and the message contents [70]. Thus, as a clue to the runtime prediction, the features extracted from the program running log must incorporate such information; this is applicable to all graph algorithms in Section 5. We will detail the implementation of feature extraction in Section 6.2 based on this guidance.

6.2 Running Time Prediction

We next develop a running time predictor P for AAP, and apply it to pageRank, CF, SSSP and CC.

Notations. We start with some notations. The feature vector input for the predictor P is denoted by $X_j = [x_1, x_2, \ldots, x_M]$, where x_i is the i-th value in X_j, and M is the number of features used. For each training/test vector input X_j, there exists a corresponding true runtime t_j, and $d_j = \{X_j, t_j\}$ denotes this data pair. The training set is denoted by $\mathcal{D} = [d_1^{\mathcal{D}}, d_2^{\mathcal{D}}, \ldots, d_N^{\mathcal{D}}]$, and the test set is $\mathcal{E} = [d_1^{\mathcal{E}}, d_2^{\mathcal{E}}, \ldots, d_S^{\mathcal{E}}]$. Each feature vector is extracted from one line in the query log, where the program running information is recorded. Note that we organize both training data $\mathcal{D}$ and test data $\mathcal{E}$ in sequence according to the timestamp of each vector in the query log. With this time-elapse sequence, we can simulate the running process of the system in our experiments. There exist T regression trees in an RFR model, and the number of all data pairs is denoted by $R = N + S$. The message received by worker j is denoted by $M_j = [f_1, f_2, \ldots, f_\Omega]$, where each f_i is the ID of a border node. Each node ID is a unique integer in the whole graph, and the maximum of all node IDs is Ψ. We define an indicator function Φ_k as follows: it equals 1 if k is true, and 0 otherwise.

Performance measure. In order to evaluate the performance of runtime prediction models in AAP, we employ the *mean squared relative error* (MSRE) [65] as the evaluation metric. For S test samples, the MSRE of the prediction model P is given by Equation (6).

$$\text{MSRE} = \frac{\sum_{j=1}^{S} (\frac{t_j - \tilde{t_j}}{t_j})^2}{S} = \frac{\sum_{j=1}^{S} (\frac{t_j - P(X_j)}{t_j})^2}{S}. \tag{6}$$

The main reason for choosing MSRE is as follows. Compared with mean squared error (MSE) and mean absolute error (MAE), MSRE utilizes the relative error divided by the true running time, which can reduce the impact of errors from tests with long runtime. For example, suppose that there exist three true running times, namely, 100 ms, 500ms and 1200ms, and the prediction of the model P for each run is 80ms, 600ms and 800ms, respectively. If we use squared error to quantify

x_1	x_2	x_3	x_4	x_5	x_6	x_7	Z_j
Fragment ID	Total Node Number	Border Node Number	Edge Number	Proportion of Border Nodes	Graph Density	Message Number	Message Embedding

Table 2. Each dimension's meaning in the feature vector X_j for runtime prediction.

the performance of the model P, the errors for each test are 400, 600 and 160,000, respectively, and the MSE of the model P is 56,800. In this case, the error of the third test (1200ms) contributes the largest to the MSE, which conceals the errors of the first two. In contrast, if we use relative error instead, the errors for each test are 0.04, 0.04 and 0.11, where the numerical difference is much smaller. Thus, the error of short true runtime can be better captured by MSRE.

Stages of predictor. For each algorithm running under AAP, we design a two-stage running time prediction solution, including feature extraction and regression prediction.

(1) Feature extraction. In the first stage, we use a vector to represent the program running information of a round. As shown in Table 2, the feature vector X_j for worker P_j consists of 6 scalar values to encode the fragment scale and topological structure, namely, the fragment ID (x_1), the number of nodes (x_2), the number of border nodes (x_3), the number of edges (x_4), the proportion of border nodes ($x_5 = \frac{x_3}{x_2}$), and the graph density of this fragment ($x_6 = \frac{x_4}{x_2(x_2-1)}$). This is applicable to all algorithms in Section 5. Observe the following.

(a) We use fragment ID as a feature to represent the topological structure of a graph fragment. Note that using the number of nodes and edges can hardly represent the fragment's topology, while it is too costly to compute a vector representation of the topology. Thus, as a trade-off between prediction effectiveness and efficiency, the fragment ID is incorporated into the feature vector.

(b) In addition, to represent the message passed each round, we use x_7 to denote the number of messages, and design a vector $Z_j = [z_1, z_2, \ldots, z_L]$ as the embedding of all messages, where L is a hyper-parameter of the predictor P. The feature vector then becomes $X_j = [x_1, \ldots, x_7, Z_j]$. The message feature vector Z_j aggregates message f_k by mapping the node IDs with equal intervals and accumulating the number of each mapping. Here the mapping interval is $I = \lceil \frac{\Psi}{L} \rceil$, and the ith scalar value in Z_j can be computed by $z_i = \sum_{k=1}^{\Omega} \Phi_{\lceil \frac{f_k}{I} \rceil = i}$, where L is a parameter, and Φ is the indicator function defined earlier. A large L creates long and informative feature vectors of the messages, which can increase the prediction accuracy of RFR models. However, long vectors also increase the training time and prediction overhead. Thus, one needs to strike a balance between prediction accuracy and efficiency when choosing the value of L. For example, for a graph containing 100 nodes, where their node IDs range from 1 to 100 ($\Psi = 100$), if a message comes as $F = [3, 87, 26, 47, 33]$ and $L = 4$, then the mapping interval I is 25, and the corresponding message vector is $Z = [1, 3, 0, 1]$.

The main advantage of this simple mapping-based feature extraction is its efficiency, which is crucial when processing large-scale graphs. This method has a prerequisite: nodes that are "closely connected" should be close in node ID, where two nodes are closely connected if they are within a small number of hops of each other. This prerequisite ensures the following: inter-connected border nodes that may eventually trigger similar updates are mapped to the same scalar value in Z, which can later be captured by the prediction model. As will be seen in Section 8, this prerequisite can be easily satisfied. Since all these values can be directly accessed or calculated with few basic operations by the worker, the computation overhead of the feature extraction is quite small.

(2) Regression prediction. In the second stage, we provide the pre-trained regression model with the vector as input, and obtain runtime prediction. We select random forest regression as the base of the prediction model with adjustments for specific cases. Here Equation (6) serves as the loss function

in the regression model training, which balances the impacts from long predictions and short predictions in the total loss value. We follow the RFR training methods developed in [49], where more training details can be found. The structure of each regression tree is decided by the recursive split learning process, and we use cost-complexity pruning with 10-fold cross-validation to obtain a trade-off forest structure between the computation cost and the prediction accuracy [41, 49].

Predictors for graph algorithms. As remarked in Section 6.1, the training for query independent algorithms has to be done within initial rounds given limited training samples, while prediction models for query dependent algorithms can be trained offline with previous query logs. In light of this, below we give our running time prediction solution to each of PageRank, CF, SSSP and CC.

PageRank *and* CF. The model training and runtime prediction have to be completed within one query. Thus, we train the RFR model with the query log information from the initial N rounds, and use the newly-trained model to predict the running time of the following rounds.

In our experimental study, we find that the MSRE for PageRank and CF is very small (see Tables 3 and 6 in Section 8). This verifies the effectiveness of our solution. In addition, the mean prediction time (in 10^{-3} ms) is neglectable compared with the mean running time (in 10^2 ms) of IncEval rounds, showing the high prediction speed of our method. The learning process is also quite efficient.

SSSP. Since the training and prediction do not have to be completed within one query, we run different queries for multiple times and use the query logs to train the model. Then we deploy the pre-trained model to predict the algorithm runtime when new queries come.

Different from PageRank and CF, after the prediction stage, instead of using the output of the model P as the final result, we calibrate the prediction of P based on the initial IncEval true running time t_1. The main reason for this calibration is that the vanilla RFR can hardly predict accurate time, but it can capture the changing trend of the running time. In light of this, if the i-th output of P is $\tilde{t}_i$, the final runtime prediction after calibration is taken as $t_1 \frac{\tilde{t}_i}{\tilde{t}_1}$. We find that with this ratio-based calibration using the true running time as the start point, we can better approximate the runtime (see Table 5 in Section 8). Moreover, the overhead of prediction is small and neglectable.

Note that the message embedding strategy is relatively simple for representing the complex message contents in SSSP. With our simplified embedding method, rounds with different running times and different messages may be projected to the same vector in the hypothesis space [19], which hinders the learning of the RFR model. Nonetheless, we overcome this limitation by using the ratio-based calibration, which effectively improved the raw RFR model.

CC. Similar to SSSP, we also calibrate the raw RFR outputs using the initial true running value. However, unlike SSSP, the training and testing of the prediction method have to be completed within one query, because CC is query independent. Thus, the training policy of PageRank is employed for CC. That is, log information from initial IncEval rounds is used for training and the learned predictor is applied to the rest of the rounds. As will be seen shortly, our method for CC is efficient and offers accurate prediction (see Table 4 in Section 8). This demonstrates that our simple message embedding is expressive enough to predict the running time of CC.

6.3 Message Arrival Rate Estimation

We predict the message arrival rate in the next round based on localized mean. More specifically, we count the total number of messages received by worker i in the last τ seconds, and use the mean message arrival rate in this time window as the rate estimation in the following round.

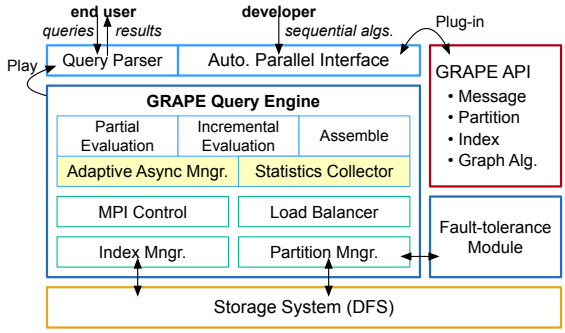


Fig. 5. GRAPE+ Architecture

The reason for adopting this simple averaging estimation is twofold. First, the overall variation of the message arrival rate is continuous with few abrupt changes. Thus, the gap between the estimation and the true arrival rate in the next round is small. This is verified by our experimental study (see Figure 11 in Section 8). Second, although state-of-the-art sequence learning models, *e.g.,* RNN [61] and LSTM [48], are quite accurate in sequential prediction, those models are too expensive to deploy. In contrast, our estimation based on localized mean incurs little computation overhead.

7 IMPLEMENTATION OF GRAPE+

As proof of concept, we implemented GRAPE+ from scratch, in C++ with 17000 lines of code.

The architecture of GRAPE+ is shown in Figure 5, to extend GRAPE by supporting AAP. Its top layer provides interfaces for developers to register their PIE programs, and for end users to run registered PIE programs. The core of GRAPE+ is its engine, to generate parallel evaluation plans. It schedules workload for working threads to carry out evaluation plans. Underlying the engine are several components, including (1) an MPI controller [7] to handle message passing, (2) a load balancer to evenly distribute workload, (3) an index manager, and (4) a partition manager. GRAPE+ employs distributed file systems, *e.g.,* NFS, AWS S3 and HDFS, to store graph data.

GRAPE+ extends GRAPE by supporting the following.

Adaptive asynchronization manager. As opposed to GRAPE, GRAPE+ dynamically adjusts relative progress of workers. This is carried out by a scheduler in the engine. Based on statistics collected (see below), the scheduler adjusts parameters and decides which threads to suspend or run, to allocate resources to useful computations. It is based on runtime and arrival rate estimation, as described in Section 6 and will be illustrated in Section 8. In particular, the engine allocates communication channels between workers, buffers messages generated, packages the messages into segments, and sends a segment each time. It further reduces costs by overlapping data transfer and computation.

Statistics collector. During a run of a PIE program, the collector gathers information for each worker, *e.g.,* the amount of messages exchanged, the evaluation time in each round, historical data for a query workload, and the impact of the last parameter adjustment.

Fault tolerance. Asynchronous runs of GRAPE+ make it harder to identify a consistent state to rollback in case of failures. Hence as opposed to GRAPE, GRAPE+ adapts Chandy-Lamport snapshots [24] for checkpoints. The master broadcasts a checkpoint request with a token. Upon receiving the request, each worker ignores the request if it has already held the token. Otherwise, it snapshots its current state before sending any messages. The token is attached to its following messages. Messages that arrive late without the token are added to the last snapshot. This gets us

a consistent checkpointed state, including all messages passed asynchronously. When a failure happens, the master resumes the computation from the latest checkpoint, and continues by processing saved messages [66]. The confined recovery techniques of [79] can also be deployed on GRAPE+.

When deploying GRAPE+ in a POC scenario that provides continuous online payment services, we found that it took about 40 seconds to get a snapshot of the entire state, and 20 seconds to recover from failure of one worker. In contrast, it took 40 minutes to start the system and load the graph.

Consistency. Each worker P_i uses a buffer $\mathbb{B}_{\bar{x}_i}$ to store incoming messages, which is incrementally expanded. As remarked in Section 3, GRAPE+ allows users to provide an aggregate function f_{aggr} to resolve conflicts when a status variable receives multiple values from different workers. The only race condition is that when old messages are removed from $\mathbb{B}_{\bar{x}_i}$ by IncEval (see Section 3), the deletion is atomic. Thus consistency control of GRAPE+ is not much harder than that of GRAPE.

8 EXPERIMENTAL STUDY

Using real-life and synthetic graphs, we conducted seven sets of experiments to evaluate the (1) efficiency, (2) communication cost, (3) scale-up of GRAPE+; (4) the effectiveness of AAP and the impact of graph partitioning strategies on its performance; the accuracy and efficiency of our estimation of (5) runtime and (6) message arrival rate. Finally, (7) we gave case studies of the effectiveness of dynamic adjustment. We compared the performance of GRAPE+ with (a) Giraph [8], synchronized GraphLab$_{\text{sync}}$ [43], Galois [29, 64],Pregel+ [87], and Husky [88] under BSP, (b) asynchronized GraphLab$_{\text{async}}$, GiraphUC [45], Maiter [91] and TDataflow (Timely-Dataflow) [9, 62] under AP, (c) Petuum [84] under SSP, (d) PowerSwitch [83] under Hsync, (e) GRAPE+ simulations of BSP, AP and SSP, denoted by GRAPE+$_{\text{BSP}}$, GRAPE+$_{\text{AP}}$, GRAPE+$_{\text{SSP}}$, respectively. We find that GraphLab$_{\text{async}}$, GraphLab$_{\text{sync}}$, PowerSwitch, Husky, Galois, Pregel+ and GRAPE+ outperform the other systems. Indeed, Table 1 shows the performance of SSSP and PageRank of the systems with 192 workers; results on the other algorithms are consistent. Hence we only report the performance of these seven systems in details. For CF, we also compared with Petuum [84], and find that GRAPE+ and Petuum perform the best among the competitors. For systems with multiple versions, the specific versions used are: (a) GraphLab v2.2[1], (b) PowerSwitch v1.0[2], (c) Galois v4.0[3], (d) Giraph v1.1.0[4], (e) Husky v0.1.2[5], and (f) Petuum v1.1[6]. In all the experiments we also evaluated GRAPE+$_{\text{BSP}}$, GRAPE+$_{\text{AP}}$ and GRAPE+$_{\text{SSP}}$. Note that GRAPE [38] is essentially GRAPE+$_{\text{BSP}}$, a special setting of GRAPE+.

Experimental setting. We used the following real-life and synthetic graphs.

Graphs. We used six real-life graphs of different types, such that each algorithm was evaluated with at least two real-life graphs. These include (1) Friendster [6], a social network with 65 million users and 1.8 billion links; we randomly assigned weights to test shortest path SSSP; (2) traffic [3], an (undirected) US road network with 23 million nodes (locations) and 58 million edges; (3) UKWeb [2], a Web graph with 133 million nodes and 5 billion edges; and (4) ClueWeb12 [5], a Web page network with 733 million pages and 42 billion links. We also used two recommendation networks (bipartite graphs) to evaluate collaborative filtering CF, namely, (5) movieLens [4], with 20 million movie

[1]https://github.com/jegonzal/PowerGraph/tree/v2.2
[2]https://github.com/xiechenny/powerswitch/releases/tag/ver1.0
[3]https://github.com/IntelligentSoftwareSystems/Galois/tree/release-4.0
[4]https://github.com/apache/giraph/releases/tag/release-1.1.0
[5]https://github.com/husky-team/husky/releases/tag/v0.1.2
[6]https://github.com/sailing-pmls/bosen/releases/tag/v1.1.0

ratings (as weighted edges) between 138000 users and 27000 movies; and (6) Netflix [12], with 100 million ratings between 17770 movies and 480000 customers.

To test the scalability of GRAPE+, we developed a generator to produce synthetic graphs $G = (V, E, L)$ controlled by the numbers of nodes $|V|$ (up to 300 million) and edges $|E|$ (up to 10 billion).

The synthetic graphs and some real-life graphs, *e.g.,* UKWeb and ClueWeb12, are too large to fit in a single machine. Parallel processing is a must for processing these graphs.

Queries. For SSSP, we sampled 10 source nodes for each graph G used such that each node has paths to or from at least 90% of the nodes in G, and constructed an SSSP query for each of them.

Algorithms. We evaluated SSSP, CC, PageRank and CF over GRAPE+ by using their PIE programs developed in Sections 2 and 5. We used "default" code provided by the competitor systems when available. More specifically, for instance, (a) for Galois we used its pull version of SSSP, CC and PageRank; (b) for Petuum we used its latest model-parallel implementation of CF; (c) for Pregel+ we used ordinary mode version of SSSP, CC, and PageRank; and (d) for TDataflow we used the version of PageRank in [11], and slightly modified the original codes of CC to load the graphs from files. Otherwise we made our best efforts to develop "optimal" algorithms, *e.g.,* SSSP for TDataflow.

We used XtraPuLP [72] as the default graph partition strategy, which is widely used in practice. To evaluate the impact of stragglers, we randomly reshuffled a small portion of each partitioned input graph when conducting the evaluation, and made the graphs skewed. For systems that do not support external graph partitioning, we used their own default graph partitioning strategies, which include (1) Random[7] for PowerSwitch, GraphLab and its variants, and TDataflow, (2) consistent hashing for Husky [88], and (3) OEC (Outdoing Edge-cut) [29] for Galois. We did not do the reshuffling for these systems.

We deployed the systems on an HPC cluster. For each experiment, we used up to 20 servers, each with 16 threads of 2.40GHz, and 128GB memory. On each thread, a GRAPE+ worker is deployed. Each system was configured according to its recommended setting. To ensure fairness and reproducibility, the resource used in each set of experiment is *the same for all systems tested.*

We remark that we have only used the default and recommended setting for each system in our experiments to compare the "average" performance over all graphs. As shown in [35], the choice of graph partitioning strategies should be "application driven", *i.e.,* graph partitioning strategies and system configurations could be fine-tuned to further improve the performance. As a consequence, it should be remarked that some of the 12 different systems might work better on some of 6 real-life graphs adopted in our experiments when their settings are further optimized.

We ran each experiment 5 times. The average is reported here.

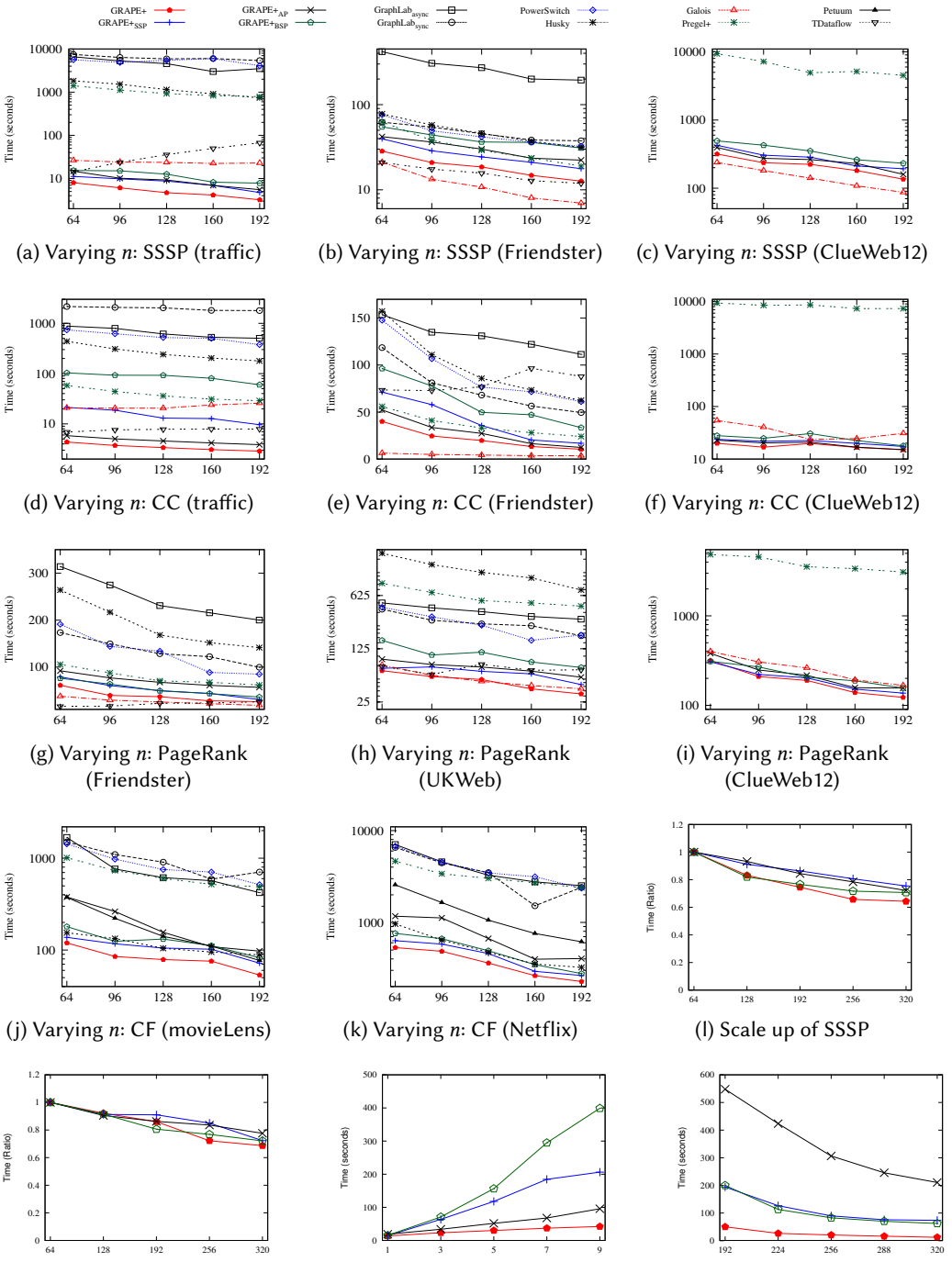
Experimental results. We next report our findings.

Exp-1: Efficiency. We first evaluated the efficiency of GRAPE+ by varying the number n of workers used, from 64 to 192. We evaluated (a) SSSP and CC with real-life graphs traffic, Friendster and ClueWeb12; (b) PageRank with Friendster, UKWeb and ClueWeb12; and (c) CF with movieLens and Netflix, based on applications of these algorithms in transportation network analysis, social network analysis, Web page classification, Web rating and recommendation.

(1) SSSP. Figures 6a-6c report the performance of SSSP. We can see the following.

(a) GRAPE+ outperforms its competitors in most cases. Over real-life graph traffic (resp. Friendster)

[7] https://github.com/jegonzal/PowerGraph/blob/master/src/graphlab/options/graph_help.txt



(a) Varying n: SSSP (traffic)
(b) Varying n: SSSP (Friendster)
(c) Varying n: SSSP (ClueWeb12)
(d) Varying n: CC (traffic)
(e) Varying n: CC (Friendster)
(f) Varying n: CC (ClueWeb12)
(g) Varying n: PageRank (Friendster)
(h) Varying n: PageRank (UKWeb)
(i) Varying n: PageRank (ClueWeb12)
(j) Varying n: CF (movieLens)
(k) Varying n: CF (Netflix)
(l) Scale up of SSSP
(m) Scale up of PageRank
(n) Impact of partitioning
(o) Speedup by AAP

Fig. 6. Performance Evaluation

and with 192 workers, GRAPE+ is on average 1673.0 (resp. 3.0), 1085.0 (resp. 15.0), 1270.0 (resp. 2.6), 227.4 (resp. 2.4) and 244.0 (resp. 1.1) times faster than synchronized GraphLab$_{sync}$, asynchronized GraphLab$_{async}$, hybrid PowerSwitch, Husky and Pregel+, respectively. On traffic and with 192 workers, GRAPE+ is on average 9.3 times faster than TDataflow; but on Friendster, TDataflow performs slightly better than GRAPE+; for instance, TDataflow takes 11.9 seconds with 192 workers on Friendster, while GRAPE+ takes 12.6 seconds. This is because to evaluate the effect of stragglers, we made the graph partitions used by GRAPE+ more skewed than the default ones used by TDataflow. Nonetheless, GRAPE+ performs better than TDataflow on graphs such as traffic, since the diameter of traffic is large, and on such graphs, incremental computation of GRAPE+ is more effective and converges in less rounds than the computation of TDataflow. On ClueWeb12, we only report the results for Pregel+, (distributed version of) Galois, GRAPE+ and its variants, since it is too large to run on other competitors. The problem has been reported earlier in [54, 93]. On ClueWeb12, GRAPE+ consistently outperforms Pregel+ by 171.4 times on average.

On Friendster and ClueWeb12, Galois performs slightly better than GRAPE+. For instance, it takes 21.1 (resp. 238.2) seconds on Friendster (resp. ClueWeb12) with 64 workers, while GRAPE+ takes 28.4 and 317.2 seconds, respectively. There are two reasons for this. (a) Galois adopts a communication-optimizing substrate to explore structural and temporal invariants of graph partitions, which reduces its communication cost [29]. (b) For the same reason given above, since we make graph partitions used by GRAPE+ more skewed, Galois performs better than GRAPE+ on Friendster; but due to the large diameter of traffic, GRAPE+ is more effective than Galois.

The performance gain of GRAPE+ comes from the following: (i) efficient resource utilization by dynamically adjusting relative progress of workers under AAP; (ii) reduction of redundant computation and communication by the use of incremental IncEval; and (iii) optimization inherited from strategies that are available for sequential algorithms. Note that under BSP, AP and SSP, GRAPE+$_{BSP}$, GRAPE+$_{AP}$ and GRAPE+$_{SSP}$ can still benefit from (ii) and (iii).

As an example, GraphLab$_{sync}$ took 34 and 10749 rounds over Friendster and traffic, respectively, while by using IncEval, GRAPE+$_{BSP}$ and GRAPE+$_{SSP}$ took 21 and 30 (resp. 31 and 42) rounds, respectively, and hence reduced synchronization barriers and communication costs. In addition, GRAPE+ inherits the optimization techniques from single-machine algorithm (our familiar sequential Dijkstra algorithm) by employing priority queues to prioritize vertex processing; in contrast, this optimization strategy is beyond the capacity of the vertex-centric systems.

(b) GRAPE+ is on average 2.3, 1.8, and 1.6 (resp. 2.2, 1.6, and 1.4) times faster than GRAPE+$_{BSP}$, GRAPE+$_{AP}$ and GRAPE+$_{SSP}$ over traffic (resp. Friendster), up to 2.7, 2.0 and 1.9 times, respectively. Since GRAPE+, GRAPE+$_{BSP}$, GRAPE+$_{AP}$ and GRAPE+$_{SSP}$ are *the same system* under different modes, the gap reflects the effectiveness of *different models*. We find that the idle waiting time of AAP is 29.4% and 41.7% of that of BSP and SSP, respectively. Moreover, when measuring stale computation in terms of the total extra computation and communication time over BSP, the stale computation of AAP accounts for 51.9% of that of AP, respectively. These verify the effectiveness of AAP by dynamically adjusting relative progress of different workers.

(c) GRAPE+ takes less time when the number n of workers increases. It is on average 2.5, 2.3 and 2.3 times faster on traffic, Friendster and ClueWeb12, respectively, when n varies from 64 to 192. That is, AAP makes effective use of parallelism by reducing stragglers and stale computations.

(2) CC. As reported in Figures 6d-6f on traffic, Friendster and ClueWeb12, respectively, (a) GRAPE+ significantly outperforms other systems, namely GraphLab$_{sync}$, GraphLab$_{async}$, PowerSwitch, Husky,

Pregel+ and TDataflow. When $n = 192$, GRAPE+ is on average 313.0, 93.0, 68.0, 34.0, 167.0 and 1.1 times faster than the six systems, respectively. Compared with Galois, GRAPE+ is 8.9 and 2.1 times faster on traffic and ClueWeb12, respectively. This is because GRAPE+ processes only affected vertices in IncEval by capitalizing on auxiliary indices that were inherited from sequential algorithms, which are more effective than label propagation used in Galois. On Friendster, however, Galois does better than GRAPE+ since the diameter of Friendster is small; on such graphs label propagation of Galois works as well as the indices of GRAPE+, and Galois speeds up the computation by using a communication-optimizing substrate. (b) GRAPE+ is faster than its variants under BSP, AP and SSP on average by 9.6, 1.3 and 2.4 times, up to 27.4, 1.5 and 5.0 times, respectively. This again verifies the performance gain of AAP by reducing both stragglers and stale computations. (c) GRAPE+ scales well with the number of workers used: it is on average 2.7 times faster when n varies from 64 to 192.

(3) PageRank. As shown in Figures 6g-6i over real-life graphs Friendster, UKWeb and ClueWeb12, respectively, when $n = 192$, (a) on Friendster and UKWeb, GRAPE+ is on average 5.0, 9.0, 5.0, 14.3, 13.6 and 3.6 times faster than GraphLab$_{\text{sync}}$, GraphLab$_{\text{async}}$, PowerSwitch, Husky, Pregel+ and TDataflow, respectively. (b) GRAPE+ has performance comparable to that of Galois: it does slightly better over UKWeb and ClueWeb12 but slightly worse over Friendster, for the same reason as given above. (c) GRAPE+ outperforms GRAPE+$_{\text{BSP}}$, GRAPE+$_{\text{AP}}$ and GRAPE+$_{\text{SSP}}$ by 1.6, 1.7 and 1.2 times, respectively, up to 2.5, 2.2 and 1.6 times. This is because GRAPE+ reduces redundant stale computations, especially those of stragglers. We find that on average, stragglers took 101, 76 and 62 rounds under BSP, AP and SSP, respectively, as opposed to 38 rounds under AAP. (d) GRAPE+ is on average 2.3 times faster when n varies from 64 to 192.

(4) CF. We used movieLens [4] and Netflix with training set $|E_T| = 90\%|E|$, as shown in Figures 6j-6k, respectively. (a) GRAPE+ performs the best among all, and Petuum does better than the other competitor systems. On average (b) GRAPE+ is 57.1, 46.1, 48.3, 10.1, 7.4 and 6.2 times faster than GraphLab$_{\text{sync}}$, GraphLab$_{\text{async}}$, PowerSwitch, Pregel+, Husky and Petuum, respectively. Galois failed to run distributed CF, even with its own implementation[8]. (c) GRAPE+ beats GRAPE+$_{\text{BSP}}$, GRAPE+$_{\text{AP}}$ and GRAPE+$_{\text{SSP}}$ by 1.4, 2.1 and 1.2 times, up to 1.7, 3.2 and 1.5 times, respectively. Moreover, (d) GRAPE+ is on average 1.8 times faster when n varies from 64 to 192.

Single-thread. Among the real-life graphs, traffic, Friendster, movieLens and Netflix can fit in a single machine, but not UKWeb and ClueWeb12. On a single machine, it takes 6.7s (resp.157.8s) and 4.3s (resp.88.7s) for SSSP and CC over traffic (resp. Friendster), and 2354.5s for CF over Netflix, respectively. Using 64-192 workers (threads), GRAPE+ is on average from 1.6 to 12.9 times faster than single-thread, depending on how heavy stragglers are. Observe the following. (a) GRAPE+ incurs extra overhead of parallel computation that is not experienced by a single machine, just like other parallel systems. (b) Large graphs such as UKWeb and ClueWeb12 are beyond the capacity of RAM in a single machine, and parallel computation is a must for such graphs.

We also evaluated GRAPE+ against an extension of COST [60] to find out the hardware configuration (the number of cores) required by GRAPE+ to outperform a competent single-threaded implementation. Large graphs such as UKWeb and ClueWeb12 are beyond the capacity of RAM in a single machine; hence to compute the COST of GRAPE+ we converted these graphs to Hilbert curve representation [60], and delta-encoded edges; we used the codes in [10] as the single-threaded implementation. These results are consistent with that of GRAPE reported in [39]. For algorithms conducted in this paper, GRAPE+ achieves speed-up over single-threaded implementations with

[8]https://github.com/IntelligentSoftwareSystems/Galois/issues/39

just 2 or 4 cores over all tested datasets, except that on ClueWeb12 our system GRAPE+ needs 14 cores, which also account for the minimum number of cores we need to load ClueWeb12 into the memory, and run the algorithms. That is, GRAPE+ introduces very small parallelizing overhead. This is because (i) GRAPE and GRAPE+ share the same underlying implementation, and (ii) the overhead for adjusting relative progress introduced by AAP is negligible.

Exp-2: Communication. Following [46], we tracked the total bytes sent by each machine during a run, by monitoring the system file /proc/net/dev. The communication costs of PageRank over Friendster and CC over traffic are reported in Table 1, when 192 workers were used. The results on other algorithms are consistent and hence not shown. These results tell us the following.

(1) On average GRAPE+ ships 7.1%, 3.4%, 17.3%, 42.7%, 55.6% and 15.5% of data shipped by $\text{GraphLab}_{\text{sync}}$, $\text{GraphLab}_{\text{async}}$, PowerSwitch, Husky, Pregel+ and TDataflow, respectively. This is because GRAPE+ (a) reduces stale computations and hence unnecessary traffic, and (b) ships only changed values of update parameters by IncEval by means of incremental evaluation.

(2) On average, Galois ships 37.3% less data than GRAPE+, because Galois exploits structural and temporal invariant of data partitions to optimize communication. Besides, it implements memorization of address translation, which reduces the overhead of conversion in communication between hosts [29]. We will incorporate similar strategies into future versions of GRAPE+ (see Section 9).

(3) The communication cost of GRAPE+ is 1.04X, 78% and 95% compared to that of $\text{GRAPE+}_{\text{BSP}}$, $\text{GRAPE+}_{\text{AP}}$ and $\text{GRAPE+}_{\text{SSP}}$, respectively. Since AAP allows workers with small workload to run faster and have more iterations, the amount of messages may increase. Moreover, workers under AAP additionally exchange their states and statistics to adjust relative speed. Despite these, its communication cost is not much worse than that of BSP, and is better than AP and SSP.

Exp-3: Scale-up of GRAPE+. As observed in [60], the speed-up of a system may degrade when using more workers. Thus we evaluated the scale-up of GRAPE+, which measures the ability to keep similar performance when both the size of graph $G = (|V|, |E|)$ and the number n of workers increase proportionally. We varied n from 96 to 320, and for each n, deployed GRAPE+ over a synthetic graph of size varied from $(60M, 2B)$ to $(300M, 10B)$, proportional to n.

As reported in Figures 6l and 6m for SSSP and PageRank, respectively, GRAPE+ preserves a reasonable scale-up. That is, the overhead of AAP does not weaken the benefit of parallel computation. Despite the overhead for adjusting relative progress, GRAPE+ retains scale-up comparable to that of BSP, AP and SSP. The results on other algorithms are consistent (not shown).

Exp-4: Effectiveness of AAP. To further evaluate the effectiveness of AAP, we tested (a) the impact of graph partitioning on AAP, and (b) the performance of AAP over larger graphs with more workers. We evaluated GRAPE+, $\text{GRAPE+}_{\text{BSP}}$, $\text{GRAPE+}_{\text{AP}}$ and $\text{GRAPE+}_{\text{SSP}}$. We remark that these are *the same system under different modes*, and hence the results are not affected by implementation.

Impact of graph partitioning. Define $r = \|F_{\text{max}}\|/\|F_{\text{median}}\|$, where $\|F_{\text{max}}\|$ and $\|F_{\text{median}}\|$ denote the size of the largest fragment and the median size, respectively, indicating the skewness of a partition.

As shown in Fig. 6n for SSSP over Friendster, in which the x axis is r, (a) different partitions have an impact on the performance of GRAPE+, just like on other parallel graph systems. (b) The more skewed the partition is, the more effective AAP is. Indeed, AAP is more effective with larger r. When $r = 9$, AAP outperforms BSP, AP, SSP by 9.5, 2.3, and 4.9 times, respectively. For a well-balanced partition ($r = 1$), BSP works well since the gap between the runtime of workers is rather small, *i.e.,* the chances of having stragglers are small. In this case AAP works as well as BSP.

AAP *in a large-scale setting*. We tested synthetic graphs with 300 million vertices and 10 billion edges, generated by GTgraph [1] following the power law and the small world property. We used a cluster of up to 320 workers. As shown in Fig. 6o for PageRank, AAP is on average 4.3, 14.7 and 4.7 times faster than BSP, AP and SSP, respectively, up to 5.0, 16.8 and 5.9 times with 320 workers. Compare with Exp-1, AAP is far more effective on larger graphs with more workers, a setting closer to real-life applications, in which stragglers and stale computations are often heavy. In practice, stragglers and stale computations may arise when, *e.g.,* an evenly partitioned graph gets skewed due to updates; or when computation resources like CPU cores and process caches are shared among different applications [30]. Recent study also shows that when a graph is evenly partitioned, the computations may still be skewed due to various computation patterns [16, 35]. These further verify the effectiveness of AAP. The results on other algorithms are consistent (not shown).

Exp-5: Running time estimation. We experimented runtime prediction with pageRank, CC, SSSP and CF, based on random forest regression (Section 6), using the same real-life graphs as above. For each of these four algorithm, we report the results on one dataset to demonstrate the performance of our runtime estimation, covering Web page graphs (UKWeb), road networks (traffic), social networks (Friendster), and recommendation networks (Netflix). Estimation results on other graphs, covered by the four types above, are similar and thus are omitted.

The training/test samples were extracted from the query log generated in each round of each worker. Note that each worker is in charge of one fragment. Thus while the number of rounds on each fragment is small, the total number of training/test samples is adequate for the model. For example, if we assign 64 fragments to 64 workers, each worker processes one fragment. When each fragment runs for 10 rounds on average, the total number of training/test samples is $640 = 64 \times 10$.

The number of regression trees in each model was set to be 10. We report both the training time and the mean prediction time. The baseline prediction method estimates the runtime using the mean running time of all previous rounds. Times are all in milliseconds. In order to assure the closeness prerequisite for message embedding (Section 6.2), the node ID in each graph is assigned by breadth-first search, ensuring that closely connected nodes are usually close in node ID [73].

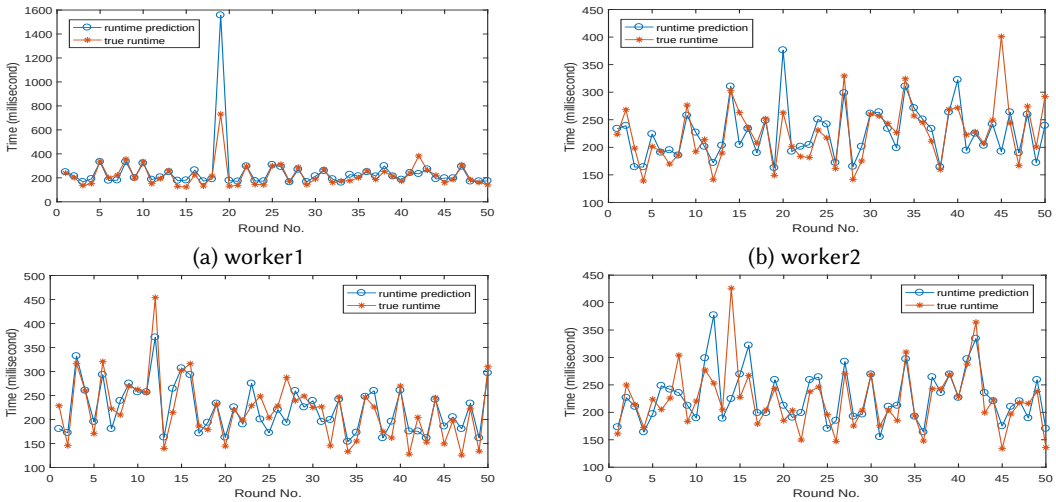
Based on [82], we implemented our RFR-based predictors on the same HPC cluster as the other experiments, using C++. The master P_0 was responsible for query log collection and model training. The training of each model was also conducted in parallel, where we used 20 threads to train each model. After the training process, the prediction model was deployed on each thread of a worker. The length L of each message feature vector was set as 300 by default unless stated otherwise. Each experiment was run for 5 times and the average is reported here.

(1) PageRank. Following the prediction method of Section 6.2, we used the initial 10% of the query log as the training data ($\frac{N}{R} = 10\%$) and the rest for testing. Since PageRank is query-independent, both the training data and test data are from the same query log. Table 3 summarizes the performance of our prediction method on UKWeb when the number of fragments varied from 64 to 192.

From the results we can see the following. (i) The training and prediction overhead of our model is small, and is far less even than the time taken by a single IncEval round. (ii) Our prediction accuracy is far better than that of the mean estimation baseline. (iii) The training and prediction costs do not increase much when more samples are used. (iv) While more fragments yield more training samples, this does not necessarily give us lower MSRE (see the 192-fragment and 64-fragment tests in Table 3). This is because a random forest regression model is trained by recursively splitting a high-dimensional space into small cells, and it is controlled by a subset of training samples that

Fragment number	Sample number	Training time (ms)	Mean prediction time (ms)	Mean IncEval time (ms)	MSRE	MSRE baseline
64	1432	9.51	5.92×10^{-3}	230.49	0.0135	0.1030
96	2418	17.81	5.02×10^{-3}	147.89	0.0214	0.1192
128	2850	21.73	4.51×10^{-3}	114.17	0.0303	0.1422
160	5035	33.97	5.27×10^{-3}	89.13	0.0255	0.1405
192	5271	35.94	4.11×10^{-3}	73.03	0.0328	0.1507

Table 3. Accuracy and overhead of our runtime prediction method for PageRank-UKWeb



(a) worker1 (b) worker2

(c) worker3 (d) worker4

Fig. 7. Random 50 test results in the 64 fragment experiment (PageRank-UKWeb)

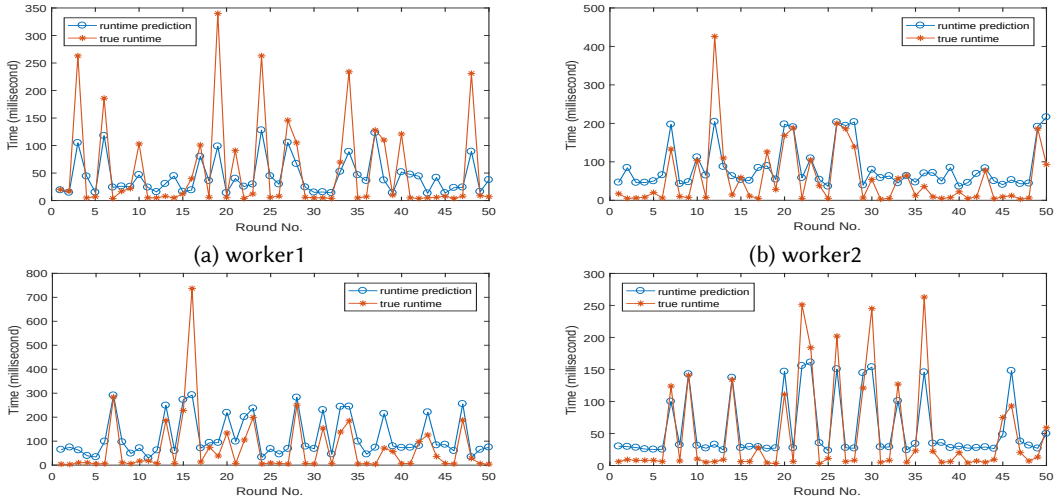
Fragment number	Sample number	Training time (ms)	Mean prediction time (ms)	Mean IncEval time (ms)	MSRE	MSRE baseline
64	37	1.60	1.83×10^{-2}	69.58	0.1397	81.63
96	69	1.58	1.51×10^{-2}	64.41	0.1968	89.51
128	100	1.51	1.47×10^{-2}	45.94	0.1448	98.46
160	138	1.78	1.33×10^{-2}	36.80	0.1810	46.09
192	150	1.95	1.21×10^{-2}	35.83	0.1845	48.37

Table 4. Accuracy and overhead of our runtime prediction method for CC-traffic

decide the boundaries of each cell [20, 22]. When more fragments are adopted, more boundaries are involved and the running time patterns also get more diverse.

To illustrate the results, we visualize in Figure 7 the true runtime and the corresponding prediction in the 64 fragments experiment. Each worker was in charge of one fragment, *i.e.,* there were 64 workers in the 64-fragment setting. We demonstrate the results of randomly selected 50 rounds for four workers only in the test data due to limited space. Note that since PageRank is query-independent, it is possible that PageRank runs less than 50 rounds for one fragment. Nonetheless, since there exist 64 fragments, we can predict the running times for 50 rounds in different fragments.

From Figure 7, we can see that most of the predictions were close to the true value. However, the predictions of relatively large true values (over 400ms) were not very accurate, which were the main sources of the MSRE. This was mainly because the number of training samples with long running time was limited, which hindered the model's learning in this range.



(a) worker1 (b) worker2

(c) worker3 (d) worker4

Fig. 8. Random 50 test results in the 64 fragment experiment (CC-traffic)

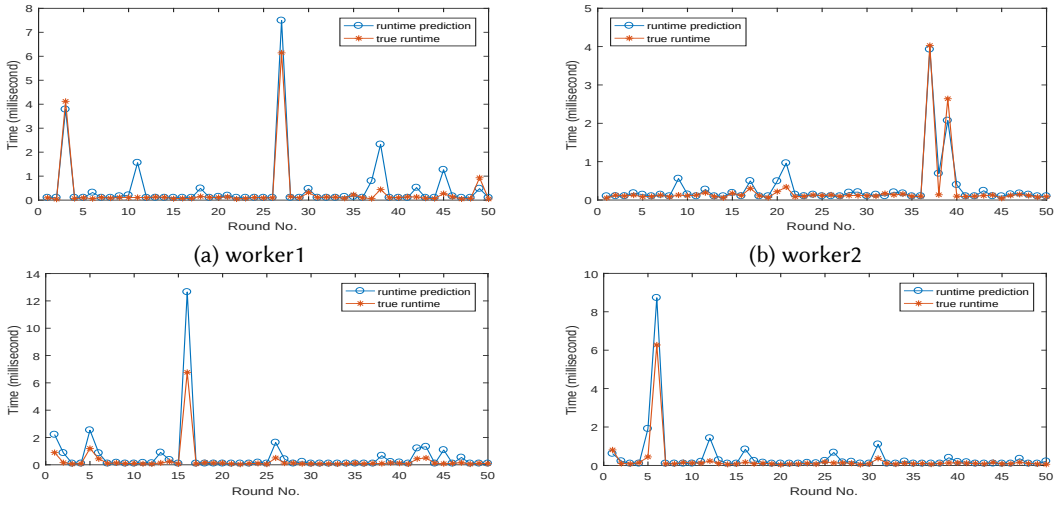
Fragment number	Sample number	Test number	Mean prediction time (ms)	Mean IncEval time (ms)	MSRE	MSRE Baseline
64	367,060	73,412	7.58×10^{-3}	5.25	0.3055	3.16×10^{3}
96	536,675	107,335	6.15×10^{-3}	3.75	0.2452	2.98×10^{3}
128	889,140	177,828	5.79×10^{-3}	2.57	0.2567	1.53×10^{3}
160	1,165,325	233,065	7.18×10^{-3}	1.68	0.4268	2.83×10^{3}
192	1,292,530	258,506	5.79×10^{-3}	1.65	0.5465	2.24×10^{3}

Table 5. Accuracy and overhead of our runtime prediction method for SSSP-Friendster

(2) CC. We also used the initial 10% of the query log as the training data ($\frac{N}{R} = 10\%$) and the rest for testing, all from the same log, since CC are query-independent. The results on traffic are reported in Table 4 and Figure 8. Observe the following. (i) RFR does not violate the training constraint since the training time is less than or comparable to the mean IncEval time, and it is far less than the total CC running time. (ii) Our simple message feature embedding for CC is not effective enough in runtime prediction and hence the RFR model needs more training samples. Like the case of PageRank, the errors were mostly introduced by long rounds (over 200 ms). However, the accuracy of the RFR predictor is still way better than the baseline. (iii) Figure 8 shows that our RFR-based method gives fairly accurate prediction for the changing trend of the running time.

(3) SSSP. Since SSSP is query-dependent, the training data and test data can be generated from different logs. In this experiment, the training and test samples were extracted from 5 previous query logs and 1 test query log, respectively. Since we can train the model offline, we care about the prediction time only. The length L of each message feature vector was set as 500 to provide more precise feature embedding for the RFR model. As shown in Table 5 and Figure 9, the results on Friendster are similar to their counterparts for CC. Its MSRE is at least 3 orders of magnitude better than the baseline. However, the accuracy for long rounds is not as good as for short ones. This is because most training samples were from short rounds (shorter than 1ms), and few large values were involved in training.

(4) CF. Similar to PageRank, CF is query-independent and we have only one log. Thus, we used the initial 10% of the query log as the training data ($\frac{N}{R} = 10\%$) and the rest for testing. The results on Netflix are shown in Table 6 and Figure 10, which verify that our method is able to precisely



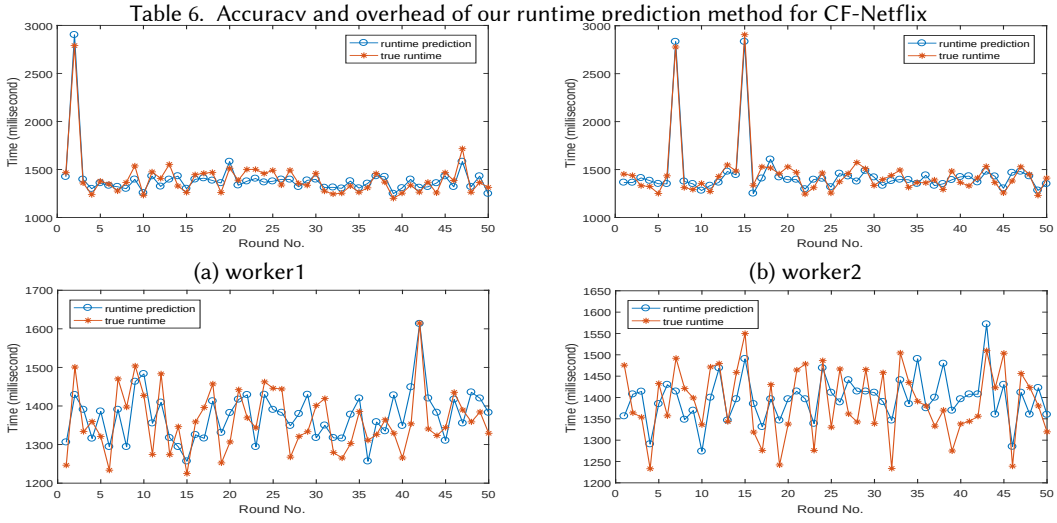
(a) worker1 (b) worker2

(c) worker3 (d) worker4

Fig. 9. Random 50 test results in the 64 fragment experiment (SSSP-Friendster)

Fragment number	Sample number	Training time (ms)	Mean prediction time (ms)	Mean IncEval time (ms)	MSRE	MSRE baseline
64	1,277	26.20	7.36×10^{-3}	1397.70	0.0025	0.0082
96	1,812	40.46	5.51×10^{-3}	871.70	0.0027	0.1289
128	2,300	50.91	5.11×10^{-3}	652.84	0.0031	0.1180
160	1,936	38.17	5.64×10^{-3}	632.48	0.0070	0.0122
192	2,360	61.04	6.45×10^{-3}	531.52	0.0075	0.0136

Table 6. Accuracy and overhead of our runtime prediction method for CF-Netflix

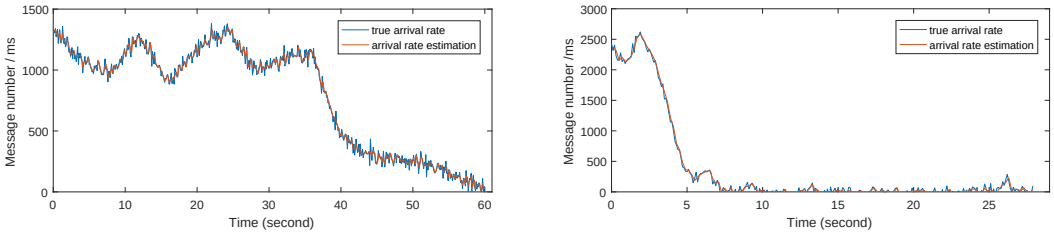


(a) worker1 (b) worker2

(c) worker3 (d) worker4

Fig. 10. Random 50 test results in the 64 fragment experiment (CF-Netflix)

predict the running time for almost all cases. Observe the following. (i) Our method substantially outperforms the baseline, and accurately predicts the runtime of each round. (ii) In most cases, running time is fluctuated around a stable value with few abrupt changes. This explains the relative small MSRE of the baseline compared with its counterparts for SSSP, CC and PageRank.



(a) PageRank (Friendster) (b) SSSP (Friendster)

Fig. 11. Message arrival rate estimation of PageRank and SSSP (fragment number 64).

Exp-6: Estimation of message arrival rate. We also tested our simple estimation method for message arrival rate (Section 6.3). We experimented with pageRank, CC, SSSP and CF. We report the results of PageRank and SSSP in Figure 11, where graphs were partitioned into 64 fragments like in Exp-5. The results for CF and CC are similar and are hence omitted.

Figure 11 shows the mean message arrival rate of each fragment, where the vertical axis is the number of messages received per millisecond and the horizontal axis is the time in seconds. To calculate the true message arrival rate (colored in blue in Figure 11), we accumulated the number of messages sent in the whole system every 100 millisecond. We normalized the message arrival rate by dividing the sum by 100 and fragment number 64. Following the method of Section 6.3, we set time window τ as 200 milliseconds, and used the mean value of the true arrival rates recorded in the time window to estimate the arrival rate in the next round.

The results tell us the following. (a) The estimated arrival rate is quite close to the true arrival rate. (b) Despite of spikes, the true arrival rate in Figure 11 is overall continuous without abrupt breaks, which guarantees the effectiveness of our localized mean based estimation.

Exp-7: Case studies. Finally, we conducted two case studies to understand how AAP adaptively adjusts delay stretches and reduces response time, with PageRank and CF.

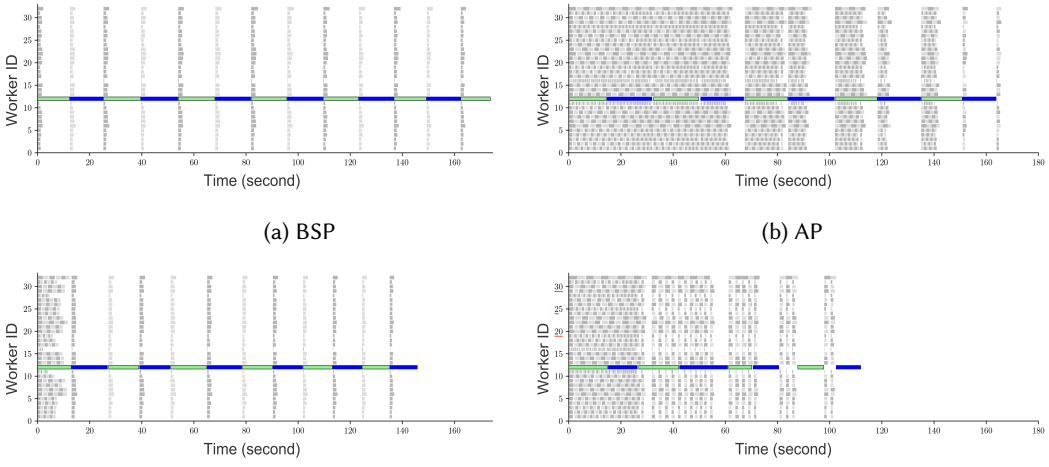
(1) PageRank. Figure 12 shows the timing diagrams of GRAPE+$_{\text{BSP}}$, GRAPE+$_{\text{AP}}$, GRAPE+$_{\text{SSP}}$ and GRAPE+ for PageRank over real-life graph Friendster. Among 32 workers used in the tests, P_{12} is a straggler (colored in blue and green). We find that stragglers often arise in the presence of streaming updates, even when we start with an evenly partitioned graph.

(a) BSP. As shown in Figure 12a, straggler P_{12} dominated the running time. Each superstep of the BSP run was slowed down by the straggler due to the global synchronization barriers. The other 31 workers mostly idled, and the run took 13 rounds and 174s.

(b) AP. GRAPE+$_{\text{AP}}$ did slightly better and took 166s, as shown in Figure 12b. Idling was substantially reduced. However, fast workers performed far more rounds of computation under AP than under BSP, and most of these rounds are redundant and useless. The cost was still dominated by straggler P_{12}. Indeed, after a period of time, a fast worker behaved as follows: it moved ahead, became inactive (idle), got activated by messages produced by P_{12}, and so on, until P_{12} converged.

(c) SSP. Figure 12c depicts a run of GRAPE+$_{\text{SSP}}$ with staleness bound $c = 5$, *i.e.,* fast workers are allowed to outpace the slowest ones by 5 rounds. It did better at the beginning. However, when the fast workers ran out of c steps, there still existed a gap from straggler P_{12}. Then SSP degraded to BSP and fast workers were essentially synchronized with the straggler. The run took 145s.

(d) AAP. Figure 12d shows a run of GRAPE+. It dynamically adjusted delay stretch DS_i for each worker P_i by function δ (Section 3). We set predicate S = true since PageRank does not need bounded staleness (Section 5.3), and we used initial $\text{L}_\perp = 0$ to start with.



(a) BSP (b) AP

(c) SSP (c=5) (d) AAP

Fig. 12. Case study of PageRank

AAP adjusted delay stretch DS_{12} at straggler P_{12} as follows. (i) Until round 6, function δ kept $DS_{12} = \eta_{12}$ since messages arrived in a near uniform speed before round 6; there was no need to wait for extra messages. (ii) At round 6, DS_{12} was increased by 31 based on predicted running time t_{12} of IncEval at worker P_{12} and message arrival rate s_{12}, by using the methods of Section 6. As a result, worker P_{12} was put on hold for 8s to accumulate messages before entering round 7. This effectively reduced redundant computations. Indeed, P_{12} converged in 8 rounds, and the run of GRAPE+ took 112s.

Observe the following. (i) Starting from round 3 of P_{12}, fast workers were actually grouped together and ran BSP within the group, by adjusting their relative progress; meanwhile this group and straggler P_{12} were under AP. As opposed to the BSP degradation of SSP, this BSP group *does not* include straggler P_{12}. Workers in the group had similar workload and speed; there was no straggler among them. These workers effectively utilized resources and performed almost optimally. (ii) Straggler P_{12} was put on hold from round 7 to accumulate messages; this effectively reduced redundant computations and eventually led to less rounds for P_{12} to converge. (iii) The estimations of t_i and s_i were quite accurate with the methods of Section 6. (iv) If users opt to set $\mathsf{L}_\perp$ as, *e.g.*, 31, function δ can start reducing redundant computations early and straggler P_{12} can find "optimal" stretch DS_i sooner. It is because of this that we allow users to set $\mathsf{L}_\perp$ in function δ.

(2) CF. We also analyzed the runs of CF on Netflix with 64 workers. Note that CF requires staleness bound c, as opposed to SSSP, CC and pageRank. From the experiments we find the following.

(a) BSP. On one hand, BSP converged in the least rounds (351), but on the other hand, it incurred excessive idleness and was actually slower than AAP and SSP.

(b) AP. While idleness was nearly zero, AP took the most rounds (4500) and was slower than AAP and SSP, as also noted in [84]. That is, a large part of the computations under AP is stale and useless.

(c) SSP. Tuning c was helpful. However, it is hard to find an optimal c for SSP. We had to run $\text{GRAPE+}_{\text{SSP}}$ 50 times to find the optimal c_o, through a process of trail and fail.

(d) AAP. To enforce bounded staleness, predicate S is defined as false if $r = r_{\max}$ and $|r_{\max} - r_{\min}| > c$, for c from 2 to 50 in different tests. In the first a few rounds, function δ set delay stretch L_i for each worker P_i as 60% of the number of workers, *i.e.*, P_i waited and accumulated messages from 60% of other workers before the next round. It then adjusted L_i dynamically for each P_i.

AAP performed the best among the 4 models. Better yet, AAP is robust and *insensitive to c*. Given a random c, AAP dynamically adjusted L_i and outperformed SSP even when SSP was provided with the optimal c_o that was manually found after 50 trail tests.

Summary. From the experimental study we find the following.

(1) GRAPE+ outperforms most of the state-of-the-art systems. Over real-life graphs and with 192 workers, GRAPE+ is on average (a) 2080.0, 838.0, 550.0, 728.0, 1850.0, 636.0, 115.0, 2.8, 92.0 and 5.1 times faster than Giraph, GraphLab$_{\text{sync}}$, GraphLab$_{\text{async}}$, GiraphUC, Maiter, PowerSwitch, Husky, Galois, Pregel+ and TDataflow for SSSP, (b) 835.0, 314.0, 93.0, 368.0, 34.5, 51.1, 40.7, 3.2, 160.2 and 1.1 times faster than these systems for CC, (c) 339.0, 4.8, 8.6, 346.0, 9.7, 4.6, 14.3, 1.1, 13.6 and 3.6 times faster for PageRank, and (d) 57.1, 46.1, 48.3, 10.1, 7.4 and 6.2 times faster than GraphLab$_{\text{sync}}$, GraphLab$_{\text{async}}$, PowerSwitch, Pregel+, Husky and Petuum for CF, respectively. Among these Galois has the closest performance to GRAPE+ and occasionally outperforms GRAPE+.

(2) GRAPE+ incurs 7.1%, 3.4%, 17.3%, 1.7%, 8.2%, 14.1%, 8.6%, 42.7%, 55.6% and 15.5% of the communication cost of GraphLab$_{\text{sync}}$, GraphLab$_{\text{async}}$, PowerSwitch, Petuum, Giraph, Maiter, GiraphUC, Husky, Pregel+ and TDataflow on average, respectively. These show that AAP and IncEval can substantially reduce communication cost by reducing computation rounds and redundant computation.

(3) AAP effectively reduces stragglers and redundant stale computations. It is on average 4.8, 1.7 and 1.8 times faster than BSP, AP and SSP for these problems over real-life graphs, respectively. On large-scale synthetic graphs, AAP is on average 4.3, 14.7 and 4.7 times faster, respectively, up to 5.0, 16.8 and 5.9 times with 320 workers. On large-scale real-life graph ClueWeb12, AAP is on average 2.8, 1.3 and 1.5 times faster, respectively, up to 3.8, 1.5 and 1.9 times with 192 workers.

(4) The heavier stragglers and stale computations are, or the larger the graphs are and the more workers are used, the more effective AAP is in speeding up parallel computations.

(5) GRAPE+ scales well with the number n of workers used in parallel computation. It is on average 2.4, 2.7, 2.3 and 1.7 times faster when n varies from 64 to 192 for SSSP, CC, PageRank and CF, respectively. Moreover, it has good scale-up with large-scale graphs.

(6) Our prediction methods accurately and efficiently estimate runtime, and RFR works especially well. The training time is less than the average time taken by a single IncEval round.

(7) Our simple estimation of message arrival rate is accurate and efficient.

9 CONCLUSION

We have proposed AAP to remedy the limitations of BSP and AP by reducing both stragglers and redundant stale computations. As opposed to [85], we have shown that as an asynchronous model, AAP does not make programming harder, and it retains the ease of consistency control and convergence guarantees. We have also developed the first condition to warrant the Church-Rosser property of asynchronous runs, and a simulation result to justify the power and flexibility of AAP. Our experimental results have verified that AAP is promising for large-scale graph computations.

One topic for future work is to improve runtime estimation for different computations. Another topic is to handle streaming updates by capitalizing on the capability of incremental IncEval. A third topic is to adopt techniques of, *e.g.,* [78] and [29], to further reduce the communication cost.

ACKNOWLEDGMENTS

Fan is supported in part by ERC 652976, Royal Society Wolfson Research Merit Award WRM/R1/180014, EPSRC EP/M025268/1, Shenzhen Institute of Computing Sciences, and Beijing Advanced Innovation Center for Big Data and Brain Computing. Lu is supported in part by NSFC 61602023.

REFERENCES

[1] 2006. GTgraph. *http://www.cse.psu.edu/ kxm85/software/GTgraph/.*
[2] 2006. UKWeb. *http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05/.*
[3] 2010. Traffic. *http://www.dis.uniroma1.it/challenge9/download.shtml.*
[4] 2011. Movielens. *http://grouplens.org/datasets/movielens/.*
[5] 2012. ClueWeb12. *http://www.lemurproject.org/clueweb12.php/.*
[6] 2012. Friendster. *https://snap.stanford.edu/data/com-Friendster.html.*
[7] 2012. MPICH. *https://www.mpich.org/.*
[8] 2014. Giraph. *http://giraph.apache.org/.*
[9] 2014. Timely Dataflow. *https://github.com/frankmcsherry/timely-dataflow/.*
[10] 2015. COST. *https://github.com/frankmcsherry/COST/.*
[11] 2015. PageRank in timely dataflow. *https://github.com/frankmcsherry/pagerank/.*
[12] 2017. Netflix Prize data. *https://www.kaggle.com/netflix-inc/netflix-prize-data.*
[13] Umut A. Acar. 2005. *Self-Adjusting Computation.* Ph.D. Dissertation. CMU.
[14] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling parallel programs by work stealing with private deques. In *PPoPP.*
[15] Konstantin Andreev and Harald Racke. 2006. Balanced graph partitioning. *Theory of Computing Systems* 39, 6 (2006), 929–939.
[16] Dmitrii Avdiukhin, Sergey Pupyrev, and Grigory Yaroslavtsev. 2019. Multi-Dimensional Balanced Graph Partitioning via Projected Gradient Descent. *PVLDB* 12, 8 (2019), 906–919.
[17] Jrgen Bang-Jensen and Gregory Z. Gutin. 2008. *Digraphs: Theory, Algorithms and Applications.* Springer.
[18] Nguyen Thien Bao and Toyotaro Suzumura. 2013. Towards highly scalable pregel-based graph processing platform with x10. In *WWW '13.* 501–508.
[19] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence* 35, 8 (2013), 1798–1828.
[20] Gérard Biau and Erwan Scornet. 2016. A random forest guided tour. *Test* 25, 2 (2016), 197–227.
[21] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (1999), 720–748.
[22] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.
[23] Sergey Brin and Lawrence Page. 2012. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks* 56, 18 (2012), 3825–3833.
[24] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (1985), 63–75.
[25] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 13.
[26] Yang Chen, Sean Goldberg, Daisy Zhe Wang, and Soumitra Siddharth Johri. 2016. Ontological Pathfinding. In *SIGMOD.*
[27] Carl C Cowen, KR Davidson, and RP Kaufman. 1980. Rearranging the alternating harmonic series. *The American Mathematical Monthly* 87, 10 (1980), 817–819.
[28] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric P. Xing. 2015. High-Performance Distributed ML at Scale through Parameter Server Consistency Models. In *AAAI.*
[29] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-Optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *PLDI.* 752–768.
[30] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.
[31] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008).
[32] Ramón Díaz-Uriarte and Sara Alvarez De Andres. 2006. Gene selection and classification of microarray data using random forest. *BMC bioinformatics* 7, 1 (2006), 3.
[33] James Dinan, Stephen Olivier, Gerald Sabin, Jan Prins, P. Sadayappan, and Chau-Wen Tseng. 2007. Dynamic Load Balancing of Unbalanced Computations Using Message Passing. In *IPDPS.*
[34] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental Graph Computations: Doable and Undoable. In *SIGMOD.*

[35] W. Fan, R. Jin, M. Liu, P. Lu, X. Luo, R. Xu, Q. Yin, W. Yu, and J. Zhou. 2020. Application driven graph partitioning. In *SIGMOD*.

[36] Wenfei Fan, Ping Lu, Xiaojian Luo, Jingbo Xu, Qiang Yin, Wenyuan Yu, and Ruiqi Xu. 2018. Adaptive Asynchronous Parallelization of Graph Algorithms. In *SIGMOD*. 1141–1156.

[37] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. 2017. GRAPE: Parallelizing Sequential Graph Computations. *PVLDB* 10, 12 (2017), 1889–1892.

[38] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Bohan Zhang, Zeyu Zheng, Yang Cao, and Chao Tian. 2017. Parallelizing Sequential Graph Computations. In *SIGMOD*.

[39] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, XiaoJian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing Sequential Graph Computations. *TODS* 43, 18 (2018).

[40] Michael L Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM* 34, 3 (1987), 596–615.

[41] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. 2001. *The Elements of Statistical Learning*. Vol. 1. Springer series in statistics New York, NY, USA:.

[42] Muriel Gevrey, Ioannis Dimopoulos, and Sovan Lek. 2003. Review and comparison of methods to study the contribution of variables in artificial neural network models. *Ecological modelling* 160, 3 (2003), 249–264.

[43] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *USENIX*.

[44] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*.

[45] Minyang Han and Khuzaima Daudjee. 2015. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *PVLDB* 8, 9 (2015), 950–961.

[46] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M Tamer Ozsu, Xingfang Wang, and Tianqi Jin. 2014. An experimental comparison of Pregel-like graph processing systems. *PVLDB* 7, 12 (2014).

[47] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NIPS*. 1223–1231.

[48] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[49] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. 2014. Algorithm runtime prediction: Methods & evaluation. *Artif. Intell.* (2014), 79–111.

[50] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. 2010. A Model of Computation for MapReduce. In *SODA*. 938–948.

[51] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: A system for dynamic load balancing in large-scale graph processing. In *EuroSys '13*. 169–182.

[52] Mijung Kim and K Selçuk Candan. 2012. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data & Knowledge Engineering* 72 (2012), 285–303.

[53] Konrad Knopp. 2013. *Theory and application of infinite series*. Courier Corporation.

[54] Seongyun Ko and Wook-Shin Han. 2018. TurboGraph++: A Scalable and Fast Graph Analytics System. In *SIGMOD*. Houston, TX, USA, 395–410.

[55] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *IEEE Computer* 42, 8 (2009), 30–37.

[56] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *USENIX*.

[57] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *PVLDB* 5, 8 (2012).

[58] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *SIGMOD*.

[59] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *ACM Comput. Surv.* 48, 2 (2015), 25:1–25:39.

[60] Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! But at what COST?. In *HotOS*.

[61] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *INTERSPEECH*.

[62] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *SOSP*. ACM, 439–455.

[63] Nasser M Nasrabadi. 2007. Pattern recognition and machine learning. *Journal of electronic imaging* 16, 4 (2007), 049901.

[64] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*. 456–471.
[65] Heungsun Park and LA Stefanski. 1998. Relative-error prediction. *Statistics & probability letters* 40, 3 (1998), 227–236.
[66] Paul N Pruitt. 1998. *An asynchronous checkpoint and rollback facility for distributed computations*. Ph.D. Dissertation. College of William and Mary.
[67] G. Ramalingam and Thomas Reps. 1996. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms* 21, 2 (1996), 267–305.
[68] G. Ramalingam and Thomas Reps. 1996. On the computational complexity of dynamic graph problems. *TCS* 158, 1-2 (1996).
[69] Semih Salihoglu and Jennifer Widom. 2013. GPS: A graph processing system. In *SSDBM*.
[70] Zechao Shang and Jeffrey Xu Yu. 2013. Catch the wind: Graph workload balancing on cloud. In *ICDE*. 553–564.
[71] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In *SIGMOD*.
[72] George M Slota, Sivasankaran Rajamanickam, Karen Devine, and Kamesh Madduri. 2017. Partitioning Trillion-edge Graphs in Minutes. In *IPDPS*.
[73] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *KDD*. 1222–1230.
[74] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, and John McPherson Shirish Tatikonda. 2013. From "Think Like a Vertex" to "Think Like a Graph". *PVLDB* 7, 7 (2013), 193–204.
[75] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111.
[76] Leslie G. Valiant. 1990. General Purpose Parallel Architectures. In *Handbook of Theoretical Computer Science, Vol A*.
[77] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2016. Synergistic analysis of evolving graphs. *TACO* 13, 4 (2016), 32.
[78] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. 2014. ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 861–878.
[79] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. 2017. Coral: Confined recovery in distributed asynchronous graph processing. *ACM SIGOPS Operating Systems Review* 51, 2 (2017), 223–236.
[80] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. 2013. Asynchronous Large-Scale Graph Processing Made Easy. In *CIDR*.
[81] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2015. Managed communication and consistency for fast data-parallel iterative analytics. In *SOCC*. 381–394.
[82] Marvin N Wright and Andreas Ziegler. 2015. Ranger: A fast implementation of random forests for high dimensional data in C++ and R. *arXiv preprint arXiv:1508.04409* (2015).
[83] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. SYNC or ASYNC: Time to fuse for distributed graph-parallel computation. In *PPOPP*.
[84] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. 2015. Petuum: A New Platform for Distributed Machine Learning on Big Data. *IEEE Transactions on Big Data* 1, 2 (June 2015), 49–67.
[85] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. 2017. Big Graph Analytics Platforms. *Foundations and Trends in Databases* 7, 1-2 (2017), 1–195.
[86] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *PVLDB* 7, 14 (2014), 1981–1992.
[87] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*. 1307–1317.
[88] Fan Yang, Jinfeng Li, and James Cheng. 2016. Husky: Towards a more efficient and expressive distributed computing framework. *PVLDB* 9, 5 (2016), 420–431.
[89] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments. In *OSDI*.
[90] Timothy A. K. Zakian, Ludovic Capelli, and Zhenjiang Hu. 2018. Automatic Incrementalization of Vertex-Centric Programs. *http://www.cs.ox.ac.uk/people/timothy.zakian/icpp_draft.pdf*.
[91] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2014. Maiter: An Asynchronous Graph Processing Framework for Delta-Based Accumulative Iterative Computation. *TPDS* 25, 8 (2014), 2091–2100.
[92] Zhensheng Zhang and Christos Douligeris. 1991. Convergence of Synchronous and Asynchronous Algorithms in Multiclass Networks. In *INFOCOM*. IEEE, 939–943.
[93] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *OSDI*. 301–316.