Adaptive Asynchronous Parallelization of Graph Algorithms

Wenfei Fan[1,2,3], Ping Lu[2], Xiaojian Luo[3], Jingbo Xu[2,3], Qiang Yin[2], Wenyuan Yu[2,3], Ruiqi Xu[1]
[1]University of Edinburgh [2]BDBC, Beihang University [3]7 Bridges Ltd.
{wenfei@inf., ruiqi.xu@}ed.ac.uk, {luping, yinqiang}@buaa.edu.cn, {xiaojian.luo, jingbo.xu, wenyuan.yu}@7bridges.io

ABSTRACT

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 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.

KEYWORDS

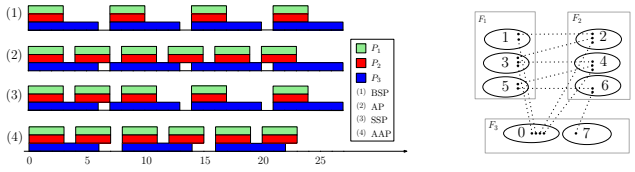
parallel model; parallelization; graph computations; Church-Rosser

ACM Reference Format:
Wenfei Fan, Ping Lu, Xiaojian Luo, Jingbo Xu, Qiang Yin, Wenyuan Yu, and Ruiqi Xu. 2018. Adaptive Asynchronous Parallelization of Graph Algorithms. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3196918

1 INTRODUCTION

Bulk Synchronous Parallel (BSP) model [48] has been adopted by graph systems, *e.g.,* Pregel [39] and GRAPE [24]. Under BSP, iterative computation is separated into supersteps, and messages from one superstep are only accessible in the next one. This simplifies the analysis of parallel algorithms. However, its global synchronization barriers lead to stragglers, *i.e.,* some workers take substantially longer than the others. As workers converge asymmetrically, the speed of each superstep is limited to that of the slowest worker.

To reduce stragglers, Asynchronous Parallel (AP) model has been employed by, *e.g.,* GraphLab [26, 38] and Maiter [57]. Under AP, a worker has immediate access to 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 costs. It is also observed that AP makes it hard to write, debug and analyze programs [50], and complicates the consistency analysis (see [54] for a survey).



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

Figure 1: Runs under different parallel models

A recent study shows that neither AP nor BSP consistently outperforms the other for different algorithms, input graphs and cluster scales [52]. 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) [30] and a hybrid model Hsync [52]. SSP relaxes BSP by allowing fastest workers to outpace the slowest ones by a fixed number c of steps (bounded staleness). It reduces stragglers, but incurs redundant stale computations. Hsync suggests to switch between AP and BSP, but it requires 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 BSP programming, ensure consistency, and guarantee correct convergence under a general condition?

AAP. To answer the questions, we propose 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: 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, P_3 takes 6 units, and it takes 1 unit to pass messages. This is carried out under different models as follows, as shown in Fig. 1(a) (it depicts runs for computing connected components shown in Fig. 1(b), to be elaborated in Example 4).

(1) BSP. As depicted in Fig. 1(a) (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.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196918

(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. 1(a) (2), at clock time 7, the second round of P_3 can only use the messages from the first round of P_1 and P_2. This round of 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. 1(a) (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 decides when to start the next round based on the progress of others. As shown in Fig. 1(a) (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, since P_3 can converge in less rounds by utilizing the latest updates from fast workers. □

AAP reduces stragglers by not blocking fast workers. This is particularly 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 the partition cost. Moreover, AAP activates a worker only after it receives sufficient up-to-date messages and thus reduces redundant 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: some worker groups may follow BSP, while at the same time, the others run AP or SSP.

(2) Programming paradigm. AAP works with the programming model of GRAPE [24]. 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 [54].

(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 [20], PRAM (Parallel Random Access Machine) [49], BSP, AP and SSP. That is, algorithms developed for these models can be migrated to AAP without increasing the complexity.

System	PageRank		SSSP	
	Time(s)	Comm(GB)	Time(s)	Comm(GB)
Giraph	6117.7	767.3	416.0	99.4
$\text{GraphLab}_{\text{sync}}$	99.5	138.0	37.6	110.0
$\text{GraphLab}_{\text{async}}$	200.1	333.0	194.1	368.7
GiraphUC	9991.6	3616.5	278.9	121.9
Maiter	199.9	134.3	258.9	107.2
PowerSwitch	85.1	39.9	32.46	41.5
GRAPE+	26.4	37.3	12.7	18.3

Table 1: PageRank **and** SSSP **on parallel systems**

(5) Performance. AAP outperforms BSP, AP and SSP for a variety of graph computations. As an example, for PageRank [15] and SSSP (single-source shortest path) on Friendster [1] with 192 workers, Table 1 shows the performance of (a) Giraph [2] (an open-source version of Pregel) and GraphLab [38] under BSP, (b) GraphLab and Maiter [57] under AP, (c) GiraphUC [28] under BAP, (d) PowerSwtich [52] under Hsync, and (e) GRAPE+, an extension of GRAPE by supporting AAP. GRAPE+ does better than these systems.

Contributions and organization. This paper introduces AAP, from foundations to implementation and empirical evaluation.

(1) Programming model. We present the programming model of GRAPE (Section 2). We show that it works well with AAP.

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

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

(5) Implementation. As proof of concept, we develop GRAPE+ by extending GRAPE [23] from BSP to AAP (Section 6).

(6) Experiments. Using real-life and synthetic graphs, we evaluate the performance of GRAPE+ (Section 7), compared with the state-of-the-art graph systems listed in Table 1, and Petuum [53], a parameter server under SSP. Over real-life graphs and with 192 workers, (a) GRAPE+ is at least 2.6, 4.8, 3.2 and 7.9 times faster than these systems for SSSP, CC, PageRank and CF, respectively, up to 4127, 1635, 446 and 51 times. It incurs communications costs as small as 0.0001%, 0.027%, 0.13% and 57.7%, respectively. (b) On average 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 on average 4.3, 14.7 and 4.7 times faster, respectively. (c) GRAPE+ is on average 2.37, 2.68, 2.17 and 2.3 times faster for SSSP, CC, PageRank and CF, respectively, when the number of workers varies from 64 to 192.

Related work. Several parallel models have been studied for graphs. PRAM [49] supports parallel RAM access with shared memory, not for the shared-nothing architecture that is used nowadays.

MapReduce [20] is adopted by, *e.g.,* GraphX [27]. However, it is not very efficient for iterative graph computations due to its blocking and I/O costs. BSP [48] with vertex-centric programming works better for graphs as shown by [39]. However, it suffers from stragglers. As remarked earlier, AP reduces stragglers, but it comes with redundant stale computation. It also bears with race conditions and their locking/unblocking costs, and complicates the convergence analysis (see Section 4.1) and programming [50].

SSP [30] promotes bounded staleness for machine learning. Maiter [57] reduces stragglers by accumulating updates, and supports prioritized asynchronous execution. BAP model (*barrierless asynchronous parallel*) [28] reduces global barriers and local messages by using light-weighted local barriers. As remarked earlier, Hsync proposes to switch between AP and BSP [52].

Several graph systems under these models are in place, *e.g.,* Pregel [39], GPS [44], Giraph++ [47], GRAPE [24] under BSP; GraphLab [26, 38], Maiter [57], GRACE [50] under (revised) AP; parameter servers under SSP [30, 37, 45, 51, 53]; GiraphUC [28] under BAP; and PowerSwitch under Hsync [52]. Blogel [55] works like AP within blocks, and in BSP across blocks. Most of these are vertex-centric. While Giraph++ and Blogel [47] process blocks [47], they inherit vertex-centric programming by treating blocks as vertices. 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. It is based on fixpoint computation, which simplifies the convergence and consistency analyses of AP.

(2) SSP mainly targets machine learning, with different correctness criteria. When accelerating graph computations is concerned, in contrast to SSP, (a) AAP reduces redundant stale computations by enforcing a "lower bound" on accumulated messages, which also serves as an "upper bound" to support bounded staleness if needed. As will be seen in Section 3, performance can be improved when stragglers are forced to wait, rather than to catch up as suggested by SSP. (b) AAP dynamically adjusts the bound, instead of using a predefined constant. (c) Bounded staleness is not needed by SSSP, CC, and PageRank as will be seen in Section 5.3.

(3) 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 the bound to optimize performance, and (c) combining incremental evaluation with accumulative computation. AAP operates on graph fragments, while Maiter is vertex-centric.

(4) 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.

(5) Closer to AAP is Hsync, and PowerSwitch has the closest performance 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 Section 3, fast workers may follow BSP within a group, while meanwhile, the other workers may adopt AP. Moreover, the parameters are adjusted dynamically, and hence AAP does not have to predict switching points and pay the price of switching cost.

Close to this work is GRAPE [24]. AAP adopts the programming model of GRAPE, and GRAPE+ extends GRAPE. However, (1) the objective of this work is to introduce AAP and to explore appropriate models for graph computation. In contrast, GRAPE adopts BSP. (2) We show that as an asynchronous model, AAP retains the programming paradigm and ease of consistency control of GRAPE. (3) We identify a condition for AAP to converge at correct results and have the Church-Rosser property, which is not an issue for GRAPE. (4) We prove stronger simulation results (see Section 4.2). Moreover, AAP can optimally simulate BSP, AP, SSP and GRAPE (Section 4.2). (5) We evaluate GRAPE+ and GRAPE by comparing with graph systems of different models, while the experimental study of [24] focused on synchronous systems only.

There has also been work on mitigating the straggler problem, *e.g.,* dynamic repartitioning [13, 33, 40], work stealing [10, 14], shedding [21], LATE [56], and fine-grained partition [17]. AAP is complementary to these methods, to reduce stragglers and stale computation by adjusting relative progress of workers.

2 THE PROGRAMMING MODEL

AAP adopts the programming model of [24], which we review next. As will be seen in Section 3, AAP is able to parallelize sequential graph algorithms just like GRAPE. That is, the asynchronous model does not make programming harder than GRAPE.

Graph partition. AAP supports data-partitioned parallelism. It is to work on graphs partitioned into smaller fragments.

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 a edge-cut [11] or vertex-cut [34] 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 [24], to answer queries $Q \in \mathcal{Q}$ under AAP, one only needs to specify three functions.

(1) PEval: a sequential algorithm for $\mathcal{Q}$ that given a query $Q \in \mathcal{Q}$ and a graph G, computes the answer $Q(G)$.

Input: A fragment $F_i(V_i, E_i, L_i)$.
Output: A set $Q(F_i)$ consists of current $v.\text{cid}$ for $v \in V_i$.

Message preamble: /*candidate set C_i is $F_i.O$*/
For each node $v \in V_i$, a variable $v.\text{cid}$;

1. $\mathbb{C} := \text{DFS}(F_i)$; /* find local connective components by DFS */
2. **for each** $C \in \mathbb{C}$ **do**
3. create a new "root" node v_c;
4. $v_c.\text{cid} := \min\{v.\text{id} \mid v \in C\}$;
5. **for each** $v \in C$ **do**
6. link v to v_c; $v.\text{root} := v_c$; $v.\text{cid} := v_c.\text{cid}$;
7. $Q(F_i) := \{v.\text{cid} \mid v \in V_i\}$;

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

Figure 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** $v^{\text{in}}.\text{cid} \in M_i$ **do** /* use min as f_{aggr} */
3. $v.\text{cid} := \min\{v.\text{cid}, v^{\text{in}}.\text{cid}\}$;
4. $v_c := v.\text{root}$;
5. **if** $v.\text{cid} < v_c.\text{cid}$ **then**
6. $v_c.\text{cid} := v.\text{cid}$; $\Delta := \Delta \cup \{v_c\}$;
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. $v.\text{cid} := v_c.\text{cid}$;
10. $Q(F_i) := \{v.\text{cid} \mid v \in V_i\}$;

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

Figure 3: IncEval **for** CC **under** AAP

(2) IncEval: a sequential incremental algorithm for 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 [43].
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 complete answer $Q(G)$.

Taken together, the three functions are referred to as *a* PIE *program for* Q (PEval, IncEval and Assemble). PEval and IncEval can be *existing sequential* (incremental) algorithms for 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. 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 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: 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.

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 [12].

AAP parallelizes CC with the same PEval and IncEval of GRAPE [24]. More specifically, a PIE program ρ 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): (a) for each node $v \in V_i$, an integer variable $v.\text{cid}$, initially $v.\text{id}$; (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; and (c) min as aggregate function f_{aggr}: if there are multiple values for the same $v.\text{cid}$, 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 $v_c.\text{cid}$, and (b) links all the nodes in C to v_c, and sets their cid as $v_c.\text{cid}$. 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 F_i, by "merging" components when possible. As shown in Fig. 3, by using min as f_{aggr}, it (a) updates the cid of each border node to the minimum one; and (b) propagates the change to its root v_c and 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 have been methods for incrementalizing graph algorithms, to get incremental algorithms from their batch counterparts [9]. Moreover, it is not hard to develop IncEval by revising a batch algorithm in response to changes to update parameters, as shown by the cases of CC (Example 4) 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 strategy 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 7.

(3) The programming model aims to facilitate users to develop parallel programs, especially for those who are more familiar with conventional sequential programming. This said, programming with GRAPE still requires domain knowledge of algorithm design, to declare update parameters and design an 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 Q of queries on a graph G, AAP takes as input a PIE program ρ (*i.e.,* PEval, IncEval, Assemble) for 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 for $i \in$

$[1, m]$. It works with a master P_0 and n shared-nothing physical workers $(P_1, \ldots, P_n)$, where $n < m$, *i.e.,* multiple virtual workers are 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 additionally declares update parameters $C_i.\bar{x}$, and defines an aggregate function f_{aggr}. At each worker P_i, (a) PEval computes $Q(F_i)$ over local 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 [24].

More specifically, 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 strategy $\mathcal{P}$; and

(2) a buffer $\mathbb{B}_{\bar{x}_i}$, to keep track of messages from other workers.

As opposed to GRAPE, 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 updates. Stretch DS_i is dynamically adjusted by a function δ based on the following.

(1) Staleness η_i, measured by the number of messages in buffer $\mathbb{B}_{\bar{x}_i}$ received by 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 keeps track of its current round r_i. These are to control the relative speed of workers.

For example, to simulate SSP [30], when $r_i = r_{\text{max}}$ and $r_i - r_{\text{min}} > c$, we can set $\text{DS}_i = +\infty$, to prevent P_i from moving too far ahead.

We will present 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 $j \in [1, m], j \neq i$.

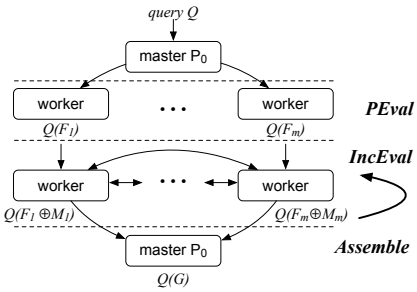


Figure 4: Workflow of AAP

More specifically, $M_{(i,j)}$ consists of triples (x, val, r), where $x \in C_i.\bar{x}$ is associated with a node v that is in $C_i \cap C_j$, and C_j is deduced from the index I_i; val is the value of x, and r indicates the round 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) work 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 P_i, it does the following:

- compute $M_i = f_{\text{aggr}}(\mathbb{B}_{\bar{x}_i} \cup C_i.\bar{x})$, *i.e.,* IncEval applies the aggregate function 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 its local fragment F_i (*i.e.,* $C_i.\bar{x}$); and
- 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 $j \in [1, m], j \neq i$; it sends $M_{(i,j)}$ 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 is imposed.

When IncEval completes its current round at P_i or when P_i receives a new message, 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, 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 $\mathbb{B}_{\bar{x}_i} = \emptyset$, 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 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 (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 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.

Example 3: Recall the PIE program ρ for CC from Example 2.

Under AAP, it works in three phases as follows.

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

(2) IncEval first computes updates M_i by applying min to 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, just like PEval does.

The process iterates until no more changes can be made.

(3) Assemble is invoked at master at this point. It computes and returns connected components as described in Example 2. □

The example shows that AAP works well with the programming model of GRAPE, *i.e.,* AAP does not make programming harder.

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.,* 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 triggers 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.

Moreover, AAP can simulate Hsync [52] by using function δ to implement the same switching rules of Hsync.

Dynamic adjustment. AAP is able to dynamically adjust delay sketch DS_i at each worker P_i; for example, function δ may define

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

where the parameters of function δ are described as follows.

(1) Predicate $S(r_i, r_{\min}, r_{\max})$ is to decide whether 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, to strike a balance between stale-computation reduction and useful outcome expected from the next round of IncEval at P_i. AAP adjusts L_i as follows. Users may opt to initialize L_i with a uniform bound $\mathsf{L}_\perp$, to start stale-computation reduction early (see Appendix B for an example). AAP adjusts L_i at each round at P_i, based on (a) predicted running time t_i of the next round, and (b) the 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. We approximate t_i and s_i by aggregating statistics of consecutive rounds of IncEval. One can get more precise estimate by using a random forest model [31], with query logs as training samples.

(3) Variable $T^i_{\mathsf{L}_i}$ estimates how longer P_i should wait to accumulate L_i many messages. We approximate it as $\frac{\mathsf{L}_i - \eta_i}{s_i}$, using 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. We use T^i_{idle} to prevent P_i from indefinite waiting.

Example 4: As an instantiation of Example 1, recall the PIE program ρ for CC from Example 2 and illustrated in Example 3. 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. 1(b), (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 1(a).

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

(b) Under AP, a run is shown in Fig. 1(a) (2). Note that before getting cid 0, workers P_1 and P_2 invoke 3 rounds of IncEval and exchange cid 1 among 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. 1(a) (3). It is almost the same as Fig. 1(a) (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, P_3 can suspend IncEval until it receives enough changes as shown in Fig. 1(a) (4). For instance, function δ starts with $\mathsf{L}_\perp = 0$. It sets $\mathsf{DS}_i = 0$ if $|\eta_i| \ge 1$ for $i \in [1, 2]$ since no messages are predicted to arrive within the next time unit. In contrast, it sets $\mathsf{DS}_3 = 1$ if $|\eta_3| \le 4$ 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 1(a) (1)–(3), the straggler reaches fixpoint in less rounds. □

From our experimental study, we find that AAP reduces the costs of iterative graph computations mainly from three directions.

(1) AAP reduces redundant stale computations and stragglers by adjusting relative progress of workers. In particular, (a) some computations are substantially improved when stragglers are forced to accumulate messages; this actually enables the stragglers to converge in less rounds, as shown by Example 4 for CC. (b) When the time taken by different rounds at a worker does not vary much (*e.g.,* PageRank in Appendix B), fast workers are "automatically" grouped together after a few rounds and run essentially BSP within the group, while the group and slow workers run under AP. This shows that AAP is more flexible than Hsync [52].

(2) Like GRAPE, AAP employs incremental IncEval to minimize unnecessary recomputations. The speedup is particularly evident when IncEval is bounded [43], localizable or relatively bounded [22]. For instance, IncEval is *bounded* [42] 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 $|M_i| + |\Delta O_i|$. As an 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 [54], asynchronous executions complicate the convergence analysis. Nonetheless, we develop a condition under which AAP guarantees to converge at correct answers. In addition, AAP is generic. We show that parallel models MapReduce, PRAM, BSP, AP and SSP can be optimally simulated by AAP. Proofs of the results in this section can be found in Appendix.

4.1 Convergence and Correctness

Given a PIE program ρ (*i.e.,* PEval, IncEval, 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 its result $\rho(Q, G) = Q(G)$.

We formalize termination and correctness as follows.

Fixpoint. Similar to GRAPE [24], 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:

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

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

where $i \in [1, m]$, R_i^r denotes partial results in round r at worker P_i, fragment $F_i^0 = F_i$, $F_i^r[\bar{x}_i]$ 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 $f_{\text{aggr}}(\mathbb{B}_{x_i} \cup C_i.\bar{x})$ as we have seen in Section 3.

The computation reaches a fixpoint if for all $i \in [1, m]$, $R_i^{r_i+1} = R_i^{r_i}$, *i.e.,* no more changes to partial results $R_i^{r_i}$ at any worker. At this point, Assemble is applied to $R_i^{r_i}$ for $i \in [1, m]$, and computes $\rho(Q, G)$. If so, we say that 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 [58]. A run of ρ can be represented as traces of PEval and IncEval at all workers (see, *e.g.,* Fig. 1(a)).

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 notation.

(1) We assume a partial order $\preceq$ on partial results R_i^l. This contrasts with GRAPE [24], which defines partial order only on update parameters. The partial order $\preceq$ is needed to analyze the Church-Rosser property of asynchronous runs under AAP. To simplify the discussion, assume that R_i^l carries its update parameters $C_i.\bar{x}$.

We define the following properties of IncEval.

- IncEval is *contracting* if for all queries $Q \in \mathcal{Q}$ and fragmented graphs G via $\mathcal{P}$, $R_i^{l+1} \preceq R_i^l$ for all $i \in [1, m]$ in the same run.
- IncEval is *monotonic* if for all queries $Q \in \mathcal{Q}$ and graphs G, for all $i \in [1, m]$, if $\bar{R}_i^s \preceq R_i^t$ then $\bar{R}_i^{s+1} \preceq R_i^{t+1}$, where $\bar{R}_i^s$ and R_i^t denote partial results in (possibly different) runs.

For instance, consider the PIE program ρ for CC (Example 2). The order $\preceq$ is defined on sets of connected components (*CC*s) in each fragment, such that $S_1 \preceq S_2$ if for each *CC* C_2 in S_2, there exists a *CC* C_1 in S_1 with $C_2 \subseteq C_1$ and $\text{cid}_1 \leq \text{cid}_2$, where cid_i is the id of C_i for $i \in [1, 2]$. Then one can verify that the IncEval of ρ is both contracting and monotonic, since f_{aggr} is defined as min.

(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 ones for GRAPE [24], condition T3 does not find a counterpart in [24].

The termination condition of GRAPE remains intact under AAP.

Theorem 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.

Proof: By T1 and T2, each update parameter can be changed finitely many times. This warrants the termination of ρ since ρ terminates when no more changes can be incurred to its update parameters. □

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

Theorem 2: *Under conditions T1, T2 and T3,* AAP *correctly parallelizes a* PIE *program ρ for a query class $\mathcal{Q}$ if ρ is correct for $\mathcal{Q}$, with any partition strategy $\mathcal{P}$.* □

Proof: We show the following under the conditions. (1) Both the synchronous run of ρ under BSP and asynchronous runs of ρ under AAP reach a fixpoint. (2) No partial results of ρ under BSP are "larger" than any fixpoint of asynchronous runs. (3) No partial results of asynchronous runs are "larger" than the fixpoint under BSP. From (2) and (3) it follows that ρ has the Church-Rosser property. Hence AAP correctly parallelizes ρ as long as ρ is correct for Q. □

Recall that AP, BSP and SSP are special cases of AAP. From the proof of Theorem 2 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 see this, we examine convergence conditions for GRAPE [24], Maiter [57], BAP [28] and SSP [19, 30].

(1) As remarked earlier, the condition of GRAPE does not ensure the Church-Rosser property, which is not an issue for BSP.

(2) Maiter [57] focuses on 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 [57], 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 [57] is quite restrictive. Further, the proof of [57] does not suffice for the Church-Rosser property. A counterexample could be conditional convergent series, for which asynchronous runs may diverge [18, 35].

(3) It is shown that BAP can simulate BSP under certain conditions on message buffers [28]. 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 [19, 30]. In contrast, our conditions are deterministic: under T1, T2 and T3, all AAP runs guarantee to converge at correct answers. Moreover, we consider AAP computations not limited to machine learning.

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 this paper focuses on graph computations, AAP is not limited to graphs as a parallel computation model. It is as generic as BSP and AP, and does not have to take graphs as input.

Following [49], 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.

As remarked in Section 3, BSP, AP and SSP are special cases of AAP. From this one can easily verify the following.

Proposition 3: AAP *can optimally simulate* BSP, AP *and* SSP. □

By Proposition 3, algorithms developed for, *e.g.,* Pregel [39], GraphLab [26, 38] and GRAPE [24] 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).

We next show that AAP can optimally simulate MapReduce and PRAM. It was shown in [24] 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, referred to as designated messages in [24].

Theorem 4: *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 [32], and AAP can simulate GRAPE, it suffices to show that GRAPE can optimally simulate MapReduce with the message scheme of Section 2.

A MapReduce algorithm $\mathcal{A}$ can be specified as a sequence $(B_1, \ldots, B_k)$ of subroutines, where B_r ($r \in [1, k]$) consists of a mapper μ_r and a reducer ρ_r [20, 32]. To simulate $\mathcal{A}$ by GRAPE, we give a PIE program ρ in which (1) PEval is the mapper μ_1 of B_1, and (2) IncEval simulates reducer ρ_i and mapper μ_{i+1} ($i \in [1, k-1]$), and reducer ρ_k in the final round. We define IncEval that treats the subroutines $B_1, \ldots, B_k$ of $\mathcal{A}$ as program branches. Assume that $\mathcal{A}$ uses n processors. We add a clique G_W of n nodes as input, one for each worker, such that any two workers can exchange data stored in the status variables of their border nodes in G_W. We show that ρ incurs no more cost than $\mathcal{A}$ in each step, using n processors. □

5 PROGRAMMING WITH AAP

We have seen how AAP parallelizes CC (Examples 2–4). We next examine two PIE algorithms given in [24], including SSSP and CF. We also give a PIE program for PageRank. As opposed to [24], we parallelize these algorithms in Sections 5.1–5.3 under AAP. These show that AAP does not make programming harder.

5.1 Graph Traversal

We start with the *single source shortest path* problem (SSSP). 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 $\text{dist}(s, v)$ 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 $\text{dist}(s, v)$ for all nodes v in G.

AAP parallelizes SSSP in the same way as GRAPE [24].

(1) PIE. AAP takes Dijkstra's algorithm [25] for SSSP as PEval and the sequential incremental algorithm developed in [42] as IncEval. It declares a status variable x_v for every node v, denoting $\text{dist}(s, v)$, initially ∞ (except $\text{dist}(s, s) = 0$). The candidate set C_i at each F_i is $F_i.O$. The status variables in the candidates set are updated by PEval and IncEval as in [24], and 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 2. To see this, define order $\preceq$ on sets S_1 and S_2 of nodes in the same fragment F_i such that $S_1 \preceq S_2$ if for each node $v \in F_i$, v_1.dist $\leq v_2$.dist, where v_1 and v_2 denote the copies of v in S_1 and S_2, respectively. Then by the use of min as aggregate f_{aggr}, IncEval is both contracting and monotonic.

5.2 Collaborative Filtering

We next consider collaborative filtering (CF) [36]. 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 minimizes 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) [36], a popular algorithm for CF. We give the following PIE program.

(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.* $|P| \ll |U|$, it takes $F_i.O \cup F_i.I$, *i.e.,* the shared product nodes related to F_i, as 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 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; it sends the updated status variables as in PEval as long as the bounded staleness condition is not violated.

Assemble simply takes the union of the factor vectors of all nodes from all the workers, and returns the collection.

(2) Correctness has been verified under the bounded staleness condition [30, 53]. Along the same lines, we show that the PIE program converges and correctly infers missing CF factors.

5.3 PageRank

Finally, we study PageRank [15] 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 [15] 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.

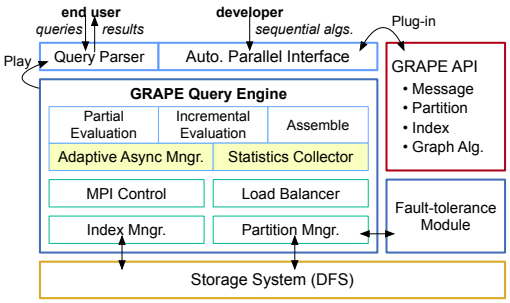


Figure 5: GRAPE+ **Architecture**

- Output: The PageRank scores of nodes in G.

AAP parallelizes PageRank along the same lines as [47, 57].

(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 $F_i.O$ 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 the variable x_u for each u linked from v by an incremental change $d * x_v/N_v$. At the end of its process, it sends the values of $C_i.\bar{x}$ to its neighboring workers.

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

Assemble collects the scores of all the nodes in G 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 2. The proof makes use of the following property, as also observed by [57]: 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 [30, 53]. By Theorem 2, 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 IMPLEMENTATION OF GRAPE+

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

The architecture of GRAPE+ is shown in Fig. 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 the evaluation plans. Underlying the engine are several components, including (1) an MPI controller [5] to handle message

passing, (2) a load balancer to evenly distribute workload, (3) an index manager to maintain indices, and (4) a partition manager for graph partitioning. 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. 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 [16] 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 we deployed GRAPE+ in a POC scenario that provides continuous online payment services, we found that on average, 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 when new messages arrive. 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.

7 EXPERIMENTAL STUDY

Using real-life and synthetic graphs, we conducted four sets of experiments to evaluate the (1) efficiency, (2) communication cost, and (3) scale-up of GRAPE+, and (4) the effectiveness of AAP and the impact of graph partitioning strategies on its performance. We also report a case study in Appendix B to illustrate how dynamic adjustment of AAP works. We compared the performance of GRAPE+ with (a) Giraph [2] and synchronized $\text{GraphLab}_{\text{sync}}$ [26] under BSP, (b) asynchronized $\text{GraphLab}_{\text{async}}$, GiraphUC [28] and Maiter [57] under AP, (c) Petuum [53] under SSP, (d) PowerSwitch [52] under Hsync, and (e) GRAPE+ simulations of BSP, AP and SSP, denoted by $\text{GRAPE+}_{\text{BSP}}$, $\text{GRAPE+}_{\text{AP}}$, $\text{GRAPE+}_{\text{SSP}}$, respectively.

We find that $\text{GraphLab}_{\text{async}}$, $\text{GraphLab}_{\text{sync}}$, PowerSwitch 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 four systems in details. In all the experiments we also evaluated $\text{GRAPE+}_{\text{BSP}}$, $\text{GRAPE+}_{\text{AP}}$ and $\text{GRAPE+}_{\text{SSP}}$. Note that GRAPE [24] is essentially $\text{GRAPE+}_{\text{BSP}}$.

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

Graphs. We used five real-life graphs of different types, such that each algorithm was evaluated with two real-life graphs. These include (1) Friendster [1], a social network with 65 million users and 1.8 billion links; we randomly assigned weights to test SSSP; (2) traffic [7], an (undirected) US road network with 23 million nodes (locations) and 58 million edges; (3) UKWeb [8], a Web graph with 133 million nodes and 5 billion edges. We also used two recommendation networks (bipartite graphs) to evaluate CF, namely, (4) movieLens [4], with 20 million movie ratings (as weighted edges) between 138000 users and 27000 movies; and (5) Netflix [6], 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, *e.g.,* UKWeb, Friendster, are too large to fit in a single machine. Parallel processing is a must for them.

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.

Graph computations. 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 it is available. Otherwise we made our best efforts to develop "optimal" algorithms for them, *e.g.,* CF for PowerSwitch.

We used XtraPuLP [46] as the default graph partition strategy. 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.

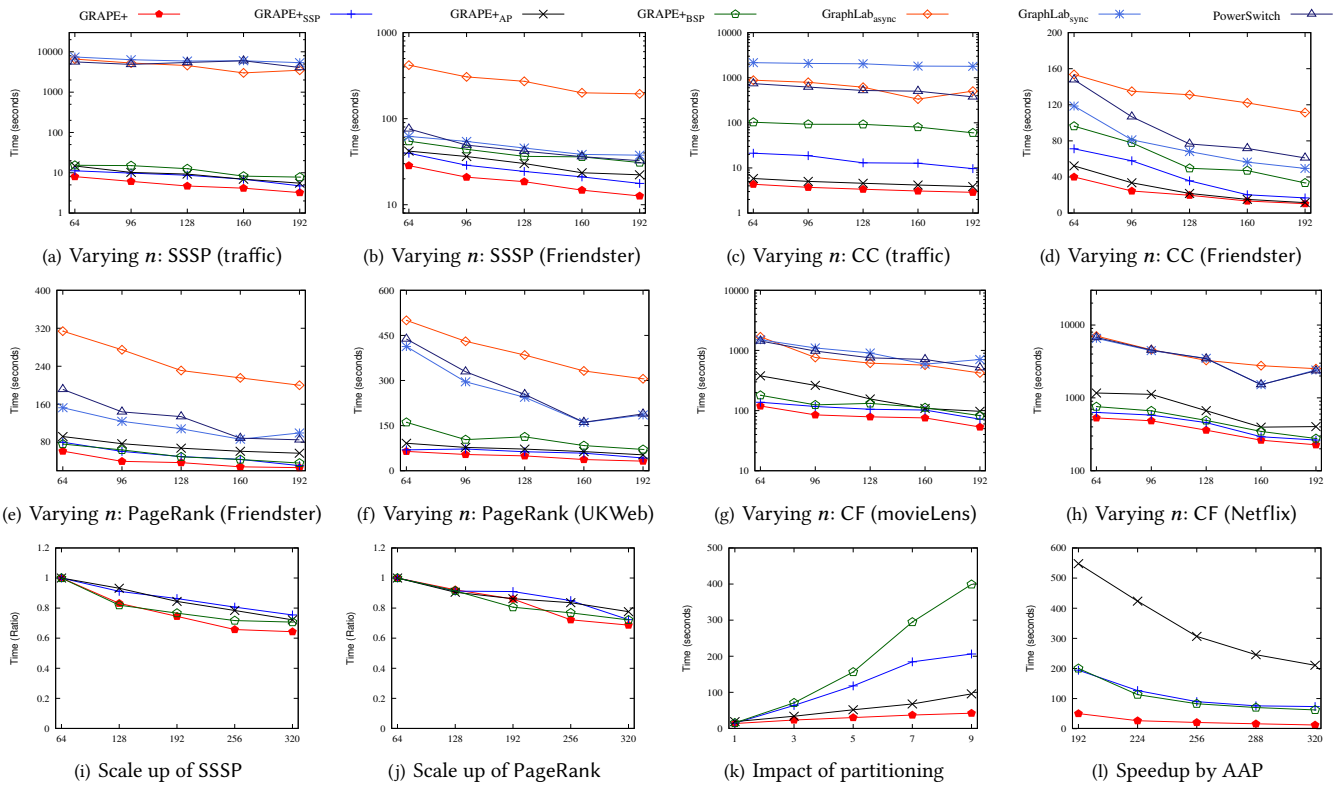
We deployed the systems on an HPC cluster. For each experiment, we used up to 20 servers, each with 16 threads of 2.40GHz, and 128GB memory. On each thread, a GRAPE+ worker is deployed. We ran each experiment 5 times. The average is reported here.

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 and Friendster; (b) PageRank with Friendster and UKWeb, and (c) CF with movieLens and Netflix, based on applications of these algorithms in transportation networks, social networks, Web rating and recommendation.

(1) SSSP. Figures 6(a) and 6(b) report the performance of SSSP.

(a) GRAPE+ consistently outperforms these systems in all cases. Over traffic (resp. Friendster) and with 192 workers, it is on average 1673 (resp. 3.0) times, 1085 (resp. 15) times and 1270 (resp. 2.56) times faster than synchronized $\text{GraphLab}_{\text{sync}}$, asynchronized $\text{GraphLab}_{\text{async}}$ and hybrid PowerSwitch, respectively.



(a) Varying n: SSSP (traffic) (b) Varying n: SSSP (Friendster) (c) Varying n: CC (traffic) (d) Varying n: CC (Friendster)

(e) Varying n: PageRank (Friendster) (f) Varying n: PageRank (UKWeb) (g) Varying n: CF (movieLens) (h) Varying n: CF (Netflix)

(i) Scale up of SSSP (j) Scale up of PageRank (k) Impact of partitioning (l) Speedup by AAP

Figure 6: Performance Evaluation

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 for sequential algorithms. Note that under BSP, AP and SSP, GRAPE+$_{\text{BSP}}$, GRAPE+$_{\text{AP}}$ and GRAPE+$_{\text{SSP}}$ can still benefit from (ii) and (iii).

As an example, GraphLab$_{\text{sync}}$ took 34 (resp. 10749) rounds over Friendster (resp. traffic), while by using IncEval, GRAPE+$_{\text{BSP}}$ and GRAPE+$_{\text{SSP}}$ 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 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 2.42, 1.71, and 1.47 (resp. 2.45, 1.76, and 1.40) times faster than GRAPE+$_{\text{BSP}}$, GRAPE+$_{\text{AP}}$ and GRAPE+$_{\text{SSP}}$ over traffic (resp. Friendster), up to 2.69, 1.97 and 1.86 times, respectively. Since GRAPE+, GRAPE+$_{\text{BSP}}$, GRAPE+$_{\text{AP}}$ and GRAPE+$_{\text{SSP}}$ 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 32.3% and 55.6% 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 37.2% and 47.1% of that of AP and SSP, respectively. These verify the effectiveness of AAP by dynamically adjusting relative progress of different workers.

(c) GRAPE+ takes less time when n increases. It is on average 2.49 and 2.25 times faster on traffic and Friendster, respectively, when n varies from 64 to 192. That is, AAP makes effective use of parallelism by reducing stragglers and redundant stale computations.

(2) CC. As reported in Figures 6(c) and 6(d) over traffic and Friendster, respectively, (a) GRAPE+ outperforms GraphLab$_{\text{sync}}$, GraphLab$_{\text{async}}$ and PowerSwitch. When $n = 192$, GRAPE+ is on average 313, 93 and 51 times faster than the three systems, respectively. (b) GRAPE+ is faster than its variants under BSP, AP and SSP, on average by 20.87, 1.34 and 3.36 (resp. 3.21, 1.11 and 1.61) times faster over traffic (resp. Friendster), respectively, up to 27.4, 1.39 and 5.04 times. (c) GRAPE+ scales well with the number of workers used: it is on average 2.68 times faster when n varies from 64 to 192.

(3) PageRank. As shown in Figures 6(e)-6(f) over Friendster and UKWeb, respectively, when $n = 192$, (a) GRAPE+ is on average 5, 9 and 5 times faster than GraphLab$_{\text{sync}}$, GraphLab$_{\text{async}}$ and PowerSwitch, respectively. (b) GRAPE+ outperforms GRAPE+$_{\text{BSP}}$, GRAPE+$_{\text{AP}}$ and GRAPE+$_{\text{SSP}}$ by 1.80, 1.90 and 1.25 times, respectively, up to 2.50, 2.16 and 1.57 times. This is because GRAPE+ reduces stale computations, especially those of stragglers. On average stragglers took 50, 27 and 28 rounds under BSP, AP and SSP, respectively, as opposed to 24 rounds under AAP. (d) GRAPE+ is on average 2.16 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 6(g)-6(h), respectively. On average (a) GRAPE+ is 11.9, 9.5, 10.0 times faster than GraphLab$_{\text{sync}}$, GraphLab$_{\text{async}}$ and PowerSwitch, respectively. (b) GRAPE+ beats GRAPE+$_{\text{BSP}}$, GRAPE+$_{\text{AP}}$ and GRAPE+$_{\text{SSP}}$ by 1.38, 1.80 and 1.26

times, up to 1.67, 3.16 and 1.38 times, respectively. (c) GRAPE+ is on average 2.3 times faster when n varies from 64 to 192.

Single-thread. Among the graphs traffic, movieLens and Netflix can fit in a single machine. On a single machine, it takes 6.7s, 4.3s and 2354.5s for SSSP and CC over traffic, and CF over Netflix, respectively. Using 64-192 workers, GRAPE+ is on average from 1.63 to 5.2, 1.64 to 14.3, and 4.4 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 not experienced by a single machine, just like other parallel systems. (b) Large graphs such as UKWeb are beyond the capacity of a single machine, and parallel computation is a must for such graphs.

Exp-2: Communication. Following [29], 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 and SSSP over Friendster 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 22.4%, 8.0% and 68.3% of data shipped by $\text{GraphLab}_{\text{sync}}$, $\text{GraphLab}_{\text{async}}$ and PowerSwitch, respectively. This is because GRAPE+ (a) reduces redundant stale computations and hence unnecessary data traffic, and (b) ships only changed values of update parameters by incremental IncEval.

(2) The communication cost of GRAPE+ is 1.22X, 40% and 1.02X 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 SSP.

Exp-3: Scale-up of GRAPE+. As observed in [41], 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 6(i) and 6(j) 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. 6(k) 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 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 [3] following the power law and the small world property. We used a cluster of up to 320 workers. As shown in Fig. 6(l) 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. Compared to the results in Exp-1, these show that 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. These further verify the effectiveness of AAP.

The results on other algorithms are consistent (not shown).

Summary. We find the following. (1) GRAPE+ consistently outperforms the state-of-the-art systems. Over real-life graphs and with 192 workers, GRAPE+ is on average (a) 2080, 838, 550, 728, 1850 and 636 times faster than Giraph, $\text{GraphLab}_{\text{sync}}$, $\text{GraphLab}_{\text{async}}$, GiraphUC, Maiter and PowerSwitch for SSSP, (b) 835, 314, 93 and 368 times faster than Giraph, $\text{GraphLab}_{\text{sync}}$, $\text{GraphLab}_{\text{async}}$ and GiraphUC for CC, (c) 339, 4.8, 8.6, 346, 9.7 and 4.6 times faster than Giraph, $\text{GraphLab}_{\text{sync}}$, $\text{GraphLab}_{\text{async}}$, GiraphUC, Maiter and PowerSwitch for PageRank, and (d) 11.9, 9.5 and 30.9 times faster than $\text{GraphLab}_{\text{sync}}$, $\text{GraphLab}_{\text{async}}$ and Petuum for CF, respectively. Among these PowerSwitch has the closest performance to GRAPE+. (2) It incurs as small as 0.0001%, 0.027%, 0.13% and 57.7% of the communication cost of these systems for these problems, respectively. (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 than BSP, AP and SSP, respectively, up to 5.0, 16.8 and 5.9 times with 320 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. (5) GRAPE+ scales well with the number n of workers used. It is on average 2.37, 2.68, 2.17 and 2.3 times faster when n varies from 64 to 192 for SSSP, CC, PageRank and CF, respectively. Moreover, it has good scale-up.

8 CONCLUSION

We have proposed AAP to remedy the limitations of BSP and AP by reducing both stragglers and redundant stale computations. As opposed to [54], 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 adjustment function δ for different computations. Another topic is to handle streaming updates by capitalizing on the capability of incremental IncEval.

REFERENCES

[1] Friendster. *https://snap.stanford.edu/data/com-Friendster.html.*
[2] Giraph. *http://giraph.apache.org/.*
[3] GTgraph. *http://www.cse.psu.edu/ kxm85/software/GTgraph/.*
[4] Movielens. *http://grouplens.org/datasets/movielens/.*
[5] MPICH. *https://www.mpich.org/.*
[6] Netflix prize data. *https://www.kaggle.com/netflix-inc/netflix-prize-data.*
[7] Traffic. *http://www.dis.uniroma1.it/challenge9/download.shtml.*
[8] UKWeb. *http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05/*, 2006.
[9] U. A. Acar. *Self-Adjusting Computation.* PhD thesis, CMU, 2005.
[10] U. A. Acar, A. Charguéraud, and M. Rainey. Scheduling parallel programs by work stealing with private deques. In *PPoPP*, 2013.
[11] K. Andreev and H. Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.
[12] J. Bang-Jensen and G. Z. Gutin. *Digraphs: Theory, Algorithms and Applications.* Springer, 2008.
[13] N. T. Bao and T. Suzumura. Towards highly scalable pregel-based graph processing platform with x10. In *WWW '13*, pages 501–508, 2013.
[14] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.
[15] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks*, 56(18):3825–3833, 2012.
[16] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.
[17] Y. Chen, S. Goldberg, D. Z. Wang, and S. S. Johri. Ontological pathfinding. In *SIGMOD*, 2016.
[18] C. C. Cowen, K. Davidson, and R. Kaufman. Rearranging the alternating harmonic series. *The American Mathematical Monthly*, 87(10):817–819, 1980.
[19] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. P. Xing. High-performance distributed ML at scale through parameter server consistency models. In *AAAI*, 2015.
[20] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.
[21] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C. Tseng. Dynamic load balancing of unbalanced computations using message passing. In *IPDPS*, 2007.
[22] W. Fan, C. Hu, and C. Tian. Incremental graph computations: Doable and undoable. In *SIGMOD*, 2017.
[23] W. Fan, J. Xu, Y. Wu, W. Yu, and J. Jiang. GRAPE: Parallelizing sequential graph computations. *PVLDB*, 10(12):1889–1892, 2017.
[24] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, B. Zhang, Z. Zheng, Y. Cao, and C. Tian. Parallelizing sequential graph computations. In *SIGMOD*, 2017.
[25] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34(3):596–615, 1987.
[26] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *USENIX*, 2012.
[27] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.
[28] M. Han and K. Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *PVLDB*, 8(9):950–961, 2015.
[29] M. Han, K. Daudjee, K. Ammar, M. T. Ozsu, X. Wang, and T. Jin. An experimental comparison of Pregel-like graph processing systems. *VLDB*, 7(12), 2014.
[30] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.
[31] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artif. Intell.*, pages 79–111, 2014.
[32] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *SODA*, pages 938–948, 2010.
[33] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys '13*, pages 169–182, 2013.
[34] M. Kim and K. S. Candan. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data & Knowledge Engineering*, 72:285–303, 2012.
[35] K. Knopp. *Theory and application of infinite series.* Courier Corporation, 2013.
[36] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.
[37] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling Distributed Machine Learning with the Parameter Server. In *USENIX*, 2014.
[38] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 2012.
[39] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.
[40] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, 2015.
[41] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what cost? In *HotOS*, 2015.
[42] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.
[43] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.
[44] S. Salihoglu and J. Widom. GPS: a graph processing system. In *SSDBM*, 2013.
[45] X. Shi, B. Cui, Y. Shao, and Y. Tong. Tornado: A system for real-time iterative analysis over evolving data. In *SIGMOD*, 2016.
[46] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri. Partitioning trillion-edge graphs in minutes. In *IPDPS*, 2017.
[47] Y. Tian, A. Balmin, S. A. Corsten, and J. M. Shirish Tatikonda. From "think like a vertex" to "think like a graph". *PVLDB*, 7(7):193–204, 2013.
[48] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
[49] L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science, Vol A.* 1990.
[50] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.
[51] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *SOCC*, pages 381–394, 2015.
[52] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In *PPOPP*, 2015.
[53] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A New Platform for Distributed Machine Learning on Big Data. *IEEE Transactions on Big Data*, 1(2):49–67, June 2015.
[54] D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1-2):1–195, 2017.
[55] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.
[56] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.
[57] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *TPDS*, 25(8):2091–2100, 2014.
[58] Z. Zhang and C. Douligeris. Convergence of synchronous and asynchronous algorithms in multiclass networks. In *INFOCOM*, pages 939–943. IEEE, 1991.

Acknowledgments. The authors are supported in part by 973 2014CB340302, ERC 652976, NSFC 61421003, EPSRC EP/M025268/1, and Beijing Advanced Innovation Center for Big Data and Brain Computing. Luo is also supported in part by NSFC 61602023.

APPENDIX A: PROOFS

Proof of Theorem 1

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. Using this property, we show that under conditions 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 in the trace of AAP. 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, the assigned values to x follow a partial order. Thus x has to be assigned $N_x + 1$ distinct values, which contradicts to the assumption that there are only N_x distinct values for x. □

Proof of Theorem 2

By Theorem 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 show the Church-Rosser property, 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 $(R_1^{r_1}, \ldots, R_m^{r_m}) = (\tilde{R}_1^{r^*}, \ldots, \tilde{R}_m^{r^*})$.

It suffices to prove the following:

(1) $R_i^{r_i} \preceq \tilde{R}_i^r$ 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 5);

(2) $\tilde{R}_i^{r^*} \preceq R_i^r$ 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^* (Lemma 6).

In the sequel, we will pick a run σ of ρ under AAP and compare the fixpoint in σ and σ^* by proving Lemma 5 and Lemma 6.

Notation. We first refine the partial order $\preceq$ on partial results R_i^l, which include update parameters $\bar{x}$. The order $\preceq$ implies a partial order $\preceq_p$ on the domain of update parameters. Denote by S_x and S'_x the 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 $\preceq_p$. For example, if $\preceq_p$ is the linear order over integers, $S_x = \{|1, 3, 3, 4, 5|\}$ and $S'_x = \{|3, 7, 7, 8|\}$, then $S_x \trianglelefteq S'_x$ since the minimal element 1 in S_x satisfies the condition. We extend $\trianglelefteq$ to collections $\bar{x}_i$. That is, we write $S_{\bar{x}_i} \trianglelefteq S'_{\bar{x}_i}$ if $S_x \trianglelefteq S'_x$ for each $x \in \bar{x}_i$.

Let S and T be the collections of updated parameters associated with R_i^s and R_i^t, respectively. Then the order $R_i^s \preceq R_i^t$ "covers" both the order on the "real" partial results R_i^s and R_i^t, and the order $S \trianglelefteq T$ on updated parameters. We restate conditions T2 and T3 as follows.

(T2) If $R_i^{s+1} = \mathsf{IncEval}(Q, R_i^s, F_i^s[\bar{x}_i], M_i)$ and $M_i = f_{\mathsf{aggr}}(\mathbb{B}_{\bar{x}_i} \cup C_i^s.\bar{x})$, then $R_i^{s+1} \preceq R_i^s$ and $C_i^{s+1}.\bar{x} \trianglelefteq M_i \trianglelefteq \mathbb{B}_{\bar{x}_i} \cup C_i^s.\bar{x}$.

(T3) If $\bar{R}_i^s \preceq R_i^t$ and their associated collections of update parameters S and T satisfy $S \trianglelefteq T$, then $\bar{R}_i^{s+1} \preceq R_i^{t+1}$ and $f_{\mathsf{aggr}}(S) \trianglelefteq f_{\mathsf{aggr}}(T)$, where $\bar{R}_i^s$ and R_i^t denote partial results in (possibly different) runs.

Lemma 5: *$R_i^{r_i} \preceq \tilde{R}_i^r$ for $i \in [1, m]$ and $r \geq 0$.* □

Proof: The lemma follows from the following two claims.

Fixpoint. In the run σ, each worker terminates at a fixpoint, *i.e.,* $R_i^{r_i} = \mathsf{IncEval}(Q, R_i^{r_i}, F^{r_i}[\bar{x}_i], M_i)$, where $M_i = f_{\mathsf{aggr}}(\emptyset \cup C_i^{r_i}.\bar{x})$. Here the fixpoint means that when the set of messages from other workers is $\emptyset$, $R_i^{r_i}$ cannot be further improved.

Consistency. When all workers terminate, the values of the update parameters are consistent, *i.e.,* for each $i, j \in [1, m]$ with $i \neq j$, and for each variable $x \in C_i \cap C_j$, x has the same value in $R_i^{r_i}$ and $R_j^{r_j}$.

Assuming these claims, we show Lemma 5 by induction on r.

• *Basis 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 can show *w.l.o.g.* that $R_1^{r_i} \preceq \tilde{R}_1^{r+1}$ by using the inductive hypothesis and the monotonicity of IncEval. Let $\tilde{R}_1^{r+1} = \mathsf{IncEval}(Q, \tilde{R}_1^r, F_1^r[\bar{x}_1], M_1)$, where $M_1 = f_{\mathsf{aggr}}(m_1 \cup \ldots \cup m_k \cup \tilde{C}_1^r.\bar{x})$ (updates to $\bar{x}_1$), and m_j is message from $\tilde{R}_{i_j}^r$ for $j \in [1, k]$. By the fixpoint claim, $R_1^{r_1} = \mathsf{IncEval}(Q, R_1^{r_1}, F_1^{r_1}[\bar{x}_1], M'_1)$, where $M'_1 = f_{\mathsf{aggr}}(\emptyset \cup C_1^{r_1}.\bar{x})$, and P_1 terminates at round r_1. By the inductive hypothesis, $R_1^{r_1} \preceq \tilde{R}_1^r$. □

It remains to verify the two claims.

Proof of the Fixpoint Claim. It suffices to show that when AAP terminates, $C_i^{r_i}.\bar{x} \subseteq (\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x})$ ($i \in [1, m]$), where $\mathbb{B}_{\bar{x}_i}$ 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 $C_i^{r_i}.\bar{x} \subseteq (\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x})$, we can verify that $R_i^{r_i} = \mathsf{IncEval}(Q, R_i^{r_i}, F^{r_i}[\bar{x}_i], M_i)$ in two steps.

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

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

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

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

It remains to show that $C_i^{r_i}.\bar{x} \subseteq (\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x})$. We verify this by contradiction. Suppose that $C_i^{r_i}.\bar{x} \not\subseteq (\mathbb{B}_{\bar{x}_i} \cup C_i^{r_i-1}.\bar{x})$. By T2, 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. Then 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 values of x^* on P_{j_ℓ} for $\ell \in [1, k]$ when P_{j_ℓ} processes the message. There are two cases. (i) If some val_ℓ ($\ell \in [1, k]$) satisfies $\mathsf{val}_\ell \preceq_p \mathsf{val}$, then AAP sends a message containing $(x^*, \mathsf{val}_\ell, r_\ell)$ to P_i. It contradicts to 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 $\mathsf{val}_1, \ldots \mathsf{val}_k$, then by the contraction of IncEval (T2), $P_{j_1}, \ldots, P_{j_k}$ update the value of x^* to $\mathsf{val}'_1, \ldots, \mathsf{val}'_k$, respectively, such that $\mathsf{val}'_1 \preceq_p \mathsf{val}, \ldots, \mathsf{val}'_k \preceq_p \mathsf{val}$. 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. □

Proof of the Consistency Claim. We 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 $\mathsf{val}_i = \mathsf{val}_j$. Let val_i^0 and val_j^0 be the initial values of x in F_i and F_j, respectively; then $\mathsf{val}_i^0 = \mathsf{val}_j^0$. Observe the following. (a) If PEval and IncEval do not update x, then $\mathsf{val}_i = \mathsf{val}_j$. (b) Otherwise assume *w.l.o.g.* that $\mathsf{val}_i \preceq_p \mathsf{val}_i^0$ and $\mathsf{val}_i \neq \mathsf{val}_i^0$. Consider the round updating x on F_i by assigning some $\mathsf{val}'_i \neq \mathsf{val}_i$ to x. 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, we have that $\mathsf{val}_j \preceq_p \mathsf{val}_i$. By a similar argument, we can show that $\mathsf{val}_i \preceq_p \mathsf{val}_j$. Putting these together we have that $\mathsf{val}_i = \mathsf{val}_j$. □

Lemma 6: *$\tilde{R}_i^{r^*} \preceq R_i^r$ for $i \in [1, m]$ and $r \geq 0$.* □

Proof: Since the BSP run σ^* is a special run under AAP, by the fixpoint and consistency claims (Lemma 5), we know the following: (a)

$\tilde{R}_i^{r^*}$ is a fixpoint, *i.e.,* $\tilde{R}_i^{r^*} = \mathsf{IncEval}(Q, \tilde{R}_i^{r^*}, F_i^{r^*}[\bar{x}_i], M_i)$, where $M_i = f_{\mathsf{aggr}}(\emptyset \cup C_i^{r^*}.\bar{x})$; (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 $\tilde{R}_i^{r^*} \preceq R_i^r$.

(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 on decreasing t. For the basis case where $t = 0$, $\tilde{R}_j^{r^*} \preceq \tilde{R}_j^0 = R_j^0$ by the contraction of IncEval (T2). For the inductive step, suppose that (j, t) has children $(j, t-1)$, (j_1, t_1), (j_2, t_2), …, (j_k, t_k), and that the inductive hypothesis holds for $(j, t-1)$, (j_1, t_1), (j_2, t_2), …, (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, …, m_k are messages from workers P_{j_1}, …, 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 $M_j' = f_{\mathsf{aggr}}(\emptyset \cup C_i^{r^*}.\bar{x})$ 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 5.

This completes the proof of Lemma 6 and hence Theorem 2. □

Proof of Theorem 4

Since PRAM can be simulated by MapReduce [32], 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 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 [20, 32]. Denote by I_r the output of subroutine B_r ($r \in [1, k]$).

(*i*) 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.

(*ii*) 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}, …, and G_{k_j} be the obtained groups.

(*iii*) 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.

(*iv*) 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 and 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 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.

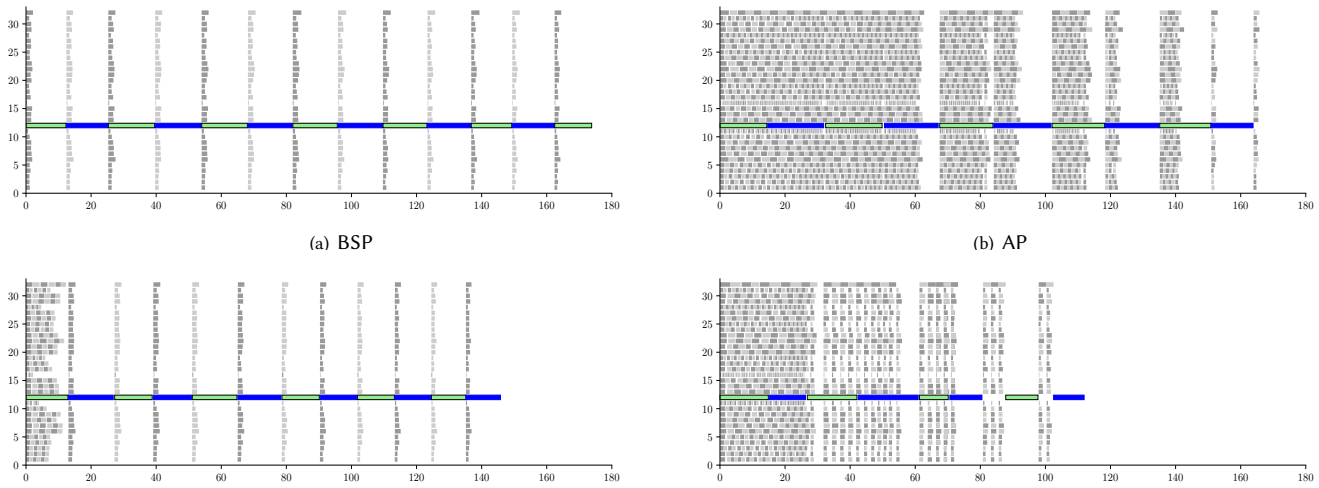
The aggregation function first takes a union of the update parameters of all w_i ($i \in [1, n]$), and then groups the tuples by *key*.

(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*$\rangle$ pairs 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*$\rangle$ pairs 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 $O(C)$ amount of data. Formally, this is verified by induction on k for the number of subroutines $(B_1, \ldots, B_k)$ in $\mathcal{A}$. □



(a) BSP (b) AP (c) SSP (c=5) (d) AAP

Figure 7: Case Study of PageRank

APPENDIX B: MORE EXPERIMENTAL RESULTS

We did two case studies to show how AAP adaptively adjusts delay stretches and reduces response time, with PageRank and CF.

(1) PageRank. Figure 7 shows the timing diagrams of GRAPE+$_{\mathsf{BSP}}$, GRAPE+$_{\mathsf{AP}}$, GRAPE+$_{\mathsf{SSP}}$ and GRAPE+ for PageRank over real-life Friendster. Among 32 workers used, P_{12} is a straggler (colored in blue and green). Stragglers often arise in the presence of streaming updates, even when we start with an evenly partitioned graph.

(a) BSP. As shown in Figure 7(a), the straggler P_{12} dominated the running time. Each superstep of the BSP run was slowed down by the straggler due to the global synchronization. The other 31 workers mostly idled, and the run took 13 rounds and 174s.

(b) AP. GRAPE+$_{\mathsf{AP}}$ did slightly better and took 166s, as shown in Figure 7(b). Idling was substantially reduced. However, fast workers performed far more rounds of computation than BSP, and most of these are redundant. The cost was still dominated by the 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 7(c) depicts a run of GRAPE+$_{\mathsf{SSP}}$ 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 7(d) 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 initial $\mathsf{L}_\perp = 0$ to start with.

AAP adjusted delay stretch DS_{12} at straggler P_{12} as follows. (i) Until round 6, function δ kept $\mathsf{DS}_{12} = \eta_{12}$ 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 63 based on predicted running time t_{12} of IncEval at P_{12} and message arrival rate s_{12}, which were estimated by aggregating consecutive executions of IncEval at all workers. 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 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 estimate of t_i and s_i was quite accurate since the speed of IncEval at the same worker did not drastically vary. (iv) If users opt to set $\mathsf{L}_\perp$ as, *e.g.,* 31, function δ can start reduce 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. We find the following.

(a) BSP. BSP converged in the least rounds (351), but it incurred excessive idleness and was 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 [53].

(c) SSP. Tuning c was helpful. However, it is hard to find an optimal c for SSP. We had to run GRAPE+$_{\mathsf{SSP}}$ 50 times to find the optimal c_o.

(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 tests.