ASPIRE: Exploiting *As*ynchronous *P*arallelism in *I*terative Algorithms using a *Re*laxed Consistency based DSM

Keval Vora   Sai Charan Koduru   Rajiv Gupta
CSE Department, University of California, Riverside
{kvora001,scharan,gupta}@cs.ucr.edu

Abstract

Many vertex-centric graph algorithms can be expressed using asynchronous parallelism by relaxing certain read-after-write data dependences and allowing threads to compute vertex values using stale (i.e., not the most recent) values of their neighboring vertices. We observe that on distributed shared memory systems, by converting synchronous algorithms into their asynchronous counterparts, algorithms can be made tolerant to high inter-node communication latency. However, high inter-node communication latency can lead to excessive use of stale values causing an increase in the number of iterations required by the algorithms to converge. Although by using bounded staleness we can restrict the slowdown in the rate of convergence, this also restricts the ability to tolerate communication latency. In this paper we design a *relaxed memory consistency model* and *consistency protocol* that simultaneously tolerate communication latency and minimize the use of stale values. This is achieved via a coordinated use of *best effort refresh* policy and *bounded staleness*. We demonstrate that for a range of asynchronous graph algorithms and PDE solvers, on an average, our approach outperforms algorithms based upon: prior relaxed memory models that allow stale values by at least 2.27x; and Bulk Synchronous Parallel (BSP) model by 4.2x. We also show that our approach frequently outperforms *GraphLab*, a popular distributed graph processing framework.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.3.4 [*Programming Languages*]: Processors—Runtime environments

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
OOPSLA '14, October 19 - 21, 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-2585-1/14/10...$15.00.
http://dx.doi.org/10.1145/2660193.2660227

Keywords Distributed Shared Memory; Communication Latency; Bounded Staleness; Best Effort Refresh; PDE Solvers; Graph Mining; Graph Analytics

1. Introduction

The data- and compute-intensive nature of many modern applications makes them suitable candidates for execution on large scale clusters. A subclass of these applications are iterative graph algorithms (e.g., graph mining, graph analytics, PDE solvers) that can be expressed via *asynchronous parallelism* by relaxing certain read-after-write data dependences. This allows the threads to perform computations using stale (i.e., not the most recent) values of data objects. It is well known that asynchronous algorithms can improve the efficiency of parallel execution on shared-memory multiprocessors by avoiding the costly synchronizations that are required by the corresponding synchronous algorithms [6, 14]. Recent works [44, 48] have proposed programming support and scheduling techniques for taking advantage of asynchrony in exploiting loop level parallelism on shared-memory systems.

Exploitation of asynchrony to enhance performance on distributed shared memory (DSM) based clusters is largely an unexplored topic. With DSM, although an application thread can access all objects as if they are stored in shared memory, the access latency experienced by the thread varies based upon the node at which the object is physically stored. Therefore the key to obtaining high-performance on a cluster is successfully tolerating the high inter-node communication latency. On our *Tardis* cluster which has 16 nodes, the latency of accessing an object stored in the DSM increases by 2.3 times if instead of being found in the local cache (DSM) it has to be brought from a remote node. We observe that to tolerate the long inter-node communication latency, the asynchronous nature of algorithms can be exploited as follows. When a thread's computation requires an object whose stale value is present in the local cache, instead of waiting for its most recent copy to be fetched from a remote node, the stale value can be used. By tracking the *degree of staleness* of cached objects in terms of the number of object updates that have since been performed, we can limit the extent to

which use of stale values is permitted. However, fruitfully exploiting this idea is challenging due to the following:

- *Allowing high degree of staleness* can lead to excessive use of stale values which in turn can slow down the algorithm's convergence, i.e. significantly more iterations may be needed to converge in comparison to the synchronous version of the algorithm. This can wipe out the benefits of exploiting asynchrony.
- *Allowing low degree of staleness* limits the extent to which long latency fetch operations can be tolerated as the situation in which the cached value is too stale to be used becomes more frequent causing delays due to required inter-node fetches.

In this paper we address the above challenge by designing a relaxed memory consistency model and cache consistency protocol that simultaneously *maximize the avoidance of long latency communication operations* and *minimize the adverse impact of stale values on convergence*. This is achieved by the following two ideas:

- First, we use a consistency model that supports bounded staleness and we set the permissible bound for using stale values to a high threshold. By doing so, we *increase the likelihood of avoiding long latency fetch operations* because if a stale value is present in the cache it is highly likely to be used.
- Second, we design a cache consistency protocol that incorporates a policy that *refreshes* stale values in the cache such that when a stale value is used to avoid a long latency operation, the likelihood of the value being minimally stale is increased. Thus, this refresh policy *avoids slowing down the convergence* of the algorithm due to use of excessively stale values.

Hence, significant performance improvements can be obtained by effectively tolerating inter-node fetch latency. The focus of our work is on asynchronous graph algorithms and thus we rely upon an object-based DSM as opposed to a page-based DSM. However, our ideas are equally applicable to page-based systems.

Extensive research has been conducted on memory consistency models for parallel environments [2, 7, 11, 13, 19, 21, 22, 26, 27, 31, 38, 39, 43, 60, 67] and a hierarchy of these models can be found in [23, 49]. From the perspective of our work these models can be categorized into three main categories. Some of the models are too strong – for example, *cache consistency* [21] guarantees that each read of a memory location obtains the most recently written value, i.e. it does not permit the use of stale values. Other models are too weak – for example *slow memory* [26] and *PRAM* [43] have been shown to allow programming of asynchronous algorithms [22] but their use requires significant programming effort to ensure program correctness. As shown in [24], the time taken for flow of updates in PRAM increases rapidly with the number of processes, which delays convergence.

Finally, the models that are most relevant to our work are ones that allow *bounded staleness*, i.e. they allow reads to obtain stale values as long as they are not too stale as determined by a specified bound. Staleness is measured either in terms of number of versions by which the value is out-of-date (InterWeave [11]) or in terms of number of iterations since the object received its value (SSP [13]). While we also make use of bounded staleness in our work, as we demonstrate in this paper, in context of asynchronous algorithms this alone is not enough. It must be coupled with a strategy for mitigating the adverse impact of stale values on the algorithm's convergence. Finally, a number of graph processing frameworks have been developed. This includes distributed frameworks such as *Giraph* [1], *Pregel* [47], *GraphLab* [46], and *PowerGraph* [20]. The shared-memory based graph processing frameworks provide efficient solutions while addressing specific challenges: constrained memory (*GraphChi* [37]) and easy programmability (*Ligra* [59]). In [50] a shared-memory based implementation of a graph processing DSL is developed using *Galois* [36] system. These frameworks are largely based upon the Bulk Synchronous Model (BSP) [64] and do not provide support for programming asynchronous algorithms that leverage the use of stale objects to improve performance. *GRACE* [66], a shared memory based graph processing system, uses message passing and provides asynchronous execution by using stale messages. Since shared-memory processing does not suffer from communication latencies, these systems can perform well for graphs that fit on a single multicore server.

The key contributions of this work are as follows:

1. We present a vertex centric approach for programming asynchronous algorithms with ease using Distributed Shared Memory (DSM) that is based upon a memory consistency model that incorporates bounded staleness.
2. We design a consistency protocol that tracks staleness and incorporates a policy for refreshing stale values to tolerate long latency of communication without adversely impacting the convergence of the algorithm. Our experiments show that the use of this fetch policy is critical for the high performance achieved.
3. We demonstrate that, on an average, our asynchronous versions of several graph algorithms outperform algorithms based upon: prior relaxed memory models that allow stale values by at least 2.27x and Bulk Synchronous Parallel (BSP) [64] model by 4.2x. We also show that our approach performs well in comparison to *GraphLab*, a popular distributed graph processing framework.

The remainder of the paper is organized as follows. Section 2 shows how we express asynchronous parallel algorithms. In Section 3, we present the *relaxed object consistency model* that incorporates the use of stale objects for fast

access and the design of our *DSM system* that implements the relaxed execution model. The key component of DSM, the *relaxed consistency protocol*, is presented in Section 4. We discuss the implementation of our prototype, experimental setup, and results of evaluation in Section 5. Related work and conclusions are given in Sections 6 and 7.

2. Asynchronous Parallelism

We consider various iterative algorithms that move through the solution search space until they converge to a stable solution. In every iteration, the values are computed based on those which were computed in the previous iteration. This process continues until the computed values keep on changing across subsequent iterations. The asynchronous model for parallel computation allows multiple threads to be executed in parallel using a set of possibly outdated values accessible to those threads. By using an outdated value, a thread avoids waiting for the updated value to become available. Algorithm 1 shows a basic template for such convergence based iterative algorithms. A set of threads execute the DO-WORK method which iteratively performs three tasks: fetch inputs, compute new values, and store newly computed values. At the end of DO-WORK, if a thread detects that the values have not converged, it votes for another iteration. This process ends when no thread votes for another iteration, which is detected in the MAIN method.

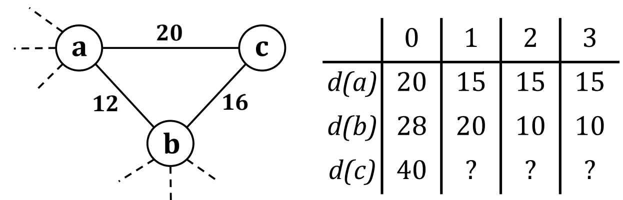
The DSM-FETCH method fetches an object from the DSM (line 7). For example, in a vertex centric graph algorithm, the vertex to be processed is initially fetched using this method. Then, to fetch its neighbors, again, DSM-FETCH is used (line 10). In synchronous versions of these algorithms, this DSM-FETCH method incurs very high latency because it may result in a remote fetch to access the most recent value. However, when these algorithms are implemented using the asynchronous computational model, some of the objects used in the computation are allowed to reflect older values, i.e., they do not reflect the most recent changes. Hence, these methods may return a stale vertex value.

Let us consider the single source shortest path (SSSP) algorithm which computes the shortest path from a given source node to each node in a graph. To guarantee convergence, the iterative algorithm assumes that the graph does not have a negative cycle. Figure 1 shows an example subgraph along with the distance values calculated for nodes a and b at the end of iterations $i = 0, 1, 2, 3$ and the initial value for c at the end of iteration $i = 0$. Since the algorithm is implemented based on asynchronous parallelism, a perfectly valid execution scenario can be as follows:

- During $i = 1$, c observes $d^0(a) = 20$ and $d^0(b) = 28$. Hence, $d(c)$ is set to $min(20 + 20, 16 + 28) = 40$.
- During $i = 2$, only updated value $d^1(b) = 20$ is observed by c and $d(c)$ is set to $min(20 + 20, 16 + 20) = 36$.

Algorithm 1 A basic template for Iterative Algorithms

```
 1: function DO-WORK(thread-id)
 2:     curr ← GET-START-INDEX(thread-id)
 3:     end ← GET-END-INDEX(thread-id)
 4:     error ← ϵ
 5:     while curr < end do
 6:         oid ← GET-OBJECT-ID(curr)
 7:         object ← DSM-FETCH(oid)
 8:         r-objects ← ∅
 9:         for r-id ∈ object.get-related-object-ids() do
10:             r-objects ← r-objects ∪ DSM-FETCH(r-id)
11:         end for
12:         old-value ← object.get-value()
13:         comp-value ← f (object, r-objects)
14:         object.set-value(comp-value)
15:         DSM-STORE(object)
16:         error = MAX(error, |old-value − comp-value|)
17:         curr ← curr + 1
18:     end while
19:     /* Local termination condition */
20:     if error > ϵ  then
21:         vote to continue
22:     end if
23: end function
24:
25: function MAIN
26:     INITIALIZE-DSM(object-set)
27:     do
28:         parallel-for all threads do
29:             DO-WORK(thread-id)
30:         end parallel-for
31:         BARRIER
32:         /* Global termination condition */
33:     while at least one thread votes to continue
34: end function
```



	0	1	2	3
d(a)	20	15	15	15
d(b)	28	20	10	10
d(c)	40	?	?	?

Figure 1: An example subgraph for SSSP.

- During $i = 3$, c observes that $d^2(a) = 15$, but it remains oblivious to the change in $d(b)$ which leads to $d(c)$ to be set to $min(20 + 15, 16 + 20) = 35$.
- During $i = 4$, c observes that $d^2(b) = 10$ and that $d^2(a)$ is still the same. Hence, $d(c)$ is set to $min(20 + 15, 16 + 10) = 26$.

	0	1	2	3	4	5	6	7	8	
d(a)	20	-	16	-	-	15	14	13	-	
d(b)	28	22	21	-	-	18	17	9	-	
d(c)	40	-	38	36	-	-	34	33	25	*threshold = 0*
d(c)	40	-	-	37	-	-	35	33	-	*threshold = 1*
d(c)	40	-	-	-	-	-	34	-	-	*threshold = 2*

Figure 2: Execution instances showing intermediate values of $d(c)$ for statically set staleness thresholds of 0, 1 and 2.

In the above execution, $d(c)$ was always computed using $d(a)$ and $d(b)$, one of which did not reflect recent changes. Using older values made the intermediate values of $d(c)$ inconsistent. However, if the updated values of both $d(a)$ and $d(b)$ are guaranteed to be observed during future iterations, the algorithm converges to its correct solution. Hence, an intuitive and straightforward way to hide fetch latencies in these algorithms is to allow the use of previously fetched older values. This can be achieved by using *delta coherence* as shown in [11], that allows objects which are no more than x versions out-of-date. However, statically maintaining x as a threshold will not yield the expected performance benefits.

Figure 2 shows execution instances when staleness threshold is statically set to 0, 1 and 2. When the threshold is 0, change in $d(b)$ from 28 to 22 at the end of iteration 1 is immediately seen in iteration 2. This change gets hidden when the threshold > 0; for e.g., with threshold = 1, $d(b) = 21$ is noticed directly in iteration 3 and $d(b) = 22$ is never seen. As we can see in Figure 2, setting the staleness threshold to 1 or 2 allows computations to avoid immediate fetches; however, the computations choose to work with stale values (for e.g., in iterations 4 and 5) even when algorithms could have progressed using fresher values. These computations can be considered redundant or wasteful. Note that $d(a)$ and $d(b)$ can continue to remain unchanged (as in iterations 3 and 4) across a long series of consecutive iterations, making the situation worse for any threshold > 0. A key observation is that if any subset of the set of values is used to compute the new value updates across subsequent iterations, the algorithm can be deemed to have progressed across these iterations. Hence, it is important for the updated values to be observed by required computations in order to make a faster progress towards the correct stable solution.

Also, when a requested object's staleness has already crossed the static threshold, a DSM fetch is required. This enforces a limit on the number of remote fetches that can be avoided. For example, for a threshold x, every x^{th} access to an object can potentially cause a fetch from its global copy which may be present on a remote location.

Note that SSSP's monotonic nature along with the miniature subgraph in the example allows the discussion at hand to be simple; however, the observations drawn from this example apply to other more complex situations too.

To formalize the discussion, we define following terms:

Current Object: An object whose value reflects the most recent change.

Stale Object: An object which was current at some point in time before the present time.

Staleness of an object: The number of recent changes which are not reflected in the object's value.

In our example, during $i = 2$, $d^1(a) = 20$ is a current object and $d^0(b) = 28$ is a stale object. It is easy to follow that the staleness value of current objects is always 0.

In summary, we draw the following conclusions for asynchronous versions of convergence based iterative algorithms.

- Since these algorithms do not enforce strict data dependence constraints (in particular, read-after-write dependences for objects), they can tolerate use of stale objects.
- To maintain a good convergence rate, it is recommended that these algorithms rely more on the current values of objects and less on the stale values of objects.

Even though these conclusions inherently seem contradictory, they give us a key insight that maintaining a right mix of current and stale objects along with carefully balancing staleness of these objects can lead to better performance of these algorithms on DSM.

By allowing computations to use stale objects, the path to the final solution in the solution search space may change. This means that the total number of iterations required to converge to the final solution may also vary. However, since local caches can quickly provide stale objects, data access will be faster and time taken to execute a single iteration will drastically reduce. This reduction in time can result in significant speedups for various iterative algorithms if we minimize the staleness of values available without stalling computation threads. Note that since stale objects are often used for computations, termination of these iterative algorithms needs to be handled carefully and additional checks must be performed along with the algorithm specific termination conditions (discussed further in Section 4).

This analysis motivates the need for a relaxed model that can provide fast access to, possibly old, data that is minimally stale in order to achieve better performance for convergence based iterative algorithms.

3. Relaxed Object Consistency Model

The relaxed object consistency model we present accomplishes two goals. First, it achieves *programmability* by providing a single writer model that makes it easy to reason about programs and intuitive to write correct asynchronous

parallel algorithms. Second, it enables *high performance* through low latency access of objects which requires careful (minimal) use of stale objects. To achieve the above goals, we have identified four constraints that together describe our consistency model and are enforced by our cache consistency protocol. Next we present these constraints and an overview of how they are enforced.

***Object consistency constraints* for Programmability.** We define our consistency model with respect to a *single object*, i.e. we do not enforce any ordering among operations on different objects even if they are potentially causally related. For each object, we rely upon having a *single writer*, i.e. the same machine is responsible for updating a particular data item in every iteration. Our iterative object centric approach for programming asynchronous algorithms naturally maps to the single writer discipline and allows the programmer to intuitively reason about the program. We enforce the single writer discipline by fixing the assignment of computations to machines such that threads on the same machine update the same set of objects in every iteration. Although our consistency model does not enforce any ordering on the writes to an object from different machines, the programmer does not need to be concerned about this as chaotic writes to the same object by multiple machines are prohibited by ensuring that there is only a single writer for each object. Using the single writer discipline gives us another advantage – our consistency protocol does not have to deal with multiple writes to same objects. This simplifies the consistency protocol by eliminating the need to maintain write/exclusive object states. Now we are ready to state two of our constraints on writes and reads to each object and describe how they are enforced.

(Local Updates) *Local writes must be immediately visible*. This constraint enforces an ordering on multiple writes to an object by the same machine. To satisfy this constraint and provide strict ordering of writes to an object by its single writer, threads in our system do not maintain any thread-local cache and all writes directly go to the global copy of the object. Our system employs *machine level caches* to make remote objects locally available; these caches are *write through* to make writes visible across different machines.

(Progressive Reads) *Once an object is read by a thread, no earlier writes to it can be read by the same thread.* This constraint makes our model intuitive to programmers by guaranteeing that the updated values for an object will be seen in the order of writes to that object. Since we only have one global copy of an object at its home machine, any stale-miss or a refresh on stale-hit (described later) at another machine will only make its local copy current.

As we see, the above two constraints are primarily required to guarantee correctness and allow programmers to intuitively reason about program execution.

***Object consistency constraints* for Performance.** For high performance we must permit the use of stale objects and avoid long latency communication operations. The constraints we present next involve the use of stale object values.

(Bounded Staleness) *A read is guaranteed to receive an object whose staleness is no more than a particular threshold.* A bound on staleness allows the threads to notice the updated values at some point in the future. This constraint is satisfied by altering the definition of a cache hit as described in the next section. The strictness of this bound can be relaxed using asynchronous invalidate messages, as done in our protocol.

(Best Effort Refresh) *A series of reads by the same thread for the same object should preferably reflect updated values, independent of the threshold.* The previous constraint alone does not guarantee that updates will be observed by threads that depend on those updates. Hence, this final constraint is introduced to allow threads to quickly observe the updated values which helps the algorithm to progress at a faster rate. This final constraint is enforced by our cache consistency protocol which specifically employs a mechanism for *asynchronously refreshing objects on stale-hits* to allow fast access to updated objects.

Any DSM implementation that wishes to satisfy our object consistency model *must* satisfy the first three constraints. This does not mean that the fourth constraint can be ignored. Even though the fourth constraint is a loose constraint, the protocol is expected to do its best to satisfy this constraint.

4. Relaxed Consistency Protocol

Next we introduce the new consistency protocol which satisfies the model proposed in the previous section. In Section 4.1, we introduce various notations and terms which will be used to discuss the working of the protocol in Section 4.2.

4.1 Definitions and Notation

Formally, $M = \{m_0, m_1, \ldots, m_{k-1}\}$ is the set of k machines (nodes) in the cluster. The mapping function h maps an object o to its *home machine* m_i i.e., on DSM, if o resides on m_i, then $h(o) = m_i$.

Every machine $m_i \in M$ has a cache c_i which locally stores objects and tracks their staleness. An entry in the cache is of the form $\langle o, staleness \rangle$ where o is the actual object and $staleness$ is its staleness value. Since we do not use thread-level caching, these caches provide the fastest data access.

Every machine $m_i \in M$ has a directory d_i to track the set of machines which access the objects mapped to that machine. A directory entry for an object o is of the form $d_i^o = \{m_j \mid m_j \in M \text{ and } o \in c_j\}$ $\forall$ o such that $h(o) = m_i$.

Also, we keep a threshold t which is used to determine the usability of locally available o. Hence, o can be considered

$$\begin{aligned} current &\quad \text{if } staleness = 0 \\ stale &\quad \text{if } 0 < staleness \leq t \\ invalid &\quad \text{if } staleness > t \end{aligned}$$

We change the meaning of a hit and a miss in the cache as follows. If the requested object in local cache is either current or stale, it is a hit. Otherwise, it is a miss. Hence, for an object o requested at a machine m_i, we determine a hit or a miss as follows:

$$\begin{aligned} hit &\quad \text{if } o \in c_i \text{ and } staleness \leq t \\ miss &\quad \text{otherwise} \end{aligned}$$

For ease of discussion, we further categorize a hit and a miss. For a requested object which is present in the local cache, it is a *current-hit*, a *stale-hit* or a *stale-miss* if the object in cache is current, stale or invalid, respectively. If the requested object is not in the local cache, it is simply a *cache-miss*. Hence, for an object o requested at a machine m_i, the result can be one of the following:

$$\begin{aligned} current\text{-}hit &\quad \text{if } o \in c_i \text{ and } staleness = 0; \\ stale\text{-}hit &\quad \text{if } o \in c_i \text{ and } 0 < staleness \leq t; \\ stale\text{-}miss &\quad \text{if } o \in c_i \text{ and } staleness > t; \\ cache\text{-}miss &\quad \text{if } o \notin c_i. \end{aligned}$$

4.2 Protocol

In this section, we discuss the basic working of our protocol using the terms introduced in the previous section. The traditional directory based coherence mechanism [10, 40] is useful to enforce strict consistency. Our protocol extends the directory based protocol to control the degree of coherence, as required at runtime. In the following discussion, we assume that machine m_i requests for object o.

On a cache-miss, m_i first sends a read request to $m_j = h(o)$. On receiving the read request from m_i, m_j sets $d_j^o \leftarrow d_j^o \cup \{m_i\}$. After fetching o from the DSM, m_i adds $\langle o, 0\rangle$ to c_i . While adding $\langle o, 0\rangle$ to c_i, if c_i is full, object o' is evicted from c_i based on the Least Recently Used (LRU) replacement policy. To evict o', m_i sends an eviction message to $m_p = h(o')$. On receiving the eviction message from m_i, m_p sets $d_p^{o'} \leftarrow d_p^{o'} \setminus \{m_i\}$.

When m_i writes o back to the DSM, it sends a write message to $m_j = h(o)$ and continues immediately. On receiving the write message from m_i, m_j *asynchronously* sends an invalidation message to all $m_q \in d_j^o \setminus \{m_i\}$ and sets $d_j^o \leftarrow d_j^o \cup \{m_i\}$. When m_q receives invalidation for o, it sets $\langle o, staleness\rangle \leftarrow \langle o, staleness + 1\rangle$ in c_q. We use invalidate-on-writes instead of broadcasting updates so that we can avoid consecutive updates to be propagated to remote nodes which makes the intermediate updates, before the object is actually read, redundant.

On a stale-miss, the current value of o is fetched from the DSM and m_i sets $\langle o, staleness\rangle \leftarrow \langle o_{curr}, 0\rangle$ in c_i.

On a current-hit, the local copy of o is used by m_i. No further processing is required in this case.

On a stale-hit, the local copy of o is used by m_i and a DSM fetch request is issued asynchronously to *refresh* the local copy of o. When the current value of object is received from the DSM, m_i sets $\langle o, staleness\rangle \leftarrow \langle o_{curr}, 0\rangle$ in c_i.

By allowing cache misses in the traditional protocol to be considered as cache hits in our protocol, $o \in c_i$ can remain outdated until its $staleness \leq t$. To allow visibility of more recent values of o for subsequent reads on m_i, the protocol incorporates *refresher threads*. The refresher thread observes that m_i has read a stale value of o from c_i as its $staleness > 0$; hence, it initiates a fetch to update $o \in c_i$ with its current value from the DSM. This prevents o from remaining outdated for long time and thus causes subsequent reads to receive fresher values.

Figure 3 shows the state transition diagram for object entries in machine caches. The gray text in parentheses indicates the source of the event and the small text at the bottom of the transitions show how an object's staleness is maintained. The shared state represents that the object is current. On receiving an invalidation message for a current object, the state of object changes to stale state. Every invalidation message increments the staleness by 1. A hit occurs when the object is either in shared or stale state and the staleness of the object is at most equal to the threshold value. If the current staleness is greater than the threshold value, a stale-miss occurs and the current value is fetched. This allows the state of the object to be changed to shared state. Figure 4 shows the state transition diagram for object entries in directories. The gray text on transitions indicates how the set of machines locally accessing the object is maintained. Since the global copies in DSM are always current, the object is always considered to be in shared state. The set of machines having copies of the object (stale or current) is maintained appropriately during the incoming read, write, and evict requests. Each write request leads to invalidation messages to respective set of machines.

Termination Semantics. Since stale objects will often be used during computations, termination of iterative algorithms needs to be handled carefully. Along with the algorithm specific termination conditions, additional checks must be performed to make sure that the final values are not computed using stale values, i.e., all the required updates are visible to all machines. This is achieved by making sure that there are no outstanding invalidate or refresh messages and that all the objects used for computation in last iteration were current objects.

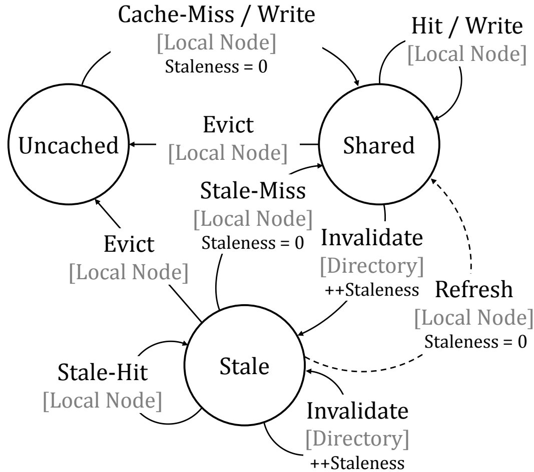


Figure 3: State transition diagram for cache entries. The operations are shown in black on the transition and the source of those operations are shown in gray.

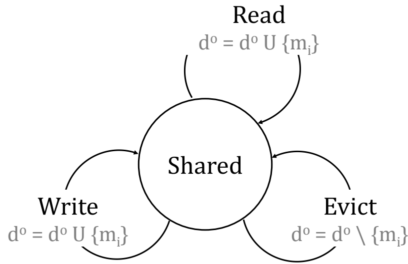


Figure 4: State transition diagram for directory entries. The protocol messages are shown in black on the transitions and the operations to maintain the set of machines currently accessing the object are shown in gray.

4.3 Optimizations

The above protocol satisfies the relaxed object consistency model proposed in Section 3. To further enhance the overall performance of the system, we perform the following standard optimizations over the protocol.

Work Collocation. Since the single writer model requires a unique node to perform all computation that updates a given object throughout the execution, the home node can itself be used as the single writer. In other words, computation for objects can be assigned to the nodes that maintain the global copy of those objects (similar to locality in [15]). This eliminates the write latency of remote objects and reduces the protocol traffic as write messages are converted into local signals on the home machine.

Message Aggregation. When multiple messages are sent to the same destination node, these messages can be aggregated into fewer number of messages (similar to bulk transfer in [51]). Message aggregation can significantly reduce the latencies incurred by transfer of multiple small messages since individually sending small messages is significantly slower than sending fewer number of large messages.

Message Flattening. When same requests are made for the same destination node, messages can be flattened to remove redundant requests and send minimal messages to the destination node. Message flattening becomes useful when multiple computations depend on a remote high degree vertex (flattening read and refresh messages) or when same objects are updated multiple times (flattening invalidation messages).

Object replication. If the nature of computation is known prior to execution, the objects required for computations on a given node can be replicated in the machine caches during startup to reduce remote access requests for warming up the caches in the the first iteration.

5. Experimental Setup

In this section, we discuss few details of our system, the benchmark programs and the inputs used to evaluate our relaxed consistency protocol.

5.1 System Prototype

Next we describe our prototype implementation including the design of the DSM, runtime, and the cluster it runs on.

DSM. To avoid the complexities introduced by false sharing and coherence granularity, we built an object based DSM in C++ similar to *dyDSM* [33]. The objects are distributed across the cluster such that there is exactly one global copy of each object in the DSM. METIS [30] is used to partition graphs across various machines to minimize edge-cuts. The relaxed consistency protocol was implemented in the DSM to relax the strict consistency and leverage stale values. Each node maintains a directory which is populated during initialization based on the objects placed on that node. Also, each node maintains a fixed size local cache for faster object access. The size of the local cache is large enough to hold objects required for computations on the local machine. However, we do not replicate all objects in these caches during initialization and allow them to fill up only with objects that are needed on that node.

The protocol is implemented via a set of threads on each node where each thread is responsible for a separate function. These threads primarily communicate through the read, write, evict, and invalidation messages and perform required operations to maintain the directory and cache metadata. The protocol messages are communicated using MPI send/recv commands. Message aggregation is used to combine multiple messages for the same destination in order to reduce communication latencies.

The heart of our protocol is the way it satisfies the *Best Effort Refresh* constraint enforced by our model. A separate *refresher thread* is responsible for updating the objects that will be required in the near future. The refresher thread blocks on a *refresh-queue* which maintains *object-ids* of objects that need to be refreshed. It picks up the object-id from this refresh-queue and issues a fetch from dyDSM. After receiving the current object, the refresher thread updates the stale copy in local cache with its current value.

Runtime. To evaluate the effectiveness of our protocol, we have developed a runtime that facilitates writing of parallel algorithms that run using the DSM. Each node has a single computation thread which repetitively executes the DO-WORK method as presented in Algorithm 1. The workload is distributed across all the nodes during initialization and later, the computation thread works on the same workload in every iteration. Hence, this implementation satisfies the single writer model. Since the computation thread is expected to block when the required protocol read, write, and evict requests are sent to the home machine, sending of these messages is taken care by the computation thread itself.

The computation thread is responsible for communicating the need to refresh objects which it will require in near future. On a stale-hit, the computation thread uses the locally available stale object. However, before using this stale object, it enqueues the object-id in the refresh-queue. Since the refresh thread will update the object with its current value in the local cache, the next request for this object by the computation thread will reflect its refreshed value, allowing computations to observe updated values. Staleness of objects used for computation is checked to ensure that termination is done correctly. Also, message queues are checked to be empty to make sure that there are no outstanding messages.

To transfer objects to and from the DSM, the runtime provides DSM-Fetch and DSM-Store methods. This hides the internal details of the protocol and allows parallel algorithms to directly execute over the DSM using these two methods.

System. We evaluate our protocol on *Tardis* which is a commercial 16-node cluster, running CentOS 6.3, Kernel v2.6.32-279. Each node has 64 GB memory and is connected to a Mellanox 18 port InfiniBand switch.

5.2 Benchmarks and Inputs

We use a wide range of modern applications (as listed in Table 1) and evaluate their asynchronous implementations. These applications are based on vertex centric model where each vertex iteratively computes a value (e.g., colors for GC, ranks for PR, and shortest paths for SSSP) and the algorithm stops when these vertex values become stable. We obtained these algorithms from various sources, implemented them in C++, and inserted the DSM fetch and store calls. Because of their vertex centric nature, they follow the same template as shown in Algorithm 1. These applications belong to important domains (e.g., scientific simulation and social network analysis) and are divided into following three categories.

Application	Type
Heat Simulation (HS) Wave Simulation (WS)	Partial Differential Equations (PDEs)
Graph Coloring (GC) Connected Components (CC) Community Detection (CD) [45] Number of Paths (NP)	Graph Mining
PageRank (PR) [53] Single Source Shortest Path (SSSP)	Graph Analytics

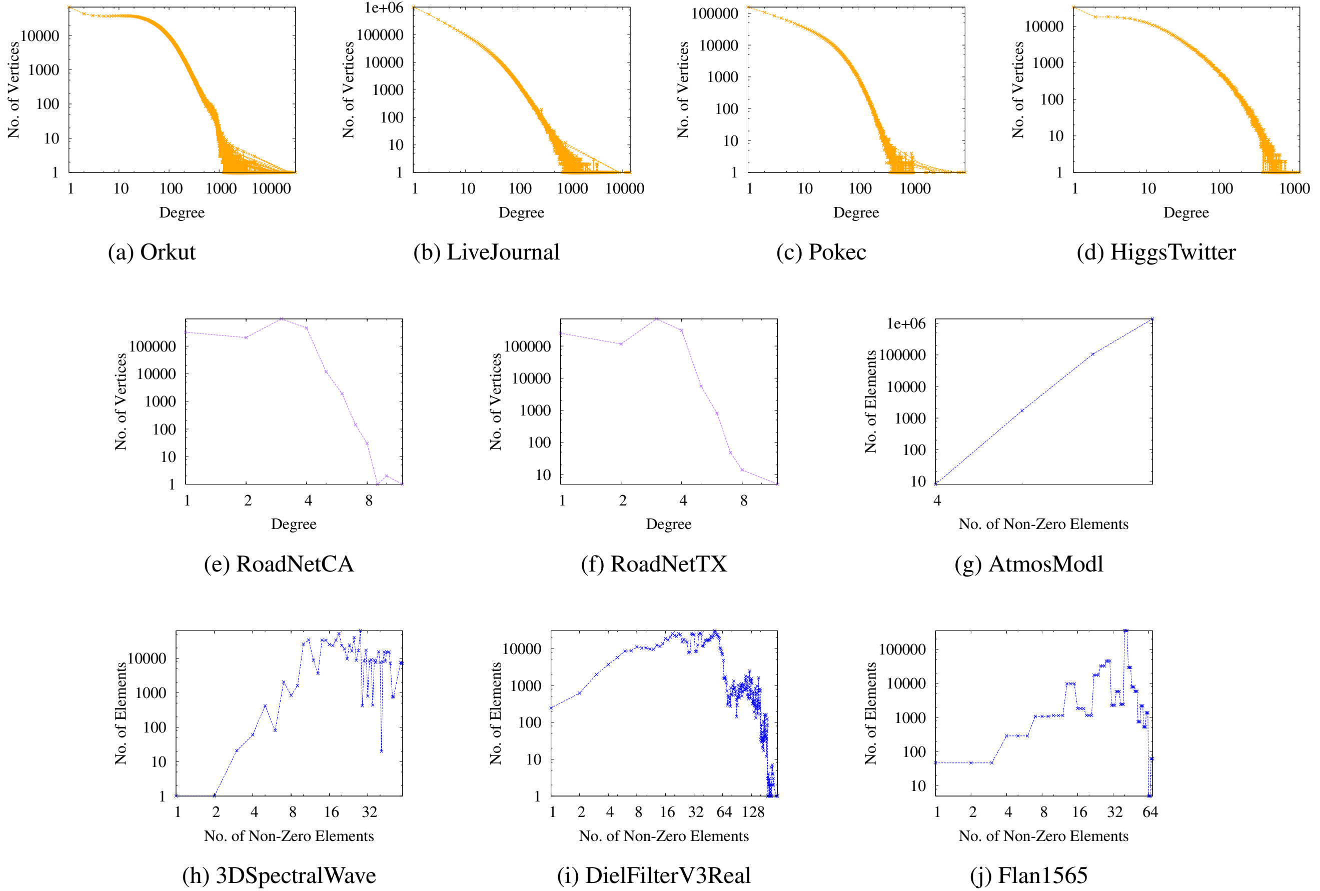
Table 1: Convergence based Iterative Algorithms.

Graph	Edges	Vertices
Orkut [52]	234,370,166	3,072,441
LiveJournal [52]	68,993,773	4,847,571
Pokec [54]	30,622,564	1,632,803
HiggsTwitter [25]	14,855,875	456,631
RoadNetCA [41]	5,533,214	1,971,281
RoadNetTX [41]	3,843,320	1,379,917
AtmosModl [4]	10,319,760	1,489,752
3DSpectralWave [62]	30,290,827	680,943
DielFilterV3Real [16]	89,306,020	1,102,824
Flan1565 [18]	114,165,372	1,564,794

Table 2: Real-world graphs & matrices used in experiments.

(i) Partial Differential Equations. The algorithms for solving partial differential equations (PDEs) are convergence based iterative algorithms which makes them suitable for asynchronous parallelism. We implemented two benchmarks which solve specific PDEs, namely, heat simulation (HS) and wave simulation (WS). Both the benchmarks iteratively determine the value of current cell based on the values of neighboring cells. The algorithms converge when all the cell values stabilize based on a pre-specified tolerance.

(ii) Graph Mining. These set of applications analyze various structural properties of graphs. We implemented four applications in this category: Graph Coloring (GC), Connected Components (CC), Community Detection (CD), and Number of Paths (NP). CC and CD are based on iterative label propagation [69]. The vertices are assigned labels which are initialized during setup. In every iteration, the label of a vertex is chosen based on its current label and the labels of its neighboring vertices. For CC and CD, the vertices start with unique labels; however, subsequent iterations in CC choose the minimum label whereas those in CD choose the most frequent labels [45]. For GC, the vertices are initialized with an invalid color and in subsequent iterations, a vertex is as-



(a) Orkut (b) LiveJournal (c) Pokec (d) HiggsTwitter

(e) RoadNetCA (f) RoadNetTX (g) AtmosModl

(h) 3DSpectralWave (i) DielFilterV3Real (j) Flan1565

Figure 5: Real-world graph datasets used for evaluation of graph mining and analytics benchmarks and sparse matrices used for PDE benchmarks. Orkut, LiveJournal, Pokec and HiggsTwitter are graphs from popular social networking and blogging platforms which represent power-law distribution. RoadNetCA and RoadNetTX are road networks for California and Texas states which exhibit a non-skewed distribution. AtmosModl, 3DSpectralWave, DielFilterV and Flan1565 are sparse matrices based on atmospheric model, 3D spectral elastic wave modeling, high-order vector finite element method in EM and 3D steel flange model. The x-axis indicates the number of elements in a row (or column) which are non-zero.

signed a unique color which is not assigned to any of its neighbors. If two neighboring vertices are assigned the same color, one of the vertex (chosen arbitrarily but fixed) is assigned a new unique color. For NP, all vertices except the source vertex have number of paths initialized to 0 and for source vertex, it is initialized to 1. In subsequent iterations, vertices calculate the number of paths by adding those of their neighbors.

(iii) Graph Analytics. These applications model their problems as a graph and are targeted to address specific queries which are not dependent on the structure of graphs alone. PageRank (PR) and Single Source Shortest Path (SSSP) fall in this category. For SSSP, all vertices except the source vertex have their distance initialized to ∞ and for source vertex, it is initialized to 0. In subsequent iterations, distance to a vertex is calculated by summing up the distances of all neighboring vertices and the weights on corresponding edges connecting those vertices and then, choosing the minimum sum. PR is a popular algorithm which iteratively computes the rank of a page based on the ranks of its neighbors [53].

Input Data Sets. We ran the benchmarks on publicly available [61, 63] real-world graphs and matrices listed in Table 2. The graph inputs are used to evaluate the Graph Mining and Analytics benchmarks, whereas the matrices are used for PDEs.

The graphs cover a broad range of sizes and sparsity (as shown in Fig. 5) and come from different real-world origins. *Orkut*, *LiveJournal* and *Pokec* are directed social networks which represent friendship among the users. *HiggsTwitter* is a social relationship graph among twitter users involved in tweeting about the discovery of Higgs particle. *RoadNetCA* and *RoadNetTX* are the California and Texas road networks respectively, in which the roads are represented by edges and

		Orkut	LiveJournal	Pokec	HiggsTwitter	RoadNetCA	RoadNetTX
CD	**SCP**	1,530.79	1,141.16	572.24	70.70	2.64	1.10
	RCP	974.92	307.95	238.43	36.46	1.95	1.17
CC	**SCP**	1,846.50	1,045.28	316.16	261.08	77.85	62.16
	RCP	710.13	316.46	154.70	78.11	69.10	66.26
GC	**SCP**	1,568.78	629.59	228.89	72.87	0.53	0.56
	RCP	733.03	254.53	101.09	35.48	0.95	0.64
NP	**SCP**	182.36	141.67	81.02	31.87	139.30	174.23
	RCP	124.67	117.64	39.97	12.20	140.70	179.28
PR	**SCP**	4,191.12	3,754.83	1,767.52	602.56	12.06	7.19
	RCP	2,710.07	2,047.06	275.29	88.85	11.82	8.39
SSSP	**SCP**	1,735.11	759.69	248.48	49.96	74.32	71.92
	RCP	714.98	317.62	118.72	39.98	72.30	73.28

Table 3: Execution times (in sec) of SCP and RCP for various graph mining and analytics benchmarks on a 16-node cluster.

the vertices represent the intersections. *AtmosModl*, *3DSpectralWave*, *DielFilterV3Real* and *Flan1565* are sparse matrices (as shown in Fig. 5) which represent models from various domains like atmospheric models, electromagnetics, hexahedral finite elements, and 3D consolidation problem. The graph inputs are used to evaluate the Graph Mining and Analytics benchmarks, whereas the matrices are used for PDEs.

6. Experimental Results

Now we present a detailed evaluation of our system including comparison with closely related protocols and systems.

6.1 Benefits of Exploiting Staleness

To study the benefit of using stale values we compare the performance of the following two protocols:

- **RCP**: This is the *Relaxed Consistency Protocol* developed in this paper. The threshold is set to a very high number [1] (=100) and refresher threads are used; and
- **SCP**: This is the *Strict Consistency Protocol* that does not allow the use of stale values at all and is based upon the traditional directory-based write through cache coherence strategy. This is used as the baseline, to evaluate the above protocols that allow the use of stale values.

In order to better understand the effectiveness of our protocol, we do not use the *object replication* optimization during this evaluation.

Across inputs. Table 3 and Table 4 compare the execution times (in sec) for *SCP* and *RCP* on a 16-node cluster. On an average, *RCP* achieves 4.6x speedup over *SCP* for PDE benchmarks and 2.04x speedup for graph mining and analytics benchmarks. The speedups vary across different benchmark and input combinations: for example, speedups for PR

[1] Since our RCP protocol does a good job in satisfying the Best Effort Refresh constraint, the need of using low threshold values is eliminated. Through experiments, we found that thresholds above 4 do not show any difference mainly because the refresher threads quickly eliminate objects with higher staleness values.

		AtmosModl	3DSpectralWave	DielFilterV3Real	Flan1565
HS	**SCP**	110.11	464.68	75.12	180.38
	RCP	23.99	40.00	29.27	93.79
WS	**SCP**	61.45	237.76	106.14	218.55
	RCP	14.51	28.87	48.00	149.18

Table 4: Execution times (in sec) of SCP and RCP for PDE benchmarks on a 16-node cluster.

across different inputs range from 1.02x to 6.8x whereas for HS, they vary from 1.9x to 11.6x. Note that *RCP* and *SCP* give similar performance for *RoadNetCA* and *RoadNetTX*. This is because these graphs are sparse and do not have a skewed degree distribution (Figure 5e and Figure 5f); hence, partitioning them over the DSM leads to very few edge cuts. Thus, *SCP* does not suffer much from remote fetch latencies that are tolerated by *RCP*.

Across configurations. The speedups achieved by *RCP* over *SCP*, on clusters of different sizes, are shown in Figure 6. These speedups are based upon the *Pokec* graph input for graph analytics and mining algorithms and the *AtmosModl* matrix input for PDEs. On average, *RCP* achieves 1.6x speedup on 2 nodes, 1.9x speedup on 4 nodes, 2.6x on 8 nodes and 3.3x on 16 nodes. Since *RCP* mainly focuses on reducing remote fetches by using locally available stale values, speedups achieved by *RCP* increase as the cluster grows. This is also the reason for achieving no performance benefits for few benchmarks (e.g., CD, NP) when the cluster is small; the overheads of the system mask the little benefits achieved by the protocol.

6.2 Bounded Staleness vs. RCP

The *delta coherence* [11] protocol supports *bounded staleness* by allowing use of objects which are no more than x versions out-of-date. In other words, it allows the use of stale values by statically using x as the staleness bound, but

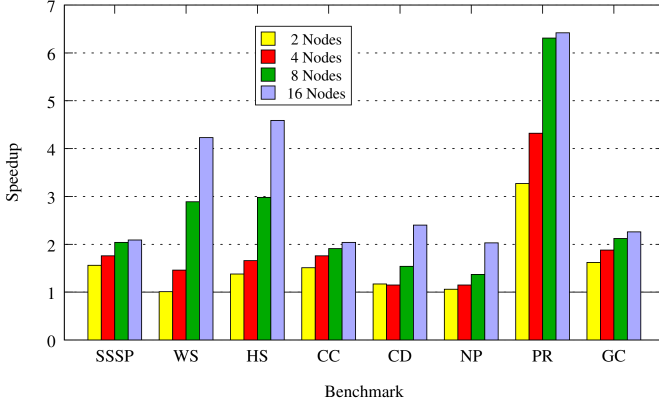


Figure 6: Speedups achieved by RCP over SCP on clusters of 2, 4, 8, and 16 nodes.

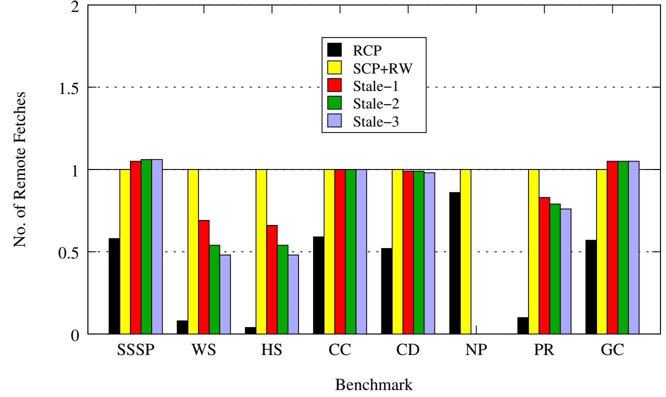


Figure 8: Number of remote fetches that stall computation threads normalized wrt SCP+RW.

it does not use refresher threads. In this section we demonstrate that via the use of stale values delta coherence can tolerate remote access latency, but the use of stale values slows down the algorithm's convergence. In the remainder of this section, **Stale-n** refers to the delta coherence protocol with a staleness bound of n. In order to separate out the benefits achieved from latency tolerating property of *RCP*, we relax the writes in *SCP* similar to that in *RCP*. We denote *SCP* with relaxed writes as **SCP+RW**. Writes in *Stale-n* are also similarly relaxed. The detailed results presented are based upon the *Pokec* graph input for graph analytics and mining algorithms and the *AtmosModl* matrix input for PDEs.

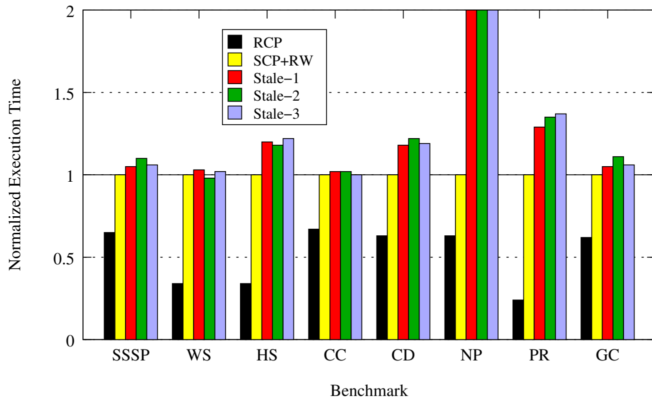


Figure 7: Execution times of RCP and Stale-n ($n = 1, 2, 3$) on a 16-node cluster normalized wrt SCP+RW.

The execution times of *RCP* and *Stale-n* ($n = 1, 2, 3$), normalized with respect to the *SCP+RW*, are shown in Figure 7. *RCP* consistently achieves better performance than all other protocols considered. On an average, *RCP* executions times are lower than *SCP+RW* by 48.7%. The performance of *Stale-n* varies across different thresholds because when the threshold is increased, the convergence gets adversely affected (e.g., PR, SSSP). No staleness value ($n = 1, 2, 3$) consistently performs the best for *Stale-n*. On an average, the best performing *Stale-n*, which sometimes performs better than *SCP+RW* (e.g., WS), increases execution time over *SCP+RW* by 10.5% while the worst performing *Stale-n* always performs worse than *SCP+RW*. On an average, *RCP* gives 56% reduction in execution time over the best *Stale-n*.

To further analyze the relative performances of *RCP* and *Stale-n*, we present additional data, once again normalized with respect to *SCP+RW*. Let us consider the fraction of remote fetches that fall on the critical path of execution, i.e. they cause the computation thread to stall while waiting for the remote fetch to complete. From the results shown in Figure 8 we determine, that on an average, computation thread under *RCP* blocks for only 41.83% of remote fetches which are due to compulsory cache misses. In contrast, the best *Stale-n* causes stalls on 85.6% of remote fetches. Since remote fetches are long latency operations, we expect *RCP* to perform better than both *SCP+RW* and best *Stale-n*.

Figure 9 shows the distribution of staleness of values used. We observe that in *RCP* the staleness of values is typically 0 or 1 – in fact on an average 97.4% of values have staleness of 0 and 2.2% of values have staleness of 1. *Stale-2* and *Stale-3* use slightly more stale values in CC, CD and PR. It is interesting to see that in GC, *RCP* uses more stale values than *Stale-2* and *Stale-3*; this is mainly because *Stale-2* and *Stale-3* use stale color values to quickly stabilize and hence, color values do not change much. However, in *RCP*, as color values are received through refresh, the stabilized colors do get changed, in turn leading to more changes and hence, stale values. Using stale values slows down the convergence of these algorithms which can be seen from the data in Figure 10. On an average, *RCP* requires 49.5% more iterations than *SCP+RW* while *Stale-2* and *Stale-3* require 146.4% and 176.2% more iterations than *SCP+RW*. Note that *Stale-n* versions for NP did not terminate within 5 times the time required when run with *SCP+RW*; hence, we do not show the data related to remote fetches, iterations, and staleness values for these cases.

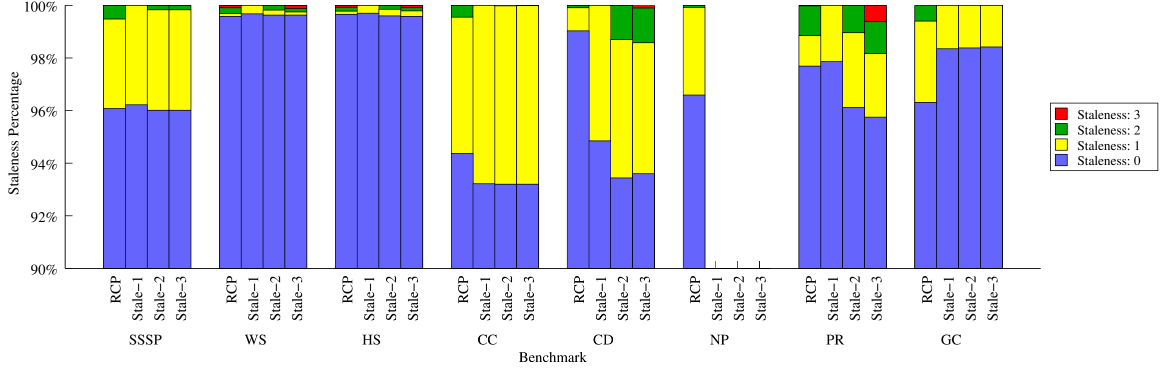


Figure 9: Percentage of objects used with staleness n.

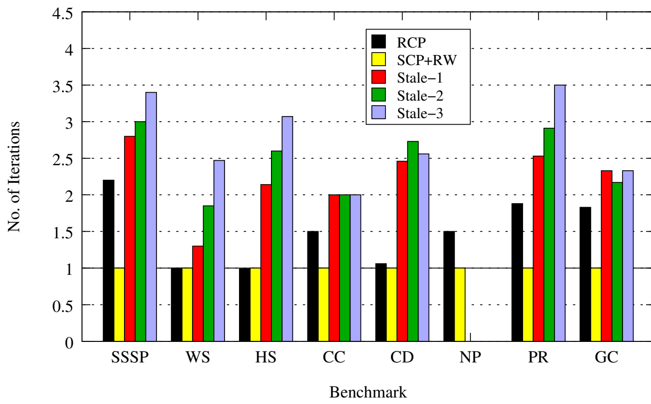


Figure 10: Number of iterations performed before converging normalized wrt SCP+RW.

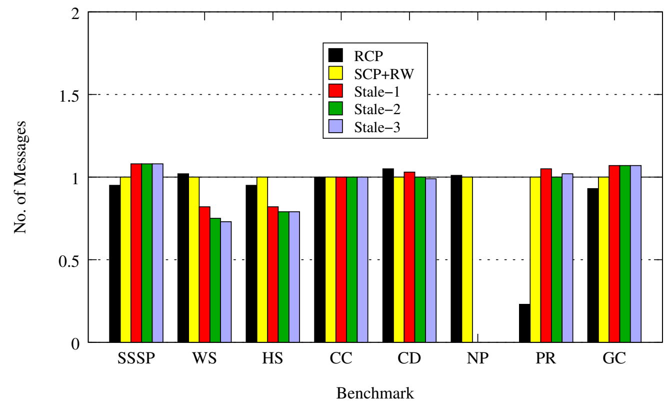


Figure 11: Number of protocol messages normalized wrt SCP+RW.

It is interesting to note that even though *Stale-n* effectively tries to avoid remote fetches on the critical path, it is often done at the cost of delaying convergence. This delay in convergence, in turn, results in more remote fetches. Thus, the overall performance can be adversely affected (e.g., HS, PR). On the other hand, even though *Stale-n* sometimes converges in nearly same number of iterations (e.g., WS), the overall performance gains are less when compared to benefits achieved from *RCP*. This is mainly because the computation thread often blocks when the staleness of local objects crosses beyond the threshold n and hence the reduction in remote fetches is not as significant as in *RCP*. This interplay between reduction in remote fetches and increase in the number of iterations for convergence makes the best choice of n to be specific for each benchmark. This can be clearly observed by comparing WS and PR: *Stale-2* performs better than *Stale-1* for WS whereas the latter performs better than the former for PR. Hence, in *Stale-n*, selecting the value of n is benchmark specific and hard to do. *RCP* releases users from such concerns and outperforms *Stale-n* in each case.

Finally, we examine the communication overhead of *RCP* and *Stale-n* w.r.t. *SCP+RW* by comparing the number of protocol messages required in each case. As shown in Figure 11, in most cases, *RCP* requires fewer protocol messages compared to *SCP+RW* and in the remaining cases, it requires less than 5% additional messages. This is mainly because there is a drastic reduction in the number of remote fetch requests for *RCP* (as seen in Figure 8), most of which are reduced to asynchronous refresh requests. PR using *RCP* requires only 22.8% messages of that required by *SCP+RW* because invalidate and refresh messages are far fewer than the reduction in the remote fetch requests. Even though WS and HS also experience a similarly large reduction in remote fetch requests using *RCP*, they require sufficiently more invalidate and refresh messages which leads to their overall communication costs to be nearly same as *SCP+RW*.

6.3 Design Choices of RCP

To better understand our design choices for the relaxed consistency protocol, we evaluate the protocols using synthetic benchmarks that exploit different application specific properties. The synthetic benchmarks are designed similar to other benchmarks which mainly fetch neighboring vertices and compute new vertex values. The data is based upon the

HiggsTwitter graph input and the programs were run for a pre-configured number of iterations.

Piggy-backing Updates vs. RCP Invalidates. Figure 12 shows the effect of allowing multiple writes on *RCP*, *SCP* and *SCP* with piggy-backed updates (**SCP+PB**) where the updates are sent to remote nodes along with invalidation messages. The execution times are normalized with respect to configurations where an object is written once per iteration. We observe that even though *SCP+PB* performs better than *SCP*, the execution times for both the configurations increase drastically when objects are written more often in an iteration. It is interesting to note that the benefits achieved from *SCP+PB* over *SCP* reduce as the number of times objects are written per iteration increases; this happens because redundant updates are sent by *SCP+PB* which becomes costly. On the other hand, *RCP* easily tolerates multiple writes and consistently performs similar to the baseline. The resulting execution times normalized with respect to *SCP* are shown in Figure 13.

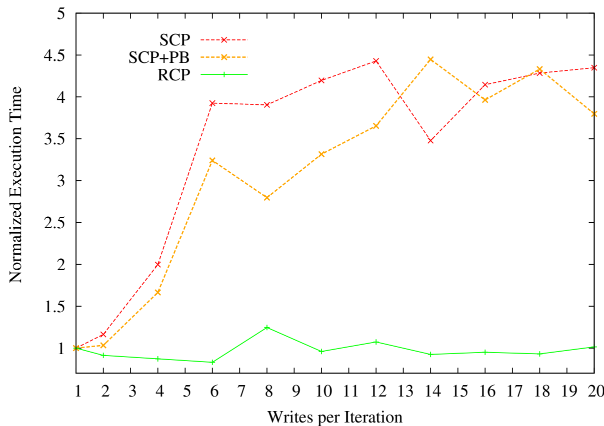


Figure 12: Execution times of SCP with and without piggy-backed updates and RCP for different write configurations normalized wrt single write versions.

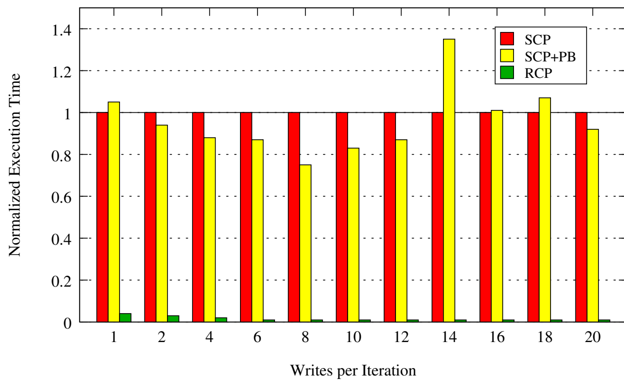


Figure 13: Execution times of SCP with and without piggy-backed updates and RCP for different write configurations normalized wrt SCP.

Sensitivity of RCP to Object Sizes. In Figure 14, we compare the performance of *SCP*, *SCP+PB*, and *RCP* for different object sizes. Object sizes are varied by adding a bloat to the base object which, by itself, only consists of a double value (8 bytes). We can see that sending updates with invalidates performs similar to when only invalidates are sent and *RCP* consistently performs better than the other 2 configurations.

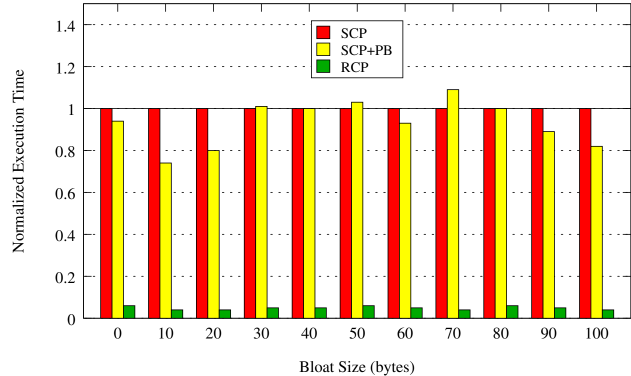


Figure 14: Execution times of SCP with and without piggy-backed updates and RCP for different object sizes.

Sensitivity of RCP to Communication Delay. In Figure 15 we show the impact of fetch latency on the maximum staleness of object values used for computations. We observe that as the fetch latencies are increased, the maximum staleness varies. As expected, we notice that most of the objects have low staleness values, leaving very few objects which are very stale. Hence, it is important to control the staleness using an upper bound which avoids potential staleness runaways in such cases.

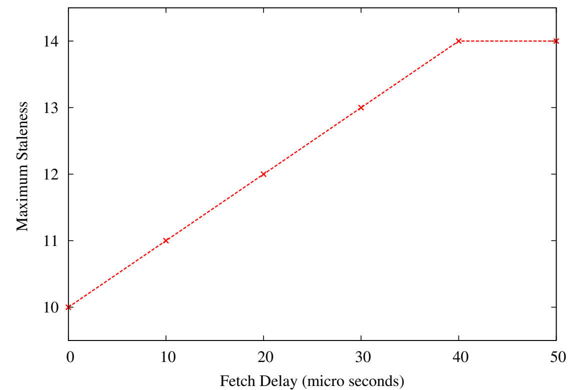


Figure 15: Maximum staleness for objects used in RCP with varying communication delay.

6.4 Comparison with Other Systems

To exhibit the effectiveness of exploiting asynchrony, we compare the performance of asynchronous algorithms running using *RCP* with the performance of the popular *bulk*

		SSSP	PR	GC	CC	NP
Orkut	**RCP**	161.88	822.95	92.79	90.31	2.35
	GraphLab	239.4	829.3	248.66	102.02	140.5
LiveJournal	**RCP**	21.73	343.96	17.44	22.09	133.43
	GraphLab	15.7	295.1	X	66.99	150.2
Pokec	**RCP**	9.47	169.47	8.81	7.1	1.74
	GraphLab	8.7	159.9	173.47	40.52	76.4
HiggsTwitter	**RCP**	2.5	15.64	3.59	4.1	0.48
	GraphLab	5.5	X	263.45	16.21	32.5
RoadNetCA	**RCP**	49.47	7.70	0.93	56.96	16.51
	GraphLab	60.3	88.4	50.22	220.9	37.4
RoadNetTX	**RCP**	44.21	5.05	0.53	50.94	15.83
	GraphLab	18.3	78	60.76	115.6	X

Table 5: Execution times (in sec) of SSSP, PR, GC, CC, and NP using RCP and GraphLab (GL) on a 16-node cluster. An x indicates that execution did not complete either because it crashed or continued for over 60 minutes.

synchronous parallel (BSP) model as it is supported by existing graph processing frameworks. In addition, we also compare the performance of our system with GraphLab [46], a popular distributed graph processing framework.

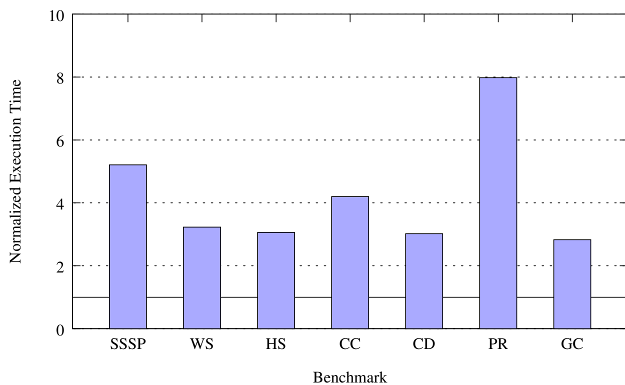


Figure 16: Execution times for BSP based implementations normalized wrt their asynchronous versions that use RCP.

RCP vs. Bulk Synchronous Parallel. The Bulk Synchronous Parallel (BSP) is a parallel computation model that was proposed to efficiently parallelize applications on a set of processors [64]. Algorithms based on the BSP model perform a series of supersteps where each superstep is composed of the following three phases: *Computation* - multiple processes and threads concurrently execute several computations using locally available data values; *Communication* - the computed data values are made available to required processes and threads; and *Synchronization* - a barrier is executed by all the processes to conclude the superstep.

Recent advances in developing generic frameworks for parallel applications heavily rely on this model. However, by maintaining a separate communication phase, the updated values are forcibly propagated throughout the system, which unnecessarily introduces significant overhead for asynchronous algorithms.

Figure 16 shows the execution times of *BSP* based implementations for our benchmarks normalized with respect to their corresponding asynchronous versions that make use of our proposed protocol. On an average, our asynchronous algorithms are faster than *BSP* based algorithms by 4.2x. Apart from PR, *BSP* versions take 2.8x to 5.2x more than their asynchronous versions using our protocol. PR takes 7.9x more time with *BSP* mainly because it spends more time in the communication phase compared to other benchmarks. Again, *BSP* version for NP did not terminate within 10 times the time required when run with *RCP* and hence, we do not show its performance.

Comparison with GraphLab. GraphLab [46] is a popular graph processing framework which is closest to our work because it provides shared memory abstractions to program over a distributed environment. We compare the performance of GraphLab with *RCP* using five benchmarks – SSSP, PR, GC, CC, and NP – four of which are provided in the graph analytics toolkit distributed with the GraphLab software. Since GraphLab provides both synchronous and asynchronous versions of some of these programs, we report the best times obtained here. In order to have a fair comparison, similar to replication of boundary vertices in GraphLab, caches were pre-populated with replicas of boundary vertices to eliminate latencies incurred by cache warmups.

In Table 5 we report the absolute execution times (in sec) for SSSP, PR, GC, CC, and NP using RCP and GraphLab for the four power law graph inputs, as GraphLab has been designed to efficiently handle such graphs. The relative performance of GraphLab and *RCP* varies significantly across inputs and benchmarks. We observe that for the Orkut and HiggsTwitter inputs *RCP* consistently outperforms GraphLab. For the LiveJournal and Pokec inputs, GraphLab provides

superior performance for SSSP and PR. Finally, for GC and CC benchmarks *RCP* consistently outperforms GraphLab across different inputs. Overall, the performance of *RCP* compares favorably with GraphLab for power law graphs. It should be noted that *RCP* is based on the the Relaxed Consistency Model which is orthogonal to GraphLab's consistency models. Hence, this model can also be incorporated in GraphLab.

Although GraphLab has been designed primarily for power law graphs, we did test it for other inputs. As shown in Table 5, on the RoadNetTX graph the above benchmarks took 0.5 sec to 50.9 sec using *RCP* and 18.3 sec to 115.6 sec on GraphLab. We also coded other benchmarks on GraphLab and compared their performance for different inputs. For both NP and WS, *RCP* consistently outperformed GraphLab.

7. Related Work

Weak Memory Models. Extensive research has been conducted on weak memory models that relax consistency in various ways. A hierarchy of these models can be found in [49] and [23]. We have already discussed the relevant ones; here we provide a complete characterization of the models.

A number of models are too strong for asynchronous algorithms. *Release consistency* [19] and its variants like *lazy release consistency* [31, 67] relax consistency by delaying visibility of updates until certain *specially labeled* accesses. *Causal memory* [2] is weaker than Lamport's *sequential consistency* [38] which guarantees that processes agree on the relative ordering of operations that are *potentially causally related* [39]. *Entry consistency* [7] guarantees consistency only when a thread enters a critical section defined by synchronization variables. *Scope consistency* [27] enforces that all updates in the previous consistency session are visible at the start of current consistency session for the same scope. Even though entry consistency and scope consistency can be used to mimic relaxed coherence, by manually controlling synchronization variables and consistency scopes, none of these models inherently relax the consistency.

Other models are too weak and hence not a good fit for asynchronous algorithms. *Pipelined RAM (PRAM)* [43] provides fast data access similar to our proposed model: on a read, it simply returns the local copy and on write, local values are updated and the new value is broadcast to other processors. However, it allows inconsistent data-views because relative order of updates from different processes can vary. Also, it enforces a strict constraint that all processors must agree on the order of all observed writes by a single processor. This means, broadcast of these updates cannot be skipped, and hence, as shown in [24], the time taken for flow of updates in PRAM increases rapidly as the number of processes increase. This increase in flow of updates can delay the convergence of asynchronous algorithms. Our pull-based model allows object values to be skipped since it does not constrain the order of observed writes to different objects and hence, is more flexible to provide faster convergence. *Slow memory* [26] makes the consistency model weak enough for a read to return some previously written value. This allows the cache to be non-coherent, but requires extra programming effort to guarantee convergence.

Mermera [22] tries to solve the same problem for iterative asynchronous algorithms by using slow memory. Programs written on mermera handle the correctness and convergence issue by explicitly maintaining a mix of slow writes, coherent writes, and specialized barriers (to flush slow writes). Also, slow memory is based on delayed updates; if it is implemented to support a DSM which is not update based, once the stream of delayed updates enters the memory, local copies will be invalidated and the same issue (waiting for remote fetch) arises. To enable ease of programming and allow intuitive reasoning about execution, our consistency model guarantees the same correctness semantics as the traditional cache consistency model and, at the same time, allows relaxation of consistency for use of stale objects.

Finally, a number of models support bounded staleness. This is same as *delta coherence* as used in *InterWeave* [11], that allows use of objects that are no more than x versions out-of-date. Even though this mechanism proved to be useful to reduce network usage, maintaining a static staleness upper bound x is not useful; a low value of x will only hide few remote fetches because stale objects will quickly become useless while a high value of x can significantly delay the convergence as updates are slowly propagated through the system, allowing many wasteful computations. This issue is also faced by the *stale synchronous parallel* (SSP) model [13]. SSP defines staleness as the number of iterations since the object at hand received its value. Their experiments show that the convergence behavior begins to degrade when the staleness bound is increased past a certain value. Hence, statically bounding staleness is not the correct approach to improve performance of asynchronous iterative algorithms. The challenge is to allow use of stale objects when up-to-date values are not available but, at the same time, minimize the staleness of such objects. *Delta consistency* introduced in [60] has a similar approach as delta coherence, but enforces a temporal bound on staleness. This requires mechanisms for temporal synchronization to compute the *global virtual time* (GVT). Again, since there is no global ordering for writes from different processors to the same location, correctness semantics need to be externally ensured. Also, none of these models proactively try to maintain low staleness by fetching and updating values. This means, fetches on critical paths are blocked often because the values become too stale which limits their performance benefits.

DSM Coherence Frameworks. Many coherence frameworks, for page as well as object based systems, support multiple coherence schemes to effectively deal with variety of behaviors. *Shasta* [56] is a page based DSM that provides flexibility of varying coherence granularity for shared data structures. *CASHMERe* [34] provides a scalable shared memory which uses page sized coherence blocks. It uses an asynchronous protocol but, the focus is not towards relaxation of coherence. *TreadMarks* [32] is designed to reduce communication for maintaining memory consistency. It uses lazy release consistency and multiple writer based protocols that provide strict consistency guarantee and incurs less communication. In [3] dynamic adaption between single writer and multiple writer protocols is proposed to balance false sharing with computation and memory costs. This work is tangential to our goal which is improving performance of asynchronous iterative algorithms by allowing controlled use of stale objects.

We deal at a higher abstraction level by using object based DSM like *Orca* [5] and *Munin* [8]. Munin uses coherence mechanisms based upon object types. Also, relaxation of coherence in Munin is limited in between synchronization points and at them the delayed updates are flushed to remote copies. *Object View* [42] shares similar goals as Munin and Orca; it provides extensions to Java to specify intended use of objects by computation threads. This allows runtime to use low-overhead caching protocols customized to application requirements. *Problem Oriented Object Memory* [35] allows relaxation of strict consistency by letting objects to fall in different consistency models. Since we aim to specifically improve performance of asynchronous algorithms, we do not distinguish objects based on usage types. *Cachet* [58] dynamically adapts across multiple micro-protocols that are optimized based upon access patterns. [9] focuses on reducing communication required to maintain consistency among distributed memories. [28, 57, 68] and others try to relax consistency using basic memory models previously described. Since they inherently aim to provide a consistent DSM, relaxation of consistency is not explored in these works.

Graph Processing Frameworks. Recently, there have been many advances in developing frameworks for distributed graph processing. Google's *Pregel* [47] provides a synchronous vertex centric framework for large scale graph processing which uses message passing instead of a shared memory abstraction. *GraphLab* [46] provides a framework for asynchronous execution of machine learning and data mining algorithms on graphs. It allows users to choose among three different data consistency constraints to balance program correctness and performance. *PowerGraph* [20] provides efficient distributed graph placement and computation by exploiting the structure of power-law graphs. It provides both, synchronous and asynchronous execution and enforces serializability by avoiding adjacent vertex programs from running concurrently. *GraphChi* [37] provides efficient disk-based graph processing on a single machine for input graphs that cannot fit in memory.

Ligra [59] presents a simple shared memory abstraction for vertex algorithms which is particularly good for problems similar to graph traversal. [50] presents a shared-memory based implementations of these DSLs on a generalized *Galois* [36] system and compares its performance with the original implementations. These frameworks are based on the Bulk Synchronous Parallel (BSP) [64] model and the MapReduce [15] philosophy which allow users to write code from a local perspective and let the system translate the computations to larger datasets. However, they do not provide support for programming asynchronous algorithms by using stale values. The ideas we presented in this paper can be incorporated in above frameworks to support asynchronous algorithms. *GRACE* [66], a shared memory based graph processing system, uses message passing and provides asynchronous execution by using stale messages. Since shared-memory processing does not suffer from communication latencies, these systems can perform well for graphs which can fit on a single multicore server. Another programming model that has been used in recent works is one that supports thread level speculation [55]. This model also does not permit the use of stale values; in fact the use of a stale value when detected causes misspeculation. In Section 6.4, we showed that asynchronous algorithms outperform their BSP counterparts.

8. Conclusion

We presented an effective solution for exploiting the asynchronous nature of iterative algorithms for tolerating communication latency in a DSM based cluster. We designed a relaxed object consistency model and the RCP protocol. This protocol tracks staleness of objects, allows threads to utilize stale values up to a given threshold, and incorporates a policy for refreshing stale values. Together, these features allow an asynchronous algorithm to tolerate communication latency without adversely impacting algorithm's convergence. We demonstrate that for a wide range of asynchronous graph algorithms, on an average, our approach outperforms: prior relaxed memory models that allow stale values by at least 2.27x; and BSP model by 4.2x. We also show that our approach performs quite well in comparison to *GraphLab*.

Acknowledgments

This research was supported by a Google Research Award and NSF grant CCF-1318103 to the University of California, Riverside.

References

[1] Apache Giraph. http://giraph.apache.org/.

[2] M. Ahamad, G. Neiger, J.E. Burns, P. Kohli, and P.W. Hutto. Causal Memory: Definitions, Implementation and Programming. *Distributed Computing*, 9(1):37–49, 1995.

[3] C. Amza, A.L. Cox, W. Zwaenepoel, and S. Dwarkadas. Software DSM Protocols That Adapt Between Single Writer and Multiple Writer. *HPCA*, pages 261–271, 1997.

[4] A. Bourchtein. Atmospheric models. *http://www.cise.ufl.edu/research/sparse/matrices/Bourchtein/atmosmodl.html*, 2009.

[5] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE TSE*, 18(3):190–205, 1992.

[6] G.M. Baudet. Asynchronous Iterative Methods for Multiprocessors. *JACM*, 25(2):226–244, 1978.

[7] B.N. Bershad and M.J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. *TR*, CMU-CS-91-170, 1991.

[8] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. *SOSP*, pages 152–164, 1991.

[9] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-related Communication in Distributed Shared-memory Systems. *ACM TOCS*, 13(3):205–243, 1995.

[10] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *Computer*, 23(6):49–58, 1990.

[11] D. Chen, C. Tang, B. Sanders, S. Dwarkadas, and M.L. Scott. Exploiting High-level Coherence Information to Optimize Distributed Shared State. *PPoPP*, pages 131–142, 2003.

[12] Y-S. Cheng, M. Neely, and K. M. Chugg. Iterative Message Passing Algorithm for Bipartite Maximum Weighted Matching. In *IEEE International Symposium on Information Theory*, pages 1934–1938. 2006.

[13] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing. Solving the Straggler Problem with Bounded Staleness. *HotOS*, pages 22–22, 2013.

[14] W. L. M. D. Chazan. Chaotic relaxation. In *Linear Algebra and Its Application*, pages 2:199–222, 1969.

[15] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *CACM*, 51(1):107–113, 2008.

[16] A. Dziekonski, A. Lamecki, and M. Mrozowski. High-order vector finite element method in EM. *http://www.cise.ufl.edu/research/sparse/matrices/Dziekonski/dielFilterV3real.html*, 2011.

[17] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software Behavior Oriented Parallelization. *PLDI*, pages 223–234, 2007.

[18] C. Janna, and M. Ferronato. 3D model of a steel flange, hexahedral finite elements. *http://www.cise.ufl.edu/research/sparse/matrices/Janna/Flan_1565.html*, 2011.

[19] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. *ISCA*, pages 15–26, 1990.

[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. *OSDI*, pages 17–30, 2012.

[21] J. R. Goodman. *Cache Consistency and Sequential Consistency*. Univ. of Wisconsin-Madison, CS Department, 1991.

[22] A. Heddaya and H. Sinha. An overview of Mermera: A system and formalism for non-coherent distributed parallel memory. *Hawaii International Conf. on System Sciences*, vol. 2, pages 164–173, 1993.

[23] A. Heddaya and H. Sinha. *Coherence, Non-coherence and Local Consistency in Distributed Shared Memory for Parallel Computing*. TR BU-CS-92-004, Boston Univ., 1992.

[24] A. Heddaya and H. Sinha. *An Implementation of Mermera: A Shared Memory System that Mixes Coherence with Non-coherence*. TR BUCS-TR-1993-006, Boston Univ., 1993.

[25] M. De Domenico, A. Lima, P. Mougel, and M. Musolesi. The Anatomy of a Scientific Rumor. *Scientific Reports*, 2013.

[26] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. *ICDCS*, pages 302–309, 1990.

[27] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge Between Release Consistency and Entry Consistency. *SPAA*, pages 277–287, 1996.

[28] V. Iosevich and A. Schuster. Distributed Shared Memory: To Relax or Not to Relax? In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *Euro-Par*, *LNCS 3149*, pages 198–205, Springer, 2004.

[29] U. Kang, D. Horng, et al. Inference of Beliefs on Billion-Scale Graphs. *KDD-LDMTA*, 2010.

[30] G. Karypis, V. Kumar. A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, Vol. 20, pp. 359–392, 1999.

[31] P. Keleher, A.L. Cox, and W. Zwaenepoel. *Lazy Release Consistency for Software Distributed Shared Memory*, *ISCA*, pages 13–21, 1992.

[32] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *WTEC*. pages 10–10, 1994.

[33] SC. Koduru, M. Feng, and R. Gupta. Programming Large Dynamic Data Structures on a DSM Cluster of Multicores. *PGAS Programming Models*, 2013.

[34] L. Kontothanassis, R. Stets, G. Hunt, U. Rencuzogullari, G. Altekar, S. Dwarkadas, and M.L. Scott. Shared Memory Computing on Clusters with Symmetric Multiprocessors and System Area Networks. *TOCS*, 23(3):301–335, 2005.

[35] A. Kristensen and C. Low. Problem-oriented Object Memory: Customizing Consistency. *OOPSLA*, pages 399–413, 1995.

[36] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L.P. Chew. Optimistic Parallelism Requires Abstractions. *PLDI*, pages 211–222, 2007.

[37] A. Kyrola, G. Blelloch, C. Guestrin GraphChi: Large-scale Graph Computation on Just a PC. *OSDI*, pages 31–46, 2012.

[38] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE TC*, C-28 (9):690–691, 1979.

[39] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7):558–565, 1978.

[40] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. *The Directory-based Cache Coherence Protocol for the DASH Multiprocessor*, *ISCA*, pages 148–159, 1990.

[41] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. Internet Mathematics, pages 29–123, 2009.

[42] I. Lipkind, I. Pechtchanski, and V. Karamcheti. Object Views: Language Support for Intelligent Object Caching in Parallel and Distributed Computations. *OOPSLA*, pages 447–460, 1999.

[43] R. Lipton and J. Sandberg. *PRAM: A Scalable Shared Memory*. Princeton University, Department of Computer Science, TR-180-88, 1988.

[44] L. Liu and Z. Li. Improving Parallelism and Locality with Asynchronous Algorithms. *PPoPP*, pages 213–222, 2010.

[45] X. Liu and T. Murata. Advanced modularity-specialized label propagation algorithm for detecting communities in networks. *Physica A: Statistical Mechanics and its Applications*, 389(7): 1493–1500, 2010.

[46] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[47] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. *SIGMOD*, pages 135–146, 2010.

[48] R. Meyers and Z. Li. ASYNC Loop Constructs for Relaxed Synchronization. In *LCPC*, *Languages and Compilers for Parallel Computing*, pages 292–303, 2008.

[49] D. Mosberger. Memory Consistency Models. *SIGOPS Oper. Syst. Rev.*, 27(1):18–26, 1993.

[50] D. Nguyen, L. Andrew and K. Pingali. A Lightweight Infrastructure for Graph Analytics. *SOSP*, 2013.

[51] W-Y. Liang, C-T. King, and F. Lai. Adsmith: An Efficient Object-Based Distributed Shared Memory System on PVM. *International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 173-179, 1996.

[52] J. Yang and J. Leskovec. Defining and Evaluating Network Communities based on Ground-truth. ICDM, 2012.

[53] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. 1999.

[54] L. Takac and M. Zabovsky. Data analysis in public social networks. International Scientific Conference and International Workshop Present Day Trends of Innovations, 2012.

[55] L. Rauchwerger and D. A. Padua. The LRPD Test: Speculative Run-time Parallelization of Loops with Privatization and Reduction Parallelization. PLDI, 218–232, 1995.

[56] D. J. Scales and K. Gharachorloo. Design and Performance of the Shasta Distributed Shared Memory Protocol. *ICS*, pages 245–252, 1997.

[57] M. Schulz, J. Tao, and W. Karl. Improving the Scalability of Shared Memory Systems through Relaxed Consistency. *WC3*, 2002.

[58] X. Shen, Arvind, and L. Rudolph. CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-memory Systems. *ICS*, pages 135–144, 1999.

[59] J. Shun, and G. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *PPoPP*, pages 135–146, 2013.

[60] A. Singla, U. Ramachandran, and J. Hodgins. Temporal Notions of Synchronization and Consistency in Beehive. *SPAA*, pages 211–220, 1997.

[61] J. Leskovec. Stanford Large Network Dataset Collection. *http://snap.stanford.edu/data/index.html*, 2011.

[62] C. Sinclair. 3-D spectral-element elastic wave modeling in freq. domain. *http://www.cise.ufl.edu/research/sparse/matrices/Sinclair/3Dspectralwave.html*, 2007.

[63] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, Vol 38, pages 1:1 - 1:25, 2011.

[64] L. G. Valiant. A Bridging Model for Parallel Computation. *CACM*, 33(8):103–111, 1990.

[65] J. Leskovec, L. A. Adamic, and B. A. Huberman. The Dynamics of Viral Marketing. *ACM Trans. Web*, 2007.

[66] G. Wang, W. Xie, A. Demers, and J. Gehrke. Asynchronous Large-Scale Graph Processing Made Easy. *CIDR*, 2013.

[67] B.-H. Yu, Z. Huang, S. Cranefield, and M. Purvis. Homeless and Home-based Lazy Release Consistency Protocols on Distributed Shared Memory. *Australasian Conf. on Computer Science-Vol 26*, pages 117–123, 2004.

[68] Y. Zhou, L. Iftode, J. P. Sing, K. Li, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. *PPoPP*, pages 193–205, 1997.

[69] X. Zhu and Z. Ghahramani. Learning from Labeled and Unlabeled Data with Label Propagation. Technical Report CMU-CALD-02-107, Carnegie Mellon University, 2002.