Oolong: Asynchronous Distributed Applications Made Easy *

Christopher Mitchell Russell Power Jinyang Li
New York University
{cmitchell, power, jinyang}@cs.nyu.edu

Abstract

We present Oolong, a distributed programming framework designed for sparse asynchronous applications such as distributed web crawling, shortest paths, and connected components. Oolong stores program state in distributed in-memory key-value tables on which user-defined triggers may be set. Triggers can be activated whenever a key-value pair is modified. The event-driven nature of triggers is particularly appropriate for asynchronous computation where workers can *independently* process part of the state towards convergence without any need for global synchronization. Using Oolong, we have implemented solutions for several large-scale asynchronous computation problems, achieving good performance and robust fault tolerance.

1 Introduction

Distributed computation has traditionally been a powerful yet complex method of solving large problems. As cloud computing services such as Amazon EC2 and Windows Azure have become prevalent and economical, more application programmers have sought to harness distributed computation to achieve scalable performance. Consequently, there has been a surge in demand for programming frameworks that can shield application programmers from the complexities of distribution and fault tolerance.

Most existing frameworks target synchronous computation that iteratively reads or updates a large fraction of a dataset with global synchronization between iterations. For example, MapReduce[3] and Dryad[5, 16] applications such as word-count and sorting stream an entire dataset for processing in a single round. Piccolo[14] and Pregel[9] rely on global barriers to execute applications such as PageRank and K-Means round-by-round. Asynchronous computation differs in that execution does not proceed in lockstep across rounds. Specifically, the result of past processing is immediately used to determine the course of current execution. By contrast, with synchronous computation, the result of past processing only affects computation in the next global round. Because they eschew global synchronization, asynchronous solutions are much more efficient for many problems. Unfortunately, there is no way to express asynchronous computation with frameworks such as MapReduce or Piccolo.

We propose a programming framework called Oolong to address the needs of asynchronous applications. In a typical asynchronous application, processing is only performed on the portion of state that has been modified, and as a result of such processing, additional state changes are generated which cause more work to be done. Thus,

*This material is based upon work supported by the National Science Foundation under Grant No. 1065169 and by a Google research grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
APSys '12, July 23-24, 2012, Seoul, S. Korea
Copyright 2012 ACM 978-1-4503-1669-9/12/07... $15.00

incremental execution in asynchronous computation can be viewed as *reacting* to some state modification. Oolong stores program state in distributed in-memory tables [14] and allows such event-driven program structure to be expressed naturally in terms of *triggers*, user-specified code blocks that can be invoked whenever the associated table entry has been modified. Oolong triggers perform asynchronous computation by processing requested state modifications and scheduling additional state changes.

Frameworks exposing global state employ periodic checkpointing to recover from untimely node failure. Checkpointing is expensive, so a long period is used, meaning that significant progress is lost if a node fails. Oolong provides *continuous checkpointing* which stores the delta of in-memory tables and trigger state with low overhead so that little progress is lost upon recovery.

Our evaluation demonstrates that Oolong yields 2x to 13x speedups on Single-Source Shortest Paths and connected components compared with the synchronous Piccolo framework.

2 Oolong Design

Oolong targets a popular subclass of distributed computation with the following properties:

In-memory Distributed State The input and intermediate state of the computation fit in the aggregate memory of all the nodes.

Asynchronous The computation iteratively processes data with little or no global synchronization needed.

Sparse Execution The computation accesses a small fraction of its dataset at a time to progress towards convergence, rather than repeatedly sweeping across all input data.

Many computation problems have asynchronous solutions with the above characteristics. For example, a distributed web crawler stores the crawling state for each URL it has encountered in memory and each node requests for those web pages that have not yet been crawled without any global synchronization among nodes. Similarly, many graph problems such as shortest paths and connected components also have asynchronous solutions.

Design Overview Oolong stores the intermediate state of an application in key-value tables distributed across the memory of all participating machines, like Piccolo[14]. Oolong lets programmers specify two sections of code attached to tables, *accumulators* and *triggers*. Like Piccolo, accumulators in Oolong combine updates to each key-value pair; they can neither perform I/O nor access global state. While Piccolo accumulators have no return value, Oolong accumulators can return true to indicate that a trigger should be scheduled for execution on that key. Unlike accumulators, triggers are executed asynchronously by each node after the updates have been applied. Furthermore, triggers can perform arbitrary computation such as doing I/O, reading or updating different key-value pairs.

2.1 Programming API

An Oolong application consists of three parts: 1) a section of code for declaring key-value tables and launching application kernels. 2) application kernels that run on many nodes to perform parallel processing. 3) accumulators and triggers that are associated with tables and respond to table updates.

In Figure 1, we illustrate Oolong's programming API using an example application, Single-Source Shortest Paths (SSSP). SSSP calculates the shortest distance from a single source vertex to all other vertices in a directed graph.

`SSSP_Main` in Figure 1 first creates two tables, `dists` and `graph`, each of which is divided into 1000 shards and distributed among all workers. The main function then initializes all vertices' distances from the source vertex to infinity and associates `SSSP_Trigger` with the `dists` table. The computation begins when the main function updates the source vertex's distance to 0. This update is incorporated into the corresponding `dists` table entry using the user-defined accumulator function, `SSSP_Accumulator`, which chooses the minimum of the current and updated distance. The update also causes the corresponding trigger associated with the vertex to be scheduled for execution sometime in the future. The trigger, `SSSP_Trigger`, simply updates each of its corresponding vertex's outgoing neighbors with a new

```
def SSSP_Init_Kernel(dists):
  for k in my_partitions(dists):
    dists.put(key,infinity)

def SSSP_Accumulator(dist, new_dist):
  if new_dist < dist:
    dist = new_dist
    # Activate trigger
    return true
  else
    # Do not activate trigger
    return false

def SSSP_Trigger(nodeID):
  d = dists.get(nodeID)
  for t in graph.get(nodeID).outgoing:
    dists.update(t,1+d)

# Main function
def SSSP_Main(src):
  dists = Table(key=int, value=double, \\
    shards=1000, accum=SSSP_Accumulator)
  graph = Table(key=int, value=Node, shards
  =1000)

  # Init all nodes' dists to inf
  Run(SSSP_Init_Kernel, args=dists)
  # Enable triggers
  AssociateTrigger(dists, SSSP_Trigger)

  # Activate triggers by updating src's dist
  dists.update(src, 0)
```

Figure 1: Single-Source Shortest Paths in Oolong. (Pseudocode uses Python-like syntax.)

potentially-shorter distance from the source. The accumulators for those vertices will further activate their triggers if their distances from the source become shorter.

As long as new shortest distances are found and triggers are scheduled and run, the `Run` function will block. The master periodically asks workers how many updates they have applied and how many triggers are scheduled; when no new updates have been applied since the last check and no triggers are scheduled, no further processing is possible and the problem has converged. Once the `Run` function returns, the computation has converged and the `dists` table contains the final shortest path distance from the source vertex to all vertices.

2.2 System Design

Oolong runs with one master node and many worker nodes. The master is responsible for assigning table shards to workers and scheduling kernel execution. The workers store table shards, execute kernels and triggers, and respond to data read/write requests from peers. A diagrammatic explanation of the inter- and intra-worker communication is shown in Figure 2.

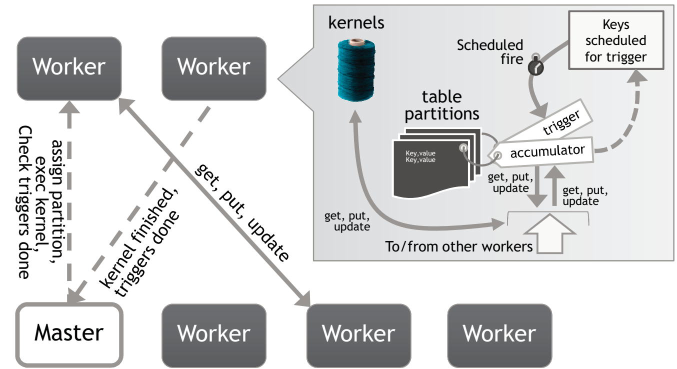


Figure 2: Workers, master, and messages in Oolong; the tables and triggers inside a worker. Accumulators can cause asynchronous triggers to be scheduled; triggers can update and read keys and cause additional triggers to be scheduled.

Trigger execution and convergence Each worker employs multiple threads to execute activated triggers concurrently since any individual trigger might be blocked doing I/O or reading remote table entries. We represent the current set of scheduled triggers using a bitmap. The "dirty" bit corresponding to a table entry is set by the accumulator if its return value is true. Each worker periodically scans the bitmap to execute scheduled triggers. The "dirty" bit for a key is cleared when a trigger is run for that key. While a trigger is being executed, the corresponding table entry can be updated, which may schedule another trigger for that key.

Failure recovery Oolong can checkpoint application state to local disk or other fault-tolerant storage to recover from worker crashes. Workers in Oolong can record full checkpoints containing the entire current application state including a copy of the trigger bitmap, and *delta checkpoints* that contain only the updates applied to tables since the last full checkpoint. All checkpointing is coordinated by the master and done using the Chandy-Lamport snapshot algorithm. Oolong saves full checkpoints periodically and performs continuous

delta checkpointing in between full checkpoints. If a worker crashes, all workers restart from the most recent full checkpoint, then replay deltas and continue the application. Full checkpoints are fast to restore, but are more expensive to save than delta checkpoints. By combining the two, we achieve a good balance of checkpoint overhead, restore time, and checkpoint size.

3 Evaluation

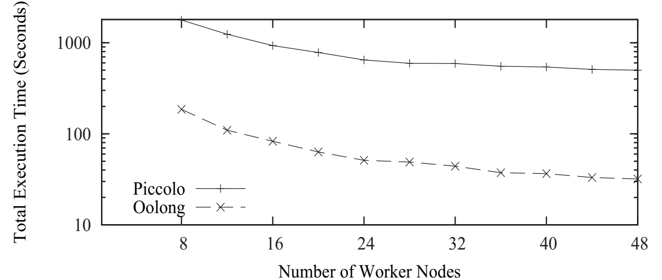
Oolong was implemented in C++, building on the codebase of the Piccolo project [14]. It utilizes the OpenMPI distributed message-passing system, and employs Google's protobuf library for serialization. APIs in C++ and Python are provided to Oolong application programmers.

We have implemented several asynchronous applications including SSSP, connected components, and PageRank. We compare the performance of these applications against their synchronous counterparts running on top of Piccolo. The experiments were run on a local cluster of 7 machines of which six are dual processor nodes (2 quad-core 2.27GHz Xeon, 16GB RAM) and one is a single processor node (1 quad-core Xeon, 8GB RAM). We ran the master on the single processor node and ran one to eight workers on each dual processor node.

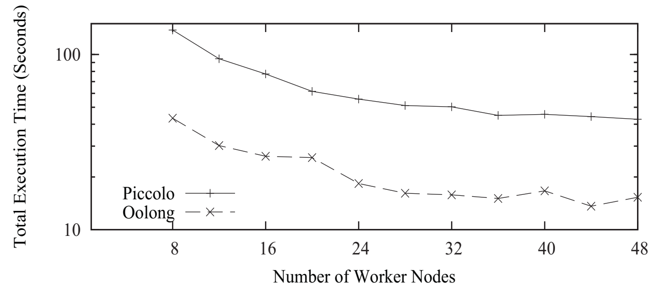
3.1 Performance

SSSP Performance We evaluated both asynchronous SSSP (on Oolong) and its synchronous version (on Piccolo) using a synthetically generated random graph of 100M vertices. Figure 3(a) shows the SSSP execution time as more workers are added. Both asynchronous and synchronous SSSP programs display almost ideal scalability as additional workers are added, achieving a 5.81x speedup when moving from 8 to 48 workers. SSSP under Oolong outperforms the synchronous version, achieving an average speedup of 13.0x over Piccolo on the same number of workers.

The speedup achieved by Oolong 's SSSP implementation can be attributed to two factors: 1) Asynchronous SSSP on Oolong eliminates unnecessary work in repeating computations or scanning the entire graph for not-yet-converged vertices. 2) With no global iteration, asynchronous SSSP saves the overhead of setting up and tearing down more kernels repeatedly. In particular, we note that the last three iterations of the synchronous SSSP needed to process fewer than 100 vertices in a 10M-vertex graph, a waste avoided by adopting an asynchronous framework.



(a) Shortest paths (100M-vertex graph)



(b) Connected components (LiveJournal graph)

Figure 3: Performance and scalability for Piccolo and Oolong .

Connected Components Performance The connected components problem involves finding the vertices in a graph that are connected only to each other and not to any other vertices. We generated graphs using the same methodology as the PrIter project [18], and also tested real-world graphs from SNAP [2, 6]. Figure 3(b) shows the scalability and performance of Piccolo and Oolong finding connected components in a 4.8M vertex (138M directed-edge) graph of LiveJournal relationships. Piccolo 's synchronous implementation converged in six iterations on 48 workers in 42.7 seconds. The same graph was processed by Oolong in 15.31 seconds, a 2.8x speedup. The PrIter framework has a reported convergence time of ~130 seconds on 8 commodity CPU cores for a much smaller graph of 400K vertices [18], a graph that Oolong was able to process in 5.3 seconds on 8 cores. We note that this comparison is approxi-

mate since PrIter has been evaluated on a cluster with a different hardware configuration.

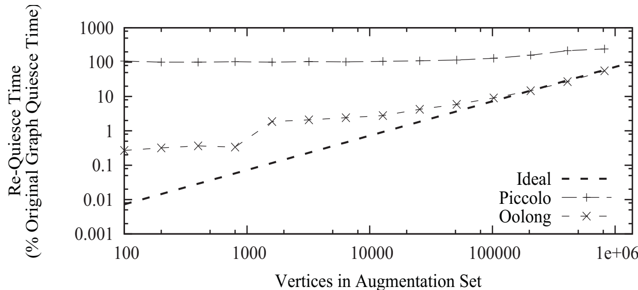


Figure 4: Time to re-quiesce PageRank by adding new vertices to an already-quiesced 1M-vertex graph. The ideal re-quiesce time is computed as the fraction of new vertices to the number of original vertices.

Incremental PageRank Computation The PageRank algorithm assigns scores to vertices in a web graph based on the link structure of the graph. Our asynchronous implementation processes PageRank updates for each vertex and sends additional updates to the vertex's neighbors if the change in its PageRank score exceeds a preconfigured threshold. Unlike the SSSP or connected components problem, asynchronous PageRank converges to similar but not identical final scores as the synchronous solution. The asynchronous implementation approximates it well, however; the Kendall-Tau distance between the asynchronous and synchronous PageRank solutions is 0.994. Oolong's PageRank achieves a 1.7x speedup over Piccolo on 48 workers. More significantly, it is easy to extend asynchronous PageRank to perform incremental computation: we simply continue the asynchronous execution to re-quiesce a modified graph. Figure 4 shows the Piccolo and Oolong runtimes to re-quiesce a 1M-vertex graph as a function of the number of new vertices added to the graph. Piccolo requires as many additional iterations to re-quiesce the graph as for the initial run, hence its additional execution time remains roughly invariant regardless of how much the graph is changed. By contrast, Oolong 's incremental computation time scales with the size of the change.

3.2 Complexity and Development

The primary benefit of Oolong over synchronous frameworks such as Piccolo is that problems naturally-suited to an asynchronous solution may be more cleanly expressed than when forced into a synchronous design. We found that this benefit translated to conceptually-simple code for many Oolong applications.

From an application writer's perspective, our trigger design offers expressiveness and flexibility, but also introduces two sources of complexity. First, there is a risk of infinite trigger loops; programmers must carefully design asynchronous computation to ensure eventual convergence. Second, it is sometimes difficult to debug a trigger-based asynchronous computation. For example, because asynchronous application flow is non-deterministic, multiple runs may display inconsistent failure symptoms. Despite above complexity, our experience with Oolong has been positive. For example, the development time for the Oolong SSSP was only 3 hours; it was written by a coder with no background in writing Oolong programs, given only the interface documentation and a brief explanation of the system.

4 Related Work

Most existing distributed programming frameworks focus on synchronous computation and rely on global barriers between iterations to converge a computation. Examples include MapReduce [3], Dryad [5], Spark [17], Piccolo [14], Pregel [9], Ciel [11], TransMR [15], and PrIter [18]. Piccolo is optimized for computation whose intermediate state fit in the aggregate memory of the cluster. Oolong extends Piccolo to provide support for asynchronous computation. Our distributed key-value store resembles the tuple space concept in Linda [1]. However, tuple spaces are not designed for high-frequency access, and lack the primitives for triggered computation.

Database systems provide user-specified code triggers which performs computation in response to updates in a single transaction [10]. The trigger primitive in Oolong is inspired by database triggers. Unlike databases, which rely on transactions, Oolong achieves fault tolerance with low overhead by performing continuous checkpointing in the background.

Recent distributed systems work has also addressed the importance of active (event-triggered)

procedures in distributed storage. Comet proposes Active Storage Objects that execute event-triggered handler procedures on key-level granularity, but are intended for storage rather than computation systems [4]. Google's Percolator system maintains "observers" on a per-column basis that can trigger arbitrary code when any row is modified [13]. Like databases, fault tolerance in Percolator is achieved by using transactions. Network Datalog uses a set of trigger-esque constructs to provide correctness and execute integrity rules for network protocols [12, 7].

GraphLab [8] offers asynchronous features for graph-based problems, particularly those within Machine Learning. It focuses on providing sequential consistency for such computation with centralized scheduling and explicitly-expressed data dependency. Oolong's accumulator offers a cheaper way to cope with concurrent updates compared with enforcing sequential consistency.

5 Conclusion

Many problems are more efficiently solved with asynchronous distributed computation than synchronous execution. Existing distributed frameworks are poorly suited to running asynchronous applications without global barriers. Oolong provides user-specified triggers, run on table updates, that balance independent progress towards a problem solution across a large number of workers with globally-visible state. Effective computation time on large clusters of failure-prone hardware is maximized with a continuous checkpointing scheme for failure recovery.

References

[1] Ahuja, S., Curriero, N., and Gelernter, D. Linda and friends. *Computer 19*, 8 (1986), 26–34.

[2] Backstrom, L., Huttenlocher, D., Kleinberg, J., and Lan, X. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (New York, NY, USA, 2006), KDD '06, ACM, pp. 44–54.

[3] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51* (January 2008).

[4] Geambasu, R., Levy, A. A., Kohno, T., Krishnamurthy, A., and Levy, H. M. Comet: An Active Distributed Key-Value Store. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (2010).

[5] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev. 41* (March 2007), 59–72.

[6] Leskovec, J., Adamic, L. A., and Huberman, B. A. The dynamics of viral marketing. *ACM Trans. Web 1*, 1 (May 2007).

[7] Loo, B. T., Hellerstein, J. M., Stoica, I., and Ramakrishnan, R. Declarative routing: extensible routing with declarative queries. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2005), SIGCOMM '05, ACM, pp. 289–300.

[8] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. Graphlab: A new framework for parallel machine learning. *CoRR abs/1006.4990* (2010).

[9] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 International Conference on Management of Data* (2010).

[10] McCarthy, D., and Dayal, U. The Architecture of an Active Data Base Management System. In *Proceedings of the 1989 ACM Sigmod International Conference on Management of Data* (1989).

[11] Murray, D. G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., and Hand, S. Ciel: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation* (2011), NSDI'11.

[12] Nigam, V., Ja, L., Loo, B. T., and Scedrov, A. Maintaining distributed logic programs incrementally. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming* (2011).

[13] Peng, D., and Dabek, F. Large-Scale Incremental Processing Using Distributed Transactions and Notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (2010).

[14] Power, R., and Li, J. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (2010).

[15] Rapolu, N., Kambatla, K., Jagannathan, S., and Grama, A. TransMR: Data-Centric Programming Beyond Data Parallelism. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing* (2011).

[16] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, l., Gunda, P. K., and Currey, J. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI* (2008), R. Draves and R. van Renesse, Eds., USENIX Association, pp. 1–14.

[17] Zaharia, M., Chowdhury, N. M. M., Franklin, M., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. Tech. Rep. UCB/EECS-2010-53, EECS Department, University of California, Berkeley, May 2010.

[18] Zhang, Y., Gao, Q., Gao, L., and Wang, C. Priter: a distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (New York, NY, USA, 2011), SOCC '11, ACM, pp. 13:1–13:14.