Tim Kaler May 15, 2012

Masters of Engineering Thesis Proposal

1 Optimizing parallel iterative graph computation

I propose to develop a deterministic parallel framework for performing iterative computation on a graph which schedules work on vertices based upon a valid coloring. Preliminary work modifying Graphlab, a parallel framework for implementing iterative machine learning algorithms on data graphs, has demonstrated the merits of this approach by improving performance and eliminating non-determinism. Further, I've identified opportunities to modify the graph representation to improve data locality as well as improvements and theoretical analysis of a deterministic parallel coloring algorithm used to pre-compute a coloring of the input graph.

2 Graphlab

Graphlab is a parallel framework for performing iterative computation on a data graph to implement machine learning algorithms. [8]

I implemented a version of the Graphlab engine which uses the Cilk runtime system and reducer hyperobjects to provide serial semantics and avoid the explicit management of worker threads. I also implemented a dynamic task scheduler which provides exclusive access to vertex and edge data without the use of locks. Together, these modifications make Graphlab computations deterministic without sacrificing performance, and often result in sizable speedup for large enough benchmarks. Further, compatibility is maintained with nearly all pre-existing Graphlab programs.

2.1 The Graphlab Abstraction

The Graphlab abstraction models machine learning algorithms as iterative computation on a data graph. The abstraction was developed in response to the observation that other models like Map Reduce often do a poor job of parallelizing machine learning tasks which have sparse dependencies between variables, as these models treat all variables as independent. To represent such problems, Graphlab creates a data graph which encodes the dependencies between variables. The user then writes an update function which reads and modifies a small section of the graph, called a "scope." The user can then pick a scheduler from one of the

offered types, and allow the Graphlab engine to iteratively invoke their update function on scheduled vertices in parallel.

2.2 Scopes and Consistency Levels

In Graphlab an update function operates upon a *scope* which consists of a vertex, its adjacent edges, and neighbors. To prevent data races when overlapping scopes are updated in parallel, Graphlab provides four different consistency levels: full consistency, edge consistency, vertex consistency, and null consistency. A fully consistent scope acquires a read/write lock on all data within the scope, i.e. the vertex, its adjacent edges, and neighbors. An edge consistent scope acquires a read/write lock on the vertex and its adjacent edges, but only a read lock on neighboring vertices. A vertex consistent scope only acquires a read/write lock on the current vertex. A null consistent scope acquires no locks, and does not prevent data races.

When the appropriate consistency level is used, Graphlab guarantees sequential consistency. However, even when the highest consistency level is used many Graphlab programs are non-deterministic as there are races to acquire locks within scopes which can result in updates being applied in a different order on each execution of a program.

2.3 Static and Dynamic Schedulers

Graphlab's scheduler interface exposes the methods $get\_next\_task(int\ cpu\_id)$, which returns an update task for the given cpu to process. Graphlab provides two types of schedulers: static and dynamic. A static scheduler iterates over the all vertices until a termination condition is reached. A dynamic scheduler implements the $add\_task(update\_task\ task)$ method which is called from within the programmer's update function to schedule additional work.

Graphlab provides two static schedulers: round-robin, and chromatic. The round-robin scheduler simply iterates over all vertices in some order. The chromatic scheduler colors the graph, and then iterates over all vertices in order of color while ensuring that only vertices of the same color are updated in parallel.

For dynamic schedulers Graphlab provides a fifo, priority, sweep schedulers. The fifo scheduler and priority schedulers provide vertices in order of insertion and priority respectively, and have low parallelism. The sweep scheduler iteratively updates a set of vertices that were scheduled during the previous iteration. In addition, Graphlab provides multiqueue versions of the fifo and priority scheduler which maintains a separate fifo, and priority queue for each cpu. Graphlab also provides special purpose schedulers such as the splash and sampling schedulers which are similar to the priority scheduler, but specialized for specific applications.

3 Design of the Cilk Engine and Schedulers

I designed a new engine and scheduler interface for Graphlab utilizing the Cilk runtime system and Cilk hyperobjects.

Cilk is a linguistic extension to C++ which allows programmers to easily add fork-join parallelism to their programs through the use of the two keywords: *cilk_spawn*, and *cilk_sync*. The keyword *cilk_spawn* indicates that the proceeding function call may execute in parallel with its continuation, and the *cilk_sync* keyword acts as a barrier: preventing any thread from proceeding beyond that point until all execution strands are joined. Cilk uses a provably good work scheduler, and provides serial semantics in the absence of data races. A detailed description of the Cilk runtime system and reducer hyperobjects may be found in [3] and [5] respectively.

3.1 Cilk Schedulers

Schedulers in the Cilk version of Graphlab implement the *get_task_bag*() method, which returns a *bag* of update tasks which may be executed in parallel. A scheduler may maintain several different bags, and upon a call to *add_task* will insert the scheduled task into the appropriate one. For a detailed description of the Bag data structure and the theoretical properties of the Bag reducer hyperobject one may refer to Leiserson et al. [6].

3.2 Chromatic Sweep Scheduler

A sweep scheduler is a dynamic scheduler in which work is scheduled on vertices iteratively. Each iteration updates those vertices which were scheduled in the previous iteration, until an iteration occurs in which no vertices are scheduled.

The chromatic sweep scheduler provides the additional guarantee that it will only present mono-chromatic bags of vertices to the engine. If the graph is given a valid coloring, then no two neighboring vertices share the same color. In this case, the engine may provide edge consistency without the use of locks or other synchronization primitives.

The scheduler itself needs only a single atomic compare and swap operation to prevent the same update task from being scheduled twice. This may occur since two vertices who share a common neighbor may be updated in parallel, and the update of a vertex can cause an update task to be scheduled on a neighbor.

```
void add_task(update_task_type task) {
  if (__sync_bool_compare_and_swap(&vertex_task_added[task.vertex()],
          0, 1)) {
    // bags is an array of Bag_reducer<update_task_type>*
    bags[graph.color(task.vertex())]->insert(task);
  }
}
```

The chromatic sweep scheduler guarantees edge consistency provided the graph is given a valid coloring. Other degrees of consistency can be provided by giving the graph alternate colorings. For example: null consistency can be obtained by assigning all vertices the same color, and full consistency can be obtained by giving the graph a distance-2 coloring.

3.3 Cilk Engine

The run method of the Cilk engine iteratively processes the bags returned by the scheduler's *get_task_bag* method.

```
while (true) {
  // get a bag of the vertices that we can traverse in parallel.
  Bag<update_task_type>* task_bag = scheduler->get_task_bag();
  if (task_bag->numElements() > 0) {
    parallel_process(task_bag);
  } else {
    break;
  }
}
```

The *parallel_process* method is similar in structure to that used in the parallel breadth first search algorithm presented in Leiserson et al. [6] The update tasks are executed at the leaves of the invocation tree of *parallel_process* by creating a scope for the vertex, and then invoking the application specific update function on that scope.

3.4 Deterministic Parallel Coloring

By default, the chromatic sweep scheduler uses a simple deterministic parallel coloring algorithm. The method used is analogous to the prefix method described by Blelloch et al. to

parallelize the sequential maximal independent set algorithm [2]. Given a total order of the vertices, π, and the smallest uncolored vertex, v, a range of vertices $[v, v + P]$ is processed in parallel and each vertex which has no smaller uncolored neighbor is colored. Currently, a prefix size of $P = 512$ has been selected to maximize performance on a 12 core machines.

When vertices are given a random order, this algorithm has reasonable parallelism on average. Such a random order may be obtained efficiently by using an efficient deterministic parallel random number generator, such as that implemented by Leiserson et al. [7] The parallel coloring algorithm produces the same result as a greedy sequential coloring processing vertices in order. Therefore, ordering heuristics may be used which tend to reduce the number of colors needed by the algorithm. For example, ordering vertices in decreasing order of degree often reduces the number of colors used without significantly affecting the runtime of the parallel coloring algorithm. A mixed approach which orders vertices of the same degree randomly, appears to be an effective strategy.

On a 12 core machine, the parallel coloring algorithm is about 2-3 times faster than an optimized sequential coloring algorithm for random graphs, and as much as 5 times faster for bi-partite graphs. The theoretical properties of this coloring algorithm have not yet been fully investigated, and are the subject of future work for my thesis.

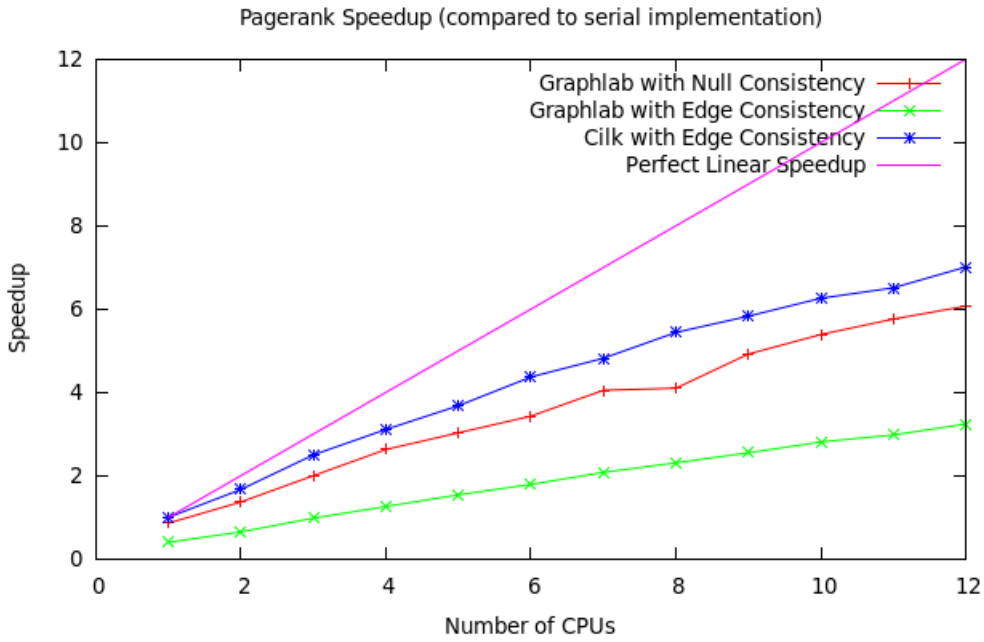
3.5 Performance Analysis

The Cilk implementation provides approximately 2x speedup over the original Graphlab on large enough benchmarks which require edge consistency to guarantee correctness. Further, the Cilk implementation is competetive in performance with the least consistent schedulers provided by Graphlab: matching its performance on large enough benchmarks. For small benchmarks (involving few iterations) , the additional overhead of coloring the graph can outweigh the speedup gained by reducing synchronnization overhead. Even in this case, however, the "cost" of determinism provided by the Cilk implementation is small: approximately 10% of the total time it takes the load the graph into memory.

3.5.1 Pagerank Benchmark

To evaluate the performance of my engine and scheduler implementation I created a graph of 1 million vertices and 10 million edges formed by picking edges from a power law distribution. I utilized a simple pagerank algorithm, based on the random surfer model, which was included as a Graphlab demo application. The update function for the pagerank algorithm reads data from incoming edges, and writes data to outgoing edges. Therefore, edge consistency is required to avoid data races.

I compared the speedup obtained over an efficient serial implementation when running the Graphlab and Cilk versions of Graphlab on multiple cores. The most speedup was obtained when using the Cilk implementation with edge consistency, even when compared to the Graphlab implementation providing no consistency guarantees.



3.5.2 Singular Value Decomposition Benchmark

I used the NPIC500 dataset [4] which was produced via a crawl of approximately 200 million webpages that counted the co-occurrence of noun-phrase, context pairs. The data set is pruned to remove all pairs whose co-occurrence count was less than 500.

On 12 cores the Graphlab implementation using edge consistency took 15.2 seconds, and 5.4 seconds when using null consistency. When using the Cilk engine and the chromatic sweep scheduler, which provides edge consistency, this benchmark finished in 5.7 seconds, including the 0.2 seconds spent coloring the input graph.

3.5.3 Conclusions

I have concluded that the Cilk engine and chromatic sweep scheduler is able to provide edge consistency without significant cost. The time to color the input graph is small, roughly 10% of the time it takes to load the graph. After this precomputation, the cilk engine

performs as well as, and sometimes better than, the original Graphlab engine providing the lowest consistency guarantees. Furthermore, the Cilk version of Graphlab guarantees that all programs which only require edge consistency to avoid data races will be deterministic.

4 Future Research

There are a number of avenues for research which can be incorporated into my thesis. I propose to pursue these, as well as others, as a part of the unified goal of writing an efficient parallel framework for iterative graph computation.

4.1 Cache oblivious graph layouts

There appear to be opportunities to improve spatial locality by laying out the graph efficiently in memory. Bender et al. describe an algorithm for finding a memory layout that allows for cache oblivious graph traversals [1]. Their methods apply most readily to constant dimensional meshes, but also extend to graphs with non-constant degree. Further investigation is required to understand how to apply these methods for the case in which only a subset of the graph's vertices are updated, and to gauge the performance of the algorithm on graphs of non-constant degree.

4.2 Improved Parallel Coloring

There are opportunities to improve my parallel coloring algorithm. My deterministic parallel coloring algorithm orders the vertices and then colors a vertex once it has no smaller uncolored neighbors. For parallelizing maximal independent set, Blelloch et al. propose an alternative to the prefix based method which could also be used to implement a faster algorithm. An associated dependency DAG can be formed in which vertex v depends on u if: v and u are neighbors, u is uncolored, and u is smaller than v. Then the set of vertices that can be colored in the current iteration are the roots of this DAG. This structure suggests an alternate implementation in which a bag of root vertices is maintained. One possible implementation would maintain an atomic counter for each vertex which is initialized to the number of its smaller neighbors. Then after coloring a vertex the counters of all larger neighbors would be decremented, and any uncolored neighbor with a counter equal to zero would be inserted into the bag. There also exist alternate approaches which avoid the use of atomic instructions, but sometimes put non-root vertices into the bag. These methods are required to perform an additional check before coloring, but may still perform

less overall work than the prefix method. These methods should be investigated as they may significantly outperform the prefix based method.

Furthermore, the theoretical properties of the parallel coloring algorithm currently being used have not yet been investigated. The methods of Blelloch et al. should be employed to provide theoretical bounds on the expected performance of the parallel coloring algorithm being used when the graph's vertices are given a random order.

4.3 Task order within bags and pre-fetching

Currently update tasks are processed in a non-determined order from the bag. The innermost loop of the parallel_process method iterates over a small block of update tasks. It would be advantageous if the vertex and edge data associated with the tasks in these blocks were local, as all the update tasks in the block are likely executed serially by a single cpu. Furthermore, it may be useful use a pre-fetch instruction to fetch, in a batch, all vertex and edge data associated with the update tasks in the block.

4.4 Cilk version of Graphlab

I propose to develop and release an independent implementation of the Graphlab abstraction written using Cilk. My previous work rewriting the Graphlab engine and schedulers has demonstrated that we may encapsulate the complexity of scheduling by using Cilk, and offer varying consistency guarantees via graph coloring. A separate implementation of the Graphlab abstraction would allow us to embrace these two ideas more fully. Also by freeing myself of the burden of maintaining support for all existing Graphlab programs, we'll have the freedom to tweak the model to obtain greater performance usability.

References

[1] Michael A. Bender, Bradley C. Kuszmaul, Shang-Hua Teng, and Kebin Wang. Optimal cache-oblivious mesh layouts. *Theor. Comp. Sys.*, 48(2):269–296, February 2011.

[2] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. Greedy sequential maximal independent set and matching are parallel on average. *CoRR*, abs/1202.3205, 2012.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING*, pages 207–216, 1995.

[4] NPIC500 Dataset. Npic500 data set (v. 2009-09-01). Downloaded from http://www.cs.cmu.edu/ ~ jbetter/all-pairs-t500-matrix-data-code.tar.gz, May 2012.

[5] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.

[6] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 303–314, New York, NY, USA, 2010. ACM.

[7] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 193–204, New York, NY, USA, 2012. ACM.

[8] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Graphlab: A distributed framework for machine learning in the cloud. *CoRR*, abs/1107.0922, 2011.