Supporting efficient aggregation in a task-based STM

Jean-Philippe Martin, Christopher J. Rossbach, Derek G. Murray, Michael Isard
Microsoft Research Silicon Valley
{jpmartin,crossbac,dmurray,misard}@microsoft.com

Abstract

Parallel computing resources are ubiquitous; tools that simplify their programming are not. Software Transactional Memory (STM) makes parallel programming more accessible by eliminating locks, using optimistic concurrency to promise scalability for programs with simple synchronization. However, while modern STMs do simplify the task of writing *correct* code, they often complicate the task of writing *efficient* code: contention for shared resources can yield performance problems that are difficult for programmers to understand and address. Traditional TMs abort concurrent write-sharing transactions, detecting contention by tracking accesses to memory cells or objects. We argue this is the wrong layer of abstraction at which to perform this operation because many parallel constructs rely on data structures for which write- sharing is the common case such as work-lists and aggregators. In such contexts, the traditional approach is unnecessarily conservative and untenable for performance.

We propose mechanisms that enable the programmer to tell the system how to respond to contention at a higher level of abstraction, eliminating whole classes of contention at minimal additional cost in complexity. Our prototype C# STM, Aggro, implements these mechanisms and provides a task- based programming model, eliminating explicit thread management. We show that Aggro can outperform lock-based implementations and other optimistic systems such as Galois.

1. Introduction

Tools that make parallel programming accessible have failed to proliferate as fast as parallel computing resources. Parallel platforms are everywhere but writing parallel code remains difficult. Software transactional memory (STM) promises to simplify parallel programming by eliminating problematic constructs such as locks while reducing contention on synchronization resources: the promised result is scalable programs with simple synchronization code.

While TM has been used with some success to scale data structures, for example within the OS [21] and the Java virtual machine [10], it has not yet gained widespread use in more general-purpose parallel algorithm implementation. We believe that contention, in the form of transaction aborts due to read/write or write/write conflicts, remains a major obstacle to the successful adoption of TM. Many parallel algorithms include commutative aggregation operations, either explictly or implictly as we describe below. These algorithm typically generate "false" conflicts under many TM implementations; transactions abort due to to detected conflicts even though judicious re-ordering of some operations could prevent the abort and preserve program semantics.

Built-in aggregation operations are known to be essential to achieve high performance in many parallel computing frameworks and turn up in systems as different as MPI [17], MapReduce [4], OpenMP [3] and Piccolo [19]. However, they have not previously been integrated directly in a TM system. This paper presents Aggro, a new STM implementation that allows users to perform arbitrary commutative aggregation operations when updating shared objects. Aggro demonstrates substantial performance and scalability improvements available from the inclusion of aggregation in a TM system.

Aggro is designed for workloads with large available parallelism, but where the control flow or data flow are sufficiently irregular that systems like MPI, MapReduce or OpenMP are a poor fit. Examples include: loopy belief propagation, where the order of updates is dependent on previous updates; stable matching, where the choice of partner in each round depends on the choices made by other parties; and mesh refinement, where the set of triangles to update depends on those updated so far. Such workloads have the property that most updates can be performed in parallel, and that conflicts between updates are rare and well-handled by TM mechanisms. Unfortunately, when implemented in a standard TM, write-contention on data structures used during aggregation (such as worklists [14]) becomes a bottleneck. Previous systems that address similar problems eitehr lack general purpose transaction support [13, 19] or require the user to provide explicit commutative compensation functions [12]. Aggro permits straightforward transactional accesses to most shared state with special update functions used *only* for those objects that require aggregation.

Because of the massively parallel nature of Aggro's expected workloads we implement Aggro in an object and task-based style similar to that adopted by AME [8], where each task (or "event") executes in its own transaction and can schedule other tasks that become runnable on commit. Unlike AME, Aggro defaults to using private, non-transactional, accesses, and requires explicit API calls to read and write shared objects. The use of an API for shared object updates means that it is straightforward to add aggregation operations without modifying the syntax of the underlying language, in this case C#. While we believe the task-based style is well suited to the type of application that Aggro targets, there is no reason that aggregation operations could not similarly be added to thread-based TM systems with similar performance gains on suitable workloads.

2. Motivation

Aggregation is a fundamental construct in many parallel workloads. For the purposes of this paper, we define "aggregation" as repeated application of a function f on a set of inputs:

$$y = f(x_1, f(x2, f(x3, ...f(x_{n-1}, x_n))...))) \qquad (1)$$

where f is both *commutative and associative*, so a system is at liberty to group and reorder the function applications to improve performance. Common aggregation functions include Sum and Max, and set insertions (in the absence of deletion operations).

For many applications, aggregation is the primary "communication" phase in a computation: data items are independently processed in one phase, and the results are combined using aggregation in the next. Such applications can be efficiently implemented using MPI, MapReduce and other dataflow systems because the phases are predictable and it is profitable to set up custom communication patterns such as trees or rings to optimize aggregation operations for the particular known memory and network hierarchies.

In another class of applications all state is shared and may be unpredictably and asynchronously updated, so some fine-grain synchronization scheme is required to mediate updates in addition to or instead of coarse-grain phasing. Such irregular parallelism is common when application state represents a graph or mesh (e.g. mesh refinement, stable matching, where concurrent reads or updates to the same node can conflict). While specialized systems such as GraphLab [13] are a good fit for many such applications, they do not efficiently support for shared mutatable state. Aggro is designed to target general-purpose irregular parallelism, including graph computations, using TM mechanisms to give simple serializable semantics to the programmer while enabling parallelism using optimistic concurrency.

Mesh refinement is a good example of an application that Aggro is applicable to. One step of a mesh refinement application examines and modifies multiple elements of the mesh, potentially adding new nodes and splitting edges. It is important, to retain the algorithm's invariants, that these updates occur serially with respect to other updates. Most updates do not conflict, however, so transactional memory with optimistic concurrency is a natural mechanism to adopt. Unfortunately, straightforward implementations of mesh refinement use a work set abstraction, since the application of an update must trigger subsequent operations to look for refinement opportunities in neighboring parts of the mesh. A naïve implementation of a work set might use a singly-linked list (the STAMP benchmark "bayes" adopts this approach, for example) and since every update mutates the list, contention on the datastructure leads to aborts that become a bottleneck for the TM [14].

While previous work has proposed mechanisms tailored to reducing aborts in specific cases [1, 18, 20, 22, 24], Aggro instead proposes a general mechanism for aggregation that can be used in many cases including the reduction of contention on a work set datastructure. We show in Section 5 that Aggro significantly improves the scalability of a variety of workloads including mesh refinement, belief propagation and stable matching, using a transactional API that is easy for the user to reason about due to its serializability.

3. API and Guarantees

Aggro is not intended (like many TMs) to be a drop-in replacement for locks. Rather, Aggro's goal is to simplify the development of *new* massively parallel applications. To that end, Aggro is written in a managed language (C#) and supports a simple, task-based programming model and a language-level API to read and update shared objects using object identifiers (*OID*s). Programmers write code for tasks. All tasks run within transactions. Because shared state access is explicit, STM overheads for non-transactional work are easily avoided.

3.1 Using Aggro

An Aggro program is a partial order of tasks. Tasks are objects that implements a standard interface, with one key method: *run* (Table 1). The partial order of tasks forms a directed graph where each node represents a task, and each edge indicates an ordering constraint. We call this the *task graph*. Since all tasks run as transaactions we use the terms "task" and "transaction" interchangeably. Every Aggro program must define a *main task* (analogous to the `main()` entry point in normal programs). The runtime schedules

```
1  class Main : Task { void run(STM stm) {
2    OID answer = stm.alloc();
3    string[] inputs = loadInput();
4    stm.scheduleLoop(0,inputs.Length,new Process(inputs,answer));
5    stm.schedule(new PrintAnswer(answer));
6  }}
7  class Process : LoopTask {
8    string[] input;
9    OID answer;
10   void run(int start, int count, STM stm) {
11       int[] letterCounts=new int[256];
12       for (int i=start; i<start+count; i++)
13           foreach (char c in input[i])
14               letterCounts[c]++;
15       stm.put(answer, new AddVector(letterCounts));
16 }}
17 class PrintAnswer : Task {
18   OID answer;
19   int[] totals;
20   void run(STM stm) {
21       totals = stm.get_shared(answer);
22   }
23   void onCommit() {
24       printHistogram("Letters␣histogram:␣", totals);
25 }}
```

Figure 1. Sample Aggro program

Task.run(STM s)	normal entry point
schedule(Task[] t)	parallel schedule
schedule(Task t)	serial schedule
alloc()	Returns a new OID
get_exclusive(OID o)	read an object
get_shared(OID o)	read-only access
put(OID o, Object v)	write an object
put(OID o, new Add(int incr))	atomic add
put(OID o, new AddVector(int[] incr))	atomic add
put(OID o, new Merge(Histogram h))	histogram
put(OID o, new Max(int incr))	atomic max
put(OID o, new Union(Bag b))	atomic union

Table 1. APIs for running and scheduling tasks, managing shared objects (OID), and aggregators

tasks respecting the partial order, calling each task object's *run* method. If the transaction cannot commit (e.g. due to read/write conflict) Aggro runs an abort handler, and will retry the transaction later. Aggro manages parallelism for the programmer: tasks do not spawn threads or access locks.

Figure 1 shows an Aggro program we will use to illustrate concepts as we introduce them [1]. The `run` method has one argument: the STM object (see e.g. line 1), which the programmer uses to interact with the runtime, for example schedule new tasks via the *schedule* functions (see Table 1). The task graph is is assembled dynamically in response to schedule calls. A task may make multiple `schedule` calls, which the runtime uses to infer the partial order: all the tasks an the first call to schedule will be ordered before all the tasks in a subsequent call call. Formally, if we have a task graph with a task A with successors {B} and A makes n calls to schedule, with argument $T_i = [t_{i,0}, \ldots, t_{i,n_i}]$ for each invocation $0 \leq i < n$, then the graph has an edge from A to each of the $t_{0,j}$, each of those in turn have an edge to each of the $t_{1,j}$, and so on, until all the $t_{(n-1),j}$ who each have an edge to all in {B}. This

[1] constructors are omitted in the pseudocode

```
A::run(STM stm) { stm.schedule([B,C]);
                  stm.schedule(E); }
B::run(STM stm) { stm.schedule(D); }
```

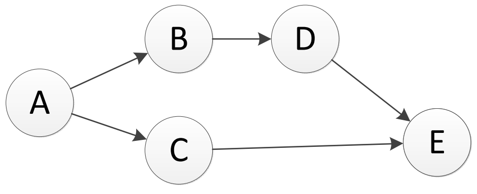

Figure 2. Task graph example

is equivalent to n fork-joins. Figure 2 shows a few calls and the resulting task graph; the uppercase letters are tasks.

Version management in Aggro relies on the OID API. Tasks that read an object use `get_shared` to get a pointer to the object. Tasks that update use `get_exclusive` to acquire a writable *copy* of the object: consequently, updates are effectively buffered and the runtime makes them visible on commit. Aborting a task simply discards the list of written objects. Pointers to immutable variables can be however passed freely (e.g. the *input* variable in the example). The task interface provides *onCommit* and *onAbort* callbacks to handle sideeffects such as I/O; output can be buffered until commit time (e.g. line 23 in Figure 1). Aggro guarantees serializable execution, given correct usage of the API; any call to `get` will return the value from the latest put that precedes it in the serial execution order. Aggro also guarantees progress: if each transaction terminates regardless of input, then the whole program is guaranteed to terminate. Aggro does not guarantee *opaque reads* [5][2]. Conventional C# sandboxing is used e.g. to prevent native method calls from being based on inconsistent state.

3.2 Aggregators

The defining feature of Aggro is specialized support for aggregators such as *add*. Aggregators provide the expected semantics (e.g. *add* will add a value to a variable) but are implemented to avoid aborts due to the write-write conflicts that are fundamental to a traditional TM implemenentation. A program uses aggregators by reading/writing aggregator objects (e.g. line 15 in Figure 1) instead of reading/writing a value. Table 1 shows several pre-defined aggregators Aggro supports. When the programmer reads from an OID previously written with an aggregator, the read returns *the result of executing the aggregator*. The guarantee is that such a read will observe the same value as if the aggregation operations were executed atomically inside of the transaction that invokes them. Concurrent writes to an aggregator type for the same OID do not conflict, and are executed in parallel (modulo serialization required by the runtime to ensure that the final value is correct). Programmers can add their own aggregators. The only requirement is that aggregators is that they must commute. Aggregation is designed to be efficient when many aggregators run concurrently before a subsequent `get` or `put`.

4. Implementation

Aggro uses two datastructures to track reads and writes and decide which transaction sees what values, which commit and which abort. A per-*OID record* (Figure 3) tracks the current committed version of an object. A *transaction context* (Figure 4) stores read-write sets and transaction state. Aggro is a multiple-reader, single-writer (except or aggregations) STM: it detects write/write conflicts eagerly

[2] Only transactions that commit are guaranteed consistent reads: opacity requires that even aborting transactions see a consistent view of memory.

field	type
owner	transaction record ref
committed	object ref
scratch	object ref

Figure 3. OID record

field	type
state	enum {RUNNING,DOOMED, COMMITTING,COMMITTED}
busy	boolean
readset	List<OID,object>
writeset	List<OID>
exclusiveset	List<OID>

Figure 4. Transaction context

and read/write conflicts at commit time. We avoid explicit locks on *OID records* by using a *owner* field to point to a writing transaction (the field is re-purposed when aggregators are in use for an OID). Aggro uses commit-time read-set validation to ensure serializability.

4.1 Aggregation

Our STM also offers *aggregators*. Workloads with a lot of write-write conflicts (for example when adding up many subtotals or concatenating lists) are a poor fit for traditional STMs because they cause too many aborts. Here instead we offer built-in support for aggregation: instead of writing a value as is traditional, we allow the user to write an *aggregator* (e.g. set union or addition). Aggregators commute, so the STM can avoid aborting transactions even in situation where it would otherwise appear that two transactions conflict because they are writing to the same OID.

To use an aggregator, the programmer calls the normal put function but instead of writing a value, they write an aggregator object. Table 1 lists the aggregators we provide, but programmers can also add their own. The only constraint is that aggregation operators should commute with other instances of the same operator.

The key idea behind implementing aggregators is that we change the ownership rule: instead of allowing at most one owner for each OID, we make an exception for the case of aggregators and allow multiple transactions to share ownership of the OID in that case. When in that mode, other aggregation operators can be added in parallel. The OID is returned to the normal mode of operation when get is called to read the resulting value.

A natural way to implement aggregators would be to keep, with each OID record, a list of owners and pending aggregators. When the corresponding transaction commits, we could apply its aggregators. This is what we do, except that instead of burdening each OID record with additional information, we overload the owner field even more than before: we use a special "owner" (called *OperationGroup*) to indicate an OID that is being aggregated. The aggregator's fields include *committed* and *tentative* arrays: they have one entry per thread, so that each thread can aggregate in parallel.

Those arrays are only collapsed back into a single value when the code reads or writes a value to that OID instead of an aggregator. We call this operation a *collapse*. o ensure correctness, the code prevents collapses while there is a pending aggregator (one that has been written but not yet committed or aborted). This scenario doesn't happen in the case we optimize, where the result of the aggregation is read after all the parallel aggregators have committed. The collapse function calls out to the two methods all aggregators must implement: *combine*, which takes two aggregators and returns an equivalent single aggregator, and *execute* which takes an aggregator and a value and returns the new value. For example, combining Add(1) and Add(2) would return Add(3). Executing this

	Configuration
Processor	4×AMD Opteron 6168 1.9GHz
cores	48
L1	separate i+d, 64 KB per core
L2	unified 512 KB per core
L3	unified 2×6 MB shared per 12 cores
Memory	128 GB
OS	Windows Server 2008 R2 64-bit
Default TM parameters	Timestamp contention management no backoff 'Mixed' conflict detection

Table 3. Machine and platform parameters for all experiments.

then would return e.g. 7 if that OID's initial value was 4 before the aggregators.

When a transaction writes an aggregator to an OID, *put_aggregator* code is invoked, which holds the reader lock and applies the aggregator to the calling thread's cell in the *tentative* array. The code is carefully written to deal with multiple concurrent puts (using CAS), and the lock ensures that collapses are mutually exclusive with commits. The written aggregator is initially put in the *tentative* array. It is then removed if the transaction aborts, or applied to the *committed* array when the transaction commits (the *operations* list in the transaction context keeps track of the aggregators for this purpose). Aggregators may be executed in a different order than their corresponding transactions. We rely on the aggregators commuting so that the end result is the same.

5. Evaluation

In this section, we evaluate Aggro using benchmarks from the Lonestar [11] and STAMP [14] suites, along with our own benchmarks to demonstrate that Aggro has good baseline STM performance and that aggregator support can improve scalability in the presence of write-sharing. Table 2 shows benchmarks used in the evaluation, while machine and platform parameters are detailed in Table 3. All metrics and data shown represent the median over five runs of each benchmark.

We argue that Aggro's support for STM, task management, and aggregators makes implementation of parallel algorithms easier at a performance cost that makes it attractive relative to other available tools. To substantiate this argument, we consider multiple implementations of each benchmark representing the diverse tools available to the programmer, providing a detailed context in which to understand the performance of Aggro. Table 4 characterizes the different implementations. To the extent possible, for each benchmark, we present the performance using traditional concurrency tools (threads, locks, barriers), standard STM techniques, as well as another optimistic concurrency system that provides higher level abstractions for managing write-shared data: specifically Galois [12]. We use Aggro without aggregators to represent traditional STM support. Consequently, because Aggro is written in C#, while STAMP and Lonestar are written in C and C++, our Aggro-tx implementations are, in fact, re-implementations of the benchmarks. We follow the algorithms in those benchmarks with as much fidelity as possible, however performance differences deriving from the managed environment (such as garbage-collection overheads) are an inevitable source of noise in our comparisons.

5.1 STM performance: Delaunay triangulation

Delaunay triangulation triangulates a graph of input points to produce a mesh. Our Aggro and managed implementations follow the algorithm and approach in [11]. The native version implements the parallel Dewall algorithm [2] using threads and locks, while Galois is taken directly from Galois release 2.1.4. The workload is rich in irregular data parallelism, but does not feature write-contention that can benefit from aggregators, so the Aggro implementation does not use them making Aggro identical to Aggro-tx: consequently, we omit Aggro-tx results. The workload is commonly evaluated in the optimistic concurrency literature, we evaluate it here to demonstrate that Aggro has competitive STM performance and good scalability for workloads that do not require aggregation.

Figure 5 plots execution times versus thread count for Aggro, Galois, and native. The data show that Aggro provides competitive performance even at low thread counts. Aggro is $\sim$55% slower than single-threaded native code, on par with slowdowns to be expected from a managed environment. Moreover Aggro introduces $\sim$8% overhead compared to a single-threaded C# implementation, demonstrating that Aggro's TM and tasking support are lightweight. As thread counts increase, Aggro outscales all other implementations. The native implementation uses worklists and a divide-and-conquer recursion, which, in the absence of support for speculation, requires partitioning steps to be serialized with subsequent paralllel work at each recursion level. Consequently, native is hobbled both by a fundamental lack of scalability and lock contention at higher core counts: Aggro outperforms it by over 15× at 48 cores. The Galois version outperforms Aggro up to 8 cores due to its lack of managed runtime overheads, but Aggro out-scales it at higher core counts, outperforming it by 5× at 48 cores, an improvement enabled by a factor 4 reduction in the number of conflicts.

Our experiments with this workload confirm that Aggro, like other STMs, can take advantage of the parallelism present in irregular workloads like Delaunay Triangulation. These experiments also show that managed runtime overheads can have a first-order impact on performance and managed STMs must take care to design around this fact. We were surprised to discover that the Aggro version of Delaunay initially outperformed the C# version even on a single thread because Aggro is designed to avoid garbage collection overheads that the managed version was not designed to avoid. The Aggro version stores OIDs for neighbouring triangles in the mesh, while the managed used pointers. Because garbage collector increase significantly as the number of pointers grows, changing the managed version to use indices instead of pointers halved the run time. The managed implementation in these experiments uses this optimization.

5.2 Aggregation performance

5.2.1 K-means

The k-means benchmark clusters N D-dimensional input points into K clusters, mapping each point in the input to its nearest cluster center. Given an initial assignement of cluster centers, the workload iteratively assigns each point to its nearest center and then recomputes the cluster centers as the average of all points mapped to that cluster. The version of k-means from STAMP, upon which our C# reimplemenations are based, partitions the input points across available threads, using transactions to synchronize updates to the cluster centers: as a consequnce, the cluster centers can be very heavily write-contended (our experiments use the input from STAMP that produces the highest contention). The classical approach to parallelizing this kind of aggregation is to give each forked thread a private copy of the cluster centers eliminating the need to synchronize during the parallel phase, adding a subsequent sequential step that merges private updates to produce the final aggregated value. The tradeoffs in this context include additional complexity for barrier synchronization and managing per-thread replicated data. This strategy is used in managed, while the Aggro-tx the synchronization used in STAMP.

The Aggro implementation, in contrast, uses the AddVector operation shown in Table 1 to implement accumulators for the new cluster means, eliminating the need for programmer-managed per-thread "private" data structures while preserving the simplicity of synchronziation in the STAMP version of the benchmark. To elucidate the tradeoffs between the privatization strategy above, and aggregation support provided by Aggro, we additionally evaluated an Aggro-based implementation that uses privatization along with transactions, rather than aggregators.

bnc	description
Delaunay Triangulation	From Lonestar[11] 2.1.4. Triangulates 100,000 random points to form a mesh.
Delaunay Refinement	From Lonestar [11] 2.1.4. Refines a triangulated mesh. Uses the “massive.2” input from Lonestar.
kmeans	From STAMP [14] K-means clustering algorithm, use the “random-n2048-d16-c16.txt” input to map 2048 16-dimensional vectors to 15 clusters.
word-count	map reduce-like application producing a histogram of words in Tolstoy’s “War and Peace”.

Table 2. Benchmarks used to evaluate Aggro.

name	PL	description
Aggro	C#	Uses all Aggro features: STM, task management, and aggregators where possible.
Aggro-tx	C#	baseline STM: Uses Aggro STM and task management support not aggregators.
Galois	C++	Implementation drawn from Galois 2.1.4 release.
managed	C#	Represents best-case managed performance without optimistic concurrency. Explicit threading, scheduling/partitioning. Synchronized with locks and event support from .NET 4.0.
native	C++	Represents best-case performance without optimistic concurrency. Explicit threading, scheduling/partitioning. Synchronized with locks and event support from the Win32 API.

Table 4. Implementations used in experiments for evaluating Aggro.

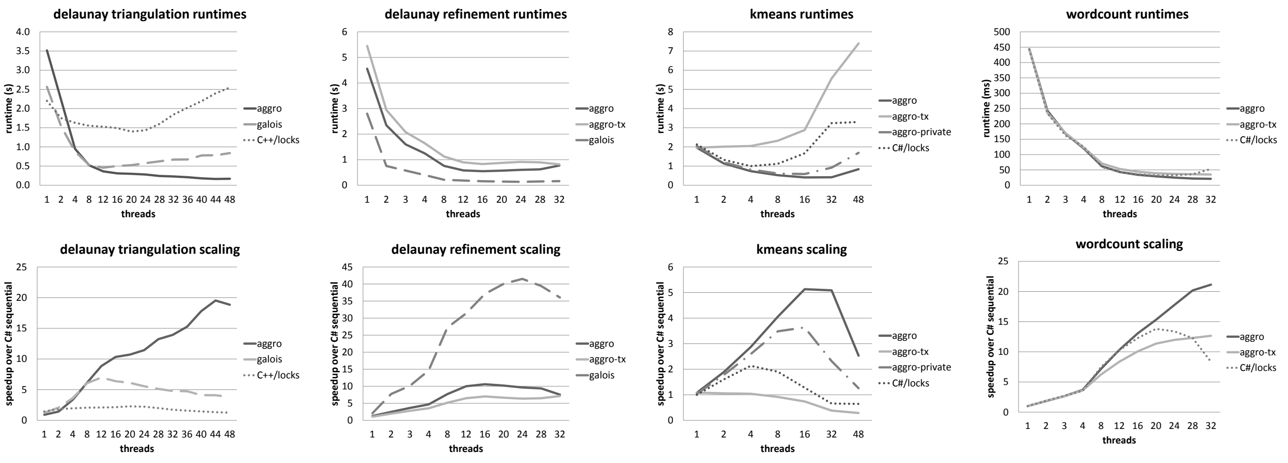

Figure 5. Runtime and Speedup over C# sequential for all benchmarks

Figure 5 plots runtime versus thread count for the different versions of kmeans. The data illustrate that without aggregation support, write-contention renders Aggro-tx completely unscalable. The managed and Aggro versions show that eliminating write-contention with per-thread copies of the cluster centers restores scalability. The Aggro version completely eliminates write-contention without introducing additional overheads for managing per-thread replication of data structures, enabling Aggro to provide the best performance, with a peak speedup of 5× over sequential code at 32 cores. Aggro improves over even managed by a geometric mean speedup of 2.86× over all core counts, compared to 1.19× for managed, which is perhaps surprising given that managed does not incur STM and task-management overheads incurred by Aggro. The performance win derives from better cache locality resulting from fewer copies of the cluster means, as well as overheads for thread management and barrier synchronization in .NET 4.0.

5.2.2 Delaunay mesh refinement

Delaunay mesh refinement accepts a triangulated mesh as input and produces as output a transformed mesh in which all triangles satisfy a quality metric: all angles must be at least 30 degrees. The benchmark is well-described in [12]. The algorithm operates on a worklist of “bad” triangles: each step in the refinement replaces a subgraph that does not satisfy the quality constraint with a larger subgraph that does: however, the local refinement may produce new non-conforming triangles, which must be added to the worklist. Consequently, the worklist can be write-contended in the common case, limiting scalability for lock-based and STM implementations alike.

The Aggro-tx implementation synchronizes additions to the worklist with transactions. Like the Galois version, our Aggro version takes advantage of the commutativity of set insertion to define conflicting operations at a higher level of abstraction and thus reduce contention: Aggro uses the Union aggregator shown in Table 1 to add new items to the worklist.

Figure 5 plots runtime versus core count for our refinement implementations. Data shown are for the ’massive’ input provided with the Galois release, but other input sizes yield similar trends. The data show that the Aggro-tx’s frequent additions to the task graph to represent the the worklist of ’bad’ triangles leads to very high abort/retry rates in Aggro-tx, hurting its scalability relative to the other implementations. In contrast, Aggro eliminates aborts due to conflicts on the worklist. The geometric mean across all thread-counts of reduction in abort rate relative to the Aggro-tx implementation is $\sim 13\times$. The data also illustrate that refinement incurs overheads in Aggro not incurred by the triangulation workload. For triangulation, Aggro outperforms Galois, while Galois is consistently 2× faster than Aggro for refinement.

The difference in scalability derives ultimately from false ordering dependences induced by the structure of the Aggro task graph. Refinement uses a worklist, and by using aggregators to manage additions to the worklist explicitly, we induce an alternating pattern of parallel and serial phases. The serial phase reads the worklist schedules a parallel task for each outstanding item, and clears the worklist. In the parallel phase, each task refines part of the mesh, conditionally adding new bad triangles to the worklist through the aggregator. Recall from Section 3 that Aggro enforces ordering constraints in the task graph which are in turn induced by the order of calls to `schedule`: all predecessor tasks must complete before a Aggro will run a task. Consequently, load imbalance in the parallel phase cause many threads to idle waiting for all parallel tasks to become runnable. The impact on scalability increases with thread-count: at 16 cores Aggro spends a cumulative 7% of the workload idling.

While the ordering dependences induced between waves by Aggro are false, the structure is preferable to one in which tasks that create new bad triangles add to the Aggro task graph directly for a number of reasons. First, it allows additions to the worklist to be aggregated, in this case by removing duplicates and randomizing order to minimize conflicts in the coming parallel phase, neither of which is currently possible through the scheduler API. Second, it reduces contention on the task graph by batching updates to it. Finally, it constrains the task graph to a structure that Aggro is well-optimized to handle: Aggro is optimized for large waves of parallel sibling tasks, and overheads for managing the task graph increase as the structure of the graph becomes less regular. Future work with Aggro will explore support for worklists as a first-class concept to help eliminate the negative impact of false ordering dependences.

5.2.3 Word count

The word count benchmark is a standard map-reduce computation [4] that produces a histogram of words in an input corpus. Our managed parallel implementation relies on the traditional approach to parallelizing this type of aggregation: it uses additional code to manage, merge, privatized per-thread updates. Our Aggro-tx implementation also performs privatization, but uses separate transactions to write partial results to the global histogram; the Aggro version uses the Add aggregator from Table 1 to ameliorate write-contention.

Figure 5 shows runtime and speed up over sequential for our different wordcount implemenations. The managed version provides the best performance through 20 threads, but at higher core counts is bottlenecked by contention on the concurrent bag that is used to aggregate partial histograms from different workers. Aggregation support in Aggro eliminates 100% of the conflicts that occur in the Aggro-tx version which enables better scaling at higher core core counts. The Aggro-tx achieves peak speed up of only $\sim$13$\times$ at 32 cores, compared to $\sim$22$\times$ for Aggro.

6. Related work

A comprehensive review of the TM literature through 2010 can be found in [6]. Many TM systems support extensions or changes to the programming model that give the programmer tools to increase performance, generally at the cost of additional complexity or reasoning. Programmers can use privatization [23] and (publication) to precisely manage when data is accessible by other transactions, reducing conflicts as result. Early release [22] allows a programmer to eject individual objects or memory cells from the read-write set of a transaction, in effect dropping isolation on those objects. Escape actions [15, 24] and open nesting [16] are similar in spirit, providing transactions with mechanisms to pause and perform operations without isolation. In all cases, additional programmer effort is required, as the correctness guarantee falls on the programmer. Aggregators are specialized to support patterns common in parallel computations, and relative to these techniques, introduce little additional programmer complexity.

With Galois classes [12] and transactional boosting [9] the programmer provides inverse operations for each operation optimistinc operation on concurrent data structures. Abstract nested transactions (ANTs) [7] mitigate the performance impact of benign or structural conflicts, but the programmer must identify the regions of code likely to be involved. Aggro is similar to these mechanisms in that it focuses on operations for which conflict detection and resolution can be better performed using mechanisms other than checkpoint/rollback and RW-set tracking.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP*, 2007.

[2] P. Cignoni, C. Montani, and R. Scopigno. Dewall: A fast divide and conquer delaunay triangulation algorithm in ed. *Computer-Aided Design*, 30(5):333 – 341, 1998.

[3] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, Jan./Mar. 1998.

[4] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[5] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, 2008.

[6] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. 2010.

[7] T. Harris and S. Stipic. Abstract nested transactions. In *TRANSACT*, 2007.

[8] M. Isard and A. Birrell. Automatic mutual exclusion. In *HotOS*, 2007.

[9] E. Koskinen and M. Herlihy. Concurrent non-commutative boosted transactions. In *TRANSACT*, 2009.

[10] C. Kotselidis, M. Lujan, M. Ansari, K. Malakasis, B. Kahn, C. Kirkham, and I. Watson. Clustering jvms with software transactional memory support. In *IPDPS*, 2010.

[11] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[12] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[13] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, 2010.

[14] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In D. Christie, A. Lee, O. Mutlu, and B. G. Zorn, editors, *IISWC*, pages 35–46. IEEE, 2008.

[15] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *ASPLOS*, 2006.

[16] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, Dec. 2006.

[17] MPI. http://www.mcs.anl.gov/mpi/.

[18] Y. Ni, V. Menon, A. Tabatabai, A. Hosking, R. Hudson, J. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP*, 2007.

[19] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *OSDI*, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association.

[20] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *MICRO*, 2008.

[21] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *SOSP*, 2007.

[22] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *Proceedings of the Workshop on Transactional Memory Workloads*, 2006.

[23] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *PODC*, 2007.

[24] C. Zilles and L. Baugh. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *TRANSACT*, 2006.