Disentangling Multi-object Operations

(EXTENDED ABSTRACT)

Yehuda Afek [*] Michael Merritt[†] Gadi Taubenfeld[‡] Dan Touitou[§]

Abstract

We consider the problem of implementing atomic operations on multiple shared memory objects, in systems which directly support only single-object atomic operations. Our motivation is to design algorithms that exhibit both low contention between concurrent operations and a high level of concurrency, by disentangling long chains of conflicting operations. That is, operations that access widely disjoint parts of a data structure, or are widely separated in time, should not interfere with each other.

The algorithm reported here extends and is based on the work of Attiya and Dagan [AD96], where a non-blocking solution is presented for two-object atomic operations. For any number, k, we present a wait-free solution for atomically accessing up to k objects. Notions of local contention and local step complexity are defined, and it is shown that the solution has low local contention and local step complexity. Relations between multi-objects and the familiar resource allocation problem are explored-the algorithm presented also provides a solution to the resource allocation problem.

1 Introduction

Consider a data structure stored in shared memory and accessed concurrently by n processes. The shared data structure is abstracted as an array of memory locations. To perform its operation, a process needs to get exclusive access to the set of locations necessary to carry out the operation. We assume that the set of locations required by each such *multi-object* operation is fixed at the beginning of the operation. (As opposed to more general transactions on data structures that decide on the locations to use on the fly).

Except for a wait-free requirement, implementing multi-object operations is essentially the well-known *resource allocation problem*, where locations are abstractions of more general shared resources. Typically, a resource in a concurrent system may be accessed by only one process at a time. To share a single resource, processes coordinate using mutual exclusion algorithms. When processes require simultaneous access to multiple resources, the task of coordinating them properly is the resource allocation problem. Hence, non-blocking and wait-free multi-object algorithms provide fault-tolerant solutions to the resource allocation problem, at least in those cases in which the actions taken in the critical section are such that processes can help each other. If helping processes in the critical section is not an option, fault-tolerant multi-object algorithms such as ours can be modified to simply wait for other processes instead of helping them—resulting in resource-allocation algorithms for the fault-free model with small failure locality when failures do occur [CS96].

Our motivation then is to design algorithms for shared data structures and shared resources, supporting multi-object operations, that have low contention and a high level of concurrency. That is, operations that access disjoint parts of the data structure, or are widely separated in time, should not interfere with each other. For example, operations on disjoint sets of components can proceed independently, and avoid concurrency-control overhead [Bar93, IR94, TSP92, ST95, AM95a]. When there is considerable contention, conflicting operations could form long chains and complex webs, transitively effecting operations that are otherwise disjoint. Extremely fast algorithms might be designed that "disentangle" these webs, and require coordination only among local neighborhoods of contending operations. Recent work by Attiya and Dagan achieves this goal for operations on pairs of locations, obtaining a non-blocking algorithm for 2-location atomic updates [AD96]. Using algorithmic ideas from the work of Attiya and Dagan, we describe a *wait-free* solution for any fixed number of lo-

[*]Computer Science Dept., Tel-Aviv Univ., Israel 69978, and AT&T Labs. Partial support provided by BSF grant 94-00297.

[†]AT&T Labs, 180 Park Av., Florham Park, NJ 07932-0971. Partial support provided by BSF grant 94-00297.

[‡]The Open Univ., 16 Klausner st., P.O.B. 39328, Tel-Aviv 61392, Israel, and AT&T Labs.

[§]nSOF Parallel Software, Israel

cations, k. To describe the properties of this solution, we also introduce new definitions for specifying the local behavior of algorithms—the degree to which they succeed in disentangling webs of conflicting operations.

In the remainder of this section, we briefly introduce our local complexity measures, provide an overview of the algorithm itself, and close with a discussion of the extensive literature on this and related problems. Section 2 formally defines and motivates our local complexity measures, and Section 3 presents our algorithm in more detail, concluding with a statement of its properties.

1.1 Local complexity

In defining our complexity measures we use the familiar notion of a *conflict graph*. Informally, two multi-object operations are in conflict if they are concurrent and the set of locations they need intersect. A conflict graph is a graph whose set of nodes is the set of operations such that every two conflicting operations are connected by an edge.

We define two measures, both simple and easy to apply, capturing the dependence of an operations' contention and step complexity on its local neighborhood in the conflict graph. Informally, an algorithm has d-local contention if any two operations that access the same location simultaneously are at distance at most d from each other in the conflict graph. An algorithm has d-local step complexity if the step complexity of any operation is bounded by a function of the number of operations within distance d of it in the conflict graph. (Exact definitions are given in Section 2.)

Notice that d-local step complexity does not imply d-local contention. For example, we could define a single huge Read/Modify/Write (RMW) object, consisting of the union of all the individual objects, and define each multi-object operation as a single RMW operation on this object. Because each operation consists of a single atomic step (on a huge object), this algorithm has small d-local step complexity. (Indeed, the step complexity is 1, and is d-local for any $d \geq 1$.) But any two operations may access the single object, whether they conflict or not, so the algorithm does not exhibit local contention.

Most of the non-blocking multi-object implementations in the literature (e.g. [IR93, AM95a, ST95]) are based on the same universal single-location operation, LoadLink/StoreConditional (LL/SC). Instead of LL/SC, our algorithm is based on the existence of a general, atomic, single-location RMW operation, where each location may hold many fields. This allows us to concentrate on the new ideas of our algorithm, rather than on the difficulties of the extra synchronization necessary between LL/SC pairs. Nevertheless, we discuss in Section 3.3 how to replace the large RMW object associated with each location by an implementation that uses LL/SC on single words (given in [ADT95]) and does not change the local contention and local step-complexity properties of our algorithm.

1.2 The algorithm in a nutshell

In this paper we present a wait-free multi-object algorithm on k objects, that has d-local contention, and d-local step complexity, for $d = O(\log^* n)$. We first develop a non-blocking algorithm and then combine it with a simple mechanism which turns it into a wait-free algorithm.

Following Attiya and Dagan [AD96], each multi-object operation in our non-blocking algorithm has four phases: *filter*, *decide*, *lock*, and *execute*.

The full conflict graph on the active operations has no *a priori* structure, making it difficult to coordinate using a local algorithm. The *filter* phase guarantees that at any point of time, the subgraph induced (on the conflict graph) by the active operations that are through the *filter*, is a forest of $(k-1)$-ary trees.

The *filter* itself has two parts: in the first, each operation marks its $k-1$ lower memory locations as its children; in the second part it marks the highest location as its parent. At any time, each location may be marked by two operations: as the child of at most one operation and as the parent of at most one operation.

To ensure that the *filter* is non-blocking and local (with no long waiting chains) the $k-1$ child locations must be marked in one atomic operation by the k multi-object operation. To accomplish this, we use one of the key observations in generalizing the Attiya-Dagan algorithm: our algorithm is applied *recursively* to atomically mark the $k-1$ lower memory locations, using ordinary single object RMW operations as the base of the recursion. In Section 3, Figure 1 illustrates the recursive data structures required by the filter phases, and Figure 2 provides a detailed example.

Once through the *filter*, in the *decide* phase we apply the 3-coloring algorithm of Goldberg, Plotkin, and Shannon [GPS87] to color the operations in each directed tree. The 3-coloring algorithm is a generalization of the Cole-Vishkin algorithm [CV86] which was used by Attiya and Dagan for the same purpose over a path (1-ary tree).

The three colors are used to determine the *locking order*, the order in which operations acquire the locks for their locations (children then parent, or vice-versa) during the *lock* phase, so that waiting chains are kept short. The three colors are ordered, and each node determines its locking order based on the relative values of its color and that of its parent. In the three-coloring of the tree, monotonic paths toward the root cannot be longer than three nodes, and so at most two consecutive nodes can hold locks on their children, and wait for a lock on their parent location – forming a waiting chain.

Since the third node is labelled with an extreme color, it will have locked its parent location before obtaining the locks for its children. Now through the *lock* phase, this operation has gained exclusive access to the k memory locations it needs for its operation. After executing the operation, it releases the locks, and removes itself from the forest, allowing any waiting children to contend for the lock on the parent location.

This abstract description ignores many details of the non-blocking algorithm. An important omission is the standard technique of a blocked process *helping* the process that blocks it, which is how algorithms are made non-blocking [Her91, Her93]. If at any phase process p is blocked by another process q, then p helps q to proceed until q does not block p anymore, and then p continues. In addition, both algorithms guarantee that no operation will help another at distance greater than $O(\log^* n)$.

We apply a familiar technique for converting the non-blocking algorithm to a wait-free version [Her91, Her93, AM95a]: A process that properly terminates its non-blocking operation, must help other pending operations, before it starts a new operation of its own. To obtain a wait-free algorithm with local contention and step complexity, we apply this technique separately to each location, and each terminating operation helps only pending operations with which it directly conflicts.

1.3 Related work

Early work in multi-objects focused on the synchronization power of m-registers, multi-objects in which the operations are restricted to reads and writes of m locations [Her91, MT94]. The synchronization power of other multi-objects is studied in [AMT96]. Recent attention has focused on the multi-object problem out of a desire to find truly fast implementations of shared data structures that are also non-blocking or wait-free.

General multi-object algorithms: Early work on wait-free and non-blocking data structures provided methods to transform sequential implementations of arbitrary shared objects into wait-free concurrent implementations, assuming the existence of a universal primitive [Her91, Plo88, Plo89, JT92]. Those methods demonstrate the potential of wait-free algorithms, but are too inefficient to use in practice. Herlihy introduced a universal translation method to transform a sequential implementation into either a non-blocking or a wait-free one using LL/SC [Her93]. These methods are simple and easily implementable, but still each process has to copy the entire data structure even for a minor update. In addition, while using the wait-free method, each process must take $O(n)$ steps even when accessing the data structure alone. The problem was also considered in a timing based model [AT93].

A search for efficient implementations proceeded. Barnes presented algorithms with the idea of "simulating" the update without effecting the shared memory, and then to apply the changes atomically using a non-blocking implementation of a multi-word RMW [Bar93]. This idea was developed further by Israeli and Rappoport [IR94]. Turek, Shasha, and Prakash explored similar constructions using an atomic compare&swap primitive [TSP92]. These non-blocking algorithms avoid copying the entire data structure, and allow disjoint access: operations in disjoint portions of the conflict graph can proceed independently. Processes waiting for each other recursively can form long chains of size n.

Shavit and Touitou present a non-blocking implementation of multi-word atomic RMW from LL/SC [ST95]. This non-blocking algorithm also avoids copying the entire data structure and allows disjoint access, and has only constant-length helping chains: a process has to help only its closest neighbors, which reduces contention. Simulation results indicate that it may perform well in practice. But it can give rise to waiting chains of size n: although it has to help only its closest neighbors, the number of steps a process performs is effected by neighbors at distance n from it.

Anderson and Moir explore a slightly different problem, proposing to use implementations that assume the concurrent execution of only $k < n$ processes, and to protect the implementations using a k-exclusion and a k-renaming algorithm [AM94]. In this way a process that accesses the data structure alone has a $O(k)$ complexity instead of $O(n)$. This fault model is not as general as wait-freedom. In addition, the algorithms neither avoid copying nor have disjoint access.

Afek, Dauber, and Touitou present a wait-free, universal translation method that avoids copying the entire data structure and in which the step complexity depends only on the (global) contention, but that does not allow disjoint access [ADT95].

In a later paper, Anderson and Moir present a wait-free methodology that avoids copying the data structure, but has $\Omega(n)$ step complexity for even a single operation [AM95b]. They also present a wait-free multi-word atomic implementation that allows disjoint access, but that gives rise to helping chains of length n [AM95a]. Moreover, an operation has $\Omega(n)$ step complexity even if it accesses the data structure alone.

A final predecessor to our work, the non-blocking algorithm of Attiya and Dagan implements 2-location atomic updates with helping and waiting chains of length only $O(\log^* n)$ [AD96]. They also introduce the notion of the sensitivity of one operation to another, and show that in their algorithm, operations are only sensitive to others within an $O(\log^* n)$ distance in the conflict graph. We generalize their algorithm in two essential ways, replacing their unary trees (disjoint paths) with k-ary trees, and using recursive calls to atomic $(k-1)$-multi-object operations. The result is a non-blocking algorithm implementing atomic k-multi-object atomic operations, for any fixed k. We then show how to modify the non-blocking algorithm and obtain a wait-free version. Our generalization solves an important question [Bar93, IR94], demonstrating the possibility of a truly fast multi-object algorithm, in which complexity of operations is bounded only by the local contention, rather than by the global contention as in [ADT95].

We introduce the notions of d-local contention and d-local step complexity, measure our algorithm according to these metrics, and advocate them as appropriate for investigating the local properties of concurrent algorithms.

The resource allocation problem: Research on coordinating the sharing of multiple resources, rather than multiple memory locations, has a history that antedates this recent interest in multi-object operations. Various abstractions have been suggested to model this problem, which is essentially the same as the multi-object problem. In his pioneering work, Dijkstra introduced the dining Philosophers problem in which the potential conflict graph is very simple—a cycle [Dij72]. Since then, dozens of solutions to this problem have been proposed. Lynch showed that the dining philosophers problem can be extended to an arbitrary graph network, in which a philosopher needs to acquire the resources on all incident edges [Lyn81]. Chandy and Misra further generalized this by introducing the Drinking Philosophers problem, in which a philosopher needs some non-empty subset of its resources; this subset of resources may change over time. These papers assume a fault-free model [CM84].

A few recent papers on the resource allocation problem in asynchronous message passing systems have considered the notion of *failure locality* [SP88, AS90, CS95, CS96]. Failure locality of an algorithm is defined as the smallest m such that any process, for which there are no failures within a distance m in the conflict graph, is free from starvation. All these papers achieve a constant failure locality, while trying to minimize response time and message complexity. (Failure locality does not imply short waiting chains.) The above papers do not use any kind of helping method and are neither non-blocking nor wait-free.

It is also possible that highly concurrent resource allocation algorithms, such as that due to Awerbuch and Saks, may be the basis for efficient non-blocking or wait-free multi-object algorithms [AS90].

2 Complexity measures

In this section we formally define complexity measures that capture how much operations interfere with, or effect, each other in a given algorithm.

DEFINITION 1

- *Two multi-object operations are* in conflict *if they overlap in time and the set of memory locations (or resources) on which the operations are based, intersect.*
- *A* conflict graph of a time interval *is a graph whose set of nodes is the set of operations that are active at some time during the interval, and there is an edge between any two operations that are in conflict.*
- *The* conflict graph of a set of operations *Op is the conflict graph of the time interval that begins with the earliest start of the operations in Op, and ends with the latest finish (or ∞, if one or more operations in Op have not terminated).*
- *An algorithm has* d-local step complexity *if the step complexity of any operation O is bounded by a (monotonic) function of the number of operations within distance d of O in the conflict graph of O.*
- *An algorithm has* d-local contention *if in any run of the algorithm two operations O_1 and O_2 access the same primitive object simultaneously only if O_1 and O_2 are within distance d of each other in the conflict graph of $\{O_1, O_2\}$.*

Notice that local step complexity implies non-blocking but not wait-freedom. The multi-object algorithm presented here is wait-free, has $O(\log^* n)$-local contention, and $O(\log^* n)$-local step complexity.

2.1 Local step complexity

The conflict graph of a set of operations determines what it means for two multi-object operations to be close to each other, either in time or in space.

Attiya and Dagan defined the notion of *sensitivity*, which measures the minimum distance between two operations that guarantees that they will not effect each other's step complexity[AD96]. Although sensitivity of distance d implies d-local step complexity, the Attiya-Dagan notion of sensitivity is neither robust (independent of the semantics of the shared objects) nor does their algorithm exhibit d-local contention for any d. In their algorithm, $(n-1)$ concurrent StoreConditional operations can contend for the same address, even though the operations are not adjacent or even connected in any conflict graph. All but one will fail, and so they don't effect each other's step complexity (assuming only one step is counted for each StoreConditional). But if the StoreConditionals's are replaced with a subroutine built from e.g., RMW objects, then the fact that they are all concurrent gives them the chance to effect each other's step complexity inside the implementation, in terms of the number of accesses to the subprimitive objects and local memory.

Moreover, reasoning about sensitivity is difficult. Defining and tracing the possible *effect* of one operation on another in a run requires understanding the semantics of the multi-object transactions and of the operations on individual locations, to determine how information can flow between operations. Also, determining the effect of one event on another typically requires reasoning about pairs of runs (comparing a run to a similar run in which the event does not occur). Because our algorithm is wait-free, we can avoid such careful reasoning about the effect of one operation on another, instead focusing on the coarser measure of an upper bound on the total step complexity of an operation. (That is, an operation distant from O may effect O's step complexity, by increasing or decreasing it, so long as it does not exceed the upper bound imposed by the number of O's closer neighbors. More concretely, suppose O has m other operations as neighbors in the conflict graph, and in addition a long chain of operations hanging from it. Then 1-local step complexity allows O to work in a different manner depending on whether this chain exists, in particular increasing or decreasing its step complexity, as long as it remains bounded by a function of m. If the algorithm exhibits sensitivity of distance 1, the number of operations performed by O must be *exactly* the same, whether the chain exists or not.)

2.2 Local contention

Following Dwork, Herlihy and Waarts and the fact that hardware is not magic, we assume that the cost of accessing a shared location is effected by the contention concurrent with the access [DHW93]. For example, in a *contention sensitive* implementation, the time taken by a single low-level operation could be a linear or logarithmic function of the number of simultaneously contending low-level operations. In implementing a multi-object operation, we seek to keep the total number of low-level operations (the step complexity) small, but also to minimize the contention of each of those operations. In algorithms with d-local contention, operations are permitted to simultaneously access the same shared location only if they are within distance d of each other in the conflict graph. In algorithms exhibiting 1-local contention, for example, only neighboring operations may simultaneously access the same shared location X.

Our definition of contention could be called *simultaneous contention*, because it focuses on the number of operations *simultaneously* accessing a shared location X. An alternative definition of contention would then be *serial contention*, the total number of operations that access X and overlap with the operation. For example, a single long operation O on X could be delayed by a series of short operations on X. In such a run, the simultaneous contention of O is only two, but the cost of O might better be measured in terms of its serial contention.

We rejected the stronger definition of serial contention for two reasons: it is much more complicated to reason about, and it is not compositional. By *compositional*, we mean that an algorithm with d-local contention that makes primitive operations on X, should remain d-local if X is replaced by an arbitrary implementation of X from components $x_1, \ldots, x_d$. That is, the contention of operations on the x_i objects should also be bounded (to multi-operations that are within a d-neighborhood of each other). This property is trivially true for simultantaneous contention: two operations can simultaneously access an x_i only if they are also simultaneously accessing X! However, it is possible to construct examples of algorithms with d-local serial contention on X, that do not have d-local serial contention on the x_i. (A single operation on X can be expanded into a series of operations on an x_i.)

3 The multi-object algorithm

This section expands on the nutshell description of the algorithm from the introduction. We begin with a nonblocking version of the algorithm, and then describe how to modify it to produce a wait-free version.

3.1 Outline of the the non-blocking k-object algorithm

The non-blocking algorithm has four major phases, as in [AD96]: *filter*, *decide*, *lock*, and *execute*. The operations that have passed the *filter* induce a subgraph of the conflict graph that is a forest of (k-1)-ary trees. In the *decide* phase each operation scans $O(\log^* n)$ of its ancestors in its tree to color the operations in the tree with 3 colors. The three colors are then used in the *lock* phase to decide which memory locations to lock first, the one in the direction of the parent, or the $k-1$ children. If operation O fails to lock a location because it is held by another operation, then O helps the other operation until it releases the lock, and tries the lock again. Once the k memory locations are locked, the operation is executed. Finally, the k locks are released, the operation removes itself from the tree, and terminates. In more detail:

Initialization: Each operation starts by computing a unique operation id, *oid*, and initializing necessary data structures.

Filter: The purpose of the *filter* phase is to select from all the operations that are simultaneously accessing the shared memory, a subset whose conflict graph is a forest of rooted $(k-1)$-ary trees. It is similar to the *filter* phase in the Attiya-Dagan algorithm, where $k = 2$ [AD96]. As

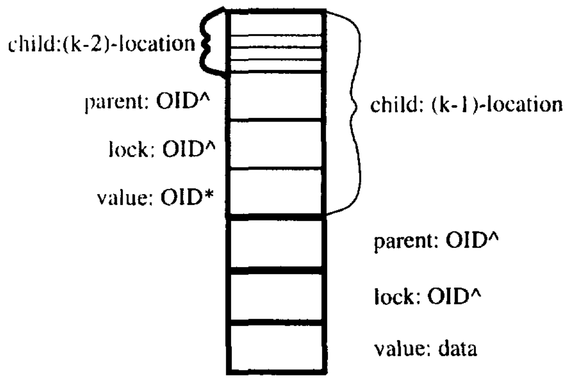

Figure 1: Key fields of a shared location, showing recursion in the child field.

illustrated in Figure 1, in addition to a **lock** and a **value** field, each memory location contains two other fields: a **parent** field and a **child** field. The **child** field is recursively constructed in the same way, **lock**, **value**, **parent**, and the (recursive) **child**, to support (k-1)-object multi-object updates, at the next lower level of recursion. At the lowest level, the child field consists of a single **value** field.

Let OID be the set of all operation identifiers, define OID$^\wedge$ = OID $\cup$ $\{\perp\}$ and define OID* to be the set of all strings of OID's, with the additional symbols $\perp$ and *root*. The **parent** fields hold elements of OID$^\wedge$. The **value** field at the highest level holds the data values of the k-object multi-object, and at the lower levels, hold elements of OID*.

An operation passes the *filter* by writing its *oid* in the **child** fields of its $k-1$ lowest-index locations, and in the **parent** field of its highest index location. The $k-1$ **child** fields are updated atomically using a recursive call to a $(k-1)$-multi-object RMW algorithm. Hence, the **child** field is actually a $(k-1)$-level memory location, and the *oid* stored there is written to its **value** field by the $(k-1)$-level recursive call. The **parent** field is updated using single-word RMW operations. The $(k-1)$-level RMW that marks the **child** fields does so only if it finds them all unmarked by other operations; similarly, the RMW to the **parent** field succeeds only if it is unmarked.

For example, suppose that a process performs an atomic multi-object operation t on k memory locations $\ell_1, ..., \ell_k$, where $\ell_1 < \ell_2 < \cdots < \ell_k$. Memory location ℓ_k is called the ***high*** location of t and the $\ell_1, \ldots, \ell_{k-1}$ locations are the $(k-1)$ *low* locations of t. In the ***filter*** phase, t uses a $(k-1)$-multi-object RMW algorithm to try to mark its *oid* in each of the **child** fields in memory locations, $\ell_1, \ldots, \ell_{k-1}$. The recursive call prevents interleaving of the marking by conflicting operations, which could introduce deadlocks when each requires a child marked by the other, or long waiting chains in which each process needs a child marked by another.

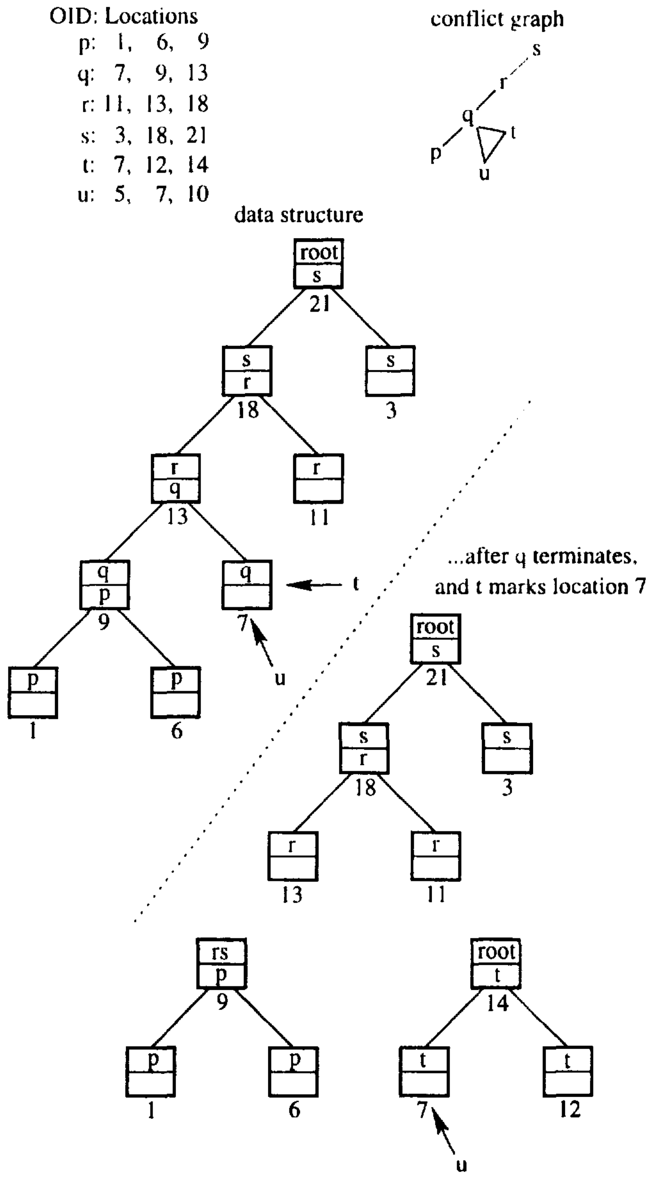

Figure 2: A set of conflicting operations, the corresponding conflict graph, and two snapshots of the data structure from an example execution.

If operation t succeeds in marking all $k-1$ **child** fields, then it tries to mark the **parent** field of ℓ_k. If it succeeds in that field too, then t is through the *filter*. An operation that has marked the **child** field of ℓ_k (and is otherwise through the *filter*) is t's parent in the tree, and those operations that have marked the **parent** field of any of the $\ell_1, \ldots, \ell_{k-1}$ locations are t's children. Figure 2 illustrates a specific example of a set of conflicting operations that each access three locations, the corresponding conflict graph, and two snapshots of the data structure that could be produced as different operations enter the filter. Operations p, q, r, and s, are through the filter, and the figure illustrates the resulting binary tree. Operations t and u are trying to mark the **child** field of location 7. Since this is already marked by q, both attempts will fail, and the processes executing operations t and u will help q finish its operation and unmark the location. The bottom part of the figure illustrates the update to the data structure which occurs when q terminates, and t wins the competition with u to mark location 7 and exit the filter.

As just illustrated, if an operation t fails in its attempt to mark the **child** fields, then it helps one of the operations that is blocking it, and retries. Similarly, if an operation t fails in marking the **parent** field of the high location (ℓ_k) then it unmarks the **child** fields at the low locations, helps the operation that already marked the **parent** field at the high location, and retries.

There is an exception to the behavior of *filter* at the root of the tree, motivated as follows. During the *decide* phase, operations read the *oid*'s in a $O(\log^* n)$ path toward the root to compute a 3-coloring. Operations closer than this to the root pad the string they have read with a default string. If operations could join an existing tree by becoming the parent of its root, some nodes in the tree might read to the old root and use a padded path, while later operations read past the old root and see the new operations. Such inconsistent views of the path could lead to illegal colorings. Therefore, once a memory location is the root of a tree (either the **child** field is unmarked at the time an operation marks the **parent** field, or a successful operation unmarks the **child** field), no operation may again mark it as a **child** until the operation that marked the **parent** field has finished. Thus, when an operation marks the **parent** field and finds the **child** field of the same location vacant, it simultaneously marks the **child** field with a special **root** symbol. (In Figure 2, operation s has marked location 21 as a root.) An operation that fails to mark a **child** field because it finds the **root** symbol in it, helps the operation that has marked the **parent** field of the same location.

A somewhat more involved mechanism is used when a successful operation unmarks a **child** field, creating a new root, if the **parent** field of that location is currently marked. (In Figure 2, this occurs when operation q unmarks location 7.) To ensure the descendants of the successful operation read the same $\log^* n$ string, the successful operation writes the string it observed into the **child** field. Like the **root** symbol, this string prevents other operations from marking the child field, until the operation that marked the **parent** unmarks it. (Hence, in the figure, the **child** field of location 7 is marked with the string rs when q unmarks it.)

Decide: In this phase each operation decides its locking direction–whether to first lock the high location (ℓ_k) and then lock the $k-1$ low locations ($r\ell_1, \ldots, \ell_{k-1}$), or to first lock the $k-1$ low locations and then the high location. To make this decision without creating long waiting chains along the $(k-1)$-ary tree, we color the operations in the tree with 3 colors using the Goldberg Plotkin and Shannon algorithm [GPS87]. Each operation locks its high (parent side) location first if its immediate ancestor in the tree has a larger color, otherwise it locks the low locations first.

To apply the coloring algorithm, operation p has to read the ids of $O(\log^* n)$ of its ancestors in the tree. As discussed above, if while reading the *oids* p reaches the root of its tree, there are two possibilities. Either the operation v at the root had a parent w that finished, or it never had a parent. In the latter case, the **child** field will be marked **root**, and p extends its sequence with a default padding to the proper length. (This is the case in the first part of Figure 2. If operation p reads the data structure in this state, it finds **root** in the **child** field of location 21.) Otherwise, v once had a parent w, and before terminating, w wrote its ancestor chain to the **child** field, so that p will see the same *oid*'s as w and other earlier readers. (This is the case in the bottom part of Figure 2. If operation p reads the data structure in this state, it finds the string rs in the **child** field of location 9.) Then p concatenates this sequence to what it had previously read in the tree, and uses this combined sequence to compute its color.

Lock: In this phase, operation p uses the order decided on in the *decide* phase to lock its k locations. If any of these locations is already locked by a conflicting operation q, p helps q until q releases the lock, and tries to lock again. The 3-coloring ensures there are no monotonic chains of colors longer than 3. This ensures that in this phase there are no waiting chains longer than 4.

Execute: When operation p starts the *execute* phase, it holds locks for all k locations it requires, and so can update them one at a time, in (virtual) mutual exclusion. However, some care is required since p could be only one of several operations simultaneously trying to perform the same operation on these k locations. (As p might be either helping another operation, or p's operation might itself be helped by others.) The locks are held, not by p, but by the operation being executed by p (and possibly by other helpers). As with the entire helping technique, the atomic update is effectively interpreted, one step at a time. To synchronize, after each step, a step counter is recorded both in an operation record specific to this operation, and in the shared locations. This ensures that none of the component steps of the operation is performed more than once, and allows different processes to execute successive component steps on behalf of the same operation. Moreover, we assume that the output (returned to the calling process) of a k-object RMW operation consists of the k-tuple of location values just before they are updated. The output values of a particular k-object RMW operation may be needed by other helping processes, so at the beginning of the *execute* phase these values are copied (using RMW) from each of the k locations into the operation record.

Unlock & release: After finishing the *execute* phase, each operation unlocks the k locations and unmarks the **child** and **parent** fields.

Wait-free solution: The non-blocking algorithm is made wait-free by applying a modification of a standard technique [Her91, Her93, AM95a]:

> A process that properly terminates its operation, must help (some) pending operations before it is allowed to start a new operation.

To obtain an algorithm with local contention and step complexity, we apply this technique so that each terminating operation helps *only* those pending operations with which it directly conflicts. To do so, the operation must be able to detect the operations with which it directly conflicts. Hence, we associate with each location a record of the active operations on that location. After the initialization phase, operation p enters itself into the record associated with each of the k locations it needs. Then it performs the non-blocking algorithm, and after finishing it, process P that performed operation p removes operation p from these records and helps the other operations that entered these records. In the non-blocking algorithm, no operation p can be prevented from progressing unless other operations within a bounded distance repeatedly interfere–by marking needed **child** or **parent** fields, or by obtaining needed locks. If any operation p is blocked for too long, conflicting operations will all eventually find p in the corresponding record, until none can be blocking p any longer: the result is a wait-free algorithm.

As described, in making the algorithm wait-free, upon terminating the non-blocking phase, each operation p helps its immediate neighbors. However, by helping more operations at this point (to e.g. distance 2, or constant distance c), it is possible to prevent additional operations from interfering with p, and hence being helped, during the non-blocking portion of the algorithm. This requires a queue at each activation record. During the non-blocking algorithm, each neighbor of *oid* p that helps p enters itself into the queue associated with p's activation record. When p terminates, it uses this queue and the queues associated with the locations p updates, to determine its immediate neighbors, and transitively, to distance c.

Helping: A standard helping technique is used both in the basic non-blocking algorithm and in the wait-free portion. To coordinate among helping processes, in a pre-processing step each operation publishes its code in an operation record. The operation then is essentially interpreted, by both the original process, and any that help it. Accesses to the individual locations in the shared memory alternate with updates to the operation record's local variables and program counter. Careful record keeping in both the shared locations and the operation record ensure that no step of the operation is executed more than once on the shared memory [Bar93, IR94].

3.2 Results

In this subsection we state the main properties of the algorithm. Proofs are omitted from this extended abstract.

Within the *filter*, a process helps only operations that are either 1- or 2-neighbors of its operation. In the *decide* phase, an operation may access locations at distance up to $O(\log^* n)$ while reading the $O(\log^* n)$ operations in a chain toward the root of its tree. If at some point it reaches a root, it stops. However, it might also reach a node q which is not through the *filter*. (Process q has marked its **child** field but not its **parent**.) Process p helps q to either mark its **parent** field, or (the **parent** is already marked), to unmark the needed **child** field. During the *lock* phase, p may help other processes until they release their locks, which in turn may need to help other processes, etc. But by using the 3-coloring, helping chains are bounded by 4.

Thus, in any level of the recursion, a process may access common locations with other processes at distance at most $O(\log^* n)$ from it at that level. This explanation motivates the following:

Theorem 1 *The algorithm has $O(\log^* n)$-local contention complexity.*

The step complexity of an operation p is impacted by the cost of the recursion, and the number of distinct neighbors helped by p, either during the non-blocking portion of the algorithm, or during the wait-free construction, when it helps all neighbors within a constant distance c. The exact complexity depends on the number and distance of neighbors that are helped in the wait-free construction. Helping neighbors to a greater distance increases the cost of the wait-free helping, but prevents operations from interfering with p, and hence being helped, during the non-blocking portion of the algorithm. It can be shown that for a fixed k, the step complexity of any multi-object operation on k locations, is bounded by a function of the number of operations within distance $O(\log^* n)$ of it in its conflict graph. Hence:

Theorem 2 *The algorithm has $O(\log^* n)$-local step complexity.*

As already discussed, the wait-free mechanism assures wait-freedom by guaranteeing that no immediate neighbor of p can block p repeatedly. The non-blocking portion of the algorithm has several crucial properties: first, p can only be blocked by helping other operations, and no operation is helped twice. Second, p only helps operations within a bounded distance in the conflict graph.

Now suppose p is blocked indefinitely. Then by the first property, p must help infinitely many operations, and by the second property, these occur within a bounded distance in the conflict graph. Then there is a minimum distance from p in the conflict graph, d, in which infinitely many operations occur. Consider each of the (finitely many) nodes at distance $d-1$ from p. At least one, q, must have infinitely many neighbors—but then q will eventually be helped, and removed from the conflict graph after some finite time, a contradiction. Hence, we have:

Theorem 3 *The algorithm is wait-free.*

3.3 Replacing large RMW with LL/SC

Some of the RMW objects used in our algorithm are rather large. Either $O(\log^* n)$ for each memory location, or larger for the wait-free mechanism.

The *individual update* method of Afek, Dauber and Touitou replaces each RMW object by several LL/SC objects, and can be used to transform this implementation into one that uses LL/SC operations on small words [ADT95]. (Words which hold $O(logn)$ bits or a constant number of *oid*'s.) The individual update method is one of several methodologies Afek, et al. describe to transform a sequential data structure implementation (in particular a large RMW) into a fast wait-free one using LL/SC. It ensures that every process completes its operation within $O(kf(s)\log(f(s))$ steps, where $f(s)$ is the step complexity of the sequential implementation, and k is the number of processes concurrently contending with the operation on the data structure. For that reason we state:

Corollary 4 *There exists a wait-free k-object implementation with $O(\log^* n)$-local step complexity and $O(\log^* n)$-local contention which uses only LL/SC primitives on words of length $O(logn)$.*

4 Discussion

We have presented a wait-free algorithm for the multi-object problem that exhibits local contention and step complexity: operations that access widely disjoint parts of a data structure, or are widely separated in time, do not interfere with each other.

We assume that the number of locations required by any *multi-object* operation is bounded by some fixed number k, and that the set of locations required by each *multi-object* operation is already known at the beginning of the operation. It would be interesting to find a solution, that has low local contention and local step complexity, which supports more general multi-object operations in which there is no bound on the number of locations which can be accessed atomically, and the specific locations can be chosen on the fly.

As we discussed, except for the wait-free or non-blocking fault-tolerance requirements, the multi-object problem is essentially the well-known resource allocation problem, and hence our algorithm can be used to solve both problems. It would be interesting to determine whether existing highly concurrent resource allocation algorithms (such as that due to Awerbuch and Saks [AS90]) can be modified to also produce efficient multi-object algorithms.

Finally, the complexity measures we study are local contention and local step complexity. It would be interesting to find algorithms (or lower bounds) narrowing the local neighborhoods—for example, are their algorithms with d-local contention, where d is a constant? Also, it would be interesting to find algorithms which have $O(log^* n)$-local step complexity, in which the step complexity bound is smaller. For example, is it possible to save steps by unwrapping the recursive construction? Or can the algorithm be optimized for the special case of $k = 3$?

References

[AD96] H. Attiya and E. Dagan. Universal operations: Unary versus binary. In *Proc. 15th ACM Symp. on Principles of Distributed Computing,* pages 223–232, 1996.

[ADT95] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proc. 27th ACM Symposium on Theory of Computing,* pages 538–547, May 1995.

[AM94] J. Anderson and M. Moir. Using k-exclusion to implement resilient, scalable shared objects. *Proc. of the 13th ACM Symposium on Principles of Distributed Computing,* pages 141–150, August 1994.

[AM95a] J. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pages 184–193, August 1995.

[AM95b] J. Anderson and M. Moir. Universal construction of large objects. *Proceedings of the 9th International Workshop on Distributed Algorithms*. September 95.

[AMT96] Y. Afek and M. Merritt and G. Taubenfeld. The power of multi-objects. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 213–222, May 1996.

[AS90] B. Awerbuch and M. Saks. A dining philosophers algorithm with polynomial response time. In *Proc. 31th IEEE Symp. on Foundations of Computer Science*, pages 65–74, October 1990.

[AT93] R. Alur and G. Taubenfeld. How to share an object: A fast timing-based solution. In *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, pages 470–477, December 1993.

[Bar93] G. Barnes. A Method for implementing lock-free shared data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[CM84] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. on Programming Languages and Systems*, 6, 1984.

[CS95] M. Choy and A. K. Singh. Efficient fault tolerant algorithms for resource allocation in distributed systems. *ACM Trans. on Programming Languages and Systems*, 17(3):535–559, May 1995.

[CS96] M. Choy and A. K. Singh. Localizing failures in distributed synchronization. *IEEE Transactions on parallel and distributed systems*, 7(7):705–716, July 1996.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1) 32–53, July 1986.

[Dij72] E. W. Dijkstra. Hierarchical ordering of sequential processes. In *Operating Systems Techniques*, 1972. Eds: C. A. R. Hoare and R. H. Perrott, Eds. Academic Press.

[DHW93] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. In *Proc. 25th Ann. ACM Symp. on Theory of Computing*, New York, pages 174–183, 1993.

[GC96] M. Greenwald and D. R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proc. 2nd ACM Symp. on Operating System Design and Implementation. USENIX*, Seattle, pages 123–136, October, 1996.

[GPS87] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon Parallel symmetry-breaking in sparse graphs. *Proc. 19th Ann. ACM Symp. on Theory of Computing*, New York, pages 315–324, 1987.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 11(1):124–149, January 1991.

[Her93] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems* 15(5): 745-770, November 1993.

[IR93] A. Israeli and L. Rappoport. Efficient wait free implementation of a concurrent priority queue. In *Workshop on Distributed Algorithms on Graphs 1993. Lecture Notes in Computer Science 725, Springer Verlag*, pages 1-17.

[IR94] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory. In *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pages 151–160, 1994.

[JT92] P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. *Proc. of the 6th Int. Workshop on Distributed Algorithms: Lecture Notes in Computer Science, 647*, pages 69–84. Springer Verlag, November 1992.

[Lyn81] N. A. Lynch. Upper bounds for static resource allocation in a distributed systems. *Journal of Computer and System Sciences*, 23:254–278, 1981.

[MT94] M. Merritt and G. Taubenfeld. Atomic m-register operations. *Distributed Computing*, 7:213–221, 1994. Also appeared in *Proc. 5th Int. Workshop on Distributed Algorithms*, 1991, pages 289–294.

[Plo88] S. A. Plotkin. *Chapter 4: Sticky Bits and Universality of Consensus*. PhD thesis, M.I.T., August 1988.

[Plo89] S. A. Plotkin. Sticky bits and universality of consensus. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 159–175, August 1989.

[SP88] E. Styer and G. L. Peterson. Improved algorithms for distributed resource allocation. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 105–116, August 1988.

[ST95] N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pages 204–213, January 1995.

[TSP92] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock based concurrent data structure algorithms non-blocking. In *Proceedings of the 1992 Conference on the Principles of Database Systems*, pages 212-222, 1992.