Towards Load Balanced Distributed Transactional Memory

Gokarna Sharma and Costas Busch

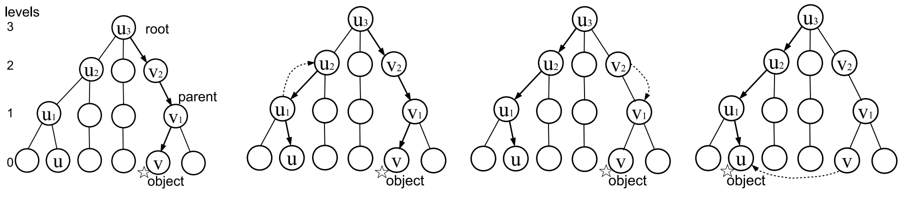
Department of Computer Science, Louisiana State University
Baton Rouge, LA 70803, USA
{gokarna,busch}@csc.lsu.edu

Abstract. We consider the problem of implementing transactional memory in d-dimensional mesh networks. We present and analyze MultiBend, a novel load balanced directory-based protocol, which is designed for the *data-flow* distributed implementation of software transactional memory. It supports three basic operations, *publish*, *lookup*, and *move*, on a shared object. A pleasing aspect of MultiBend is that it is load balanced (minimizes maximum node and edge utilization) which is achieved by using paths of multiple bends in the mesh. This protocol guarantees an $\mathcal{O}(d^2 \log n)$ approximation for the load and also for the distance stretch of *move* requests, where n is the number of nodes in the network. For fixed d, both the load and the *move* stretch are optimal within a constant and a loglog factor, respectively. It also guarantees $\mathcal{O}(d^2)$ approximation for *lookup* requests which is optimal within a constant factor for fixed d. To the best of our knowledge, this is the first distributed directory protocol that is load balanced.

1 Introduction

In distributed networked systems processors are the nodes of a network which communicate through a message passing environment. We assume that there is a shared memory address space, which is equally split among the processors. Each processor has its own *cache*, where copies of objects reside. In Transactional Memory (TM) [10,9,16,8,12] a *transaction* represents a sequence of shared memory operations (i.e., reads and writes) that are all performed atomically. The individual entries at the shared memory, called *objects*, can be shared by multiple transactions on different network nodes. A transaction can either *commit* (i.e., take effect) or *abort* (i.e., have no effect at all). If a transaction aborts, it is typically restarted until it commits. When a transaction running at a processor node issues a read or write operation for a shared memory location, the data object at that location is loaded into the processor-local cache.

We consider the *data-flow distributed implementation of software transactional memory* (DTM) suggested by Herlihy and Sun [11], where transactions are immobile (i.e., running at some particular node) and shared objects are moved to those nodes that need them. In DTM, transactions can only operate on local shared objects and, if remote shared objects are required, the transaction must communicate with one or more remote processor nodes. Some distributed *cache-coherence* mechanism should ensure that shared objects remain *consistent*, i.e., writing to an object automatically *locates* and *invalidates* other cached copies of that object. A DTM protocol typically supports



(a) Initially, owner v publishes the object **(b)** The request continues up phase **(c)** The request continues down phase **(d)** Object is moved directly from v to u

Fig. 1. Illustration of MultiBend for a *move* request issued by node u for the object at node v, where the nodes shown are leader nodes of the respective sub-meshes

three kinds of operations: (i) *publish* operation which allows a node which created an object in its memory space to publish it so that other nodes in the network can find it; (ii) *lookup* operation, the protocol should locate the current copy of the object and move it to the requesting node's cache (*shared* access), without modifying the old copy; (iii) *move* operation, where a transaction attempts to access an object to update explicitly the DTM protocol should locate the current cached copy of the object and move it to the requesting node's cache invalidating the old copy. [3,17] also studied DTM.

Typically the performance of a DTM protocol is measured with respect to the communication cost, which is the total number of messages sent in the network. The communication cost for an operation (resp. for a set of operations) is compared to the optimal communication cost for that operation (resp. for that set of operations) to provide an approximation ratio, which is generally referred to as *stretch*. In the context of DTM, previous approaches [6,17,3,11,15] focused only on stretch bounds for various network topologies (Table 1 summarizes their properties) and they do not control the congestion. The network congestion can also affect the overall performance of the algorithm and sometimes it is a major bottleneck. We measure the network congestion as the worst node or edge utilization (the maximum number of times the object requests use any edge or node in the network while accessing the shared object).

Contributions. We present MultiBend, a DTM protocol that is suitable for d-dimensional mesh networks and is load balanced in the sense that it minimizes the congestion (maximum node and edge utilization), and at the same time maintains low stretch. Mesh networks are appealing due to their use in parallel, distributed, and high-performance computing. The low stretch is achieved through a novel labeled hierarchical directory-based approach which we first introduced in [15] for general networks and we adapted it here appropriately for the mesh network. The load balancing is achieved through an *oblivious routing* approach (e.g., [13,4,5]) for communication between different hierarchy level leader nodes. In particular, we use the oblivious routing algorithm in Busch *et al.* [5] that gives near optimal congestion while maintaining small path length stretch in the mesh networks.

For the performance analysis of MultiBend, we consider sequential and concurrent execution of requests. For the *move* operations in both the cases, MultiBend guarantees $\mathcal{O}(d^2 \log n)$ amortized stretch and $\mathcal{O}(d^2 \log n)$ approximation of the optimal congestion on any node or any edge in d-dimensional mesh networks, where n is the number

Table 1. Comparison of DTM protocols, where n, S_*, and D, respectively, are the number of nodes, stretch, and the diameter of the network kind on which they operate

Protocol	**Stretch**	**Network Kind**	**Load Balanced**	**Runs On**
Arrow [6]	$\mathcal{O}(S_{ST}) = \mathcal{O}(D)$	General	No	Spanning tree
Relay [17]	$\mathcal{O}(S_{ST}) = \mathcal{O}(D)$	General	No	Spanning tree
Combine [3]	$\mathcal{O}(S_{OT}) = \mathcal{O}(D)$	General	No	Overlay tree
Ballistic [11]	$\mathcal{O}(\log D)$	Constant-doubling	No	Hierarchical directory with independent sets
Spiral [15]	$\mathcal{O}(\log^2 n \cdot \log D)$	General	No	Hierarchical directory with sparse covers
MultiBend (this paper)	$\mathcal{O}(\log n)$	2-D mesh	Yes	Hierarchical decomposition of the mesh
	$\mathcal{O}(d^2 \log n)$	d-D mesh		

of nodes in the mesh. For fixed d, the *move* stretch is optimal within a loglog factor comparing to the $\Omega(\log n / \log\log n)$ lower bound by Alon *et al.* [1]; the congestion approximation is also optimal within a constant factor in light of the $\Omega(\frac{C^*}{d} \log n)$ lower bound on the approximation ratio of an oblivious algorithm due to Maggs *et al.* [13]. The communication cost of the *publish* operation is proportional to the diameter of the network (i.e., $\mathcal{O}(n)$) and it is a fixed initial cost which is only considered once and compensated by the costs of the *move* (or *lookup*) operations which are issued thereafter. Note that *lookup* operations have always $\mathcal{O}(d^2)$ stretch even when considered individually while their overall congestion is $\mathcal{O}(d^2 \log n)$ approximation in the d-dimensional mesh. To the best of our knowledge, this is the first DTM protocol that achieves low stretch in a load balanced way. It has been shown that the stretch and the congestion cannot be controlled simultaneously in general networks [5].

Techniques. For simplicity, consider an 2-dimensional $n = m \times m$ mesh network and one shared object; the general case for d-dimensional mesh is given in Section 6. (We consider transactions with only one shared object which is typical in the DTM literature [3,11,17,6,15]. A protocol for one object can be generalized to accommodate transactions with multiple objects by appropriately replicating that protocol in such a way that the replication avoids livelock of transactions.) MultiBend is a directory-based consistency protocol implemented on a hierarchy of sub-meshes (as clusters). There are $k + 1 = \mathcal{O}(\log n)$ levels such that side lengths of the sub-meshes increase by a factor of 2 between two consecutive levels. In each sub-mesh one node is chosen to act as a leader to communicate with different level sub-meshes. At the bottom level (level 0) each sub-mesh consists of individual nodes, while at the top level (level k) there is a single sub-mesh for the whole graph with a special leader node called *root*. The hierarchy forms a tree of leaders such that higher level leaders have as children the lower level leaders. Only the bottom level nodes can issue requests (*publish*, *lookup*, and *move*) for the shared object, while the nodes in higher levels are used to propagate the requests in the graph. (The difference between Spiral [15] and MultiBend is that Spiral uses sparse covers as clusters while MultiBend uses sub-meshes as clusters.)

The protocol maintains a *directory path* which is directed from the root to the bottom-level node that owns the shared object. The directory path is updated whenever the object moves from one node to another. As soon as the object is created by some bottom level node, it publishes the object by visiting its sequence of increasingly higher level leaders path towards the root, making each parent pointing to its child leader (Fig. 1a). These leader pointers correspond to path segment between the leaders and the concatenation of these path segments form the initial directory path. A *move* request from a node is served by following leader ancestors of that node, setting downward links toward it until it intersects the directory path to the owner node, and resetting the directory path it follows while descending towards the owner (Figs. 1b−1c); the directory path now points to the requesting node. As soon as the *move* reaches the owner, the object is forwarded to the requester along some shortest path in the mesh (Fig. 1d). A *lookup* operation is served similar to *move* without modifying the directory path.

In order to route the request between two consecutive leaders, we use *multi-bend* paths. In the oblivious routing algorithm of [5] they use a one-bend path between pairs of randomly selected nodes in the mesh. A one-bend path consists of two straight lines, one line in each dimension which meet at a corner where the bend occurs. Following [5], we use at most two-bend paths between leaders. The one-bend path is sufficient when the parent sub-mesh completely contains the child sub-mesh (they are at different level). There is an attribute in our algorithm where every level has actually two sub-levels with possibly the same side length sub-meshes (at least one same side length). For this a two-bend path is needed between the leaders of the same level sub-meshes.

The concatenation of the one-bend or two-bend paths form multi-bend paths. In order to obtain low congestion, every time we access the leader node of the sub-mesh we immediately replace it with another leader chosen uniformly at random among the nodes in the sub-mesh. The directory is then updated appropriately with the new leader information by updating the parent and children leaders. We note that the update cost is low in comparison to the cost of serving the requests because only the information in the nearby region needs to be updated due to the new leader. We argue that this step is necessary to control the congestion. This is because when a fixed leader is used, the node congestion on that leader is proportional to the number of requests that visit that leader. Moreover, in the fixed leader case, edge congestion can also be proportional to the number of requests as all the requests use fixed edges along the shortest path between two subsequent leaders. We also note that, using this random leader approach, if the congestion requirement on edges (or nodes) can be relaxed by the factor of κ, then leader change is only needed after every κ requests.

Outline of Paper. We proceed with network model and preliminaries in Section 2. In Section 3, we give hierarchy construction for the 2-dimensional mesh. We present MultiBend protocol in Section 4 and analyze it in Section 5. In Section 6, we extend MultiBend for the d-dimensional mesh. (Many proofs and details are omitted due to space restrictions.)

2 Network Model and Preliminaries

We begin with some necessary definitions which are adapted from [5,15]. We represent a distributed network as a d-dimensional mesh. The d-dimensional mesh $M = (V, E)$ is a d-dimensional grid of nodes (network machines) V, where $|V| = n$, with side length m_i in each dimension such that $n = \prod_{i=1}^{d} m_i$, and edges (interconnection links between machines) $E \subseteq V \times V$. Each computing node $u \in V$ is connected with each of its $2d$ neighbors (except the nodes at the boundaries of the mesh). We denote by $|E|$ the number of edges in M. A path p in M is a sequence of nodes with respective sequence of edges connecting the nodes, such that the length of the path p, denoted $\text{length}(p)$, is the number of edges it uses. A *sub-path* of p is any path obtained by a subsequence of consecutive edges in p; we may also refer to a sub-path as a *fragment* of p. Let $\text{dist}(u, v)$ denote the shortest path length (distance) between nodes u and v.

Consider a routing problem Π defined as a set of pairs of source and destination nodes. A routing algorithm for Π provides paths from every source to its respective destination. An algorithm is *oblivious* if the path choice for each pair of source destination is independent of the path choices of any other pair. The edge congestion C for any set of paths is the maximum congestion on any edge (link) of the network. Let C^* denote the optimal congestion attainable by any routing algorithm. We have symmetric definitions for node congestion. For a sub-mesh $M' \subseteq M$ (i.e., M' is any mesh that contains inside M), let $\text{out}(M')$ denote the number of edges at the boundary of M', which connect nodes in M' with nodes outside M'. For routing problem Π, we define the *boundary congestion* as follows. Consider some sub-mesh M' of the network M. Let Π' denote the messages (pairs of sources and destinations) in Π which have either their source or destination in M', but not both. All the messages in Π' will cross the boundary of M'. The paths of these messages will cause congestion at least $|\Pi'|/\text{out}(M')$. Define the boundary congestion of M' to be $B(M', \Pi) = |\Pi'|/\text{out}(M')$. For problem Π, the boundary congestion $B = \max_{M' \subseteq M} B(M', \Pi)$. Clearly, $C^* \geq B$.

We bound the stretch of the MultiBend protocol, which is the ratio of the total communication cost for a request (or for a set of requests) to the optimal cost for that operation (or for that set of requests). The congestion is the maximum number of times any node or edge is used by the object requests. We assume that M represents a network in which nodes do not crash, it implements FIFO communication between nodes (i.e. no overtaking of messages occurs), and messages are not lost. We also assume that, upon receiving a message, a node is able to perform a local computation and send a message in a single atomic step. *TM memory proxy* module [11] at each node provides interfaces both to the transactions at that node and to the proxies at other nodes on how to publish and access shared objects (details in [11,15]). The conflicts, if any, between a local transaction and a transaction running in some other node, in accessing the object, is resolved using well-known contention managers, e.g., [7,2,14].

3 Hierarchical Directory for the 2-Dimensional Mesh

We describe how to represent the 2-dimensional mesh M with equal side lengths $m = 2^k, k \geq 0$, as a hierarchy of sub-meshes (we discuss the d-dimensional case later in Section 6). We decompose M into two types of sub-meshes, type-1 and type-2 (see Fig. 2), as given below, adapting some notions from [5].

– *Type-1 sub-meshes.* There are $k+1$ levels of type-1 sub-meshes, $\ell = 0, 1, \cdots, k$. The mesh M itself is the only level k sub-mesh. Every level ℓ sub-mesh can be partitioned into 4 sub-meshes by dividing each side by 2. Each resulting sub-mesh is a

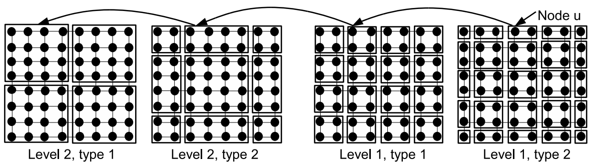


Fig. 2. Decomposition for the $2^3 \times 2^3$ 2-dimensional mesh. The arrows show the parent sub-meshes of a node u in its multi-bend path towards the root at level 3.

type-1 sub-mesh at level $\ell - 1$. According to this decomposition, at level ℓ, there are $2^{2\ell}$ sub-meshes each with side length $m_\ell = 2^{k-\ell}$. Note that the level 0 sub-meshes are the individual nodes of the mesh.

– *Type-2 sub-meshes.* There are $k-1$ levels of type-2 sub-meshes, $\ell = 1, \cdots, k-1$. The type-2 sub-meshes at level ℓ are obtained taking the type-1 sub-meshes and shifting them by $m_\ell/2$ simultaneously in both dimensions. Some of the shifted sub-meshes may have to be truncated at the borders of M.

We assign integer sub-levels to different type sub-meshes at each level. As there are only two types of sub-meshes at any level $0 < \ell < k$, we assign sub-level 1 to type-2 and sub-level 2 to type-1 sub-meshes (Fig. 2). For levels 0 and k we have only one sub-level as there are only type-1 sub-meshes. (i, j) denotes the level i sub-level j.

We now define a hierarchy of leveled sub-meshes. The sub-mesh hierarchy $\mathcal{Z} = \{Z_0, Z_1, \ldots, Z_k\}$, is a hierarchy of $k+1$ levels of sub-meshes such that: (i) At level k all nodes in M belong to exactly one sub-mesh, i.e., mesh M itself is the only level k sub-mesh; (ii) At level 0 each node in M is the one sub-mesh by itself; and (iii) In any level i, $1 \le i \le k-1$, Z_i contains type-1 and type-2 sub-meshes of level i.

Multi-bend Paths. We define a path $p(u)$ for each node $u \in V$ which is a "multi-bend" path of u. The path $p(u)$ is built by visiting the leader nodes in all the sub-meshes that u belongs to starting from level 0 up to k. In each level, the sub-meshes are visited according to the order of their sub-levels. From an abstract point of view, the path bends (changes dimensions) multiple times while it visits sub-mesh leaders of higher levels.

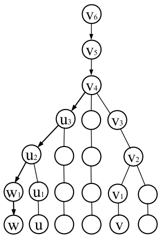
In every sub-mesh X we choose a *leader* node arbitrarily at the initialization of $\mathcal{Z}$ which we denote as $\ell(X)$. If one node is the leader on many sub-level sub-meshes, we add a virtual copy node of it and create a virtual link between the virtual copy and y itself in subsequent sub-meshes. Denote the leader of sub-level (i, j) sub-mesh $X_{i,j}(u)$ as $\ell_{i,j}(u) = \ell(X_{i,j}(u))$. Since the top most Z_k consists of a single sub-level it has a unique leader which we denote $\ell_{k,0}(u) = r$ (the root). Trivially, every node $u \in V$ is a leader of its own sub-mesh at level 0, $\ell_{0,1}(u) = u$. Note that $\ell(X)$ is changed for every request by electing a new leader uniformly at random among the nodes of X. This step is necessary to minimize the congestion among the nodes and edges.

For any pair of nodes $u, v \in V$, let $s(u, v)$ denote a dimension-by-dimension (i.e., change in path from one dimension to other dimension in every bend) shortest path (an at most two-bend path) from u to v. For any set of nodes $u_1, u_2, \ldots, u_f \in V$, let $s(u_1, u_2, \ldots, u_f)$ denote the concatenation of shortest paths $s(u_1, u_2)$, $s(u_2, u_3), \ldots,$ $s(u_{f-1}, u_f)$. Formally, the multi-bend path of node u is: $p(u) = s(u, \ell_{1,1}(u), \ell_{1,2}(u), \ldots, \ell_{k-1,1}(u), \ell_{k-1,2}(u), r)$.

We say that two multi-bend paths *intersect* if they have a common node. We also say that two multi-bend paths intersect at level i if they visit the same leader node at level i (they may intersect outside leaders but we do not consider that). Therefore,

Lemma 1 (Sharma *et al.* [15]). *For any two nodes $u, v \in V$, their multi-bend paths $p(u)$ and $p(v)$ intersect at level at most* $\min\{k, \lceil \log(\text{dist}(u, v)) \rceil + 1\}$.

Canonical Paths. We need later paths obtained from fragments of multi-bend paths; the fragments are formed while the object moves. These paths start at level 0 and may go up to the root. We will refer to such paths as *canonical.* As shown in the figure on the right, the newly formed path from w to v_6 is a canonical path that obtained from the fragment of $p(w)$ from w to u_2, the fragment of $p(u)$ from u_2 to v_4, and the fragment of $p(v)$ from v_4 to v_6. Formally, a canonical path q up to sub-level $(\alpha, \beta) \leq (k, 1)$ is $q = s(x_{0,2}, x_{1,1}, x_{1,2}, x_{2,1}, x_{2,2}, \ldots, x_{\alpha,\beta})$, such that $x_{i,j}$'s are leader nodes along the path. A canonical path can be either *partial* when the top node is below level k (below the root), or *full* when the top node is the root. A multi-bend path $p(u)$ is a full canonical path. Any prefix of a multi-bend path is a partial canonical path. The path q up to level α is the concatenation of paths constructed by the 2 bend dimension to dimension paths in sub-meshes of (at least one) sides $2^1, 2^2, \cdots, 2^\alpha$, which sums at most $\text{length}(q) \leq 2^{\alpha+3}$. Thus,



Lemma 2. *For any canonical path q up to level α,* $\text{length}(q) \leq 2^{\alpha+3}$.

4 The MultiBend Algorithm

We present the MultiBend protocol (Algorithm 1) which implements a DTM for shared objects over a 2-dimensional mesh graph M. Consider some shared object ξ. The protocol guarantees that any moment of time only one node holds the shared object ξ which is the *owner* of the object. The owner is the only node who can modify the object (write the object); the other nodes can only access the object for read.

The basic idea is to maintain a *directory path* in a sub-mesh hierarchy $\mathcal{Z}$, which is a directed path from the root node r to the bottom-level node that currently owns the shared object ξ. Initially, the directory path is formed from the multi-bend path $p(v)$ of the creator node v when it issues the *publish*(ξ) operation by assigning pointers along the edges of $p(v)$ directed toward v (Fig. 1a shows hierarchy $\mathcal{Z}$ after a successful publish operation). The pseudo-code for *publish* is given in Lines 1–2 of Algorithm 1.

We define the notion of parent node before giving details of *lookup* and *move*. We denote *parent* node y of a node x in the multi-bend path $p(u)$ as $y = \text{parent}_{p(u)}(x)$, i.e., if y is the sub-level (i, j) sub-mesh leader in $p(u)$ then x is the leader of the immediate lower sub-level sub-mesh leader. Note that the leader of a level 0 sub-mesh is the node itself. Each node knows its parent in the hierarchy, except the root, whose parent is $\perp$ (null). A node might have a $link$ (a downward pointer) towards one of its children (otherwise $\perp$); the $link$ at the root is not $\perp$.

Lets assume a *lookup*(ξ) and a *move*(ξ) operation issued by a node u. Both operations are implemented in two phases: (i) in the *up phase*, a request message is sent from u

Algorithm 1. MultiBend protocol

```
1: When $y$ receives $m = \langle v, up, publish\rangle$ from $x$:        // Publish operation
2:    set $y.link = x$; if $y$ is not a root node then send $m$ to $\text{parent}_{p(v)}(y)$;
3: When $y$ receives $m = \langle u, phase, lookup\rangle$ from $x$:      // Lookup operation
4:    if $m = \langle u, up, lookup\rangle$ then                          // lookup up phase
5:       if $y.link = \perp$ then
6:          if $y.slink$ list is empty then
7:             elect a leader $w$ at sub-mesh containing $\text{parent}_{p(u)}(y)$; send $m$ to $w$;
8:          else elect a leader $w$ at sub-mesh containing first pointer of $y.slink$ list; send
               $\langle u, down, lookup\rangle$ to $w$;
9:       else elect a leader $w$ at sub-mesh containing $y.link$; send $\langle u, down, lookup\rangle$ to $w$;
10:   if $m = \langle u, down, lookup\rangle$ then                        // lookup down phase
11:      if $y$ is a leaf node then
12:         send the read-only copy of $\xi$ to $u$ and remember $u$;
13:      else elect a leader $w$ at sub-mesh containing $y.link$; send $m$ to $w$;
14: When $y$ receives $m = \langle u, phase, move\rangle$ from $x$:       // Move operation
15:   if $m = \langle u, up, move\rangle$ then                            // move up phase
16:      assign $oldlink \leftarrow y.link$ and set $y.link = x$;
17:      add $y$ in $slink$ list of $y$'s special patent;
18:      if $oldlink = \perp$ then
19:         elect a leader $w$ at sub-mesh containing $\text{parent}_{p(u)}(y)$; send $m$ to $w$;
20:      else send $\langle u, down, move\rangle$ to $oldlink$;
21:   if $m = \langle u, down, move\rangle$ then                          // move down phase
22:      if $y$ is in the $slink$ list then erase $y$ from $slink$;
23:      if $y$ is not a leaf node then $oldlink \leftarrow y.link$; $y.link \leftarrow \perp$; send $m$ to $oldlink$;
24:      else send the writable copy of $\xi$ to $u$;
25:      invalidate($\xi$) from the owner node $v$ and the read-only copies from other nodes;
26: Leader election procedure:
27:   select a node $w$ in the sub-mesh containing leader $z$ uniformly at random;
28:   copy information at old leader $z$ to new leader $w$;
29:   inform the parent and child of $z$ about the new leader $w$;
30:   construct a sub-path $p_i$ from $w_{i-1}$ to $w$ by picking a dimension by dimension
      shortest path (where the sub-path is either one-bend or two-bend);
```

upward in the hierarchy $\mathcal{Z}$ along the multi-bend path $p(u)$ towards the root r until the request intersects at a node (i.e. node x) with the directory path; (ii) in the *down phase*, the request message follows the directory path from node x to the object owner; then the owner sends a copy of ξ to u (along some shortest path in M). For the *lookup* it is a read-only copy of ξ without modifying the hierarchy (see Lines 3–13 of Algorithm 1). But, for the *move* it is a writable copy invalidating the old copy of ξ and modifying the directory path (Figs. 1b−1d). In the up phase, the *move* sets the directions of the edges in the fragment of $p(u)$ between u and x to point toward u. In the down phase it deletes the downward pointers (or links) in the fragment of the directory path from x to v. Now the new directory path points toward u. When the down phase reaches v, u obtains a copy of the object (see Lines 14–25 of Algorithm 1). This process has resulted to a canonical directory path that consists of two multi-bend path fragments, a fragment of u's multi-bend path between r and x and a fragment of v's multi-bend path between x and u. Subsequent *move* operations may result into further fragmentation of the directory path into multiple (more than two) multi-bend path fragments.

Need of Special Parent. A *lookup* request may not find immediately the directory path to ξ, even if the *lookup* originates near the owner node of ξ. This is because after several *move* operations the directory path may become highly fragmented. The notion of a *special parent* node helps to avoid this situation and guarantee efficient *lookup*s, such that whenever a downward link is formed at a node z the special parent of z is also informed about z holding a downward pointer. A *special-parent* node of y, denoted as $\text{sparent}_{p(u)}(y)$, at sub-level (i, j) in the multi-bend path $p(u)$ is the leader node of one of the sub-meshes $X \in Z_\eta(u)$ at level η, where $\eta = i + 4$, i.e., $\text{sparent}_{p(u)}(y)$ is some *ancestor* node of y at level η in $p(u)$. Every node knows its special parent and has $slink$ (downward pointer) towards *special-child* node from its special-parent $\text{sparent}_{p(u)}(y)$ (otherwise it is $\perp$). We maintain a list of $slink$ pointers if one node is the special parent for several sub-meshes. The special parent is selected in such a way that any nearby *lookup*, close to z will either reach z or its special parent. We will prove that *lookup*s are always efficient using special parents (see Lines 6,8,17, and 22 of Algorithm 1).

Load Balancing. MultiBend (Algorithm 1) uses a leader election procedure (Lines 26–30 of Algorithm 1) such that *lookup* and *move* requests can be served in a load balanced way. The procedure works as follows: Let z be a leader node of the sub-mesh M' in $\mathcal{Z}$. We elect a new leader at M' by selecting a node $w \in M'$ uniformly at random. After the leader is elected, the information at old leader z is moved to new leader w and the parent and child of z are informed about the new leader w. The pointers inside M' are also updated to point to the new leader. After that, sub-path p_i from w_{i-1} (a leader of the sub-mesh that is sending a message to M') to w is formed by picking a dimension by dimension shortest path; the sub-path p_i is one-bend if sub-mesh containing w and the sub-mesh containing w_{i-1} are both type-1 sub-meshes, otherwise, p_i is of at most two bend path. If the sub-path is the two-bend path then it is picked by a random ordering of dimensions on a random node. The *lookup* uses this procedure in Lines 7,8,9, and 13 of Algorithm 1. The *move* invokes it at Line 19 of Algorithm 1.

We observe that at any time a request locks at most three nodes (level $\text{prev}(i, j)$, (i, j), and $\text{next}(i, j)$) along the multi-bend path or a downward path. In concurrent situations this may be a problem; but we describe how we handle concurrent requests efficiently later in Sections 5 and 6. Note also that the special parent node doesn't need to be locked because only one specific $slink$ pointer value needs to be updated.

5 Performance

We give the stretch and congestion analysis of MultiBend for sequential executions; the stretch and congestion analysis for concurrent executions is deferred to the full paper due to space limitations. Moreover, the correctness proof is omitted as it can be extended from [3,11,15].

Move Cost. We now give the analysis of MultiBend in sequential executions. Lets define a sequential execution of a set $\mathcal{E}$ of $l+1$ object ξ requests $\mathcal{E} = \{r_0, r_1, \cdots, r_l\}$, where r_0 is the initial *publish* request and the rest are the subsequent *move* requests.

For the sake of analysis, similar as in [15], we define a two-dimensional array B of size $(k+1) \times (l+1)$, where $k+1$ and $l+1$ are the number rows and columns, respectively. The $k+1$ rows of B can be denoted as $\{row_0, row_1, \cdots, row_k\}$, and the $l+1$ columns of B can be denoted as $\{col_0, col_1, \cdots, col_l\}$. All the locations of the array are initially empty ($\perp$). We fix that $[0,0]$ is the lower left corner element and $[k,l]$ be the upper right corner element. The levels visited by each request r_i in the hierarchy $\mathcal{Z}$ while searching for the object are registered in each $col_i, 0 \leq i \leq l$. The maximum level reached by a request r_i in $\mathcal{Z}$ is called the *peak level* for that request. We have that $l \leq k$. The peak level reached by r_0 (the *publish* request) is always k and r_0 is registered at all the locations of col_0 starting from $col_0[0]$ to $col_0[k]$.

Let $A^*(\mathcal{E})$ denote the optimal cost for serving requests in $\mathcal{E}$ and $A(\mathcal{E})$ denote the total communication cost using the MultiBend. We will bound the stretch $\max_{\mathcal{E}} A(\mathcal{E})/A^*(\mathcal{E})$. For any $c, d, 0 \leq c < d \leq l$, a *valid pair* $W^j_{(c,d)}$ of two non-empty entries in $row_j, 0 \leq j \leq h$ is defined as $W^j_{(c,d)} = (row_j[c], row_j[d])$, such that $row_j[c] \neq \perp$ and $row_j[d] \neq \perp$, and $\forall e, c < e < d$, $row_j[e] = \perp$. In other words, $W^j_{(c,d)}$ is a pair of two subsequent non-empty entries in a row. Moreover, we denote by S_j the total count of the number of entries $row_j[i], 0 \leq i \leq l$, such that $row_j[i] \neq \perp$, and by W_j the total number of non-empty pairs ($W^j_{(c,d)}$) in it. We have that $W_j = S_j - 1$.

Theorem 1. *The move stretch of* MultiBend *is* $\mathcal{O}(\log n)$ *for sequential executions.*

Proof (sketch). Let $A^*_h(\mathcal{E})$ be the optimal communication cost for level h in the hierarchy $\mathcal{Z}$. According to the execution setup, S_h are the number of requests in $\mathcal{E}$ that reach level h, and W_h are the total number of valid pairs at that level. For any two subsequent requests that originate from nodes u and v and reach level h, $\text{dist}(u,v) \geq 2^{h-1}$ (according to Lemma 1), since otherwise their multi-bend paths would intersect at level $h-1$ or lower. Therefore $A^*_h(\mathcal{E}) \geq W_h 2^{h-1} \geq (S_h - 1)2^{h-1}$, as $W_h = S_h - 1$. Considering all the levels from 1 to k, $A^*(\mathcal{E}) \geq \max_{1 \leq h \leq k} A^*_h(\mathcal{E}) \geq \max_{1 \leq h \leq k}(S_h - 1)2^{h-1}$.

Similarly, let $A_h(\mathcal{E})$ be the total communication cost of MultiBend for all the requests in $\mathcal{E}$ that reach level h in the hierarchy $\mathcal{Z}$, while probing the shared object in their up phase. We have that $A_h(\mathcal{E}) \leq (S_h - 1)2^{h+3}$ (Lemma 2). By combining the cost for each level, $A(\mathcal{E}) = \sum_{h=1}^{k} A_h(\mathcal{E}) \leq \sum_{h=1}^{k}(S_h - 1)2^{h+3}$. We do not need to consider level 0 for $A^*(\mathcal{E})$ and $A(\mathcal{E})$ because there is no communication at that level.

Since the execution $\mathcal{E}$ is arbitrary and $\sum_{h=1}^{k}(S_h - 1)2^{h+3} \leq k \cdot \max_{1 \leq h \leq k}(S_h - 1)2^{h+3}$, $\max_{\mathcal{E}} A(\mathcal{E})/A^*(\mathcal{E}) \leq 16 \cdot k = \mathcal{O}(\log n)$, as $k = \lceil \log n \rceil + 1$. □

Congestion. We relate the congestion of the paths selected by MultiBend to the optimal congestion C^*. In particular, we prove the following theorem (this bound is valid for both *move* and *lookup* operations, as both do random leader change in the same way):

Theorem 2. MultiBend *achieves* $\mathcal{O}(\log n)$ *approximation on congestion w.h.p.*

Proof (sketch). Recall that every request to predecessor nodes are routed by MultiBend by selecting some paths. Precisely, these paths are the multi-bend paths. Let e denote an edge in the mesh graph M and $C(e)$ denote the load on e (the number of times the edge e is used by the paths of the requests). We bound the probability that some multi-bend path uses edge e. Consider the formation of a sub-path p_i from a sub-mesh M_1 to a sub-mesh M_2, such that $M_1 \subseteq M_2$ and e is a member of M_2. If M_1 is of type-1 then all of its sides are equal to m_ℓ, where ℓ is the level of M_1. Then the sub-path p_i uses edge e with probability at most $2/m_\ell$. Moreover, a one-bend sub-path is enough to route the request from M_1 to M_2.

Let P' be the set of paths that go from M_1 to M_2 (or vice-versa). Let $C'(e)$ denote the congestion that the messages P' cause on e. Using the similar argument as given in previous paragraph for an edge e, the upper bound in $C'(e)$, denoted as $E[C'(e)]$, is bounded by $E[C'(e)] \leq 2|P'|/m_\ell$. Moreover, from the definition of the boundary congestion $B \geq B(M_1, \Pi) \geq |P'|/\text{out}(M_1)$. Thus, $C^* \geq |P'|/\text{out}(M_1)$. Since M_1 has all sides of length m_ℓ nodes, $\text{out}(M_1) \leq 4m_\ell$. Therefore, $E[C'(e)] \leq 8C^*$. We charge this congestion to sub-mesh M_2. Between every sub-level $(i, 2)$ sub-meshes, $1 \leq i \leq k-1$, as M_1 of sub-level $(i, 2)$ is completely contained in M_2 of sub-level $(i+1, 2)$ and there are at most $k < \log n + 2$ levels, the expected congestion on edge e, denoted as $E[C(e)]$, is bounded by $E[C(e)] \leq 8C^*(\log n + 2)$.

According to our construction, there is one type-2 sub-mesh M_1' between every two type-1 sub-meshes M_1 and M_2 in the sub-mesh hierarchy. As the type-2 sub-mesh M_1' may not be the proper subset of M_2, the set of paths from M_1 to M_1' may go through four possible type-2 sub-meshes and they may bend at most two times before they reach to the leader node of M_2. This will increase the congestion by at most the factor of 4 between every two type-1 sub-meshes M_1 and M_2. Moreover, as we know only sub-meshes up to level $k < \log n + 2$ can contribute to the congestion on edge e and there are at most $(\log n + 2)$ levels of type-2 sub-meshes, $E[C(e)]$ increases by a constant factor only due to the type-2 sub-meshes. As every request selects its path independently of every other request (Lines 26–30 of Algorithm 1), using standard Chernoff bound, we obtain a concentration result on the congestion C. □

Publish Cost. We can prove the following theorem for any *publish* operation.

Theorem 3. *The publish operation has communication cost $\mathcal{O}(n)$.*

Lookup Cost. It can be shown that a lookup request r from w finds either the directory path to the owner v ($\text{dist}(w, v) \leq 2^i$) or a $slink$ to the directory path towards v at level at most η, where $\eta = i + 4$. Therefore, we obtain:

Theorem 4. *The stretch of* MultiBend *is constant for a lookup operation.*

6 Extension to the d-Dimensional Mesh

We outline the alternative decomposition that has $\mathcal{O}(d^2 \log n)$ approximation for both the path stretch and the congestion in d-dimensional mesh networks. The decomposition will have type-1 sub-meshes and other shifted sub-meshes. We set $\lambda = \max\{1, m_\ell/2^{\lceil \log d+1 \rceil}\}$, where m_ℓ is the side length of the level ℓ type-1 sub-mesh.

The type-1 sub-meshes are shifted by $(j-1)\lambda$ nodes in each dimension to get the type-j sub-meshes. According to this decomposition, there will be at most $2(d+1)$ different types of sub-meshes at any level. The hierarchy $\mathcal{Z}$ is formed similar to 2-dimensional mesh but now there will be $2d+1$ sub-levels. The multi-bend and canonical paths can also be defined similar to Section 3. We summarize the performance bounds below:

Theorem 5. *In d-dimensional mesh networks,* MultiBend *has $\mathcal{O}(d^2 \log n)$ amortized stretch for* move *operations and $\mathcal{O}(d^2 \log n)$ approximation on congestion w.h.p. Moreover, the* publish *operation has cost $\mathcal{O}(n)$ and the* lookup *operation has stretch $\mathcal{O}(d^2)$.*

References

1. Alon, N., Kalai, G., Ricklin, M., Stockmeyer, L.J.: Lower bounds on the competitive ratio for mobile user tracking and distributed job scheduling. Theor. Comput. Sci. 130(1), 175–201 (1994)
2. Ansari, M., Luján, M., Kotselidis, C., Jarvis, K., Kirkham, C., Watson, I.: Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In: HiPEAC. pp. 4–18 (2009)
3. Attiya, H., Gramoli, V., Milani, A.: A provably starvation-free distributed directory protocol. In: SSS. pp. 405–419 (2010)
4. Azar, Y., Cohen, E., Fiat, A., Kaplan, H., Räcke, H.: Optimal oblivious routing in polynomial time. J. Comput. Syst. Sci. 69(3), 383–394 (2004)
5. Busch, C., Magdon-Ismail, M., Xi, J.: Optimal oblivious path selection on the mesh. IEEE Trans. Comput. 57(5), 660–671 (2008)
6. Demmer, M.J., Herlihy, M.: The arrow distributed directory protocol. In: DISC. pp. 119–133 (1998)
7. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: PODC. pp. 258–264 (2005)
8. Hammond, L., Carlstrom, B.D., Wong, V., Chen, M., Kozyrakis, C., Olukotun, K.: Transactional coherence and consistency: Simplifying parallel hardware and software. IEEE Micro 24(6), 92–103 (2004)
9. Herlihy, M., Luchangco, V., Moir, M., Scherer, III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC. pp. 92–101 (2003)
10. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21(2), 289–300 (1993)
11. Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. Distrib. Comput. 20(3), 195–208 (2007)
12. Kotselidis, C., Ansari, M., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Distm: A software transactional memory framework for clusters. In: ICPP. pp. 51–58 (2008)
13. Maggs, B., auf der Heide, F.M., Voecking, B., Westermann, M.: Exploiting locality for data management in systems of limited bandwidth. In: FOCS. pp. 284–293 (1997)
14. Sharma, G., Busch, C.: A competitive analysis for balanced transactional memory workloads. Algorithmica 63(1-2), 296–322 (2012)
15. Sharma, G., Busch, C., Srinivasagopalan, S.: Distributed transactional memory for general networks. In: IPDPS, pp. 1045–1056 (2012)
16. Shavit, N., Touitou, D.: Software transactional memory. Distrib. Comput. 10(2), 99–116 (1997)
17. Zhang, B., Ravindran, B.: Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory. In: OPODIS. pp. 48–53 (2009)