# TuFast: A Lightweight Parallelization Library for Graph Analytics

Zechao Shang
*University of Chicago*
zcshang@cs.uchicago.edu

Jeffrey Xu Yu
Chinese University of Hong Kong
yu@se.cuhk.edu.hk

Zhiwei Zhang
Hong Kong Baptist University
cszwzhang@comp.hkbu.edu.hk

***Abstract*—Recently, there has been significant interest in large-scale graph analytics systems. However, most of the design efforts focus on accelerating graph analytics on giant graphs and/or in a distributed environment. Little attention focuses on the programmer usability perspective, which is critical on implementing ad-hoc analytics on moderate size graphs. In this paper, we present a lightweight transactional memory (TM) library TUFAST which provides easy-to-use primitives for the end-user to agilely develop fast shared memory graph parallelization on a multi-core server. TUFAST exploits recent CPU instructions set Hardware Transactional Memory (HTM), which has been available in off-the-shelf CPUs. HTM offers free transactional semantic but also suffers from capacity limitation. Our framework resolves the capacity challenge and efficiently utilizes HTM on graph parallelization by exploiting the graph degree information. Large scale graphs have a power-law degree distribution: a large proportion of the vertices with a small degree, fits in single HTM transactions; a small proportion of vertices with a big degree fits a pessimistic approach like locking; other vertices with a moderate degree can be processed with an optimistic approach with HTM acceleration. Our hybrid approach automatically adapts to the degree of graphs dynamically during the processing. The graph analytical jobs expressed via our library are straightforward and concise, and outperform state-of-the-art distributed and multi-core graph analytical systems by up to 4 orders of magnitude.**

## I. INTRODUCTION

Graphs have been widely used to depict complex relationships among different objects, and have been extensively used to manage, query, and analyze complex information. State-of-the-art multi-core graph processing systems (i.e. Ligra [1] and Galois [2]) and distributed graph processing systems (i.e. PowerGraph [3]) accelerate the graph analytical jobs by grouping (vertex) operations into batched operations: they ask the end-users to implement the graph analytical jobs in a *new programming paradigm* (i.e. vertex-centric operation in Pregel [4], *map/reduce* in Ligra and Galois, *gather/apply/scatter* in PowerGraph, sparse matrix-vector multiplication in GraphMat [5]), and then maximize the computing throughput by grouping the operations in the new paradigm into batched computations. Although the batched computations accelerate the computational time and may have better scalability, the lack of shared-memory accesses sacrifices user-friendliness : despite recent researches [6], [7] on automatically parallelize monotonic graph programs, it is usually a nontrivial job to transform the sequential graph algorithm into the new paradigm, or design a completely new efficient algorithm for the new paradigm. Thus, these graph processing systems incur a significant human-power overhead on designing, implementing, testing, debugging, and deploying ad-hoc graph analytical jobs.

In this paper, we introduce a lightweight and easy-to-use library TUFAST, which automatically parallelize graph analytical jobs in multi-core environments. TUFAST is based on the transactional memory (TM) semantic: the end-users implement a sequential program, and mark the region that shall behave like under sequential execution when being executed concurrently, replace read/write operations on shared variable by TM read/write operations, so TUFAST can ensure the correctness of parallelized graph jobs.[1] TUFAST's users are relieved from the error-prone manual implementations.

The core of TUFAST is a new hybrid TM scheduler. TM scheduler coordinates the TM operations, and enforces the correctness among transactional regions (*transactions* for short). We observed that the diversity of graph degrees causes the heterogeneity of transaction contention rate. Transactions with higher contention rates are more likely to be interfered by other concurrent transactions. Therefore we prevent data race by a more pessimistic approach and pay more beforehand overheads. Otherwise, we use a low-overhead approach. Based on this observation, we introduced a three-mode Hybrid TM (HyTM): the high-contention transactions are protected by locks; the medium-contention transactions are scheduled by optimistic transaction schedulers assisted by HTM; and the low-contention transactions are guarded by the HTM. By carefully routing transactions to different sub-schedulers, TUFAST ensures each transaction can achieve its best performance. Moreover, to adapt to the runtime fluctuation of contention rate caused by the graph job itself or execution environment, we monitor the dynamic contention rate and optimize TUFAST's performance by adjusting the performance-critical parameters.

We implemented several graph algorithms in TUFAST, and conducted experimental studies on real large-scale graphs. Through experimental results, we demonstrate TUFAST outperforms manually-implemented parallel graph algorithms that execute HTM tasks on both high- and low-degree vertices, state-of-the-art parallel graph processing systems without

[1]Specifically, TUFAST guarantees *serializability* semantic, which ensures the concurrent execution equals to a sequential execution. However, the end-users can impose appropriate scheduling or synchronization barrier to order the graph operations as needed. See Section III for more usage examples.

```
for iter = 1 to N
    parallel_for v : all vertices
        BEGIN(degree[v]) // a degree hint
        if READ(v, match[v]) == null
            for u : neighbor of v
                if READ(u, match[u]) == null
                    WRITE(v, match[v], u)
                    WRITE(u, match[u], v)
                    break
        COMMIT
```

Fig. 1: Parallelized maximal matching implementation (with TUFAST support)

| Function | Description |
|---|---|
| BEGIN(size) | Start with an optional hint SIZE |
| COMMIT() | Commit |
| ABORT() | Abort |
| READ(v,addr) | Read addr (related with vertex v) |
| WRITE(v,addr,val) | Write val to addr (related with vertex v) |

TABLE I: TUFAST graph TM functions

HTM, and state-of-the-art distributed graph processing systems. The superior performance of TUFAST does not only prove our hybrid strategy outperforms other HyTM and transaction schedulers, but also demonstrates graph systems that support direct in-place-update on shared memory are more flexible and efficient than batched synchronization graph systems.

**Organization:** We demonstrate the syntax of TM and the exemplar usage of TM-based graph programming in Section II. We present the preliminarily backgrounds, including an introduction to Intel HTM, graph properties, and transaction scheduler performance in Section III. We propose our efficient HyTM TUFAST in Section IV. We discuss related work in Section V and report experimental studies in Section VI. We conclude this paper in Section VII.

## II. TM-BASED GRAPH PROGRAMMING

**TM:** TUFAST provides transactional operations READ and WRITE, and transaction boundary operations BEGIN, COMMIT, ABORT. The developer calls READ and WRITE operations to read/write memory addresses, and calls BEGIN and COMMIT to indicate the beginning and ending of a transaction. We illustrate the usage of TUFAST API in Figure 1 by an example of greedy graph maximal matching. Each *matching attempt* tries to match one unmatched vertex with one of its unmatched neighbors. The algorithm executes matching attempts on all vertices in parallel. We implement matching attempts as transactions, and access shared read/write variables (macth[]) via transactional operations (READ and WRITE).

**Optional Size Hint:** To facilitate TUFAST's performance, the developer is suggested to annotate each transaction with a *size hint*, which is approximately the number of shared data that the transaction intends to visit. The optional size hint is provided to the BEGIN API at the beginning of the transaction (degree[v] in Figure 1). Such hints are non-binding and

```
// to be executed on every vertex v
if (currentRound % 4 == 0)
    if (match[v] == null)
        for u : neighbor of v
            send(destination = u, msg = v)
elseif (currentRound % 4 == 1)
    if (match[v] == null)
        select one incoming message (v, msg = u)
        send(destination = u, msg = v)
elseif (currentRound % 4 == 2)
    if (exists incoming message (v, msg = u))
        match[v] = u
        send(destination = u, msg = v)
elseif (currentRound % 4 == 3)
    if (exists incoming message (v, msg = u))
        match[v] = u
```

Fig. 2: Maximal matching implementation on vertex-centric paradigm. It requires a non-trivial transformation from the sequential algorithm to this "four-way handshake" implementation.

```
for v : all vertices
    d[v] = infinite
d[source] = 0
push source into Q
while Q is not empty
    v = poll(Q)
    BEGIN(degree[v])
    for u : neighbor of v
        if READ(d[u]) + dis[u][v] < READ(d[v])
            WRITE(d[v], READ(d[u]) + dis[u][v])
            if v is not in Q
                push v into Q
    COMMIT
```

Fig. 3: Implementation of shortest path algorithm Bellman-Ford (with a FIFO queue) or SPFA (with a priority queue) (with TUFAST support)

do not affect the correctness of TUFAST. Instead, we take this hint as a suggestion of the transaction's length and the transaction's contention rate. We schedule each transaction to sub-schedulers for the maximal performance according to the size hints (c.f. Section IV-D).

**Remarks on Usability:** Compared with other programming paradigms (i.e. *map/reduce*, *gather/apply/scatter*, *vertex-centric* programming, sparse matrix-vector multiplication), TM-based parallelization programming resembles the sequential programming by allowing the developer to freely read and write any shared variable at any time. Without any limitation posed by the programming paradigms, the programmers can implement parallel graph analytics as easy as implementing sequential ones. We illustrate the usability via two examples. The first example is maximal matching we discussed previously. TM-based implementation (Figure 1) naturally express the operation that *pair vertex $v$ with its neighbor $u$*. In comparison, the computing paradigms that disallow shared-variable accesses need to send asynchronous messages to match a vertex to its neighbor. A typical implementation in vertex-centric paradigm is illustrated in Figure 2. (Function

send sends messages between vertices.) The vertex-centric matching works in a *four-way handshake*: each unmatched vertex send matching requests to its neighbors; unmatched neighbor choose request to reply; two vertex confirms the match and write down the matching information in following two rounds. This implementation is not intuitive and requires non-trivial knowledge about the graph algorithms.

Other programming paradigms often aim at optimizing throughput by batching. Thus, they usually disallow certain fine-grained control in the computation workflow. Our second example is two shortest path algorithms, Bellman-Ford and SPFA [8]. These two algorithms are almost same: they iteratively execute "relaxing" operations on each vertices. The only difference between them is Bellman-Ford uses a round-robin scheduling, and SPFA prioritizes vertices that are closer to the sources. Computing paradigms such as BSP [4] cannot support prioritized scheduling without a considerable effort (e.g. PrIter [9]) because BSP groups operations in a batch and disallow fine-grained prioritization within a batch. In contrast, with TUFAST's transactional semantic, our implementation is merely a direct translation from pseudo-code from Wikipedia [8]. Since TUFAST ensures any transaction does not interfere other concurrent transactions, we do not need to worry about the data race, and can switch between two algorithm by switching between a FIFO queue and a priority queue.

## III. PRELIMINARY

**Intel Hardware TM:** HTM utilizes the cache coherence protocol, which has previously been inaccessible to end-users until wrapped as HTM instructions. Hardware transaction instruction set was introduced in the Intel Haswell CPU series in 2013, and is available on most off-the-shelf Intel CPUs. To use the HTM instruction, programs issue an `XBEGIN` instruction to indicate the beginning of the transaction. After that, the programs read and write memory and perform arithmetic instructions as usual. At the end of the transaction, the programs should issue an `XEND` instruction to indicate a transaction commit request. Similar to the database transaction concurrency control, HTM ensures other CPU cores cannot see the uncommitted modifications. Once the transaction commits successfully, all changes become visible to other cores. If the transaction aborts, the changes are discarded. With HTM, the transactional semantic is almost free. The transaction is implemented within the cache coherence protocol. CPU always enforces the protocol regardless whether programs issue HTM instructions or not. The (marginal) cost of the HTM instructions is almost negligible.

The most critical drawback of HTM is the capacity limitation. In the current Intel implementation, HTM utilizes the cache coherence protocol of its level one (L1) cache, which has only 32 kilobytes (KB). During the HTM execution, when the core accesses more than 32KB of distinct memory, cache overflows and the transaction immediately aborts. In this case, the whole transaction has to restart. Moreover, since most programs usually do not have an even memory access pattern

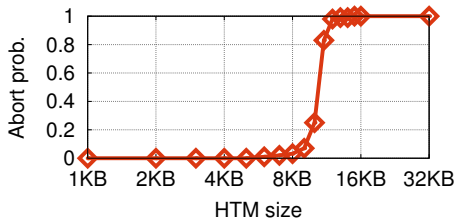



Fig. 4: The probability of a *raw* hardware transaction that would fail due to capacity overflow

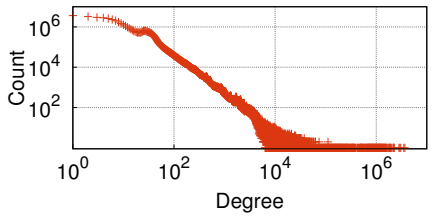



Fig. 5: The degree distribution of dataset `twitter-mpi`.

and modern CPUs implement set-associate cache, when the first cache overflow happens at one slot, other cache slots are still possibly underflow. Therefore, the cache overflow may occur before 32KB of unique memory access. Early experiments [10] show that a 10KB random access has up to 25% chance of abort. We conducted experiments by repeatedly executing transaction: we choose a consecutive 1GB memory, and pick two cores which continuously execute transactions at random locations with a specific transaction size and report their abort rates. We illustrate the probability of transaction aborting in Figure 4. When the size exceeds 30KB, the abort probability is almost 1.

**Large-Scale Graph:** Large-scale graphs like social networks and Web networks usually render a power-law degree-distribution [11]. (Certain graphs like road networks may have less skewed degree distributions. These smaller graphs are not the main focus of this paper.) We draw the degree distribution of a Twitter user's follower/followee relationship graph in Figure 5 (c.f. Section VI) The x-axis represents the (out)degree while the y-axis represents the number of vertices with such a degree, both in a log scale. The distribution is close to a straight line in the log scale, which implies power-law distribution. One of the corollaries of power-law degree distribution is the existence of a large maximum degree. The maximum degree of this graph is 3,691,240. Assuming each `int` variable occupies `4` bytes, the 32KB HTM capacity limitation is equivalent to 8,192 `int`s, which is far less than the maximum degree of the graphs. It is not possible to use one single HTM to wrap all accesses to a large-degree vertex's neighbors.

**Contention Rate and Transaction Processing:** Existing transaction schedulers can be roughly categorized into two classes: *pessimistic* and *optimistic* . The former uses locks to prevent data races. The latter assumes that data races are rare and progresses speculatively. After execution, the optimistic

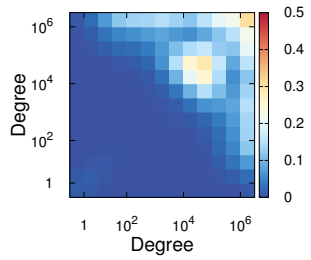



Fig. 6: Probability of data race on concurrent access.

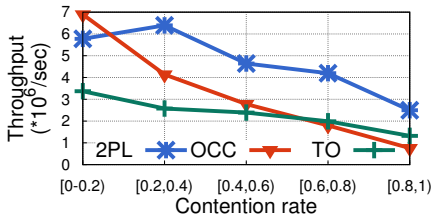



Fig. 7: Transaction scheduler's performance under various contention rates.

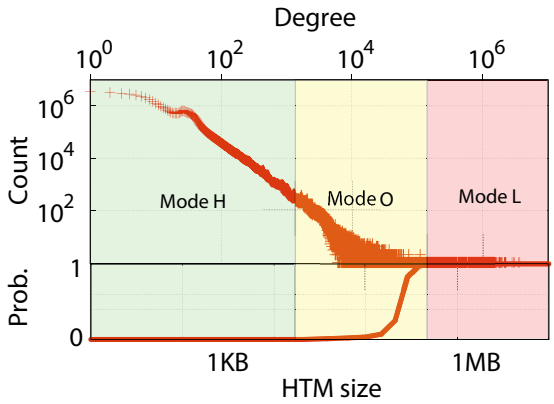



Fig. 8: The three-mode approach. The top shows the degree distribution of dataset `twitter-mpi` (c.f. Figure 5). The bottom shows the abort probability (c.f. Figure 4). The middle shows how we route the transactions into three sub-schedulers according to the degree of vertex and potential probability of abort.

scheduler verifies the execution and aborts it if anything goes wrong. The main difference between these two classes is their assumptions on the probability of data race, or in other words, the contention rate. The pessimistic schedulers assume a higher contention rate, and pay locking overheads ahead. They can efficiently prevent a high number of potential aborts when the contention rate is high as expected. However when the contention rate is low, the over-payed overheads hinder performance. On the other hand, optimistic schedulers assume a low contention rate. They save on overheads if the actual contention rate is low as expected. Otherwise, they abort transactions frequently.

Because the contention rate has a substantial effect on performance, we analyze the contention rate in graphs. We took the dataset `twitter-mpi` as an example, and analyzed the contention rate in a micro-benchmark. We assume each transaction reads a vertex and its neighbors, and writes the vertex. The contention rates are summarized in heat map Figure 6: each cell represents the the probability that two concurrent vertices contend, when the degrees of them are in corresponding x-axis and y-axis. The figure clearly illustrate two observations: the contention rate is highly skewed throughout all vertices; and high-degree vertices have higher contention rates.

To illustrate how contention rate affects transaction schedulers' performance, we performed experimental studies on a synthetic graph. The graph has an even degree distribution, and we controlled the contention rate by choosing the vertices jobs. We executed the jobs via three fundamental transaction schedulers: two-phase locking (2PL), optimistic concurrency control (OCC), and timestamp ordering (TO). The performance results are shown in Figure 7. We observed that there is no consistent winner under all contention rate scenarios: 2PL performs better when the contention rate is higher, while OCC outperforms when the contention rate is close to zero.

## IV. Efficient Graph Parallelization

As illustrated in previous sections, no single transaction scheduler is efficient in a heterogeneous workload. Therefore, we propose a three-mode HyTM in Section IV-A. We design three sub-schedulers for low, moderate and high contention rate scenarios separately. By sharing same locks and metadata, they are integrated as one HyTM. We prove the correctness of our HyTM in Section IV-B, and discuss how to route transactions to three sub-schedulers in Section IV-C. We design a novel algorithm to choose the performance-critical parameters dynamically in Section IV-D. We discuss implementation details in Section IV-E.

### A. A Three-Mode Solution

As discussed before, the heterogeneity of graph degree distribution calls for a discriminative HyTM. We propose a three-mode (**H**TM-only mode, HTM-assisted **O**ptimistic mode, and **L**ock mode) HyTM TUFAST and process transactions using three sub-schedulers. We introduce three sub-schedulers here, and discuss how to route transactions to them in Section IV-C.

The intuition of the three-mode approach is to process transactions according to their sizes and conflict rates. If a transaction is small enough, it should fit within the HTM's capacity. We enclose the transaction in one single HTM. This reduces unnecessary overhead. If the transaction size is large then there is no approach to improve the parallelism, so we process the transaction with locks. For these transactions with a moderate size, we process them in a *hardware-assisted optimistic* way. As shown in the bottom of Figure 8, the transaction size may not fit within the capacity of HTM but is not too far away. We cut the transaction into several pieces and process each piece in one HTM. The HTM can help to detect the conflicts on-the-fly with little overhead: if another concurrent transaction interferes, HTM detects it immediately.

**Algorithm 1** Transaction processing (H mode)

```
Require: vertices locks L[]
 1: procedure START(v)
 2:     XBEGIN(ABORT_HANDLER)

 3: procedure ABORT()
 4:     XABORT

 5: procedure READ(v, addr)
 6:     Try lock L[v] in shared mode
 7:     if fails then
 8:         ABORT()
 9:     return load(addr)

10: procedure WRITE(v, addr, value)
11:     Try lock L[v] in exclusive mode
12:     if fails then
13:         ABORT()
14:     store(addr, value)

15: procedure COMMIT()
16:     XEND
17:     Release all locks
```

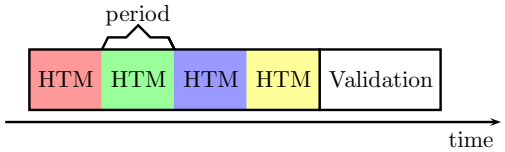



Fig. 9: O Mode with assistance of HTM

If all pieces are executed successfully, we validate the overall execution using an optimistic transaction validation.

**H Mode:** In H mode, the read/write operations are enclosed by HTM instructions. At the beginning, we issue the `XBEGIN` instruction to mark the start of transaction. HTM requires a mandatory fallback handler, which will be jumped to by HTM on the abort of the transaction. During the transaction execution, for each read and write operation, we check the fine-grained lock associated with each vertex respectively. If the vertex is locked in a non-compatible mode, we abort the transaction. At the end of transaction, we mark the finish of transaction by instruction `XEND`.

**O Mode:** The O mode is an optimistic transaction scheduler assisted by HTM. Optimistic scheduler works by performing operations in a private workspace. All read operations are read from shared memory, while the read operations and read results are also recorded. The write operations are not performed on the shared memory. Instead the write addresses and values are recorded in the private workspace. At the ending of the transaction, we verify whether the data read records are valid, i.e. the values read in past have not been overwritten during the execution and match the current values in corresponding addresses. This ensures the write operations are based on consistent data, and we can write the recorded write operations to shared memory.

**Algorithm 2** Transaction processing (O mode )

```
Require: vertices locks L[]
18: procedure START(v)
19:     retries ← 0
20:     reads ← ∅
21:     writes ← ∅
22:     counter ← 0
23:     XBEGIN(ABORT_HANDLER)

24: procedure ABORT()
25:     Release all locks

26: procedure READ(v, addr)
27:     if counter = period then
28:         counter ← 0
29:         XEND
30:         XBEGIN(ABORT_HANDLER)
31:     if v in writes then          ▷ Find the vertex in written log
32:         return writes[v].val
33:     if v not in reads then                ▷ Add the read record
34:         reads ← reads ∪ {addr, val=load(addr)}
35:     return load(addr)

36: procedure WRITE(v, addr, val)
37:     writes ← writes ∪ {v, (addr, val)}

38: procedure COMMIT()
39:     XEND
40:     for v in writes do                ▷ Update data and version
41:         Request L[v] in exclusive mode
42:         if fails then
43:             ABORT()
44:     for v in reads do                      ▷ Verify read access
45:         if load(reads[v].addr) ≠ reads[v].val or v
    locked then
46:             ABORT()
47:     for (addr, val) in writes.values do
48:         store(addr, val)
49:         Release lock L[v]
```

Optimistic scheduler, as suggested by the name, takes optimistic view towards the possibilities of transaction contention. It executes transaction without checking the contentions and postpone the inspection. However, as discussed before, the conflict detection via HTM is free. In additional to the traditional optimistic transaction scheduler, we plan to utilize HTM to help us detect contentions early. We enclose every `period`-th operations by one HTM, as illustrated in Figure 9. HTM will be notified if it is interfered by concurrent transactions. However the detection is not perfect. As in the example in Figure 9, if another thread issues an operation which conflicts with an operation in red (1st) zone while the optimistic scheduler is in the green (2nd) zone, this conflict cannot be detected by HTM. Therefore to ensure correctness, larger `period` is more desired. revisionOn the other hand, larger `period` may cause more capacity-triggered aborts. We will discuss how to choose `period` in Section IV-D.

**L Mode:** The L mode implements two-phase lock (2PL). For each read and write operation, 2PL requests corresponding

**Algorithm 3** Transaction processing (L mode)

**Require:** vertices locks `L[]`

50: **procedure** `START`($v$)

51: **procedure** `ABORT`()
52: Release all locks

53: **procedure** `READ`($v$, $addr$)
54: Request lock `L[v]` for *read* mode ▷ Request lock first
55: **return** load(`addr`)

56: **procedure** `WRITE`($v$, $addr$, $value$)
57: Request lock `L[v]` for *exclusive* mode
58: store(`addr`, `value`) ▷ Modify the vertex data

59: **procedure** `COMMIT`()
60: Release all locks

lock before performing the operation. At the end of the transaction, 2PL releases all the locks.

### B. Correctness

We prove that the protocol specified in Algorithm 1 to 3 only produces strict (conflict) serializable execution. Before the proof, we introduce some notations. Interested readers may refer to Weikum et al. [12]. For any two operations $o_1$ and $o_2$ in two different transactions $t_1$ and $t_2$, we say these two operations *conflict* if they access same data and at least one of two operations is write. In such case, we note $o_1 \leftarrow o_2$ if and only if $o_1$ happens before $o_2$. And we say $t_1 < t_2$ if $o_1 \leftarrow o_2$. For each transaction $i$, we define its *commit time* $ct_i$ as the time when the user calls `COMMIT`. For a set of committed transactions $T = \{t_1, t_2, \dots\}$, we say they are *strict conflict serializable* when $t_i < t_j$ implies $ct_i < ct_j$.

We prove by enumerating how $t_i$ is scheduled. First, if $t_i$ is scheduled by *H mode* (HTM scheduler), before $t_i$ issues commit instruction, its read/write operations are not visible to any other transaction at all. Therefore since one of $t_j$'s operations observes one of $t_i$'s operations, $t_j$'s operation must occur later than $t_i$'s commit time. Then $t_j$'s commit time is later than $t_j$'s commit time.

When $t_i$ is scheduled by *O mode* (HTM-accelerated optimistic scheduler), the proof is similar to the previous one. Before $t_i$ starts to commit and verify its read set and write set, other transactions cannot observe $t_i$'s work.

When $t_i$ is scheduled by *L mode* (2PL-based scheduler), just before $t_i$ enters it commit time, it holds the locks for all the data read or written by $t_i$. Assume $ct_i > ct_j$. Then one of $t_j$'s operation is written to the shared memory when $t_i$ holds the corresponding lock. This is impossible: $t_j$ will have to wait for the availability of lock if $t_j$ is scheduled by L mode; and will abort itself if it is scheduled by other two modes.

### C. Transaction Scheduling

We process transactions by three sub-schedulers based on the size of the transactions and the likelihood of transaction data races. However, the likelihood of data race depends on both the static setting (for example, the graph, and the graph application) and the runtime execution (for example, other concurrent transactions). It is impossible to know the likelihood of data races in advance. However, there is a strong correlation (c.f. Figure 6) between the size of the transaction and the number of data races. Thus, the user of TUFAST can provide an optional hint on the transaction size in the `BEGIN(SIZE)`. This hint comes from the user's domain knowledge on the expected size of the transaction, which is usually proportional to the degree of the vertex. Our hybrid transactional memory TUFAST routes transactions base on these optional size hint.

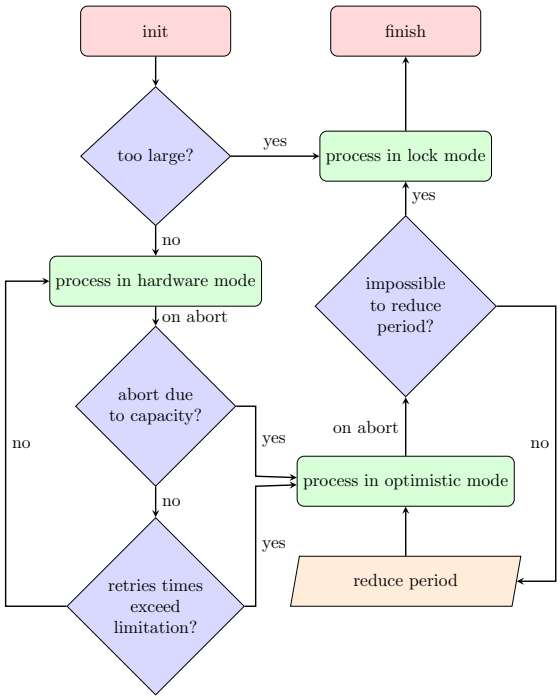



Fig. 10: Workflow of three schedulers

We illustrate the work flow of the transaction routing in Figure 10. We start from the H mode unless the size of transaction makes H/O mode impossible (then we proceed with L mode). If the H mode fails, we retry several times, as suggested by Intel [13]. After several retries, we proceed to the next mode: O mode. However, when an H mode abort is due to capacity overflow, we do not retry and instead directly proceed to O mode, because an abort caused by capacity overflow will repeat on retry. The O mode executes the transaction in optimistic scheduler accelerated by HTM. In the O mode, if the transaction has been aborted, we try to restart it, with an adjustment to the parameter `period`. If all retries chances are exhausted, we proceed to the final mode, L mode, which utilizes locks to schedule the transaction.

### D. Parameter Selection

In HyTM TUFAST there are several performance critical parameters. We discuss how they affect the performance of TUFAST, and how to choose appropriate values for them. We

choose other parameters which do not have severe impacts on the performance base on empirical studies.

The first parameter is the length of the HTM in O mode (c.f. Figure 9) `peroid`. We wrap every `peroid`-th operations in one HTM transaction. We use HTM to detect possible data races as early as possible. An early detection of data race reduces the unnecessary works because an optimistic transaction with data race cannot pass later validation phase. Therefore, we terminate the transaction when we detect the data race. For the best effect of detection ability, the period shall be as large as possible. On the other hand, the bigger the HTM transaction is, the higher possibility that it will abort. When it aborts, all uncommitted works are lost, and we have to start from the beginning of the HTM transaction. Aborted transaction wastes all efforts and we should avoid it. Base on these two observations, a careful choice of period is critical. What makes the choice harder is there is impossible to know whether the next operation triggers abort until we execute it. To analyze this problem, we propose a simplified model of HTM abort. An ongoing HTM will abort on the next operation with probability $p$. We reduce our problem to a threshold $P$: we choose to *proceed* with the next read/write operation if the number of uncommitted operations in the current HTM is less than or equal to $P$; and choose to *commit* otherwise. The expectation of committed operation is

$$(1-p)^P \times P$$

We find the maximal value of this expectation is achieved at $P = [\frac{1+p}{p}]$ ($[\cdot]$ represents the nearest integer). Therefore, we start with the `period` as $P$. If O mode fails to complete with the current `period`, we fall back and re-execute the O mode with a period equals to `period/2`. We stop reducing `period` when `period` is less than 100 and proceed with L mode. By continuously monitoring $p$ during the execution, we enforce this strategy adaptively base on the recent workload.

Another performance-critical parameter is the numbers of retries before we give up H mode. If a transaction is aborted due to conflict, it still has a chance to success on a retry. However, it is a waste of time if we retry too many times. The retry threshold depends on both the cost of retry and the success probability. The cost of retry mainly depends on the cost of memory access. Before the first-time HTM execution, with high probability that the requested data are not in the cache. Therefore, the first-time accesses are mostly cache miss accesses. After the first execution (and an abort), the successfully executed part of the transaction is already in the L1 cache line. Accessing them again is much cheaper than the initial access. As suggested by a experimental study [14], loading data from memory may costs 355 cycles, and loading data form L2 and L3 cache costs 11 and 44 cycles, respectively. Therefore, it is worth to retry the HTM for several times before giving up and moving to the next phase. We study how the number of retries affect the performance in experimental studies.

### E. Lock Implementation

L mode directly uses locks to prevent data races. H mode and O mode also request locks. Generally speaking, lock requests cause deadlock. We use deadlock detection to avoid possible deadlocks. If thread A is waiting a lock which is currently being hold by thread B, we say A *waits* B. Deadlock detection prevent cyclic *wait* relationship to ensure deadlock does not happen. Deadlock detection is a relative time-consuming operation because each time the detector checks all live *wait* relationships. Being said so, we observe that H mode and O mode do not need deadlock detection, because they only try to request locks. If they cannot successfully acquire the requested lock, they just abort. In other words, they cannot be a part of the "hold and wait" deadlock scenario because they do not wait for other locks. Thus, we only perform deadlock detection on transactions executed under L mode. In a large-scale graph, according to the "power-law" degree distribution, the number of vertices that have large degrees (and being executed by L mode) is small.

We could also save the deadlock detection in certain scenarios. Although a TM user can access memory arbitrarily, we observe that in real graph applications, transactions usually access their neighbors in a certain patten: iterate over all neighbors (in an unordered way). For example, greedy graph matching (c.f. Figure 1) inspects all neighbors to find an unmatched one; Page Rank sums up all neighbors' Page Rank values; BFS updates all neighbors' distance values. When a graph application satisfies this access pattern, we can prevent deadlock by deadlock prevention. The user assigns a global order (usually the natural order of their IDs) to all vertices, and access each vertex's neighbors according to this order. Thus, the locks are requested according to this order, and deadlock will not occur. In this case, user can choose to disable the deadlock detection.

## V. Related Work

**Main Memory Transaction**: The recent advance of main memory database [15], [16], even main memory cloud [17] stimulates both the theory and the practice of main memory transaction processing. The Shore-MT [18], [19] team from EPFL and Hyper [20] team from TU München propose several new transaction processing methods during the research of their projects. Other academic transaction processing systems, to name a few, include VLL [21], Silo [22], ERMIA [23], MOCC [24], THEDB [25], Quro [26], BCC [27], Orthrus [28], IC3 [29], TicToc [30]. Industry also adopts main memory transaction processing in their products including Microsoft Hekaton [31], [32], PostgreSQL [33] and SAP HANA [34]. Some of them try to address the problem of skewed access pattern problem by identifying cold and hot records. However, none of them solves the problem of highly skewed degree distribution (i.e. transaction size). Consider a transaction which accesses $10^6$ cold data. The existing schedulers classify the cold data as low conflict ones and process them via optimistic (sub-)schedulers. However, we argue they failed to take the

transaction size into consideration. Even it is less likely to conflict on cold data, a huge number of them still causes considerably high conflict rate of the transaction itself.

**HyTM**: First proposed in Herlily et al. [35], transactional memory (TM) has been an active research topic for the past two decades. Theory of TM and historical designs can be found in books [36] and [37]. Representative HyTM protocols include SigTM [38], HyTM [39], Hybrid NOrec [40], and Invyswell [41]. However, similar to main-memory transaction schedulers, these HyTMs that are not targeted towards graph application fail to address the problem of degree distribution skew as discussed above.

Besta et al. [42] proposes Atomic Active Messages (AAM) and utilizes it to build graph applications. Although AAM is implemented and accelerated by HTM, the semantic of AAM is close to compare-and-swap (CAS) instead of TM. Thus it only serves as a primitive tool and cannot be used to parallelize graph applications directly.

**Graph Processing in Single Server**: Besides the graph systems Ligra [1] and Galois [2] introduced in previous sections, in recent years researchers and industry pioneers proposed several graph systems which are capable to process graphs efficiently in a single server, including Graphspan [43], Mosaic [44], Graphene [45], FlashGraph [46], GraphQ [47], Chaos [48], TurboGraph [49], and GraphChi [50]. However these systems store and process graphs in SSD, HDD or NVRAM. The design choice of secondary storage based system is not same as the design choice of in-memory systems, and as shown in following experimental studies, the performance of secondary storage systems is not close to in-memory systems due to slower throughput and higher latency.

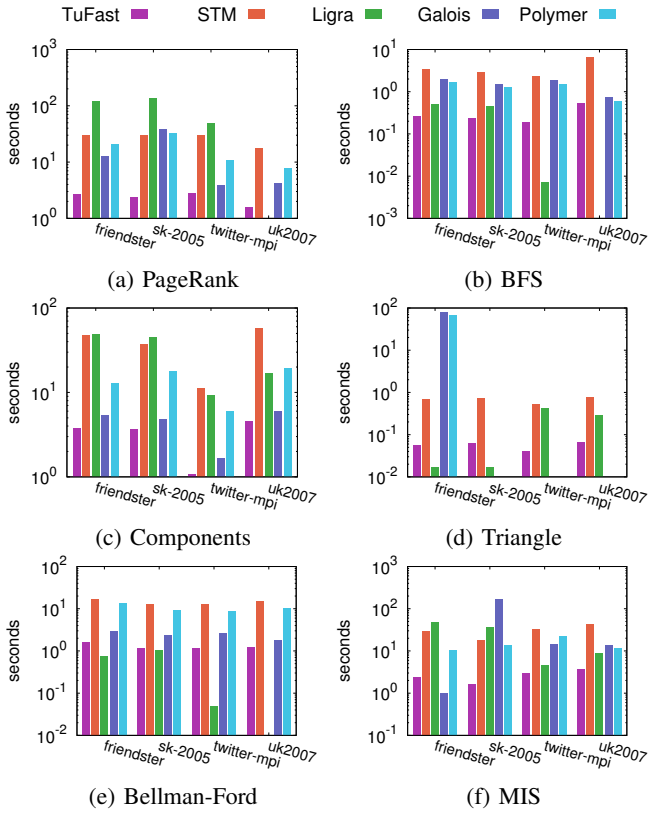
## VI. Experiment

We briefly introduce the experimental studies we conducted and analyze our results in details later. First, in Section VI-A, we compare TuFast with other graph processing systems, including main-memory graph systems and distributed ones on graph application workloads. We then compare TuFast with other classical transactional schedulers and state-of-the-art HTM-based schedulers in Section VI-B. TuFast outperforms other schedulers because it carefully schedule vertex jobs to sub-schedulers tailored for various probability of conflicts. We then analyze the execution trace of TuFast to better understand its performance in Section VI-C and test the parameter sensitivity of TuFast in Section VI-D.

**Setting**: We conducted extensive experiments on a server with two Intel Xeon E5 2670[2] V2 @ 2.30GHz CPUs and 48GB main memory. All experiments are repeated 10 times and the average value is reported. We compile the programs using GCC 4.8.2 and Intel TBB 4.4.

**Real large graphs**: Four real large graph datasets are used in the experiments (Table II). The dataset `friendster` is an on-line game network. Two people are connected by an edge if they are friends. It is available at SNAP.[3] The dataset `twitter-mpi` is a follower network in Twitter, crawled by MPI in 2010.[4] All other datasets are large social networks graph and web page graphs used in different domains which can be downloaded from WebGraph.[5] The statistics of datasets are shown in Table II.

[2]Although Intel disabled the HTM instruction temporarily, it can be enabled by configuring model specific registers [51].



(a) PageRank (b) BFS (c) Components (d) Triangle (e) Bellman-Ford (f) MIS

Fig. 11: Performance compared with single multi-core systems

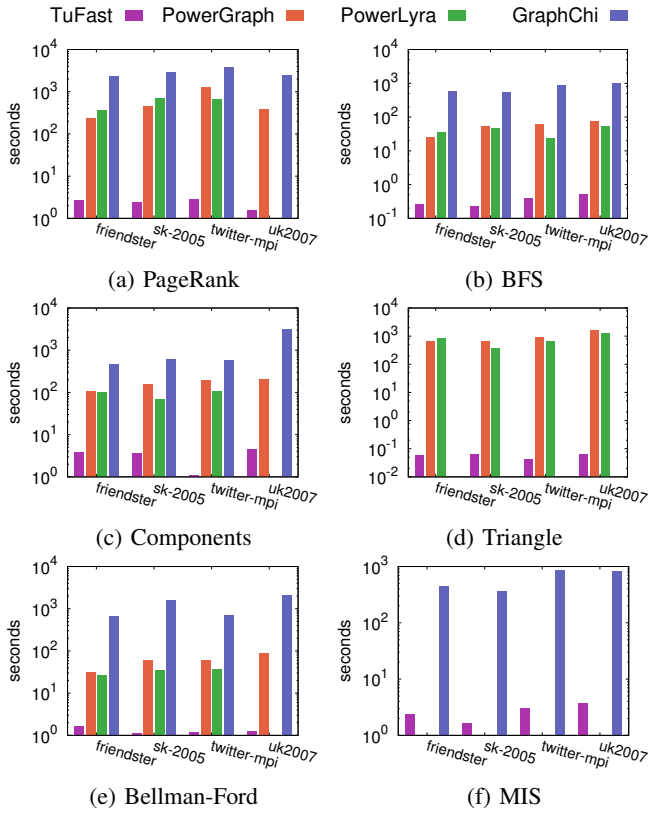
| Dataset | $\|V\|$ | $\|E\|$ | $\|E\|/\|V\|$ | Size |
|---|---|---|---|---|
| `friendster` | 65.6M | 1806M | 27.53 | 16.0G |
| `twitter-mpi` | 52.6M | 1963M | 38.50 | 17.0G |
| `sk-2005` | 50.6M | 1949M | 38.50 | 17.0G |
| `uk-2007-05` | 105.8M | 3738M | 35.31 | 33.0G |

TABLE II: Four real graph datasets

### A. Parallelization Performance

We measure TuFast's performance on graph applications, and compare it with other systems. We conduct experiments on Page Rank, breadth-first search (BFS), weakly connected components (Components), triangle counting (Triangle), Bellman-Ford shortest path, and minimal independent set (MIS). For the shortest path problem, we generate the edge weight randomly. For the MIS problem, we convert our graphs into undirected ones.

[3]http://snap.stanford.edu/data
[4]http://twitter.mpi-sws.org/
[5]http://law.di.unimi.it/datasets.php




(a) PageRank (b) BFS (c) Components (d) Triangle (e) Bellman-Ford (f) MIS

Fig. 12: Performance compared with graph systems on cluster

We first focus on single multi-core servers. Besides our solution TUFAST, we also integrate it with a software TM (STM) solution TinySTM 1.0.5 [52], [53] by replacing all hardware instructions by software counterparts. We also conduct experiments on state-of-art multi-core graph systems Ligra [1], Galois [2], and a NUMA-aware graph computing system Polymer [54]. Ligra utilizes a message passing systems similar to Pregel [4]. Each worker, which is usually a vertex, communicate with other workers via messages. Messages are periodically propagated and delivered by the system in a batched way. Batched communication amortize the communication overheads to increase the performance of the system. However, it suffers from other problems, like message queue maintenance cost, extra memory footprint, message staleness, lack of global information, etc. Galois is a mixed system: its default configuration prevents data races using locks like our L mode. However, it tries to eliminate the locks by static program analysis, and discard locks if they are not unnecessary (for example, embarrassingly parallel).

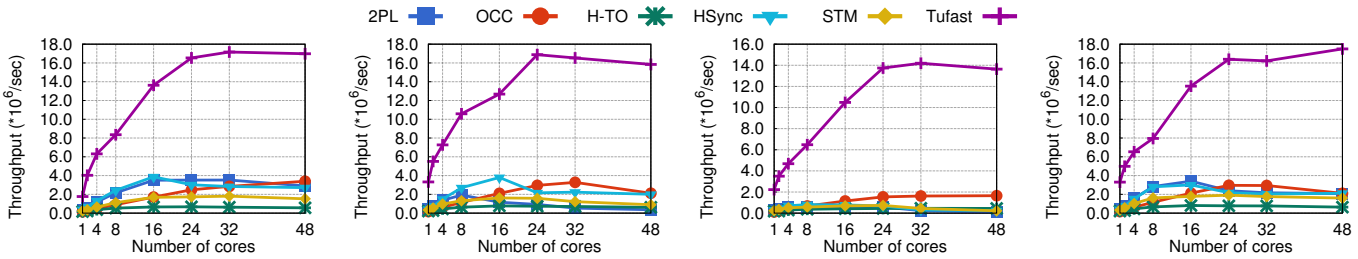
We report the results in Figure 11. TUFAST outperforms other systems with one or two orders of magnitude on some algorithms, and performs close to them on the others. The performance depends on the communication pattern of workload. For example, BFS requires the system to visit all vertices. In this case, the system bottleneck is the bandwidth of memory because there is virtually no computation. Nor can a system expedite computation by jumping to a far-away vertex. Therefore, a system with less overheads achieves better performance. The performance of the systems are close and Ligra outperforms on dataset `twitter-mpi` due to lower overheads. However, Ligra fails to finish on dataset `uk-2007-05` because its architecture requires additional memory to serve as message buffers. In Triangle algorithm, each vertex only require information from its neighbors. The system does not need to perform global communication to finish the task. In this case, systems with lower overheads perform better. On the contrary, the algorithm PageRank requires subtle coordination: the target of the system is to meet a convergence condition. Informally speaking, systems that spread information faster have advantage. In this case, TUFAST outperforms Ligra and Galois because TUFAST supports in-place-update. Therefore, workers always read the most fresh information as results of other workers' recent updates. They do not have to wait until next super-step to read updates, which is the case in BSP-like systems like Ligra. Expedited information propagation plays an important role in Components and MIS as well. Vertices in Components need newest component ID from their neighbors. And MIS jobs need to know whether their neighbors are chosen or not. In these cases TUFAST outperforms other systems by up to two orders of magnitude. Polymer optimizes NUMA memory access, but suffers from same performance issue that slows down Ligra or Galois. TUFAST always outperforms STM due to the lower overhead.

We also conducted experiments on a cluster with 16 node Amazon EC2 `m3.2xlarge` (8 cores, 30GB memory). We use PowerGraph [3], a distributed graph system and PowerLyra [55], an optimized PowerGraph. Besides them, we perform experimental studies on GraphChi [50] on a `r3.8xlarge` instance with 32 cores, 244GB memory and 320GB SSD. The memory size for GraphChi is set to 200GB, which is more than sufficient to store the graphs and other auxiliary data structures in memory entirely. We exclude all I/O time from the results reported. We compare TUFAST's performance in multi-core server and other three systems' performance in cluster or large-memory server and illustrate them in Figure 12. We observe TUFAST outperforms other systems by one to four orders of magnitude. Distributed solutions PowerGraph and PowerLyra fail to complete in some experiments due to memory capacity limitation. Although the cluster has sufficient resources (CPU and memory), these systems cannot fully utilize the resources because graph applications' computing bottleneck is the communication. GraphChi fails to utilize the memory efficiently although memory is sufficient.

### B. Scheduler Throughput

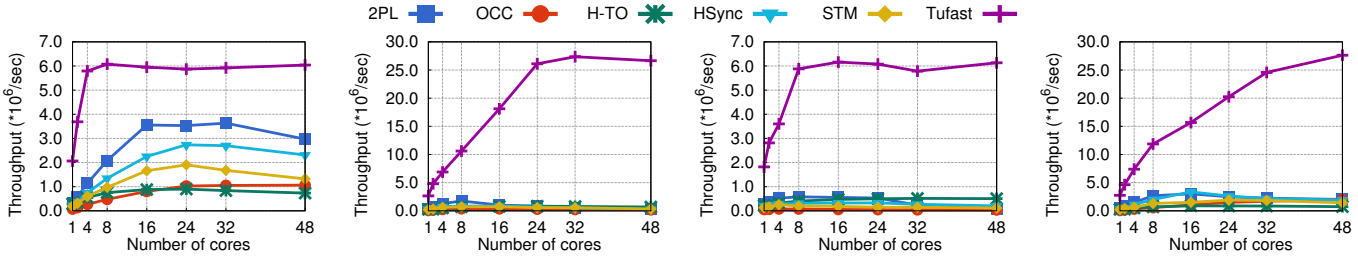
Besides experimental studies on real graph applications, we measure the throughput of TUFAST's scheduler and state-of-the-art schedulers.

**Workloads:** Based on the access pattern of real applications, we design a benchmark has two abstract workloads on different scope of read/write operations. Each transaction accesses a vertex and its all neighbors. One workload reads vertex $v$ and its neighbors and only writes the $v$ itself, the other reads and




(a) `friendster` (b) `sk-2005` (c) `twitter-mpi` (d) `uk-2007-05`

Fig. 13: Throughput on RM *(x-axis for number of cores, y axis for throughput)*




(a) `friendster` (b) `sk-2005` (c) `twitter-mpi` (d) `uk-2007-05`

Fig. 14: Throughput on RW *(x-axis for number of cores, y axis for throughput)*

writes $v$ and its neighbors. We note them RM (Read Mostly) and RW (Read and Write).

**Schedulers:** We implement two-phase lock (2PL) and an optimistic transaction scheduler Silo optimized for main-memory database (OCC) [22]. We adopt a software transactional memory library TinySTM (STM) [52], [53], the hybrid scheduler HSync (HSync) [56]. We also compare other HTM-based schedulers, including the HTM-accelerated timestamp ordering (H-TO) [10].

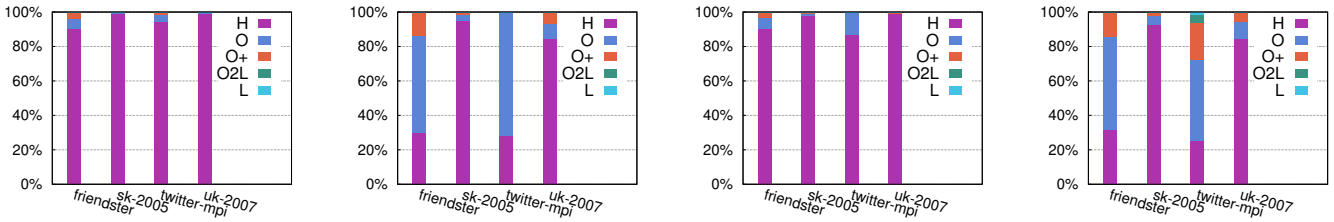
The throughput of transaction schedulers are illustrated in Figure 13 and Figure 14, for workload RM and RW respectively. From the figures, we find our three-mode solution TUFAST outperforms all others. For RM, TUFAST is `5.00x` to `8.25x` faster than the fastest of other solutions. For RW, TUFAST is `2.03x` to `39.46x` faster than them.

Besides the confirmation of TUFAST's superior performance, we also discover several interesting trends. First, for all schedulers, the hybrid ones (TUFAST, HSync) outperform homogeneous ones (2PL, OCC, STM and HTO). As discussed in previous sections, this is due to strong skewness in the graph degree distribution. Homogeneous schedulers are not able to process high degree vertices, medium degree vertices and low degree vertices in efficient ways simultaneously. Since all of these three class of transactions have strong impacts on the overall efficiency, failing to handle anyone hurts the throughput. Second, we find HTM-based schedulers performs better than non-HTM ones. This is due to the low overhead of HTM.

### C. Mode Breakdown

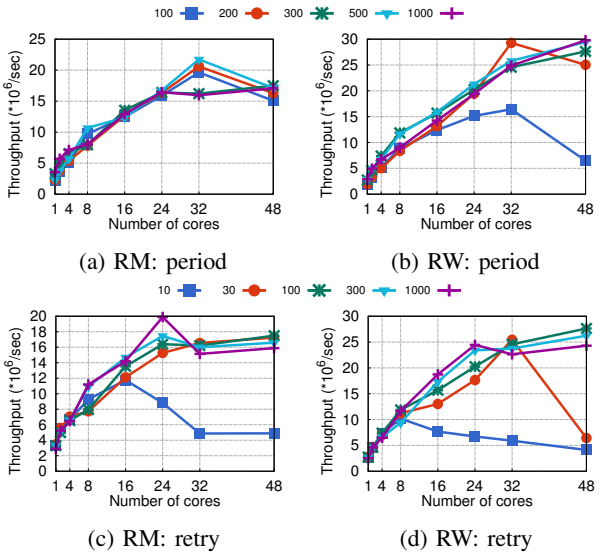
To analyze the execution trace of three modes and to better understand the root of TUFAST's superior performance, we group and aggregate the execution time of committed transaction by mode. We illustrate the results in Figure 15. In each figure, *H* represents the transactions executed in mode H. *O* represents the transactions executed in mode O and successfully committed at the first time. *O+* represents the transactions executed in mode O which aborted in the first trial but successfully committed after an adjustment of `peroid`. *O2L* represents the transactions executed in mode O and failed, then finally proceeded in mode L. *L* represents the transactions executed in mode L. For both workload RM and RW, we measure the number of transactions in every class (in 15a and 15c) and the total workload of transactions (i.e. the count of transactions' operations) in each class (in 15b and 15d).

We discuss the Figure 15b while the other figures are similar. In the figure we find the mode H is a major part of the total workloads. With the help of HTM, we can efficiently process the transactions without paying too much overhead. This helps TUFAST save more time. On the other hand, transactions executed in mode O (O and O+) are also a major part of the workload. Although these transactions do not fit within the HTM's capacity limitation, their sizes are close to the limitation. If we had gave up using HTM and process them via traditional OCC or 2PL schedulers instead, we may pay extra overheads that should have been saved by HTM. This also shows in a HTM-enabled system, how to make the maximum utilization of the cost-free HTM is the key to success. Last, we emphasize the transactions processed by mode L (including O2L and L). Although their workload is a small part of total workload, their sizes could be as big as millions. Optimistic schedulers are not able to handle such big transactions and we must protect them using locks.




(a) Num. of transactions (RM) (b) Total workload (RM) (c) Num. of transactions (RW) (d) Total workload (RW)

Fig. 15: Proportion of modes




(a) RM: period (b) RW: period




(c) RM: retry (d) RW: retry

Fig. 16: Parameter Sensitivity

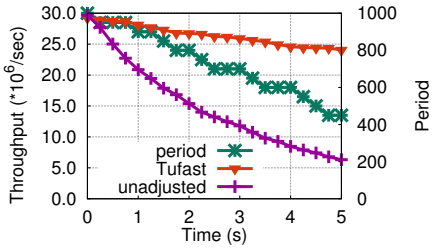



Fig. 17: Parameter-aware and parameter-ignorance solution: PR on uk-2007-05.

### D. Parameter Selection

In TUFAST, there are two performance-critical parameters: the period in mode O, which determines the length of each fragment; and the number of retries before TUFAST give up mode H.

We first test the parameter sensitivity. We use the benchmark presented in Section VI-B, and conduct experiments on dataset uk-2007-05. The results on other datasets are similar. In Figure 16a and 16b we illustrate the throughput when we keep the retry times to 100 and vary the period from 100 to 1000. In Figure 16c and 16d we fix the period as 100, and vary the retry times from 10 to 1000. We observe our parallelizer TUFAST is not sensitive to "medium" parameters, specifically when period is at least 300 and retry is at least 100.

The above experimental study shows TUFAST is insensitive to parameter selection when the workload is static. However, in real graph application, the workload varies as the algorithm makes progress. For example, in PR algorithm, initially all vertices are active. As the algorithm progresses, some loosely connected vertices' Page Rank values converge to stable ones. Thus these vertices vote to halt and do not participate in the computing. Remaining vertices (in a densely connected subgraph) continue computing. The remaining vertices tend to have high degrees and they are likely to be cause data races more frequently. Thus, a static `period` may not always be the best choice throughout the computation. We conduct experiments on dataset uk-2007-05 with PageRank algorithm. We report the throughput with static parameter (1000), the throughput with an adaptive period (c.f. Section IV-D) and the adaptive period itself in Figure 17. We observe adaptive parameter selection increases the throughput significantly, compared with the static parameter.

## VII. CONCLUSION

How to efficiently process various graph analytics jobs on huge size graphs is one of the most important problems in th "big data" era. Although HTM offers attractable transaction semantics for free, the size limitation hinders its application on graph problems. In this paper, we propose a HyTM TUFAST. The center of the design is to route transactions with different sizes to different sub- schedulers so each scheduler works within its favorite conflict rate zone. We confirm TUFAST's superior performance with extensive experimental studies.

## ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their valuable comments. The work was supported by grant of Research Grants Council of the Hong Kong SAR, China No. 14221716, 14203618, 12259116, 12232716, 12201518, and grant of National Natural Science Foundation of China No. 61602395.

## REFERENCES

[1] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *PPoPP*, pp. 135–146, 2013.
[2] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. *SOSP*, pp. 456–471, 2013.
[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. *OSDI*, pp. 17–30, 2012.
[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. *SIGMOD*, pp. 135–146, 2010.

[5] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015.

[6] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, Z. Zheng, B. Zhang, Y. Cao, and C. Tian. Parallelizing sequential graph computations. *SIGMOD '17*, pp. 495–510, 2017.

[7] W. Fan, P. Lu, X. Luo, J. Xu, Q. Yin, W. Yu, and R. Xu. Adaptive asynchronous parallelization of graph algorithms. *SIGMOD '18*, pp. 1141–1156, 2018.

[8] Shortest path faster algorithm. https://en.wikipedia.org/wiki/Shortest_Path_Faster_Algorithm, 2018. [Online; accessed 29-May-2018].

[9] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Priter: A distributed framework for prioritized iterative computations. *SOCC '11*, pp. 13:1–13:14, 2011.

[10] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. ICDE, pp. 580–591, 2014.

[11] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Rev.*, 51(4):661–703, Nov. 2009.

[12] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., 2001.

[13] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-029. March 2014.

[14] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. *SOSP*, pp. 33–48, 2013.

[15] H. Plattner and A. Zeier. *In-Memory Database Management: Technology and Applications*. Springer, 2nd edition, 2011.

[16] P.-A. Larson. Special issue on main-memory database systems. *IEEE Bulletin of the Technical Committee on Data Engineering*, 36, 2013.

[17] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, Jan. 2010.

[18] A. Ailamaki, R. Johnson, I. Pandis, and P. Tözün. Toward scalable transaction processing: Evolution of Shore-MT. *Proc. VLDB Endow.*, 6(11):1192–1193, Aug. 2013.

[19] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-mt: A scalable storage manager for the multicore era. *EDBT*, pp. 24–35, 2009.

[20] A. Kemper and T. Neumann. Hyper: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. *ICDE*, pp. 195–206, 2011.

[21] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *Proc. VLDB Endow.*, 6(2):145–156, Dec. 2012.

[22] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. *SOSP*, pp. 18–32, 2013.

[23] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. *SIGMOD '16*, pp. 1675–1687, 2016.

[24] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proc. VLDB Endow.*, 10(1):49–60, Oct. 2016.

[25] Y. Wu, C.-Y. Chan, and K.-L. Tan. Transaction healing: Scaling optimistic concurrency control on multicores. *SIGMOD '16*, pp. 1689–1704, 2016.

[26] C. Yan and A. Cheung. Leveraging lock contention to improve oltp application performance. *Proc. VLDB Endow.*, 9(5):444–455, Jan. 2016.

[27] Y. Yuan, K. Wang, R. Lee, X. Ding, J. Xing, S. Blanas, and X. Zhang. Bcc: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proc. VLDB Endow.*, 9(6):504–515, Jan. 2016.

[28] K. Ren, J. M. Faleiro, and D. J. Abadi. Design principles for scaling multi-core oltp under high contention. *SIGMOD '16*, pp. 1583–1598, 2016.

[29] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. *SIGMOD '16*, pp. 1643–1658, 2016.

[30] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. *SIGMOD '16*, pp. 1629–1642, 2016.

[31] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server 's memory-optimized OLTP engine. *SIGMOD*, pp. 1243–1254, 2013.

[32] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, Dec. 2011.

[33] T. Horikawa. Latch-free data structures for DBMS: Design, implementation, and evaluation. *SIGMOD*, pp. 409–420, 2013.

[34] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in sap hana database: The end of a column store myth. *SIGMOD*, pp. 731–742, 2012.

[35] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[36] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2nd edition, 2010.

[37] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Morgan & Claypool Publishers, 2010.

[38] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. *ISCA '07*, pp. 69–80, 2007.

[39] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *ASPLOS XII*, pp. 336–346, 2006.

[40] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. *ASPLOS XVI*, pp. 39–52, 2011.

[41] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy. Invyswell: A hybrid transactional memory for haswell's restricted transactional memory. *PACT '14*, pp. 187–200, 2014.

[42] M. Besta and T. Hoefler. Accelerating irregular computations with hardware transactional memory and active messages. *HPDC '15*, pp. 161–172, 2015.

[43] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. *ASPLOS '17*, pp. 389–404, 2017.

[44] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a trillion-edge graph on a single machine. *EuroSys '17*, pp. 527–543, 2017.

[45] H. Liu and H. H. Huang. Graphene: Fine-grained io management for graph computing. 15th USENIX Conference on File and Storage Technologies (FAST 17), pp. 285–300, 2017.

[46] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. 13th USENIX Conference on File and Storage Technologies (FAST 15), pp. 45–58, 2015.

[47] K. Wang, G. Xu, Z. Su, and Y. D. Liu. Graphq: Graph query processing with abstraction refinement—scalable and programmable analytics over very large graphs on a single pc. 2015 USENIX Annual Technical Conference (USENIX ATC 15), pp. 387–401, 2015.

[48] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. *SOSP '15*, pp. 410–424, 2015.

[49] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. *KDD '13*, pp. 77–85, 2013.

[50] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. *OSDI*, pp. 31–46, 2012.

[51] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. *SOSP '15*, pp. 87–104, 2015.

[52] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. *PPoPP '08*, pp. 237–246, 2008.

[53] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

[54] K. Zhang, R. Chen, and H. Chen. Numa-aware graph-structured analytics. *PPoPP 2015*, pp. 183–193, 2015.

[55] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *EuroSys*, 2015.

[56] Z. Shang, F. Li, J. X. Yu, Z. Zhang, and H. Cheng. Graph analytics through fine-grained parallelism. *SIGMOD '16*, pp. 463–478, 2016.