An Efficient Transactional Memory Algorithm for Computing Minimum Spanning Forest of Sparse Graphs *

Seunghwa Kang David A. Bader
Georgia Institute of Technology

Abstract

Due to power wall, memory wall, and ILP wall, we are facing the end of ever increasing single-threaded performance. For this reason, multicore and manycore processors are arising as a new paradigm to pursue. However, to fully exploit all the cores in a chip, parallel programming is often required, and the complexity of parallel programming raises a significant concern. Data synchronization is a major source of this programming complexity, and Transactional Memory is proposed to reduce the difficulty caused by data synchronization requirements, while providing high scalability and low performance overhead.

The previous literature on Transactional Memory mostly focuses on architectural designs. Its impact on algorithms and applications has not yet been studied thoroughly. In this paper, we investigate Transactional Memory from the algorithm designer's perspective. This paper presents an algorithmic model to assist in the design of efficient Transactional Memory algorithms and a novel Transactional Memory algorithm for computing a minimum spanning forest of sparse graphs. We emphasize multiple Transactional Memory related design issues in presenting our algorithm. We also provide experimental results on an existing software Transactional Memory system. Our algorithm demonstrates excellent scalability in the experiments, but at the same time, the experimental results reveal the clear limitation of software Transactional Memory due to its high performance overhead. Based on our experience, we highlight the necessity of efficient hardware support for Transactional Memory to realize the potential of the technology.

Categories and Subject Descriptors G.2.2 [*Discrete Mathmatics*]: Graph Theory—graph algorithms

General Terms Algorithms, Experimentation, Performance

Keywords Minimum Spanning Tree, Minimum Spanning Foreset, Transactional Memory

* This work was supported in part by NSF Grants CNS-0614915 and CAREER CCF-0611589 and by MIT Lincoln Laboratory. We acknowledge Sun Microsystems for their Academic Excellence Grant and donation of Niagara 1 and Niagara 2 systems used in this work.

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA.

1. Introduction

Over the past several decades, parallel programming was often considered in its own niche for supercomputing applications, and for desktop applications, sequential programming was sufficient owing to the ever increasing single-threaded performance. However, this does not hold anymore. Power wall, memory wall, and ILP wall [2] are forcing a paradigm shift to multicore and manycore architectures. However, we are also facing multiple challenges in software productivity to harness the increasing number of cores, and the data synchronization issue is identified as one of the most prominent bottlenecks for the widespread and efficient use of multicore computing. A major class of supercomputing applications is based on array data structures and has regular data access patterns; thus data partitioning for parallel programming is relatively simple, and data synchronization is required only at the boundaries of partitioned data. In contrast, another class of important applications uses pointer-based irregular data structures. This raises complex data synchronization issues and significantly exacerbates the situation.

Researchers have attempted to solve this problem in several directions targeting a spectrum of programming efforts and flexibility trade-offs. Thread Level Speculation [28, 31] extracts an additional level of parallelism without programmer intervention by speculatively executing the codes with unresolved data dependency. Kulkarni et al. [16] have developed the Galois run-time system to automatically parallelize irregular applications with user provided hints. However, to achieve the highest level of flexibility or chip utilization over widely varying applications using ever increasing number of cores, explicit parallel programming is often mandated.

Lock and barrier are the most popular data synchronization primitives for parallel programming until now but are problematic in programming complexity, scalability, or composability for complex applications. Coarse-grained locking has affordable programming complexity but often sequentializes the program execution. Fine-grained locking provides superior scalability at the expense of notoriously complex programming. Frequent invocation of barrier primitives can be a significant performance bottleneck as well. Also, for the lock-based codes, even correct implementations cannot be simply combined to form larger programs [13].

Transactional Memory (TM) [14] attempts to remedy this problem and promises to provide the scalability of fine-grained locking at the programming complexity of a single lock approach. TM also has great potential to resolve the composability problem [13]. Within a TM framework, we can seamlessly combine different transaction-based codes while maintaining atomicity of the transactions. Moreover, the assumption under TM fits quite well to the requirements to benefit from parallelization. von Praun et al. [29] classify applications based on data dependence densities and argue that applications with high data dependency cannot benefit significantly from parallelization while applications with low data depen-

dency can. TM also becomes more efficient as the degree of data dependency drops, and this suggests that if an application can benefit from parallelization, then that application can also benefit from TM.

Programmers only need to demarcate the critical sections for shared data accesses based on TM semantics. Then, if there is no conflict in the shared data accesses, the underlying TM system executes critical sections (or transactions) in parallel. If there are conflicts, the affected transactions are aborted and re-executed to provide the semantics of sequential execution for transactions. TM systems based on software (STM) [10, 15, 17], hardware (HTM) [1,5,12,21,24], and hybrid (HyTM) [8,20,26] approaches are proposed, which have trade-offs in flexibility, dedicated hardware requirements, and performance overhead for the different stages of deployment. There are already multiple open source STM implementations, and in 2009, Sun Microsystems expects to release its Rock processor [27], the first microprocessor with hardware support for accelerating TM.

Up to this point, most research on Transactional Memory has focused on systems and architectures, but few have considered its impact on algorithms and applications. There are several existing benchmark codes [7, 9, 11, 23] for TM systems, but these applications do not reveal the full potential of TM for real-world applications. Scott et al. [25] implement Delaunay triangulation using their TM system as a real-world application. Yet, for their implementation, data synchronization is required only at the boundaries of partitioned data. Watson et al. [30] report the implementation of Lee's circuit routing algorithm using TM. They achieve a high level of parallelism using TM in combination with the deep understanding of the algorithm to reduce the number of aborted transactions.

In this paper, we investigate the impact of Transactional Memory on algorithm design for applications with intensive data synchronization requirements. We believe that TM has great potential to facilitate scalable parallel algorithm design without involving the complexity of fine-grained locking or the ingenuity of lockfree algorithms. We provide an algorithmic model for TM to assist the development of efficient TM algorithms. Perfumo et al. [23] suggest *commit phase overhead*, *read-to-write ratio*, *wasted time*, *useful work*, and *cache accesses per transactional access* as metrics for characterizing STM applications. System researchers may feel comfortable with these metrics, but algorithm designers may not. Our model aims to reveal the key features of TM systems to be considered in designing algorithms. Then, we move our focus to graph algorithms. Graph algorithms are typically pointer-based and have irregular data access patterns with intensive data synchronization requirements. Graph algorithms also have interesting characteristics that fit well to TM systems at the same time. As a case study, we design an efficient Transactional Memory algorithm for computing minimum spanning forest (MSF) of sparse graphs. Our algorithm adopts a key idea from Bader and Cong's MSF algorithm [3], which implements a lockfree parallel MSF algorithm by combining Prim's algorithm with Borůvka's algorithm. Our algorithm replaces the Borůvka's algorithm part with the data merging and exploits TM to achieve high scalability with minimum parallel overhead. We present our algorithm with special emphasis on the TM specific design issues. We implement and test our algorithm using the Stamp [19] framework based on Transactional Locking II [10] STM. Our implementation demonstrates remarkable speedup for the different sparse graphs throughout the experiments. Yet, as our algorithm executes a significant fraction of the code inside transactions, the high STM overhead nullifies the speedup. This reveals the limitation of STM and highlights the necessity of the low overhead hardware support. The following summarizes our key contributions.

1. We identify the TM's potential to reduce the gap between finding the parallel tasks in computation and its actual implementation by replacing fine-grained locking based approach or lockfree implementations.
2. We provide an algorithmic model to assist designing efficient TM algorithms.
3. We design an efficient MSF algorithm for sparse graphs using TM support. Our parallel algorithm has minimum additional computation over sequential Prim's algorithm.
4. We exemplify multiple TM related programming issues while presenting our newly designed MSF algorithm.
5. We provide experimental results on the Stamp framework using Transactional Locking II STM. Based on the experimental results, we reveal the limitation of STM and the necessity for low overhead hardware support.

The remainder of this paper is organized as following. We describe the impact of Transactional Memory in algorithm design in Section 2. Section 3 summarizes the previous work in MSF algorithm design for sparse graphs, and Section 4 describes our efficient parallel MSF algorithm assuming TM support. In Section 5, we provide the detailed experimental results on the STM, and the discussion about the limitation of STM and the requirements for HTM is provided in Section 6. Finally, Section 7 concludes the paper.

2. Transactional Memory and Algorithm Design

2.1 Transactional Memory and Lockfree Algorithms

Lock is a popular synchronization primitive in multi-threaded programming. If multiple threads access shared variables concurrently, fine-grained locking is required to avoid sequentialization on a single lock. However, this approach involves a notoriously difficult programming challenge. Remembering the relationship between multiple locks and multiple shared variables is already a non-trivial issue. Deadlock and livelock problems, starvation, and other related issues, often increase the programming complexity to a prohibitive level. Therefore, many algorithm designers use coarse-grained locking instead and make an effort to reduce the size of critical sections to minimize the sequentialization problem. Unfortunately, this does not always work especially with the large number of cores. Ambitious researchers devote their effort to write lockfree algorithms that avoid race conditions using novel algorithmic innovations instead of locks. Yet, lockfree algorithm design requires deep understanding of an algorithm and the underlying system. As a consequence, lockfree algorithms are usually hard to understand and often rely on architecture-specific assumptions. Also, lockfree algorithms often involve additional computation to avoid data races.

Transactional Memory has great potential to change this situation. Programmers only need to mark the start and the end of transactions in their explicitly parallel code. Then, the TM system will automatically run non-conflicting transactions in parallel to achieve high performance while roll-back and re-execute conflicting transactions to provide the semantics of sequential execution for transactions. If the TM system can support this mechanism with high scalability and low performance overhead, we may not need to rely on lockfree algorithms anymore. Bader and Cong's parallel MSF algorithm [3] is a lockfree algorithm and incurs the overheads discussed above. As an illustration of our arguments, we design a new algorithm for computing a MSF using Transactional Memory. This will be further discussed in Section 4.

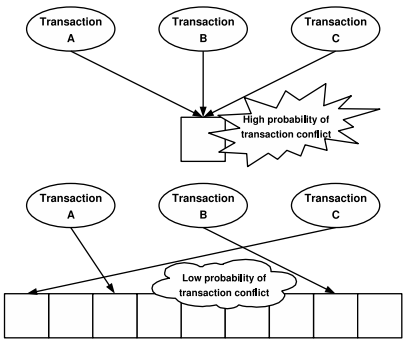

Figure 1. When multiple transactions access one shared variable at the same time, those transactions are highly likely to conflict (top). In contrast, if concurrent transactions mostly access different shared variables, those transactions will conflict with low probability (bottom).

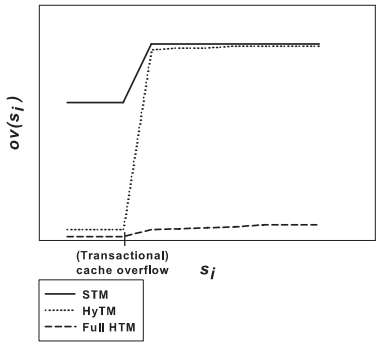

Figure 2. Transactional overhead $ov(s_i)$ varies as a function of the transaction size s_i. This figure assumes per core transactional cache for HyTM or full HTM and two-level memory hierarchy.

2.2 Algorithmic Model for Transactional Memory System

Transactional Memory can help programmers to write explicitly parallel programs, but TM cannot guarantee the scalable performance for every implementation. To best exploit TM, we need to write an algorithm that fits well with the underlying TM framework. However, it is difficult for algorithm designers to decide which design choice will lead to the better result without in depth knowledge of TM systems. We design our algorithmic model for TM towards this goal. Our model focuses on revealing the common key features of widely varying TM systems, which need to be considered in algorithm design. Our model does not aim to estimate precise execution time as this complicates the model without providing significant intuition in algorithm design.

Let, $T_{par.}$ and $T_{seq.}$ denote parallel and sequential execution time, respectively, and p is the number of concurrent threads (assuming w.l.o.g. one thread per core). Then, the following equation states our model assuming that an algorithm does not have an inherently sequential part.

$$
\begin{aligned}
T_{par.} &= \frac{T_{tr.} + T_{non-tr.}}{p} \\
T_{tr.} &= T_{seq.} \times \sum_{i=1}^{\#\,transactions} (\tau_i \times \#\,trials \times ov(s_i)) \\
T_{non-tr.} &= T_{seq.} \times (1 - \tau) \\
\#\,trials &= \sum_{j=1}^{\infty} j \times {p_i}^{j-1} \times (1 - p_i) \\
\tau &= \sum_{i=1}^{\#\,transactions} \tau_i
\end{aligned}
$$

where, $\frac{T_{tr.}}{p}$ and $\frac{T_{non-tr.}}{p}$ denote the execution time for the transactional part and the non-transactional part of an algorithm with p threads. An algorithm may consists of multiple transactions ($\#\,transactions$), which account for the different fraction of the total number of operations to run the algorithm (τ_i) and conflict with other transactions with the probability p_i. Transactions may execute several times to commit ($\#\,trials$) due to conflicts. p_i in each trial may vary according to the system specific contention management schemes. Also, an operation inside a transaction runs $ov(s_i)$ times slower than an operation outside a transaction owing to the TM overhead, where s_i is the transaction size. We rely on algorithm designers to estimate the value of p_i, as algorithm designers are likely to best understand the data access patterns of their own algorithm. To systematically estimate the level of data conflict in the legacy code, we refer the readers to von Praun et al.'s work [29]. $ov(s_i)$ is specific to the underlying TM system and also varies as the function of the transaction size. Figure 2 portrays the overhead assuming per core transactional cache. If the underlying TM system has a shared transactional cache, the overhead becomes a function of the aggregate concurrent transaction size.

Our model provides several guidelines to algorithm designers. STM incurs large $ov(s_i)$ regardless of the transaction size, and this suggests that algorithm designers should focus on minimizing τ. Yet, this increases the programming complexity and hampers the benefit of Transactional Memory. Moreover, if τ is small, a single lock approach may suffice to achieve scalability. The key disadvantage of a single lock approach is the sequentialization of critical sections on a single lock. This may not significantly limit the scalability of the algorithm if every thread spends only a negligible fraction of the time inside critical sections. HyTM incurs only low overhead for the transactions that fit to the transactional cache. The excessively large transactions still can incur high overhead. Full HTM may able to provide the relatively low overhead even in the case of overflow, and Ananian et al. [1] present UTM (Unbounded Transactional Memory) towards this goal. However, full HTM requires significant modification to the existing microprocessor architecture or even the DRAM architecture. This may not happen within the foreseeable future. Even with the full HTM support, an excessively large transaction can still be a problem as it spans a longer period of time and accesses multiple memory locations; thus likely to get aborted multiple times or aborts many other transactions and increases p_i. In summary, our model suggests that algorithm designers should focus on minimizing p_i while avoiding excessively large transactions.

2.3 Transactional Memory and Graph Algorithms

Graph algorithms represent one important class of applications that use pointer-based irregular data structures and have multiple features that can take advantage of Transactional Memory. While some of the synthetic graphs, scientific meshes, and regular grids can be easily partitioned in advance [16], many other graphs representing real-world data are hard to partition and thus, are challenging to efficiently execute on parallel systems [4]. Many graph algorithms also finish their computation by visiting multiple vertices in the input graph, and even with the optimal partitioning, we can traverse from one partition to the other partitions with a relatively small number of hops. Therefore, any vertex in the graph can be con-

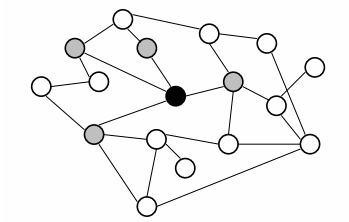

Figure 3. A common graph operation: **pick one vertex** (black) and **find (and apply operations on) its neighbors** (gray)

currently accessed by multiple threads, and this necessitates data synchronization over the entire graph. However, if the graph size is large enough relative to the number of threads, two or more threads will access the same vertex at the same time with very low probability. This leads to low p_i in our TM model. Then, TM can abstract the complex data synchronization requirements involving a large number of vertices, and high scalability can be achieved owing to the low degree of data contention.

In addition, we can easily decompose many graph algorithms to multiple transactions. Figure 3 depicts a common graph operation: **pick one vertex** and **find (and apply operations on) its neighbors**. Many graph algorithms iterate this process, and this common routine can be implemented as a single transaction. Thus, an algorithm can be implemented as an iteration of transactions. This leads to large τ and exemplifies the key benefit of TM over lock-based approaches. A single lock approach will sequentialize nearly the entire computation while fine-grained locking involves a prohibitively difficult programming challenge. In contrast, TM can significantly reduce the gap between high level algorithm design and its efficient implementation, as the independent tasks can easily map to transactions without compromising scalability. However, if the underlying TM system incurs high overhead, this overhead can nullify the high scalability as most operations need to be executed inside transactions. This necessitates low overhead hardware mechanism to obtain superior performance from high scalability. We may see low overhead HTMs (or HyTMs) within the near future, but excessively large transactions are likely to incur high overhead for longer time period, if not forever, as full HTM design is more challenging. If we decompose graph algorithms in the above way, a vertex degree determines the size of a transaction, and for sparse graphs, most transactions have relatively small size; thus, transactional cache overflow may not be an issue. Still, there can be a small number of vertices with exceptionally large vertex degree, and this can be a performance bottleneck. In our MSF algorithm, we modify the algorithm to address this issue, and this is further discussed in Section 4.2.

3. Minimum Spanning Forest Algorithm for Sparse Graphs

3.1 Sequential Minimum Spanning Forest Algorithm

Given a fully connected undirected graph, the minimum spanning tree (MST) algorithm finds a set of edges which fully connects all the vertices in the graph without cycles while minimizing the sum of edge weights. If an input graph has multiple connected components, the minimum spanning forest (MSF) algorithm finds multiple MSTs, one MST per each connected component. MST has multiple practical applications in the area of VLSI layout, wireless communication, distributed networks, problems in biology and medicine, national security and bioterrorism, and is often a key module for solving more complex graph algorithms [3]. Prim's, Kruskal's, and Borůvka's algorithms are three well known approaches for solving MST problems. Even though there are other algorithms with lower asymptotic complexity, they run slower in the experiments than these algorithms [22] due to large hidden constants in the asymptotic complexities. Based on the experimental results in [3, 22], there is often no clear winner among these three algorithms as execution time depends on the topology of the input graph and the characteristics of the computing system.

3.2 Parallel Minimum Spanning Forest Algorithm

Prim's and Kruskal's algorithms are inherently sequential, while Borůvka's algorithm has natural parallelism. Therefore, most previous parallel MST algorithms are based on Borůvka's approach (see Bader and Cong's paper [3] for a summary). Even though these algorithms' achieve parallel speedup for relatively regular graphs, none of these Borůvka's algorithm-based implementations runs significantly faster than the best sequential algorithm for a wide range of irregular and sparse graphs. Bader and Cong's paper [3] also introduces a new parallel MST algorithm which marries Prim's algorithm with Borůvka's algorithm. This new algorithm grows multiple MSTs using Prim's algorithm until conflict occurs. While growing MSTs, each thread marks the vertices in its own MST as visited and also colors all the neighbors of the marked vertices with its own color. If the algorithm encounters a vertex marked or colored by other threads, conflict occurs. Then, the thread starts to grow a new MST with a new vertex. If there is no remaining unmarked and uncolored vertex, the algorithm switches to the Borůvka's algorithm stage, finds the connected components, and compacts the graph. The algorithm iterates this process until the number of remaining vertices falls below the given threshold value. When this happens, the algorithm runs the best sequential algorithm to finish the computation. If an input graph has multiple connected components, this algorithm finds a MSF of the graph. Actually, if there is only one thread, this algorithm behaves similar to Prim's algorithm while it works as Borůvka's algorithm if the number of threads is equal to the number of vertices.

This algorithm avoids data races without using locks. To achieve this goal, this algorithm uses two arrays, $visited$ and $color$. However, it is not trivial to fully understand the mechanisms to avoid data races. Also, this algorithm assumes sequential consistency for the underlying system, and if the underlying system supports only a relaxed consistency model, additional fence operations are required. Therefore, even assuming that high level description of the algorithm is given, expertise in both algorithm and architecture is required to implement the algorithm correctly.

Moreover, the lockfree nature of the algorithm is achieved at the cost of additional performance overhead. If a vertex is colored by other threads but not marked as visited, adding this vertex to the thread's own MST may not lead to true conflict in most cases. However, this algorithm takes a conservative position and treats this case as a conflict to avoid possible race conditions. If an input graph has relatively small diameter, this can lead to excessive number of conflicts, and the algorithm may not perform much useful work in the Prim's algorithm stage. Then, the Prim's algorithm stage will just waste computing cycles, and the Borůvka's algorithm stage will perform the most useful work. Even when the graph has a large diameter, and conflicts occur infrequently, this algorithm runs both Prim's algorithm and Borůvka's algorithm and also additional computation to achieve lockfree nature; this leads to additional performance overhead. Due to this overhead, this algorithm requires 2 to 6 processors to outperform the best sequential algorithm depending on the input graphs.

4. An Efficient Transactional Memory Algorithm for Computing Minimum Spanning Forest of Sparse Graphs

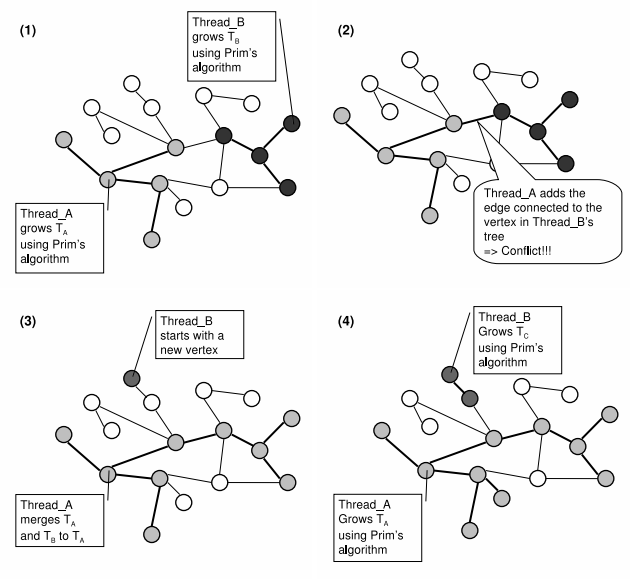

Figure 4. A high level illustration of our algorithm

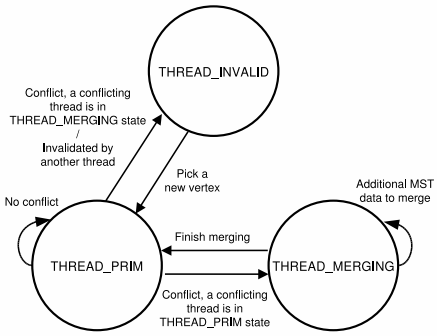

Figure 5. State transition diagram for our algorithm

In our new MSF algorithm, each thread runs Prim's algorithm independently similar to Bader and Cong's algorithm [3]. However, our MSF algorithm does not have the Borůvka's algorithm stage. Instead, if one thread attempts to add a vertex in another thread's MST, or conflict occurs, one thread merges two MSTs, and the other thread starts Prim's algorithm again with a new randomly picked vertex. Our new algorithm also does not detect conflict in a conservative way as Bader and Cong's algorithm. Instead, our new algorithm relies on the semantics of transactions to avoid data races. Figure 4 illustrates our algorithm in high level, and Figure 5 and Algorithm 1 give further detail. There are three states: THREAD_INVALID, THREAD_PRIM, and THREAD_MERGING. A thread starts in THREAD_INVALID state, and changes its state from THREAD_INVALID state to THREAD_PRIM by assigning a new color for the thread and also randomly picking a new start vertex. Then it runs Prim's algorithm in THREAD_PRIM state, and grows its MST (newly added vertices are colored with the thread's color). In this state, we iterate a common graph operation, **pick one vertex** and **find (and apply operations on) its neighbors**, as a single transaction as depicted in Algorithm 1. When two threads (assume $Thread\_A$ and $Thread\_B$) conflict, we merge two trees of the conflicting threads by merging their MST data. When $Thread\_A$ detects a conflict and finds $Thread\_B$ is in THREAD_PRIM state, $Thread\_A$ invalidates $Thread\_B$ and merges the MST data of two threads. $Thread\_B$ restarts by picking a new vertex. When $Thread\_A$ detects a conflict and finds $Thread\_B$ is in THREAD_MERGING state, $Thread\_A$ releases its MST data to $Thread\_B$ and moves to THREAD_INVALID state to restart with a new vertex. In a single-threaded case, our algorithm runs nearly identical to sequential Prim's algorithm with a small additional overhead related to the state management.

```
while true do
    TM_BEGIN();
    if current thread's state is THREAD_PRIM then
        w = heap_extract_min(H); /* pick one vertex */
        if w is already in the current thread's MST then
            TM_END();
            continue();
        end
        if w is unmarked by other threads then
            Add w to the current thread's MST and insert its
            neighbors to the heap H. /* find (and apply
            operations on) its neighbors */
            TM_END();
        else
            /* w is already marked by another thread */
            if the other thread marked w is in
            THREAD_PRIM state then
                Steal the other thread's MST data and
                invalidate that thread.
                TM_END();
                break; /* go to THREAD_MERGING state
                */
            else
                /* the other thread marked w is in
                THREAD_MERGING state */
                Release own MST data to that thread.
                TM_END();
                break; /* go to THREAD_INVALID state */
            end
        end
    else
        /* invalidated by other thread */
        TM_END();
        break; /* go to THREAD_INVALID state */
    end
end
```

Algorithm 1: Pseudo-code for THREAD_PRIM state. TM_BEGIN() marks the start of a transaction, and TM_END() marks its end.

To implement our algorithm, a complex data synchronization issue arises. Every vertex can be accessed concurrently, and every thread's state variable can be modified by multiple threads at the same time. To implement this algorithm with fine-grained locking, one thread may need to acquire up to four locks (one for its own state variable, two for the source and destination vertices in a new MST edge, and one more for the conflicting thread's state variable). This can lead to many complex scenarios that can cause race conditions, deadlocks, or other complications, and it is far from trivial to write correct and scalable code. In contrast, Transactional Memory can gracefully abstract all the complications related to the data

synchronization issues in our algorithm. As our algorithm executes a large fraction of the code inside the transactions (τ is large), this may not fit well with STM, but future TM systems with efficient hardware support may resolve this problem. Also, if an input graph is large enough, the level of data contention will be low (low p_i in our TM model), and our algorithm will fit well with TM systems.

However, our algorithm has other sources for parallel overhead. First, we need to pick a new start vertex multiple times. If a newly picked vertex is already included in other threads' MST, we need to pick a new one again. Second, MST data merging can take a significant amount of time. The first overhead may not be significant at the beginning of the algorithm, and even at the end of the algorithm (when almost every vertex is included in other threads' MST), this will not significantly slow down other threads' execution if we ignore the impact on the memory subsystem. To minimize the impact on the memory subsystem, if a thread picks a vertex in another thread's MST, we suspend the thread for a short time before picking a new vertex. Therefore, we need to focus on estimating and minimizing the overhead of MST data merging.

4.1 MST Data Merging and Composability

Assuming $Thread\_A$ merges its own MST data with $Thread\_B$'s MST data, MST data merging consists of two tasks. First, as all the vertices in $Thread\_A$ and $Thread\_B$'s MST need to be marked with a same color after the merging, $Thread\_A$ needs to recolor all the vertices in $Thread\_B$'s MST to $Thread\_A$'s color. Second, $Thread\_A$ needs to merge its own heap (a heap data structure is maintained to find the minimum weight edge efficiently as other Prim's algorithm based implementations) with $Thread\_B$'s heap. We can also expect that there will be a few merges of large MST data at the beginning of the algorithm followed by more frequent merges of large MST data with small MST data at the end of the algorithm.

If we recolor all the vertices in other thread's MST in a naïve way, it will significantly increase the parallel overhead of our algorithm. Instead, we add one level of indirection. We create a color array that maps an index to a color. When we add a new vertex to the MST, we mark that vertex with the index to the color array element for the thread instead of the color of the thread. Thus, for re-coloring, we need to only update the color array elements for the MST data of $Thread\_B$. The number of color array elements to be updated is identical to the number of MST data mergings that happened in $Thread\_B$'s MST data.

For actual implementation, this requires two additional shared data arrays to map an index to a color and a color to the owner thread. Under fine-grained locking, this requires additional lock arrays, and we need to re-design the entire lock acquisition protocol to avoid dead-lock or other lock related issues. Within a TM framework, in contrast, we can easily extend our algorithm without re-designing data synchronization schemes. This exemplifies TM's benefit over lock in composability.

To exploit the fact that there will be more frequent merges of large and small MST data, we switch $Thread\_A$'s MST data with $Thread\_B$'s MST data before merging if $Thread\_B$'s heap is larger than $Thread\_A$'s heap. Then, we merge two heaps by inserting the elements of the smaller heap ($Thread\_B$'s heap) to the larger heap ($Thread\_A$'s heap). These inserts are the most expensive parallel overhead in our algorithm.

We grow multiple MSTs concurrently, and this involves the overhead of MST data merging. However, this also decreases the average heap size throughout the execution. Instead of one thread growing a large MST to the end, there will be multiple small MSTs grown by multiple threads. A small MST will have less elements in its heap. If a heap has n elements, heap inserts or extracts will cost $O(log(n))$ memory accesses. Therefore, a single heap operation will cost less for the smaller MST. Especially, if a heap does not fit into the cache memory, multiple non-contiguous memory accesses in a single heap operation will lead to multiple cache misses, and these cache misses can be the most expensive cost of the algorithm. Combined with the modern cache subsystem with multiple layers, smaller heap size can have more significant impact on the performance than it appears in the asymptotic notation. If the impact of this is larger than the MST data merging overhead, our algorithm can scale super-linearly.

4.2 Avoiding Excessively Large Transactions and Strong Atomicity

As discussed in Section 2.2, excessively large transactions are undesirable for performance. There are two sources for large transactions in our algorithm. First, if we merge tree data inside a transaction, this will create a very large transaction. Second, if there are vertices with high degree, this will lead to large transactions.

In our state transition diagram, if a thread in THREAD_PRIM state (let $Thread\_A$) attempts to add a vertex in the MST of a thread in THREAD_MERGING state (let $Thread\_B$), $Thread\_A$ invalidates itself and appends its own MST data to $Thread\_B$'s queue for MST data. If MST data are fetched out from the queue for merging, this data cannot be accessed by other threads, and only the queue access needs to be executed inside a transaction. Therefore, MST data merging in our algorithm does not create a large transaction.

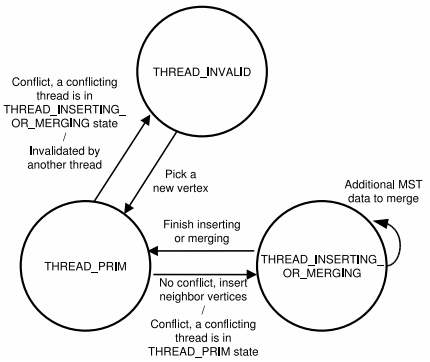

Figure 6. Modified state transition diagram for our algorithm

High degree vertices can also create a large transaction. If we extract a vertex and insert all the neighbors of the extracted vertex in a single transaction, the size of a transaction grows proportional to the vertex degree. This can be solved in a similar way to the first case. If we change our state transition diagram to Figure 6 and insert neighbors outside a transaction, we can avoid large transactions even with very high degree vertices.

These changes require accessing shared data both inside and outside transactions. At first glance, as MST data fetched out from the queue can only be accessed by a single thread, one can easily assume that this may not cause a problem. This is true assuming strong atomicity [6] but can leave a subtle hole under weak atomicity, which does not define the correct semantics among interleaved transactional and non-transactional codes. If a transaction doomed to abort reads data updated outside a transaction, which in turn can raise a segmentation fault, or non-transactional code reads data updated by a doomed transaction before it is restored, this can introduce a bug that is hard to find. One can easily assume that accessing shared data outside a transaction is a naïve program bug, but this may not true if we consider algorithm optimization. Our experience advocates the necessity for a strong atomicity guarantee in TM semantics.

4.3 Color Filtering

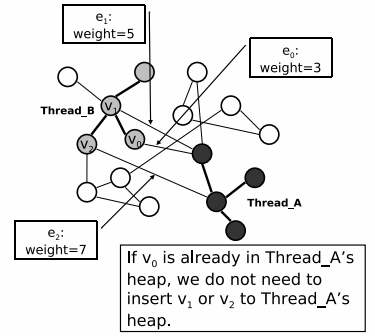

Figure 7. Color filtering: filter out unnecessary inserts

Our algorithm works well if an input graph has a large diameter as conflict occurs only infrequently in that case. However, if an input graph has a small diameter and conflict occurs more frequently, increased MST data merging cost can reduce the performance benefit of parallelization. However, for the graphs with a small diameter, there is another opportunity for the optimization. Assume we insert a neighbor vertex into the heap. If the heap already includes a vertex, which connects the current thread's MST to other threads' MST with lower weight than the new vertex to insert, we do not need to insert the new vertex as it cannot be an MST edge. Figure 7 illustrates the case. We can use the color of a vertex to filter out heap inserts. We maintain an additional data structure that maps a color to the minimum edge weight to connect the current thread's MST to the MST of that color. Considering that the heap operations account for the large fraction of the total execution time, reduced heap inserts can mitigate the increased parallel overhead owing to the frequent conflicts. Actually, this has similar impact to the connected component and compact graph steps in Borůvka's algorithm without incurring the high overhead of those steps.

4.4 Heap Pruning

When we merge the heap of two threads (assume $Thread\_A$ and $Thread\_B$), $Thread\_A$'s heap can include the vertices in $Thread\_B$'s MST, and $Thread\_B$'s heap can include the vertices in $Thread\_A$'s MST. If $Thread\_B$'s heap has less elements, we merge two heaps by inserting $Thread\_B$'s heap elements to $Thread\_A$'s heap. In this case, we do not insert $Thread\_B$'s heap element which belongs to $Thread\_A$'s MST. Yet, $Thread\_A$'s heap can still include the vertices in $Thread\_B$'s MST. Thus, after merging, the heap can include the vertices in its own MST, which cannot be an MST edge. When we extract the heap element, we check whether the extracted vertex is in its own MST or not, and this does not affect the correctness. Still, if the merging iterates multiple times, and near the end of the algorithm, there can be a very large heap with almost every vertex included in its own MST. At this time, there will be only one thread that includes almost all the vertices in the graph with few remainings. This thread will spend the most time for extracting the vertices in its own MST, while all the other threads are idling. To prevent this situation, we count the number of extracted heap elements that belong to its own MST. If this number grows above the given threshold, we scan the heap and remove the vertices in its own MST. By combining this heap pruning with the above color filtering, this can significantly reduce the heap size, especially for the graphs with a small diameter.

5. Experimental Results on STM

5.1 Test Graphs

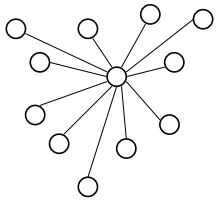

Figure 8. A pathological case for our algorithm.

We use a variety of graphs to experimentally test the performance of our algorithm. These inputs are chosen because they represent diverse collection of real-world instances, or because they have been used in the previous MST studies. We use the road map graph from 9th DIMACS implementation challenge website (`http://www.dis.uniroma1.it/~challenge9`) and also use EM-BFS and GTgraph graph generators available from the DIMACS website in addition to LEDA [18]. The scalability of our algorithm is highly affected by the number of MST data mergings and accordingly, is related to the diameter of an input graph. The test graphs summarized in Table 1 cover different graphs with varying diameters.

5.2 Experimental Setup

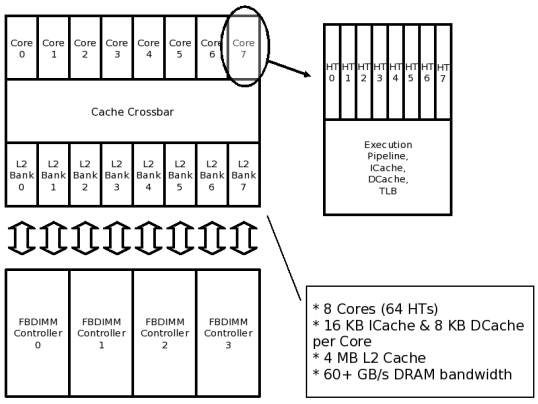

Figure 9. Sun UltraSparc T2 (Niagara 2) processor.

We test our implementation on a Sun UltraSparc T2 Niagara 2 processor (Figure 9) with 32 GB main memory. The Niagara 2 processor has 8 cores, and each core has 8 hardware threads (HTs). 8 HTs share a single execution pipeline. If an application kernel has frequent non-contiguous memory accesses and the execution time is limited by memory access latency, HTs can efficiently share the single execution pipeline without significant performance degradation in a single-threaded performance. As our MSF algorithm has multiple non-contiguous memory accesses and the memory latency of the accesses is the key cost of the kernel, the Sun Niagara 2 processor can be a suitable architecture for the algorithm. Especially, the Sun Rock processor architecture, likely to be the first microprocessor with hardware TM support, is based on the UltraSparc architecture, and we can better estimate the performance of our algorithm on the upcoming Sun Rock processor by the experimental results on the Niagara 2 processor.

graph type	generator	comments
2-D grid	LEDA	-
3-D grid	LEDA	3-D grid has relatively small diameter.
Web graph	EM-BFS	Web graph with 2K levels, and each level has 3M/2K vertices.
USA West graph	-	USA West roadmap graph. Included as a real word graph.
Random	GTgraph	Edges added by randomly picking two vertices among entire vertices of the graph. Reveals the worst case behavior of our algorithm except for few pathological cases (Figure 8)

Table 1. Test graphs for the experiments. For the graph generators which generate edges with uniform weight, we modify the code to generate random weight edges.

	2-D grid	3-D grid	web graph	USA West	Random
with STM overhead	983.1	1197	911.2	1143	1017
without STM overhead	12.96	18.60	12.00	17.33	97.34
MST data merging time	1.386	5.459	1.330	0.2734	59.31

Table 2. Comparison of the single-threaded execution time (in seconds) with the STM overhead (compiled with *Makefile.stm* in the Stamp framework), single-threaded execution time without STM overhead (compiled with *Makefile.seq* in the Stamp framework), and total MST data merging time in 64 threads case.

We use the Stamp (STAMP 0.9.9) framework [19] based on Transactional Locking II (TL2-x86 0.9.5) STM [10] for the implementation and the experiments. We use *gcc 4.0.4* compiler with *-O3 -mcpu=niagara2 -mtune=niagara* optimization flags. We compile the code with *Makefile.stm* (in the Stamp framework) for the STM-based executable and *Makefile.seq* (also included in the Stamp framework) for the executable without STM overhead. Also, we increase the STM lock array size (*_TABSZ* in *tl2.c*) from 2^{20} (default value) to 2^{25} to reduce the number of false transaction conflicts.

5.3 Experimental Results

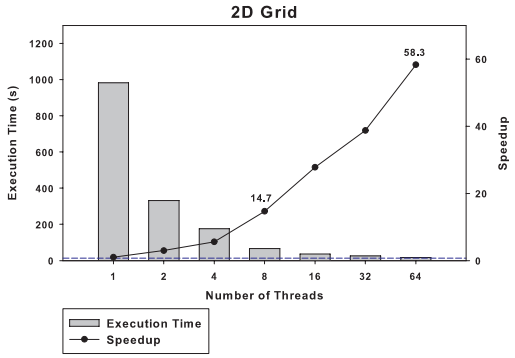

Figure 10. Execution time and speedup for the 2-D grid graph (3.24M vertices, 6.48M edges).

Figures 10, 11, 12, 13, and 14 summarize the execution time and speedup in the experiments. The dashed horizontal line in the figures denotes the single-threaded execution time for running the same algorithm (nearly identical to the sequential Prim's algorithm) without STM overhead. Our algorithm scales more than 8 times (18.5 in the best case) for 8 cores for all the test graphs and demonstrates remarkable speedup up to 64 HTs for all the test graphs except for the random graph. Super-linear speedup is achieved by reduced average heap size and the color filtering as expected in Section 4. For the graphs with a relatively large diameter (the 2-D grid, the web graph, and the USA West graph), our algorithm exhibits smaller number of conflicts, which leads to the high scalability. For the graphs with a smaller diameter (the 3-D grid and the random graph as the worst case), the number of conflicts increases

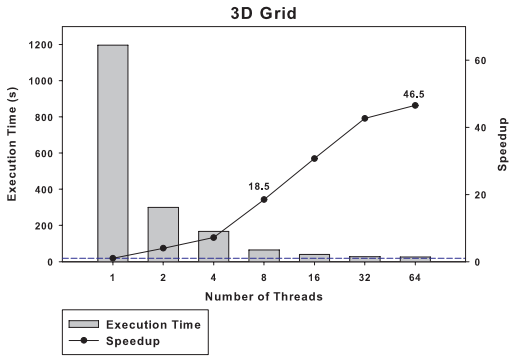

Figure 11. Execution time and speedup for the 3-D grid graph (3.38M vertices, 10.1M edges).

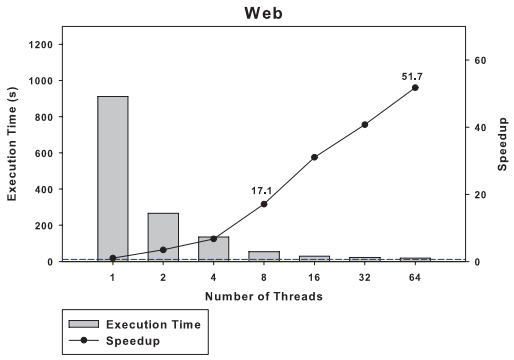

Figure 12. Execution time and speedup for the web graph (3M vertices, 6M edges).

but the color filtering compensates for this increase. Our implementation demonstrates remarkable speedup for the 2-D grid, the 3-D grid, the web graph, and the USA West graph. Even for the random graph that involves more frequent data mergings, we achieve parallel speedup using up to 16 threads. The lower scalability after 8 threads in the random graph case is also affected by higher spatial

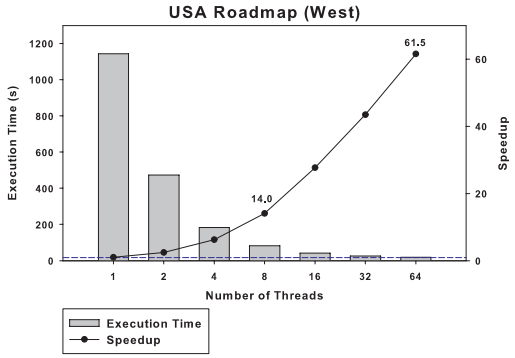

Figure 13. Execution time and speedup for the USA West roadmap graph (6.26M vertices, 7.62M edges).

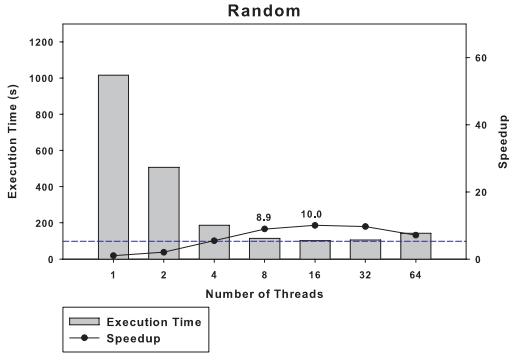

Figure 14. Execution time and speedup for the random graph (3M vertices, 90M edges).

locality; the random graph has higher average vertex degree, which leads to higher spatial locality in accessing neighbor arrays, and in turn, reduce the effectiveness of hardware threads. *However, even with this level of scalability, our parallel algorithm runs only at the comparable speed to the single-threaded case which does not incur the STM overhead.*

6. Limitations of STM and Requirements for HTM

Even though our STM implementation demonstrates remarkable scalability, the high overhead of the STM system nullifies the speedup. A single memory read or write operation outside a transaction is translated to a single LOAD or STORE instruction. A single shared data read or write operation inside a STM transaction, in contrast, involves multiple checks and data structure accesses, and this requires significantly larger number of instructions. Initial bookkeeping and commit time overhead exacerbates the situation. This overhead is not acceptable if we execute a large fraction of the code inside transactions. Ironically, if we execute only a small fraction of the code inside transactions, a single lock approach will suffice to achieve scalability, and we may not need Transactional Memory. This reveals the clear limitation of STM.

HTM can change this situation as a single read or write operation inside a HTM transaction requires only one LOAD or STORE instruction in the most currently proposed HTM papers. Still, initial bookkeeping or final commit time overhead can increase the cost of transactions, but this can be managed to a moderate level with hardware support. If HTM can realize this low overhead mechanism in commercial microprocessors, then we can replay the high scalability of our algorithm with only moderate overhead assuming sufficient memory bandwidth and scalable memory subsystem. Then, our algorithm can run significantly faster than the best sequential algorithm for irregular sparse graphs to the level that has not yet been demonstrated by others.

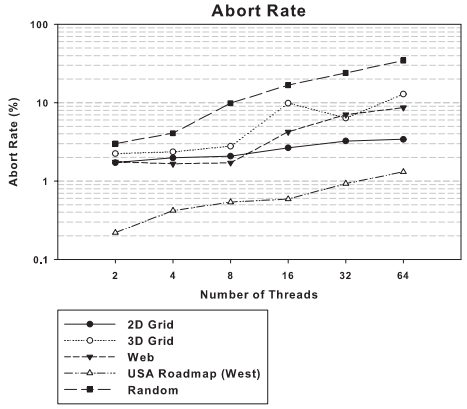

Figure 15. Abort rate for the varying number of threads on the STM.

One remaining point to check is the MST data merging overhead, as it can be underestimated owing to the high STM overhead. In Table 2, we can compare the MST data merging time in 64 threads case (the sum of MST data merging time for all 64 threads) with the single-threaded execution time without STM overhead, which will be similar to the single-threaded execution time in the efficient HTM system. Based on the comparison, we can identify that MST data merging time will not significantly lower the scalability except for the case of the random graph.

Also, we can expect lower abort rates in HTM systems as execution time for the transactions will account for a smaller fraction of the total execution time owing to the lower transactional overhead. Accordingly, there will be fewer concurrent transactions, and this will contribute to the lower abort rates than the case of STM (summarized in Figure 15).

7. Conclusions

Transactional memory is a promising architectural feature to reduce the difficulty of writing parallel programs with complex data synchronization issues. In this paper, we provide an algorithmic model for TM systems, and design an efficient transactional memory algorithm for computing a minimum spanning forest of sparse graphs. We evaluate our algorithm on an existing STM system, and identify the limitation of STM and the requirements for HTM. If TM researchers in academia and industry can deliver the low overhead commercial HTM systems, we believe TM can significantly reduce the complexity of data synchronization in parallel programming while achieving high performance. As future work, we will study the impact of TM on different algorithms and applications. We also plan to test our algorithm on future HTM systems, when they become available in the market.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proc. 11th Int'l Conf. on High-Performance Computer Architecture (HPCA)*, San Francisco, CA, Feb. 2005.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, Dec. 2006.

[3] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *Journal of Parallel and Distributed Computing*, 66(11), 2006.

[4] D. A. Bader and K. Madduri. SNAP, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks. In *Proc. 22nd Int'l Parallel and Distributed Processing Symp. (IPDPS)*, Miami, FL, Apr. 2008.

[5] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proc. 34th Ann. Int'l Symp. on Computer Architecture (ISCA)*, San Diego, CA, Jun. 2007.

[6] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Proc. 4th Ann. Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, Madison, WI, Jun. 2005.

[7] J. Chung, C. C. Minh, B. D. Carlstrom, and C. Kozyrakis. Parallelizing SPECjbb2000 with transactional memory. In *In Proc. Workshop on Transactional Memory Workloads (WTW)*, Ottawa, Canada, Jun. 2006.

[8] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. 12th Int'l Conf. on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, Oct. 2006.

[9] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the adaptive transactional memory test platform. In *Proc. 3rd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, Salt Lake City, UT, Feb. 2008.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. 20th Int'l Symp. on Distributed Computing (DISC)*, Stockholm, Sweden, Sep 2006.

[11] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: a benchmark for software transactional memory. In *Proc. 2nd ACM SIGOPS/EuroSys European Conf. on Computer Systems (EuroSys)*, Lisbon, Portugal, Mar. 2007.

[12] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. 31st Ann. Int'l Symp. on Computer Architecture (ISCA)*, Munich, Germany, Jun. 2004.

[13] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Chicago, IL, Jun. 2005.

[14] M. Herlihy, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th Ann. Int'l Symp. on Computer Architecture (ISCA)*, New York, NY, May 1993.

[15] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proc. 22nd Ann. Symp. on Principle of Distributed Computing (PODC)*, Boston, MA, Jul. 2003.

[16] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *Proc. 13th Int'l Conf. on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, Mar. 2008.

[17] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proc. 19th Int'l Symp. on Distributed Computing (DISC)*, Cracow, Poland, Mar. 2005.

[18] K. Mehlhorn and S. Näher. The LEDA platform of combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, 1995.

[19] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. 11th IEEE Int'l Symp. on Workload Characterization (IISWC)*, Seattle, WA, Sep. 2008.

[20] 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. In *Proc. 34th Ann. Int'l Symp. on Computer Architecture (ISCA)*, San Diego, CA, Jun. 2007.

[21] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proc. 12th Int'l Conf. on High-Performance Computer Architecture (HPCA)*, Austin, TX, Feb. 2006.

[22] B. M. E. Moret and H. D. Shapiro. An empirical assessment of algorithms for constructing a minimal spanning tree. In *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science: Computational Support for Discrete Mathematics 15*, pages 99–117. American Mathematical Society, 1994.

[23] C. Perfumo, N. Sonmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, and M. Valero. The limits of software transactional memory (STM):dissecting Haskell STM applications on a many-core environment. In *Proc. 5th ACM Int'l Conf. on Computing Frontiers (CF)*, Ischia, Italy, May 2008.

[24] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proc. 32nd Ann. Int'l Symp. on Computer Architecture (ISCA)*, Madison, WI, Jun. 2005.

[25] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay triangulation with transactions and barriers. In *Proc. 10th IEEE Int'l Symp. on Workload Characterization (IISWC)*, Boston, MA, Sep. 2007.

[26] A. Shriraman, M. F. Spear, H. H., V. J. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proc. 34th Ann. Int'l Symp. on Computer Architecture (ISCA)*, San Diego, CA, Jun. 2007.

[27] M. Tremblay and S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC processor. In *Proc. Int'l Solid State Circuits Conf. (ISSCC)*, San Francisco, CA, Feb. 2008.

[28] T. Vijaykumar, S. Gopal, J. E. Smith, and G. Sohi. Speculative versioning cache. In *Proc. 5th Int'l Conf. on High-Performance Computer Architecture (HPCA)*, Las Vegas, NV, Jan. 1998.

[29] C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Salt Lake City, UT, Feb. 2008.

[30] I. Watson, C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *Proc. 16th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Brasov, Romania, Sep. 2007.

[31] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In *Proc. 5th Int'l Conf. on High-Performance Computer Architecture (HPCA)*, Las Vegas, NV, Jan. 1998.