UPPSALA
UNIVERSITET

Hardware Transactional Memory in High-Performance Computing

Karl Ljungkvist, Gustav Persarvet, Tomas Toss
Supervisors: Martin Karlsson, Sverker Holmgren

Report in Scientific Computing Advanced Course

June 2010

PROJECT REPORT

Department of Information Technology

Abstract

When writing multi-threaded programs, handling of concurrency is a critical issue and this is traditionally solved by using locks. Transactional memory provides an alternative way of how to go around these issues. This paper is a report on our experiments and experiences using transactional memory in high performance computing applications. Two applications typical in HPC, a finite element matrix assembly and a molecular dynamics simulation, were implemented and tested on a prototype system supporting transactional memory in hardware. The outcome of our experiments suggests that there might be classes of HPC applications suitable for transactional memory. Some topics for future research are also discussed.

Keywords Transactional Memory, Hardware Transactional Memory, HPC applications, Concurrency.

Contents

1 Introduction 3
- 1.1 Parallelism and Concurrency . . . 3
- 1.2 Locks . . . 3

2 Transactional Memory 4
- 2.1 Transactional Memory in Software . . . 4
- 2.2 Transactional Memory in Hardware . . . 5
- 2.3 The Test System . . . 5

3 Applications 6
- 3.1 Parallel Assembly of FEM Matrices . . . 6
- 3.2 Molecular Dynamics . . . 7

4 Experimental Methodology 9

5 Results 10
- 5.1 FEM . . . 10
- 5.2 Molecular Dynamics . . . 12

6 Conclusions 15

1 Introduction

1.1 Parallelism and Concurrency

Computers are becoming more and more parallel. Although servers and computing clusters have been parallel for a long time, the trend towards parallelism is now stronger than ever. A consumer desktop 10 years ago had a single serial processor, whereas today processors with four cores are becoming increasingly common. To make use of the available computing resources, the program has to be written for parallel execution. This is usually done by either one of two techniques[1]. Very large systems typically consists of multiple computers (*nodes*) in an interconnect, where each node has a private memory set. On these systems, parallelization is usually done using separate *processes* on each node, and communication has to be performed explicitly. The other approach is to use *threads*, which are lightweight processes sharing the same address space. This is typically used on smaller systems where multiple processors, possibly with multiple cores each, all sit in the same box sharing a common memory.

When using the shared memory model, communication is trivial since each thread can access the memory used by all other threads. However, this can lead to so-called *concurrency* conflicts, where two threads are accessing the same piece of memory simultaneously resulting in unpredictable results (see Table 1).

Table 1: *Concurrency issues might arise if several threads manipulate the same memory location.*

`account`	Transaction 1 (withdraw 100)	Transaction 2 (deposit 200)
1000		
	`x1 = account`	
	`x1 = x1 - 100`	
		`x2 = account`
		`x2 = x2 + 200`
900	`account = x1`	
1200		`account = x2`

1.2 Locks

The solution to this problem is to make the the critical code section *atomic*, which means that it can either be executed in its entirety, or not at all. The classical way to achieve this is to use *locks*, whereby the critical code section is protected by a mutually exclusive lock, which keeps the threads from executing the same critical section at the same time. There are many problems that can arise when using locks though, such as deadlocks, convoying and priority inversion [1]. Deadlocks occur when several threads locks the same set of locks in reversed order. Convoying occur when a thread holding a lock gets descheduled, e.g. if a page fault occurs, or if its scheduled time runs out. This might lead to a situation where the threads that are about to run the critical section cannot do so until the descheduled thread is rescheduled. Priority inversion occurs when a thread with low priority is hindered from running by a thread with higher priority, while holding a lock that the latter need. There is also an extra cost when using locks. Not only need the locks be initialized; they also need to be set and unset each time a thread reaches a critical section in the code. If the concurrency is fine grained, many locks might be needed to resolve

[1]A combination of them is of course also possible.

it properly; e.g. N^2 number of locks to resolve concurrency issues in all elements in a matrix. Another problem with locks is that in some applications the number of cases that the locks are actually needed is low compared to the number of computations. This means that the threads will be spending a lot of time setting locks even though there will not be many conflicts, e.g. queue-dequeue [1] and FEM matrix assembly. Locks are therefore pessimistic in their approach; even if there might not be a conflict, the critical section must be protected to guarantee correct results.

Another approach to solving the concurrency issues is to use Transactional Memory (TM), to which we now turn.

2 Transactional Memory

A transaction is defined as a section of code which can either commit (succeed) or abort (fail). If a transaction commits, all changes it has made to the memory are made permanent, and if it aborts, all changes are rolled back and the pre-transactional state of the system is recalled. Because of this, the transaction is atomic.

Different reasons can make a transaction fail, but in this context the most interesting one is in the event of concurrency. When used in a parallel shared-memory program, if two transactions access the same piece of memory simultaneously, one of them will abort while the other one succeeds. When a transaction fails, the simplest measure is to just retry the transaction, but more advanced schemes are also possible. Transactions always seem to have been executed in serial order to all other threads, and different threads will not observe any difference in execution order for the transactions [1].

TM can solve some of the problems associated with locks, e.g. the problem with few but irregular conflicts compared to the number of calculations. Instead of locking the critical sections, TM just executes the critical sections, and if a conflict is detected, one of the threads aborts its transaction and retries, or applies some other appropriate strategy. Therefore TM is optimistic. It first tries to run the critical section and then handles problems when they occur (if they do). TM does not, though, automatically run faster than locks. In programs with heavy concurrency issues, one might as well use locks, since if TM is used, most of the transactions will fail to commit due to concurrency, and the efficiency will drop.

Locks do not only resolve concurrency, but are also used to introduce ordering constraints. For instance, a global barrier can be implemented which cannot be passed until all threads have reached it, thus making all threads wait for each other. TM on the other hand has no sense of order; to our knowledge, currently there is no way to define which transaction is to be executed first.

2.1 Transactional Memory in Software

There are several ways to implement transactional memory. One way is Software Transactional Memory (STM), where a software library is implemented to provide the necessary operations to perform the transactions. The library will have to monitor the different threads and their memory accesses and abort transactions in the event of a conflict. It would also have to include a fail policy defining what to do in the event of an abortion (simple restart, back off, etc).

An obvious problem with STM is the substantial overhead due to the extra instructions added when making calls to the library, as well as the memory-monitoring code. Instead, if this could be taken care of by additional hardware, the overhead would be reduced dramatically.

2.2 Transactional Memory in Hardware

There are several ways to implement Hardware Transactional Memory (HTM). Already in 1993, Moss and Herlihy showed that HTM could be implemented as a modified cache coherence protocol, with very successful simulation results [1].

A more thorough investigation of the design choices that exist when implementing a HTM system was done by Bobba et al. [2]. Three important design parameters; conflict detection, version management of the shared memory and conflict, were defined and their individual effects on the performance of the system were studied. This was done by simulating a diverse set of implementations with different choices for these parameters. Identifying a set of critical performance problems, 'transaction pathologies', it was found that by making the right design choices, a system with optimal performance characteristics could be constructed.

Since hardware will always be a limited resource, a severe problem with HTM is that it introduces upper bounds on the transactions, such as their number or their size. If HTM is implemented as a coherency protocol of cache lines, there is a maximum number of cache lines that can be touched in a transaction. Most transactions are short enough, but still a moderate part of them are too long for a regular HTM system to handle. A solution to this was proposed by Ananian et al. in their Unbounded Transactional Memory implementation (UTM), where speculative memory is kept in cache but is allowed to spill out into the main memory if necessary [3].

STM is free of the bounds inherent in HTM, and it also allows considerably more advanced features. To get these advantages but still have the performance benefits of HTM, a combination of hardware and software could be used. Further development of this concept has been done, for instance by Saha, Adl-Tabatabai and Jacobson with their Hardware Accelerated STM (HASTM) [4]. They propose a complex STM system capable of advanced semantics such as nested transactions, but utilizing hardware support for performance. Their simulation results show that its performance is comparable to HTM systems, but with the high flexibility of an STM system.

Another approach to combine hardware and software is to alternate between a number of different hardware and software TM implementations. Lev, Moir and Nussbaum suggests a Phased Transactional Memory (PhTM) system, which might execute transactions in either software or hardware or a hybrid mode [5]. Which of these modes should be used can either be decided a priori depending on the workload, the type of transaction (not all modes might support it), etc. More interesting, though, is the possibility to switch mode depending on the commit status of a transaction. This allows for very aggressive HTM modes, e.g. best-effort HTM, which can provide very good performance at the price of not guaranteeing any single transaction to commit. For instance, initially, high-performance best-effort HTM can be used, and if a certain number of trials fail, a more safe STM mode can be activated. A big advantage of this modular approach is that the constituent TM systems can be relatively simple, and new modes can easily be added.

2.3 The Test System

Our studies were conveyed on a prototype system with support for best-effort transactional memory in hardware. The interface to the TM consists of two instructions, one which starts a transaction, and one which commits it. There is also a register containing information about the reason for a transaction failure. Since there was not sufficient compiler support, inline assembler code had to be used to insert the HTM instructions into the code.

The processor we have used has 16 cores and is capable of executing up to 32 threads simultaneously. The cores are arranged in clusters of four, where each cluster shares an L2 cache of 512 KB (2 MB in total). The system has 128 GB of RAM.

3 Applications

In this section two HPC applications are described and it is shown how these can be parallelized with, and without transactional memory. The choice of the test applications has been made such that the algorithms' structure should take advantage of transactional memory. The expected difference in performance will also be discussed.

The general properties sought for when choosing suitable applications are temporal independence of the computations, an unpredictable concurrency pattern and a moderate amount of concurrency. The reason that these properties are important is that TM, to our knowledge, not in it self supplies synchronizing capabilities. An application having much temporal dependencies would therefore require extensive use of synchronizing primitives (such as locks). The unpredictable occurrence of concurrency is important to be able to utilize the low overhead of starting and committing a transaction, compared to the overhead of locks.

Two applications were chosen; an assembly of a finite-element stiffness matrix, and a molecular dynamics simulation. Both applications were first implemented in a serial reference version, followed by locks and HTM variants. The implementations received approximately the same amount of optimizations, measured in the time and effort put into writing the code. They are naïve in the sense that the implementations are as simple as possible. This was an important point since much more work has been done on figuring out clever algorithms for locks than has been done for TM.

3.1 Parallel Assembly of FEM Matrices

FEM (Finite element methods) are used in the field of scientific computing when solving differential equations, usually defined on domains with complicated geometry [6]. These equations may be extremely difficult to solve analytically, and FEM computes an approximation of the true solution with a guaranteed accuracy. The domain is approximated by discrete points, which can be visualized in 2D as a triangular mesh. At every point in the mesh, a *basis function* is introduced, which has a value of 1 at the current point and 0 at all the others. Using these basis functions, the original PDE is discretized and written in the form of a linear system of equations. The matrix premultiplying the solution is called the stiffness matrix, which describes the total interaction between the gradients of the basis functions. Every element in this matrix corresponds to an approximation of the following expression

$$\int_{\Omega} \nabla\varphi_i \cdot \nabla\varphi_j \mathrm{d}x$$

where Ω is the entire domain. A basis function φ_i is defined only at a very small fraction of the triangles, namely the ones with point i as a vertex. Because of this, for a given choice of i and j, only the triangles where both φ_i and φ_j are non-zero have to be considered. It is therefore convenient to split the integral into a sum of integrals over all the triangles shared by the current two basis functions:

$$\sum_{k\in\mathcal{K}} \int_{k} \nabla\varphi_i \cdot \nabla\varphi_j \mathrm{d}x$$

The most straightforward way of parallelizing the evaluation of these integrals is to compute each term in the sum in parallel. This will give a contribution to some of the elements, which then have to be added to the corresponding locations in the matrix. Since for an arbitrary triangle, the node points that constitute its vertices are almost completely random (i.e. unpredictable), the pattern in which the elements of the matrix are accessed is very irregular (see Figure 1).

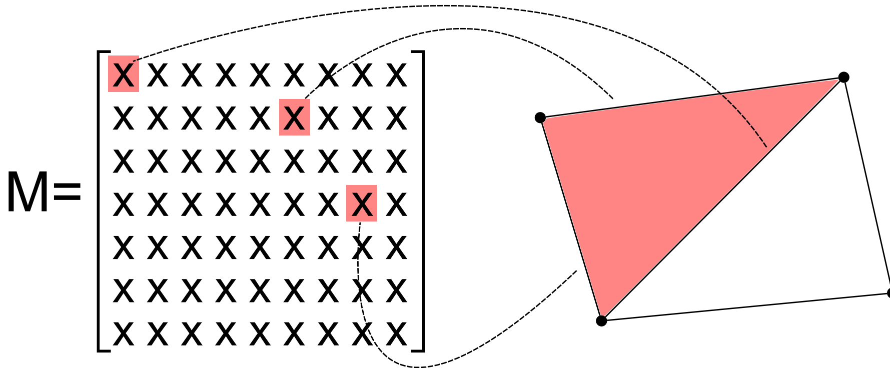


Figure 1: *Which elements of the matrix that gets a contribution from a certain triangle is very difficult to predict.*

The naïve implementation of parallelizing the matrix assembly would require one lock for every element in the matrix. The locks are needed to avoid the concurrency that would appear when two threads modify one and the same element of the matrix simultaneously. Using transactional memory, it is enough to enclose every summation to the matrix with a transaction clause. In theory, the program which utilizes transactional memory should perform better, since activating and deactivating a lock are expensive operations involving memory synchronization. If there are few memory accesses in per lock/unlock, as in this case, the lock-activating operations will in fact constitute a large portion of the execution time. Furthermore, there is no need to initialize any locks.

The simplest choice of basis functions are linear ones; i.e. consisting of planar surfaces. In that case, the gradients are constant and evaluating the integrals is simply a matter of finding these gradients and calculating the area of the triangle. To find out how well the transactional memory system would perform in comparison to the naïve version, we constructed parallel implementations using pthreads in conjunction with pthread mutices and HTM transactions respectively. The stiffness matrix is assembled in parallel, given a file containing a mesh configuration. A pseudo code of the implementation is listed in Algorithm 1.

With a more advanced choice of basis functions, the integral evaluation becomes more involved and is computed via a suitable quadrature formula. It is easy to see that this will lead to a considerable increase in computational work per triangle. This would increase the amount of computations per memory access, which in turn would increase the parallelism, leading to a better scaling program. To simulate the assembly of such a matrix, an alternative program was made performing a large number of floating point operations per triangle. This resulted in an 'artificial matrix' assembly program, which is basically the same as that in Algorithm 1, only with the extra operations added before line 15.

For both matrix versions, we implemented the above described HTM version, a parallel locks version, as well as a reference serial version. As input data, a reference mesh was created in MATLAB using the `initmesh` function with an `hmax` value of 0.1. This yielded a mesh with 526 points and 970 triangles.

3.2 Molecular Dynamics

Molecular dynamics (MD) aims at calculating the interactions and motions of molecules and atoms in a given domain. This makes it possible to study materials at atomic level. The method

Algorithm 1 *Finite Element Matrix Assembly*

```
1: m = NUM_TRIANGLES/NUM_THREADS
2: for k = 0, ..., NUM_THREADS-1 do
3:     thread = create_thread(assemble_local_matrix(k*m, (k+1)*m))
4: end for
5: assemble_local_matrix(lower_index, upper_index)
6: for k = lower_index, ..., upper_index do
7:     vertices = Global_Points[triangle_K.node]
8:     area = triangle_area(vertices)
9:     V[0] = {{1, vertices[0]->x, vertices[0]->y},
10:            {1, vertices[1]->x, vertices[1]->y},
11:            {1, vertices[2]->x, vertices[2]->y}}
12:    alpha = inverse(V)
13:    for i = 0, ..., 2 do
14:        for j = 0, ..., 2 do
15:            contrib = area * (alpha[1][i]*alpha[1][j] + alpha[2][i]*alpha[2][j])
16:            transaction
17:                Global_Matrix[triangle_K.node[i]][triangle_K.node[j]] += contrib
18:            end transaction
19:        end for
20:    end for
21: end for
```

is used in both physics and chemistry. Due to the nature of the system, to get a meaningful result, the time resolution has to be very high. This, in combination with the fact that one often wants to simulate many particles, makes MD simulations in general very computationally intensive. To calculate a particle's motion, every force affecting the particle must be calculated. When all the affecting forces have been computed, the new position of the particle can be computed. However, the calculation of the affecting forces is dependent on the positions of all the particles in the system. This implies that all forces affecting the particles in the system must be computed before the next time step can begin, and also that the amount of computations scales as the square of the number of particles.

Since forces are mutual, there is concurrency in the parallel implementation listed below. This can be seen in Figure 2. It is possible that multiple threads are trying to update the force acting on a specific particle at the same time. Solving this problem efficiently is not trivial; a naïve way is to associate one lock with each particle in the system. This will, however, introduce a significant memory and computational overhead. In theory, if keeping the same algorithm for the program, the use of transactional memory would eliminate these overheads. Every store of a local force contribution can be enclosed in a transaction clause. The global barrier synchronizing every time step can not however, be removed.

The pseudo code for computing the forces is shown below in Algorithm 2. The transaction only consists of updating the particles force, since these are the only instruction where concurrency can occur. In our case, Lennard-Jones pair potential is used to calculate the forces between the particles, and Velocity-Verlet time-stepping scheme is used to update the positions and the velocities of the particles <REF>. In all experiments the number of particles were 500, and 20 time steps were used. As initial conditions, a uniform random distribution of the particles was assumed.

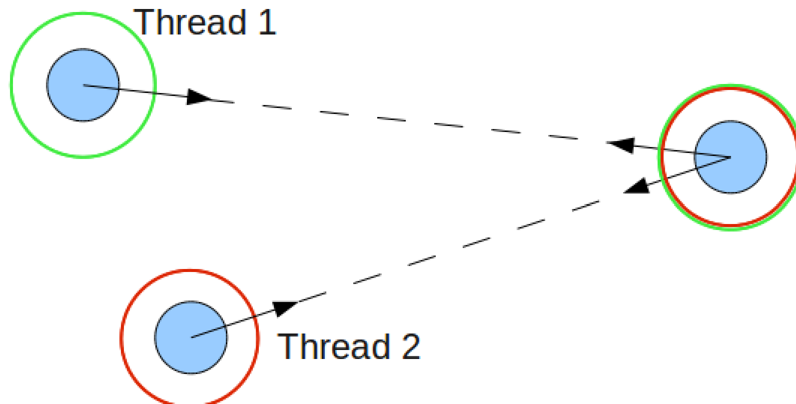


Figure 2: *Both threads tries to write to the same particle, which leads to concurrency issues*

Algorithm 2 *Molecular Dynamics Force Calculation*

```
1: calc_forces(thread_id)
2: for i = thread_id; i < NUM_PARTICLES-1; i += NUM_THREADS do
3:     for j = i+1, ..., NUM_PARTICLES-1 do
4:         force = calc_force(particle[i], particle[j])
5:         transaction
6:             particle[i].force += force
7:             particle[j].force -= force
8:         end transaction
9:     end for
10: end for
```

In order to implement TM into the code, we had to use inline assembler due to the compiler not having support for HTM yet. We chose to divide the force update into four parts (since the program computes a 2D model) – two per particle involved. This made the transactions very short, and made sure that only one store had to be done again if a transaction failed. A back off mechanism was also implemented, so that if a transaction failed, it would wait a short time before resuming. This would make sure that if the conflict was due to concurrency, the committing transaction should have time to finish before the aborting transaction restarted with the wrong data again.

4 Experimental Methodology

To test the performance of our implementations, we made runs for 1, 2, 4, 8, 16 and 32 threads. Since the run times showed large variations, each program was run several times. For the matrix assembly, each program was run 100 times, whereas for the MD simulation, each program was run 600 times. To get an understanding of the distribution of these times, the mean and minimum times were extracted for each program. Additionally, in the case of the molecular dynamics program, the distribution of run times was extracted and histograms were created. This was done because the run times were widely distributed.

5 Results

5.1 FEM

As expected, the standard stiffness matrix assembly was far too memory intensive to give any good scaling. The memory bandwidth is simply saturated from the beginning and all the threads have to wait for this single shared resource. As we see in Table 2, the run times are pretty much constant as the number of threads increase. This problem is shared by both implementations; however, it is also clear that the Locks version has a substantially higher run time than the HTM version. The reason for this becomes evident when the results for the artificial matrix assembly in Table 3 are considered.

Table 2: *Timings of the stiffness matrix assembly. Corresponding serial times:* $T_{mean} = 0.16365016$, $T_{min} = 0.160636$

(a) *Locks implementation*

$N_{threads}$	T_{mean}	T_{min}
1	1.84456610	1.814382
2	1.83092093	1.805148
4	1.85088277	1.817621
8	1.84239955	1.811181
16	1.84406599	1.828046
32	1.83919032	1.813848

(b) *HTM implementation*

$N_{threads}$	T_{mean}	T_{min}
1	0.17560922	0.172989
2	0.16995378	0.165402
4	0.16898129	0.167311
8	0.16508953	0.162202
16	0.16591953	0.162082
32	0.16717919	0.162981

Table 3: *Timings of the artificial matrix assembly. Corresponding serial times:* $T_{mean} = 1.57316664$, $T_{min} = 1.565162$

(a) *Locks implementation*

$N_{threads}$	T_{mean}	T_{min}
1	3.25853081	3.226169
2	2.53796925	2.521336
4	2.18921254	2.159421
8	2.01406703	1.985071
16	1.93932987	1.891760
32	1.97228127	1.869328

(b) *HTM implementation*

$N_{threads}$	T_{mean}	T_{min}
1	1.58357847	1.571706
2	0.87974116	0.872681
4	0.52337622	0.515900
8	0.34610028	0.338548
16	0.26784294	0.250196
32	0.30294744	0.220934

In this case, as expected, the run times are consistently higher because of all the extra operations. However, we can see in Figure 3 and Figure 4, that now there is a clear scaling of both the Locks and HTM versions. The Locks implementation is still considerably slower, and more importantly, we see in Figure 4 that the times for the Locks implementation flattens out at a value very close to the one for the stiffness matrix assembly (~ 1.85 s). This suggests that there is some serial work independent of the number of operations per element. Of course, this is the creation and deletion of all the N^2 locks. This was confirmed by timing these parts alone, which yielded a run time of approximately 1.8 s. The HTM version on the other hand shows reasonable scaling up to 8 threads, and continue to scale all the way to 32 threads. Obviously, the lack of locks account for the majority of this drop. But the fact that both of them scale is due to the higher amount of computations per memory access.

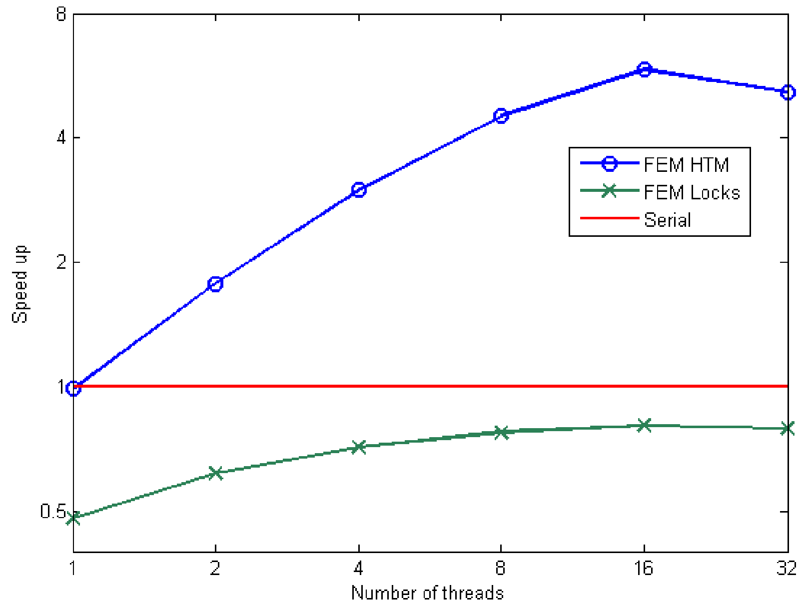


Figure 3: *Speedup of the different implementations of the artificial matrix assembly (mean time relative to serial time)*

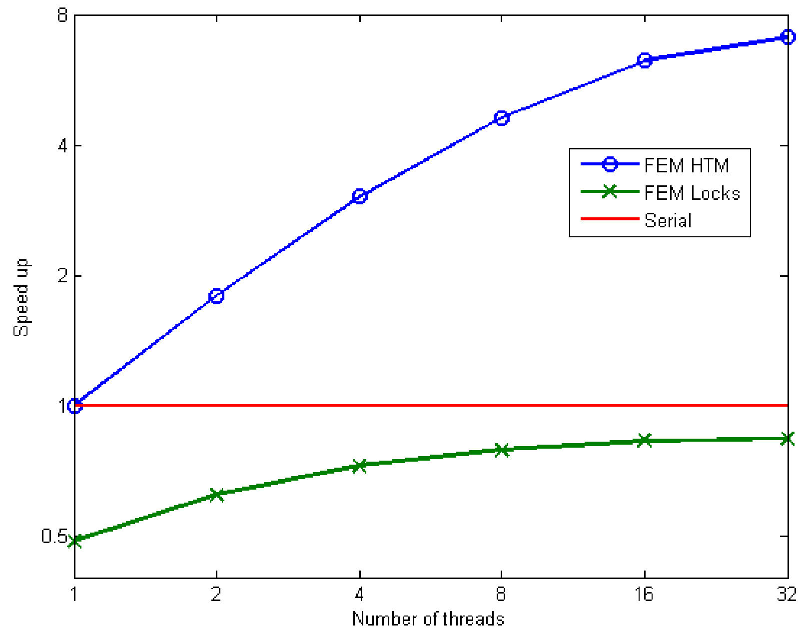


Figure 4: *Speedup of the different implementations of the artificial matrix assembly (best time relative to serial time)*

5.2 Molecular Dynamics

Figures 5 and 6 illustrate the results for the MD test problem; the first showing the speedup when considering the best run times, and the other one showing the speedup when considering the average run times. We see that, in the best case for HTM in Figure 5, the program scales very well up to four threads, then there is no gain in execution time between four and eight threads, and finally, it begins to scale reasonably well again. The efficiency is good up to four threads; but after that the efficiency falls below 50%. The locks implementation both executes slower across the board, and also scales worse, apart from the the unexpected results between four and eight threads. In both cases, the multi-threaded programs outperforms the serial version only after they get access to more than eight threads. This is probably due to the setting/unsetting of locks, global barriers, cache misses, and in the case of HTM, four inline assembler sections that the compiler could not optimize.

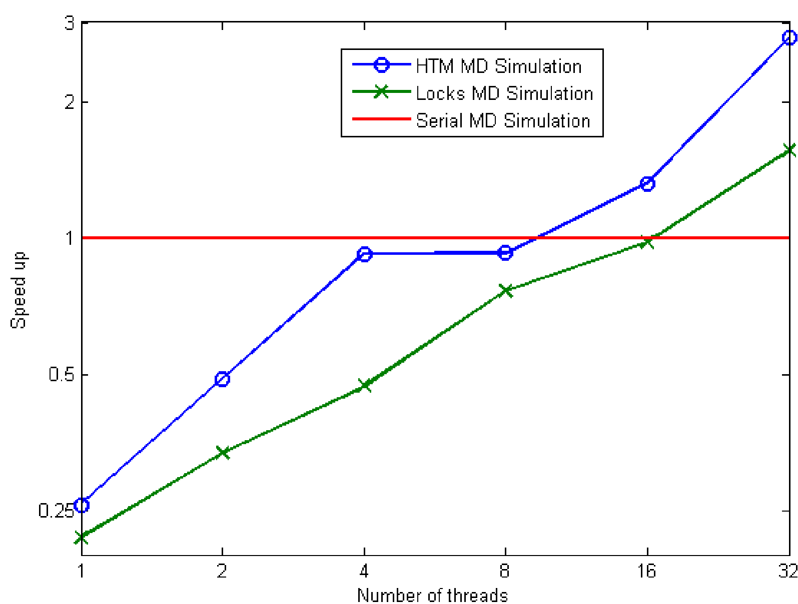


Figure 5: *The speedup of the best run time over all runs relative to serial time.*

In Figure 6 we see that the speedup of the average execution times are much worse than the speedup of the best execution times. This is especially true for the four and eight threaded programs, which run slower than one thread. This is because of the abundance of outliers, as can be seen in Figure 7. When the code that read and printed the contents of the register containing the failure reason for the transactions was included in the program, the majority of failure codes indicated concurrency problems. A possible explanation is that the operating system has scheduled several threads to the same core, and that these threads have large enough data sets to overwrite each others' data in the L1 cache. This has not been looked into though, and would need more research to be verified.

As can be seen from Table 4 the minimum time of the HTM program scales nearly linearly (halves the execution time if the number of threads are doubled) until the number of threads becomes four. Then the minimum time does not scale to eight threads at all, and finally it halves again between 16 and 32 threads. By then though, the program is about eleven times faster than

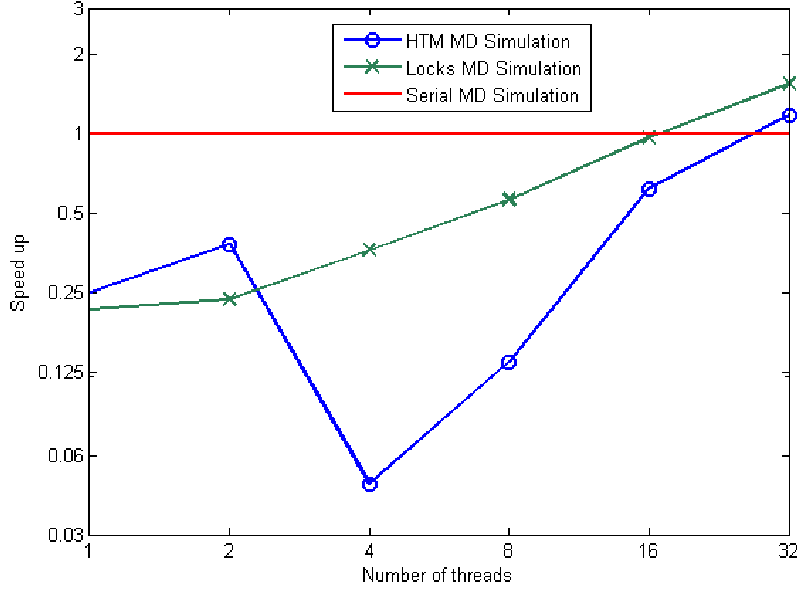


Figure 6: *The speedup of the average run time over all runs relative to serial time.*

the single threaded MD HTM program. This reduction of time by a factor of 2 from 16 to 32 threads indicates that the MD program might get a better speedup if the unexpected problem occurring at four and eight threads is solved.

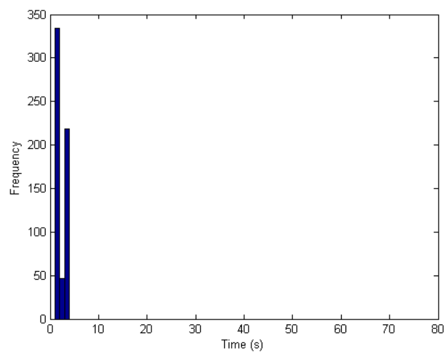
Table 4: *Timings of molecular simulation program. Corresponding serial times:* $T_{mean} = 0.913067$, $T_{min} = 0.910717$

(a) *Locks implementation*

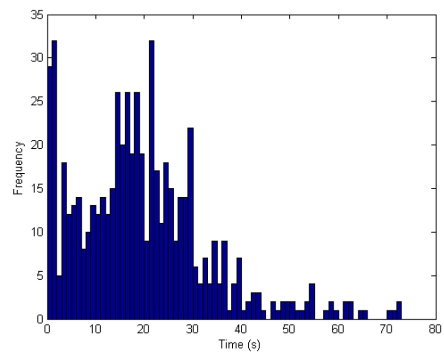
$N_{threads}$	T_{mean}	T_{min}
1	4.238470	4.170597
2	3.897253	2.719990
4	2.526680	1.933285
8	1.619485	1.189671
16	0.942010	0.928054
32	0.590960	0.583739

(b) *HTM implementation*

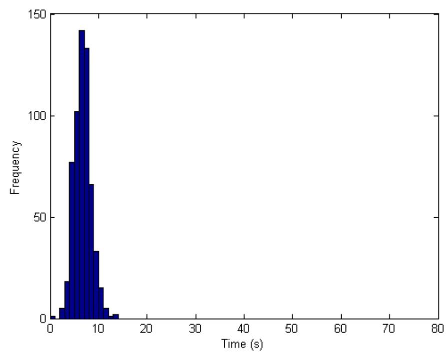
$N_{threads}$	T_{mean}	T_{min}
1	3.642271	3.528310
2	2.392158	1.856523
4	19.487336	0.983725
8	6.734671	0.980176
16	1.477065	0.688027
32	0.776266	0.327792



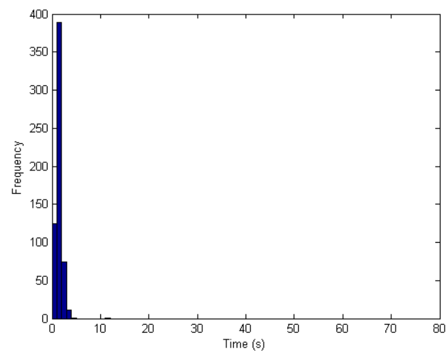
(a) *2 threads*



(b) *4 threads*



(c) *8 threads*



(d) *16 threads*

Figure 7: *Distribution of run times for 2, 4, 8 and 16 threads.*

6 Conclusions

The results attained from our experiments indicate that there might be classes of HPC applications suitable for TM. Our work suggests that these include the two applications studied in this paper. With TM implemented in hardware, we have seen that it has potential to perform very well, and not worse than the corresponding locks implementation. With the proper compiler support, TM may simplify programming of multi-threaded programs compared to a locks implementation. Currently, STM-aware compilers are capable of implementing transactions using just a few lines of code from the developer. An HTM-aware compiler would also be able to perform additional optimizations, possibly further enhancing the performance.

However, it is still too early to predict how well it can be expected to perform on a given problem. While some test runs scaled very well, often the run times could show a very large variation (as in the case of the four-threaded MD simulation). We are unable to judge whether this is due to problems inherent in TM, if it is due to the HTM implementation at hand, or if our application implementations are flawed. Further research is necessary to be able to make a final verdict of the future use of HTM in HPC applications. Investigation of a wide variety of applications, as well as applications of higher complexity, is needed. Likewise, a more in-depth study of the performance problems observed is a topic of further interest.

References

[1] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. *Proceedings of the 20th annual international symposium on Computer architecture*, pages 289 - 300, May 1993.

[2] Jayram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift and David A. Wood. Performance Pathologies in Hardware Transactional Memory. ISCA '07, June 9-13, 2007, San Diego, California, USA.

[3] C. Scott Ananian, Krste Asanović , Bradley C. Kuszmaul, Charles E. Leiserson, Sean Lie. Unbounded Transactional Memory. *MIT Computer Science and Artificial Intelligence Laboratory*, Feburary 2004

[4] Bratin Saha, Ali-Reza Adl-Tabatabai and Quinn Jacobson. Architectural Support for Software Transactional Memory. *MICRO '06*

[5] Yossi Lev, Mark Moir and Dan Nussbaum. PhTM: Phased Transactional Memory. *TRANSACT 2007*

[6] Mats G. Larson, Fredrik Bengzon. The Finite Element Method: Theory, Implementation, and Practice. *Springer*, December 18, 2009