Parallelizing a Real-Time Physics Engine Using Transactional Memory

Jaswanth Sreeram and Santosh Pande

College of Computing, Georgia Institute of Technology
`jaswanth@gatech.edu, santosh@cc.gatech.edu`

Abstract. The simulation of the dynamics and kinematics of solid bodies is an important problem in a wide variety of fields in computing ranging from animation and interactive environments to scientific simulations. While rigid body simulation has a significant amount of potential parallelism, efficiently synchronizing irregular accesses to the large amount of mutable shared data in such programs remains a hurdle. There has been a significant amount of interest in transactional memory systems for their potential to alleviate some of the problems associated with fine-grained locking and more broadly for writing correct and efficient parallel programs. While results so far are promising, the effectiveness of TM systems has so far been predominantly evaluated on small benchmarks and kernels.

In this paper we present our experiences in parallelizing ODE, a real-time physics engine that is widely used in commercial and open source games. Rigid body simulation in ODE consists of two main phases that are amenable to effective coarse-grained parallelization and which are also suitable for using transactions to orchestrate shared data synchronization. We found ODE to be a good candidate for applying parallelism and transactions to - it is a large real world application, there is a large amount of potential parallelism, it exhibits irregular access patterns and the amount of contention may vary at runtime. We present an experimental evaluation of our implementation of the parallel transactional ODE engine that shows speedups of up to 1.27x relative to the sequential version.

1 Introduction

The trend towards multi-core and many core processors is pushing more and more applications towards parallelism and is spurring extensive research in concurrent programming models and languages. The potential performance benefits of extracting parallelism and the complexity of specifying efficient concurrent programs are both significant.

Applications that simulate the dynamics and kinematics of rigid bodies or physics engines are examples of applications that are known to have significant amount of parallelism but it this parallelism is often difficult to exploit owing to their complexity. Physics engines that support real-time interactive applications such as games are growing rapidly in sophistication both in their feature-set as

well as their design. The popular Unreal 3 game engine is known to consist of over 300,000 lines of code and as described in [12], parallelizing parts of it was a challenging endeavour. Traditional approaches to efficient shared data synchronization such as fine-grained locking are often impractical owing to the size and complexity of the application and the large amounts of hierarchical mutable shared state. On the other hand coarse-grained locking has been found to be too inefficient for maintaining the highly interactive nature of these applications. Further, using fine-grained locks in such applications extracts a significant price in terms of programmer productivity - a factor that deeply affects their commercial development cycle.

Researchers have suggested developing parallel programs in this domain using *transactional memory* to manage accesses to shared state [12]. Software or Hardware Transactional memory has been proposed as a relatively programmer-friendly way to achieve atomicity and orchestrate concurrent accesses to shared data. In this model programmers annotate their programs by demarcating atomic sections (using a keyword such as "atomic" in a language-based TM implementation or specific function calls to a library based TM). The programmer also annotates accesses to shared data within these sections. At run time, these atomic sections are executed speculatively and the TM system continuously keeps track of the set of memory locations each transaction accesses and detects conflicts. This conflict detection step involves checking if a value speculatively read or written has been updated by another concurrent transaction. If so then one of the two speculatively executed transactions is aborted.

Software Transactional Memory systems reduce the burden of writing correct parallel programs by allowing the programmer to focus simply on specifying where atomicity is needed instead of how it is achieved. Further, the benefits of TMs are most apparent when a) the rate of real data sharing conflicts at run time is quite low i.e., most of the concurrent accesses to shared data are disjoint and b) using fine grain locking is difficult either due to the irregularity of the access patterns or the data structures. There has been a substantial amount of interest in hardware and software transactional memory systems recently. However in spite of this recent interest and the significant amount of research most of the studies investigating the use and optimization of these systems have been limited to smaller benchmarks and suites containing small to moderate sized programs [3,4,8,9,6]. Previous studies [18,7] have noted the lack of large real-world applications that use transactional memory without which an effective evaluation of the effectiveness of TM systems in realistic settings becomes difficult.

In this paper we present our experiences in parallelizing and using transactions in the Open Dynamics Engine (ODE), a single-threaded real-time rigid body physics engine [2]. It consists of roughly 71000 lines of C/C++ code with an additional 3000 lines of code for drawing/rendering. In [7] the authors outline a set of characteristics that are desirable in an application using TM. Briefly they are:

1. Large amounts of potential parallelism: As we show in the Section 3, there is a significant amount of data parallelism in the two principal stages in an ODE simulation.
2. Difficult to fine-grain parallelize: ODE exhibits irregular access patterns many structures that can be accessed concurrently.
3. Based on a real-world application: ODE is used in hundreds of open-source and commercial games [2].
4. Several types of transactions: The parallel version of ODE we describe in the rest of this paper has critical sections that access varying amount of shared data, have sizes that vary widely and the amount of contention between them changes during execution.

We started with the single-threaded implementation of ODE and found that the two longest running stages in a time step could be parallelized effectively. While we found many opportunities for fine-grained parallelization at the level of loops in constraint solvers, we choose to focus on a coarser-grained work offloading in order to amortize the runtime overheads. We then modified this parallel program by annotating critical sections and accesses to shared data with calls to an STM library. Our modifications added roughly 4000 lines of code in the ODE.

The rest of this paper is organized as follows: Section 2 presents an overview of collision detection and dynamics simulation in ODE. Section 3 describes the parallelization scheme for ODE and the usage of transactions for atomicity. Section 4 briefly discusses a few issues pertaining to the parallelization. Section 5 presents our experimental evaluation of the application. Related work is presented in Section 6 and Section 7 concludes the paper.

Algorithm 1. Overview of a time step in ODE

```
 1: Create world; add bodies
 2: Add joints; set parameters
 3: Create collision geometry objects
 4: Create joint group for contact points
 5: // Simloop
 6: while (!pause && time < MAX_TIME) do
 7:    Detect collisions; create joints
 8:    Step world
 9:    Clear joint group
10:    time++
11: end while
```
---

2   ODE Overview

At a high level ODE consists of two main components: a collision detection engine and a dynamics simulation engine. Any simulation involving multiple bodies typically uses both these engines. The sequence of events in a typical time step is shown in Algorithm 1. The goal is typically to simulate the movement

of one or more bodies in a *world*. Before simulation begins the world and the bodies in it are created and any initial joints are attached. A *contact group* is created for storing the contact joints produced during each collision. During each time step in the simulation loop in line 6, *collision detection* is first carried out which creates contact points/joints which are used in "stepping" or *dynamics simulation* for each body in the world (line 8). After this step all the contact joints are removed from the contact group and the simulation proceeds to the next time step.

2.1   Collision Detection

The collision detection (CD) engine is responsible for finding which bodies in the simulation touch each other and computing the *contact points* for them given the shape and the current orientation of each body in the scene. A simple algorithm would simply test whether each of the "n" bodies collides with any other body in the scene but for large scenes this $O(n^2)$ algorithm does not scale. One solution to this problem is to divide the scene into a number of *spaces* and assign each body to a space. Additionally, the spaces may be hierarchical - a space may contain other spaces. Now, collision detection proceeds in two phases called *broadphase* and *narrowphase* which are as follows:

1. **Broadphase:** In this phase each space $S_1(\in S)$ is tested for collision with each of the other spaces. If S_1 is found to be potentially colliding with space $S_2 \in S$ then S_1 is tested for collision with each of the spaces or bodies inside S_2.
2. **Narrowphase:** In this phase individual bodies that have found to be potentially colliding in the broadphase are tested to check if they are actually colliding.

This approach is similar to the hierarchical bounding box approach used for fast ray tracing and many other problems. If a pair of bodies are found to be colliding the collision detection algorithm finds the points where these bodies touch each other. Each of these contact points specifies a position in space, a surface normal vector and a penetration depth. The contact points are then used to create a joint between these two bodies which imposes constraints on how the bodies may move with respect to each other. In addition to links to the bodies each of these *contact joints* connect, they also have attributes like surface friction and softness which are used in simulating motion in the next step.

By the end of the collision detection step all the contact points in the scene have been identified and the appropriate joints between bodies made. In the dynamics simulation step below, the new positions and orientations of all the bodies in the scene are computed.

2.2   Dynamics Simulation

The joint information computed in the CD step above represents constraints on the movement of the bodies in the scene (for example due to another body

in way or due to a hinge). The Dynamics Simulation (DS) engine takes this joint information and the force vectors and computes the new orientation and position for all the active bodies in the scene. It does this by solving a Linear Complementarity Problem (LCP) using a successive over-relaxation (SOR) form of the Gauss-Seidel method. The main output produced in the DS stage are the linear and angular velocities of each body in the scene. These velocities are then used to update the position and orientation of the bodies.
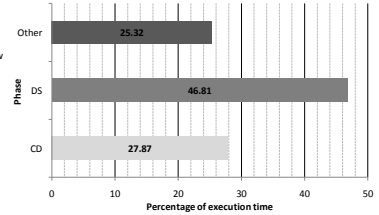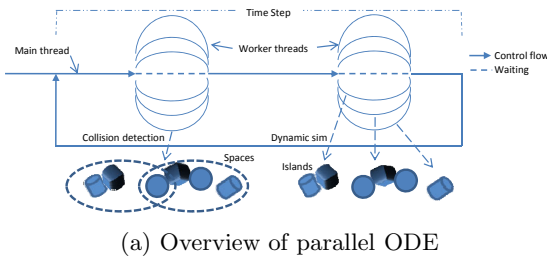

(a) Overview of parallel ODE

(b) Distribution of execution time among phases in single-threaded execution

Fig. 1. ODE overview

3   Parallel Transactional ODE

The broad approach to parallelizing ODE is illustrated in Figure 1a. At a high-level parallelism is achieved by offloading coarse-grained tasks in the CD and DS stages on the main thread onto concurrent worker threads that use transactions to synchronize shared data accesses.

3.1   Global Thread Pool

In order to avoid the overheads of creating and destroying threads, before the simulation begins the main thread creates a global thread pool consisting of t POSIX threads that are initialized to be in a *conditional wait* state. Additionally the pool contains a t-wide status vector that describes each thread's status, a set CM of t mutexes and a set CV of t condition variables. During the course of the simulation the main thread offloads work to a worker thread by scanning the pool for an idle thread, marshalling the arguments and setting the condition variable for the thread to start execution.

3.2   Parallel Collision Detection Using Spatial Decomposition

Detecting collisions between bodies in the world is inherently parallel and indeed the naive $O(n^2)$ algorithm described above can be parallelized by simply performing collision detection for each pair of bodies in a separate thread. However a better scheme would involve a more coarse-grained distribution of work

in which a space or a pair of spaces in the world is handled by a separate thread. Before the parallel CD stage starts each of the bodies in the world is assigned to a space S_i. Let S represent the set of spaces in the world i.e., $S = \bigcup_i S_i$. Detecting collisions among bodies contained in the same space can be done independently of (and in parallel with) other spaces. Additionally, detecting collisions between each distinct pair of spaces can be done in parallel. The broadphase stage of parallel CD proceeds as follows.

1. The main thread picks an unprocessed pair of spaces S_1 and S_2 and signals an idle thread $t_{1,2}$ in the thread pool to perform collision detection on them. Additionally the main thread signals idle threads t_1 and t_2 to perform collision detection on bodies contained withing S_1 and S_2 respectively.

2. Thread $t_{1,2}$ first checks if spaces S_1 and S_2 can potentially be touching. It does this by checking if there is an overlap between their axis aligned bounding boxes (AABBs). As described above, the AABB for a space informally is simply the smallest axis aligned box that can completely contain all the bodies in that space. If there is overlap between the AABBs of the two spaces then $t_{1,2}$ has to check if there exist bodies b_1 and b_2 such that $b_1 \in S_1$, $b_2 \in S_2$ and the AABBs of b_1 and b_2 overlap. If they do, b_1 and b_2 are potentially colliding and the narrowphase later on checks if they are actually colliding. After this step thread $t_{1,2}$ marks the space pair $(1, 2)$ as processed.

3. Thread t_1 finds bodies in S_1 that are potentially colliding. This is done again by analyzing the AABBs of bodies in S_1. Thread t_2 does the same for bodies in S_2. Spaces S_1 and S_2 are then marked as processed by their respective threads.

4. All the potentially colliding bodies found above are checked to find actual collisions in the narrowphase. If a pair of bodies do actually collide the appropriate thread computes contact points for the collision (using the positions and orientations of the bodies). These contact points are used by the thread to create contact joints between the pair of bodies.

This approach to assigning collision spaces to threads makes $(\binom{n}{2} + n)$ thread offloads where n is the number of spaces. An alternate approach is to assign a single thread t_i to each space S_i. This thread computes the collisions for objects within S_i and then performs broadphase and narrowphase collision checking between S_i and all S_j such that $i < j \leq n$. This approach activates only n threads but is likely to be more efficient than the former only if the spaces are *well balanced*. That is all the spaces at each level in the containment-hierarchy contain approximately the same number of subspaces or bodies. Consider a deep space hierarchy with space S_{root} as the root space that contains all other spaces S_i and bodies. In the alternate approach the thread t_{root} has to process collisions between S_{root} and all other spaces/bodies. By definition, S_{root} would collide with every other contained body or space. Thus in general this approach would result in a schedule where threads processing spaces that are high-up in the hierarchy are heavily loaded while threads assigned to spaces that are lower are lightly loaded. However in the former approach, each space-space pair can be processed

in parallel - each pair $\{S_{root}, S_j\}$ for $1 < j \leq n$ can be processed in parallel thereby reducing the overall imbalance.

Shared data

Although the collision detection stage described above is quite parallel the participating threads make concurrent accesses to several shared data structures that must be synchronized. The important data structures that are accessed concurrently are the Global Memory buffer that is used to satifsy allocation requests, the joint, contacts and body lists and attributes pertaining to the state of the world and its parameters including the number of active bodies and joints.

We use an STM library to orchestrate calls to these shared data. STM enables efficient disjoint access parallelism - two concurrent threads that do not access the same memory word can execute in parallel. This is in contrast to using more pessimistic coarse-grained locking in which a thread that *could* access/modify shared data (being accessed by some other thread) has to wait to acquire the appropriate lock regardless of whether an actual access takes place or not. The STM library we used is based on the well-known TL2 system described in [1]. In other works such as [18] the authors used an automated compiler-based STM system in which the programmer simply annotates atomic sections and the compiler automatically annotates accesses occurring inside them with calls to the TM runtime. Instead we used the TL2 library based system which means the programmer has to manually identify atomic sections and accesses occurring within them. This choice is because of two reasons. Firstly the TL2 STM has been shown to have lower overheads than other comparable STM systems in several studies [1]. This is especially important since we are using it in the context of a real-time interactive application. Secondly using a library STM offers better flexibility and we are in some cases able to reduce TM overheads by using domain knowledge to elide TM tracking of specific shared data.

3.3   Parallel Island Processing

Island Formation

After the joints in the world have been determined in the CD step the next stage is dynamics simulation or simulating the motion of the bodies under the constraints specified by their shapes and the joints found. This uses the SOR-LCP formulation mentioned above and finding solutions to this problem involves several nested loops that are compute-intensive. However, parallelizing these loops with the work-loading model would result in a very fine-grained parallel system (which is unlikely to scale well [11] and the overheads of synchronization and thread control would likely eliminate any speedups gained. Therefore we choose a more coarse-grained approach in which several connected bodies are processed independently and in parallel with other bodies. All the bodies in the world are assigned to "islands". An island is simply a group of bodies in which each body is connected to one or more bodies in the same island through one or more joints. These islands therefore represent sets of connected bodies that can be processed separately since simulating a body (with some number of joints) does

not require accesses to bodies in other islands. In parallel dynamics simulation the main thread first forms islands. The algorithm iterates over all the bodies in the world adding bodies to islands if they haven not already been added. A body is said to be *tagged* when it has been added to some island. Given a body b, the algorithm first finds the untagged neighbors of b and adds adds them to a stack. The algorithm then pops and examines each body in this stack, adding their untagged neighbors. The joints between all these neighbors are collected in a joint list. When the stack is empty, the joint and body lists represent an island of connected bodies that can be processed. The main thread then moves on to the next untagged body in the world in the outermost loop.

Island Processing
While island formation is sequential, processing the bodies in each island can be performed independently of other islands. Immediately after an island is formed, the main thread uses heuristics to check whether the island is suitable to be offloaded to a worker thread. If so, the main thread marshals pointers to body and joint lists for that island, finds an idle thread in the global thread pool and signals it to start processing that island. The main thread then resumes with finding the next island. If the island formed is deemed to be not suitable for offloading, the main thread can process that island itself before continuing with further island formation. A variety of heuristics can be used to decide whether a particular island should be processed in a worker thread or if it should be processed in the main thread. Our system uses a threshold on the number of bodies and number of joints in the island. Because of the overhead of offloading computation to worker threads, if there are very few bodies or joints in the islands then it may be more efficient to process them in the main thread instead. Additionally, if an island is found to have fewer bodies than needed to offload processing to a worker thread, the main thread checks whether the next island in combination with the previous one meets the threshold. If so both these islands are offloaded together to a single worker thread. The main thread chooses and signals a thread from the global thread pool to start island processing. The worker thread uses the body and joint lists and the force vectors to set up a system of equations representing the constraints on the set of bodies and finds. We refer the reader to [2] for details of the constraint solver that is used for finding solutions. The island processing step finishes after computing new values for linear and angular velocity, position and orientation quaternion for each body in the island and atomically updating body with these values.

3.4   Phase Separation

During body simulation in ODE, all the contact joints are typically computed first before dynamics simulation can start since the latter needs these joints to be able to solve the constraint satisfaction problem. In the sequential case this was guaranteed since the dynamics simulation is always preceded by collision detection in each time step. However in the parallel case, the main thread can simply offload the collision detection to worker threads and enter the dynamics

simulation step while some of the worker threads are still computing the joints. Therefore there needs to be a thread barrier between the collision detection and dynamics simulation in simulating each time step. The control flow for the main thread is very different from that of the worker threads in our parallelization scheme. Therefore instead of a normal thread barrier that is released when all threads reach a certain program point, in our scheme we use a thread *join point* in the main thread. A *join point* is simply a program point at which the main thread waits for all the active worker threads to finish executing. When the main thread enters the join point, it repeatedly polls the *status* vector and yields its processor if there is at least one worker thread performing collision detection. Note that no lock acquisition is necessary for this polling as the worker thread only ever writes one type of value into its slot in the status vector - the value representing its *IDLE* state. After all worker threads have finished collision detection and have entered the *IDLE* state, the join point is met and the main thread is released. Although it limits parallelism, this join is necessary due to the producer-consumer relation between the stages for joints - the island formation algorithm requires contact joints for all bodies in the world to have been computed.

After island processing has generated new positions and orientations for all the bodies in the world, these new values are used in the collision detection step in the next stage. But after the main thread offloads island processing to worker threads, it could enter the collision detection stage in the next time step while the new body attributes are being computed. This could result in the collision detection stage reading stale position/orientation values for some bodies - the bodies which island processing has not yet updated. Therefore in addition to the dependence between the collision and dynamics simulation steps within a time step there is also a dependence between the dynamics simulation in one time step and the collision detection in the next. We therefore enforce a join point at the end of each time step to make sure that all bodies have been updated. This join point is implemented like the one described above - the main thread simply polls the status vector until all the island processing worker threads have finished.

To see why this join point is needed consider the case of a worker thread with transaction Tx_1 updating the position quaternion R_b of a body b during island processing in time step n. Assume the main thread is allowed to enter the next time step where it offloads collision detection to a worker thread and transaction Tx_1 is reading R_b. If Tx_1 commits after Tx_2 starts but before it finishes then Tx_2 is aborted when the conflict for R_b is detected and the join point would *not* have been necessary. However if Tx_2 commits before Tx_1 does, then Tx_1 is aborted and retried. Thus Tx_1 eventually produces the new value for R_b but Tx_2 ends up using the older value and this phenomenon can adversely affect simulation integrity. Now lets say add a "*last_updated*" field to each body which is updated in Tx_1. So if Tx_2 finds this field for b to be n then Tx_1 is guaranteed to have committed and Tx_2 can read the latest R_b. However if this value is $n-1$ then Tx_2 can be forced to abort to until Tx_1 commit. It may therefore be possible to eliminate the join point at the end of each time step by forcing transactions

reading stale values in the next time step to abort. This could potentially allow more parallelism by allowing the threads with transactions that only read already updated bodies to proceed instead of waiting for the other threads.

3.5   Feedback between Phases

A critical factor influencing the amount of effective parallelism achieved during the CD phase is the assignment of bodies to spaces. Spatial (in the geometric sense) assignment methods are popularly used in many dynamics simulation algorithms. In such methods, objects that are geometrically proximal to each other are assigned to the same space in the containment hierarchy. An important concern with this approach is that the scene being modelled may evolve to a state where most of the objects are contained in one or a few spaces. This may in turn result in the *thread imbalance* problem discussed in Section 3.2. To address this such methods usually propose a space reassignment step that is invoked occasionally and reassigns objects such that the threads are once again balanced. We use a novel method to perform space assignment that reduces imbalance. Our method is based in the observation that the DS phase in a timestep already computes entities (islands) of geometrically close bodies - in fact the bodies in each of these islands are touching each other! After the dynamics simulation step, the bodies in these islands have been moved so they may not be touching anymore. However if the simulation timestep is small then in the CD phase in next iteration these bodies are either still touching each other or are close to each other. Hence the CD phase bootstraps spaces with clusters of such islands before performing broadphase checks on these spaces with the result that there are fewer narrowphase checks to be performed on the contained bodies.

4   Issues

In this section we will discuss a few issues pertaining to using transactions for synchronization in parallel ODE.

4.1   Conditional Synchronization

Our implementation of parallel ODE makes extensive use of conditional synchronization for signalling between threads. Indeed constructs such as `pthread_cond_wait` and `pthread_cond_signal` enable efficient waiting, signalling and other communication between threads. However these constructs require the communicating threads to acquire/release locks during doing so. Moreover there is no direct way to transform these critical sections into transactional atomic sections. Consider the case of a worker thread t_w waiting for the main thread t_M to offload work. The thread t_w first acquires a lock on the *waiting mutex l* and calls `pthread_cond_wait(..,l)`. This call atomically unlocks the mutex and starts the conditional wait. To signal thread t_w to start execution, the thread t_M in turn acquires a lock on l, calls `pthread_cond_signal()` and releases the lock

on l. If the critical section protected by the lock acquisition/release in t_M were to be transformed into an atomic section using transactions, then if there is a conflict in the transaction in t_M the transaction cannot roll back since the signal has been set and it is irrevocable. Most STM systems including the TL2 system we used and the compiler-based STM in [10] do no provide transactional methods for conditional synchronization and signalling. Consequently our implementation uses traditional mutex based methods for conditional synchronization.

4.2 Memory Management and Application Controlled alloc/de-alloc.

Dynamic memory allocation is another important programmatic concern for STMs. Most STM systems provide methods for allocating and deallocating memory efficiently from within transactions. Additionally they often implement a large memory buffer from which allocations are made and of course memory that is allocated in a failed transaction is restored back to the buffer. Many of the important classes of objects in ODE are allocated dynamically on the heap. This includes bodies, joints, joint lists, and other shared data. However, ODE implements its own memory allocation/deallocation algorithms that purport to improve locality and to allow objects to be be efficiently garbage collected in addition to implementing its own large stack-shaped buffer from which allocation requests are met. Requests for memory allocations are made using the `ODE_Alloc()` which simply returns a pointer to the first location in memory that has not previously been allocated. If concurrent transactions in two different threads call `ODE_Alloc` at the same time, both may receive the address of the same location in memory. And as with all transactional writes to shared data, the modifications they make to this newly allocated memory region will be buffered in their respective private write-buffers. Suppose one of them finishes and commits successfully. At this point its modifications to the heap will actually be written to memory. When a conflict is detected when the second transaction tries to commit it will be aborted. As the TM runtime rolls this transaction back, the memory allocated within it will be freed thereby freeing memory that the first transaction is using. Therefore the memory allocation/deallocation library should be modified to be aware of the revocable nature of allocations. For programs that may make use of such routines from one or more of several external libraries this is a significant problem.

5   Experimental Evaluation

We used the parallel ODE library in to drive an application simulating a scene with approximately 200 colliding rigid bodies (a modified version of the `crash` program in the ODE distribution). The maximum number of worker threads in the global thread pool was varied from t = 1 to 32 in powers of 2. The number of

(a)   (b)
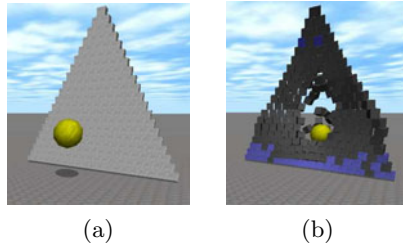
Fig. 2. Scene used in evaluating parallel ODE

threads in the results below therefore represents the maximum number of worker threads available to for offloading and the maximum number of active threads at any instant is (t+1) including the main thread.

We used the TL2 (v0.9.6) STM [1] API and library to provide support for transactions in the ODE library as well as in the driver application program. This version of TL2 is a *word-based write-buffering* STM that uses *lazy version checking* for detecting conflicts and *commit-time locking*. All experiments were carried out on a machine with an Intel Xeon dual processor with two cores per processor and with hyperthreading turned on on all cores (for a total of 8 thread contexts). This in our opinion represents an average platform that may be used to run interactive simulations in ODE. Machines with higher core counts such as (8 or 16) are less common (although they are available) and servers with core counts of 32 and more are less frequently used in running these predominantly desktop oriented simulations. Each core on this machine had a private 32K L1-D cache, 32K L1-I cache, a shared 256KB L2 per processor and a shared 8MB L3 cache and the machine was equipped with 6GB of physical memory. Each thread in our experiments was bound to exactly one core. We compiled all libraries and the driver application with g++-4.3.3 using the default flags and all experiments were run on Ubuntu Linux 2.6.28. All running times were gathered using the `gettimeofday()` call.

(a) Performance relative to single-threaded ODE

(b) Effect on Frames per Second
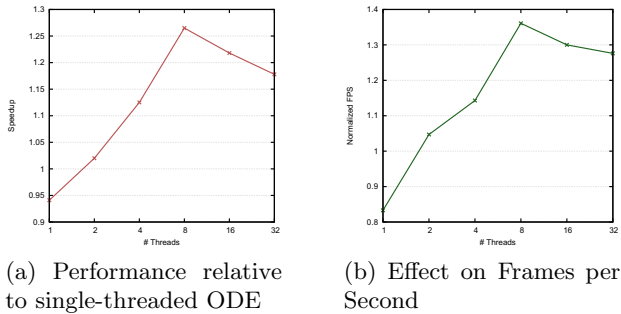
Fig. 3. Scalability

5.1   Execution Time

The graph in Figure 3a shows the improvement in execution time as speedup over the single-threaded execution time. The X-axis is the *maximum number of threads available for offloading.* The speedup scales until 8 threads at which point it is roughly 1.27x. At 16 and 32 threads it drops to roughly 1.22x and 1.18x approximately. This means that the heuristics may be too aggressive in offloading work when idle threads are available. This hurts performance since there may not be enough work for a worker thread (not enough joints or bodies in island processing for example) to justify the overhead of offloading. Moreover, at 16 and 32 threads each core is utilized by 2 and 4 threads respectively which means increased contention may also be responsible.
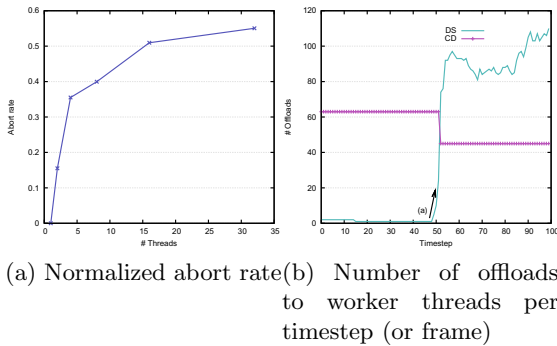

(a) Normalized abort rate (b) Number of offloads to worker threads per timestep (or frame)

Fig. 4. Aborts and Offloads

5.2   Frame Rate

Figure 3b shows the number of frames processed per second (FPS) against the number of threads in the thread pool. In our experiments each time step corresponds to one frame. The frame rate scales in a trend similar to that of execution time speedup. The improvement in frame rate peaks at 1.36x and drops to 1.27x for 32 threads. At more than 8 threads more than one thread is mapped to a processor and contention for shared data also increases reducing the per frame completion time.

5.3   Abort Rate

The abort rate for different number of threads is shown in Figure 4a. The abort rate is defined as the ratio of the number of aborts to the total number of transactions started. Therefore if a, c represent number of aborts and commits, the abort ratio is given by $a/(a + c)$. The abort ratio increases steeply up to 4 threads and continues to rise beyond. The average amount of contention between threads increases as the number of threads increases and the amount of shared data being accessed by these threads remains the same. The abort rate does not

Table 1. Read/Write set sizes

	Reads (bytes)				Writes(bytes)			
Threads	Min	Max	Avg	Total	Min	Max	Avg	Total
1	4	112	112	3094332	4	96	48	1325062
2	4	224	211	5886756	0	192	90	2520386
4	4	2536	596	16620560	0	2036	240	6791206
8	4	2868	1300	36245344	0	2328	530	14775982
16	4	3552	1393	38823380	0	2936	570	15868776
32	4	5184	1504	41912768	0	4196	614	17133684

rise as significantly going from 16 to 32 threads. This is because the average number of *concurrent* threads does not necessarily rise proportionally to the number of threads in the thread pool and therefore the number of aborts increase less steeply.

5.4   Thread Utilization

In contrast to parallelization techniques that purely depend on static decomposition of work, in the scheme for parallel dynamics simulation (DS) described above, only the maximum number of threads in the thread pool is fixed and heuristics are used to dynamically gauge whether to offload island(s) processing to worker threads. The amount of parallelism in the collision detection (CD) stage however remains relatively uniform. The plot in Figure 4b shows the average number of computation offloads occurring in each time-step (or frame) when there are a maximum of 32 threads in the global thread pool. Specifically, the plot shows the number of offloads to worker threads for the first 100 frames of simulation for the scene shown in Figure 2. The number of offloads in the CD stage remain stable and in this stage, a worker thread can be invoked on average roughly 2 times until the point in the simulation noted as *(a)* in the plot. Also, the number of offloads in the DS stage remains low and is also stable until point *(a)*. This is the time step where the stack of bodies in Figure 2 begins to disintegrate as shown in Figure 2(b). While in earlier time steps there was only one island to process, after point *a* there are many smaller islands and therefore there is more parallelism. This is reflected in Figure 4b by the sharp increase in number of offloads in the DS stage after point *(a)*. As mentioned above, the heuristics we used have a relatively low threshold on island count for offloading the work of processing an island to a worker thread. This results in the main thread aggressively offloading work which explains the high number of DS offloads after point (a). The number of offloads in the CD stage remain relatively stable since there the data distribution is based on abstract spaces and not physical artifacts such as joints and islands. Additionally, after point *(a)* the number of offloads in the CD stage are reduced due to contention with the DS stage for worker threads.

5.5   Transaction Read/Write Sets

There are three main types of transactions during execution. The first is the transaction to add a contact joint to the system for a pair of colliding bodies. The second transaction executed during island processing for atomically updating a body's attributes. The third type consists of short transactions to access various shared values such as the number of joints. Table 1 summarizes the characteristics of the read/write sets of all the transactions executed. The average read set sizes are significantly larger than the sizes of the write sets in all cases. This is in line with the average mix of read/write operations in many other transactional programs. Many of the transactions in parallel ODE perform several reads before performing their first write. One commonly occurring transaction for example is atomic insertion into a sorted object list. Here the list is traversed and each element examined to find the right position for insertion before pointer values for the neighboring list elements are updated. The average read and write set sizes remain relatively small for most transactions which shows that hardware transactional memory implementations may also be able to support parallel ODE.

5.6   Scalability Optimizations

Based on the results of the experiments described above, the following observations can be made pertaining to improving scalablity.

1. DS phase offloading: The work offloading algorithm in the island processing phase may be too aggressive in our experimental system. This stems partly from the static threshold used to decide whether processing for a particular island is to be offloaded, inlined or whether it should be combined with another island and then offloaded. The size of the islands changes substantially over the course of the simulation (for example, the one shown in Figure 2a), which results in the threshold becoming too low at several points. A low threshold results in aggressive offloading which in turn results in poor scalability. The processing step for a single island cannot be offloaded to more than one thread in our system. This is because the forces and torques acting on a body are determined by the joints connecting the body to its neighboring bodies and if these bodies were being processed by two separate threads the system of constraints imposed by these joints would have to be communicated between them which we believe would increase the level of synchronization drastically. During the early timesteps of simulating the scene shown above, there are only two islands with one of them containing all the bodies in the world and this large island is then offloaded to a single thread. This restriction therefore has the effect of severely serializing island processing until more islands are formed as a result of collisions.

2. Speculative island formation: The algorithm for discovering islands discussed earlier is sequential - the main thread discovers an island and offloads (or inlines) it before proceeding to discover the next island. This substantially

limits the amount of effective parallelism especially for very large scenes. An algorithm for speculatively discovering islands in parallel and processing them in the worker threads after the speculation has been verified would improve parallel performance greatly (in spite of the additional synchronization costs which are relatively small). Briefly, in this algorithm worker threads speculate on a "seed" body for an island and then "grow" the island. This seed body is picked from a cache of likely candidates built during the island discovery phase in the previous timestep. The worker threads then attempt to verify if the island is valid and was previously undiscovered and if so, continue to the island processing step.

3. Performance of Locks: Coarse-grained locking can be used instead of transactions to protect accesses to shared state and we believe that the performance in both cases would be comparable. Fine-grained locking would be harder to implement given the diversity of both the data structures and the accesses to them. Nevertheless we are in the process of implementing our parallel ODE system with support both coarse-grained and fine-grained locking.

6 Related Work

Several researchers have studied various aspects of parallelizing physics computations for applications from domains ranging from robotics, virtual environments and scientific simulations, to animation [16,13,19,14]. In [19] the authors describe a voxel based parallel collision detection algorithm for distributed memory machines. This algorithm is similar to the abstract space based collision detection scheme discussed in this paper. ParFUM [15] is a framework based on Charm++ for developing parallel applications that manipulate unstructured meshes and supports efficient collision detection. In [6] the authors study the performance of a parallel implementation of the Barnes-Hut algorithm for n-body simulation that uses octree based subdivision for computing particle interactions. In [17] the authors present an algorithm for *continuous collision detection* between deformable bodies that can be executed at interactive rates on present day multicore machines.

Lee-TM [7] is an implementation of Lee's routing algorithm using transactional memory. While the algorithm exhibits large amount potential parallelism the transactional implementation has been shown to have modest scalability. AtomicQuake [18] is an implementation of a parallel Quake game server using transactions. The parallelization is at the level of clients connected to the server - operations for a client are performed on the server by the worker thread that the client is mapped to. Support for transactions is provided by the compiler [10] instead of a library based TM. The programs in STAMP [3] consist of a variety of parallel transactional workloads that represent pieces of larger applications and which can be executed with one of several STM or HTM systems. TMunit [9] is a framework for developing unit tests for evaluating STM systems. RMS-TM [8] is a TM benchmark suite consisting of programs and application kernels. STMBench [5] is a synthetic benchmark that that contains transactions with

widely varying characteristics and which operate on non-trivial data structures. Thus while it is very useful for finding problems with specific implementations and stretching the limits of TM designs, it is not representative of any real-world program.

7   Conclusion

In this paper we presented a parallel transactional physics engine for rigid body simulation based on the popular Open Dynamics Engine (ODE). We were able to parallelize the two principal components of ODE - the collision detection engine and the dynamics simulation engine to make use of worker threads from a global thread pool for executing work offloaded from the main thread. We used a software transactional memory for orchestrating concurrent accesses to all shared data. Our approach of coarse-grained parallelization was not only relatively programmer friendly but also helped amortize the cost of the work-offloading. The parallel version of ODE showed speedups of up to 1.27x (for 8 threads) compared to the sequential version. As a continuation of this work we plan to investigate better cost heuristics for making offloading decisions and to investigate techniques for incorporating domain knowledge in optimizing memory transactions in addition to comparing the performance of the transactional implementation with that of versions that use fine-grained and coarse-grained locking.

References

1. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: Proceedings of the 20th International Symposium on Distributed Computing (DISC), Stockholm, Sweden (September 2006)
2. Open Dynamics Engine, http://ode.org
3. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: IISWC 2008, pp. 35–46 (2008)
4. Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: A benchmark for software transactional memory. In: Proceedings of the 2nd European Systems Conference (March 2007)
5. Carey, M.J., DeWitt, D.J., Kant, C., Naughton, J.F.: A status report on the OO7 OODBMS benchmarking effort. In: OOPSLA 1994: Proc. 9th Annual Conference on Object-oriented Programming Systems, Language, and Applications, pp. 414–426 (October 1994)
6. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture
7. Ansari, M., Kotselidis, C., Jarvis, K., Lujan, M., Kirkham, C., Watson, I.: Lee-TM: A Non-trivial Benchmark for Transactional Memory. In: Proc. 7th International Conference on Algorithms and Architectures for Parallel Processing (2008)
8. Kestor, G., Stipic, S., Unsal, O.S., Cristal, A., Valero, M.: RMS-TM: A Transactional Memory Benchmark for Recognition, Mining and Synthesis Applications. In: 4th Workshop on Transactional Computing (TRANSACT) (2009)

9. Harmanci, D., Felber, P., Sukraut, M., Fetzer, C.: TMunit: A transactional memory unit testing and workload generation tool Technical Report RR-I-08-08.1, Universite de Neuchatel, Institute Informatique (August 2008)
10. Adl-Tabatabai, A.-R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and runtime support for efficient software transactional memory. In: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 26–37 (June 2006)
11. Reinders, J.: Intel Threading Building Blocks. O'Reilly Media (2007)
12. Sweeney, T.: The Next Mainstream Programming Language: A Game Developers Perspective. Invited Talk at the International Symposium on Principles of Programming Languages (2006)
13. Brown, S., Attaway, S., Plimpton, S., Hendrickson, B.: Parallel strategies for crash and impact simulations. In: Computer Methods in Applied Mechanics and Engineering, vol. 184, pp. 375–390 (2000)
14. Grinberg, I., Wiseman, Y.: Scalable parallel collision detection simulation. In: Proceedings of the Ninth IASTED International Conference on Signal and Image Processing (2007)
15. Lawlor, O.S., Chakravorty, S., Wilmarth, T.L., Choudhury, N., Dooley, I., Zheng, G., Kal, L.V.: ParFUM: a parallel framework for unstructured meshes for scalable dynamic physics applications. In: Engineering with Computers (December 2006)
16. Figueiredo, M., Fernando, T.: An Efficient Parallel Collision Detection Algorithm for Virtual Prototype Environments. In: 10th International Conference on Parallel and Distributed Systems (2004)
17. Tang, M., Manocha, D., Tong, R.: Multi-core collision detection between deformable models. In: SIAM/ACM Joint Conference on Geometric and Physical Modeling (2009)
18. Zyulkyarov, F., Gajinov, V., Unsal, O., Cristal, A., Ayguad, E., Harris, T., Valero, M.: Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. In: 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (Febuary 2009)
19. Lawlor, O.S., Kale, L.V.: A voxel-based parallel collision detection algorithm. In: Proceedings of the 16th International Conference on Supercomputing (2002)