Vis Comput
DOI 10.1007/s00371-011-0661-0

ORIGINAL ARTICLE

A pipeline virtual environment architecture for multicore processor systems

Eric Acosta · Alan Liu

© The Author(s) 2011. This article is published with open access at Springerlink.com

Abstract We present a novel architecture to develop Virtual Environments (VEs) for multicore CPU systems. An object-centric method provides a uniform representation of VEs. The representation enables VEs to be processed in parallel using a multistage, dual-frame pipeline. Dynamic work distribution and load balancing is accomplished using a thread migration strategy with minimal overhead. This paper describes our approach, and shows it is efficient and scalable with performance experiments. Near linear speed-ups have been observed in experiments involving up to 1,000 deformable objects on a six-core i7 CPU. This approach's practicality is demonstrated with the development of a medical simulation trainer for a craniotomy procedure.

Keywords Parallel virtual environment · Physics simulation · Multicore processors · Medical simulation

1 Introduction

Interactive computer-based virtual environments (VEs) have applications ranging from entertainment to medical simulation. Detecting interactions (e.g., collision and response) and computing updates are central requirements. To improve realism, interactions and updates can use physically-based models. However, computational requirements grow quickly as VEs become complex. For example, VEs used in surgical simulation incorporate inhomogeneous biomechanical tissue and physiological models. Haptic and visual responses to user interactions occur in real-time. The high degree of complexity and need for real-time updates can be challenging for single-threaded implementations.

Parallelization can provide a solution. Several works have focused on parallelizing specific aspects, such as collision detection [13, 21, 39, 44, 46], physical model updating [25, 29, 39], and rendering [11, 20, 22, 28, 30, 38, 45]. Other approaches provide a broader framework to define a VE and parallelize its workload [4, 10, 18, 19, 36, 40, 41].

The widespread availability of computers with multi-core processors are a recent development. Cores reside on a common die, but each core functions as a distinct processor. Large shared cache designs help alleviate memory bottlenecks. Configurations up to eight cores are available at moderate cost.

This paper describes a novel architecture to develop Virtual Environments (VEs) for multicore CPU systems. We note that the work performed in many VEs can be decomposed into three distinct phases: interaction, update, and rendering. These phases occur in sequence, and repeat for the duration of the VE. With this observation, we propose an object-centric approach to defining a VE. All elements of a VE are represented by objects. An object is defined by its response to each phase, and by its internal data structures. This uniform representation permits a VE to be executed by a multithreaded pipeline. Threads adopt a decentralized migration strategy to process work within the pipeline. No work scheduler is required. In contrast to many existing parallel VE works, our approach does not depend on work-stealing [18, 19] or run-time heuristics [40] to achieve dynamic load balancing. Thread utilization is further increased by processing up to two consecutive frames in the pipeline simultaneously.

E. Acosta (✉) · A. Liu
NCA Medical Simulation Center, USUHS, Silver Spring, MD, USA
e-mail: Eric.Acosta.ctr@simcen.usuhs.edu

A. Liu
e-mail: Alan.Liu@simcen.usuhs.edu

Published online: 14 December 2011

The results of several performance experiments are provided. A VE with varying complexity is used to evaluate the scalability and load balancing of the architecture under various conditions. A head-trauma simulator, which includes real-time deformable tissues and different model types, provides a real world application of the architecture. Results show that the architecture scales efficiently with the number of CPU cores. Near-linear speedups can be achieved with relatively low overhead. We compare our approach with established parallel VEs methods. They include Intel TBB [23], Cilk Plus [24], and OpenMP [35] parallel frameworks. The experimental results show our architecture provides greater speedups.

The remainder of the paper is as follows. Section 2 reviews related work. Section 3 illustrates how physically-based VEs can be described using a common framework. Section 4 describes our implementation of the VE framework. The VE execution pipeline is detailed in Sect. 5. An application development overview is given in Sect. 6. Section 7 describes performance experiments, with results given in Sect. 8. The paper is discussed in Sect. 9 and then concluded in Sect. 10.

2 Related work

In this section, we review parallel VE methods. Some early parallel VE designs resulted in application-specific approaches. For example, Agus et al. developed a parallel solution for simulating temporal bone surgery [3]. A static work partitioning strategy is adopted. Task-specific threads compute interactions, update, and render the VE.

Many works focus on parallelizing specific aspects of a VE, such as collision detection, physical model updating, and rendering. Collision detection has been parallelized in various ways. Figueiredo et al. rely on OpenMP [35] to parallelize loops that compute overlapping axis-aligned bounding boxes and surface intersections [13]. Wieland et al. use a quad tree to group potentially colliding objects, and then assign groups to threads to perform intersection tests [44]. Zhao et al. build a hybrid bounding box/sphere hierarchy and relying on OpenMP to parallelize the traversal of the hierarchy for collision testing [46]. Huagen et al. construct a hybrid bounding volume hierarchical representation of objects and accelerate collision detection by traversing the hierarchy in parallel [21].

A number of approaches have been described to compute model deformations in parallel. For example, Montgomery et al. parallelize mass–spring simulations [29]. Parallel linear and corotational finite element methods are also described [25]. Many recent works, such as [12], focus on using the GPU as a parallel processor for computing deformations. Thomaszewski et al. describe parallel methods for collision detection and time integration for physics simulation [39]. For collision detection, traversal of a bounding volume hierarchy creates tasks for threads. Collision information from a previous frame estimates work for the current frame to guide task granularity. Thread creation overhead is reduced by avoiding creating tasks whose grain size is too fine. A parallel time integration algorithm is also described. Additional parallelism is achieved via domain decomposition, by breaking up the geometry into different parts and assigning each part to a different thread.

Parallel rendering methods have also been explored. Some are domain-specific such as volume rendering [20, 38, 45]. Others focus on dedicated hardware solutions [28, 30]. More general parallel rendering solutions have also been created [11, 22].

In contrast to focusing on specific aspects, other methods provide a more generalized parallel VE framework. OpenScenegraph is a toolkit for developing VEs using a scene graph representation [36]. Multithreading within OpenScenegraph is heavily optimized to facilitate rendering and scene graph operations. OpenScenegraph only supports single-threaded updates since it would require synchronization on most scene graph operations, resulting in significant performance loss [37].

Voß et al. describe a framework to support multi-threaded updating in scene graphs [41]. Data is replicated whenever threads need to modify the data. Modified values are tracked to synchronize the copies. The method is extended for cluster support, by generating network messages from the modified data to synchronize other scene graph copies. While this method allows for updating with multiple threads, data replication and synchronization can be an issue since each thread can have its own copy of the data.

Allard et al. describe a parallel VE framework for physically-based VEs [4]. The underlying data flow model, provided by a FlowVR distributed VE middle-ware [5], is based on message passing between filters and modules. Parallelism is achieved by statically assigning modules to run one per processor (or host as a distributed VE). In some cases, such as fluid simulation, modules cannot simply be distributed to hosts so they rely on MPI [17] to run the modules on several processors. The data-flow model makes it possible to couple different physically-based simulations, but does not work well with "tightly coupled simulation algorithms" [4].

DLoVe provides a method for developing parallel applications that are distributed over several machines [10]. Relationships between objects are defined using a constraint graph. Changes are propagated through the graph's links to ensure all relationships are satisfied. Worker executables are responsible for keeping assigned parts of the constraint graph up-to-date. At compile-time, DLoVe's partition algorithm determines how to partition the graph by assuming all

links are equally computationally expensive. Tasks, corresponding to a set of interconnected links, are assigned to the workers. A greedy round-robin scheduling algorithm tries to provide some limited load balancing as work is statically assigned to workers.

Vo et al. [40] describe a method to parallelize visualization pipelines. The method is built on top of the Visualization Toolkit [42]. A scheduler thread builds a dependency graph from modules and generates a priority queue for execution. A secondary scheduler thread decides which modules to process and allocates threads for execution. The number of threads is based on a heuristic strategy, which collects accumulated computation run-times for modules.

Hermann et al. describe a method for dynamic load balancing for the time integration step of physics simulations [18]. The method relies on a KAAPI middleware [15] to parallelize the Sofa framework [6]. Prior to running a simulation step, a data dependency graph between tasks is generated. The graph is used to statically partition tasks and map them onto processors. The processors execute tasks in parallel, and can suspend execution if required to respect data dependencies (synchronization). KAAPI's work-stealing can redistribute tasks if required for load balancing. In [18], KAAPI's static scheduling algorithm is modified to reuse a previous simulation step's graph. However, a new graph must be created when new collisions occur. The cost of generating another task graph is proportional to the number of contacts in the scene. Load balancing can be performed explicitly at the object-level by moving a computationally expensive object to an idle processor.

In [19], Hermann et al. expand their work to allow use of GPUs and CPUs to execute the time integration step for physics simulations. A task graph is used to guide static work partitioning based on grouping interacting objects. During a "warm-up" phase, CPU and GPU implementations are run to collect performance timing data. The timing data is used to select which processing unit (PU) to use during work-stealing. Work stealing is guided to favor gathering interacting objects on the same PU. An ordered list of tasks (partition) is assigned an affinity list of PUs. A PU can steal work if it is in the partition's affinity list. Although limited to the time integration step, the use of mixed GPU and CPU implementations is interesting since it assists in selecting the PU. Similar to [18], the work stealing is affected by changes in collisions due to the increasing number of steals required for load balancing. A new task graph must be recomputed with the addition/removal of object collisions. Work stealing overhead can also increase when expensive CPU–GPU–CPU transfers are triggered, so an attempt is made to minimize this case.

To date, parallel VE implementations tend to be processor-centric. They often rely on a scheduler to assign work to processors. Static work scheduling can help reduce synchronization overhead. Greater speedups may be possible with dynamic work redistribution, at the expense of increased overhead.

Our implementation provides an object-centric VE representation. In the next section, we describe a framework for VEs. The framework suggests how a VE may be abstracted as a set of objects. These objects are structured to facilitate parallel processing in a uniform manner. We later describe a method to dynamically distribute and process VE work without a scheduler. Load balancing is automatically achieved, while still maintaining low synchronization overhead. Additional parallelism is possible by allowing work from up to two consecutive simulation frames to be processed at the same time.

3 VE framework

A VE is defined by the manner in which its elements are modeled and executed. Physically-based VEs consist of various elements that need to be *modeled* at the geometrical and physical level; see Fig. 1. Geometrical modeling describes the element's shape, topology, and visual appearance. Physical modeling defines the algorithms and properties necessary to model behaviors and interactions between elements.

Once a VE is modeled, it is *executed* at rates adequate for interaction and visualization. During execution, an element may be in one of three states: interaction, update, and rendering. These states occur in sequence. In some elements, one or more states can be trivially defined.

Interaction occurs when an element exerts a change on another element. Examples of element-element interaction include collision, selection, and manipulation. Algorithms required to compute modeled VE interactions are domain-specific. Collision detection algorithms identify when and where elements interact with each other. Once identified, a collision response algorithm determines the appropriate action (e.g., force calculation). The result is applied to the elements for updating.

An element can respond to interactions in different ways. For example, a collision can cause the element to deform. It can cause the element to change its visual appearance (e.g., brighten). Manipulating the element can cause changes in geometry. The VE needs to be *updated* to reflect changes over time, including changes resulting from interactions or device input. Physically-based equations that model elements' behavior must be computed. An elements's transformations and/or geometrical data are updated if modified. Elements representing I/O devices sample new input data to reflect the current device state.

Finally, *rendering* algorithms are required to provide visual, tactile, or other sensory feedback from each element in its current state. For example, an element's geometrical representation and visual properties are used to generate an image for the user.

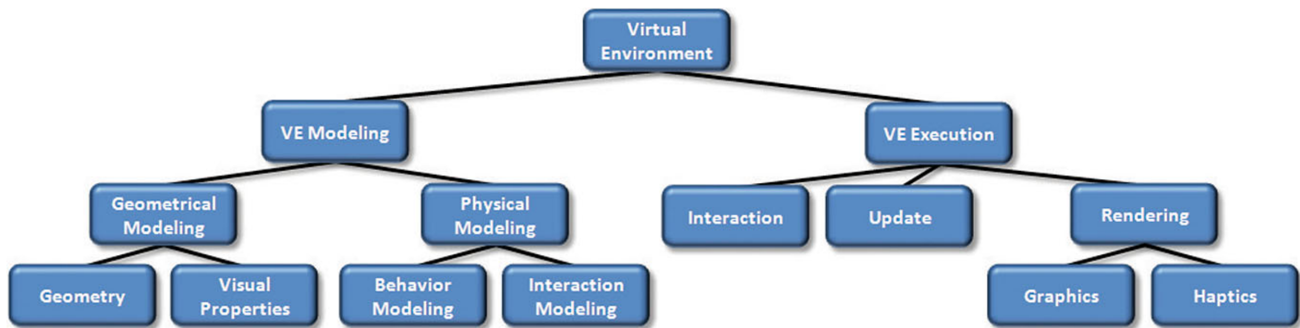

Fig. 1 Typical phases of a physically-based VE. A VE is first modeled, and then executed to simulate and render it

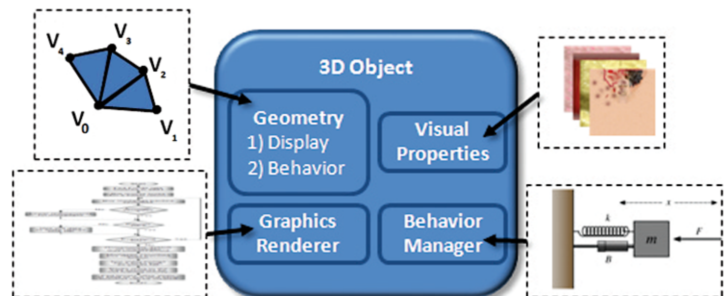

Fig. 2 Modular 3D Object can be used to model different VEs

Given this framework, we propose that a VE can be modeled as a set of elements. Each element is represented by an object. An object is defined by its response to each state, and by its internal data structures. With this organization, a pipeline can be used to implement the execution. The following section describes our approach.

4 Framework implementation

This section describes our implementation of the framework. The structure of a VE object is discussed. The pipeline architecture is presented, and its concept of operation is highlighted.

4.1 VE object structure

A VE object is defined by its data and algorithms for handling each of the three states: interaction, update, and rendering.

3D Object serves as a container that allows VE objects to be constructed by plugging in different combinations of visual properties, geometry, behavior manager (callback), and graphics renderer (callback) modules; see Fig. 2.

Visual properties (e.g., color material and/or texture maps) help model an object's appearance.

The geometry module holds the geometrical representation(s) (e.g., polygons, voxels, etc.) that defines the object's shape and topology.

The behavior manager contains the algorithm(s) responsible for updating an object's state. An object's state is updated based on interaction data (e.g. forces) and the object's physical behavior. If the object represents an I/O device (e.g. haptic device) the new input data is sampled by the behavior manager, and the object is updated based on the input. This device interfacing method allows all objects to be treated identically by the pipeline. Support for haptic devices is detailed in Sect. 6.3.

The graphics renderer defines the graphical rendering algorithm that will be used to display the object.

An interaction pair specifies algorithms that model the interaction between two objects. Interactions are defined by plugging in collision detection and collision response algorithms, Fig. 3. Another module stores collision results, and serves as the input to the collision response algorithm. Only object interactions of interest need to be modeled.

4.2 VE execution pipeline overview

Section 3 outlined the generalized steps required to execute a VE. Since VE task execution is order dependent (e.g. collision response after collision detection), the tasks can be treated as a pipeline (see Fig. 4). Object interactions are identified by Collision Detection, and their results applied to objects in Collision Response. Objects are then updated based on interactions and their defined behavior in Object Update. Object Render provides visual feedback from the objects.

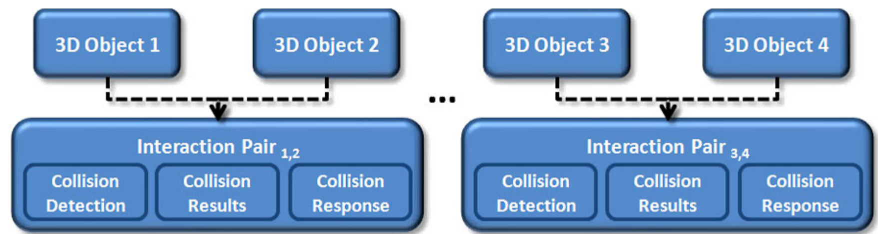

Fig. 3 Pair-wise interactions modeled for parallel processing

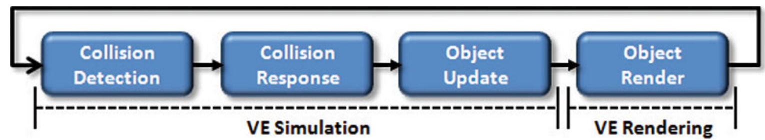

Fig. 4 VE execution tasks as a pipeline

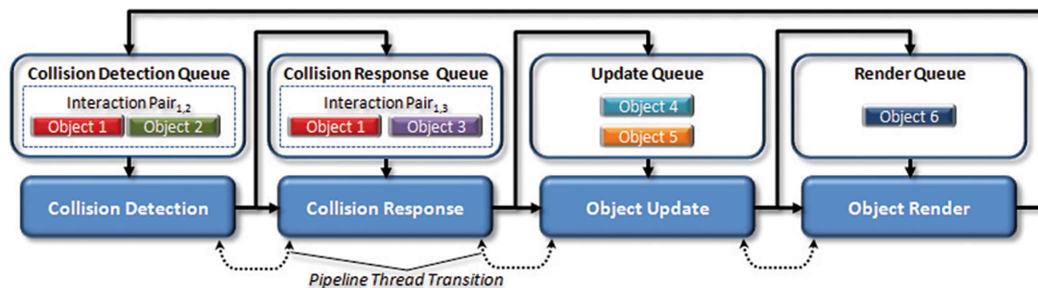

Fig. 5 Simplified parallelization of the VE execution pipeline

Figure 5 provides a simplified parallelization of the pipeline. Parallelization is achieved through multiple threads. Each thread corresponds to a CPU core dedicated to the VE. The pipeline threads are generalized and can process any pipeline stage. The generalization is made possible using a callback mechanism (Sect. 5.2) to trigger application-specific behavior.

The 3D Objects and interaction pairs used to define a VE are directly translated into work units for the pipeline. An interaction pair serves as the input into the Collision Detection stage. Results of collision tests (in the interaction pair) are then passed into the Collision Response stage for processing. Collision Response computes interaction data that is passed to the 3D Objects in the interaction pair for updating. The point between the Collision Response and Object Update stages may require synchronization and is covered in Sect. 5.4. Once an object is updated, it can be rendered.

Work units are passed between stages using queues that are thread-safe. The queues handle synchronization when adding and removing elements. Critical sections are used to synchronize queue access since they are relatively light weight locks [31]. To reduce synchronization, a carry-over strategy is utilized when possible. A carry-over occurs when a thread bypasses a queue and directly carries a work unit to the next stage. Bypassing queues allows work to be moved down the pipeline without any synchronization. If a stage produces more than one work unit, one unit is carried over and the additional units are placed on a queue. For example, both objects in an interaction pair may be ready to be updated after collision response. One object would be carried over, while the other is placed in the object update queue.

A migration algorithm controls how threads transition between stages. The algorithm enables threads to (1) process work from its current stage, (2) follow work down the pipeline, and (3) move up/down the pipeline in search of work. Section 5.3 describes the migration algorithm in more detail. This strategy is simple, and has minimal synchronization overhead. Threads converge on pipeline stages with queued work. Dynamic load distribution is automatic. As demonstrated in Sect. 8, our approach is efficient and scales well across varying workloads, number of objects, and CPU threads.

5 VE execution pipeline

5.1 VE pipeline

Section 4 provided a conceptual description of the pipeline and thread migration strategy. This section discusses optimizations that further improve performance. The resulting pipeline is given in Fig. 6.

The Collision Detection and Collision Response queues can grow rapidly with the number of objects. Moreover, a positive collision always increases the Collision Response

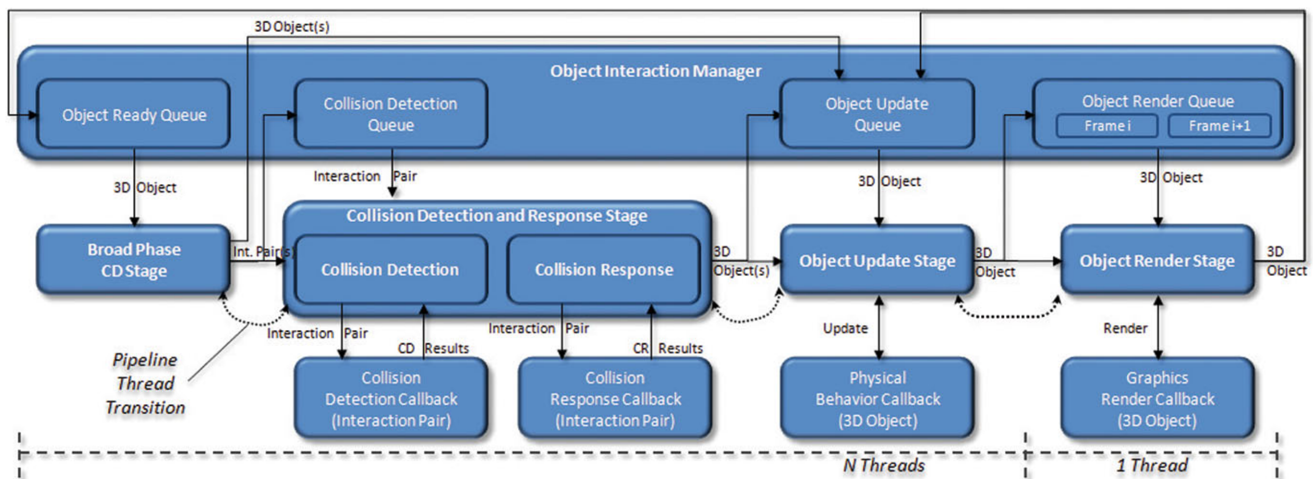

Fig. 6 The dual-frame VE execution pipeline

queue. This results in significant synchronization overhead as threads tend to cluster around the first two stages. As an optimization, collision detection and response were combined into a single interaction stage. To prune non colliding objects, a Broad Phase Collision Detection stage was introduced. The full implementation in Fig. 6 consists of four stages.

An Object Ready queue holds the set of objects that are ready to be processed by the pipeline. The Broad Phase Collision Detection stage quickly eliminates non-colliding objects using bounding volume tests. Only interaction pairs for the potentially colliding set (PCS) of objects are placed into the Collision Detection queue for exact collision testing. To further optimize work unit flow within the pipeline, 3D objects that are eliminated from the PCS are placed directly into the Object Update stage from the Broad Phase Collision Detection stage.

The next object interaction stage performs collision detection and response between objects specified in an interaction pair. Collision response is only triggered if a collision occurs. The semantics of a collision is interaction dependent. For example, a collision can be two objects intersecting or simply a condition that is set to trigger a response.

Objects specified in the interaction pair are passed to the Object Update stage if they are ready to be updated. Section 5.4 describes a synchronization constraint that determines if objects can be updated.

Once updated, an object is passed to the Object Render stage for visual rendering. Lastly, the object is placed back in the Object Ready queue to process for the next frame. Objects with no interactions can bypass the first two stages so they are directly inserted into Object Update instead.

In the current design, all pipeline stages can run in parallel. However, we found that the NVidia GPUs available in our lab (GeForce 8800, 9800 GX2, GTX 480, and Quadro 5000) do not handle multithreaded OpenGL rendering well on a Windows XP or Windows 7 64-bit platform. The best performance results when one thread executes the Object Render stage. Therefore, any number of threads can execute the first three pipeline stages simultaneously. Only one thread currently executes object rendering in parallel to the other three stages. Section 5.3 describes a token system that enforces this constraint. This issue is discussed further in Sect. 9. Once a multithreaded rendering solution is found, the token system can be used to control the number of threads that can render in parallel.

Performance is further improved by allowing up to two consecutive frames to be processed within the pipeline. This strategy can significantly increase the time that all threads are active. Consecutive VE frame processing is described in Sect. 5.5.

5.2 Callback mechanism

The architecture's pipeline employs a callback mechanism to model application specific behavior for each pipeline stage. The callback mechanism is able to implement domain-specific behavior used to model a VE (objects and interactions), while keeping the pipeline problem domain independent.

Each stage triggers a callback corresponding to its task, as shown in Fig. 6. The Collision Detection and Response stage triggers the algorithm modules within an interaction pair. The Object Update stage calls a 3D Object's update callback, which triggers its behavior manager to update the object. Similarly, the Object Render stage calls the 3D Object's render callback to execute its graphics renderer.

Figure 7 provides an example sequence diagram of the callback mechanism for the Object Update stage. The example is kept at a high-level for simplicity. In this example, a mass–spring object needs to be updated. The pipeline

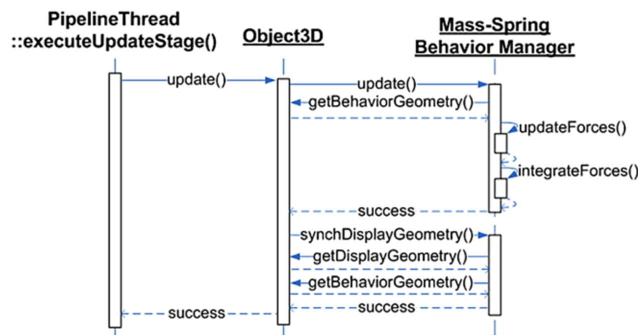

Fig. 7 Example callback sequence for mass–spring update

thread that is processing the Object Update stage triggers the 3D Object's update callback. In turn, the 3D Object triggers its behavior manager to update. The behavior manager acquires the geometry from the 3D Object in order to update the mass–spring forces and update the nodes' position via integration. Finally, the behavior manager synchronizes the 3D Object's triangle mesh geometry to update the new vertex positions and surface normals for visual rendering.

5.3 Thread migration algorithm

Each pipeline thread is designed as a generic worker that can process any stage based on where work resides. Once created, a pipeline thread loops until it is signaled to terminate. A thread enters its current stage and processes its work at each cycle. The current stage is determined by an embedded thread migration algorithm that allows threads to autonomously (1) process work from its current stage, (2) follow work down the pipeline, and (3) move up/down the pipeline in search of work.

The pipeline thread migration algorithm's general concept is outlined in Fig. 8. A thread first attempts to process a work unit that has been either carried over or acquired from the current stage's queue. The work performed at each stage is based on the defined callback. The thread then attempts to carry the current stage's output work unit down to the next stage. If more than one work unit is output, all but the carry over unit is placed in next stage's queue. Another work unit is processed from the current stage's queue if no work unit is output from the current stage. If no work is available for the current stage, a thread looks ahead at the next stage's queue and moves down if work is available. Otherwise, the thread moves up one stage in search of work.

As described in Sect. 5.1, the Object Render stage is a special case. A token system is used to control which thread can enter the rendering stage. The window's rendering context is assigned to the thread with the token. To avoid unnecessary graphics context switches (and thus overhead), only one token is provided and is permanently assigned to the

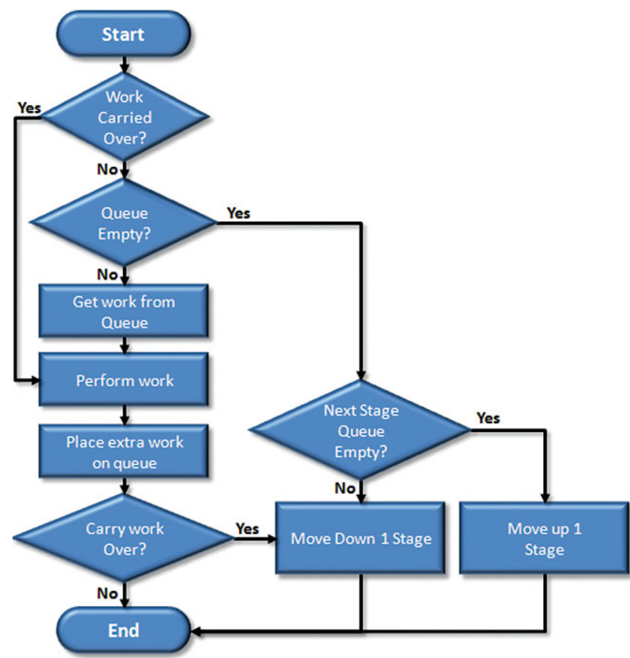

Fig. 8 Generalized pipeline thread migration algorithm

first created thread. If an update thread cannot enter the Object Render stage, it places the object in the Object Render queue and continues.

5.4 VE synchronization

A synchronization point is needed to gather/assemble results when parallel processing. A synchronization constraint is in place to prevent an object from being updated until all its interactions are considered. The constraint ensures that an object's state remains consistent throughout an entire frame. It also allows Object Update to consider all interactions when updating an object. The update constraint is enforced after the Collision Response stage, where it uses a thread-safe atomic increment operation [31] to track when all of an object's modeled interactions are processed.

A second Object Render stage constraint is used to determine when a rendering frame can be swapped. The frame is swapped only after all objects are rendered. This constraint is common for rendering, but also plays a role in allowing the pipeline to process two VE frames simultaneously, as described in Sect. 5.5.

5.5 Consecutive frame processing

The CPU utilization diminishes if threads are in a wait state. Therefore, the pipeline can handle two consecutive frames at a time. Threads can start processing the VE interaction and update workload for the next frame without waiting for the

current frame to complete. The first three stages can be processed one frame ahead. Rendering is not performed immediately for the next frame since the frame buffer has not been swapped. A secondary Object Rendering queue is added to the pipeline for the next frame's objects. Each object and interaction pair has an identifier that indicates which frame it currently belongs to.

It is possible for objects to wait in next frame's queue if rendering is a bottleneck. However, most physically based VEs are typically complex enough that rendering is not the bottleneck. Objects waiting in the secondary rendering queue are processed once the rendering frame is swapped.

5.6 Thread synchronization considerations

Since multiple threads can process interactions for objects, a method is required to consolidate collision response results to update objects. Collision Response does not directly modify an object since an object's state must be consistent through an entire frame. Instead, collision response writes interaction data to temp buffers and passes only pointers to the objects. Interaction data is accumulated by Object Update. The object update constraint described in Sect. 5.4 ensures all interactions are considered prior to updating an object. The use of the temp buffers for interaction data reduces thread synchronization by allowing the Collision Detection and Response stages to work on objects in read-only mode, while Object Update and Object Render stages have exclusive access to an object for modifications.

Overall thread synchronization requirements are reduced by forcing most synchronization to occur between adjacent pipeline stages. Critical sections provide a light weight locking mechanism [31] to make queue access thread-safe. Each queue is implemented as a circular array of pointers, so lock times are very fast. Only pointer values and head/tail indices need to be updated within the critical section. Synchronization overhead is reduced by allowing threads to carry-over work units to subsequent pipeline stages without locks.

The Collision Detection and Response queue can possibly grow large in VEs with a large number of objects. Potential synchronization overhead was eliminated by not having a Collision Response queue, and allowing the same thread to perform both tasks. The size of the Collision Detection queue is also minimized by having the Broad Phase Collision Detection stage identify potentially interacting object pairs.

Hierarchical scene graphs establish dependencies between nodes and make it difficult to parallelize VE processing. A flat tree structure is used to maintain the VE. Every 3D Object maintains its relevant properties and data (e.g., transformations, material properties, etc.) to make them independent of other objects, and easily distributable across different threads. Objects maintain a bounding box, so techniques such as view frustum culling are still possible.

Memory allocation from the heap is very expensive and introduces system-level synchronization. Efforts are made to minimize memory allocations through preallocated and reused memory. The circular arrays for the queues are sized based on the VE prior to execution. For example, the object-based queues are initialized according to the number of objects. Additionally, some memory used by the temp buffers for interaction data is not released back to the system until execution is terminated. The memory (e.g., dynamic arrays) is reused in subsequent interactions and expanded only if necessary. This is a trade-off between memory usage and VE performance.

6 Application development

6.1 Application module overview

A modular design has been created for developing applications with the architecture. An overview of the main application development modules is given in Fig. 9.

A VE Manager is the top-level container for several main application modules. The Object Manager maintains the 3D Objects used to model a VE. The Object Interaction Manager maintains the complete set of interaction pairs and the pipeline queue data structures for VE processing. During application initialization, the VE Manager creates and destroys the pipeline threads. A thread pooling design is used. An application can specify the maximum number of threads for the pool. The VE Manager can optionally detect the CPU core count and set the thread pool size accordingly. Threads are dynamically created and/or destroyed to meet the maximum thread count requirement.

The Window has modules required for graphics rendering and display. The Device I/O Manager maintains different types of device interfaces. Haptic device support is described in Sect. 6.3.

6.2 Message passing interface

A message passing interface has also been established to handle events in the VE. Messages are generated and sent to an inbox provided by each 3D Object. Messages are derived from a base class interface. Each message implements a *processMessage* callback that defines a message's purpose. For example, the callback can set flags, or modify objects' properties due to events. Messages within an object's inbox are processed in the Object Update stage immediately before triggering the object's *update* callback.

6.3 Haptic device support

Device interface modules enable I/O devices to be interfaced to a VE. Some device interfaces utilize a light-weight thread

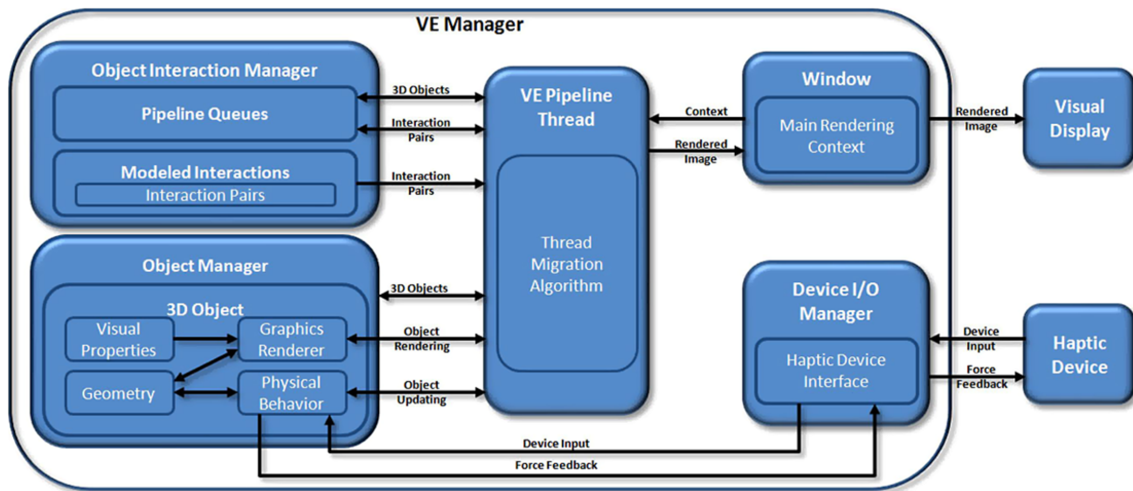

Fig. 9 Overview of the application development modules

for device communication, and provide thread-safe methods to read/write device data. A 3D Object's behavior manager can be linked to a device interface in order to represent a device in the VE. Network-based devices have also been tested using this interfacing method. Device interfaces are maintained by a Device I/O manager.

Haptic devices are a special case in which force feedback needs to be computed from a VE and sent to a haptic device. A haptics rendering loop typically needs to run at a 1 kHz update rate in order to maintain haptic stability. To support parallel haptic interactions with complex VEs, collision forces are computed at pipeline update rates outside the main haptic rendering loop (residing in the device interface). Significant force changes can result between new updates, leading to mechanical instabilities if directly inserted into the haptic loop. Thus, a multirate haptic interface [9] introduces new forces into the haptic loop and interpolate forces (at haptic rates) between new updates. The multi-rate haptic interfacing is based on (1). The new collision force ($f(P_N)$) and the local force gradient ($\nabla f(P_N)$) are computed by the pipeline based on the current haptic device position (P_N). P_n is the device's position at haptic update rates. A new force is interpolated within the haptic loop ($F(P_n)$) using $\nabla f(P_N)$. The interpolated force is a first-order approximation of the change in force based on the deviation of the haptic device's position from P_N.

$$F(P_n) = f(P_N) + \nabla f(P_N)(P_n - P_N) \qquad (1)$$

7 Performance experiments

Experiments were conducted to evaluate the performance of the architecture. Scalability and load balancing is first explored. Comparative studies with published parallel frameworks (Sect. 7.2.3) are then conducted. A serial version of each test application was developed and used as a basis.

7.1 Test applications

The two test applications used are shown in Fig. 10. The first allows varying the VE complexity to explore performance under various conditions. The second is a practical application of the architecture. A head-trauma simulator has been developed, and key aspects of the simulation are benchmarked.

7.1.1 Deformable mass–spring particles

The VE shown in Fig. 10 (left) is a 6-wall room containing N deformable mass–spring based particles (42 nodes and 162 springs each). The mass–spring system's step size for each particle is randomly set to run between 1–10× per frame to vary its complexity. Gravity is turned off in most cases so particles can bounce around the room and deform as they collide with each other and the walls. The worst-case scenario, where all objects in the VE can interact with each other, is tested. This results in $N(N+1)/2 + 6N$ possible interactions.

7.1.2 Craniotomy surgical simulator

A Virtual Reality-based training simulator is being developed to practice the skills required to perform a craniotomy. Several key steps from a craniotomy have been addressed, including scalp cutting and retraction, and simulation of various bone cutting tools (e.g., bone drills and perforators), that are typically used in clinical practice. The middle and right images in Fig. 10 show scalp scraping for retraction, and burr hole cutting with a perforator tool. 3D models have

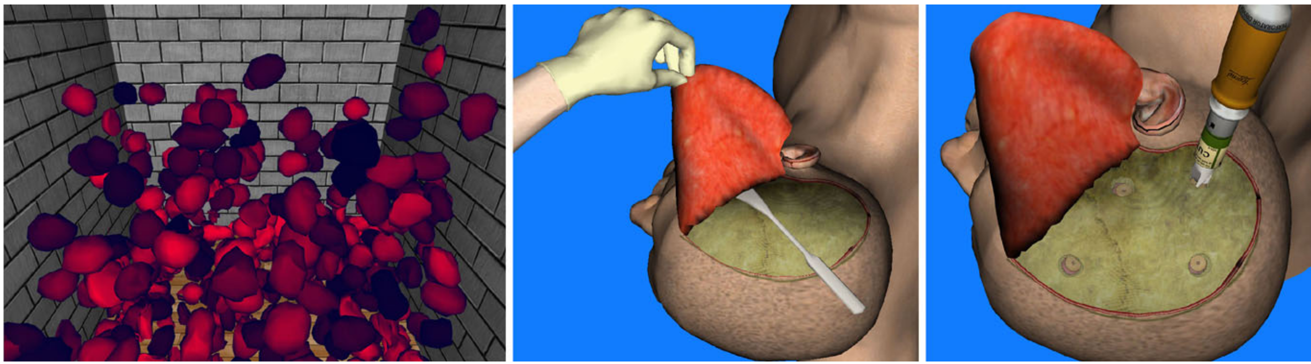

Fig. 10 Example applications developed on the pipeline VE architecture. The *left figure* shows a VE consisting of deformable mass–spring based particles that interact with each other and the walls. The *middle* and *right figures* show the skin-flap cutting/retracting and burr hole cutting steps of a craniotomy simulator

been created from real surgical tools and are controlled by haptic devices during their use. Voxel-based algorithms are used for modeling the tool-bone interactions and texture-based volume rendering displays the virtual bone material [1]. The VE consists of 30 objects (2 tools and 28 anatomical structures), of which 27 are real-time deformable mass–spring based objects.

7.2 Experiments

The described pipeline architecture and test applications are implemented in C++ with OpenGL for graphics rendering. The test system used for the experiments is a 6-core Intel i7-980x (3.33 GHz with 12 MB Cache) processor, 6 GB RAM, NVidia Quadro 5000 (2560 MB) graphics card and Windows 7 64-bit OS.

The number of threads used to execute the pipeline is varied and the time required to process a specific number of frames is measured. The measured data is then used to compute the average throughput in frames per second (FPS), speed-up over the single-thread case (S_p), and speed-up over a serial implementation developed external to the architecture (S_s). S_p and S_s are computed using (2) and (3). T_1 and T_p are execution times for 1 and p threads, respectively. T_s is the execution time for the serial implementation. Thread efficiency (E_p) is computed using (4). The experimental results are also compared with the ideal theoretical results.

$$S_p = T_1/T_p \quad (2)$$

$$S_s = T_s/T_p \quad (3)$$

$$E_p = S_p/p \quad (4)$$

7.2.1 *Scalability*

The particle VE was used to evaluate the scalability of the pipeline architecture. The number of objects, N, in the VE and the number of pipeline threads, p, were varied. Results are taken for cases when $100 \le N \le 1{,}000$ in increments of 100, and $1 \le p \le 6$. The time taken to process 800 frames is measured and used to compute the average throughput, S_p, and E_p.

7.2.2 *Load balancing*

The particle VE was also used to analyze the pipeline's load balancing. The time each thread works is measured over 800 frames. The measurement reflects the time to acquire work (e.g., queue synchronization) and actual processing time.

The S_p for dynamic load balancing methods (e.g., work stealing) can sometimes be affected by the gain or loss of collisions [18, 19]. A per-frame S_p analysis is performed to evaluate how the pipeline responds to changes in the number of collisions. The collision count and total frame processing time is captured for 600 frames. The particle VE with 300 objects is used for the experiment since it allows sufficient variation in the number of collisions over the sampled frames. Gravity is turned on so particles can drop to the floor and pile up to increase collisions.

7.2.3 *Comparative study*

VE processing pipelines were implemented using the Intel TBB [23], Intel Cilk Plus [24], and OpenMP [35] parallel frameworks. All implementations drive the same architecture modules (e.g., 3D Objects, interaction pairs, and algorithms, etc.) to compare parallel performance. Each framework provides constructs to specify parallel sections of code and the number of worker threads. The implementations rely on the frameworks' thread scheduling algorithms without introducing additional overhead. Since dependencies exist between pipeline stages (e.g., Sect. 5.4), all the work in a stage is processed in parallel prior to moving to the next stage. Similar to our approach, all work except the graphics rendering is multithreaded. Global vector arrays store the 3D Object and interaction pair work units.

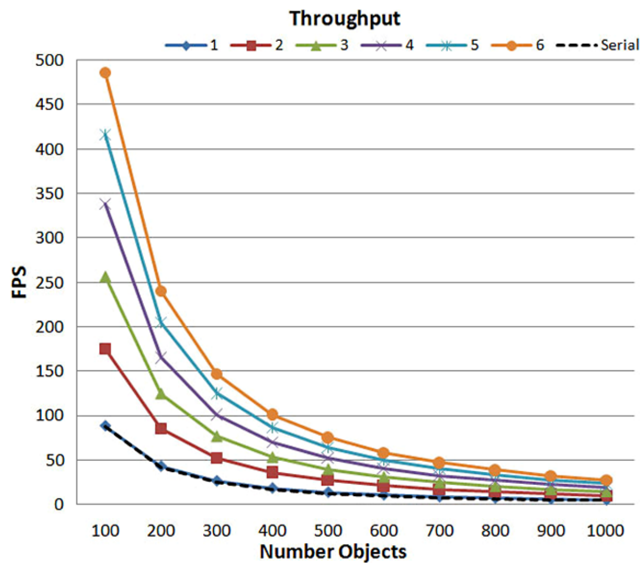

Fig. 11 Particle VE throughput. *Dashed line* is serial results

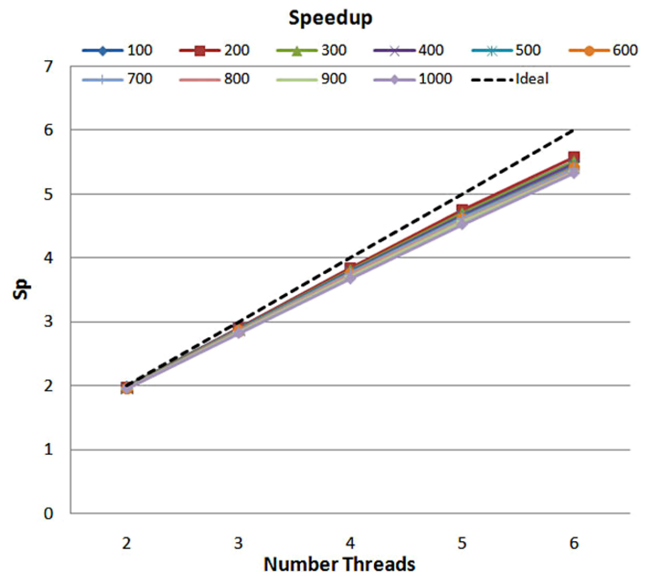

Fig. 12 Particle VE speedup. *Dashed line* is ideal results

Both Intel TBB and Cilk Plus utilize work-stealing strategies for load balancing. The Intel TBB version uses a *parallel_for* to split work into *blocked_ranges*, and assigns each range to the specified number of threads. The *blocked_ranges* are based on *auto_partitioner* so the parallel loop can optimize the subdivision based on work-stealing events. The initial subdivision is proportional to the number of threads. Subranges can be further subdivided for load balancing when a thread becomes idle and work stealing is required.

Intel Cilk Plus, the C++ extension to Cilk [14], provides a *cilk_for* construct to divide work and assign them to threads' queues. Work is distributed based on a specified grain size and the number of threads. The default grain size is used since it provides the best performance. The Cilk Plus scheduler performs work stealing when threads' queues become empty.

The OpenMP implementation uses a *parallel_for* directive to specify work to be computed in parallel. The iterations of a loop are distributed across the threads using static scheduling since it results in the best performance. Static scheduling divides the workload into chunks that are approximately equal in size and assigns each chunk to a thread.

Both test VEs are used for the performance experiments. A serial implementation of each application serves as the baseline for computing S_s. The particle VE is used to compare S_s under light (100 objects) and heavy (1,000 objects) loads. The serial particle VE throughput results are given in Fig. 11 as a dashed line. The skin retraction step of the craniotomy is automated to provide a consistent test case, and the time required to process 1,000 frames is measured. Only 1–5 threads are tested since one thread is reserved for haptics rendering. The average serial throughput for the craniotomy VE is 99 FPS.

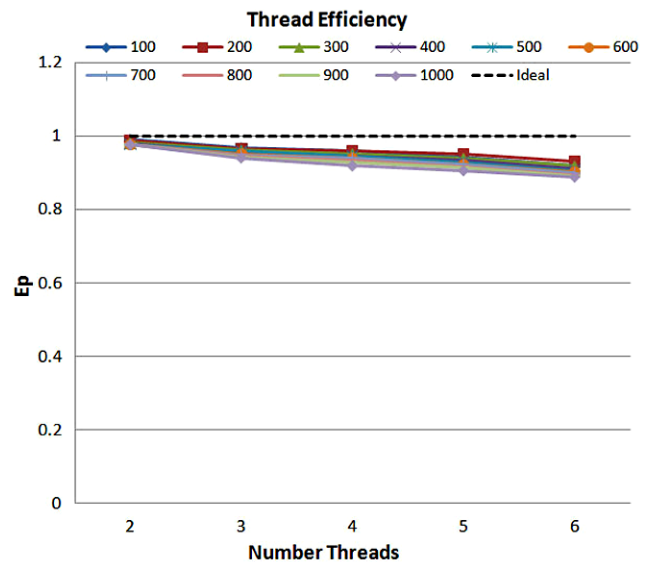

Fig. 13 Particle VE thread efficiency. Ideal results shown as *dashed line*

8 Experimental results

8.1 Scalability results

Scalability results are provided in Figs. 11, 12, and 13. The ideal theoretical values are graphed as dashed lines in Figs. 12 and 13 for reference. The experiments show that the architecture scales efficiently with the number of CPU cores. Near-linear S_p is achieved even over a wide range of objects. Each additional thread increases the VE throughput. On average, S_p only dropped by 0.2–0.6% with each additional 100 objects. These results suggest a low overhead for the architecture. A slight drop in the E_p (1.5–2.3% on av-

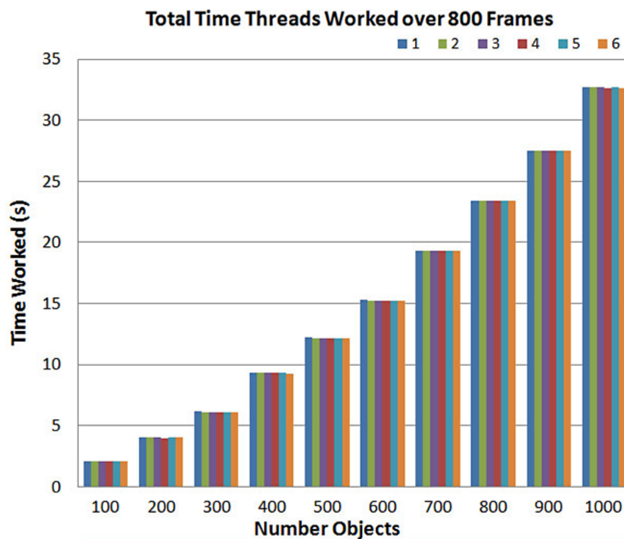

Fig. 14 Total time threads worked shows very good load balancing

Table 1 Seconds thread worked per stage over 800 frames (500 objects)

Thread	BP/CD	CD/CR	Update	Render	Total
1	2.0738	1.5049	6.2195	2.4065	12.2047
2	3.6991	1.6294	6.8257	0.0	12.1542
3	3.6730	1.6456	6.8388	0.0	12.1573
4	3.6502	1.6637	6.8459	0.0	12.1598
5	3.6016	1.7264	6.8310	0.0	12.1590
6	3.7209	1.5994	6.8414	0.0	12.1616

erage) occurs with each additional thread. However, Fig. 13 shows the E_p remains greater than 89% in all test cases.

8.2 Load Balancing Results

The results in Fig. 14 show that the total time each thread works is very balanced for all cases. Table 1 breaks down the times by pipeline stage for the 500 object case. Timings for threads 2–6 are very similar. Thread 1's time distribution differs since it also performs graphics rendering.

Figure 15 provides per-frame collisions, and S_p for 1–6 threads. The particles are initially close together so a large number of collisions (1,646) occur early. Repulsive forces cause the particles to separate over time dropping the collisions to 160. The number of collisions increases (up to 1,056) as the particles pile up on the floor. The results show no significant effect in S_p as the number of collisions change. The S_p remains fairly level independently of the number of collisions. We did observe a slight variance increase in the per-frame S_p (0.001–0.08 with 2–6 threads) as more system threads were utilized. We attribute some of the effects to the OS scheduling. The synchronization costs are also likely randomly distributed from frame-to-frame. However, variance at the per-frame level is not perceivable to the user of the VE since the frame rates are sufficiently high, as shown in Fig. 11. The 600 frames represent about 4 seconds for the 6 thread case.

8.3 Comparative study results

Results for the particle VE under light and heavy loads are given in Figs. 16 and 17, respectively. All pipelines show

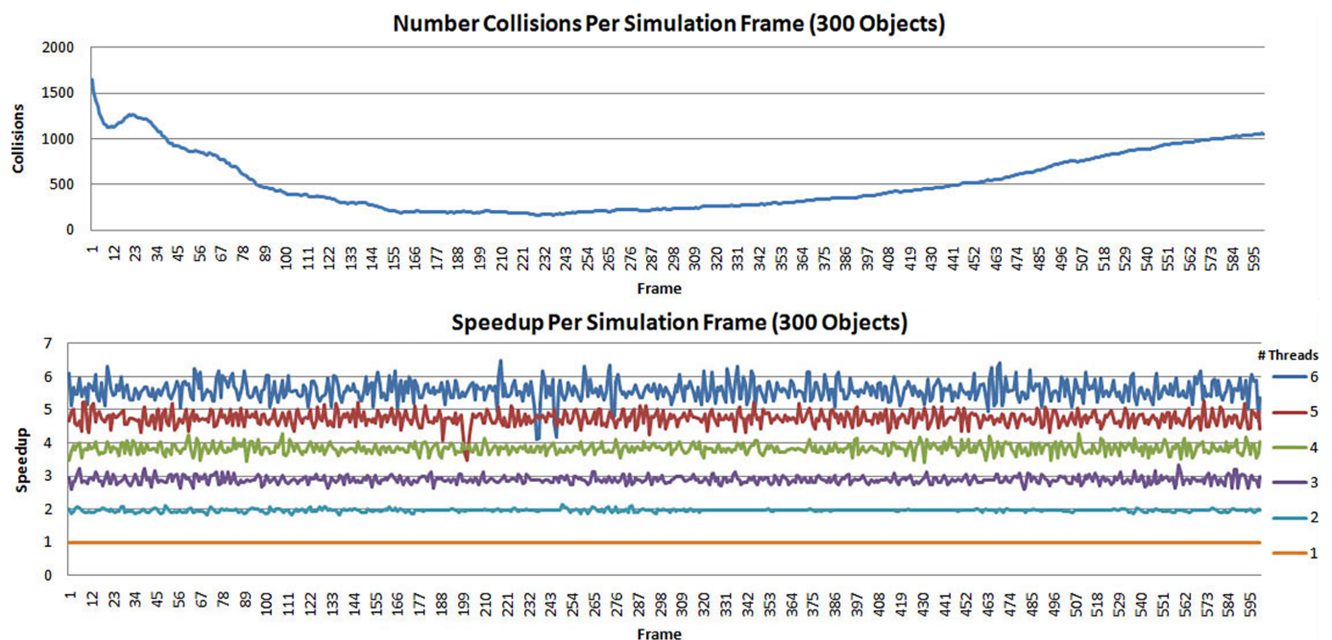

Fig. 15 The per-frame number of collisions (*top*) and speedups (*bottom*)

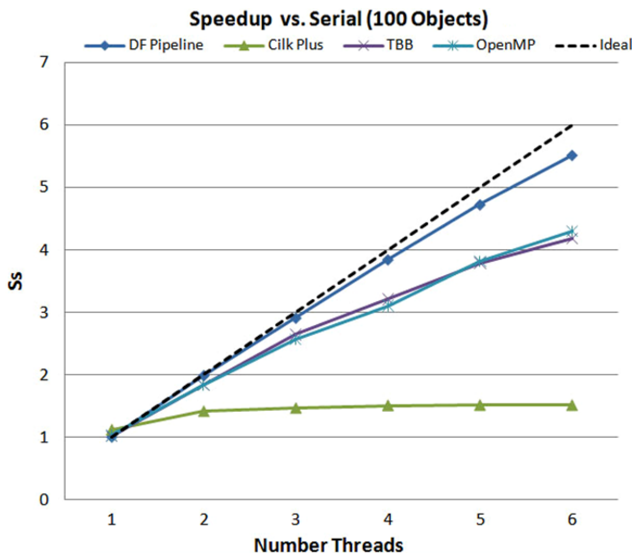

Fig. 16 Speedup over serial version under light load. Dual-frame pipeline architecture denoted as "DF Pipeline"

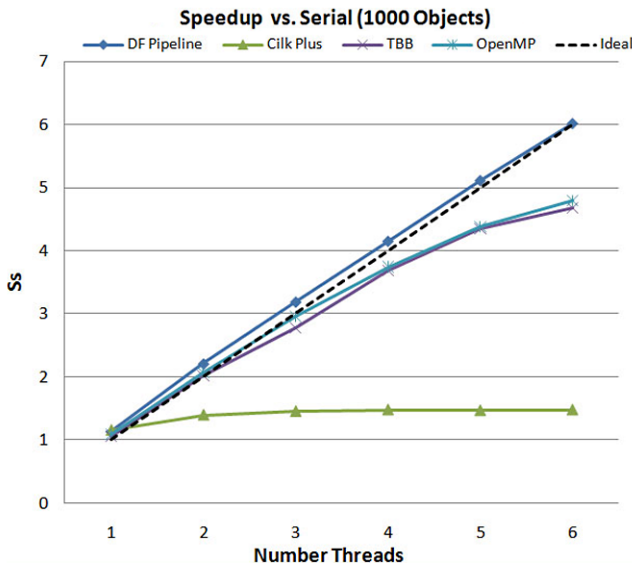

Fig. 17 Speedup over serial version under heavy load. Dual-frame pipeline architecture denoted as "DF Pipeline"

little to no overhead when only 1 thread is used. The dual-frame pipeline architecture shows the best S_s in both light and heavy loads. The OpenMP and Intel TBB results are very similar to each other since work is initially evenly divided. TBB only employs work-stealing when required. The S_s for the Intel Cilk Plus version remains fairly flat after the 2 threads case. The pipeline work is of a fine-medium grain nature, which can be sub-optimal for Cilk [2]. Several attempts at specifying a grain size did not result in better performance.

Under light loads, the architecture provides near-linear S_s. As expected, performance improved in most cases under the

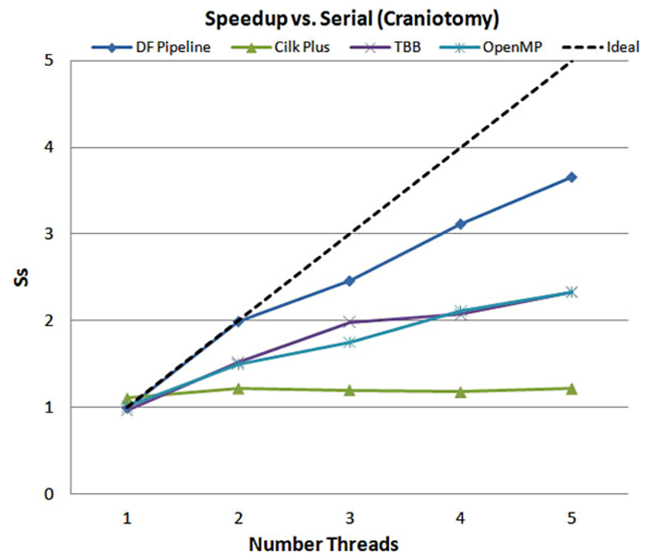

Fig. 18 Speedup over serial version. Five pipeline and one haptics thread tested. Average serial pipeline throughput is 99 FPS

heavy load condition since more work can be performed in parallel. No improvement was seen with Cilk Plus. The pipeline architecture exhibited super-linear S_s under heavy loads. This is likely due the large amount of cache provided by the CPU. In both load cases, the pipeline architecture is approximately 8–28% faster than OpenMP and TBB and 40–300% faster than Cilk Plus when 2–6 threads are used.

The S_s results for the craniotomy simulator are given in Fig. 18. Similar to the particle VE, the OpenMP, and Intel TBB performance are comparable. Cilk Plus only achieves minimal speedups. The pipeline architecture provides the best S_s. The architecture is roughly 1.3–1.6× faster than OpenMP and TBB, and 1.6–3.0× faster than Cilk Plus for the 2–5 thread cases.

9 Discussion

Section 2 reviewed other approaches taken to parallelize VEs. Many methods focus on generating lists of tasks, and assigning them to processors for parallel execution. Most methods rely on static work scheduling. A centralized work scheduler is common. Dynamic load balancing methods for VEs are described, but they require additional work stealing or tracking of run-time heuristics. The speedups from these methods are also influenced by the addition and removal of collisions.

We propose a new way of parallelizing VEs. VE data and algorithms are encapsulated into objects, and a pipeline of tasks is generated. Threads move work down the pipeline without worrying about task assignment order. The execution order is enforced by the pipeline, and the described constraints ensure VE consistency. The migration algorithm en-

ables threads to dynamically acquire work without a scheduler. No supplemental work stealing or heuristics is required. Performance speedups are not influenced by collisions since work is not explicitly assigned. Threads automatically migrate to pipeline stages where work resides. The pipeline work flow has been optimized. Processing work from up to two consecutive frames helps keep thread utilization high. The introduced carry-over strategy moves work down the pipeline without synchronization. The experimental results show that our method scales well, is efficient, and requires minimal overhead. Our method also provides better speedups than other comparable published methods.

The primary motivation of the architecture relates to the requirements of interactive computer-based VEs. In our case, it is for interactive physically-based surgical simulations. Interactive VEs require detecting interactions (e.g. collision and response), updating, and rendering. The semantics for collision detection is interaction dependent, and can range from actual intersection testing to setting a flag to trigger a response. The architecture also supports VEs without interactions by placing rendered objects directly into the update stage. This allows the architecture to be used for other types of VEs (e.g., with no interactions). For example, scripted animations could be played back using the Object Update and Object Render stages.

The pipeline parallelizes VEs independently of algorithms. Both parallel and sequential algorithms are supported. We are currently testing parallel algorithms for object updating, and early results are promising. Support for sequential algorithms is beneficial in several ways. Converting existing sequential algorithms into parallel forms can be difficult and almost always requires redesigning and rewrites. Additionally, the modular nature of the architecture supports integration of different researchers' work, a desirable feature in the surgical simulation field [6, 8, 29], without considerable modifications to existing algorithms. The drawback to using sequential algorithms is that performance may not be optimal if only a few and very large/complex objects model a VE. In this case, it may be beneficial to partition large/complex objects into smaller objects (and model interactions between the partitions) to improve workload balancing and the VE's performance. An approach similar to this has been previously suggested [18, 39].

As described in Sect. 5.1, all pipeline stages except Object Render are multithreaded. Rendering does occur in parallel to the work in the other stages, however, the token system constrains the Object Render stage to one thread. Using NVidia's PerfKit [33], we found that the GPU can be over utilized and considerable time is spent within the graphics driver. The graphics drivers do not seem to be optimized for multithreaded OpenGL rendering. A few methods, including rendering to off-screen buffers with image compositing and multiwindow rendering, have been tried. We will continue to investigate this issue in an attempt to remove the single-threaded rendering constraint.

Our initial focus has been on parallelizing CPU-based algorithms. We are currently testing the use of GPU-based algorithms within the architecture. Model updates using CUDA [32] are running successfully without any changes to the pipeline. Other algorithms, such as GPU-based collision detection [16], need to render to an off-screen buffer [34]. This requires a small enhancement to the pipeline as a side-effect of the multithreaded rendering issue just described. The Object Render stage can serve as a GPU resource by adding an auxiliary queue so the rendering thread can process GPU-based work units. This method has been tested within the craniotomy simulator in Fig. 10. The GPU-based collisions are tested between pairs of objects, so no additional changes to the pipeline are required.

As we expand to use different processing units (e.g. CPU and GPU), it would be interesting to investigate methods for optimizing processing unit selection. Currently it is up to the developer. Hermann et al. is the only work that we are aware of that has applied this technique to VEs [19]. As described in Sect. 2, work-stealing is guided to execute the time integration step for physics simulations. We hope to apply the optimization to the entire pipeline. Other recent generalized parallel works have described methods that support processing unit selection [7, 26, 27, 43].

10 Conclusion

We presented an architecture for developing parallel VEs for multicore processor systems. A common framework for interactive VEs is described. The framework encompasses a wide-range of applications, such as medical simulations, that require computing VE interactions and updating. The framework helped establish a uniform object-based representation of VEs, and a multistage parallel pipeline. The described pipeline execution model provides a decentralized and dynamic work scheduling method. Work from up to two consecutive frames can be executed in parallel. Workload balancing is automatically achieved with relatively low overheads. Results show that the architecture is efficient and scales well with the number of CPU cores.

Applications written with the pipeline VE architecture can see greater performance boosts as the number of CPU cores increase. This makes it possible to write applications for today's hardware and benefit from future systems without modifications.

Acknowledgements We would like to thank Jennifer Sieck and Valerie Henry for creating the patient models for the craniotomy simulator shown in Fig. 10. We also acknowledge Dr. Rocco Armonda for his valuable input and research materials for the craniotomy simulator. We thank Jamie Cope for developing a custom X3D importer to load VEs into the architecture.

Open Access This article is distributed under the terms of the Creative Commons Attribution Noncommercial License which permits any noncommercial use, distribution, and reproduction in any medium, provided the original author(s) and source are credited.

References

1. Acosta, E., Liu, A.: Real-time volumetric haptic and visual burr hole simulation. In: Proceedings of IEEE Virtual Reality, pp. 247–250 (2007)
2. Aldinucci, M., Torquati, M., Meneghin, M.: FastFlow: efficient parallel streaming applications on multi-core. Technical Report TR-09-12, Universita Di Pisa, Dipartimento di Informatica, Italy (2009)
3. Agus, M., Giachetti, A., Gobbetti, E., Zanetti, G., Zorcolo, A.: A multiprocessor decoupled system for the simulation of temporal bone surgery. Comput. Vis. Sci. **5**(1), 35–43 (2002)
4. Allard, J., Raffin, B.: Distributed physical based simulations for large VR applications. In: Proceedings of IEEE Virtual Reality, pp. 89–96 (2006)
5. Allard, J., Gouranton, V., Lecointre, L., Limet, S., Melin, E., Raffin, B., Robert, S.: FlowVR: a middleware for large scale virtual reality applications. In: Proceedings of Euro-par 2004 (2004)
6. Allard, J., Cotin, S., Faure, F., Bensoussan, P.-J., Poyer, F., Duriez, C., Delingette, H., Grisoni, L.: SOFA—an open source framework for medical simulation. In: Medicine Meets Virtual Reality, pp. 13–18 (2007)
7. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput., Pract. Exp. **23**, 187–198 (2011)
8. Cavusoglu, M.C., Goktekin, T.G., Tendick, F., Sastry, S.S.: GiPSi: an open source/open architecture software development framework for surgical simulation. In: Medicine Meets Virtual Reality, pp. 46–48 (2004)
9. Cavusoglu, M.C., Tendick, F.: Multirate simulation for high fidelity haptic interaction with deformable objects in virtual environments. In: IEEE International Conference on Robotics and Automation, pp. 2458–2465 (2000)
10. Deligiannidis, L., Jacob, R.J.K.: Improving performance of virtual reality applications through parallel processing. J. Supercomput. **33**(3), 155–173 (2005)
11. Eilemann, S., Makhinya, M., Pajarola, R.: Equalizer: a scalable parallel rendering framework. IEEE Trans. Vis. Comput. Graph. **15**(3), 436–452 (2009)
12. Farias, T.S.M.C., Almeida, M.W.S., Teixeira, J.M.X.N., Teichrieb, V., Kelner, J.: A high performance massively parallel approach for real time deformable body physics simulation. In: Proceedings of the 2008 20th International Symposium on Computer Architecture and High Performance Computing, pp. 45–52 (2008)
13. Figueiredo, M., Fernando, T.: An efficient parallel collision detection algorithm for virtual prototype environments. In: Proceedings of the Parallel and Distributed Systems, pp. 249–256 (2004)
14. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, pp. 212–223 (1998)
15. Gautier, T., Besseron, X., Pigeon, L.: Kaapi: a thread scheduling runtime system for data flow computations on cluster of multiprocessors. In: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, pp. 15–23 (2007)
16. Govindaraju, N.K., Lin, M.C., Manocha, D.: Quick-CULLIDE: fast inter- and intra-object collision culling using graphics hardware. In: Virtual Reality, pp. 59–66 (2006)
17. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. Scientific and Engineering Computation Series. MIT Press, Cambridge (1994)
18. Hermann, E., Raffin, B., Faure, F.: Interactive physical simulation on multicore architectures. In: Eurographics Workshop on Parallel and Graphics and Visualization (2009)
19. Hermann, E., Raffin, B., Faure, F., Gautier, T., Allard, J.: Multi-GPU and multi-CPU parallelization for interactive physics simulations. In: Euro-Par Conference on Parallel Processing: Part II (2010)
20. Huang, J., Shareef, N., Crawfis, R., Sadayappan, P., Mueller, K.: A parallel splatting algorithm with occlusion culling. In: Proceedings of Eurographics Workshop Parallel Graphics and Visualization (2000)
21. Huagen, W., Zhaowei, F., Shuming, G., Qunsheng, P.: A parallel collision detection algorithm based on hybrid bounding volume hierarchy. In: CAD/Graphics 2001 (2001)
22. Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P.D., Klosowski, J.T.: Chromium: A stream-processing framework for interactive rendering on clusters. ACM Trans. Graph., **21**(3), 693–702 (2002)
23. Intel Thread Builiding Blocks 3.0. http://threadingbuildingblocks.org. Accessed 28 July 2011
24. Intel Cilk Plus. http://software.intel.com//en-us//articles//intel-cilk-plus//. Accessed 28 July 2011
25. Jerabkova, L., Terboven, C., Sarholz, S., Kuhlen, T., Bischof, C.: Exploiting multicore architectures for physically based simulation of deformable objects in virtual environments. In: Proceedings of Virtuelle und Erweiterte Realität, 4. Workshop der GI-Fachgruppe VR/AR, Weimar, Germany (2007)
26. Kicherer, M., Buchty, R., Karl, W.: Cost-aware function migration in heterogeneous systems. In: Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, pp. 137–145 (2011)
27. Luk, C.-K., Hong, S., Kim, H.: Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 45–55 (2009)
28. Molnar, S., Eyles, J., Poulton, J.: PixelFlow: high-speed rendering using image composition. In: Proceedings of ACM SIGGRAPH, pp. 231–240 (1992)
29. Montgomery, K., Bruyns, C., Brown, J., Sorkin, S., Mazzella, F., Thonier, G., Tellier, A., Lerman, B., Menon, A.: Spring: a general framework for collaborative, real-time surgical simulation. In: Proceedings of Medicine Meets Virtual Reality, pp. 23–26 (2002)
30. Muraki, S., Ogata, M., Ma, K., Koshizuka, K., Kajihara, K., Liu, X., Nagano, Y., Shimokawa, K.: Next-generation visual supercomputing using PC clusters with volume graphics hardware devices. In: Proceedings of Supercomputing, pp. 51–58 (2001)
31. MSDN Library. http://msdn.microsoft.com/library/. Accessed 8 Sept 2010
32. NVIDIA CUDA C Programming Guide. Available at http://developer.nvidia.com/object/gpucomputing.html. Accessed 8 Sept 2010
33. NVIDIA PerKit. Available at http://developer.nvidia.com/object/nvperfkit_home.html. Accessed 8 Sept 2010
34. OpenGL Extension Reference. http://www.opengl.org/registry/. Accessed 8 Sept 2010
35. OpenMP API. http://openmp.org. Accessed 8 Sept 2010
36. OpenSceneGraph. http://www.openscenegraph.org/. Accessed 8 Sept 2010
37. OpenSceneGraph. http://www.openscenegraph.org/projects/osg/wiki/OpenSceneGraph. Accessed 8 Sept 2010
38. Schulze, J.P., Lang, U.: The parallelization of the perspective shear-warp volume rendering algorithm. In: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization, pp. 61–69 (2002)

39. Thomaszewski, B., Pabst, S., Blochinger, W.: Parallel techniques for physically based simulation on multi-core processor architectures. Comput. Graph., **32**(1), 25–40 (2008)
40. Vo, H.T., Osmari, D.K., Summa, B., Comba, J.L.D., Pascucci, V., Silva, C.T.: Streaming-enabled parallel dataflow architecture for multi-core systems. Computer Graphics Forum **29**(3), 1073–1083 (2010)
41. Voß, G., Behr, J., Reiners, D., Roth, M.: A multi-thread safe foundation for scene graphs and its extension to clusters. In: Eurographics Workshop on Parallel Graphics and Visualization, pp. 33–37 (2002)
42. Visualization Toolkit. http://www.vtk.org. Accessed 19 Aug 2011
43. Wernsing, J.R., Stitt, G.: Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. ACM SIGPLAN Not. **45**(4), 115–124 (2010)
44. Wieland, F., Carnes, D., Schultz, G.: Using quad trees for parallelizing conflict detection in a sequential simulation. In: Proceedings of 15th Workshop on Parallel and Distributed Simulation, pp. 117–123 (2001)
45. Wittenbrink, C.M., Amin, M.B., Grama, A.: Survey of parallel volume rendering algorithms. In: Proceedings of Parallel and Distributed Processing Techniques and Applications, pp. 1329–1336 (1998)
46. Zhao, W., Tan, R.-P., Li, W.-H.: Parallel collision detection algorithm based on mixed BVH and OpenMP. In: Proceedings of 7th International Conference on System Simulation and Scientific Computing, Asia Simulation Conference, pp. 786–792 (2008)

Eric Acosta received his Ph.D. in Computer Science from Texas Tech University in 2006. He is a Computer Research Scientist at the National Capital Area Medical Simulation Center. Dr. Acosta is the lead researcher and developer of several Virtual Reality-based haptic surgical simulators. His recent research has focused on developing the center's head trauma and cricothyroidotomy simulators. Dr. Acosta is also a primary architect and the developer of a pipelined virtual environment architecture for multi-core CPU systems. The architecture is a unified platform for developing the center's next generation simulators.

Alan Liu received his Ph.D. in Computer Science from the University of North Carolina at Chapel Hill in 1998. Dr. Liu is the Director of the Virtual Medical Environments Laboratory at the National Capital Area Medical Simulation Center. He is involved in defining the center's strategic research goals, and directing the development of the center's computer-based medical training systems. Dr. Liu is the principal developer and architect of several simulators. His current research focus includes the development of the center's WAVE, a 1000 sq ft total immersion virtual environment for mass casualty training, with applications in military medical readiness, and homeland defense.