Simulating Multi-Million-Robot Ensembles

Michael P. Ashley-Rollman
Carnegie Mellon University
Pittsburgh, PA 15213

Padmanabhan Pillai
Intel Labs Pittsburgh
Pittsburgh, PA 15213

Michelle L. Goodstein
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract—Various research efforts have focused on scaling modular robotic systems up to millions of cooperating devices. However, such efforts have been hampered by the lack of prototype hardware in such quantities and the unavailability of accurate and highly scalable simulations. This paper describes a simulation framework for such systems, which can model the execution of distributed software and the physical interaction between modules. We develop a scalable, multithreaded version of an off-the-shelf physics engine, and create a software execution engine that can efficiently harness hundreds of cores in a cluster of commodity machines. Our approach is shown to run 108x faster than a previous scalable simulator, and permit simulations with over 20 million modules.

I. INTRODUCTION

An effective simulator is an invaluable tool in development of control software for all kinds of robots. It is particularly crucial for large-scale modular robotics systems, like Claytronics [1]. The Claytronics project envisions millions of tiny, sub-millimeter robotic modules, called Claytronic atoms or *catoms*, that work together to form physical structures and shapes, essentially creating a form of programmable matter (Figure 1). Software development for Claytronics suffers from two critical issues. First, the sub-millimeter hardware does not yet exist. Furthermore, building this hardware will be very expensive as it requires access to semiconductor fabrication processes. It is, therefore, unlikely to be developed until software has been demonstrated to make effective use of the hardware in simulation. Secondly, although physically larger prototype robotic modules have been built, they do not exist in sufficient quantities to thoroughly test and execute the very large-scale distributed algorithms needed to control a real Claytronics system.

A good simulation framework can address both of these issues by providing a means to develop software before any actual hardware is available and perform large-scale tests and demonstrations of distributed algorithms before hardware is mass-produced. Furthermore, a good simulation framework can substantially improve the development process. For example it can allow the programmer to inspect a synchronous snapshot of distributed state, inject test messages or faults, and perform distributed debugging operations not possible on the raw hardware. Simulations can also speed up the development process by allowing a broad range of parameterized experiments to be scripted and run to quickly sample a design space.

Developing a good simulator is not an easy task. The requirements of Claytronics introduce substantial additional
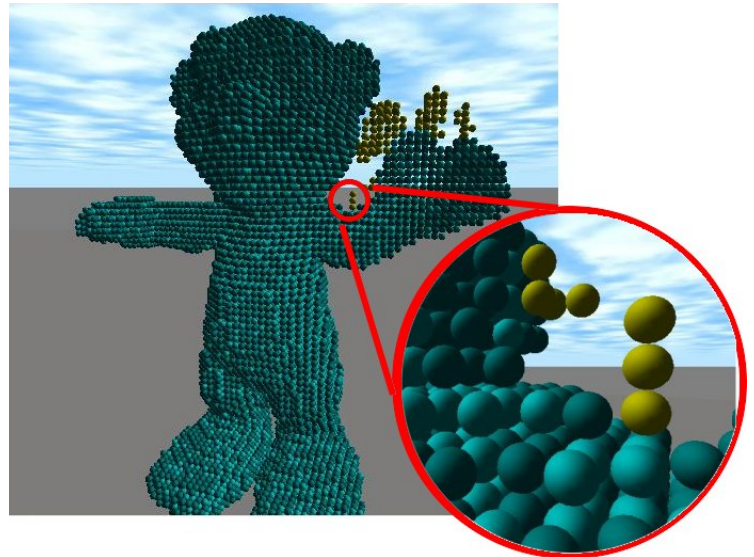

Fig. 1. The Claytronics vision for self-actuated shape-shifting materials composed of millions of tiny robotic modules called *catoms*, or Claytronic atoms. Catoms can compute, communicate, roll about their neighbors, and collectively act as a single robotic entity. Simulating the execution of distributed software on millions of catoms and their physical interactions requires a highly scalable and fast simulation framework.

challenges. In particular, the Claytronics vision seeks to create systems of many millions of interacting catoms, so any simulator must in itself be highly scalable. It must be able to simulate the execution of code and communications between a large number of robotic modules, although cycle-accurate modeling of processors is not generally necessary. Similarly, it must model both the kinematic and dynamic physical interactions between the modules, as well as their actuators and external forces on the system. Since Claytronics deals with shape change and motion, good visualization support is needed to assess system behavior.

Different experiments may require varying levels of fidelity. In particular, large-scale tests of distributed software may not require accurate physics modeling, and can make do with simple approximations. Similarly, an experiment that runs for a very long time may not require online visualization, instead making use of occasional dumps to produce coarser images. Thus, a modular simulation framework that allows a user to turn off particular components or switch between various implementations is a desirable feature. Furthermore, using only the required modules for a given experiment may allow improved speed or scalability.

In the rest of this paper, we first look at related approaches to modular robotic simulations, followed by an introduction to the original complete simulator for Claytronics. Next, we present how we parallelize and improve scalability of the physics engine used in the simulator. Following this, we describe our new, cluster-parallel simulation framework. Finally, we evaluate how well our system performs and finish with conclusions and a description of future work.

II. RELATED WORK

The modular robotics community has created a number of simulators in recent years, each with different design goals. Some of these simulators are designed only for particular hardware such as Proteo [2] or PolyBot [3] while others, such as USSR [4], are designed to work for arbitrary hardware models. Regardless, with a few exceptions, these simulators are limited in the number of modules they can handle. The USSR team, for instance, acknowledges that their framework may only support hundreds of modules which makes it unsuitable for our purposes.

There are two modular robotics simulators which claim to support 'large' numbers of modules. SRSim [5] has been used to run experiments on up to one million modules [6]. These experiments, however, simulated neither physics nor distributed execution. The simulator has a feature allowing for a centralized program to control the movements of the modules. While this permits handling large numbers of modules, it does not allow one to run distributed algorithms on them. DPRSim [7] can support simulating physics and distributed software execution on tens to hundreds of thousands of modules. Although this still falls short of the tens of millions we are targeting, it is the simulator upon which our work is based and will be discussed in greater detail below. Both of these simulators are limited in their capabilities because they can only run and harness the resources on individual machines.

Beyond the field of modular robotics, sensor networks and robotics provide some candidates for usable simulators. Unfortunately, TOSSIM [8], the *de facto* sensor network simulator, scales only to thousands of nodes and lacks any physics support. A multitude of simulators are used in the general robotics field. Most of these are limited to execution on a single machine and fail our requirement of supporting millions of modules. One notable exception is Microsoft Robotics Developer Studio [9], which is designed to make use of multiple machines. However, we are not aware of any efforts to support large number of robots, and there is little information on its scaling properties.

III. DPRSIM: THE ORIGINAL CLAYTRONICS SIMULATOR

The first complete simulator developed for Claytronics is called DPRSim [7]. It is an integrated system that includes code execution, physics, interactive visualization, debugging support, and a GUI-based world builder to construct experiment scenarios (Figure 2). Although intended to be scalable, the primary goal of its development was to create a working simulation platform that is sufficiently feature-rich to write
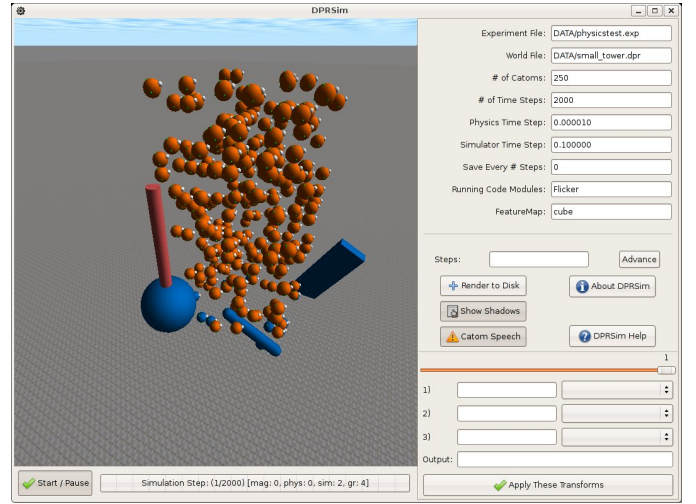

Fig. 2. Screenshot of the original DPRSim simulator for Claytronics. It incorporates simulation of distributed code execution, physics, visualization, a world builder, and interactive debugging support.

initial prototype applications for Claytronics. A command-line version also exists, which permits scriptable batch simulation experiments to be conducted.

The simulator framework has three main components that execute application code, simulate physics, and perform visualization respectively. The simulator takes two files as input: one that describes parameters of the experiment, and a second that describes the world state, essentially the starting positions of the catoms to be simulated. The main loop of the simulation is based on an abstract notion of *simulation tick*. In each tick, these three components are executed in turn to progress the simulation. We note that the tick is not directly associated with a specific notion of time; the user is free to define this as desired, for example to rationalize simulated time, forces and accelerations with real units.

Each tick, the visualization system is invoked to produce a representation of the world state. The visualizer is custom code that uses direct calls to OpenGL libraries, and can make use of accelerated hardware to speed up rendering. In addition to positions of the robotic modules, the system also displays debugging information in the form of colors, text callouts, lines, arrows, and overlay displays. If specified, a binary representation of the world state or a snapshot of the rendered scene can be saved to disk as well.

The core of the physics subsystem is the Open Dynamics Engine (ODE) [10], an open-source package simulating rigid body dynamics. This is a commonly used system for game development as well as many robotics simulation projects [11]. In addition to collision detection and integration of forces to compute catom motion, DPRSim uses the physics subsystem to efficiently determine proximity information. This is used to control which robots can communicate, determine sensor readings, and compute forces due to electromagnetic / electrostatic actuation (necessary because ODE does not have native support for EM fields).

Fig. 3. An ensemble of catoms transforming from a solid cube into a trumpet using a distributed metamodule-based shape change algorithm. This ensemble includes in excess of 165,000 modules. A complete video is available at http://www.cs.cmu.edu/~claytronics/movies/trumpet-1920x1200.mp4. This simulation is at the limit of what can be done with the original DPRSim, and requires over three weeks of processing time with physics disabled.

For software development, the most critical component is the application code execution system. This supplies a set of APIs that one may expect from an operating system or low-level monitor on a tiny robotic module, including primitives for communication, sensing, and controlling actuation hardware. The system is designed for simulating distributed algorithms as it creates and executes a separate instance of user-supplied application code for each catom, modeling inter-catom communications. Applications are implemented as a set of *code modules*, which are C++ classes that supply methods for initialization and cleanup, and a set of functions that are executed every simulation tick. The application writer has complete control over the computations performed in a tick; this fits with the notion of an abstract simulation tick, and the fact that cycle-accurate execution timings are not a goal of this simulator.

Several substantial applications have been developed on top of DPRSim, including a hierarchical localization algorithm [12] and a metamodule-based scalable shape change algorithm [13] (Figure 3). In addition, two distributed programming languages, LDP [14] and Meld [15], have been used on DPRSim — the former is an interpreted language with a runtime implemented as a code module, while the latter is a declarative logical language that compiles to C++ code modules.

Although DPRSim has been a great success and a valuable tool for Claytronics research, it is not quite capable of reaching the goal of multi-million module simulations. The large, complex, monolithic simulator has been faulted for being slow and for consuming large amounts of memory. The research team uses a special machine with 48GB of RAM to handle large DPRSim experiments. In the rest of this paper, we develop a new simulation framework to address these issues and enable larger and faster simulations.

IV. SCALING UP PHYSICS SIMULATION

The Open Dynamics Engine (ODE) is an open source physics engine for simulating rigid body dynamics. It is used in many games and 3D simulation tools to provide physics simulation support. In particular, DPRSim uses ODE heavily to model the dynamics of how catoms interact with each other. While ODE is fairly fast, it is implemented as sequential code, and misses opportunities to take advantage of parallel hardware for improved performance. Furthermore, it was not

designed for simulating millions of objects, and may bound the scalability of any simulator that uses it. To allow simulations of large systems of modular robots, we first focus on improving ODE scalability and parallelizing its execution on multiple processors.

Physics simulation with ODE uses discrete time steps, which are divided into two components expressed in Figure 4: a collision detection phase, and a time stepping phase. The former uses positions, orientations, and shapes of objects to determine which if any are touching, or are slightly embedded (collided). These points of interaction are used to create constraints that are added to a table of transient contact constraints. The latter phase attempts to step forward a small amount in simulated time. It uses a dynamics solver that adjusts positions and orientations of objects based on the current velocities and angular velocities, and then goes through the list of all constraints (including the contact constraints), and iteratively adjusts the position and velocity data to minimize constraint errors. The transient contact constraints are then deleted, and the cycle is repeated for the next time step.

Profiling of ODE execution reveals that significant processing time is spent in both collision detection and dynamics solving functions. Additionally, memory usage for collision detection was $O(n^2)$ in the number of objects that poses a limit on scaling of ODE to more than 10^5 objects, despite a small constant factor. Below, we show how we parallelize both of these algorithms across multiple cores, while avoiding costly locking and improving memory efficiency.

A. Parallel Collision Detection for ODE

In ODE, objects are stored in containers called *spaces*. Different implementations of spaces give the programmer a choice of storage methods with varying computational complexity for adding, removing, and checking for collisions among the objects. The only container type that scales to a large number of objects in ODE is the *HashSpace*, which is used in DPRSim and is the focus of our efforts. The HashSpace simply stores objects in a bidirectional linked-list, providing O(1) add and remove complexity (this is possible because objects are referenced by the pointer into this linked list).

To perform collision detection, the objects are first inserted into a hash table using multi-scale 3D spatial binning. The spatial bins at each scale are aligned, so a bin at one scale is equal to exactly 8 bins at the next smaller scale, and is wholly
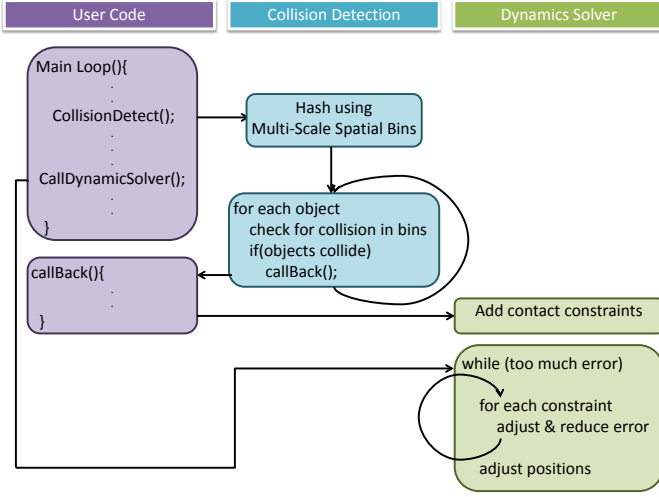
Fig. 4. Steps of physics simulation using ODE.

contained by exactly one bin at the next larger scale. Except for very large objects, each object is placed into bins at the smallest scale where the bin size is larger than the object's axis-aligned 3D bounding box. Due to alignment, each object will overlap with up to 8 bins. For each of these bins, a hash key is computed using the scale and position of the bin, and an entry added to the hash table for the object. Hash collisions are handled with chaining. The few very large objects (e.g., an infinite ground plane) are handled separately and kept aside in an oversized objects list.

Parallelizing the hash table construction using OpenMP is fairly straightforward – each thread is simply assigned a separate subset of the objects. The challenge is to manage concurrent access to the hash table entries. Standard locks proved to be too costly, and caused the threaded version to run more than an order of magnitude slower than the original code. Instead, we completely avoided software locks and coded the updates to the hash entries and oversized list using x86 atomic `xchg` and `lock cmpxchg` instructions. These provide atomic read-modify-write access at the hardware level, and are sufficiently lightweight that they can be used efficiently in the inner loop of the hash table construction routine.

Once the hash table is constructed, ODE then checks each object for collision against all objects in its assigned bins, and bins at a larger scale. If a collision is detected, a user-supplied callback function is called with the relevant object ids as parameters. This callback mechanism adds a great deal of flexibility to the system, and allows DPRSim to use ODE collision detection for a number of other purposes, including determining module neighbors and simulating interactions between nearby electrostatic actuators. To avoid repeated collision signals between a given pair of objects due to the objects residing in multiple bins, ODE keeps track of collided pairs using a bit matrix. Unfortunately, this has $O(n^2)$ space complexity, and although each entry is just 1 bit, the table will require many gigabytes of memory beyond 100,000 objects.

This limits the scalability of ODE, and for multi-threading, the table may be a point of serialization.

Our implementations removes the $O(n^2)$ table and parallelizes collision checking across multiple threads. Each thread is assigned a subset of the objects, and proceeds to test each object against others in its bins and in bins at a larger scale. In addition we skip testing against other objects of the same scale with greater ids, since, by symmetry, the iterations handling these other objects will perform the test on these pairs. The thread uses a local n-bit vector to avoid duplicate collision tests. The vector is zeroed before processing each object and used to maintain the set of objects that the current object has already been compared to. As each object is only processed once and only compared against objects with lower ids, each pair of objects is checked exactly once for collision. This implementation reduces the memory requirements to $O(n)$, and eliminates all locking in the inner loop. Of course, since multiple threads are performing collision checking, the user-provided callback function must be thread-safe.

Finally, all objects are tested for collision against those in the oversized objects list. This is partially parallelized – the threads that check each normal object against others in the hash table also test against those in the oversized list in a parallel fashion. Testing for collision among the objects in the oversized list is not parallelized, as this list is typically very small.

B. Parallel Dynamics Solver

The second major component of ODE is the dynamics solver, that attempts to step the simulated world forward in time, applying all forces to the objects while satisfying joint and contact constraints. The positions and velocities (both translational and rotational) of the objects, and the constraints that relate these form a giant linear system that needs to be solved to minimize errors (constraint violations). Rather than attempt to solve this directly in an analytical fashion, ODE's quickstep method uses a form of numerical approximation that iteratively reaches a low-error solution.

The implementation constructs two tables. The first lists all object positions and velocities (both translational and rotational). The second is a condensed form of the Jacobian matrix that defines how the object velocities change with respect to the constraints. Each row of this table corresponds to one constraint, containing a set of weights that define a linear combination of one or two objects' velocity vector elements for the constraint. The numerical solver iterates through the constraint table, computes the error for the particular constraint, and then adjusts the object parameters slightly to reduce the error. This process is repeated for a fixed number of iterations or until convergence (i.e., the global error is reduced below some threshold).

Because the inner loop of this solver performs writes to the object table, care must be taken in parallelization. The loop is very tight, so any additional instructions or locks will affect performance significantly. One cannot simply partition work to

threads based on the object affected, as the constraints operate on object pairs.

Despite this, we were able to parallelize the inner loop with multiple OpenMP threads, each processing a different subset of the constraints, with no locking. This is possible because for this algorithm, we can safely read and write concurrently from multiple threads. First of all, on x86 architectures, 64-bit reads and writes of aligned data are guaranteed to be atomic due to the fact that memory coherency is performed on cache-line sized blocks. This means that the updates of individual double-precision floating-point values in the object table will be atomic, and a concurrent read will not see a partially modified value. The entire set of parameters for the object, on the other hand, are not atomically updated. However, this does not matter for two reasons. With hundreds of thousands of objects, and millions of constraints, it is exceedingly unlikely that two threads will simultaneously process constraints that access the same object. Furthermore, if such an event occurs, a read of a partial update or an interleaved pair of updates is not catastrophic. Since the algorithm only makes small adjustments to the values in the table, these concurrent accesses can only introduce an additional small numerical error, bounded by the n-dimensional axis aligned cube defined by the previous and new values of the object parameters. The error can be large only if the differences between the values and the updates is large, but this indicates that numerical instability is already present in the system (e.g., extremely large forces, the step size is too large, etc.). Therefore, with the atomicity provided by the hardware, the extremely low likelihood of concurrent access, and little negative effect of any such access, we are able to parallelize the solver across threads with no explicit concurrency management.

V. A CLUSTER-PARALLEL SIMULATOR

The most critical aspect of Claytronics simulation is the ability to simulate the execution of code across a distributed set of modules. Much of Claytronics software research attempts to assess the behavior and scalability of algorithms across very large ensembles. To permit simulations of systems that scale beyond the resources available on individual machines, we have designed a new simulator, called DPRSim2, that exploits parallel execution across clusters of machines.

DPRSim2 is built using OpenMPI, an open-source MPI implementation, and employs a master-slave architecture. Multiple slave instances perform the bulk of the simulations. Each is responsible for a subset of the simulated robotic modules and executes the distributed software that runs on these modules. A single master node coordinates the slave nodes, and handles centralized aspects of the simulation, including initialization, saving state / checkpoints to disk, and executing centralized code. The architecture of DPRSim2 is illustrated in Figure 5.

DPRSim2 continues to use a discrete time model using an abstract notion of simulation tick. Each tick has two phases. In the first, only the master process runs, executing centralized algorithms and performing any simulation I/O. At the end of this phase, the master sends a message to each slave process,
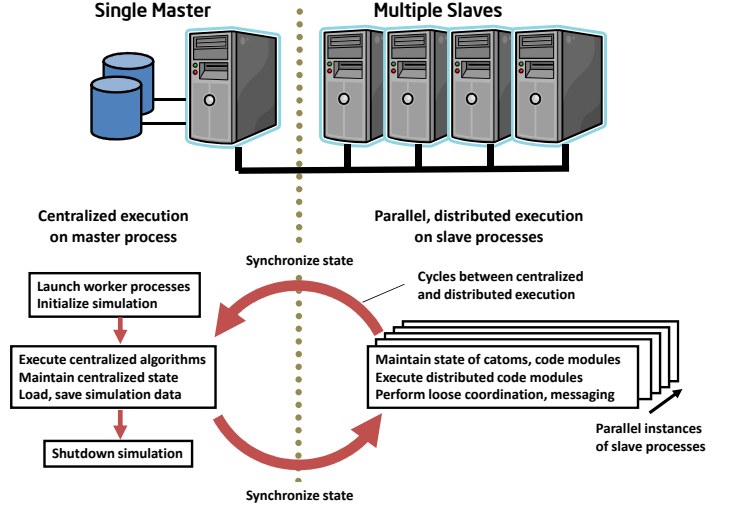

Fig. 5. Master-slave architecture of DPRSim2. Control flow between centralized and distributed execution during each simulation tick is also illustrated.

updating any simulation state and providing information to coordinate slave activities. In the second phase, the slave processes execute in parallel, simulating the execution of distributed software on the robotic modules. Communications between the simulated catoms is also handled in this phase. Messages that need to be sent to modules in a different slave process are aggregated and sent as a single MPI message to the appropriate slave. The master is used to coordinate this activity, so only the necessary messages are sent, rather than a complete $O(n^2)$ message exchange. At the end of the tick, the slaves inform the master of any state changes and pending inter-slave messages. Once the master receives tick completion messages from all slaves, it starts its execution for the next tick. The lower portion of Figure 5 illustrates the control flow cycle between centralized and distributed execution that occurs in each simulation tick.

DPRSim2 has been architected with speed and scalability in mind. Many of the integrated features of the original simulator have been removed from the core system. In particular, physics processing and visualization, two very costly components in terms of memory use and execution time, have been removed from the core simulator. Full ODE physics is available as an expansion module for those programs that require it, while other simulations can be run with simple magic movements to permit fast execution for those that don't require full physics. Visualization is relegated to external tools. However, the critical ability to execute both distributed software as well as centralized algorithms has been maintained.

DPRSim2 provides a modular, extendable framework for implementing custom simulation components and distributed software to run on simulated catoms. This framework is implemented as a hierarchy of C++ classes, and expands greatly on the concept of a CodeModule found in the original DPRSim simulator. Three classes of software extensions are defined: SimModules, CodeModules, and Components.

- SimModules are present only in the master process and are intended to provide additional functionality to the simulator. Since our parallel physics outlined earlier is limited to shared-memory systems, we incorporate it in DPRSim2 as a SimModule. Although visualization is now performed by a separate tool, we have defined a supporting SimModule that can send compressed updates to the external tool to show the current state of a running simulation.

- CodeModules reside on the slave processes, and are intended to define the software that executes on simulated catoms. The distributed shape change software we use in the evaluation is implemented as a CodeModule.

- Components run in both the master and slave context, and synchronize internal state between these contexts in each simulation tick. These are primarily intended to define the low-level services available on a catom. For example, inter-catom messaging service is implemented as a Component.

All of these categories of software modules can be defined as either runnable or nonrunnable. Runnable modules have a defined execution method that is invoked every simulation tick. Nonrunnable modules are passive and export a set of functions that are invoked on demand by other modules. DPRSim2 provides a clean inter-module access and calling interface that facilitates building and using nonrunnable service modules. In addition, DPRSim2 allows us to define multiple implementations of service modules that provide a common interface, and select the optimal version to use for a particular simulation run.

Finally, all categories of modules can also be defined as shared or not shared. The non-shared modules are instantiated for every catom, and are useful for defining the software that runs on them. Shared modules, instantiated once per slave machine, are used for centralized algorithms, including most of the simulation extensions, such as physics.

The variety of software module classes and the extensibility of the core simulator allow DPRSim2 runs to be very streamlined, with just the components necessary for a particular simulation. This aspect contributes to the scalability of the system and allows us to run larger and faster simulations.

VI. Evaluation

We discuss the effectiveness of our improvements upon both the original DPRSim and the ODE Physics engine. These are discussed separately because simulating physics still dominates the performance of the system when ODE is used. For simulations where magic movement is sufficient, the effectiveness of our enhancements is more substantial. Finally, we discuss the effect that these improvements have had on the size of the simulations that we are able to run.

A. Physics Simulations

As described in Section III, the ODE physics engine is used for multiple purposes in a typical Claytronics simulation:
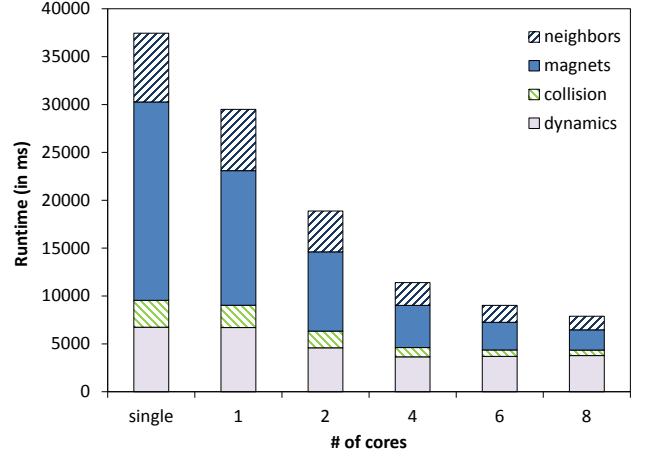
1) detecting neighboring catoms (neighbors)

Fig. 6. This graph shows the running time of 1 'tick', averaged across 10 runs. The runtime is broken up into 4 categories. The experiments were run on a machine with Dual Intel E5520 CPUs totalling 8 core @2.27GHz with 16GB of RAM.
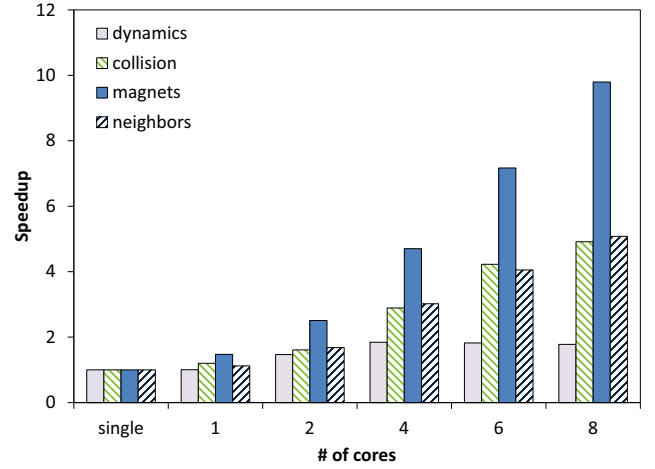

Fig. 7. This graph shows the speedup of each portion of physics run for a single tick, averaged across 10 runs. The experiments were run on a machine with Dual Intel E5520 CPUs totalling 8 core @2.27GHz with 16GB of RAM.
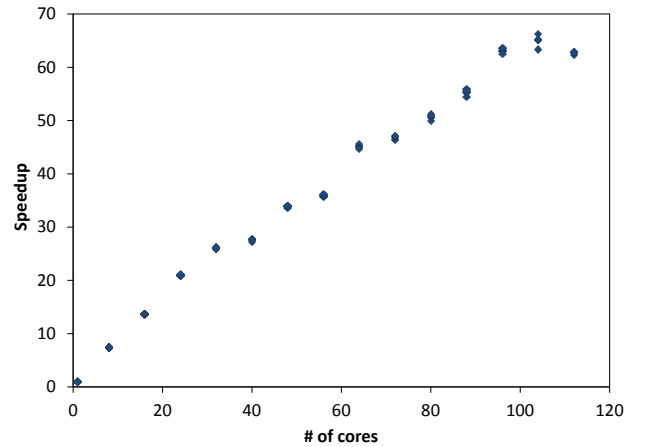

Fig. 8. This graph shows the speedup of the dprsim2 simulator running the trumpet experiment (shown in Figure 3), averaged across 5 runs. The experiments were run on 8 core machines with Dual Intel E5440 CPUs@2.83GHz with 8GB of RAM.
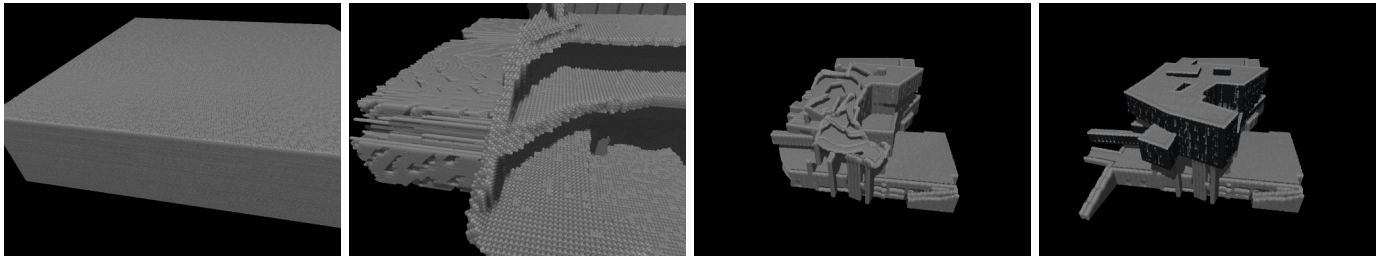
Fig. 9. An ensemble of catoms changing shape from a solid rectangular prism to the new Gates-Hillman Complex at Carnegie Mellon. This ensemble includes in excess of 1.6 million modules. It required two days of processing on 600 cores, and is an example of the kind of simulation that can now be routinely performed using DPRSim2. A complete video is available at http://www.cs.cmu.edu/~claytronics/movies/gates-color2.mp4

2) computing magnetic forces between catoms (magnets)
3) colliding catoms (collision)
4) computing catom motions (dynamics)

The breakup of time spent on these different aspects of physics are shown in Figure 6 for the original and our parallel implementation of ODE. The first three (neighbors, magnets, and collision) aspects rely on the collision detection methods of ODE, described in Section IV-A, and account for over 80% of the time spent in the original physics engine. In our multi-threaded version, they account for only about 50% of the time in the engine on account of the collision detection attaining better speed-ups than the dynamics solver.

The speed-ups for the different portions are shown in Figure 7. Notably, the multi-threaded version of collision detection is faster with a single thread than the original single-threaded version. The reason for this is the reduction of an $O(n^2)$ bit array to an $O(n)$ bit array in the collision detector, as described in Section IV-A. In particular, given that n is 250,000 in this example, this change greatly reduces the memory that needs to be touched, improving the runtime. The magnets portion benefits even more from this fix because the runtime is based on the number of magnets — as each catom has multiple magnets, there are substantially more magnets than catoms.

The dynamics solver only manages an approximate 2X speedup, even with 8 cores. We believe this is due to its tight inner loop with large number of noncontiguous memory references. This loop is likely to be memory bound, so parallelizing on multiple threads has limited gains.

The overall speedup of ODE for these simulations is 4.75X on 8 cores. This is adequate to run much larger simulations than before.

B. Distributed Software Simulations

Running on 80 cores DPRSim2 ran the large trumpet simulation in Figure 3 over 108X faster than DPRSim. The DPRSim2 simulator runs faster using a single thread than the original DPRSim simulator due to the modular architecture that allows executing simulations with only the required portions of the simulator enabled. Furthermore, the simulator itself scales up very well, as shown in Figure 8, achieving speedups of 66X on 104 cores. The limiting factor here is not load on the master node as one might expect. In the absence of

physics, the master node is still very much idle. The limitations on scaling stem from an imbalance in the workload.

The simulator is capable of allocating the same number of catoms to each core, but not all catoms require the same amount of work to process. Unfortunately, processing each catom requires a variable amount of work dependent upon the program running on them. Without understanding the programs, it is not clear how to accurately balance the workload. The varying types of workloads further complicates matters. Some programs are very heavy on computation and the most important part of allocation is ensuring the cores have equal numbers of busy catoms. Other programs are far more message heavy and benefit more from putting adjacent catoms on the same core to minimize messaging overhead. It is not clear that there is a single optimal allocation strategy as we have found that some do well for certain programs and relatively poorly for others.

C. Experiment Size

The primary goal of this work was to increase the size of experiments that we can plausibly run. In this objective, we have succeeded. The original DPRSim was capable of running experiments (without physics) with 10s of thousands of catoms over a reasonable time frame of hours or days and up to a maximum size of 100s of thousands of catoms (on a sufficiently memory rich machine) over the course of weeks, such as the example shown in Figure 3. With DPRSim2, these numbers increase by at least two orders of magnitude. We are able to routinely run experiments with 100s of thousands or millions of catoms, such as the one shown in Figure 9. We have demonstrated experiments in excess of 20 million catoms and have no reason to believe we have hit the limit of the system.

For experiments with physics, the limitations are more significant. Even with our enhancements to ODE, experiments with 100s of thousands of catoms can take a very long time to run. With the improvements to memory utilization, experiments with millions are now possible but not yet practical.

VII. CONCLUSIONS AND FUTURE WORK

We have created and presented here a simulation framework that efficiently harnesses hundreds of cores, permitting us to routinely run (algorithmic) experiments with millions of simulated robotic modules. These experiments are two orders of

magnitude larger than what was previously possible, and have greatly advanced the Claytronics project. Furthermore, we expand the capabilities of the ODE dynamics engine to permit larger experiments with physics. While these experiments can now be run with millions of modules, such experiments run very slowly. We plan to investigate the possibility of further improving our parallel physics to allow it to scale beyond shared memory systems, and harness the computation power of clusters, as we have done with the rest of DPRSim2. This will permit routine future simulations with distributed code execution and full physics interactions among tens of millions of robotic modules.

We have made our parallel simulation framework available on the Claytronics website: http://www.cs.cmu.edu/~claytronics.

REFERENCES

[1] S. C. Goldstein, T. C. Mowry, J. D. Campbell, M. P. Ashley-Rollman, M. De Rosa, S. Funiak, J. F. Hoburg, M. E. Karagozler, B. Kirby, P. Lee, P. Pillai, J. R. Reid, D. D. Stancil, and M. P. Weller, "Beyond audio and video: Using claytronics to enable pario," *AI Magazine*, vol. 30, no. 2, July 2009.

[2] Palo Alto Research Center, http://www2.parc.com/spl/projects/modrobots/lattice/proteo/index.html.

[3] Palo Alto Research Center, http://www2.parc.com/spl/projects/modrobots/chain/polybot/simulations/index.html.

[4] D. Christensen, D. Brandt, K. Stoy, and U. Schultz, "A unified simulator for self-reconfigurable robots," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, sept. 2008, pp. 870 –876.

[5] R. C. Fitch, "Heterogeneous self-reconfiguring robotics," Ph.D. dissertation, Dartmouth College, Hanover, NH, USA, 2005, chair-Rus, Daniela.

[6] R. Fitch and Z. Butler, "Million module march: Scalable locomotion planning for large self-reconfiguring robots," in *Robotics: Science and Systems, Workshop on Self-Reconfigurable Robots*, 2006.

[7] "The physical rendering simulator (dprsim)," http://www.pittsburgh.intel-research.net/dprweb/.

[8] P. Levis, N. Lee, M. Welsh, and D. Culler, "Tossim: Accurate and scalable simulation of entire tinyos applications," 2003.

[9] "Microsoft robotics developer studio," http://www.microsoft.com/Robotics/.

[10] "Open dynamics engine (ode)," http://www.ode.org.

[11] "Openrave - a planning environment for autonomous robotics," openrave.programmingvision.com/.

[12] S. Funiak, P. Pillai, M. P. Ashley-Rollman, J. D. Campbell, and S. C. Goldstein, "Distributed localization of modular robot ensembles," *International Journal of Robotics Research*, 2008.

[13] D. Dewey, S. S. Srinivasa, M. P. Ashley-Rollman, M. De Rosa, P. Pillai, T. C. Mowry, J. D. Campbell, and S. C. Goldstein, "Generalizing metamodules to simplify planning in modular robotic systems," in *Proceedings of IEEE/RSJ 2008 International Conference on Intelligent Robots and Systems IROS '08*, Nice, France, September 2008. [Online]. Available: http://www.cs.cmu.edu/~claytronics/papers/dewey-iros08.pdf

[14] M. De Rosa, S. C. Goldstein, P. Lee, J. D. Campbell, and P. Pillai, "Programming modular robots with locally distributed predicates," in *Proceedings of the IEEE International Conference on Robotics and Automation ICRA '08*, 2008. [Online]. Available: http://www.cs.cmu.edu/~claytronics/papers/derosa-icra08.pdf

[15] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai, "Meld: A declarative approach to programming ensembles," in *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS '07)*, October 2007. [Online]. Available: http://www.cs.cmu.edu/~claytronics/papers/ashley-rollman-iros07.pdf