Lock-Graph: A Tree-Based Locking Method for Parallel Collision Handling with Diverse Particle Populations

Mark Lewis[1] **and Cameron Swords**[1]
[1]Computer Science, Trinity University, San Antonio, TX, USA

Abstract— *This paper builds on earlier work that used a spatial grid for locking to provide physically accurate parallel collision handling. Instead of using a grid, this work uses a spatial tree. The tree is better able to handle heterogeneous particle populations. The method was specifically developed to handle a granular flow impact simulation where one large body impacts a population of smaller bodies. The large size of the impactor leads to a breakdown in the grid based locking strategy because grid cells are uniformly sized and must be large enough to enclose the largest particle in the population plus the relative velocity distribution multipled by a time step. The tree allows the regions that get locked to scale in size based on local characteristics, making it possible to handle dramatic size differences. Unfortunately, it also has more overhead than the grid so using it when it is not needed can slow simulations down.*

Keywords: Simulation, collisions, parallel, discrete-event

1. Introduction

Parallelizing collisions in a physically accurate way is inherently challenging. This is because, unlike longer range forces such as gravity, collisions are temporally sensitive. The behavior is sensitive to the order of collisions, as one collision can alter the path of particles to prevent other collisions or to make them happen at different times. Fortunately, the short-range nature of collisions means that over a fixed period of time, one can set bounds on how far the effect of a particular collision will travel. This range can be though of as the speed of sound in the medium times the length of time being considered. This logic leads to spatial locking where collisions are processed in order, but parallel processing of future collisions is allowed as long as those collisions are far enough away from one another (1; 2; 3).

This dependence on temporal ordering is not uncommon in discrete event simulations, indeed, it is the norm. Significant effort has gone into finding ways to parallelize such systems (4; 5; 6). A common solution to this is to implement the ability to roll back changes (7; 8; 9). For a general discrete event simulation that solution typically allows greater parallelism than trying to keep things together, but it comes at the expense of memory. For this application, roll back is not really an option. The storage of the rollback information would have a computational cost that would rival what was gained from parallelism. More importantly, these simulations are often memory constrained. Particle counts in physical, collisional simulations can get as high as a few times 10^8, and for some situations that is the limit which constrains what can be done. In the case of planetary rings, a simulation with 10^8 particles is near the minimum required for getting decent resolution with a ring that goes all the way around the planet. Even then it must be a narrow ring which precludes some types of work. Adding the memory overhead of storing older states for particles we can roll back to would further depress the maximum simulation size for a given cluster configuration.

In our previous work (1), locking was done using a uniform grid. The grid cells are made large enough that during one time step, the probability of a particle colliding with another particle whose center is located two or more cells away is effectively zero. This works well enough for the majority of ring simulations. Indeed, for simulations where the population is fairly homogeneous in size and spatial distribution and the velocity distribution has a small dispersion, this method is optimal. For those simulations, the same grid can be used as the spatial data structure used for searching potential collision pairs as well, so there is a net benefit in sharing the data structure.

Such a grid is less ideal for finding collisions when the particle distribution has heterogeneities. For example, if there are a small number of significantly larger particles, if the particles are very non-uniformly distributed in space, or if there are spatial variations in the velocity distribution. For these situations, a tree is better as the primary spatial data structure for finding collisions. We also found, for one particular simulation, that the grid approach to locking could be untenable.

Figure 1 shows a granular particle simulation in which a marble is dropped into a dish of silicon spheres. When this was first attempted using the grid for locking, the simulation bogged down to a point where it wasn't going to finish in a reasonable period of time. This prompted the development of the method described here.

2. Methodology

The key goal for the new method was to be flexible enough to handle the situation shown in fig. 1. Building off a spatial tree was a natural approach because these simulations use a tree instead of a grid for collision finding and we could

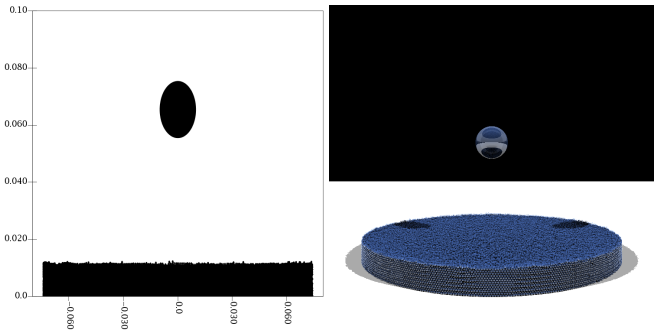

Fig. 1

THIS SHOWS A FRAME FROM THE SIMULATION THAT MOTIVATED THIS WORK. AN INITIAL SIMULATION WAS DONE IN WHICH UNIFORM SMALL PARTICLES WERE DROPPED AND ALLOWED TO SETTLE IN A DISH. THAT CONFIGURATION WORKED FINE WITH THE GRID LOCKING. THEN A SINGLE IMPACTOR WAS ADDED THAT WAS 20 TIMES LARGER IN RADIUS THAN THE OTHER PARTICLES AND DROP IT INTO THEM. THIS CAUSED THE GRID LOCKING TO FAIL, AS THE LARGE SIZE AND HIGH VELOCITY OF THE IMPACTOR LED TO A LARGE GRID SIZE.

reap the benefits of having a mutual data structure for both purposes. To be competitive with the grid, the method also needed to be fast. The grid allows O(1) access to the eight adjacent cells to check if a collision is being processed in one of them so we can know if it is safe to process a collision in the current cell. Solutions that run through the tree would be O(log n), where n is the number of particles, and would have a higher coefficient due to the overhead of moving through the tree. The method presented here uses the tree as the spatial data structure, but it also builds a graph through the tree that we call the lock-graph, which retains the O(1) access aspects of the grid and the dynamic and spatially variable nature of the tree.

The basic idea of this method is that in the spatial tree certain nodes are labeled as lock-nodes. Only the lock-nodes are significant in determining if a collision can be processed or not. The path from the root of the tree to any leaf (where particles are contained), can contain only a single lock-node. Each lock-node much be at least as large as the largest particle plus a multiple of the velocity dispersion for the particles under that node. The tree knows these values already as they are used in the collision pair searching algorithm. It would be simple enough to have a recursive function that runs through the tree finding the nodes that should be lock nodes based on the specified size criteria. However, we also need to know about “adjacent” nodes. For any given lock node there will be other nodes around it that are within range and must be searched for collisions. Simple recursion doesn’t give us that.

To make it so we can quickly find the adjacent nodes, we need to have a graph where edges connect the lock nodes in the tree that are adjacent for the purposes of collision finding. This structure is what we call the lock graph. It contains all the lock nodes with edges between any lock nodes that contain particles that could impact one another. This structure can be built at the same time that lock nodes are picked using recursion over two arguments. This pseudo-code shows how it works.

```
def buildLockGraph(n1:Node,n2:Node) {
  if(n1==n2) {
    val lock = n1.numParts>=0 ||
      n1.size<SCALE*(n1.maxRad+n1.searchRadius)
    if(!lock1) {
      buildLockGraph(n1.firstChild,n1.firstChild)
      buildLockGraph(n1.firstChild,n1.secondChild)
      buildLockGraph(n1.secondChild,n1.secondChild)
    }
  } else {
    val dx = n1.mid.x-n2.mid.x
    val dy = n1.mid.y-n2.mid.y
    val dz = n1.mid.z-n2.mid.z
    val dist = sqrt(dx*dx+dy*dy+dz*dz)
    val searchDist = n1.searchRadius+n1.maxRad+
      n2.searchRadius+n2.maxRad
    if(dist-0.866*(n1.size+n2.size)<=searchDist) {
      val lock1 = n1.numParts>=0 ||
        n1.size<SCALE*(n1.maxRad+n1.searchRadius)
      val lock2 = n2.numParts>=0 ||
        n2.size<SCALE*(n2.maxRad+n2.searchRadius)
      if(lock1 && lock2) {
        addLockEdge(n1,n2)
      } else if(lock1) {
        buildLockGraph(n1,n2.firstChild)
        buildLockGraph(n1,n2.secondChild)
      } else if(lock2) {
        buildLockGraph(n2,n1.firstChild)
        buildLockGraph(n2,n1.secondChild)
      } else {
        buildLockGraph(n1.firstChild,n2.firstChild)
        buildLockGraph(n1.firstChild,n2.secondChild)
        buildLockGraph(n1.secondChild,n2.firstChild)
        buildLockGraph(n1.secondChild,n2.secondChild)
      }
    }
  }
}
```

There are two top level cases. The two nodes could be the same or they could be different. If they are the same, we only decide if the node should be a lock-node. If it isn’t, then we need to recurse three times to the different possible combinations of the two children of the node.

If the nodes aren’t the same, the distance between their center points is calculated and that is checked against their sizes and the search distance of the two nodes. If they are too far apart, the recursion terminates. Otherwise the code determines if either one is a lock node. If both are, that edge is added to the graph. If only one is, that one recurses against both children of the other. If neither is, all four combination of the children are recursed on.

This function is called using the root of the tree for both n1 and n2. When it is done, all lock nodes have been identified and all edges have been added. This form of two argument recursion can have many uses in spatial work when the objective is to find pairs of entities. One other application was presented in (10). It can also be nicely parallelized so that it does not add to sequential overhead in the application.

At this point the lock-graph can be used in much the

same way the grid was used previously. When the processing begins on a collision, the lock-node for each of the two particles is set to being locked. To make this efficient, a map from particles to nodes is made when the graph is built. When the collision is done, the lock is released. To check if a particular collision is safe to be processed, the code checks that node for each of the two particles in the collision and all the other nodes directly connected to them in the graph.

The check in the pseudocode for whether or not it is a lock node includes a factor called SCALE. This can be adjusted to move the lock-nodes up or down in the tree. Moving them up leads to fewer lock nodes that have more particles in them, but there are fewer connections between lock nodes. Moving them down has the opposite effect. Increasing the number of particles in a lock node can lead to over-locking if the lock-nodes get too big. However, the fact that the check for whether a collision is safe or not must run through all the edges out of a node means that lower connectivity can have a positive performance impact on one of the most common operations in the simulation. The impact of this is explored in the next section.

3. Analysis

The method was implemented in a collisional dynamics simulation code in C++. This code has been used previously for modeling of planetary rings (11; 12; 13). It was tested on two general types of systems, the system for which it was created shown in fig. 1 and a system using a small cell in a planetary ring shown in Figure 2. Both simulations included a bit under 200,000 particles and they were run through ten time steps to get the timing results. In addition, each system was run in two configurations called early and late. The early was the initial conditions and the late was after the system had evolved to an equilibrium. For the ring simulation the early is truly uniform with a dynamically cold particle distribution. The late was after two orbits when particle self-gravity and collisions had altered the distribution and particles were beginning to clump a bit. For the silicon grain simulations, the early stage had a cube of well spaced particles completely inside the cylinder falling down to the surface. The late simulation is what is shown with the particle at rest at the bottom of the cylinder.

To do the time testing we used a server with 4 Quad-Core AMD Opetron 8350 processors, running at 2.0GHz processor, giving a total of 16 cores. The C++ code base was compiled using the x86 Open64 compiler from AMD. The compile flag were "-Ofast -mso -march=auto -openmp". All multithreading was done using OpenMP. The "-apo" autoparallelization flag for the compiler was not used as the objective is to test the parallelism coded explicitly into the simulation. It is worth noting that choice of compiler can be significant. While the x86 Open64 compiler was selected under the belief that it would out perform the GNU C++ compiler, incomplete results from that compiler show

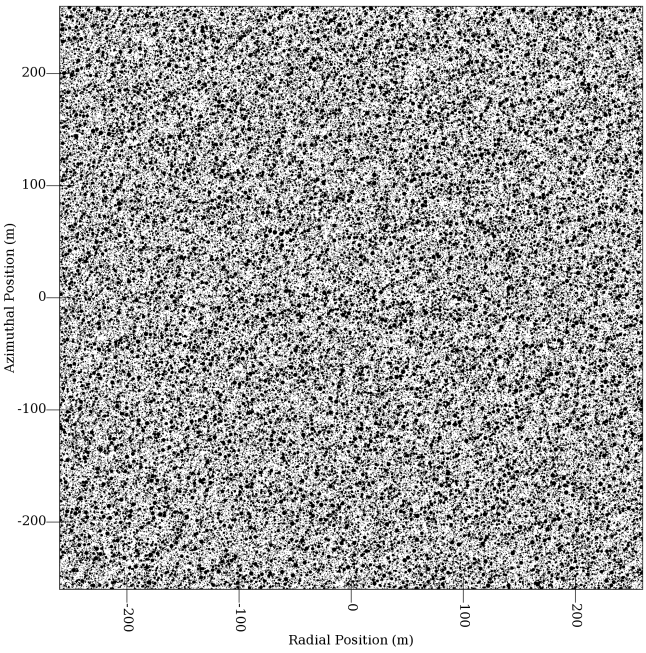

Fig. 2

THIS FIGURE SHOWS A FRAME FROM THE RING PARTICLE SIMULATION THAT WAS USED FOR TIMING RESULTS. THIS IS WITHOUT THE MOON. THE SIMULATIONS WITH A MOON HAD A 20-M PARTICLE PLACED AT THE ORIGIN.

that things aren't so clear cut. For some of the situations presented, the GNU compiler produces slightly faster code.

The results in the tables show what was reported by the Linux time command. Each simulation was run five times with the mean and standard deviation being presented.

The first table shows timing results for the ring simulation shown in fig. 2. This was the situation for which the grid was originally developed. The particles are fairly homogeneously spread out and the particle distribution is quite flat. The timing data here shows that for this system, larger lock-nodes are generally better and that a SCALE of ~6.0 is ideal. It isn't too surprising that even with that scale, the grid implementation is slightly better.

Table 2 shows the times for the ring simulation when a moonlet, 10 times larger in radius than the largest other particles, was dropped into the middle of the simulation. This setup was a bit artificial because the system was only advanced 10 steps which isn't enough time to let smaller particles settle on the surface of the moonlet, but it does test the graph with a heterogeneous particle size distribution. The results for this test were very odd as the graph timing for the initial conditions was highly variable and none of the graph runs performed as well as the grid.

A possible explanation for this is shown in Figure 3. The large particle forces one large lock node in the graph.

Scale	Graph Early (s)	Graph Late (s)
1.2	44.4 ± 0.4	58.3 ± 0.9
2.0	44.6 ± 0.1	57.0 ± 1.0
4.0	42.0 ± 0.1	47.7 ± 0.2
6.0	39.9 ± 0.2	43.7 ± 0.2
8.0	39.2 ± 0.1	43.8 ± 0.1

Table 1

THIS TABLE SHOWS TIMING RESULTS FROM RING SIMULATIONS WITH A POPULATION OF PARTICLES WHOSE SIZES WERE PULLED FROM A DIFFERENTIAL POWER-LAW DISTRIBUTION WITH A SLOPE OF -2.8 SPANNING A FACTOR OF TEN IN RADIUS. TIMES FOR THE GRID BASED LOCKING WITH THIS SIMULATION WERE 37.1 ± 0.1 EARLY AND 42.0 ± 0.1 LATE. FOR THE LARGER GRAPH SCALE THE GRAPH WAS COMPARABLE IN SPEED TO THE GRID, BUT NEVER SUPERIOR.

Scale	Graph Early (s)	Graph Late (s)
1.2	95.2 ± 5.7	75.3 ± 0.8
2.0	136.4 ± 6.2	75.5 ± 1.2
4.0	177.8 ± 7.2	67.5 ± 0.6
6.0	112.2 ± 3.0	66.2 ± 0.7
8.0	212.7 ± 7.6	66.5 ± 1.1

Table 2

THIS TABLE SHOWS TIMING RESULTS WHEN A MOONLET, 10 TIMES LARGER IN RADIUS THAN THE LARGEST BACKGROUND PARTICLES, WAS ADDED TO THE SIMULATION. THE TIMING FOR THE GRID IN THIS SITUATION WAS 93.0 ± 2.0 EARLY AND 57.9 ± 0.3 LATE.

That node has a very high connectivity. If there aren't enough collisions happening in nodes that aren't adjacent to that large node, the workload will be unbalanced and the processing will become more sequential. It is worth further exploration into whether having a more natural particle configuration or a larger cell would favor the graph more.

We also have time results for the granular flow simulations that prompted the development of the graph-lock method. This simulation differs from the ring simulations in another very significant way, the distribution of particles is distinctly 3-D. Planetary rings are remarkably flat. While the 3-D aspect of the rings is very significant to the dynamics (14), all the particles are close to the orbital plane. That is significant for the grid, which only breaks the simulation region up in 2-D. It would be possible to make a 3-D grid, but memory overhead quickly becomes a problem with 3-D grids that have decent spatial resolution, and while the neighbor count in 2-D is only eight, it goes up to 26 for 3-D, increasing the overhead of lock checking.

The results without the marble show how different this system is from the ring simulations. Here, increasing the scale of the lock nodes has little impact on performance when the particles are spread out early and slows it down when they are densely packed after having settled to the bottom. In addition, while the grid performs roughly the same speed as the graph for the early system, it is slightly

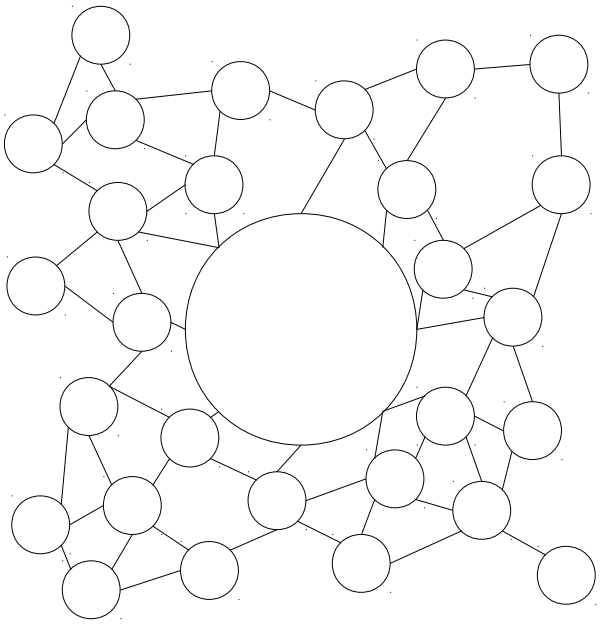

Fig. 3

THIS IS A PICTORIAL REPRESENTATION OF A LOCK GRAPH WHEN THERE IS A SINGLE LARGE PARTICLE. THE SIZE OF THE NODES ROUGHLY REPRESENTS THE AREA IT COVERS. THE NODE WITH THE LARGE PARTICLE HAS A MUCH HIGHER CONNECTIVITY AND IT IS POSSIBLE THAT CHECKS FOR ITS LOCK COULD ACTUALLY SLOW THINGS DOWN IF THERE AREN'T ENOUGH COLLISIONS HAPPENING AWAY FROM IT TO KEEP THE LOAD BALANCED.

Scale	Graph Early (s)	Graph Late (s)
1.2	21.0 ± 0.6	860 ± 39
2.0	20.6 ± 0.3	859 ± 30
4.0	20.7 ± 0.8	913 ± 40
6.0	20.5 ± 0.7	936 ± 22
8.0	19.3 ± 0.5	968 ± 33

Table 3

THESE ARE TIMING RESULTS FOR THE SILICON GRAIN SIMULATIONS WITHOUT THE DROPPING MARBLE. THE GRID BASED LOCK TIMING FOR THESE SIMULATIONS WERE 19.5 ± 0.5 EARLY AND 1004 ± 38 LATE.

slower later on. In all cases though, the 3σ bounds overlap.

Adding the marble is where the graph should in theory stand out. Again, the performance is fairly flat with graph scale except for the largest size which caused it to run significantly slower. The grid performed similarly to the largest scale for the early time. It is interesting to note that in this situation there isn't much of a difference between the early and late simulations using the graph. One result that seems a bit unusual is that the late setup runs faster with the marble than it did without. This implies that having the marble located up high actually changes the morphology of

Scale	Graph Early (s)	Graph Late (s)
1.2	647 ± 37	667 ± 50
2.0	668 ± 41	663 ± 39
4.0	686 ± 52	705 ± 24
6.0	667 ± 46	646 ± 13
8.0	2529 ± 1323	2780 ± 1185

Table 4

THESE ARE TIMING RESULTS FOR THE SILICON GRAIN SIMULATIONS WITH THE DROPPING MARBLE. THE GRID BASED LOCK TIMING FOR THESE SIMULATIONS WERE 2654 ± 1153 EARLY AND $\gg 150,000$ LATE.

the graph as a whole in such a way that it allows greater parallelism. Lastly, with the marble at the late time it wasn't possible for us to get good time bounds on the grid method. After a few days of running it still hadn't completed the first time step which allows us to give the limit in the table caption.

4. Conclusions

The lock-graph method of locking parallel threads adds another option to the toolkit of those working in spatially oriented discrete event simulations. While specifically designed for collisional simulations, it could be used to good effect in any discrete event application where events have a spatial distribution and exhibit locality. Unlike rollback methods, this does not require significant memory overhead and can be used for simulations that are often memory constrained. The use of two-argument recursion to built the lock-graph means that it has a worse case performance of O(N log N), the same was what would be required to built or maintain the tree data structure.

As is so often the case in simulation, the lock-graph is no silver bullet. There are trade-offs and some situations where it is not the ideal approach. For systems that are naturally flat and mostly homogeneous, a grid will work better. However, the use of a kD-tree for the lock-graph means that it will efficiently scale to higher dimensions where a grid would become infeasible and the ability to handle 3-D systems that exhibit significant heterogeneity makes it appropriate for systems where the grid-based approach breaks down.

Acknowledgments

This work was supported by grants from NASA Origins and NSF AAS.

References

[1] M. Lewis and B. L. Massingill, "Multithreaded collision detection in java," in *PDPTA*, H. R. Arabnia, Ed. CSREA Press, 2006, pp. 583–592.

[2] M. Lewis, M. Maly, and B. L. Massingill, "Hybrid parallelization of n-body simulations involving collisions and self-gravity," in *PDPTA*, H. R. Arabnia, Ed. CSREA Press, 2009, pp. 324–330.

[3] B. L. Massingill and M. Lewis, "Parallelizing a collisional simulation framework with plpp (pattern language for parallel programming)," in *PDPTA*, H. R. Arabnia, Ed. CSREA Press, 2006, pp. 608–614.

[4] R. M. Fujimoto, "Parallel discrete event simulation," *Commun. ACM*, vol. 33, pp. 30–53, October 1990. [Online]. Available: http://doi.acm.org/10.1145/84537.84545

[5] E. Mascarenhas, F. Knop, and V. Rego, "Parasol: a multithreaded system for parallel simulation based on mobile threads," in *Proceedings of the 27th conference on Winter simulation*, ser. WSC '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 690–697. [Online]. Available: http://dx.doi.org/10.1145/224401.224711

[6] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.*, vol. 117, pp. 1–19, March 1995. [Online]. Available: http://portal.acm.org/citation.cfm?id=201627.201628

[7] R. M. Fujimoto, K. S. Perumalla, A. Park, H. Wu, M. H. Ammar, and G. F. Riley, "Large-scale network simulation: How big? how fast?" in *MASCOTS*. IEEE Computer Society, 2003, pp. 116–.

[8] R. M. Fujimoto, *Parallel and Distributed Simulation*. John Wiley & Sons, Inc., 2007. [Online]. Available: http://dx.doi.org/10.1002/9780470172445.ch12

[9] A. M. Law, *Simulation Modeling and Analysis*, 4th ed. McGraw-Hill Higher Education, 2007.

[10] M. Lewis and H. Levison, "A tree-based hamiltonian for fast symplectic integration," in *CSC*, H. R. Arabnia, Ed. CSREA Press, 2008, pp. 30–36.

[11] M. C. Lewis and G. R. Stewart, "Expectations for Cassini observations of ring material with nearby moons," *Icarus*, vol. 178, pp. 124–143, Nov. 2005.

[12] ——, "Features around embedded moonlets in Saturn's rings: The role of self-gravity and particle size distributions," *Icarus*, vol. 199, pp. 387–412, Feb. 2009.

[13] S. J. Robbins, G. R. Stewart, M. C. Lewis, J. E. Colwell, and M. Sremčević, "Estimating the masses of Saturn's A and B rings from high-optical depth N-body simulations and stellar occultations," *Icarus*, vol. 206, pp. 431–445, Apr. 2010.

[14] H. Salo, "Numerical simulations of dense collisional systems," *Icarus*, vol. 90, pp. 254–270, Apr. 1991.

[15] H. R. Arabnia, Ed., *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems and Applications, PDPTA 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 1*. CSREA Press, 2006.