Parallelizing the Spot Model for Dense Granular Flow

18.337 Parallel Computing
Yee Lok Wong      May 8, 2008
Department of Mathematics, MIT
Part 1: Background on Granular Flow and the Spot Model
Microscopic Flow Mechanism of Granular Materials

**Gas**
Dilute, random “packing”
- Boltzmann’s kinetic theory
- Random collisions

**Crystals**
Dense, ordered packing
- Vacancy and Interstitial diffusion
- Dislocations and defects

**Granular**
Dense, random packing
- Long lasting many-body contacts
- Lack of general microscopic model
- How to describe cooperative random motion?
Spot Model

- “Spot” Model for random packing dynamics (Bazant et al., 2001)

- Developed for Silo Drainage
  - Spots - extended region of slightly enhanced interstitial
  - Spot move upwards from orifice, and also perform random walk at horizontal directions
  - When spots pass through particles, particles are displaced in the opposite direction
Velocity Correlation

- Motivation for Spot Model: Local velocity correlation suggests *correlated* motion

Experiments by MIT Dry Fluids Lab

Simulation
Spot Model Microscopic Mechanism

- Apply the spot displacement first to all particles within range
- Particles are displaced in the opposite direction
Spot Model Microscopic Mechanism

- Apply a relaxation step to all particles within a larger radius
- All overlapping pairs of particles experience a normal repulsive displacement (soft-core elastic repulsion)
- Very simple model - no “physical” parameters, only geometry.
Spot Model Microscopic Mechanism

- Combined motion is bulk spot motion, while preserving packings
- Not clear *a priori* if this will produce realistic flowing random packings
DEM Simulations

- Discrete Element Method (DEM), codes developed by Sandia National Lab.
- Each particle is accurately modeled according to Newton’s laws and a realistic friction model is employed to capture particle interactions
- Parallel code on 24 processors
- $50d \times 8d \times 110d$ container
- Drained from circular orifice $8d$ across

L. E. Silbert et al., Phys Rev E, 64, 051302 (2001)
Spot Simulations using C++

- Initial packing taken from DEM
- Spots introduced at orifice
- Spots move upwards and do random walk horizontally
- Systematically calibrate three parameters from DEM:
  - Spot radius $R_s$ (from velocity correlations)
  - Spot volume $V_s$ (from particle diffusion)
  - Spot diffusion rate $b$ (from velocity profile width)
Comparison with DEM simulation

DEM

Spot Model

<table>
<thead>
<tr>
<th>t (s)</th>
<th>DEM</th>
<th>Spot Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.05 s</td>
<td><img src="1.05s.png" alt="Image" /></td>
<td><img src="1.05s.png" alt="Image" /></td>
</tr>
<tr>
<td>2.10 s</td>
<td><img src="2.10s.png" alt="Image" /></td>
<td><img src="2.10s.png" alt="Image" /></td>
</tr>
<tr>
<td>3.15 s</td>
<td><img src="3.15s.png" alt="Image" /></td>
<td><img src="3.15s.png" alt="Image" /></td>
</tr>
<tr>
<td>4.20 s</td>
<td><img src="4.20s.png" alt="Image" /></td>
<td><img src="4.20s.png" alt="Image" /></td>
</tr>
</tbody>
</table>
Comparison with DEM simulation

- DEM: 3-7 days on 24 processors
- Spot Model Simulation: 8-12 hours on a single processor
- A factor of $\sim 10^2$ speedup
- Simulations run on AMCL
Part 2: Parallelizing the Spot Model
C++ codes

- Split into regions, each storing particles within it

```cpp
class container {
    void import();
    void put(int n, vec &v);
    void dump();
    void regioncount();
    int count(vec &p, float r);
    ...
}
```
## Important Routines

### Spot Motion

```
void spot(vec &p, vec &v, float r);
p: position
v: displacement
r: spot radius
```

### Relaxation

```
void relax(vec &p, float r, float s, float force, float damp, int steps);
p: position
r: inner relaxation radius
s: outer relaxation
force: particle repulsive force
damp: particle velocity damping
steps: relaxation steps
```
Possible for parallel computing

- **Serial**: the elastic relaxation step is the computational bottleneck since it requires analyzing all pairs of neighboring particles within a small volume.
- In a *parallel* version, ideally we can distribute this computational load across many processors.
- Since each relaxation event occurs in a local area, we can pass out different relaxation jobs to different processors.
- Serial code written in C++ --- Use MPI for parallel computing
Master/Slave

- entire state of the system (particle positions and spot positions) held on the master node
- The master node sequentially passes out jobs to the slave nodes for computation and receive them back.

Rycroft 2006
Master/Slave

- Timing results: computed 60 frames of snapshots and calculated the average time per frame.
- Run on AMCL

<table>
<thead>
<tr>
<th># of slaves</th>
<th>Time per frame (s)</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Serial)</td>
<td>289</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>241</td>
<td>1.199</td>
<td>59.96%</td>
</tr>
<tr>
<td>3</td>
<td>414</td>
<td>0.698</td>
<td>17.45%</td>
</tr>
<tr>
<td>5</td>
<td>512</td>
<td>0.564</td>
<td>9.41%</td>
</tr>
<tr>
<td>7</td>
<td>551</td>
<td>0.524</td>
<td>6.56%</td>
</tr>
</tbody>
</table>
Master/Slave

- Problems:
  - too much stress is placed on the master node
  - very poor scalability with the number of nodes, as the slaves often stand idle waiting for the master node to pass jobs to them
Distributed Algorithm

- Container is divided up between the slaves, with each slave holding the particles in that section of the container.
- A master node holds the position of the spots and computes their motion. When a spot moves, the master node tells the corresponding slave node to carry out a spot displacement of the particles within it.
- Only the position and displacement carried by the spot need to be transmitted to the slave.
- Drawback:
  - A spot’s region of influence may overlap with areas managed by other slaves.
  - Each slave must transmit particles to the slave carrying out the computation, and then receive back the displaced particles. (Communication between slaves is required)
Distributed Algorithm

- Timing results: (implemented and run on SiCortex)

<table>
<thead>
<tr>
<th># of slaves</th>
<th>Processor Grid</th>
<th>Time per frame (s)</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Serial)</td>
<td>1x1x1</td>
<td>1256</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1x1x2</td>
<td>821</td>
<td>1.529</td>
<td>50.99%</td>
</tr>
<tr>
<td>3</td>
<td>1x1x3</td>
<td>674</td>
<td>1.864</td>
<td>46.59%</td>
</tr>
<tr>
<td>4</td>
<td>1x1x4</td>
<td>569</td>
<td>2.207</td>
<td>44.15%</td>
</tr>
<tr>
<td>5</td>
<td>1x1x5</td>
<td>515</td>
<td>2.439</td>
<td>40.65%</td>
</tr>
<tr>
<td>6</td>
<td>1x1x6</td>
<td>476</td>
<td>2.639</td>
<td>37.70%</td>
</tr>
<tr>
<td>7</td>
<td>1x1x7</td>
<td>446</td>
<td>2.816</td>
<td>35.20%</td>
</tr>
<tr>
<td>8</td>
<td>1x1x8</td>
<td>425</td>
<td>2.955</td>
<td>32.84%</td>
</tr>
<tr>
<td>9</td>
<td>1x1x9</td>
<td>406</td>
<td>3.094</td>
<td>30.94%</td>
</tr>
<tr>
<td>10</td>
<td>1x1x10</td>
<td>387</td>
<td>3.245</td>
<td>29.50%</td>
</tr>
</tbody>
</table>
Distributed Algorithm

- Much better speedup compared with master/slave method, but still not optimal
- Bottleneck: Overlapping Spot Motion
  - One slave needs to transfer its particles to another slave, then wait for the computation and receives back particles that are in the region it controls
A Faster Distributed Algorithm

- **Motivation:** The elastic relaxation step can “magically” fix a lot of the unphysical packings, even if we do not apply relaxation every spot step.

![Comparison of g(r) for different relaxation ratios](chart.png)
A Faster Distributed Algorithm

Comparison of $g(t)$ for different relaxation ratios

- $rr = 1$
- $rr = 100$
- $rr = 1000$
- $rr = 0$ (no relaxation)
A Faster Distributed Algorithm

- For overlapping spot motion, both slaves responsible for the region of the spot influence carry out spot computation independently, and exchange particles that are out of range if necessary.
- May not be 100% accurate, but significantly reduce waiting time and size of messages being exchanged between slaves.
A Faster Distributed Algorithm

- Timing results: (implemented and run on SiCortex)

<table>
<thead>
<tr>
<th># of slaves</th>
<th>Processor Grid</th>
<th>Time per frame (s)</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Serial)</td>
<td>1x1x1</td>
<td>1256</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1x1x2</td>
<td>687</td>
<td>1.827</td>
<td>60.91%</td>
</tr>
<tr>
<td>3</td>
<td>1x1x3</td>
<td>458</td>
<td>2.745</td>
<td>68.63%</td>
</tr>
<tr>
<td>4</td>
<td>1x1x4</td>
<td>334</td>
<td>3.757</td>
<td>75.13%</td>
</tr>
<tr>
<td>5</td>
<td>1x1x5</td>
<td>254</td>
<td>4.950</td>
<td>82.50%</td>
</tr>
<tr>
<td>6</td>
<td>1x1x6</td>
<td>207</td>
<td>6.054</td>
<td>86.48%</td>
</tr>
<tr>
<td>7</td>
<td>1x1x7</td>
<td>176</td>
<td>7.134</td>
<td>89.18%</td>
</tr>
<tr>
<td>8</td>
<td>1x1x8</td>
<td>151</td>
<td>8.319</td>
<td>92.44%</td>
</tr>
<tr>
<td>9</td>
<td>1x1x9</td>
<td>132</td>
<td>9.502</td>
<td>95.02%</td>
</tr>
<tr>
<td>10</td>
<td>1x1x10</td>
<td>116</td>
<td>10.86</td>
<td>98.75%</td>
</tr>
</tbody>
</table>
A Faster Distributed Algorithm

- Significant speedups and very good scalability with number of slaves
- Problems with this approach occur near the boundaries of regions owned by each slave. Larger errors with increasing number of processors since the container is divided into more regions.
Conclusion

- Master/slave method didn’t do so well
- Distributed Algorithm gave satisfactory results
- Significant speedup by Faster Distributed Algorithm, but balance between accuracy and speed
- Possible future work considering other algorithms