DSMC Algorithms at the Petascale and Beyond

SAND2013-9076C

Steve Plimpton & Michael Gallis
Sandia National Laboratories

DSMC13 Conference - Santa Fe, NM
Oct 2013

Developing a new DSMC code

SPARTA $=$
Stochastic PArallel Rarefied-gas Time-accurate Analyzer

Developing a new DSMC code

SPARTA =
Stochastic PArallel Rarefied-gas Time-accurate Analyzer

General features:

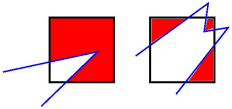
- 2d or 3d, serial or parallel
- Cartesian, hierarchical grid
    - oct-tree (up to 16 levels in 64-bit cell ID)
    - multilevel, general NxMxL instead of 2x2x2

Developing a new DSMC code

SPARTA =
Stochastic PArallel Rarefied-gas Time-accurate Analyzer

General features:

- 2d or 3d, serial or parallel
- Cartesian, hierarchical grid
  - oct-tree (up to 16 levels in 64-bit cell ID)
  - multilevel, general NxMxL instead of 2x2x2
- Triangulated surfaces cut/split the grid cells
  - 3d via Schwartzentruber algorithm
  - *Zhang & Schwartzentruber, Comp & Fluids, 69, 122 (2012)*
  - 2d via Weiler/Atherton algorithm
  - formulated so can use as kernel in 3d algorithm
- C++, but really object-oriented C
  - designed to be easy to extend
  - new collision/chemistry models, boundary conditions, etc



Petascale and next-generation machines

- 100K to **millions of nodes**
- Parallelism **within node**:
  - multi-core: 16 and growing
  - many-core: Intel Xeon Phi, 240 threads, vector len = 8
  - GPUs: NVIDIA/AMD, ∼1000 warps, vector len = 32
- Examples:
  - ORNL Titan: 18K nodes, 16 cores/node + GPU
  - ANL BG/Q: 48K nodes, 16 cores/node + 4 MPI tasks/core

Petascale and next-generation machines

- 100K to **millions of nodes**
- Parallelism **within node**:
  - multi-core: 16 and growing
  - many-core: Intel Xeon Phi, 240 threads, vector len = 8
  - GPUs: NVIDIA/AMD, ~1000 warps, vector len = 32
- Examples:
  - ORNL Titan: 18K nodes, 16 cores/node + GPU
  - ANL BG/Q: 48K nodes, 16 cores/node + 4 MPI tasks/core

- Programming model: **MPI + X**
- Focus of this talk is on inter-node parallelism (MPI)
- Next talk is about “X”: DSMC on GPUs

SPARTA benchmarking

2 machines:

- chama = **Linux cluster** at Sandia, 400 Tflops (20K cores)
  - dual Intel SandyBridge = 16 cores/node
  - Infiniband interconnect
  - up to 1024 nodes = 16K cores
- mira = **BG/Q** at Argonne, 10 Pflops (768K cores)
  - custom interconnect
  - slower cores, less memory, 4 threads per core
  - up to 8192 nodes = 128K cores = 512K MPI tasks

SPARTA benchmarking

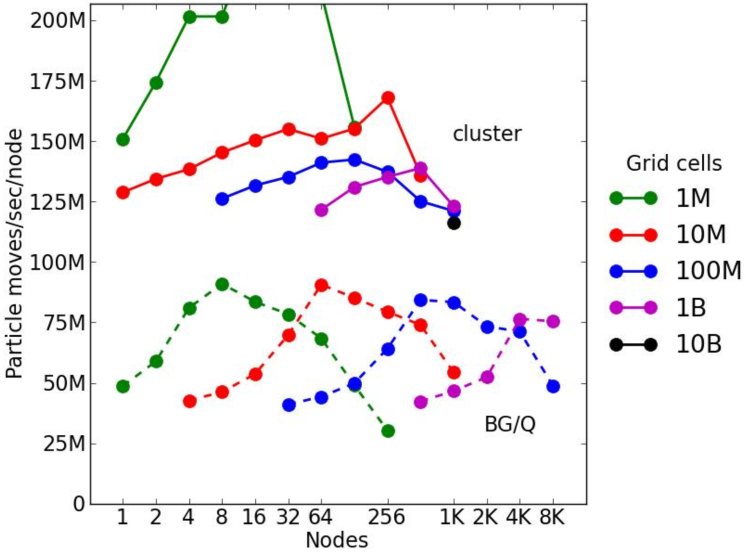
2 machines:

- chama = **Linux cluster** at Sandia, 400 Tflops (20K cores)
  - dual Intel SandyBridge = 16 cores/node
  - Infiniband interconnect
  - up to 1024 nodes = 16K cores
- mira = **BG/Q** at Argonne, 10 Pflops (768K cores)
  - custom interconnect
  - slower cores, less memory, 4 threads per core
  - up to 8192 nodes = 128K cores = 512K MPI tasks

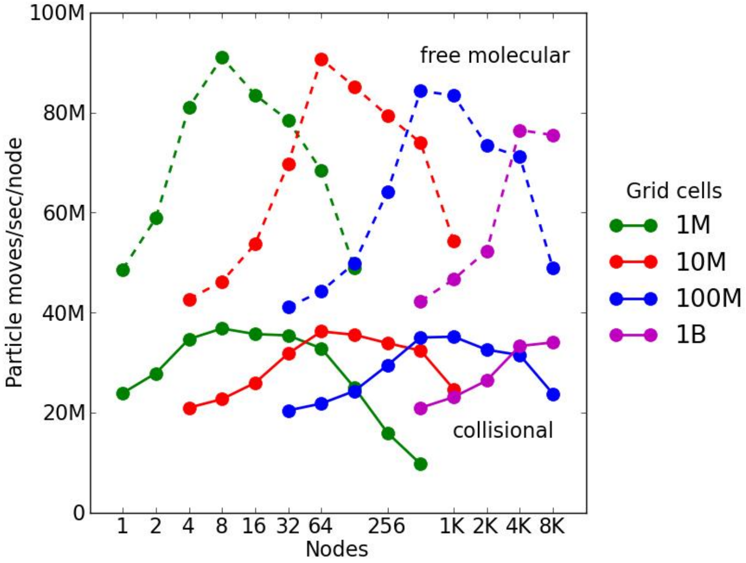
2 test cases:

- Free molecular flow
  - stress test for communication
  - 3d regular grid ⇒ 1M to 10B grid cells
  - 10 particles/cell ⇒ 10M to 100B particles
- Collisional flow
  - about 2x slower (sorting, collisions)
  - same grid cell & particle counts

Free molecular flow on two machines



Collisional flow on BG/Q

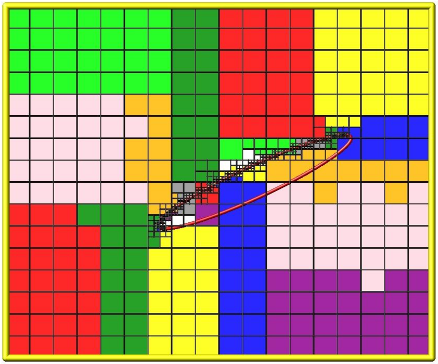


Performance issues to address for 100K-1M MPI tasks

- Load-balancing
- Efficient communication
- Problem setup and adaptive gridding
- Visualization

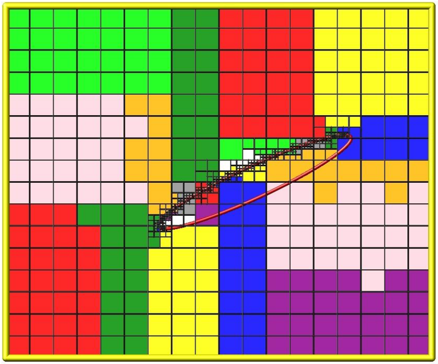
Load balancing

- Balance across procs, **static or dynamic**
- Granularity = **grid cell** with its particles
- Geometric method: recursive coordinate bisection (RCB)
- **Weighted** by cell count or particles or CPU (not yet)



Load balancing

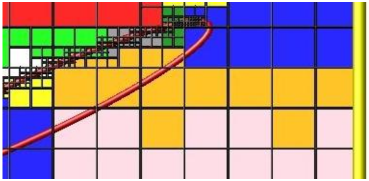
- Balance across procs, **static or dynamic**
- Granularity = **grid cell** with its particles
- Geometric method: recursive coordinate bisection (RCB)
- **Weighted** by cell count or particles or CPU (not yet)

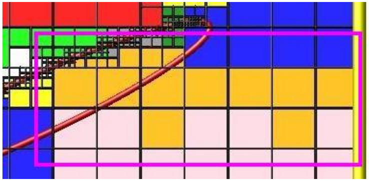


- RCB is fast
- Bigger cost is **data move**
- 1B cells on
  1024 BG/Q nodes
  worst case: move all cells
  balance time = 15 secs
  RCB = 2, move = 12,
  ghosts = 1

Efficient communication

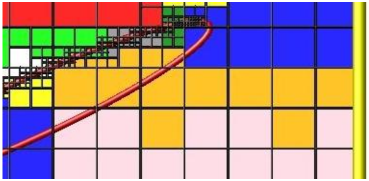
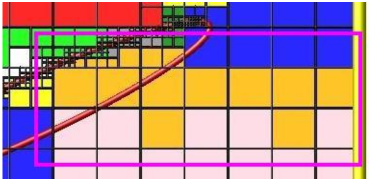
- One proc = compact clump of cells via load balancing
- Ghost region = nearby cells within **user-defined cutoff**
- Store surface info for ghost cells to complete move





Efficient communication

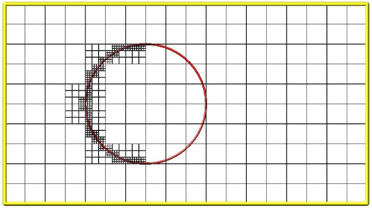
- One proc = compact clump of cells via load balancing
- Ghost region = nearby cells within **user-defined cutoff**
- Store surface info for ghost cells to complete move

- Efficiently distributes grid info across procs
- With sufficient cutoff, only **one communication per step**
- Multiple passes if needed (or can bound particle move)
- Communication with **modest count of neighbor procs**

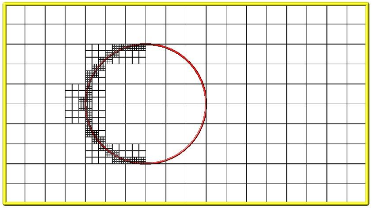
Problem setup and adaptive gridding

- Create/adapt grid **in situ**, rather than pre-process & read in
- Examples: refine around surface to user-specified resolution,
  adapt grid based on flow properties
- Algorithms should be efficient if only require **local comm**



Problem setup and adaptive gridding

- Create/adapt grid **in situ**, rather than pre-process & read in
- Examples: refine around surface to user-specified resolution,
  adapt grid based on flow properties
- Algorithms should be efficient if only require **local comm**



- Another setup task: **label cells** as outside/inside flow
- Simple if pre-processing, in situ easier for large problems
- **Idea**: label cells next to surf, **paint** outward, communicate
- Fast in practice, iteration count scales as $P^{1/3}$

On-the-fly visualization

- Not a replacement for interactive viz, but ...
- Quite useful for **debugging** & quick analysis
- At end of simulation (or during) $\Rightarrow$ **instant movie**

On-the-fly visualization

- Not a replacement for interactive viz, but ...
- Quite useful for **debugging** & quick analysis
- At end of simulation (or during) $\Rightarrow$ **instant movie**

- Render a **JPG snapshot** every N timesteps:
  - each proc starts with blank image (1024x1024)
  - proc draws its cells/surfs/particles with **depth-per-pixel**
  - merge pairs of images, keep the pixel in front, recurse
  - draw is parallel, merge is logarithmic (like MPI Allreduce)

On-the-fly visualization

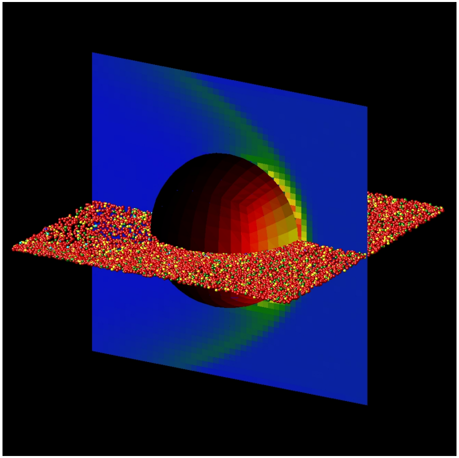
- Not a replacement for interactive viz, but ...
- Quite useful for **debugging** & quick analysis
- At end of simulation (or during) $\Rightarrow$ **instant movie**

- Render a **JPG snapshot** every N timesteps:
  - each proc starts with blank image (1024x1024)
  - proc draws its cells/surfs/particles with **depth-per-pixel**
  - merge pairs of images, keep the pixel in front, recurse
  - draw is parallel, merge is logarithmic (like MPI Allreduce)

- Images are **ray-traced** quality

Simple example of on-the-fly viz

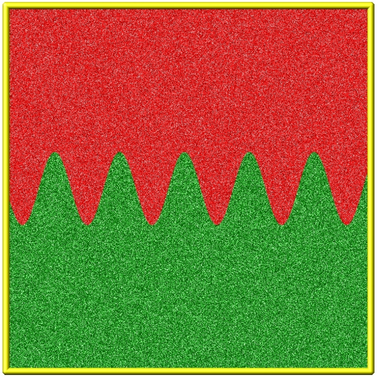
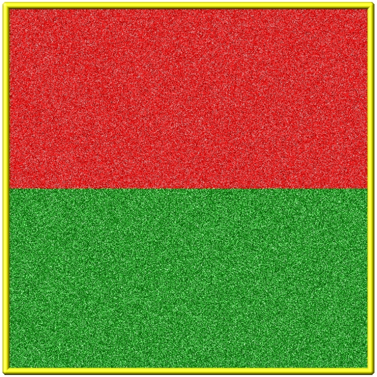
Particles + surface triangles + plane thru grid cells



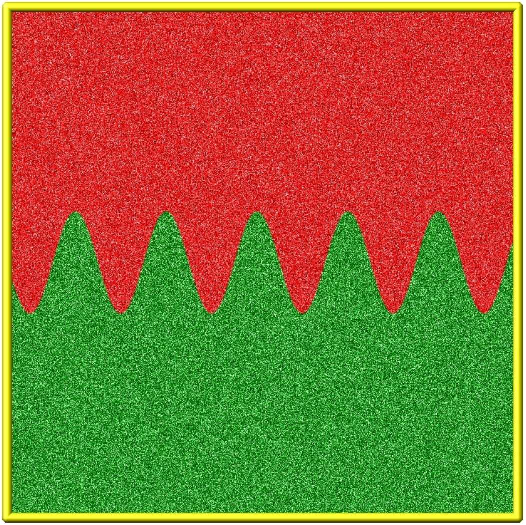
% convert image*jpg movie.gif ⇒ play in browser

Big example of on-the-fly viz

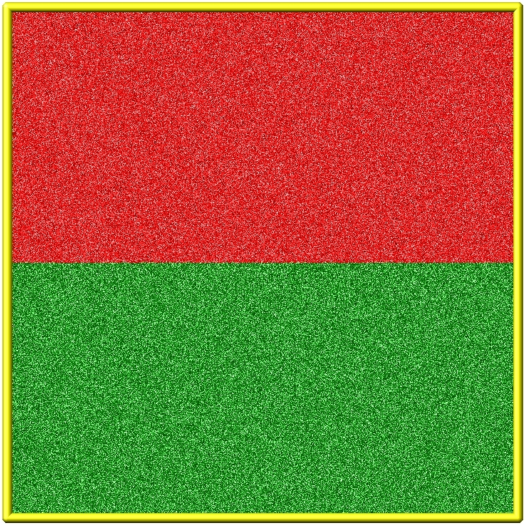
- **Rayleigh-Taylor instability** in 2d
- Two-fluid mixing under gravity, heavy over light
- 100M cells (10K x 10K), 1B particles, 10K steps, 1024 cores

Rayleigh-Taylor with rough surface



Rayleigh-Taylor with flat surface & pressure wave



Aiming for MPI+X via Kokkos

- Programming model in development at Sandia
    - hope to **minimize impact** of new chip designs on applications
    - Carter Edwards and Christian Trott
- Goal: write application kernels **only once**,
  run them efficiently on variety of hardware

Aiming for MPI+X via Kokkos

- Programming model in development at Sandia
  - hope to **minimize impact** of new chip designs on applications
  - Carter Edwards and Christian Trott
- Goal: write application kernels **only once**,
  run them efficiently on variety of hardware
- Two major components:
  1. Data access abstraction via **Kokkos arrays**
     - optimal layout & access pattern for each device
       GPU, Xeon Phi, etc
  2. **Parallel dispatch** of small chunks of work
     - auto-mapped onto back-end languages
       CUDA, OpenMP, etc

Aiming for MPI+X via Kokkos

- Programming model in development at Sandia
  - hope to **minimize impact** of new chip designs on applications
  - Carter Edwards and Christian Trott
- Goal: write application kernels **only once**,
  run them efficiently on variety of hardware
- Two major components:
  1. Data access abstraction via **Kokkos arrays**
     - optimal layout & access pattern for each device
       GPU, Xeon Phi, etc
  2. **Parallel dispatch** of small chunks of work
     - auto-mapped onto back-end languages
       CUDA, OpenMP, etc
- Key task for us is to write **DSMC kernels** so they:
  - operate at fine granularity
  - are thread-safe
  - use Kokkos-compatible data structures

Next steps for SPARTA

- **Issues** to address:
  - more efficient cache usage
  - adaptive gridding
  - validation and verification
- Planning for **open-source release** in a few months

Next steps for SPARTA

- **Issues** to address:
  - more efficient cache usage
  - adaptive gridding
  - validation and verification
- Planning for **open-source release** in a few months

- **Thanks!**
  - Nathan Fabian (Sandia), graphics wizard
  - Jeff Hammond (ANL/ALCF), help with BG/Q issues
  - Jay LeBeau (NASA Johnson)
  - Sandia management support