

# SPARTA: Building, Running, Performance

*Stan G. Moore*

Computational Multiscale  
Sandia National Laboratories  
Albuquerque, New Mexico, USA

[stamoor@sandia.gov](mailto:stamoor@sandia.gov)

Short Course

*Sparta: A Parallel, Flexible, Open-Source DSMC code*

September 24, 2023; Santa Fe, New Mexico USA



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. SAND2023-09457C



# About Me

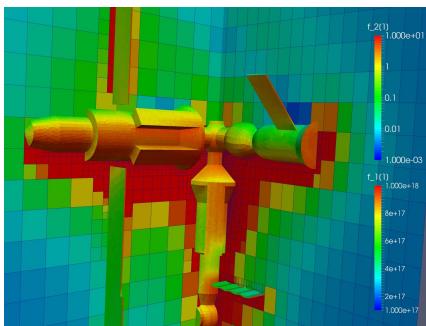
- Stan Moore
  - One of the SPARTA code developers at Sandia National Laboratories in Albuquerque, New Mexico
  - Been at Sandia for 11 years
  - Main developer of the KOKKOS package in SPARTA (runs on GPUs and multi-core CPUs)
  - PhD in Chemical Engineering, dissertation on molecular dynamics method development
  - Also work on LAMMPS code



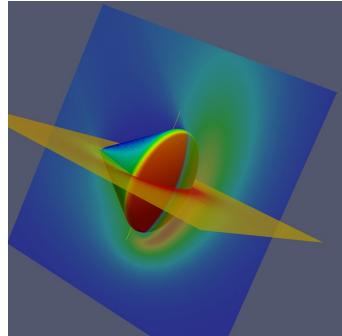
# SPARTA Intro

(Stochastic PArallel Rarefied-gas Time-accurate Analyzer)

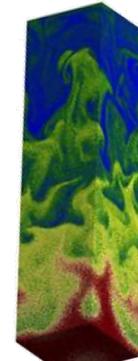
- Direct Simulation Monte Carlo (DSMC) code
- Core developers are Steve Plimpton, Michael Gallis, and Stan Moore (Sandia National Laboratories)
- Open-source, <http://sparta.github.io>
- Collaborators: ORNL, LANL, ANL, LBNL, NASA, ESA, academia



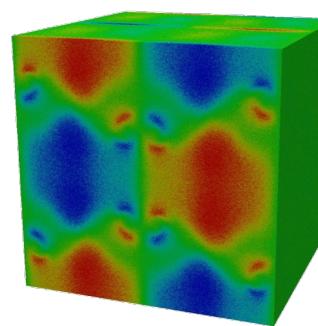
Spacecraft



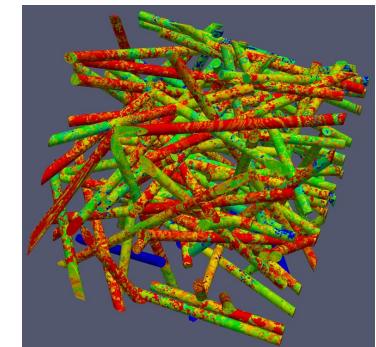
Re-entry



Instabilities



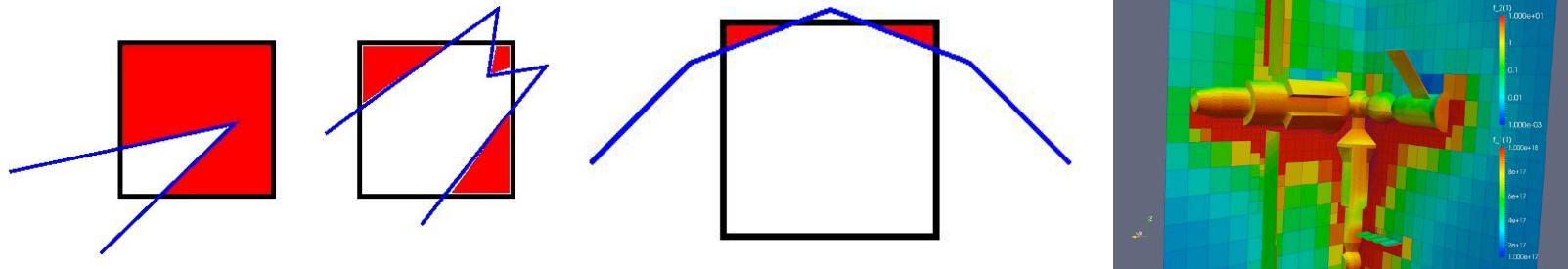
Turbulence



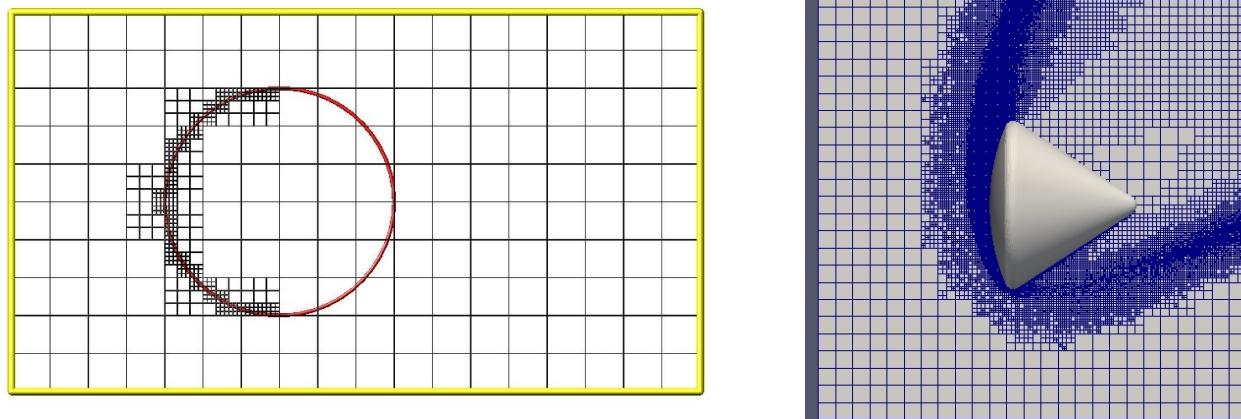
Porous Media

# SPARTA Features

- Structured grids with complex surfaces via cut and split cells



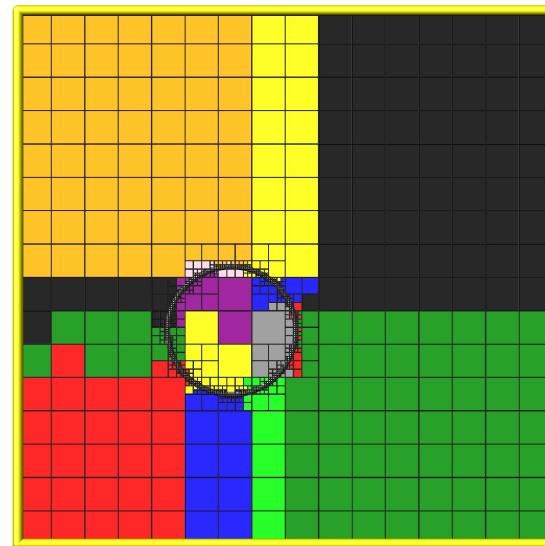
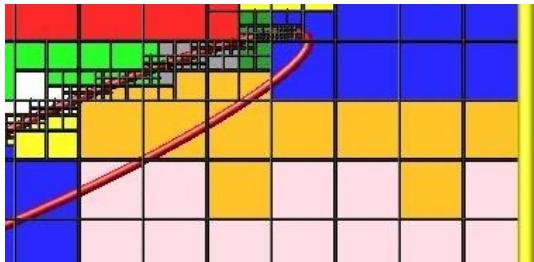
- Hierarchical grids with adaptive mesh refinement



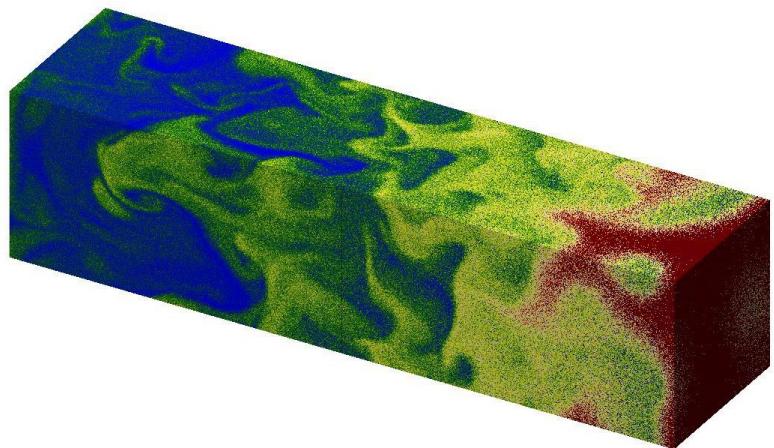
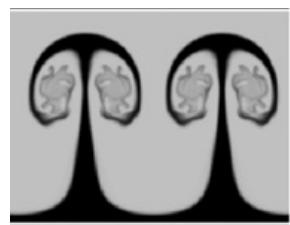
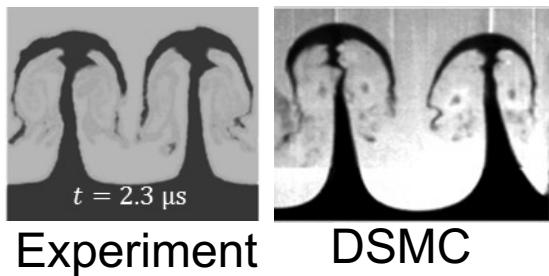
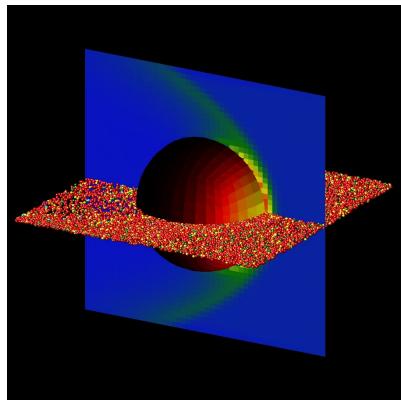
- MPI parallelism using highly scalable domain decomposition

# SPARTA Features (cont.)

- Load balancing (static and dynamic)



- In-Situ Visualization



- And more

# SPARTA Reference Paper

- <https://doi.org/10.1063/1.5108534>
- Describes SPARTA algorithms, code implementation, applications, and parallel performance of benchmarks

Physics of Fluids

ARTICLE

[scitation.org/journal/phf](https://scitation.org/journal/phf)

## Direct simulation Monte Carlo on petaflop supercomputers and beyond

F SCI

---

Cite as: Phys. Fluids 31, 086101 (2019); doi: [10.1063/1.5108534](https://doi.org/10.1063/1.5108534)  
Submitted: 30 April 2019 • Accepted: 2 July 2019 •  
Published Online: 1 August 2019

---

S. J. Plimpton,<sup>1,a)</sup> S. G. Moore,<sup>1</sup> A. Borner,<sup>2</sup> A. K. Stagg,<sup>1</sup>  T. P. Koehler,<sup>1</sup> J. R. Torczynski,<sup>1</sup>  and M. A. Gallis<sup>1,a)</sup> 

---

 View Online  
 Export Citation  
 CrossMark

# Compiling SPARTA

- [https://sparta.github.io/doc/Section\\_start.html#start\\_2](https://sparta.github.io/doc/Section_start.html#start_2)
- Need C++ compiler, MPI library
- Two build systems, Makefile:

```
cd src
```

```
make -j4 mpi # uses Makefile.mpi
```

- CMake:

```
mkdir build
```

```
cd build
```

```
cmake -C ../../cmake/presets/mpi.cmake \
```

```
..../cmake
```

# MPI STUBS Library

- Sometimes need to run in serial → 1 MPI rank (e.g. no MPI lib on your laptop)
- SPARTA always requires an MPI library, however can use MPI STUBS library bundled with SPARTA as a workaround
- MPI STUBS is automatically included in Makefile.serial:

```
cd src  
make -j4 serial
```

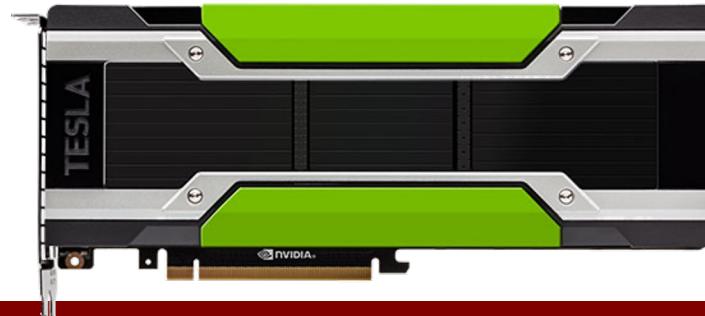
# Running SPARTA

- Run on 4 MPI ranks on CPU

```
cd bench  
mpiexec -np 4 -bind-to core \  
./src/spa_mpi -in in.collide
```

# HPC Hardware Trends

- Currently, 7 out of the top 10 supercomputers use GPUs (NVIDIA or AMD), according to the June 2023 Top500 List (<https://www.top500.org>)
- #1 ORNL Frontier uses AMD MI250X GPUs: first true exascale computer with an HPL score of 1.1 Exaflop/s
- Future exascale supercomputers will also have accelerators: ANL Aurora—Intel, NNSA El Capitan—AMD
- Special code (beyond regular C++ and MPI in SPARTA) is required to run well on NVIDIA, AMD, and Intel GPUs (e.g. CUDA, HIP, SYCL)
- Hardware and corresponding programming languages are ever changing, how to keep SPARTA up to date?



# Kokkos Performance Portability Library



- Kokkos is an abstraction layer between programmer and next-generation platforms
- Allows the same C++ code to run on multiple hardware (Intel CPU, NVIDIA GPU, Intel GPU, AMD GPU, etc.)
- Kokkos consists of two main parts:
  1. Parallel dispatch—threaded kernels are launched and mapped onto backend languages such as CUDA or OpenMP
  2. Kokkos views—polymorphic memory layouts that can be optimized for a specific hardware
- Used on top of existing MPI parallelization (MPI + X)
- Used by many codes, open-source, can be downloaded at  
<https://github.com/kokkos/kokkos>
- Goal: future-proof the code to allow it to run on future hardware without total re-write (i.e. Kokkos team develops new backends for new hardware, minimal changes needed in SPARTA)

# SPARTA KOKKOS Package

- Kokkos library abstractions are implemented in SPARTA as an optional add-on package called *KOKKOS*
- Algorithms in SPARTA ported to use Kokkos include:
  - Particle move, sort, collide, chemical reactions, emission from cell faces
  - Surface collisions (diffuse, specular, etc.)
  - Several diagnostics (temperature computation, averaging of grid quantities, etc.)
- Complex surfaces, static and dynamic load balancing and grid adaptation are all compatible with the Kokkos version (but some run on host CPU)
- Recently added Kokkos support for ambipolar approximation and surface reactions

# Kokkos Backends

- Multi-core CPUs → Kokkos OpenMP backend
- NVIDIA GPUs → Kokkos CUDA backend
- AMD GPUs → Kokkos HIP backend
- Intel GPUs → Kokkos SYCL backend
- Kokkos OpenMPTarget backend can also be used for NVIDIA, AMD, and Intel GPUs, but may be less performant than other backends

# Compiling SPARTA with Kokkos

- [https://sparta.github.io/doc/Section\\_accelerate.html#acc\\_3](https://sparta.github.io/doc/Section_accelerate.html#acc_3)
- **Need C++17 compiler** (e.g. GCC 8.2.0 or later)
- Two build systems, Makefile:

```
cd src  
make yes-kokkos  
make -j64 kokkos_cuda
```

- CMake:  

```
mkdir build  
cd build  
cmake -C ../cmake/presets/kokkos_cuda.cmake \  
      ..../cmake  
make -j64
```
- May need to change default Kokkos “arch” setting to match your machine

# Running SPARTA with Kokkos

- Run on 4 MPI ranks + 4 OpenMP threads/rank

```
scd bench
```

```
mpiexec -np 4 ./src/spa_kokkos_omp -in  
in.collide -k on t 4 -sf kk
```

- Run on 4 MPI ranks + 4 GPUs per node (1 GPU/rank)

```
mpiexec -np 4 ./src/spa_kokkos_cuda -in  
in.collide -k on g 4 -sf kk
```

- “sf kk” is the suffix command, see  
<https://sparta.github.io/doc/suffix.html>, use on command line,  
not recommended to edit individual styles in input script
- Normally only use 1 MPI rank per GPU with KOKKOS package

# Processor and Thread Affinity

- Use mpirun command-line arguments (e.g. `--bind-to core` or `-bind-to socket`) to control how MPI tasks and threads are assigned to nodes and cores
- Also set OpenMP variables such as `OMP_PROC_BIND` and `OMP_PLACES`
- Pay attention to NUMA bindings between tasks, cores, and GPUs. For example, for a dual-socket system, MPI tasks driving GPUs should be on the same socket as the GPU

# How to Tell if Running with KOKKOS Package?

- Check top of screen output or log file

SPARTA (13 Apr 2023)

```
KOKKOS mode is enabled (../kokkos.cpp:40)
    requested 4 GPU(s) per node
    requested 1 thread(s) per MPI task
```

Running on 4 MPI task(s)

# How to Tell if Running on GPUs?

- `nvidia-smi` (or `rocm-smi` for AMD GPUs), need to run on compute node)

NVIDIA-SMI 510.47.03			Driver Version: 510.47.03		CUDA Version: 11.6		
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr.	ECC
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	MIG M.
0	Tesla V100-SXM2...	On	00000004:04:00.0	Off	73%	0	Default
N/A	32C	P0	134W / 300W	1285MiB / 16384MiB			N/A
1	Tesla V100-SXM2...	On	00000004:05:00.0	Off	73%	0	Default
N/A	30C	P0	140W / 300W	1285MiB / 16384MiB			N/A
2	Tesla V100-SXM2...	On	00000035:03:00.0	Off	73%	0	Default
N/A	33C	P0	129W / 300W	1285MiB / 16384MiB			N/A
3	Tesla V100-SXM2...	On	00000035:04:00.0	Off	73%	0	Default
N/A	30C	P0	124W / 300W	1285MiB / 16384MiB			N/A

# KOKKOS Package Options

- See <https://sparta.github.io/doc/package.html>
- Defaults should be mostly optimal
- **Be sure to use GPU-aware MPI** (if applicable), otherwise performance on GPUs will suffer
- Different MPI libraries have different ways to enable GPU-aware MPI (beyond scope of this talk)

# Challenges of Chemical Reactions

- Chemical reactions can increase the number of particles in the simulation stochastically
- Newly created particles immediately participate in the parallel region, which affects the simulation outcome
- Not possible to resize Kokkos GPU array inside a parallel region, so two workarounds implemented:
  1. Over-allocate particle storage by some amount. If this space is still not sufficient, error out and restart the simulation using a larger value for over-allocation: `-pk kokkos react/extra 1.1` (default)
  2. Make backup copies of the Kokkos Views. If space is exceeded, restore the Kokkos Views from backup, increase their size, and restart the parallel region from the beginning. Guaranteed to eventually succeed, but increases 2x particle memory + overhead from making a backup copy of the Views: `-pk kokkos react/retry yes`
- Option #1 is faster but not convenient when the simulation dies and must be restarted, option #2 is slower but guaranteed to succeed

# Global Options

- See <https://sparta.github.io/doc/global.html>
- Global options define the global properties of the simulation
- Global options go in the input script (not on command line)
- Examples:

```
global particle/reorder 10
```

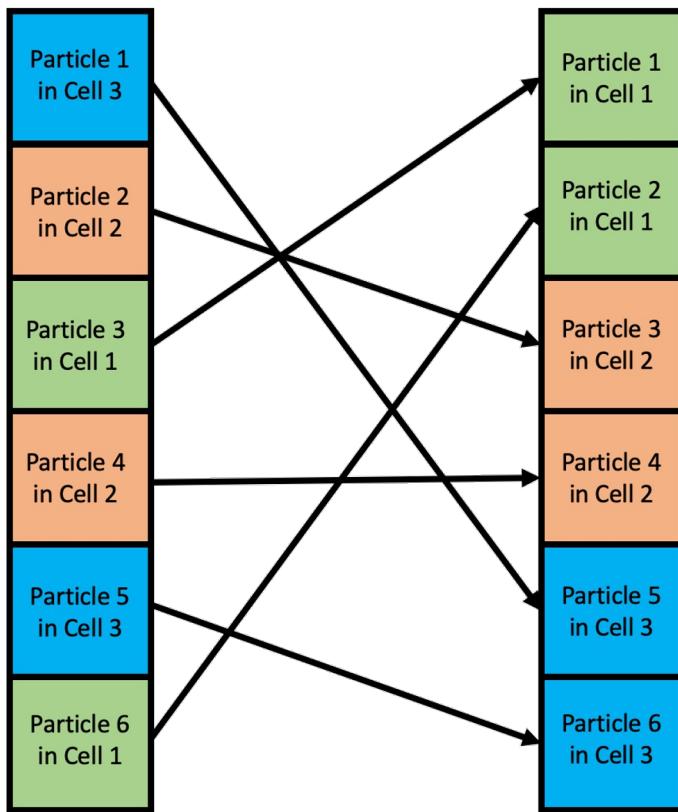
```
global optmove yes
```

```
global mem/limit 1024
```

# global particle/reorder

- Particle array starts out sorted by grid cell, but over time the particles in the array are randomized *wrt* cells
- Periodically reordering particle array by grid cell improves data locality and cache access patterns (can give speedup)
- However, sorting has overhead, so there is an optimal frequency (i.e. less than every timestep)
- **Very important for GPUs** (e.g. need to reorder every ~10 timesteps)
- Also important for CPUs to prevent performance degradation over time, but typically reorder less frequently (e.g. every 1000 timesteps)
- Currently only works with Kokkos version, but planning to support regular version in the future

# global particle/reorder

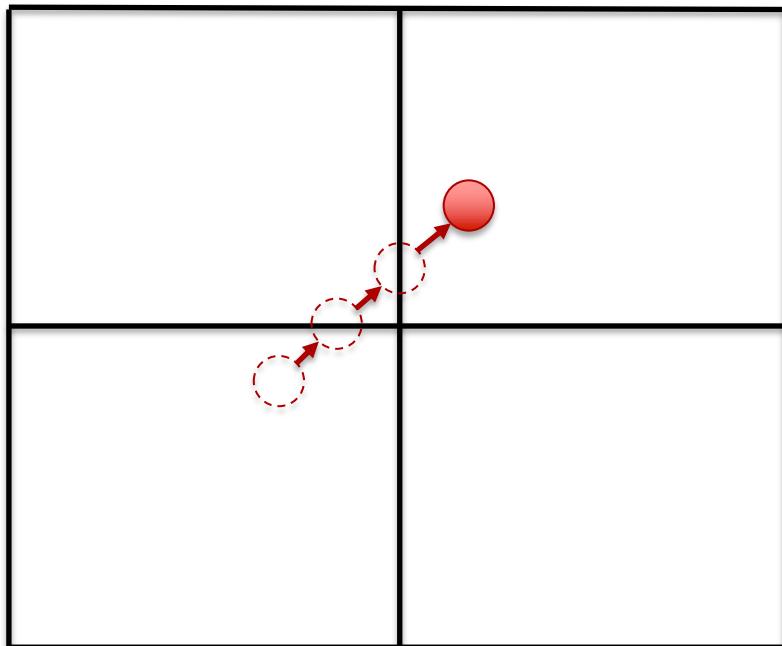


- For Kokkos version reorder is done out of place (2x particle memory overhead)

# global optmove

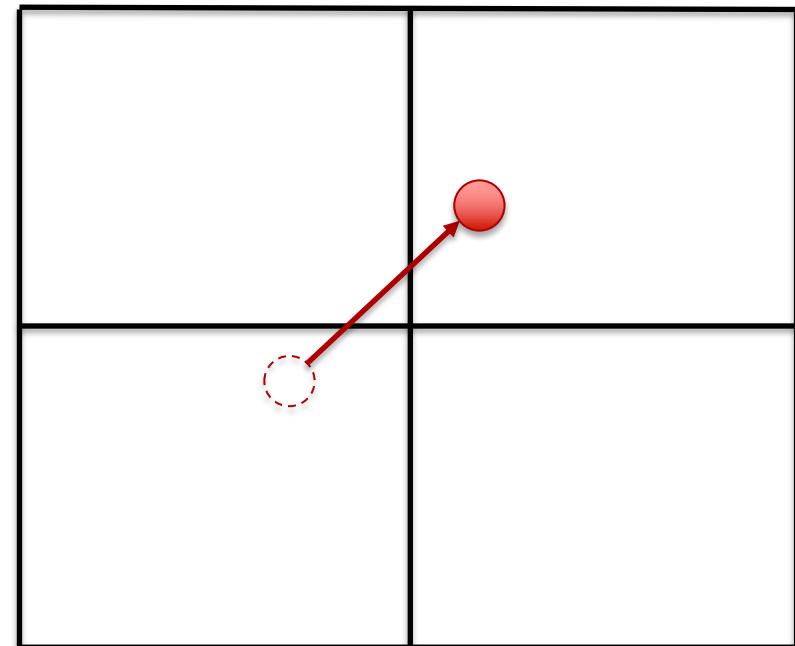
## Traditional move:

- Finds all intermediate grid cell crossings
- Works for non-uniform (adapted) grids and embedded surfaces



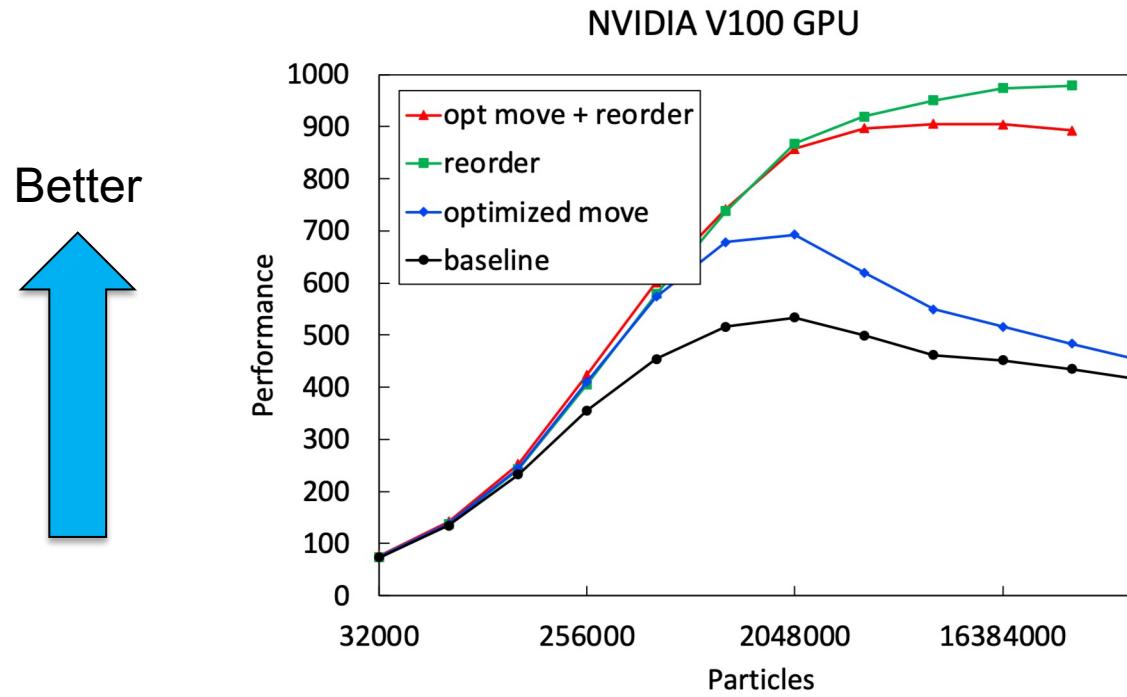
## New optimized move:

- Moves particle to final position in a single step
- Cannot handle non-uniform (adapted) grids or embedded surfaces



# Performance of Global Options

- For standard “collide” benchmark on an NVIDIA V100 GPU, optimized move helps, but particle reorder without optimized move is best
- Optimized move helps more on an AMD MI250X GPU than on NVIDIA V100 (not shown in figure)
- Performance is measured in millions of particle-timesteps/s



# global mem/limit

- Two purposes: reduce memory overhead of temporary buffers and work around 2 billion element limit for MPI operations
- Uses multiple passes several operations: load balancing, reordering of particles, and restart file read/write
- global mem/limit grid: try to make particle memory same size as grid cell memory
- global mem/limit 1024: work around 2 billion element limitation in MPI (only applicable for huge simulations)

# Load Imbalance

- Check timing breakdown in SPARTA screen output and log file

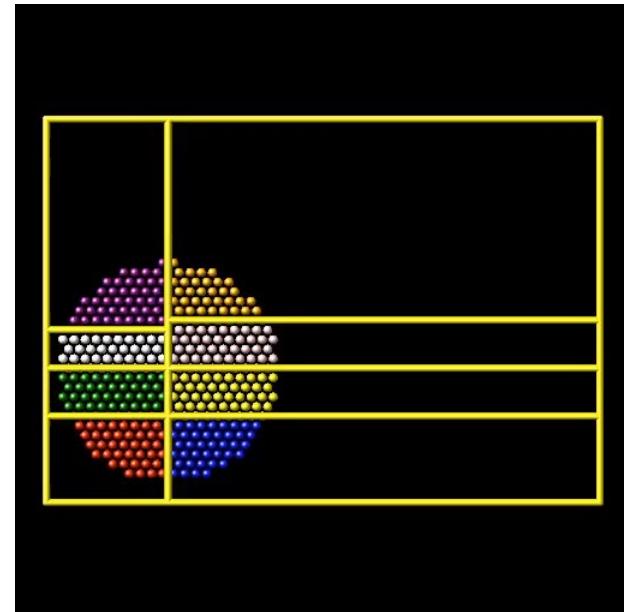
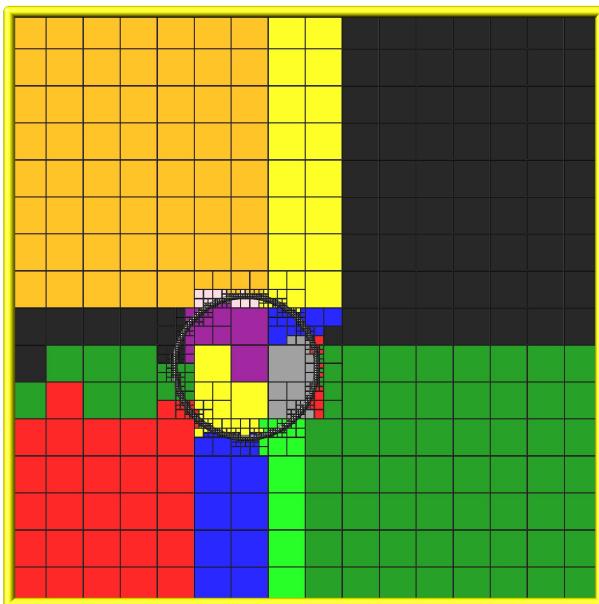
MPI task timing breakdown:

Section	min time	avg time	max time	%varavg	%total
<hr/>					
Move	15.37	203.08	1767.5	1799.9	10.16
Coll	0.054141	0.093716	0.16875	9.5	0.00
Sort	10.283	108.69	314.85	948.7	5.44
Comm	2.8089	4.0201	6.2701	38.9	0.20
Modify	3.9992	6.078	7.2366	37.1	0.30
Output	0.0038326	0.0055809	0.0084774	1.9	0.00
Other		1678			83.90

# Load Balancing

- Normally use RCB (recursive bisectioning) style

```
fix 1 balance 1000 1.1 rcb cell
```
- Can balance by cells, particles, or time, see  
[https://sparta.github.io/doc/fix\\_balance.html](https://sparta.github.io/doc/fix_balance.html)



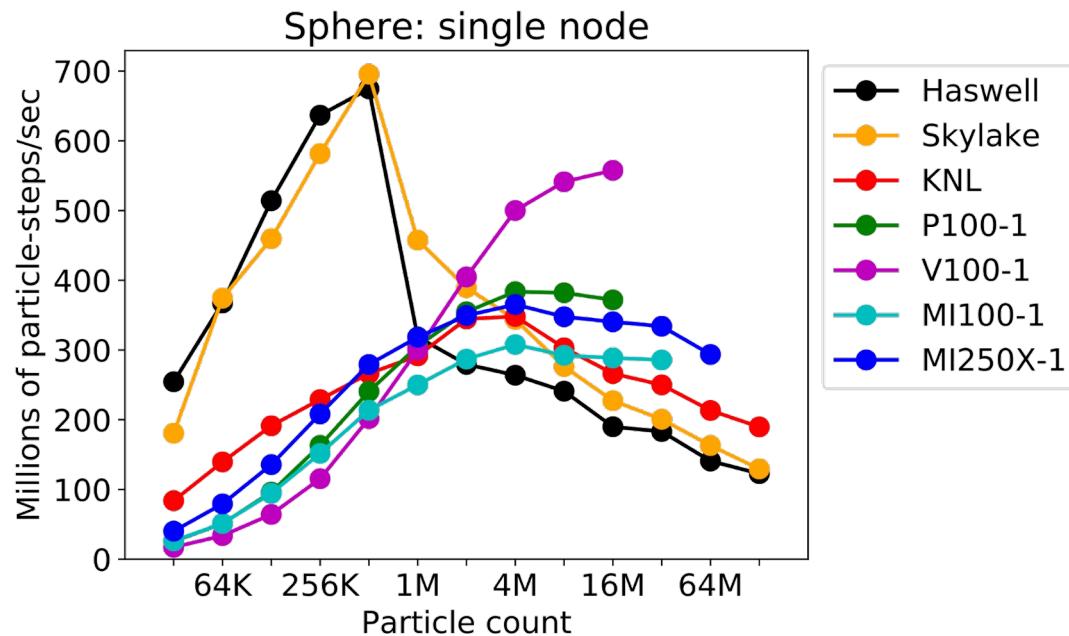
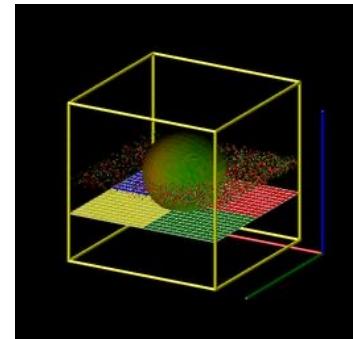
- (Movie from LAMMPS, but similar for SPARTA)

# SPARTA Benchmarking Website

- <https://sparta.github.io/bench.html>
- Performance plots for several benchmarks: free, collide, and sphere
- Kokkos and non-Kokkos results
- Single node, strong scaling, and weak scaling results
- Also lists exact MPI run command used for every run
- A little outdated: need to update for latest hardware and include global options (particle/reorder and optmove)

# Example of Kokkos Performance

- Benchmark: particles flowing around a sphere, best performance using either Kokkos or MPI-only
- No global optmove or particle/reorder options included
- 1 CPU node or 1 GPU (single die for AMD MI250X)
- Large cache effect for small problem sizes on CPUs, while GPUs need large number of particles to saturate threads
- AMD GPU results are preliminary, profiling/tuning ongoing



# Towards Exascale

- Preparing SPARTA on exascale supercomputers (OLCF Frontier, ALCF Aurora, NNSA's El Capitan)
- SPARTA successfully compiles on Frontier (Makefile.kokkos\_hip) and pre-Aurora Sunspot testbed (Makefile.kokkos\_sycl), as well as other large supercomputers
- Also preparing to run SPARTA on NNSA's Crossroads machine (Intel Sapphire Rapids CPUs)
- Collaborating with vendors such as Intel, NVIDIA, and AMD



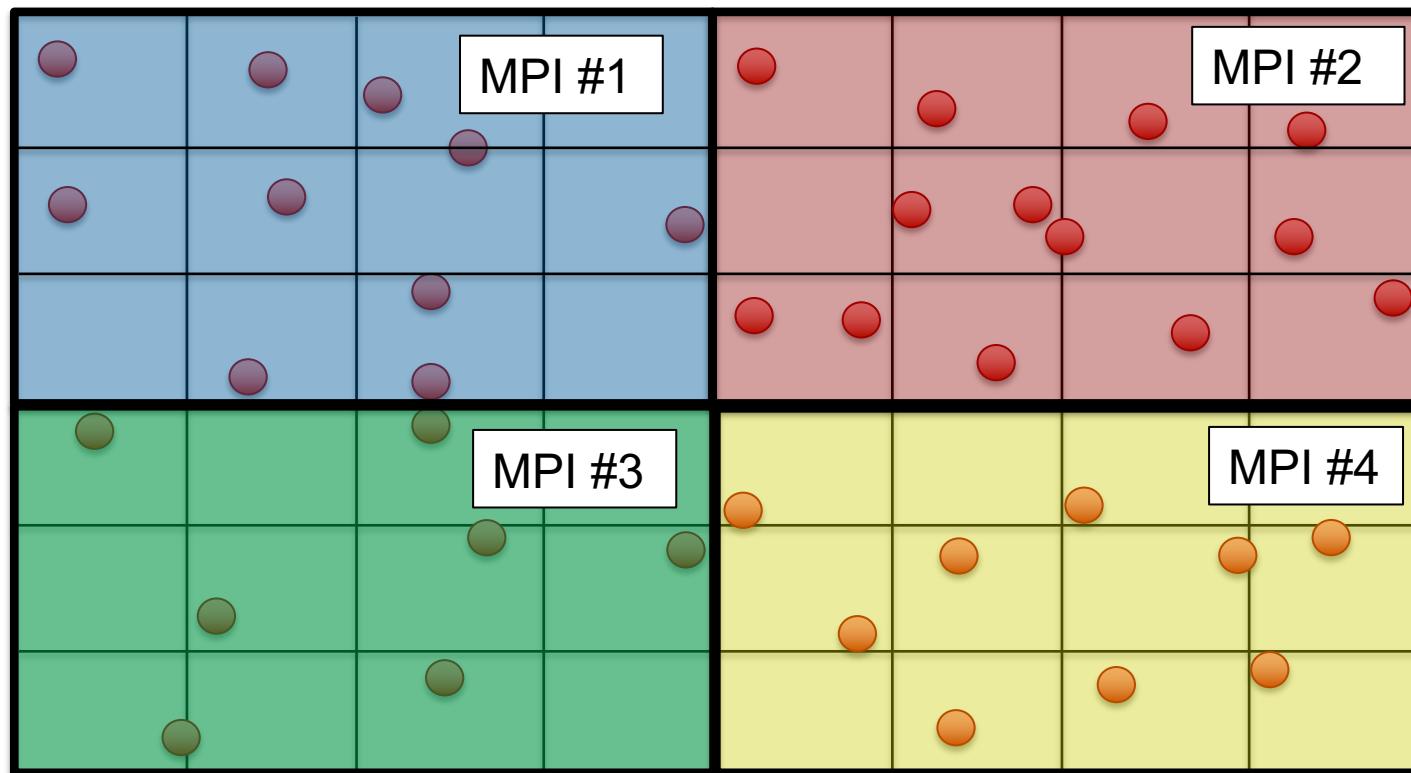
# Basics of Kokkos Programming



- Kokkos consists of two main parts:
  1. Parallel dispatch—threaded kernels are launched and mapped onto backend languages such as CUDA or OpenMP
  2. Kokkos views—polymorphic memory layouts that can be optimized for a specific hardware
- Typically thread over loops of particles or grid cells (need significant work to keep GPU busy)
- All particle or grid arrays inside threaded parallel loops must use Kokkos views, class variables on stack are accessible
- Start with closest related style already in SPARTA and use as a template

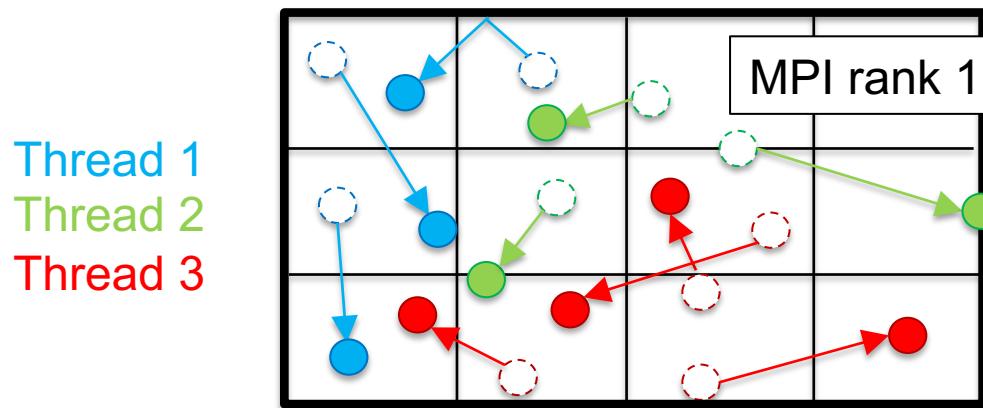
# MPI Parallelization Approach

- Domain decomposition: each processor owns a portion of the simulation domain and particles therein



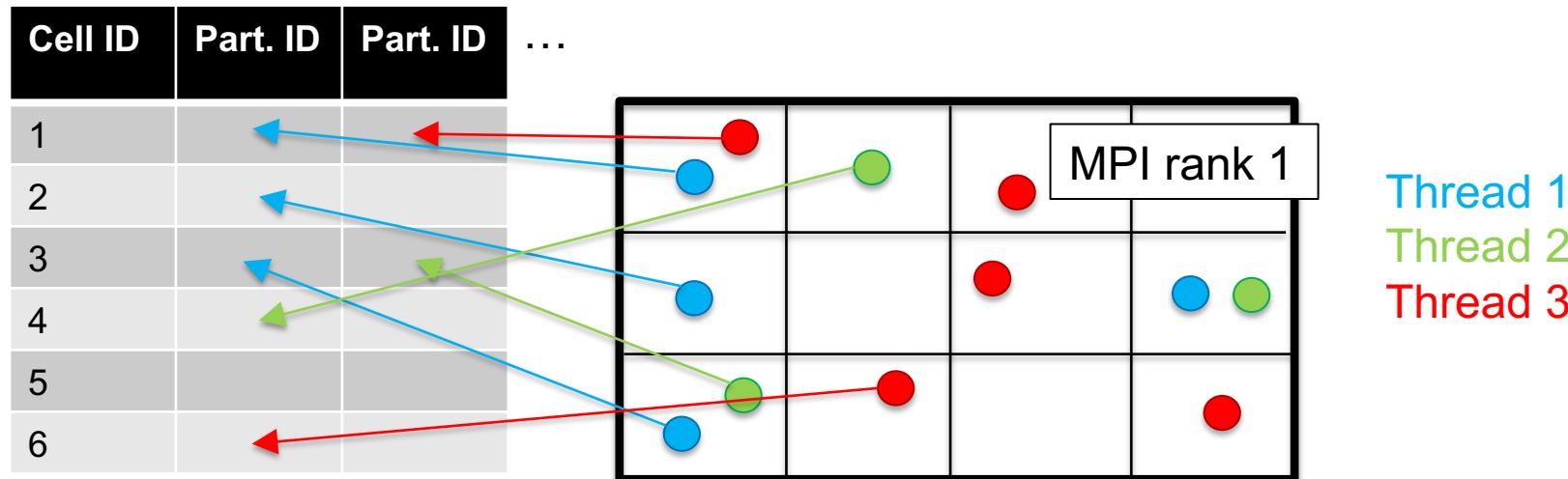
# Kokkos Threaded Move

- One thread pushes particles for a timestep or micro-iteration
- All intermediate grid crossings are found
- Performance hurt by branching, especially with surfaces. If applicable `global optmove` can reduce branching
- Statistical accumulators (i.e. number of moves, number of surface collisions, etc.) use either a parallel reduction or an atomic reduction on a global variable (controlled by Kokkos package option)



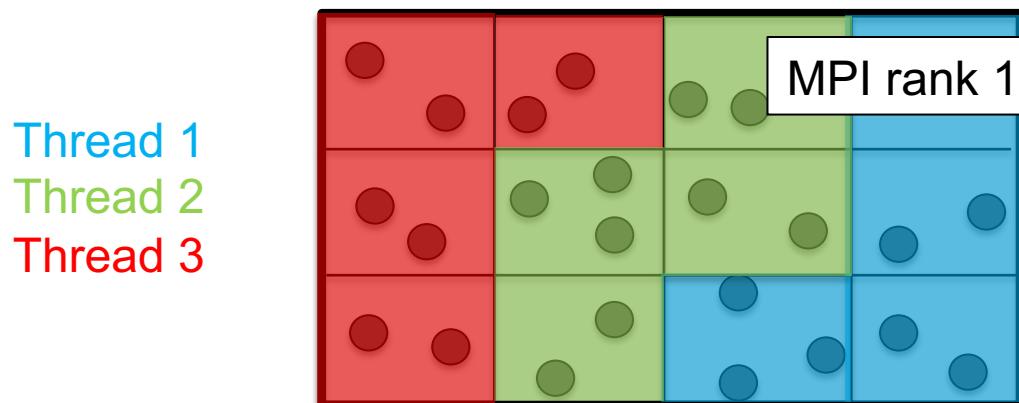
# Kokkos Threaded Sort

- Threads loop over particles to sort by grid cell
- 2D array of grid cells vs particle IDs is created, along with 1D array of counts of particles in each cell
- Requires thread atomics to avoid write conflicts
- If 2D array is too small, increase second dimension, realloc, and try again
- Periodically reordering particle list by cell id can improve performance (e.g. global particle/reorder 10)



# Kokkos Threaded Collide

- Each thread processes all the collisions in a grid cell
- Nearest neighbor algorithm also supported



# Execution and Memory Spaces

- With GPUs, *Host* execution space = CPU backed (serial or OpenMP), *Device* = GPU
- GPUs typically have high bandwidth memory that is not accessible from CPU: pointers to CPU DRAM cannot be accessed on GPU; pointers to GPU HBM cannot be access on CPU (changing in the future)
- Performance penalty when transferring data between GPU and CPU: **try to keep memory on GPU as much as possible**
- If a SPARTA style is not ported to Kokkos it will run on CPU in serial and require data transfer every time it is invoked: consider porting to Kokkos to improve performance
- SPARTA uses *Kokkos::DualView* sync and modify on Device and Host to transfer data

# Parallel Kernel Abstractions

- Kokkos supports functors, tagged kernels where the whole class is the functor, and C++ lambdas (anonymous functors)
- Functors are the most general but take the most programming effort (have to copy all the needed data into the functor)
- Typically use tagged kernels in SPARTA for convenience
- Can use C++ lambdas for simple kernels, but must use KOKKOS\_CLASS\_LAMBDA to capture *this* pointer either explicitly or implicitly, see  
[https://github.com/ibaned/lambda\\_users\\_guide](https://github.com/ibaned/lambda_users_guide)

# Kokkos Porting Example

- SPARTA designed to be very modular so adding a new style is easier, rarely need to touch “core” code
- Virtual inheritance: inherit as much from parent “compute temp” as possible to reduce code duplication
- Example: simple “compute temp/kk” style
- Only 1 loop that has any real computation (and that would benefit from threading):

```
for (int i = 0; i < nlocal; i++) {  
    v = particles[i].v;  
    t += (v[0]*v[0] + v[1]*v[1] + v[2]*v[2]) *  
        species[particles[i].ispecies].mass;  
}
```

# Kokkos Porting Example

```
KOKKOS_INLINE_FUNCTION
void ComputeTempKokkos::operator()(const int& i, double& lsum) const {
    double* v = d_particles[i].v;
    const int ispecies = d_particles[i].ispecies;
    const double mass = d_species[ispecies].mass;
    lsum += (v[0]*v[0] + v[1]*v[1] + v[2]*v[2]) * mass;
}
```

Single loop iteration

```
copymode = 1;
invoked_scalar = update->ntimestep;
ParticleKokkos* particle_kk = (ParticleKokkos*) particle;
particle_kk->sync(Device, PARTICLE_MASK|SPECIES_MASK);
d_particles = particle_kk->k_particles.d_view;           Kokkos views
d_species = particle_kk->k_species.d_view;
int nlocal = particle->nlocal;

double t = 0.0;
auto range_policy = Kokkos::RangePolicy<DeviceType>(0, nlocal);
Kokkos::parallel_reduce(range_policy, *this, t);           Parallel dispatch
copymode = 0;
```

# Unified Virtual Memory (UVM)

- Normally have to manually copy data between CPU and GPU
- CUDA managed memory: automatically manages data transfer between GPU and CPU
- Less bug prone, useful for debugging, but typically slower
- Some systems (e.g. OLCF Summit) can transparently spill out of GPU HBM into much larger CPU DRAM, by paging memory back and forth automatically
- Allows running large problems that don't fit into GPU memory, with some performance overhead
- Compile with the Makefile setting

```
KOKKOS_CUDA_OPTIONS="enable_lambda,force_u  
vm" or CMake option Kokkos_ENABLE_CUDA_UVM=ON
```

# Typical Kokkos Debugging Workflow

- Much easier to debug on CPU than GPU!
1. Match Kokkos Serial backend (stats output) with vanilla CPU version
    - Tools: Kokkos debug (view bounds checking), gdb, valgrind, AddressSanitizer, printf
    - Use `global comm/sort yes` if running on multiple MPI ranks
    - Compile with `-DSPARTA_KOKKOS_EXACT`, use `twopass` option for `create_particles` and fix `emit/face`, stats output should exactly agree
    - Compiling with “`-O0`” can help get an accurate backtrace
  2. Match Kokkos OpenMP backend running on 2 or more threads with vanilla CPU (or Kokkos Serial)
    - Will not exactly match due to different pRNG (can’t use `-DSPARTA_KOKKOS_EXACT`)
    - Tools: Intel Inspector (many false positives), printf
    - Typical issue: data race or other thread safety issues
  3. Match Kokkos CUDA backend with Kokkos Serial backend:
    - Tools: cuda-gdb, cuda-memcheck, compile with UVM, printf
    - Compiling with Kokkos debug options (adds `-lineinfo`) or `-G` can help
    - Typical issues: missing sync/modify for data transfer (find with UVM), thread safety issues

# Performance Profiling Tools

1. Timing breakdown in SPARTA log file
2. Kokkos tools: my favorite tool is “space-time-stack”, shows both kernel times and memory use
3. nvprof (deprecated) for NVIDIA GPUs and rocprof for AMD GPUs
4. NVIDIA Nsight Compute and Systems tools (replacement for nvprof)
5. gprof, TotalView, etc. for CPU kernels

# Space-Time-Stack Tool Output

```
export KOKKOS_TOOLS_LIBS=~/kokkos-
tools/profiling/space-time-
stack/kp_space_time_stack.so
```

```
BEGIN KOKKOS PROFILING REPORT:
```

```
TOTAL TIME: 30.286 seconds
```

```
TOP-DOWN TIME TREE:
```

```
<average time> <percent of total time> <percent time in Kokkos> <percent MPI
imbalance> <remainder> <kernels per second> <number of calls> <name> [type]
=====
|-> 1.22e+01 sec 40.4% 100.0% 0.0% ----- 2000
N9SPARTA_NS12UpdateKokkosE/N9SPARTA_NS13TagUpdateMoveILi3ELi0ELi0ELi1EEE [for]
|-> 2.65e+00 sec 8.8% 100.0% 0.0% ----- 2000
N9SPARTA_NS16CollideVSSKokkosE/N9SPARTA_NS23TagCollideCollisionsOneILi0ELi1EEE
[for]
|-> 2.30e+00 sec 7.6% 100.0% 0.0% ----- 200
N9SPARTA_NS14ParticleKokkosE/N9SPARTA_NS36TagParticleReorder_COPYPARTICLELIST2E
[for]
|-> 2.17e+00 sec 7.2% 100.0% 0.0% ----- 1808
N9SPARTA_NS14ParticleKokkosE/N9SPARTA_NS15TagParticleSortILi1ELi0EEE [for]
```

# Space-Time-Stack Tool Output

KOKKOS CUDA SPACE:

=====

MAX MEMORY ALLOCATED: 3304909.0 kB

MPI RANK WITH MAX MEMORY: 0

ALLOCATIONS AT TIME OF HIGH WATER MARK:

41.1% particle:sorted

41.1% particle:particles

4.3% particle:plist

3.8% grid:cells

1.9% grid:cinfo

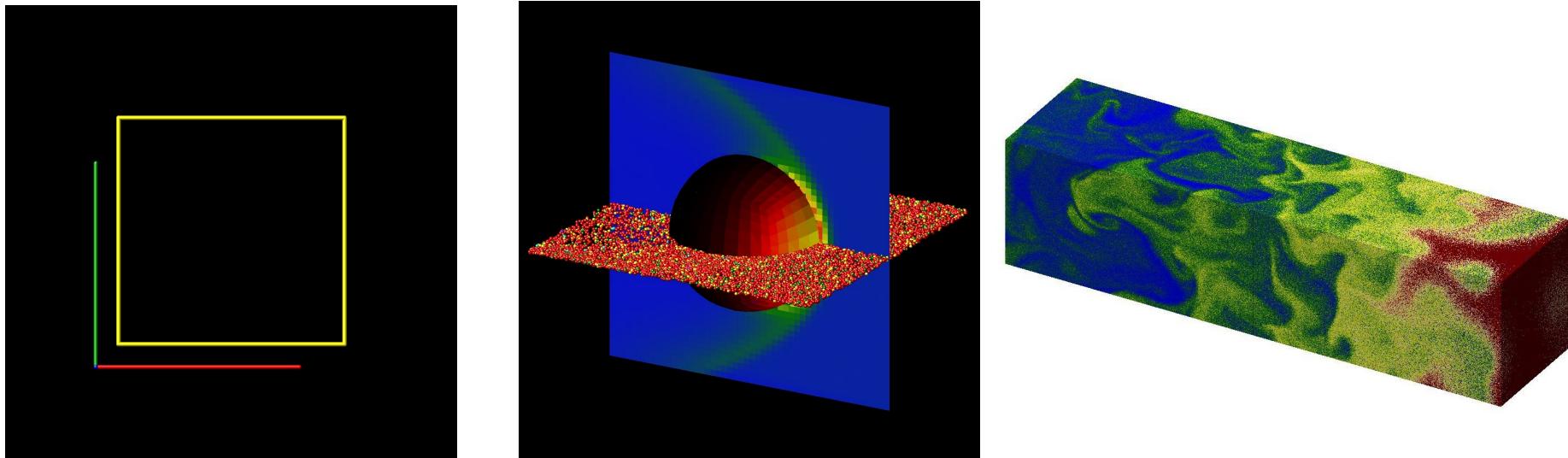
1.3% particle:sorted\_id

1.3% particle:mlist

1.3% particle:offsets\_part

# Dump Image

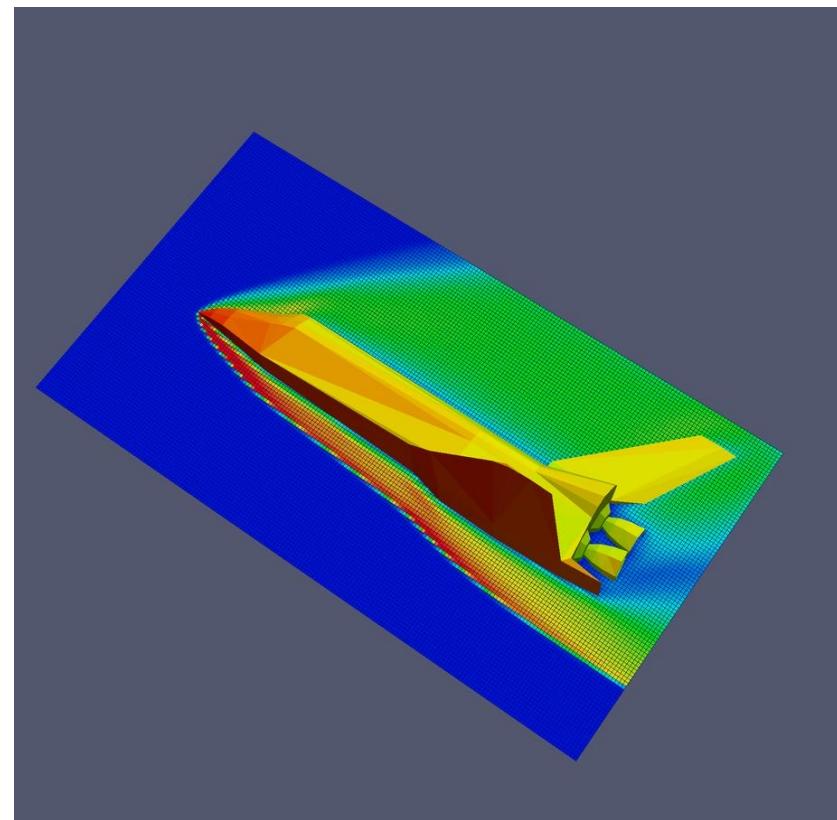
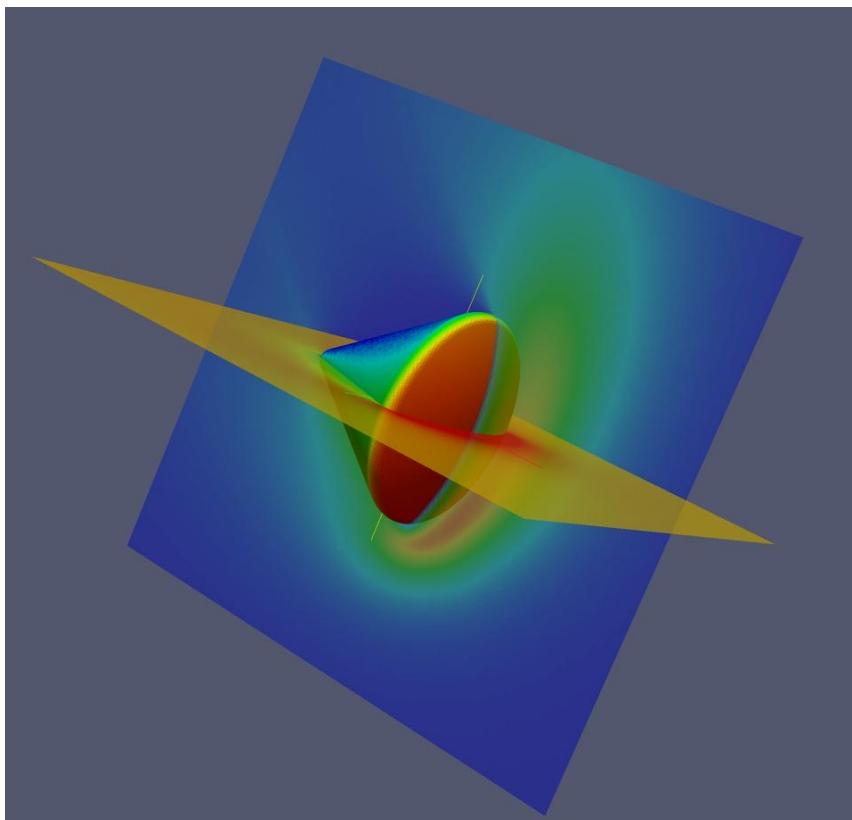
- [https://sparta.github.io/doc/dump\\_image.html](https://sparta.github.io/doc/dump_image.html)
- Can be used to visualize simulation, create movies



```
Example: dump 1 image all 50 image.*.ppm type  
type pdiam 3e-4 surf proc 0.01 size 512 512  
axes yes 0.9 0.02 particle yes zoom 15 box  
yes 0.02
```

# Paraview

- [https://sparta.github.io/doc/Section\\_howto.html#howto\\_16](https://sparta.github.io/doc/Section_howto.html#howto_16)
- Can be used to visualize simulations beyond simple dump image



# SPARTA User Support

- Mail list on SourceForge:  
<https://sourceforge.net/p/sparta/mailman/sparta-users>, get help using SPARTA code
- SPARTA development is supported on GitHub:  
<https://github.com/sparta/sparta>, submit tickets for code issues/enhancements or contribute new features



# Thank You

- Questions?