SPARTA Users Manual

name of code

http://sparta.sandia.gov - Sandia National Laboratories

Table of Contents

Table of Contents

Table of Contents

SPARTA Documentation

Version info:

The SPARTA "version" is the date when it was released, such as 10 Jan 2012. SPARTA is updated continuously. Whenever we fix a bug or add a feature, we release it immediately, and post a notice on this page of the WWW site. Each dated copy of SPARTA contains all the features and bug-fixes up to and including that version date. The version date is printed to the screen and logfile every time you run SPARTA. It is also in the file src/version.h and in the SPARTA directory name created when you unpack a tarball.

- If you browse the HTML doc pages on the SPARTA WWW site, they always describe the most current version of SPARTA.
- If you browse the HTML doc pages included in your tarball, they describe the version you have.
- The PDF file on the WWW site or in the tarball is updated about once per month. This is because it is large, and we don't want it to be part of very patch.
- There is also a Developer.pdf file in the doc directory, which describes the internal structure and algorithms of SPARTA.

SPARTA stands for Stochastic PArallel Rarefied-gas Time-accurate Analyzer.

SPARTA is a Direct Simulation Montel Carlo (DSMC) simulator designed to run efficiently on parallel computers. It was developed at Sandia National Laboratories, a US Department of Energy facility, with funding from the DOE. It is an open-source code, distributed freely under the terms of the GNU Public License (GPL).

The primary developers of SPARTA are Steve Plimpton, and Michael Gallis_mg who can be contacted at sjplimp,magalli at sandia.gov. The SPARTA WWW Site at http://sparta.sandia.gov has more information about the code and its uses.

The SPARTA documentation is organized into the following sections. If you find errors or omissions in this manual or have suggestions for useful information to add, please send an email to the developers so we can improve the SPARTA documentation.

Once you are familiar with SPARTA, you may want to bookmark this page at Section_commands.html#comm since it gives quick access to documentation for all SPARTA commands.

PDF file of the entire manual, generated by htmldoc

1. Introduction

These sections provide an overview of what SPARTA can and can't do, describe what it means for SPARTA to be an open-source code, and acknowledge the funding and people who have contributed to SPARTA.

1.1 What is SPARTA

SPARTA is a Direct Simulation Montel Carlo code that models rarefied gases, using a variety of collision, chemistry, boundary condition models.

For examples of SPARTA simulations, see the Publications page of the SPARTA WWW Site.

SPARTA runs efficiently on single-processor desktop or laptop machines, but is designed for parallel computers. It will run on any parallel machine that compiles C++ and supports the MPI message-passing library. This includes distributed- or shared-memory parallel machines as well as commodity clusters.

SPARTA can model systems with only a few particles up to millions or billions. See this section for information on SPARTA performance and scalability, or the Benchmarks section of the SPARTA WWW Site.

SPARTA is a freely-available open-source code, distributed under the terms of the GNU Public License, which means you can use or modify the code however you wish. See this section for a brief discussion of the open-source philosophy.

SPARTA is designed to be easy to modify or extend with new capabilities, such as new collision or chemistry models, boundary conditions, or diagnostics. See this section for more details.

SPARTA is written in C++ which is used at a hi-level to structure the code and its options in an object-oriented fashion. The kernel computations use simple data structures and C-like code for effciency. So SPARTA is really written in an object-oriented C style.

SPARTA was developed with internal funding at Sandia National Laboratories, a US Department of Energy lab. See this section for more information on SPARTA funding and individuals who have contributed to SPARTA.

In the most general sense, SPARTA integrates Newton's equations of motion for collections of atoms, molecules, or macroscopic particles that interact via short- or long-range forces with a variety of initial and/or boundary conditions. For computational efficiency SPARTA uses neighbor lists to keep track of nearby particles. The lists are optimized for systems with particles that are repulsive at short distances, so that the local density of particles never becomes too large. On parallel machines, SPARTA uses spatial-decomposition techniques to partition the simulation domain into small 3d sub-domains, one of which is assigned to each processor. Processors communicate and store "ghost" atom information for atoms that border their sub-domain. SPARTA is most efficient (in a parallel sense) for systems whose particles fill a 3d rectangular box with roughly uniform density. Papers with technical details of the algorithms used in SPARTA are listed in this section.

1.2 SPARTA features

This section highlights SPARTA features, with pointers to specific commands which give more details. If SPARTA doesn't have your favorite collision model, boundary condition, or diagnostic, see this section, which describes how you can add it to SPARTA.

General features

- runs on a single processor or in parallel
- distributed-memory message-passing parallelism (MPI)
- spatial-decomposition of simulation domain for parallelism
- open-source distribution
- highly portable C++
- optional libraries used: MPI
- easy to extend with new features and functionality
- runs from an input script
- syntax for defining and using variables and formulas
- syntax for looping over runs and breaking out of loops
- run one or multiple simulations simultaneously (in parallel) from one script
- build as library, invoke SPARTA thru library interface or provided Python wrapper
- couple with other codes: SPARTA calls other code, other code calls SPARTA, umbrella code calls both

Collision models

(atom style command)

Force fields

(pair style, bond style, angle style, dihedral style, improper style, kspace style commands)

- pairwise potentials: Lennard-Jones, Buckingham, Morse, Born-Mayer-Huggins, Yukawa, soft, class 2 (COMPASS), hydrogen bond, tabulated
- charged pairwise potentials: Coulombic, point-dipole
- manybody potentials: EAM, Finnis/Sinclair EAM, modified EAM (MEAM), embedded ion method (EIM), EDIP, ADP, Stillinger-Weber, Tersoff, REBO, AIREBO, ReaxFF, COMB
- electron force field (eFF, AWPMD)
- coarse-grained potentials: DPD, GayBerne, REsquared, colloidal, DLVO
- mesoscopic potentials: granular, Peridynamics, SPH
- bond potentials: harmonic, FENE, Morse, nonlinear, class 2, quartic (breakable)
- angle potentials: harmonic, CHARMM, cosine, cosine/squared, cosine/periodic, class 2 (COMPASS)
- dihedral potentials: harmonic, CHARMM, multi-harmonic, helix, class 2 (COMPASS), OPLS
- improper potentials: harmonic, cvff, umbrella, class 2 (COMPASS)
- polymer potentials: all-atom, united-atom, bead-spring, breakable
- water potentials: TIP3P, TIP4P, SPC
- implicit solvent potentials: hydrodynamic lubrication, Debye
- long-range Coulombics and dispersion: Ewald, PPPM (similar to particle-mesh Ewald), Ewald/N for long-range Lennard-Jones
- force-field compatibility with common CHARMM, AMBER, DREIDING, OPLS, GROMACS, COMPASS options
- handful of GPU-enabled pair styles

hybrid potentials: multiple pair, bond, angle, dihedral, improper potentials can be used in one simulation overlaid

potentials: superposition of multiple pair potentials

Atom creation

(read_data, lattice, create_atoms, delete_atoms, displace_atoms, replicate commands)

- read in atom coords from files
- create atoms on one or more lattices (e.g. grain boundaries)
- delete geometric or logical groups of atoms (e.g. voids)
- replicate existing atoms multiple times
- displace atoms

Ensembles, constraints, and boundary conditions

(fix command)

- 2d or 3d systems
- orthogonal or non-orthogonal (triclinic symmetry) simulation domains
- constant NVE, NVT, NPT, NPH, Parinello/Rahman integrators
- thermostatting options for groups and geometric regions of atoms
- pressure control via Nose/Hoover or Berendsen barostatting in 1 to 3 dimensions
- simulation box deformation (tensile and shear)
- harmonic (umbrella) constraint forces
- rigid body constraints
- SHAKE bond and angle constraints
- bond breaking, formation, swapping
- walls of various kinds
- non-equilibrium molecular dynamics (NEMD)
- variety of additional boundary conditions and constraints

Integrators

(run, run_style, minimize commands)

- velocity-Verlet integrator
- Brownian dynamics
- rigid body integration
- energy minimization via conjugate gradient or steepest descent relaxation
- rRESPA hierarchical timestepping

Diagnostics

- see the various flavors of the fix and compute commands

Output

(dump, restart commands)

- log file of thermodynamic info
- text dump files of atom coords, velocities, other per-atom quantities
- binary restart files
- parallel I/O of dump and restart files

- per-atom quantities (energy, stress, centro-symmetry parameter, CNA, etc)
- user-defined system-wide (log file) or per-atom (dump file) calculations
- spatial and time averaging of per-atom quantities
- time averaging of system-wide quantities
- atom snapshots in native, XYZ, XTC, DCD, CFG formats

Multi-replica models

nudged elastic band parallel replica dynamics temperature accelerated dynamics parallel tempering

Pre- and post-processing

- Various pre- and post-processing serial tools are packaged with SPARTA; see these doc pages.
- Our group has also written and released a separate toolkit called Pizza.py which provides tools for doing setup, analysis, plotting, and visualization for SPARTA simulations. Pizza.py is written in Python and is available for download from the Pizza.py WWW site.

Specialized features

These are SPARTA capabilities which you may not think of as typical molecular dynamics options:

- stochastic rotation dynamics (SRD)
- real-time visualization and interactive MD
- atom-to-continuum coupling with finite elements
- coupled rigid body integration via the POEMS library
- grand canonical Monte Carlo insertions/deletions
- Direct Simulation Monte Carlo for low-density fluids
- Peridynamics mesoscale modeling
- targeted and steered molecular dynamics
- two-temperature electron model

SPARTA non-features

SPARTA is designed to efficiently compute Newton's equations of motion for a system of interacting particles. Many of the tools needed to pre- and post-process the data for such simulations are not included in the SPARTA kernel for several reasons:

- the desire to keep SPARTA simple
- they are not parallel operations
- other codes already do them
- limited development resources

Specifically, SPARTA itself does not:

- run thru a GUI
- build molecular systems
- assign force-field coefficients automagically
- perform sophisticated analyses of your MD simulation
- visualize your MD simulation
- plot your output data

A few tools for pre- and post-processing tasks are provided as part of the SPARTA package; they are described in this section. However, many people use other codes or write their own tools for these tasks.

As noted above, our group has also written and released a separate toolkit called Pizza.py which addresses some of the listed bullets. It provides tools for doing setup, analysis, plotting, and visualization for SPARTA simulations. Pizza.py is written in Python and is available for download from the Pizza.py WWW site.

SPARTA requires as input a list of initial atom coordinates and types, molecular topology information, and force-field coefficients assigned to all atoms and bonds. SPARTA will not build molecular systems and assign force-field parameters for you.

For atomic systems SPARTA provides a create_atoms command which places atoms on solid-state lattices (fcc, bcc, user-defined, etc). Assigning small numbers of force field coefficients can be done via the pair coeff, bond coeff, angle coeff, etc commands. For molecular systems or more complicated simulation geometries, users typically use another code as a builder and convert its output to SPARTA input format, or write their own code to generate atom coordinate and molecular topology for SPARTA to read in.

For complicated molecular systems (e.g. a protein), a multitude of topology information and hundreds of force-field coefficients must typically be specified. We suggest you use a program like CHARMM or AMBER or other molecular builders to setup such problems and dump its information to a file. You can then reformat the file as SPARTA input. Some of the tools in this section can assist in this process.

Similarly, SPARTA creates output files in a simple format. Most users post-process these files with their own analysis tools or re-format them for input into other programs, including visualization packages. If you are convinced you need to compute something on-the-fly as SPARTA runs, see this section for a discussion of how you can use the dump and compute and fix commands to print out data of your choosing. Keep in mind that complicated computations can slow down the molecular dynamics timestepping, particularly if the computations are not parallel, so it is often better to leave such analysis to post-processing codes.

A very simple (yet fast) visualizer is provided with the SPARTA package - see the xmovie tool in this section. It creates xyz projection views of atomic coordinates and animates them. We find it very useful for debugging purposes. For high-quality visualization we recommend the following packages:

- VMD
- AtomEye
- PyMol
- Raster3d
- RasMol

Other features that SPARTA does not yet (and may never) support are discussed in this section.

Finally, these are freely-available molecular dynamics codes, most of them parallel, which may be well-suited to the problems you want to model. They can also be used in conjunction with SPARTA to perform complementary modeling tasks.

- CHARMM
- AMBER
- NAMD
- NWCHEM
- DL_POLY
- Tinker

CHARMM, AMBER, NAMD, NWCHEM, and Tinker are designed primarily for modeling biological molecules. CHARMM and AMBER use atom-decomposition (replicated-data) strategies for parallelism; NAMD and NWCHEM use spatial-decomposition approaches, similar to SPARTA. Tinker is a serial code. DL_POLY includes potentials for a variety of biological and non-biological materials; both a replicated-data and spatial-decomposition version exist.

1.4 Open source distribution

SPARTA comes with no warranty of any kind. As each source file states in its header, it is a copyrighted code that is distributed free-of- charge, under the terms of the GNU Public License (GPL). This is often referred to as open-source distribution - see www.gnu.org or www.opensource.org for more details. The legal text of the GPL is in the LICENSE file that is included in the SPARTA distribution.

Here is a summary of what the GPL means for SPARTA users:

(1) Anyone is free to use, modify, or extend SPARTA in any way they choose, including for commercial purposes.

(2) If you distribute a modified version of SPARTA, it must remain open-source, meaning you distribute it under the terms of the GPL. You should clearly annotate such a code as a derivative version of SPARTA.

(3) If you release any code that includes SPARTA source code, then it must also be open-sourced, meaning you distribute it under the terms of the GPL.

(4) If you give SPARTA files to someone else, the GPL LICENSE file and source file headers (including the copyright and GPL notices) should remain part of the code.

In the spirit of an open-source code, these are various ways you can contribute to making SPARTA better. You can send email to the developers on any of these items.

- Point prospective users to the SPARTA WWW Site. Mention it in talks or link to it from your WWW site.
- If you find an error or omission in this manual or on the SPARTA WWW Site, or have a suggestion for something to clarify or include, send an email to the developers.
- If you find a bug, this section describes how to report it.
- If you publish a paper using SPARTA results, send the citation (and any cool pictures or movies if you like) to add to the Publications, Pictures, and Movies pages of the SPARTA WWW Site, with links and attributions back to you.
- Create a new Makefile.machine that can be added to the src/MAKE directory.
- The tools sub-directory of the SPARTA distribution has various stand-alone codes for pre- and post-processing of SPARTA data. More details are given in this section. If you write a new tool that users will find useful, it can be added to the SPARTA distribution.
- SPARTA is designed to be easy to extend with new code for features like potentials, boundary conditions, diagnostic computations, etc. This section gives details. If you add a feature of general interest, it can be added to the SPARTA distribution.
- The Benchmark page of the SPARTA WWW Site lists SPARTA performance on various platforms. The files needed to run the benchmarks are part of the SPARTA distribution. If your machine is sufficiently different from those listed, your timing data can be added to the page.
- You can send feedback for the User Comments page of the SPARTA WWW Site. It might be added to the page. No promises.

- Cash. Small denominations, unmarked bills preferred. Paper sack OK. Leave on desk. VISA also accepted. Chocolate chip cookies encouraged.

1.5 Acknowledgments and citations

SPARTA development has been funded by the US Department of Energy (DOE), through its CRADA, LDRD, ASCI, and Genomes-to-Life programs and its OASCR and OBER offices.

Specifically, work on the latest version was funded in part by the US Department of Energy's Genomics:GTL program (www.doegenomestolife.org) under the project, "Carbon Sequestration in Synechococcus Sp.: From Molecular Machines to Hierarchical Modeling".

The following paper describe the basic parallel algorithms used in SPARTA. If you use SPARTA results in your published work, please cite the following paper and include a pointer to the SPARTA WWW Site (http://dsmc.sandia.gov):

S. J. Plimpton, **Fast Parallel Algorithms for Short-Range Molecular Dynamics**, J Comp Phys, 117, 1-19 (1995).

Other papers describing specific algorithms used in SPARTA are listed under the Citing SPARTA link of the SPARTA WWW page.

The Publications link on the SPARTA WWW page lists papers that have cited SPARTA. If your paper is not listed there for some reason, feel free to send us the info. If the simulations in your paper produced cool pictures or animations, we'll be pleased to add them to the Pictures or Movies pages of the SPARTA WWW site.

The core group of SPARTA developers is at Sandia National Labs:

- Steve Plimpton, sjplimp at sandia.gov
- Aidan Thompson, athomps at sandia.gov
- Paul Crozier, pscrozi at sandia.gov

The following folks are responsible for significant contributions to the code, or other aspects of the SPARTA development effort. Many of the packages they have written are somewhat unique to SPARTA and the code would not be as general-purpose as it is without their expertise and efforts.

- Axel Kohlmeyer (Temple U), akohlmey at gmail.com, SVN and Git repositories, indefatigable mail list responder, USER-CG-CMM and USER-OMP packages
- Roy Pollock (LLNL), Ewald and PPPM solvers
- Mike Brown (ORNL), brownw at ornl.gov, GPU package
- Greg Wagner (Sandia), gjwagne at sandia.gov, MEAM package for MEAM potential
- Mike Parks (Sandia), mlparks at sandia.gov, PERI package for Peridynamics
- Rudra Mukherjee (JPL), Rudranarayan.M.Mukherjee at jpl.nasa.gov, POEMS package for articulated rigid body motion
- Reese Jones (Sandia) and collaborators, rjones at sandia.gov, USER-ATC package for atom/continuum coupling
- Ilya Valuev (JIHT), valuev at physik.hu-berlin.de, USER-AWPMD package for wave-packet MD
- Christian Trott (U Tech Ilmenau), christian.trott at tu-ilmenau.de, USER-CUDA package
- Andres Jaramillo-Botero (Caltech), ajaramil at wag.caltech.edu, USER-EFF package for electron force field
- Pieter in' t Veld (BASF), pieter.intveld at basf.com, USER-EWALDN package for 1/r^N long-range solvers

- Christoph Kloss (JKU), Christoph.Kloss at jku.at, USER-LIGGGHTS package for granular models and granular/fluid coupling
- Metin Aktulga (LBL), hmaktulga at lbl.gov, USER-REAXC package for C version of ReaxFF
- Georg Gunzenmuller (EMI), georg.ganzenmueller at emi.fhg.de, USER-SPH package

As discussed in this section, SPARTA originated as a cooperative project between DOE labs and industrial partners. Folks involved in the design and testing of the original version of SPARTA were the following:

- John Carpenter (Mayo Clinic, formerly at Cray Research)
- Terry Stouch (Lexicon Pharmaceuticals, formerly at Bristol Myers Squibb)
- Steve Lustig (Dupont)
- Jim Belak (LLNL)

2. Getting Started

This section describes how to build and run SPARTA, for both new and experienced users.

2.1 What's in the SPARTA distribution

When you download SPARTA you will need to unzip and untar the downloaded file with the following commands, after placing the file in an appropriate directory.

```
gunzip lammps*.tar.gz
tar xvf lammps*.tar
```

This will create a SPARTA directory containing two files and several sub-directories:

README	text file
LICENSE	the GNU General Public License (GPL)
bench	benchmark problems
couple	code coupling examples, using SPARTA as a library
doc	documentation
examples	simple test problems
potentials	embedded atom method (EAM) potential files
src	source files
tools	pre- and post-processing tools

If you download one of the Windows executables from the download page, then you just get a single file:

```
lmp_windows.exe
```

Skip to the Running SPARTA sections for info on how to launch these executables on a Windows box.

The Windows executables for serial or parallel only include certain packages and bug-fixes/upgrades listed on this page up to a certain date, as stated on the download page. If you want something with more packages or that is more current, you'll have to download the source tarball and build it yourself from source code using Microsoft Visual Studio, as described in the next section.

2.2 Making SPARTA

This section has the following sub-sections:

- Read this first

- Steps to build a SPARTA executable
- Common errors that can occur when making SPARTA
- Additional build tips
- Building for a Mac
- Building for Windows

Read this first:

Building SPARTA can be non-trivial. You may need to edit a makefile, there are compiler options to consider, additional libraries can be used (MPI, FFT, JPEG), SPARTA packages may be included or excluded, some of these packages use auxiliary libraries which need to be pre-built, etc.

Please read this section carefully. If you are not comfortable with makefiles, or building codes on a Unix platform, or running an MPI job on your machine, please find a local expert to help you. Many compiling, linking, and run problems that users have are not really SPARTA issues - they are peculiar to the user's system, compilers, libraries, etc. Such questions are better answered by a local expert.

If you have a build problem that you are convinced is a SPARTA issue (e.g. the compiler complains about a line of SPARTA source code), then please send an email to the developers.

If you succeed in building SPARTA on a new kind of machine, for which there isn't a similar Makefile for in the src/MAKE directory, send it to the developers and we'll include it in future SPARTA releases.

Steps to build a SPARTA executable:

Step 0

The src directory contains the C++ source and header files for SPARTA. It also contains a top-level Makefile and a MAKE sub-directory with low-level Makefile.* files for many machines. From within the src directory, type "make" or "gmake". You should see a list of available choices. If one of those is the machine and options you want, you can type a command like:

```
make linux
or
gmake mac
```

Note that on a multi-processor or multi-core platform you can launch a parallel make, by using the "-j" switch with the make command, which will build SPARTA more quickly.

If you get no errors and an executable like lmp_linux or lmp_mac is produced, you're done; it's your lucky day.

Note that by default only a few of SPARTA optional pacakges are installed. To build SPARTA with optional packages, see this section below.

Step 1

If Step 0 did not work, you will need to create a low-level Makefile for your machine, like Makefile.foo. You should make a copy of an existing src/MAKE/Makefile.* as a starting point. The only portions of the file you need to edit are the first line, the "compiler/linker settings" section, and the "SPARTA-specific settings" section.

Step 2

Change the first line of src/MAKE/Makefile.foo to list the word "foo" after the "#", and whatever other options it will set. This is the line you will see if you just type "make".

Step 3

The "compiler/linker settings" section lists compiler and linker settings for your C++ compiler, including optimization flags. You can use g++, the open-source GNU compiler, which is available on all Unix systems. You can also use mpicc which will typically be available if MPI is installed on your system, though you should check which actual compiler it wraps. Vendor compilers often produce faster code. On boxes with Intel CPUs, we suggest using the commercial Intel icc compiler, which can be downloaded from Intel's compiler site.

If building a C++ code on your machine requires additional libraries, then you should list them as part of the LIB variable.

The DEPFLAGS setting is what triggers the C++ compiler to create a dependency list for a source file. This speeds re-compilation when source (*.cpp) or header (*.h) files are edited. Some compilers do not support dependency file creation, or may use a different switch than -D. GNU g++ works with -D. If your compiler can't create dependency files, then you'll need to create a Makefile.foo patterned after Makefile.storm, which uses different rules that do not involve dependency files. Note that when you build SPARTA for the first time on a new platform, a long list of *.d files will be printed out rapidly. This is not an error; it is the Makefile doing its normal creation of dependencies.

Step 4

The "system-specific settings" section has several parts. Note that if you change any -D setting in this section, you should do a full re-compile, after typing "make clean" (which will describe different clean options).

The LMP_INC variable is used to include options that turn on ifdefs within the SPARTA code. The options that are currently recogized are:

- -DSPARTA_GZIP
- -DSPARTA_JPEG
- -DSPARTA_XDR
- -DSPARTA_SMALLBIG
- -DSPARTA_BIGBIG
- -DSPARTA_SMALLSMALL
- -DSPARTA_LONGLONG_TO_LONG
- -DPACK_ARRAY
- -DPACK_POINTER
- -DPACK_MEMCPY

The read_data and dump commands will read/write gzipped files if you compile with -DSPARTA_GZIP. It requires that your Unix support the "popen" command.

If you use -DSPARTA_JPEG, the dump image command will be able to write out JPEG image files. If not, it will only be able to write out text-based PPM image files. For JPEG files, you must also link SPARTA with a JPEG library, as described below.

If you use -DSPARTA_XDR, the build will include XDR compatibility files for doing particle dumps in XTC format. This is only necessary if your platform does have its own XDR files available. See the Restrictions section of the dump command for details.

Use at most one of the -DSPARTA_SMALLBIG, -DSPARTA_BIGBIG, -D-DSPARTA_SMALLSMALL settings. The default is -DSPARTA_SMALLBIG. These refer to use of 4-byte (small) vs 8-byte (big) integers within SPARTA, as described in src/lmptype.h. The only reason to use the BIGBIG setting is to enable simulation of huge molecular systems with more than 2 billion atoms. The only reason to use the SMALLSMALL setting is if your machine does not support 64-bit integers.

The -DSPARTA_LONGLONG_TO_LONG setting may be needed if your system or MPI version does not recognize "long long" data types. In this case a "long" data type is likely already 64-bits, in which case this setting will convert to that data type.

Using one of the -DPACK_ARRAY, -DPACK_POINTER, and -DPACK_MEMCPY options can make for faster parallel FFTs (in the PPPM solver) on some platforms. The -DPACK_ARRAY setting is the default. See the kspace_style command for info about PPPM. See Step 6 below for info about building SPARTA with an FFT library.

Step 5

The 3 MPI variables are used to specify an MPI library to build SPARTA with.

SPARTA want SPARTA to run in parallel, you must have an MPI library installed on your platform. If you use an MPI-wrapped compiler, such as "mpicc" to build SPARTA, you should be able to leave these 3 variables blank; the MPI wrapper knows where to find the needed files. If not, and MPI is installed on your system in the usual place (under /usr/local), you also may not need to specify these 3 variables. On some large parallel machines which use "modules" for their compile/link environements, you may simply need to include the correct module in your build environment. Or the parallel machine may have a vendor-provided MPI which the compiler has no trouble finding.

Failing this, with these 3 variables you can specify where the mpi.h file (MPI_INC) and the MPI library file (MPI_PATH) are found and the name of the library file (MPI_LIB).

If you are installing MPI yourself, we recommend Argonne's MPICH 1.2 or 2.0 or OpenMPI. MPICH can be downloaded from the Argonne MPI site. OpenMPI can be downloaded the OpenMPI site. LAM MPI should also work. If you are running on a big parallel platform, your system people or the vendor should have already installed a version of MPI, which will be faster than MPICH or OpenMPI or LAM, so find out how to build and link with it. If you use MPICH or OpenMPI or LAM, you will have to configure and build it for your platform. The MPI configure script should have compiler options to enable you to use the same compiler you are using for the SPARTA build, which can avoid problems that can arise when linking SPARTA to the MPI library.

If you just want to run SPARTA on a single processor, you can use the dummy MPI library provided in src/STUBS, since you don't need a true MPI library installed on your system. See the src/MAKE/Makefile.serial file for how to specify the 3 MPI variables in this case. You will also need to build the STUBS library for your platform before making SPARTA itself. From the src directory, type "make stubs", or from the STUBS dir, type "make" and it should create a libmpi.a suitable for linking to SPARTA. If this build fails, you will need to edit the STUBS/Makefile for your platform.

The file STUBS/mpi.cpp provides a CPU timer function called MPI_Wtime() that calls gettimeofday() . If your system doesn't support gettimeofday() , you'll need to insert code to call another timer. Note that the ANSI-standard function clock() rolls over after an hour or so, and is therefore insufficient for timing long SPARTA simulations.

Step 6

The 3 FFT variables allow you to specify an FFT library which SPARTA uses (for performing 1d FFTs) when running the particle-particle particle-mesh (PPPM) option for long-range Coulombics via the kspace_style command.

SPARTA supports various open-source or vendor-supplied FFT libraries for this purpose. If you leave these 3 variables blank, SPARTA will use the open-source KISS FFT library, which is included in the SPARTA distribution. This library is portable to all platforms and for typical SPARTA simulations is almost as fast as FFTW or vendor optimized libraries. If you are not including the KSPACE package in your build, you can also leave the 3 variables blank.

Otherwise, select which kinds of FFTs to use as part of the FFT_INC setting by a switch of the form -DFFT_XXX. Recommended values for XXX are: MKL, SCSL, FFTW2, and FFTW3. Legacy options are: INTEL, SGI, ACML, and T3E. For backward compatability, using -DFFT_FFTW will use the FFTW2 library. Using -DFFT_NONE will use the KISS library described above.

You may also need to set the FFT_INC, FFT_PATH, and FFT_LIB variables, so the compiler and linker can find the needed FFT header and library files. Note that on some large parallel machines which use "modules" for their compile/link environements, you may simply need to include the correct module in your build environment. Or the parallel machine may have a vendor-provided FFT library which the compiler has no trouble finding.

FFTW is a fast, portable library that should also work on any platform. You can download it from www.fftw.org. Both the legacy version 2.1.X and the newer 3.X versions are supported as -DFFT_FFTW2 or -DFFT_FFTW3. Building FFTW for your box should be as simple as ./configure; make. Note that on some platforms FFTW2 has been pre-installed, and uses renamed files indicating the precision it was compiled with, e.g. sfftw.h, or dfftw.h instead of fftw.h. In this case, you can specify an additional define variable for FFT_INC called -DFFTW2_SIZE, which will select the correct include file. In this case, for FFT_LIB you must also manually specify the correct library, namely -lsfftw or -ldfftw.

The FFT_INC variable also allows for a -DFFT_SINGLE setting that will use single-precision FFTs with PPPM, which can speed-up long-range calulations, particularly in parallel or on GPUs. Fourier transform and related PPPM operations are somewhat insensitive to floating point truncation errors and thus do not always need to be performed in double precision. Using the -DFFT_SINGLE setting trades off a little accuracy for reduced memory use and parallel communication costs for transposing 3d FFT data. Note that single precision FFTs have only been tested with the FFTW3, FFTW2, MKL, and KISS FFT options.

Step 7

The 3 JPG variables allow you to specify a JPEG library which SPARTA uses when writing out JPEG files via the dump image command. These can be left blank if you do not use the -DSPARTA_JPEG switch discussed above in Step 4, since in that case JPEG output will be disabled.

A standard JPEG library usually goes by the name libjpeg.a and has an associated header file jpeglib.h. Whichever JPEG library you have on your platform, you'll need to set the appropriate JPG_INC, JPG_PATH, and JPG_LIB variables, so that the compiler and linker can find it.

As before, if these header and library files are in the usual place on your machine, you may not need to set these variables.

Step 8

Note that by default only a few of SPARTA optional pacakges are installed. To build SPARTA with optional packages, see this section below, before proceeding to Step 9.

Step 9

That's it. Once you have a correct Makefile.foo, you have installed the optional SPARTA packages you want to include in your build, and you have pre-built any other needed libraries (e.g. MPI, FFT, package libraries), all you need to do from the src directory is type something like this:

```
make foo
or
gmake foo
```

You should get the executable lmp_foo when the build is complete.

Errors that can occur when making SPARTA:

IMPORTANT NOTE: If an error occurs when building SPARTA, the compiler or linker will state very explicitly what the problem is. The error message should give you a hint as to which of the steps above has failed, and what you need to do in order to fix it. Building a code with a Makefile is a very logical process. The compiler and linker need to find the appropriate files and those files need to be compatible with SPARTA source files. When a make fails, there is usually a very simple reason, which you or a local expert will need to fix.

Here are two non-obvious errors that can occur:

(1) If the make command breaks immediately with errors that indicate it can't find files with a "*" in their names, this can be because your machine's native make doesn't support wildcard expansion in a makefile. Try gmake instead of make. If that doesn't work, try using a -f switch with your make command to use a pre-generated Makefile.list which explicitly lists all the needed files, e.g.

```
make makelist
make -f Makefile.list linux
gmake -f Makefile.list mac
```

The first "make" command will create a current Makefile.list with all the file names in your src dir. The 2nd "make" command (make or gmake) will use it to build SPARTA. Note that you should include/exclude any desired optional packages before using the "make makelist" command.

(2) If you get an error that says something like 'identifier "atoll" is undefined', then your machine does not support "long long" integers. Try using the -DSPARTA_LONGLONG_TO_LONG setting described above in Step 4.

Additional build tips:

(1) Building SPARTA for multiple platforms.

You can make SPARTA for multiple platforms from the same src directory. Each target creates its own object sub-directory called Obj_name where it stores the system-specific *.o files.

(2) Cleaning up.

Typing "make clean-all" or "make clean-foo" will delete *.o object files created when SPARTA is built, for either all builds or for a particular machine.

(3) Changing the size limits in src/lmptype.h

If you are running a very large problem (billions of atoms or more) and get a run-time error about the system being too big, either on a per-processor basis or in total size, then you may need to change one or more settings in src/lmptype.h and re-compile SPARTA.

As the documentation in that file explains, you have basically two choices to make:

- set the data type size of integer atom IDs to 4 or 8 bytes
- set the data type size of integers that store the total system size to 4 or 8 bytes

The default for atom IDs is 4-byte integers since there is a memory and communication cost for 8-byte integers. Non-molecular problems do not need atom IDs so this does not restrict their size. Molecular problems (which use IDs to define molecular topology), are limited to about 2 billion atoms (2^31) with 4-byte IDs. With 8-byte IDs they are effectively unlimited in size (2^63).

The default for total system size quantities (like the number of atoms or timesteps) is 8-byte integers by default which is effectively unlimited in size (2^63). If your system or MPI implementation does not support 8-byte integers, an error will be generated, and you will need to set "bigint" to 4-byte integers. This restricts your total system size to about 2 billion atoms or timesteps (2^31).

Note that in src/lmptype.h there are also settings for the MPI data types associated with the integers that store atom IDs and total system sizes, which need to be set consistent with the associated C data types.

In all cases, the size of problem that can be run on a per-processor basis is limited by 4-byte integer storage to about 2 billion atoms per processor (2^31), which should not normally be a restriction since such a problem would have a huge per-processor memory footprint due to neighbor lists and would run very slowly in terms of CPU secs/timestep.

Building for a Mac:

OS X is BSD Unix, so it should just work. See the Makefile.mac file.

Building for Windows:

The SPARTA download page has an option to download both a serial and parallel pre-built Windows exeutable. See the Running SPARTA section for instructions for running these executables on a Windows box.

If the pre-built executable doesn't have the options you want, then you can build SPARTA from its source files on a Windows box. One way to do this is install and use cygwin to build SPARTA with a standard Linus make, just as you would on any Linux box; see src/MAKE/Makefile.cygwin.

There is a also a src/WINDOWS directory that contains project files for Microsoft Visual Studio 2005, which should also work with later versions of VS. That directory contains a README.txt file which provides instructions for building SPARTA from source code using Visual Studio that are hopefully easy to follow for Windows and VS users.

Four VS project options are provided. The first includes the default packages (MANYBODY, MOLECULE, and KSPACE). The second includes all standard packages (except GPU, MEAM, and REAX which are not yet included because they require NVIDIA or Fortran compilation). The third includes all standard packages (with the exceptions) and some user packages. The included user packages are USER-EFF, USER-CG-CMM, and USER-REAXC. The fourth project includes the USER-AWPMD package.

2.3 Making SPARTA with optional packages

This section has the following sub-sections:

Package basics:

The source code for SPARTA is structured as a set of core files which are always included, plus optional packages. Packages are groups of files that enable a specific set of features. For example, force fields for molecular systems or granular systems are in packages. You can see the list of all packages by typing "make package" from within the src directory of the SPARTA distribution.

If you use a command in a SPARTA input script that is specific to a particular package, you must have built SPARTA with that package, else you will get an error that the style is invalid or the command is unknown. Every command's doc page specfies if it is part of a package. You can also type

```
lmp_machine -h
```

to run your executable with the optional -h command-line switch for "help", which will list the styles and commands known to your executable.

There are two kinds of packages in SPARTA, standard and user packages. More information about the contents of standard and user packages is given in this section of the manual. The difference between standard and user packages is as follows:

Standard packages are supported by the SPARTA developers and are written in a syntax and style consistent with the rest of SPARTA. This means we will answer questions about them, debug and fix them if necessary, and keep them compatible with future changes to SPARTA.

User packages have been contributed by users, and always begin with the user prefix. If they are a single command (single file), they are typically in the user-misc package. Otherwise, they are a a set of files grouped together which add a specific functionality to the code.

User packages don't necessarily meet the requirements of the standard packages. If you have problems using a feature provided in a user package, you will likely need to contact the contributor directly to get help. Information on how to submit additions you make to SPARTA as a user-contributed package is given in this section of the documentation.

Including/excluding packages:

To use or not use a package you must include or exclude it before building SPARTA. From the src directory, this is typically as simple as:

```
make yes-colloid
make g++
```

or

```
make no-manybody
```

```
make g++
```

Some packages have individual files that depend on other packages being included. SPARTA checks for this and does the right thing. I.e. individual files are only included if their dependencies are already included. Likewise, if a package is excluded, other files dependent on that package are also excluded.

The reason to exclude packages is if you will never run certain kinds of simulations. For some packages, this will keep you from having to build auxiliary libraries (see below), and will also produce a smaller executable which may run a bit faster.

When you download a SPARTA tarball, these packages are pre-installed in the src directory: KSPACE, MANYBODY,MOLECULE. When you download SPARTA source files from the SVN or Git repositories, no packages are pre-installed.

Packages are included or excluded by typing "make yes-name" or "make no-name", where "name" is the name of the package in lower-case, e.g. name = kspace for the KSPACE package or name = user-atc for the USER-ATC package. You can also type "make yes-standard", "make no-standard", "make yes-user", "make no-user", "make yes-all" or "make no-all" to include/exclude various sets of packages. Type "make package" to see the all of the package-related make options.

IMPORTANT NOTE: Inclusion/exclusion of a package works by simply moving files back and forth between the main src directory and sub-directories with the package name (e.g. src/KSPACE, src/USER-ATC), so that the files are seen or not seen when SPARTA is built. After you have included or excluded a package, you must re-build SPARTA.

Additional package-related make options exist to help manage SPARTA files that exist in both the src directory and in package sub-directories. You do not normally need to use these commands unless you are editing SPARTA files or have downloaded a patch from the SPARTA WWW site.

Typing "make package-update" will overwrite src files with files from the package sub-directories if the package has been included. It should be used after a patch is installed, since patches only update the files in the package sub-directory, but not the src files. Typing "make package-overwrite" will overwrite files in the package sub-directories with src files.

Typing "make package-status" will show which packages are currently included. Of those that are included, it will list files that are different in the src directory and package sub-directory. Typing "make package-diff" lists all differences between these files. Again, type "make package" to see all of the package-related make options.

Packages that require extra libraries:

A few of the standard and user packages require additional auxiliary libraries to be compiled first. If you get a SPARTA build error about a missing library, this is likely the reason. The source code for these libraries is included in the SPARTA distribution under the "lib" directory. Look at the lib/README file for a list of these or see this section of the doc pages.

Each lib directory has a README file (e.g. lib/reax/README) with instructions on how to build that library. Typically this is done in this manner:

```
make -f Makefile.g++
```

in the appropriate directory, e.g. in lib/reax. Some of the libraries do not build this way. Again, see the libary README file for details.

In any event, you will need to use a Makefile that is a match for your system. If one of the provided Makefiles is not appropriate for your system you will need to edit or add one. For example, in the case of Fortran-based libraries, your system must have a Fortran compiler, the settings for which will need to be listed in the Makefile.

When you have built one of these libraries, there are 2 things to check:

(1) The file libname.a should now exist in lib/name. E.g. lib/reax/libreax.a. This is the library file SPARTA will link against. One exception is the lib/cuda library which produces the file liblammpscuda.a, because there is already a system library libcuda.a.

(2) The file Makefile.lammps should exist in lib/name. E.g. lib/cuda/Makefile.lammps. This file may be auto-generated by the build of the library, or you may need to make a copy of the appropriate provided file (e.g. lib/meam/Makefile.lammps.gfortran). Either way you should insure that the settings in this file are appropriate for your system.

There are typically 3 settings in the Makefile.lammps file (unless some are blank or not needed): a SYSINC, SYSPATH, and SYSLIB setting, specific to this package. These are settings the SPARTA build will import when compiling the SPARTA package files (not the library files), and linking to the auxiliary library. They typically list any other system libraries needed to support the package and where to find them. An example is the BLAS and LAPACK libraries needed by the USER-ATC package. Or the system libraries that support calling Fortran from C++, as the MEAM and REAX packages do.

Note that if these settings are not correct for your box, the SPARTA build will likely fail.

2.4 Building SPARTA as a library

SPARTA itself can be built as a library, which can then be called from another application or a scripting language. See this SPARTAion for more info on coupling SPARTA to other codes. Building SPARTA as a library is done by typing

```
make makelib
make -f Makefile.lib foo
```

where foo is the machine name. Note that inclusion or exclusion of any desired optional packages should be done before typing "make makelib". The first "make" command will create a current Makefile.lib with all the file names in your src dir. The 2nd "make" command will use it to build SPARTA as a library. This requires that Makefile.foo have a library target (lib) and system-specific settings for ARCHIVE and ARFLAGS. See Makefile.linux for an example. The build will create the file liblmp_foo.a which another application can link to.

When used from a C++ program, the library allows one or more SPARTA objects to be instantiated. All of SPARTA is wrapped in a SPARTA_NS namespace; you can safely use any of its classes and methods from within your application code, as needed.

When used from a C or Fortran program or a scripting language, the library has a simple function-style interface, provided in src/library.cpp and src/library.h.

See the sample codes couple/simple/simple.cpp and simple.c as examples of C++ and C codes that invoke SPARTA thru its library interface. There are other examples as well in the couple directory which are discussed in this section of the manual. See this section of the manual for a description of the Python wrapper provided with SPARTA that operates through the SPARTA library interface.

The files src/library.cpp and library.h contain the C-style interface to SPARTA. See this section of the manual for a description of the interface and how to extend it for your needs.

2.5 Running SPARTA

By default, SPARTA runs by reading commands from stdin; e.g. lmp_linux < in.file. This means you first create an input script (e.g. in.file) containing the desired commands. This section describes how input scripts are structured and what commands they contain.

You can test SPARTA on any of the sample inputs provided in the examples or bench directory. Input scripts are named in.* and sample outputs are named log.*.name.P where name is a machine and P is the number of processors it was run on.

Here is how you might run a standard Lennard-Jones benchmark on a Linux box, using mpirun to launch a parallel job:

```
cd src
make linux
cp lmp_linux ../bench
cd ../bench
mpirun -np 4 lmp_linux <in.lj
```

See this page for timings for this and the other benchmarks on various platforms.

On a Windows box, you can skip making SPARTA and simply download an executable, as described above, though the pre-packaged executables include only certain packages.

To run a SPARTA executable on a Windows machine, first decide whether you want to download the non-MPI (serial) or the MPI (parallel) version of the executable. Download and save the version you have chosen.

For the non-MPI version, follow these steps:

- Get a command prompt by going to Start->Run... , then typing "cmd".
- Move to the directory where you have saved lmp_win_no-mpi.exe (e.g. by typing: cd "Documents").
- At the command prompt, type "lmp_win_no-mpi -in in.lj", replacing in.lj with the name of your SPARTA input script.

For the MPI version, which allows you to run SPARTA under Windows on multiple processors, follow these steps:

- Download and install MPICH2 for Windows.
- You'll need to use the mpiexec.exe and smpd.exe files from the MPICH2 package. Put them in same directory (or path) as the SPARTA Windows executable.
- Get a command prompt by going to Start->Run... , then typing "cmd".
- Move to the directory where you have saved lmp_win_mpi.exe (e.g. by typing: cd "Documents").
- Then type something like this: "mpiexec -np 4 -localonly lmp_win_mpi -in in.lj", replacing in.lj with the name of your SPARTA input script.
- Note that you may need to provide smpd with a passphrase --- it doesn't matter what you type.
- In this mode, output may not immediately show up on the screen, so if your input script takes a long time to execute, you may need to be patient before the output shows up.
- Alternatively, you can still use this executable to run on a single processor by typing something like: "lmp_win_mpi -in in.lj".

The screen output from SPARTA is described in the next section. As it runs, SPARTA also writes a log.lammps file with the same information.

Note that this sequence of commands copies the SPARTA executable (lmp_linux) to the directory with the input files. This may not be necessary, but some versions of MPI reset the working directory to where the executable is, rather than leave it as the directory where you launch mpirun from (if you launch lmp_linux on its own and not under mpirun). If that happens, SPARTA will look for additional input files and write its output files to the executable directory, rather than your working directory, which is probably not what you want.

If SPARTA encounters errors in the input script or while running a simulation it will print an ERROR message and stop or a WARNING message and continue. See this section for a discussion of the various kinds of errors SPARTA can or can't detect, a list of all ERROR and WARNING messages, and what to do about them.

SPARTA can run a problem on any number of processors, including a single processor. In theory you should get identical answers on any number of processors and on any machine. In practice, numerical round-off can cause slight differences and eventual divergence of molecular dynamics phase space trajectories.

SPARTA can run as large a problem as will fit in the physical memory of one or more processors. If you run out of memory, you must run on more processors or setup a smaller problem.

2.6 Command-line options

At run time, SPARTA recognizes several optional command-line switches which may be used in any order. Either the full word or a one-or-two letter abbreviation can be used:

- -c or -cuda
- -e or -echo
- -i or -in
- -h or -help
- -l or -log
- -p or -partition
- -pl or -plog
- -ps or -pscreen
- -sc or -screen
- -sf or -suffix
- -v or -var

For example, lmp_ibm might be launched as follows:

```
mpirun -np 16 lmp_ibm -v f tmp.out -l my.log -sc none <in.alloy
mpirun -np 16 lmp_ibm -var f tmp.out -log my.log -screen none  <in.alloy
```

Here are the details on the options:

```
-cuda on/off
```

Explicitly enable or disable CUDA support, as provided by the USER-CUDA package. If SPARTA is built with this package, as described above in Section 2.3, then by default SPARTA will run in CUDA mode. If this switch is set to "off", then it will not, even if it was built with the USER-CUDA package, which means you can run standard SPARTA or with the GPU package for testing or benchmarking purposes. The only reason to set the switch to "on", is to check if SPARTA was built with the USER-CUDA package, since an error will be generated if it was not.

```
-echo style
```

Set the style of command echoing. The style can be *none* or *screen* or *log* or *both*. Depending on the style, each command read from the input script will be echoed to the screen and/or logfile. This can be useful to figure out which line of your script is causing an input error. The default value is *log*. The echo style can also be set by using the echo command in the input script itself.

```
-in file
```

Specify a file to use as an input script. This is an optional switch when running SPARTA in one-partition mode. If it is not specified, SPARTA reads its input script from stdin - e.g. lmp_linux < in.run. This is a required switch when running SPARTA in multi-partition mode, since multiple processors cannot all read from stdin.

```
-help
```

Print a list of options compiled into this executable for each SPARTA style (atom_style, fix, compute, pair_style, bond_style, etc). This can help you know if the command you want to use was included via the appropriate package. SPARTA will print the info and immediately exit if this switch is used.

```
-log file
```

Specify a log file for SPARTA to write status information to. In one-partition mode, if the switch is not used, SPARTA writes to the file log.lammps. If this switch is used, SPARTA writes to the specified file. In multi-partition mode, if the switch is not used, a log.lammps file is created with hi-level status information. Each partition also writes to a log.lammps.N file where N is the partition ID. If the switch is specified in multi-partition mode, the hi-level logfile is named "file" and each partition also logs information to a file.N. For both one-partition and multi-partition mode, if the specified file is "none", then no log files are created. Using a log command in the input script will override this setting. Option -plog will override the name of the partition log files file.N.

```
-partition 8x2 4 5 ...
```

Invoke SPARTA in multi-partition mode. When SPARTA is run on P processors and this switch is not used, SPARTA runs in one partition, i.e. all P processors run a single simulation. If this switch is used, the P processors are split into separate partitions and each partition runs its own simulation. The arguments to the switch specify the number of processors in each partition. Arguments of the form MxN mean M partitions, each with N processors. Arguments of the form N mean a single partition with N processors. The sum of processors in all partitions must equal P. Thus the command "-partition 8x2 4 5" has 10 partitions and runs on a total of 25 processors.

Note that with MPI installed on a machine (e.g. your desktop), you can run on more (virtual) processors than you have physical processors. This can be useful for running multi-replica simulations, on one or a few processors.

The input script specifies what simulation is run on which partition; see the variable and next commands. This howto section gives examples of how to use these commands in this way. Simulations running on different partitions can also communicate with each other; see the temper command.

```
-plog file
```

Specify the base name for the partition log files, so partition N writes log information to file.N. If file is none, then no partition log files are created. This overrides the filename specified in the -log command-line option. This option is useful when working with large numbers of partitions, allowing the partition log files to be suppressed (-plog none) or placed in a sub-directory (-plog replica_files/log.lammps) If this option is not used the log file for partition N is log.lammps.N or whatever is specified by the -log command-line option.

```
-pscreen file
```

Specify the base name for the partition screen file, so partition N writes screen information to file.N. If file is none, then no partition screen files are created. This overrides the filename specified in the -screen command-line option. This option is useful when working with large numbers of partitions, allowing the partition screen files to be suppressed (-pscreen none) or placed in a sub-directory (-pscreen replica_files/screen) If this option is not used the screen file for partition N is screen.N or whatever is specified by the -screen command-line option.

```
-screen file
```

Specify a file for SPARTA to write its screen information to. In one-partition mode, if the switch is not used, SPARTA writes to the screen. If this switch is used, SPARTA writes to the specified file instead and you will see no screen output. In multi-partition mode, if the switch is not used, hi-level status information is written to the screen. Each partition also writes to a screen.N file where N is the partition ID. If the switch is specified in multi-partition mode, the hi-level screen dump is named "file" and each partition also writes screen information to a file.N. For both one-partition and multi-partition mode, if the specified file is "none", then no screen output is performed. Option -pscreen will override the name of the partition screen files file.N.

```
-suffix style
```

Use variants of various styles if they exist. The specified style can be *opt* or *gpu* or *cuda*. These refer to optional packages that SPARTA can be built with, as described above in Section 2.3. The "opt" style corrsponds to the OPT package, the "gpu" style to the GPU package, and the "cuda" style to the USER-CUDA package.

As an example, all of the packages provide a pair_style lj/cut variant, with style names lj/cut/opt or lj/cut/gpu or lj/cut/cuda. A variant styles can be specified explicitly in your input script, e.g. pair_style lj/cut/gpu. If the -suffix switch is used, you do not need to modify your input script. The specified suffix (opt,gpu,cuda) is automatically appended whenever your input script command creates a new atom, pair, fix, compute, or run style. atom, pair, fix, compute, or integrate style. If the variant version does not exist, the standard version is created.

The suffix command can also set a suffix and it can also turn off/on any suffix setting made via the command line.

```
-var name value1 value2 ...
```

Specify a variable that will be defined for substitution purposes when the input script is read. "Name" is the variable name which can be a single character (referenced as $x in the input script) or a full string (referenced as ${abc}). An index-style variable will be created and populated with the subsequent values, e.g. a set of filenames. Using this command-line option is equivalent to putting the line "variable name index value1 value2 ..." at the beginning of the input script. Defining an index variable as a command-line argument overrides any setting for the same index variable in the input script, since index variables cannot be re-defined. See the variable command for more info on defining index and other kinds of variables and this section for more info on using variables in input scripts.

2.7 SPARTA screen output

As SPARTA reads an input script, it prints information to both the screen and a log file about significant actions it takes to setup a simulation. When the simulation is ready to begin, SPARTA performs various initializations and prints the amount of memory (in MBytes per processor) that the simulation requires. It also prints details of the initial thermodynamic state of the system. During the run itself, thermodynamic information is printed periodically, every few timesteps. When the run concludes, SPARTA prints the final thermodynamic state and a total run time for the simulation. It then appends statistics about the CPU time and storage requirements for the simulation. An example set of statistics is shown here:

```
Loop time of 49.002 on 2 procs for 2004 atoms
```

```
Pair   time (%) = 35.0495 (71.5267)
Bond   time (%) = 0.092046 (0.187841)
Kspce  time (%) = 6.42073 (13.103)
Neigh  time (%) = 2.73485 (5.5811)
Comm   time (%) = 1.50291 (3.06703)
Outpt  time (%) = 0.013799 (0.0281601)
Other  time (%) = 2.13669 (4.36041)

Nlocal:    1002 ave, 1015 max, 989 min
Histogram: 1 0 0 0 0 0 0 0 0 1
Nghost:    8720 ave, 8724 max, 8716 min
Histogram: 1 0 0 0 0 0 0 0 0 1
Neighs:    354141 ave, 361422 max, 346860 min
Histogram: 1 0 0 0 0 0 0 0 0 1

Total # of neighbors = 708282
Ave neighs/atom = 353.434
Ave special neighs/atom = 2.34032
Number of reneighborings = 42
Dangerous reneighborings = 2
```

The first section gives the breakdown of the CPU run time (in seconds) into major categories. The second section lists the number of owned atoms (Nlocal), ghost atoms (Nghost), and pair-wise neighbors stored per processor. The max and min values give the spread of these values across processors with a 10-bin histogram showing the distribution. The total number of histogram counts is equal to the number of processors.

The last section gives aggregate statistics for pair-wise neighbors and special neighbors that SPARTA keeps track of (see the special_bonds command). The number of times neighbor lists were rebuilt during the run is given as well as the number of potentially "dangerous" rebuilds. If atom movement triggered neighbor list rebuilding (see the neigh_modify command), then dangerous reneighborings are those that were triggered on the first timestep atom movement was checked for. If this count is non-zero you may wish to reduce the delay factor to insure no force interactions are missed by atoms moving beyond the neighbor skin distance before a rebuild takes place.

If an energy minimization was performed via the minimize command, additional information is printed, e.g.

```
Minimization stats:
  E initial, next-to-last, final = -0.895962 -2.94193 -2.94342
  Gradient 2-norm init/final= 1920.78 20.9992
  Gradient inf-norm init/final= 304.283 9.61216
  Iterations = 36
  Force evaluations = 177
```

The first line lists the initial and final energy, as well as the energy on the next-to-last iteration. The next 2 lines give a measure of the gradient of the energy (force on all atoms). The 2-norm is the "length" of this force vector; the inf-norm is the largest component. The last 2 lines are statistics on how many iterations and force-evaluations the minimizer required. Multiple force evaluations are typically done at each iteration to perform a 1d line minimization in the search direction.

If a kspace_style long-range Coulombics solve was performed during the run (PPPM, Ewald), then additional information is printed, e.g.

```
FFT time (% of Kspce) = 0.200313 (8.34477)
FFT Gflps 3d 1d-only = 2.31074 9.19989
```

The first line gives the time spent doing 3d FFTs (4 per timestep) and the fraction it represents of the total KSpace time (listed above). Each 3d FFT requires computation (3 sets of 1d FFTs) and communication (transposes). The total flops performed is 5Nlog_2(N), where N is the number of points in the 3d grid. The FFTs are timed with and

without the communication and a Gflop rate is computed. The 3d rate is with communication; the 1d rate is without (just the 1d FFTs). Thus you can estimate what fraction of your FFT time was spent in communication, roughly 75% in the example above.

2.8 Tips for users of previous SPARTA versions

The current C++ began with a complete rewrite of SPARTA 2001, which was written in F90. Features of earlier versions of SPARTA are listed in this section. The F90 and F77 versions (2001 and 99) are also freely distributed as open-source codes; check the SPARTA WWW Site for distribution information if you prefer those versions. The 99 and 2001 versions are no longer under active development; they do not have all the features of C++ SPARTA.

If you are a previous user of SPARTA 2001, these are the most significant changes you will notice in C++ SPARTA:

(1) The names and arguments of many input script commands have changed. All commands are now a single word (e.g. read_data instead of read data).

(2) All the functionality of SPARTA 2001 is included in C++ SPARTA, but you may need to specify the relevant commands in different ways.

(3) The format of the data file can be streamlined for some problems. See the read_data command for details. The data file section "Nonbond Coeff" has been renamed to "Pair Coeff" in C++ SPARTA.

(4) Binary restart files written by SPARTA 2001 cannot be read by C++ SPARTA with a read_restart command. This is because they were output by F90 which writes in a different binary format than C or C++ writes or reads. Use the *restart2data* tool provided with SPARTA 2001 to convert the 2001 restart file to a text data file. Then edit the data file as necessary before using the C++ SPARTA read_data command to read it in.

(5) There are numerous small numerical changes in C++ SPARTA that mean you will not get identical answers when comparing to a 2001 run. However, your initial thermodynamic energy and MD trajectory should be close if you have setup the problem for both codes the same.

3. Commands

This section describes how a SPARTA input script is formatted and what commands are used to define a SPARTA simulation.

3.1 SPARTA input script

SPARTA executes by reading commands from a input script (text file), one line at a time. When the input script ends, SPARTA exits. Each command causes SPARTA to take some action. It may set an internal variable, read in a file, or run a simulation. Most commands have default settings, which means you only need to use the command if you wish to change the default.

In many cases, the ordering of commands in an input script is not important. However the following rules apply:

(1) SPARTA does not read your entire input script and then perform a simulation with all the settings. Rather, the input script is read one line at a time and each command takes effect when it is read. Thus this sequence of commands:

```
timestep 0.5
run      100
run      100
```

does something different than this sequence:

```
run      100
timestep 0.5
run      100
```

In the first case, the specified timestep (0.5 fmsec) is used for two simulations of 100 timesteps each. In the 2nd case, the default timestep (1.0 fmsec) is used for the 1st 100 step simulation and a 0.5 fmsec timestep is used for the 2nd one.

(2) Some commands are only valid when they follow other commands. For example you cannot set the temperature of a group of atoms until atoms have been defined and a group command is used to define which atoms belong to the group.

(3) Sometimes command B will use values that can be set by command A. This means command A must precede command B in the input script if it is to have the desired effect. For example, the read_data command initializes the system by setting up the simulation box and assigning atoms to processors. If default values are not desired, the processors and boundary commands need to be used before read_data to tell SPARTA how to map processors to the simulation box.

Many input script errors are detected by SPARTA and an ERROR or WARNING message is printed. This section gives more information on what errors mean. The documentation for each command lists restrictions on how the

command can be used.

3.2 Parsing rules

Each non-blank line in the input script is treated as a command. SPARTA commands are case sensitive. Command names are lower-case, as are specified command arguments. Upper case letters may be used in file names or user-chosen ID strings.

Here is how each line in the input script is parsed by SPARTA:

(1) If the last printable character on the line is a "&" character (with no surrounding quotes), the command is assumed to continue on the next line. The next line is concatenated to the previous line by removing the "&" character and newline. This allows long commands to be continued across two or more lines.

(2) All characters from the first "#" character onward are treated as comment and discarded. See an exception in (6). Note that a comment after a trailing "&" character will prevent the command from continuing on the next line. Also note that for multi-line commands a single leading "#" will comment out the entire command.

(3) The line is searched repeatedly for $ characters, which indicate variables that are replaced with a text string. See an exception in (6). If the $ is followed by curly brackets, then the variable name is the text inside the curly brackets. If no curly brackets follow the $, then the variable name is the single character immediately following the $. Thus ${myTemp} and $x refer to variable names "myTemp" and "x". See the variable command for details of how strings are assigned to variables and how they are substituted for in input script commands.

(4) The line is broken into "words" separated by whitespace (tabs, spaces). Note that words can thus contain letters, digits, underscores, or punctuation characters.

(5) The first word is the command name. All successive words in the line are arguments.

(6) If you want text with spaces to be treated as a single argument, it can be enclosed in either double or single quotes. E.g.

```
print "Volume = $v"
print 'Volume = $v'
```

The quotes are removed when the single argument is stored internally. See the dump modify format or if commands for examples. A "#" or "$" character that is between quotes will not be treated as a comment indicator in (2) or substituted for as a variable in (3).

IMPORTANT NOTE: If the argument is itself a command that requires a quoted argument (e.g. using a print command as part of an if or run every command), then the double and single quotes can be nested in the usual manner. See the doc pages for those commands for examples. Only one of level of nesting is allowed, but that should be sufficient for most use cases.

3.3 Input script structure

This section describes the structure of a typical SPARTA input script. The "examples" directory in the SPARTA distribution contains many sample input scripts; the corresponding problems are discussed in this section, and animated on the SPARTA WWW Site.

A SPARTA input script typically has 4 parts:

1. Initialization
2. Problem definition
3. Settings
4. Run a simulation

The last 2 parts can be repeated as many times as desired. I.e. run a simulation, change some settings, run some more, etc. Each of the 4 parts is now described in more detail. Remember that almost all the commands need only be used if a non-default value is desired.

(1) Initialization

Set parameters that need to be defined before atoms are created or read-in from a file.

The relevant commands are dimension

(2) Problem definition

These items must be setup to run a SPARTA calculation:

- simulation box via create_box
- grid via create_grid
- particles via create_molecules

(3) Settings

Once particles and the grid are defined, a variety of settings can be specified: simulation parameters, output options, etc.

Various simulation parameters are set by these commands: timestep

Output is set by these commands:

(4) Run a simulation

A simulation is run using the run command.

3.4 Commands listed by category

This section lists all SPARTA commands, grouped by category. The next section lists the same commands alphabetically. Note that some style options for some commands are part of specific SPARTA packages, which means they cannot be used unless the package was included when SPARTA was built. Not all packages are included in a default SPARTA build. These dependencies are listed as Restrictions in the command's documentation.

Initialization:

dimension

Problem definition:

create_box, create_grid, create_molecules, species

Settings:

timestep, velocity

Actions:

run

Miscellaneous:

clear, echo, if, include, jump, label, log, next, print, shell, variable

---

3.5 Individual commands

This section lists all SPARTA commands alphabetically, with a separate listing below of styles within certain commands. The previous section lists the same commands, grouped by category. Note that some style options for some commands are part of specific SPARTA packages, which means they cannot be used unless the package was included when SPARTA was built. Not all packages are included in a default SPARTA build. These dependencies are listed as Restrictions in the command's documentation.

boundary	bound_modify	clear	collide	compute	create_molecules
create_box	create_grid	dimension	dump	dump_image	dump_modify
echo	fix	global	if	include	jump
label	log	mixture	next	partition	print
read_surf	restart	run	seed	shell	species
stats	stats_modify	stats_style	surf_collide	timestep	uncompute
undump	unfix	units	variable	write_restart	

Fix styles

See the fix command for one-line descriptions of each style or click on the style itself for a full description:

ave/grid	ave/surf	ave/time	grid/check	inflow	print

Compute styles

See the compute command for one-line descriptions of each style or click on the style itself for a full description:

grid	ke/molecule	sonine/grid	temp

Collide styles

See the collide command for one-line descriptions of each style or click on the style itself for a full description:

vss

Surf collide styles

See the surf_collide command for one-line descriptions of each style or click on the style itself for a full description:

diffuse

6. How-to discussions

The following sections describe how to use various options within SPARTA.

The example input scripts included in the SPARTA distribution and highlighted in this section also show how to setup and run various kinds of simulations.

7. Example problems

The SPARTA distribution includes an examples sub-directory with several sample problems. Each problem is in a sub-directory of its own. Most are 2d models so that they run quickly, requiring at most a couple of minutes to run on a desktop machine. Each problem has an input script (in.*) and produces a log file (log.*) and dump file (dump.*) when it runs. Some use a data file (data.*) of initial coordinates as additional input. A few sample log file outputs on different machines and different numbers of processors are included in the directories to compare your answers to. E.g. a log file like log.crack.foo.P means it ran on P processors of machine "foo".

The dump files produced by the example runs can be animated using the xmovie tool described in the Additional Tools section of the SPARTA documentation. Animations of many of these examples can be viewed on the Movies section of the SPARTA WWW Site.

These are the sample problems in the examples sub-directories:

colloid	big colloid particles in a small particle solvent, 2d system
comb	models using the COMB potential
crack	crack propagation in a 2d solid
dipole	point dipolar particles, 2d system
eim	NaCl using the EIM potential
ellipse	ellipsoidal particles in spherical solvent, 2d system
flow	Couette and Poiseuille flow in a 2d channel
friction	frictional contact of spherical asperities between 2d surfaces
indent	spherical indenter into a 2d solid
meam	MEAM test for SiC and shear (same as shear examples)
melt	rapid melt of 3d LJ system
micelle	self-assembly of small lipid-like molecules into 2d bilayers
min	energy minimization of 2d LJ melt
msst	MSST shock dynamics
neb	nudged elastic band (NEB) calculation for barrier finding
nemd	non-equilibrium MD of 2d sheared system
obstacle	flow around two voids in a 2d channel
peptide	dynamics of a small solvated peptide chain (5-mer)
peri	Peridynamic model of cylinder impacted by indenter
pour	pouring of granular particles into a 3d box, then chute flow
prd	parallel replica dynamics of a vacancy diffusion in bulk Si
reax	RDX and TATB models using the ReaxFF
rigid	rigid bodies modeled as independent or coupled
shear	sideways shear applied to 2d solid, with and without a void
srd	stochastic rotation dynamics (SRD) particles as solvent

Here is how you might run and visualize one of the sample problems:

```
cd indent
cp ../../src/lmp_linux .           # copy SPARTA executable to this dir
lmp_linux <in.indent               # run the problem
```

Running the simulation produces the files *dump.indent* and *log.lammps*. You can visualize the dump file as follows:

```
../../tools/xmovie/xmovie -scale dump.indent
```

There is also an ELASTIC directory with an example script for computing elastic constants, using a zero temperature Si example. See the in.elastic file for more info.

There is also a USER directory which contains subdirectories of user-provided examples for user packages. See the README files in those directories for more info. See the doc/Section_start.html file for more info about user packages.

8. Performance & scalability

SPARTA performance on several prototypical benchmarks and machines is discussed on the Benchmarks page of the SPARTA WWW Site where CPU timings and parallel efficiencies are listed. Here, the benchmarks are described briefly and some useful rules of thumb about their performance are highlighted.

These are the 5 benchmark problems:

1. LJ = atomic fluid, Lennard-Jones potential with 2.5 sigma cutoff (55 neighbors per atom), NVE integration
2. Chain = bead-spring polymer melt of 100-mer chains, FENE bonds and LJ pairwise interactions with a 2^(1/6) sigma cutoff (5 neighbors per atom), NVE integration
3. EAM = metallic solid, Cu EAM potential with 4.95 Angstrom cutoff (45 neighbors per atom), NVE integration
4. Chute = granular chute flow, frictional history potential with 1.1 sigma cutoff (7 neighbors per atom), NVE integration
5. Rhodo = rhodopsin protein in solvated lipid bilayer, CHARMM force field with a 10 Angstrom LJ cutoff (440 neighbors per atom), particle-particle particle-mesh (PPPM) for long-range Coulombics, NPT integration

The input files for running the benchmarks are included in the SPARTA distribution, as are sample output files. Each of the 5 problems has 32,000 atoms and runs for 100 timesteps. Each can be run as a serial benchmarks (on one processor) or in parallel. In parallel, each benchmark can be run as a fixed-size or scaled-size problem. For fixed-size benchmarking, the same 32K atom problem is run on various numbers of processors. For scaled-size benchmarking, the model size is increased with the number of processors. E.g. on 8 processors, a 256K-atom problem is run; on 1024 processors, a 32-million atom problem is run, etc.

A useful metric from the benchmarks is the CPU cost per atom per timestep. Since SPARTA performance scales roughly linearly with problem size and timesteps, the run time of any problem using the same model (atom style, force field, cutoff, etc) can then be estimated. For example, on a 1.7 GHz Pentium desktop machine (Intel icc compiler under Red Hat Linux), the CPU run-time in seconds/atom/timestep for the 5 problems is

Problem:	LJ	Chain	EAM	Chute	Rhodopsin
CPU/atom/step:	4.55E-6	2.18E-6	9.38E-6	2.18E-6	1.11E-4
Ratio to LJ:	1.0	0.48	2.06	0.48	24.5

The ratios mean that if the atomic LJ system has a normalized cost of 1.0, the bead-spring chains and granular systems run 2x faster, while the EAM metal and solvated protein models run 2x and 25x slower respectively. The bulk of these cost differences is due to the expense of computing a particular pairwise force field for a given number of neighbors per atom.

Performance on a parallel machine can also be predicted from the one-processor timings if the parallel efficiency can be estimated. The communication bandwidth and latency of a particular parallel machine affects the efficiency. On most machines SPARTA will give fixed-size parallel efficiencies on these benchmarks above 50% so long as the atoms/processor count is a few 100 or greater - i.e. on 64 to 128 processors. Likewise, scaled-size parallel efficiencies will typically be 80% or greater up to very large processor counts. The benchmark data on the SPARTA WWW Site gives specific examples on some different machines, including a run of 3/4 of a billion LJ atoms on 1500 processors that ran at 85% parallel efficiency.

9. Additional tools

SPARTA is designed to be a computational kernel for performing molecular dynamics computations. Additional pre- and post-processing steps are often necessary to setup and analyze a simulation. A few additional tools are provided with the SPARTA distribution and are described in this section.

Our group has also written and released a separate toolkit called Pizza.py which provides tools for doing setup, analysis, plotting, and visualization for SPARTA simulations. Pizza.py is written in Python and is available for download from the Pizza.py WWW site.

Note that many users write their own setup or analysis tools or use other existing codes and convert their output to a SPARTA input format or vice versa. The tools listed here are included in the SPARTA distribution as examples of auxiliary tools. Some of them are not actively supported by Sandia, as they were contributed by SPARTA users. If you have problems using them, we can direct you to the authors.

The source code for each of these codes is in the tools sub-directory of the SPARTA distribution. There is a Makefile (which you may need to edit for your platform) which will build several of the tools which reside in that directory. Some of them are larger packages in their own sub-directories with their own Makefiles.

10. Modifying & extending SPARTA

SPARTA is designed in a modular fashion so as to be easy to modify and extend with new functionality. In fact, about 75% of its source code is files added in this fashion.

In this section, changes and additions users can make are listed along with minimal instructions. If you add a new feature to SPARTA and think it will be of interest to general users, we encourage you to submit it to the developers for inclusion in the released version of SPARTA. Information about how to do this is provided below.

The best way to add a new feature is to find a similar feature in SPARTA and look at the corresponding source and header files to figure out what it does. You will need some knowledge of C++ to be able to understand the hi-level structure of SPARTA and its class organization, but functions (class methods) that do actual computations are written in vanilla C-style code and operate on simple C-style data structures (vectors and arrays).

Most of the new features described in this section require you to write a new C++ derived class (except for exceptions described below, where you can make small edits to existing files). Creating a new class requires 2 files, a source code file (*.cpp) and a header file (*.h). The derived class must provide certain methods to work as a new option. Depending on how different your new feature is compared to existing features, you can either derive from the base class itself, or from a derived class that already exists. Enabling SPARTA to invoke the new class is as simple as putting the two source files in the src dir and re-building SPARTA.

The advantage of C++ and its object-orientation is that all the code and variables needed to define the new feature are in the 2 files you write, and thus shouldn't make the rest of SPARTA more complex or cause side-effect bugs.

Here is a concrete example. Suppose you write 2 files pair_foo.cpp and pair_foo.h that define a new class PairFoo that computes pairwise potentials described in the classic 1997 paper by Foo, et al. If you wish to invoke those potentials in a SPARTA input script with a command like

```
pair_style foo 0.1 3.5
```

then your pair_foo.h file should be structured as follows:

```
#ifdef PAIR_CLASS
PairStyle(foo,PairFoo)
#else
...
(class definition for PairFoo)
...
#endif
```

where "foo" is the style keyword in the pair_style command, and PairFoo is the class name defined in your pair_foo.cpp and pair_foo.h files.

When you re-build SPARTA, your new pairwise potential becomes part of the executable and can be invoked with a pair_style command like the example above. Arguments like 0.1 and 3.5 can be defined and processed by your new class.

As illustrated by this pairwise example, many kinds of options are referred to in the SPARTA documentation as the "style" of a particular command.

The instructions below give the header file for the base class that these styles are derived from. Public variables in that file are ones used and set by the derived classes which are also used by the base class. Sometimes they are also used by the rest of SPARTA. Virtual functions in the base class header file which are set = 0 are ones you must define in your new derived class to give it the functionality SPARTA expects. Virtual functions that are not set to 0 are functions you can optionally define.

Additionally, new output options can be added directly to the thermo.cpp, dump_custom.cpp, and variable.cpp files as explained below.

Here are additional guidelines for modifying SPARTA and adding new functionality:

- Think about whether what you want to do would be better as a pre- or post-processing step. Many computations are more easily and more quickly done that way.
- Don't do anything within the timestepping of a run that isn't parallel. E.g. don't accumulate a bunch of data on a single processor and analyze it. You run the risk of seriously degrading the parallel efficiency.
- If your new feature reads arguments or writes output, make sure you follow the unit conventions discussed by the units command.
- If you add something you think is truly useful and doesn't impact SPARTA performance when it isn't used, send an email to the developers. We might be interested in adding it to the SPARTA distribution. See further details on this at the bottom of this page.

Here are the subsequent topics discussed below, most of which are new features that can be added in the manner just described:

10.14 Submitting new features for inclusion in SPARTA

We encourage users to submit new features that they add to SPARTA to the developers, especially if you think the features will be of interest to other users. If they are broadly useful we may add them as core files to SPARTA or as part of a standard package. Else we will add them as a user-contributed package or file. Examples of user packages are in src sub-directories that start with USER. The USER-MISC package is simply a collection of (mostly) unrelated single files, which is the simplest way to have your contribution quickly added to the SPARTA distribution. You can see a list of the both standard and user packages by typing "make package" in the SPARTA src directory.

With user packages and files, all we are really providing (aside from the fame and fortune that accompanies having your name in the source code and on the Authors page of the SPARTA WWW site), is a means for you to distribute your SPARTA to the SPARTA user community and a mechanism for others to easily try out your new feature. This may help you find bugs or make contact with new collaborators. Note that you're also implicitly agreeing to support your code which means answer questions, fix bugs, and maintain it if SPARTA changes.

The previous sections of this doc page describe how to add new features of various kinds to SPARTA. Packages are simply collections of one or more new class files which are invoked as a new "style" within a SPARTA input script. If designed correctly, these additions do not require changes to the main core of SPARTA; they are simply add-on files. If you think your new feature requires non-trivial changes in core SPARTA files, you'll need to communicate with the developers, since we may or may not want to make those changes. An example of a trivial change is making a parent-class method "virtual" when you derive a new child class from it.

Here is what you need to do to submit a user package or single file for our consideration. Following these steps will save time for both you and us. See existing package files for examples.

- All source files you provide must compile with the most current version of SPARTA.
- If your contribution is a single file (actually a *.cpp and *.h file) it can most rapidly be added to the USER-MISC directory. Send us the one-line entry to add to the USER-MISC/README file in that dir, along with the 2 source files. You can do this multiple times if you wish to contribute several individual features.
- If your contribution is several related featues, it is probably best to make it a user package directory with a name like USER-FOO. In addition to your new files, the directory should contain a README, and Install.csh file. The README text file should contain your name and contact information and a brief description of what your new package does. The Install.csh file enables SPARTA to include and exclude your package. See other README and Install.sh files in other USER directories as examples. Send us a tarball of this USER-FOO directory.
- Your new source files need to have the SPARTA copyright, GPL notice, and your name at the top, like other SPARTA source files. They need to create a class that is inside the SPARTA namespace. Other than that, your files can do whatever is necessary to implement the new features. They don't have to be written in the same stylistic format and syntax as other SPARTA files, though that would be nice.
- Finally, you must also send a documentation file for each new command or style you are adding to SPARTA. This will be one file for a single-file feature. For a package, it might be several files. These are simple text files which we will convert to HTML. They must be in the same format as other *.txt files in the lammps/doc directory for similar commands and styles. The "Restrictions" section of the doc page should indicate that your command is only available if SPARTA is built with the appropriate USER-MISC or USER-FOO package. See other user package doc files for an example of how to do this. The txt2html tool we use to do the conversion can be downloaded from this site, so you can perform the HTML conversion yourself to proofread your doc page.

Note that the more clear and self-explanatory you make your doc and README files, the more likely it is that users will try out your new feature.

11. Python interface to SPARTA

The SPARTA distribution includes some Python code in its python directory which wraps the library interface to SPARTA. This makes it is possible to run SPARTA, invoke SPARTA commands or give it an input script, extract SPARTA results, an modify internal SPARTA variables, either from a Python script or interactively from a Python prompt.

Python is a powerful scripting and programming language which can be used to wrap software like SPARTA and other packages. It can be used to glue multiple pieces of software together, e.g. to run a coupled or multiscale model. See this section of the manual and the couple directory of the distribution for more ideas about coupling SPARTA to other codes. See this section about how to build SPARTA as a library, and this section for a description of the library interface provided in src/library.cpp and src/library.h and how to extend it for your needs. As described below, that interface is what is exposed to Python. It is designed to be easy to add functions to. This has the effect of extending the Python inteface as well. See details below.

By using the Python interface SPARTA can also be coupled with a GUI or visualization tools that display graphs or animations in real time as SPARTA runs. Examples of such scripts are inlcluded in the python directory.

Two advantages of using Python are how concise the language is and that it can be run interactively, enabling rapid development and debugging of programs. If you use it to mostly invoke costly operations within SPARTA, such as running a simulation for a reasonable number of timesteps, then the overhead cost of invoking SPARTA thru Python will be negligible.

Before using SPARTA from a Python script, the Python on your machine must be "extended" to include an interface to the SPARTA library. If your Python script will invoke MPI operations, you will also need to extend your Python with an interface to MPI itself.

Thus you should first decide how you intend to use SPARTA from Python. There are 3 options:

(1) Use SPARTA on a single processor running Python.

(2) Use SPARTA in parallel, where each processor runs Python, but your Python program does not use MPI.

(3) Use SPARTA in parallel, where each processor runs Python, and your Python script also makes MPI calls through a Python/MPI interface.

Note that for (2) and (3) you will not be able to use Python interactively by typing commands and getting a response. This is because you will have multiple instances of Python running (e.g. on a parallel machine) and they cannot all read what you type.

Working in mode (1) does not require your machine to have MPI installed. You should extend your Python with a serial version of SPARTA and the dummy MPI library provided with SPARTA. See instructions below on how to do this.

Working in mode (2) requires your machine to have an MPI library installed, but your Python does not need to be extended with MPI itself. The MPI library must be a shared library (e.g. a *.so file on Linux) which is not typically created when MPI is built/installed. See instruction below on how to do this. You should extend your Python with the a parallel versionn of SPARTA which will use the shared MPI system library. See instructions below on how to do this.

Working in mode (3) requires your machine to have MPI installed (as a shared library as in (2)). You must also extend your Python with a parallel version of SPARTA (same as in (2)) and with MPI itself, via one of several available Python/MPI packages. See instructions below on how to do the latter task.

Several of the following sub-sections cover the rest of the Python setup discussion. The next to last sub-section describes the Python syntax used to invoke SPARTA. The last sub-section describes example Python scripts included in the python directory.

Before proceeding, there are 2 items to note.

(1) The provided Python wrapper for SPARTA uses the amazing and magical (to me) "ctypes" package in Python, which auto-generates the interface code needed between Python and a set of C interface routines for a library. Ctypes is part of standard Python for versions 2.5 and later. You can check which version of Python you have installed, by simply typing "python" at a shell prompt.

(2) Any library wrapped by Python, including SPARTA, must be built as a shared library (e.g. a *.so file on Linux and not a *.a file). The python/setup_serial.py and setup.py scripts do this build for SPARTA itself (described below). But if you have SPARTA configured to use additional packages that have their own libraries, then those libraries must also be shared libraries. E.g. MPI, FFTW, or any of the libraries in lammps/lib. When you build SPARTA as a stand-alone code, you are not building shared versions of these libraries.

The discussion below describes how to create a shared MPI library. I suggest you start by configuing SPARTA without packages installed that require any libraries besides MPI. See this section of the manual for a discussion of SPARTA pacakges. E.g. do not use the KSPACE, GPU, MEAM, POEMS, or REAX packages.

If you are successfully follow the steps belwo to build the Python wrappers and use this version of SPARTA through Python, you can then take the next step of adding SPARTA packages that use additional libraries. This will require you to build a shared library for that package's library, similar to what is described below for MPI. It will also require you to edit the python/setup_serial.py or setup.py scripts to enable Python to access those libraries when it builds the SPARTA wrapper.

11.1 Extending Python with a serial version of SPARTA

From the python directory in the SPARTA distribution, type

```
python setup_serial.py build
```

and then one of these commands:

```
sudo python setup_serial.py install
python setup_serial.py install --home=~/foo
```

The "build" command should compile all the needed SPARTA files, including its dummy MPI library. The first "install" command will put the needed files in your Python's site-packages sub-directory, so that Python can load them. For example, if you installed Python yourself on a Linux machine, it would typically be somewhere like /usr/local/lib/python2.5/site-packages. Installing Python packages this way often requires you to be able to write to the Python directories, which may require root priveleges, hence the "sudo" prefix. If this is not the case, you can drop the "sudo".

Alternatively, you can install the SPARTA files (or any other Python packages) in your own user space. The second "install" command does this, where you should replace "foo" with your directory of choice.

If these commands are successful, a *lammps.py* and *_lammps_serial.so* file will be put in the appropriate directory.

11.2 Creating a shared MPI library

A shared library is one that is dynamically loadable, which is what Python requires. On Linux this is a library file that ends in ".so", not ".a". Such a shared library is normally not built if you installed MPI yourself, but it is easy to do. Here is how to do it for MPICH, a popular open-source version of MPI, distributed by Argonne National Labs. From within the mpich directory, type

```
./configure --enable-sharedlib=gcc
make
make install
```

You may need to use "sudo make install" in place of the last line. The end result should be the file libmpich.so in /usr/local/lib.

IMPORTANT NOTE: If the file libmpich.a already exists in your installation directory (e.g. /usr/local/lib), you will now have both a static and shared MPI library. This will be fine for running SPARTA from Python since it only uses the shared library. But if you now try to build SPARTA by itself as a stand-alone program (cd lammps/src; make foo) or build other codes that expect to link against libmpich.a, then those builds will typically fail if the linker uses libmpich.so instead. This means you will need to remove the file /usr/local/lib/libmich.so before building SPARTA again as a stand-alone code.

11.3 Extending Python with a parallel version of SPARTA

From the python directory, type

```
python setup.py build
```

and then one of these commands:

```
sudo python setup.py install
python setup.py install --home=~/foo
```

The "build" command should compile all the needed SPARTA C++ files, which will require MPI to be installed on your system. This means it must find both the header file mpi.h and a shared library file, e.g. libmpich.so if the MPICH version of MPI is installed. See the preceding section for how to create a shared library version of MPI if it does not exist. You may need to adjust the "include_dirs" and "library_dirs" and "libraries" fields in python/setup.py to insure the Python build finds all the files it needs.

The first "install" command will put the needed files in your Python's site-packages sub-directory, so that Python can load them. For example, if you installed Python yourself on a Linux machine, it would typically be somewhere like /usr/local/lib/python2.5/site-packages. Installing Python packages this way often requires you to be able to write to the Python directories, which may require root priveleges, hence the "sudo" prefix. If this is not the case, you can drop the "sudo".

Alternatively, you can install the SPARTA files (or any other Python packages) in your own user space. The second "install" command does this, where you should replace "foo" with your directory of choice.

If these commands are successful, a *lammps.py* and *_lammps.so* file will be put in the appropriate directory.

11.4 Extending Python with MPI

There are several Python packages available that purport to wrap MPI as a library and allow MPI functions to be called from Python.

These include

- pyMPI
- maroonmpi
- mpi4py
- myMPI
- Pypar

All of these except pyMPI work by wrapping the MPI library (which must be available on your system as a shared library, as discussed above), and exposing (some portion of) its interface to your Python script. This means they cannot be used interactively in parallel, since they do not address the issue of interactive input to multiple instances of Python running on different processors. The one exception is pyMPI, which alters the Python interpreter to address this issue, and (I believe) creates a new alternate executable (in place of python itself) as a result.

In principle any of these Python/MPI packages should work to invoke both calls to SPARTA and MPI itself from a Python script running in parallel. However, when I downloaded and looked at a few of them, their docuemtation was incomplete and I had trouble with their installation. It's not clear if some of the packages are still being actively developed and supported.

The one I recommend, since I have successfully used it with SPARTA, is Pypar. Pypar requires the ubiquitous Numpy package be installed in your Python. After launching python, type

```
>>> import numpy
```

to see if it is installed. If not, here is how to install it (version 1.3.0b1 as of April 2009). Unpack the numpy tarball and from its top-level directory, type

```
python setup.py build
sudo python setup.py install
```

The "sudo" is only needed if required to copy Numpy files into your Python distribution's site-packages directory.

To install Pypar (version pypar-2.1.0_66 as of April 2009), unpack it and from its "source" directory, type

```
python setup.py build
```

```
sudo python setup.py install
```

Again, the "sudo" is only needed if required to copy PyPar files into your Python distribution's site-packages directory.

If you have successully installed Pypar, you should be able to run python serially and type

```
>>> import pypar
```

without error. You should also be able to run python in parallel on a simple test script

```
% mpirun -np 4 python test.script
```

where test.script contains the lines

```
import pypar
print "Proc %d out of %d procs" % (pypar.rank(),pypar.size())
```

and see one line of output for each processor you ran on.

11.5 Testing the Python-SPARTA interface

Before using SPARTA in a Python program, one more step is needed. The interface to SPARTA is via the Python ctypes package, which loads the shared SPARTA library via a CDLL() call, which in turn is a wrapper on the C-library dlopen(). This command is different than a normal Python "import" and needs to be able to find the SPARTA shared library, which is either in the Python site-packages directory or in a local directory you specified in the "python setup.py install" command, as described above.

The simplest way to do this is add a line like this to your .cshrc or other shell start-up file.

```
setenv LD_LIBRARY_PATH
${LD_LIBRARY_PATH}:/usr/local/lib/python2.5/site-packages
```

and then execute the shell file to insure the path has been updated. This will extend the path that dlopen() uses to look for shared libraries.

To test if the serial SPARTA library has been successfully installed (mode 1 above), launch Python and type

```
>>> from lammps import lammps
>>> lmp = lammps()
```

If you get no errors, you're ready to use serial SPARTA from Python.

If you built SPARTA for parallel use (mode 2 or 3 above), launch Python in parallel:

```
% mpirun -np 4 python test.script
```

where test.script contains the lines

```
import pypar
from lammps import lammps
lmp = lammps()
print "Proc %d out of %d procs has" % (pypar.rank(),pypar.size()), lmp
pypar.finalize()
```

Again, if you get no errors, you're good to go.

Note that if you left out the "import pypar" line from this script, you would instantiate and run SPARTA independently on each of the P processors specified in the mpirun command. You can test if Pypar is enabling true parallel Python and SPARTA by adding a line to the above sequence of commands like lmp.file("in.lj") to run an input script and see if the SPARTA run says it ran on P processors or if you get output from P duplicated 1-processor runs written to the screen. In the latter case, Pypar is not working correctly.

Note that this line:

```
from lammps import lammps
```

will import either the serial or parallel version of the SPARTA library, as wrapped by lammps.py. But if you installed both via setup_serial.py and setup.py, it will always import the parallel version, since it attempts that first.

Note that if your Python script imports the Pypar package (as above), so that it can use MPI calls directly, then Pypar initializes MPI for you. Thus the last line of your Python script should be pypar.finalize(), to insure MPI is shut down correctly.

Also note that a Python script can be invoked in one of several ways:

% python foo.script % python -i foo.script % foo.script

The last command requires that the first line of the script be something like this:

#!/usr/local/bin/python #!/usr/local/bin/python -i

where the path points to where you have Python installed, and that you have made the script file executable:

% chmod +x foo.script

Without the "-i" flag, Python will exit when the script finishes. With the "-i" flag, you will be left in the Python interpreter when the script finishes, so you can type subsequent commands. As mentioned above, you can only run Python interactively when running Python on a single processor, not in parallel.

11.6 Using SPARTA from Python

The Python interface to SPARTA consists of a Python "lammps" module, the source code for which is in python/lammps.py, which creates a "lammps" object, with a set of methods that can be invoked on that object. The sample Python code below assumes you have first imported the "lammps" module in your Python script and its settings as follows:

```
from lammps import lammps
from lammps import LMPINT as INT
from lammps import LMPDOUBLE as DOUBLE
from lammps import LMPIPTR as IPTR
from lammps import LMPDPTR as DPTR
from lammps import LMPDPTRPTR as DPTRPTR
```

These are the methods defined by the lammps module. If you look at the file src/library.cpp you will see that they

correspond one-to-one with calls you can make to the SPARTA library from a C++ or C or Fortran program.

```
lmp = lammps()           # create a SPARTA object
lmp = lammps(list)       # ditto, with command-line args, list = ["-echo","screen"]

lmp.close()              # destroy a SPARTA object

lmp.file(file)           # run an entire input script, file = "in.lj"
lmp.command(cmd)         # invoke a single SPARTA command, cmd = "run 100"

xlo = lmp.extract_global(name,type)  # extract a global quantity
                                     # name = "boxxlo", "nlocal", etc
                                     # type = INT or DOUBLE

coords = lmp.extract_atom(name,type)       # extract a per-atom quantity
                                           # name = "x", "type", etc
                                           # type = IPTR or DPTR or DPTRPTR

eng = lmp.extract_compute(id,style,type)  # extract value(s) from a compute
v3 = lmp.extract_fix(id,style,type,i,j)   # extract value(s) from a fix
                                          # id = ID of compute or fix
                                          # style = 0 = global data
                                          #         1 = per-atom data
                                          #         2 = local data
                                          # type = 0 = scalar
                                          #        1 = vector
                                          #        2 = array
                                          # i,j = indices of value in global vector or array

var = lmp.extract_variable(name,group,flag)  # extract value(s) from a variable
                                             # name = name of variable
                                             # group = group ID (ignored for equal-style variables)
                                             # flag = 0 = equal-style variable
                                             #        1 = atom-style variable

natoms = lmp.get_natoms()                # total # of atoms as int
x = lmp.get_coords()                     # return coords of all atoms in x
lmp.put_coords(x)                        # set all atom coords via x
```

The creation of a SPARTA object does not take an MPI communicator as an argument. There should be a way to do this, so that the SPARTA instance runs on a subset of processors, if desired, but I don't yet know how from Pypar. So for now, it runs on MPI_COMM_WORLD, which is all the processors.

The file() and command() methods allow an input script or single commands to be invoked.

The extract_global(), extract_atom(), extract_compute(), extract_fix(), and extract_variable() methods return values or pointers to data structures internal to SPARTA.

For extract_global() see the src/library.cpp file for the list of valid names. New names could easily be added. A double or integer is returned. You need to specify the appropriate data type via the type argument.

For extract_atom(), a pointer to internal SPARTA atom-based data is returned, which you can use via normal Python subscripting. See the extract() method in the src/atom.cpp file for a list of valid names. Again, new names could easily be added. A pointer to a vector of doubles or integers, or a pointer to an array of doubles (double **) is returned. You need to specify the appropriate data type via the type argument.

For extract_compute() and extract_fix(), the global, per-atom, or local data calulated by the compute or fix can be accessed. What is returned depends on whether the compute or fix calculates a scalar or vector or array. For a

scalar, a single double value is returned. If the compute or fix calculates a vector or array, a pointer to the internal SPARTA data is returned, which you can use via normal Python subscripting. The one exception is that for a fix that calculates a global vector or array, a single double value from the vector or array is returned, indexed by I (vector) or I and J (array). I,J are zero-based indices. The I,J arguments can be left out if not needed. See this section of the manual for a discussion of global, per-atom, and local data, and of scalar, vector, and array data types. See the doc pages for individual computes and fixes for a description of what they calculate and store.

For extract_variable(), an equal-style or atom-style variable is evaluated and its result returned.

For equal-style variables a single double value is returned and the group argument is ignored. For atom-style variables, a vector of doubles is returned, one value per atom, which you can use via normal Python subscripting. The values will be zero for atoms not in the specified group.

The get_natoms() method returns the total number of atoms in the simulation, as an int. Note that extract_global("natoms") returns the same value, but as a double, which is the way SPARTA stores it to allow for systems with more atoms than can be stored in an int (> 2 billion).

The get_coords() method returns an ctypes vector of doubles of length 3*natoms, for the coordinates of all the atoms in the simulation, ordered by x,y,z and then by atom ID (see code for put_coords() below). The array can be used via normal Python subscripting. If atom IDs are not consecutively ordered within SPARTA, a None is returned as indication of an error.

Note that the data structure get_coords() returns is different from the data structure returned by extract_atom("x") in four ways. (1) Get_coords() returns a vector which you index as x[i]; extract_atom() returns an array which you index as x[i][j]. (2) Get_coords() orders the atoms by atom ID while extract_atom() does not. (3) Get_coords() returns a list of all atoms in the simulation; extract_atoms() returns just the atoms local to each processor. (4) Finally, the get_coords() data structure is a copy of the atom coords stored internally in SPARTA, whereas extract_atom returns an array that points directly to the internal data. This means you can change values inside SPARTA from Python by assigning a new values to the extract_atom() array. To do this with the get_atoms() vector, you need to change values in the vector, then invoke the put_coords() method.

The put_coords() method takes a vector of coordinates for all atoms in the simulation, assumed to be ordered by x,y,z and then by atom ID, and uses the values to overwrite the corresponding coordinates for each atom inside SPARTA. This requires SPARTA to have its "map" option enabled; see the atom_modify command for details. If it is not or if atom IDs are not consecutively ordered, no coordinates are reset,

The array of coordinates passed to put_coords() must be a ctypes vector of doubles, allocated and initialized something like this:

```
from ctypes import *
natoms = lmp.get_atoms()
n3 = 3*natoms
x = (c_double*n3)()
x0 = x coord of atom with ID 1
x1 = y coord of atom with ID 1
x2 = z coord of atom with ID 1
x3 = x coord of atom with ID 2
...
xn3-1 = z coord of atom with ID natoms
lmp.put_coords(x)
```

Alternatively, you can just change values in the vector returned by get_coords(), since it is a ctypes vector of doubles.

As noted above, these Python class methods correspond one-to-one with the functions in the SPARTA library interface in src/library.cpp and library.h. This means you can extend the Python wrapper via the following steps:

- Add a new interface function to src/library.cpp and src/library.h.
- Verify the new function is syntactically correct by building SPARTA as a library - see this section of the manual.
- Add a wrapper method in the Python SPARTA module to python/lammps.py for this interface function.
- Rebuild the Python wrapper via python/setup_serial.py or python/setup.py.
- You should now be able to invoke the new interface function from a Python script. Isn't ctypes amazing?

11.7 Example Python scripts that use SPARTA

These are the Python scripts included as demos in the python/examples directory of the SPARTA distribution, to illustrate the kinds of things that are possible when Python wraps SPARTA. If you create your own scripts, send them to us and we can include them in the SPARTA distribution.

trivial.py	read/run a SPARTA input script thru Python
demo.py	invoke various SPARTA library interface routines
simple.py	mimic operation of couple/simple/simple.cpp in Python
gui.py	GUI go/stop/temperature-slider to control SPARTA
plot.py	real-time temeperature plot with GnuPlot via Pizza.py
viz_tool.py	real-time viz via some viz package
vizplotgui_tool.py	combination of viz_tool.py and plot.py and gui.py

---

For the viz_tool.py and vizplotgui_tool.py commands, replace "tool" with "gl" or "atomeye" or "pymol" or "vmd", depending on what visualization package you have installed.

Note that for GL, you need to be able to run the Pizza.py GL tool, which is included in the pizza sub-directory. See the Pizza.py doc pages for more info:

Note that for AtomEye, you need version 3, and there is a line in the scripts that specifies the path and name of the executable. See the AtomEye WWW pages here or here for more details:

```
http://mt.seas.upenn.edu/Archive/Graphics/A
http://mt.seas.upenn.edu/Archive/Graphics/A3/A3.html
```

The latter link is to AtomEye 3 which has the scriping capability needed by these Python scripts.

Note that for PyMol, you need to have built and installed the open-source version of PyMol in your Python, so that you can import it from a Python script. See the PyMol WWW pages here or here for more details:

```
http://www.pymol.org
http://sourceforge.net/scm/?type=svn&group_id=4546
```

The latter link is to the open-source version.

Note that for VMD, you need a fairly current version (1.8.7 works for me) and there are some lines in the pizza/vmd.py script for 4 PIZZA variables that have to match the VMD installation on your system.

See the python/README file for instructions on how to run them and the source code for individual scripts for comments about what they do.

Here are screenshots of the vizplotgui_tool.py script in action for different visualization package options. Click to see larger images:

....

12. Errors

This section describes the various kinds of errors you can encounter when using SPARTA.

12.1 Common problems

If two SPARTA runs do not produce the same answer on different machines or different numbers of processors, this is typically not a bug. In theory you should get identical answers on any number of processors and on any machine. In practice, numerical round-off can cause slight differences and eventual divergence of molecular dynamics phase space trajectories within a few 100s or few 1000s of timesteps. However, the statistical properties of the two runs (e.g. average energy or temperature) should still be the same.

If the velocity command is used to set initial atom velocities, a particular atom can be assigned a different velocity when the problem is run on a different number of processors or on different machines. If this happens, the phase space trajectories of the two simulations will rapidly diverge. See the discussion of the *loop* option in the velocity command for details and options that avoid this issue.

Similarly, the create_atoms command generates a lattice of atoms. For the same physical system, the ordering and numbering of atoms by atom ID may be different depending on the number of processors.

Some commands use random number generators which may be setup to produce different random number streams on each processor and hence will produce different effects when run on different numbers of processors. A commonly-used example is the fix langevin command for thermostatting.

A SPARTA simulation typically has two stages, setup and run. Most SPARTA errors are detected at setup time; others like a bond stretching too far may not occur until the middle of a run.

SPARTA tries to flag errors and print informative error messages so you can fix the problem. Of course, SPARTA cannot figure out your physics or numerical mistakes, like choosing too big a timestep, specifying erroneous force field coefficients, or putting 2 atoms on top of each other! If you run into errors that SPARTA doesn't catch that you think it should flag, please send an email to the developers.

If you get an error message about an invalid command in your input script, you can determine what command is causing the problem by looking in the log.lammps file or using the echo command to see it on the screen. For a given command, SPARTA expects certain arguments in a specified order. If you mess this up, SPARTA will often flag the error, but it may read a bogus argument and assign a value that is valid, but not what you wanted. E.g. trying to read the string "abc" as an integer value and assigning the associated variable a value of 0.

Generally, SPARTA will print a message to the screen and logfile and exit gracefully when it encounters a fatal error. Sometimes it will print a WARNING to the screen and logfile and continue on; you can decide if the WARNING is important or not. A WARNING message that is generated in the middle of a run is only printed to the screen, not to the logfile, to avoid cluttering up thermodynamic output. If SPARTA crashes or hangs without spitting out an error message first then it could be a bug (see this section) or one of the following cases:

SPARTA runs in the available memory a processor allows to be allocated. Most reasonable MD runs are compute limited, not memory limited, so this shouldn't be a bottleneck on most platforms. Almost all large memory allocations in the code are done via C-style malloc's which will generate an error message if you run out of memory. Smaller chunks of memory are allocated via C++ "new" statements. If you are unlucky you could run out of memory just when one of these small requests is made, in which case the code will crash or hang (in parallel), since SPARTA doesn't trap on those errors.

Illegal arithmetic can cause SPARTA to run slow or crash. This is typically due to invalid physics and numerics that your simulation is computing. If you see wild thermodynamic values or NaN values in your SPARTA output, something is wrong with your simulation. If you suspect this is happening, it is a good idea to print out thermodynamic info frequently (e.g. every timestep) via the thermo so you can monitor what is happening. Visualizing the atom movement is also a good idea to insure your model is behaving as you expect.

In parallel, one way SPARTA can hang is due to how different MPI implementations handle buffering of messages. If the code hangs without an error message, it may be that you need to specify an MPI setting or two (usually via an environment variable) to enable buffering or boost the sizes of messages that can be buffered.

12.2 Reporting bugs

If you are confident that you have found a bug in SPARTA, follow these steps.

Check the New features and bug fixes section of the SPARTA WWW site to see if the bug has already been reported or fixed or the Unfixed bug to see if a fix is pending.

Check the mailing list to see if it has been discussed before.

If not, send an email to the mailing list describing the problem with any ideas you have as to what is causing it or where in the code the problem might be. The developers will ask for more info if needed, such as an input script or data files.

The most useful thing you can do to help us fix the bug is to isolate the problem. Run it on the smallest number of atoms and fewest number of processors and with the simplest input script that reproduces the bug and try to identify what command or combination of commands is causing the problem.

As a last resort, you can send an email directly to the developers.

12.3 Error & warning messages

These are two alphabetic lists of the ERROR and WARNING messages SPARTA prints out and the reason why. If the explanation here is not sufficient, the documentation for the offending command may help. Error and warning messages also list the source file and line number where the error was generated. For example, this message

ERROR: Illegal velocity command (velocity.cpp:78)

means that line #78 in the file src/velocity.cpp generated the error. Looking in the source code may help you figure out what went wrong.

Note that error messages from user-contributed packages are not listed here. If such an error occurs and is not self-explanatory, you'll need to look in the source code or contact the author of the package.

Errors:

UNDOCUMENTED

%d particles are not in correct cell

UNDOCUMENTED

All universe/uloop variables must have same # of values

Self-explanatory.

All variables in next command must be same style

Self-explanatory.

Another input script is already being processed

Cannot attempt to open a 2nd input script, when the original file is still being processed.

Arccos of invalid value in variable formula

Argument of arccos() must be between -1 and 1.

Arcsin of invalid value in variable formula

Argument of arcsin() must be between -1 and 1.

Atom vector in equal-style variable formula

Atom vectors generate one value per atom which is not allowed in an equal-style variable.

Atom-style variable in equal-style variable formula

Atom-style variables generate one value per atom which is not allowed in an equal-style variable.

Bad grid of processors for create_grid

UNDOCUMENTED

Bigint setting in dsmctype.h is invalid

UNDOCUMENTED

Box bounds are invalid

The box boundaries specified in the read_data file are invalid. The lo value must be less than the hi value for all 3 dimensions.

Can only use -plog with multiple partitions

UNDOCUMENTED

Can only use -pscreen with multiple partitions

UNDOCUMENTED

Cannot create grid when grid is already defined

UNDOCUMENTED

Cannot create/grow a vector/array of pointers for %s

SPARTA code is making an illegal call to the templated memory allocaters, to create a vector or array of pointers.

Cannot create_box after simulation box is defined

The create_box command cannot be used after a read_data, read_restart, or create_box command.

Cannot create_grid before simulation box is defined

UNDOCUMENTED

Cannot create_molecules before simulation box is defined

UNDOCUMENTED

Cannot open -reorder file

UNDOCUMENTED

Cannot open input script %s

Self-explanatory.

Cannot open log.sparta

UNDOCUMENTED

Cannot open logfile

The SPARTA log file named in a command-line argument cannot be opened. Check that the path and name are correct.

Cannot open logfile %s

The SPARTA log file specified in the input script cannot be opened. Check that the path and name are correct.

Cannot open screen file
: The screen file specified as a command-line argument cannot be opened. Check that the directory you are running in allows for files to be created.

Cannot open species file %s
: UNDOCUMENTED

Cannot open universe log file
: For a multi-partition run, the master log file cannot be opened. Check that the directory you are running in allows for files to be created.

Cannot open universe screen file
: For a multi-partition run, the master screen file cannot be opened. Check that the directory you are running in allows for files to be created.

Cannot redefine variable as a different style
: An equal-style variable can be re-defined but only if it was originally an equal-style variable.

Cannot run 2d simulation with nonperiodic Z dimension
: Use the boundary command to make the z dimension periodic in order to run a 2d simulation.

Cannot set velocity to non-zero z value for 2d simulation
: UNDOCUMENTED

Cannot use -reorder after -partition
: UNDOCUMENTED

Cannot use cwiggle in variable formula between runs
: This is a function of elapsed time.

Cannot use ramp in variable formula between runs
: This is because the ramp() function is time dependent.

Cannot use swiggle in variable formula between runs
: This is a function of elapsed time.

Cannot use vdisplace in variable formula between runs
: This is a function of elapsed time.

Compute used in variable between runs is not current
: Computes cannot be invoked by a variable in between runs. Thus they must have been evaluated on the last timestep of the previous run in order for their value(s) to be accessed. See the doc page for the variable command for more info.

Create_box z box bounds must straddle 0.0 for 2d simulation
: UNDOCUMENTED

Create_grid nz value must be 1 for a 2d simulation
: UNDOCUMENTED

Created incorrect # of particles = %ld
: UNDOCUMENTED

Dimension command after simulation box is defined
: The dimension command cannot be used after a read_data, read_restart, or create_box command.

Divide by 0 in variable formula
: Self-explanatory.

Empty brackets in variable
: There is no variable syntax that uses empty brackets. Check the variable doc page.

Expected floating point parameter in variable definition
: The quantity being read is a non-numeric value.

Expected integer parameter in variable definition
: The quantity being read is a floating point or non-numeric value.

Failed to allocate %ld bytes for array %s
: Your SPARTA simulation has run out of memory. You need to run a smaller simulation or on more processors.

Failed to reallocate %ld bytes for array %s
: Your SPARTA simulation has run out of memory. You need to run a smaller simulation or on more

processors.

Fix in variable not computed at compatible time
: Fixes generate their values on specific timesteps. The variable is requesting the values on a non-allowed timestep.

Gmask function in equal-style variable formula
: Gmask is per-atom operation.

Grmask function in equal-style variable formula
: Grmask is per-atom operation.

Group ID in variable formula does not exist
: Self-explanatory.

Illegal ... command
: Self-explanatory. Check the input script syntax and compare to the documentation for the command. You can use -echo screen as a command-line option when running SPARTA to see the offending line.

Incorrect line format in species file
: UNDOCUMENTED

Index between variable brackets must be positive
: Self-explanatory.

Indexed per-atom vector in variable formula without atom map
: Accessing a value from an atom vector requires the ability to lookup an atom index, which is provided by an atom map. An atom map does not exist (by default) for non-molecular problems. Using the atom_modify map command will force an atom map to be created.

Input line too long after variable substitution
: This is a hard (very large) limit defined in the input.cpp file.

Input line too long: %s
: This is a hard (very large) limit defined in the input.cpp file.

Invalid -reorder N value
: UNDOCUMENTED

Invalid Boolean syntax in if command
: Self-explanatory.

Invalid atom vector in variable formula
: The atom vector is not recognized.

Invalid collide style
: UNDOCUMENTED

Invalid command-line argument
: One or more command-line arguments is invalid. Check the syntax of the command you are using to launch SPARTA.

Invalid compute ID in variable formula
: The compute is not recognized.

Invalid entry in reorder file
: UNDOCUMENTED

Invalid fix ID in variable formula
: The fix is not recognized.

Invalid group function in variable formula
: Group function is not recognized.

Invalid math function in variable formula
: Self-explanatory.

Invalid math/group/special function in variable formula
: Self-explanatory.

Invalid run command N value
: The number of timesteps must fit in a 32-bit integer. If you want to run for more steps than this, perform multiple shorter runs.

Invalid seed for Marsaglia random # generator

The initial seed for this random number generator must be a positive integer less than or equal to 900 million.

Invalid seed for Park random # generator

The initial seed for this random number generator must be a positive integer.

Invalid special function in variable formula

Self-explanatory.

Invalid syntax in variable formula

Self-explanatory.

Invalid variable evaluation in variable formula

A variable used in a formula could not be evaluated.

Invalid variable in next command

Self-explanatory.

Invalid variable name

Variable name used in an input script line is invalid.

Invalid variable name in variable formula

Variable name is not recognized.

Invalid variable style with next command

Variable styles *equal* and *world* cannot be used in a next command.

Label wasn't found in input script

Self-explanatory.

Log of zero/negative value in variable formula

Self-explanatory.

MPI_SPARTA_BIGINT and bigint in dsmctype.h are not compatible

UNDOCUMENTED

Mismatched brackets in variable

Self-explanatory.

Mismatched compute in variable formula

A compute is referenced incorrectly or a compute that produces per-atom values is used in an equal-style variable formula.

Mismatched fix in variable formula

A fix is referenced incorrectly or a fix that produces per-atom values is used in an equal-style variable formula.

Mismatched variable in variable formula

A variable is referenced incorrectly or an atom-style variable that produces per-atom values is used in an equal-style variable formula.

Must use -in switch with multiple partitions

A multi-partition simulation cannot read the input script from stdin. The -in command-line option must be used to specify a file.

Non digit character between brackets in variable

Self-explantory.

Nprocs not a multiple of N for -reorder

UNDOCUMENTED

Partition numeric index is out of bounds

UNDOCUMENTED

Per-processor grid count is too big

UNDOCUMENTED

Power by 0 in variable formula

Self-explanatory.

Processor partitions are inconsistent

The total number of processors in all partitions must match the number of processors SPARTA is running on.

Region ID in variable formula does not exist

Self-explanatory.

Rmask function in equal-style variable formula

Rmask is per-atom operation.

Run command before simulation box is defined

The run command cannot be used before a read_data, read_restart, or create_box command.

Small,big integers are not sized correctly

UNDOCUMENTED

Smallint setting in dsmctype.h is invalid

UNDOCUMENTED

Species ID does not appear in species file

UNDOCUMENTED

Species ID is already defined

UNDOCUMENTED

Sqrt of negative value in variable formula

Self-explanatory.

Substitution for illegal variable

Input script line contained a variable that could not be substituted for.

Too many timesteps

UNDOCUMENTED

Unbalanced quotes in input line

No matching end double quote was found following a leading double quote.

Unexpected end of -reorder file

UNDOCUMENTED

Unexpected end of reorder file

UNDOCUMENTED

Universe/uloop variable count < # of partitions

A universe or uloop style variable must specify a number of values >= to the number of processor partitions.

Unknown command: %s

The command is not known to SPARTA. Check the input script.

Variable formula compute array is accessed out-of-range

Self-explanatory.

Variable formula fix array is accessed out-of-range

Self-explanatory.

Variable name must be alphanumeric or underscore characters

Self-explanatory.

Velocity command before simulation box is defined

The velocity command cannot be used before a read_data, read_restart, or create_box command.

Velocity command with no particles existing

UNDOCUMENTED

World variable count doesn't match # of partitions

A world-style variable must specify a number of values equal to the number of processor partitions.

Warnings:

13. Future and history

This section lists features we are planning to add to SPARTA, features of previous versions of SPARTA, and features of other parallel molecular dynamics codes I've distributed.

13.1 Coming attractions

The Wish list link on the SPARTA WWW page gives a list of features we are hoping to add to SPARTA in the future, including contact names of individuals you can email if you are interested in contributing to the developement or would be a future user of that feature.

You can also send email to the developers if you want to add your wish to the list.

13.2 Past versions

bound_modify command

Syntax:

```
bound_modify wall keyword value ...
```

- wall = *xlo* or *xhi* or *ylo* or *yhi* or *zlo* or *zhi*
- one or more keyword/value pairs may be listed

```
keywords = surf
  surf value = surface collision ID
    sc-ID = ID of a surface collision model
```

Examples:

```
bound_modify yhi surf 1
bound_modify zlo surf hotwall
```

Description:

Set parameters for one of the boundaries of the global simulation box.

Any of the 6 faces can be selected via the *wall* setting.

The *surf* keyword can only be used when the boundary is of type "s", for surface, as set by the boundary command. This keyword assigns a surface collision model to the boundary, as defined by the surf_collide command. When that command is used, a surface collision ID is defined, which is specified with this command as *sc-ID*.

The effect of this keyword is that particle collisions with this boundary will be computed by the specified surface collision model.

Restrictions: none

Related commands:

boundary

Default: none

boundary command

Syntax:

```
boundary x y z
```

- x,y,z = *o* or *p* or *r* or *s*, one or two letters

```
  o is outflow
    p is periodic
    r is specular reflection
    s is treat boundary as a surface
```

Examples:

```
boundary o p p
boundary os o o
boundary r p rs
```

Description:

Set the style of boundaries for the global simulation box in each of the x, y, z dimensions. A single letter assigns the same style to both the lower and upper face of the box. Two letters assigns the first style to the lower face and the second style to the upper face. The size of the simulation box is set by the create_box command.

The boundary style determines how particles exiting the box are handled.

Style *o* means an outflow boundary, so that particles freely exit the simulation.

Style *p* means the box is periodic, so that particles exit one end of the box and re-enter the other end. The *p* style must be applied to both faces of a dimension.

Style *r* means a specularly reflecting boundary. Particles that cross this boundary have their velocity reversed so as to re-enter the box. The new velocity is used to advect the particle for the reminder of the timestep following the collision.

Style *s* means the boundary is treated as a surface which allows the particle-surface interaction to be treated in a variety of ways via the options provided by the surf_collide command. This is effectively the same as when a particle collides with a triangulated surface read in and setup by the read_surf command.

For style *s*, the boundary face must also be assigned to a surface collision model defined by the surf_collide command. The assignment of boundary to model is done via the bound_modify command.

Restrictions:

For 2d simulations, the z dimension must be periodic.

Related commands:

bound_modify, surf_collide

Default:

```
boundary p p p
```

clear command

Syntax:

```
clear
```

Examples:

```
(commands for 1st simulation)
clear
(commands for 2nd simulation)
```

Description:

This command deletes all atoms, restores all settings to their default values, and frees all memory allocated by SPARTA. Once a clear command has been executed, it is as if SPARTA were starting over, with only the exceptions noted below. This command enables multiple jobs to be run sequentially from one input script.

These settings are not affected by a clear command: the working directory (shell command), log file status (log command), echo status (echo command), and input script variables (variable command).

Restrictions: none

Related commands: none

Default: none

collide command

Syntax:

```
collide style args
```

- style = *none* or *vss*
- args = arguments for that style

```
none args = none
  vss args = mix-ID file
    mix-ID = ID of mixture to use for group definitions
    file = filename that lists species with their VSS model parameters
```

Examples:

```
collide none
collide vss all.vss background
```

Description:

Define what style of particle-particle collisions will be performed by SPARTA. If collisions are performed, particles are sorted into grid cells every timestep and the appropriate collision model is invoked on a per-grid-cell basis.

The *none* style means that no particle-particle collisions will be performed, i.e. free molecular flow.

The *vss* style is the Variable Soft Sphere model.

NOTE: VSS model needs to be described and cited.

The *mix-ID* argument is a mixture ID which must contain all the species defined for use by the simulation. The group definitions in the mixture will be used to perform collisions between various pairs of groups.

The *file* argument is for a file which contains definitions of VSS model parameters for some number of species, not all of which need to used by this simulation. Only species currently defined by the simulation will be extracted from the file and they must be present in the file.

The format of the file is as follows. Comments or blank lines are allowed in the file. Comment lines start with a "#" character. All other lines must have format like this with values separated by whitespace:

species-ID prop1 prop2 prop3 prop4

The species-ID is a string that will be matched against the species list used by the simulation, as described above. The properties are as follows:

- prop1 = diam = diamter of particle (distance units)
- prop2 = omega = ???
- prop3 = tref = reference temperature (temperature units)
- prop4 = alpha = ???

NOTE: give correct definitions and their units

Restrictions: none

Related commands:

mixture

Default:

style = none

compute command

Syntax:

```
compute ID style args
```

- ID = user-assigned name for the computation
- style = one of a list of possible style names (see below)
- args = arguments used by a particular style

Examples:

```
compute 1 temp
```

Description:

Define a computation that will be performed on a collection of particles or grid cells or surface elements. Quantities calculated by a compute are instantaneous values, meaning they are calculated from information about the current timestep or iteration, though a compute may internally store some information about a previous state of the system. Defining a compute does not perform a computation. Instead computes are invoked by other SPARTA commands as needed, e.g. to generate thermodynamic or dump file output.

The ID of a compute can only contain alphanumeric characters and underscores.

Computes can be deleted with the uncompute command.

Code for new computes can be added to SPARTA (see this section of the manual) and the results of their calculations accessed in the various ways described above.

Computes calculate one of four styles of quantities: global, per-molecule, per-grid, or per-surf. A global quantity is one or more system-wide values, e.g. the temperature of the system. A per-molecule quantity is one or more values per molecule, e.g. the kinetic energy of each molecule. A per-grid quantity is one or more values per grid cell. A per-surf quantity is one or more values per surface element.

Computes that produce per-molecule quantities have the word "molecule" in their style name, e.g. *ke/molecule*. Computes that produce per-grid quantities have the word "grid" in their style name, e.g. *ke/grid*. Computes that produce per-surf quantities have the word "surf" in their style name, e.g. *ke/surf*. Styles with neither "molecule" or "grid" or "surf" in their style nameproduce global quantities.

Note that a single compute produces either global or per-molecule or per-grid or per-surf quantities, but never more than one of these.

Global, per-molecule, per-grid, and per-surf quantities each come in three kinds: a single scalar value, a vector of values, or a 2d array of values. The doc page for each compute describes the style and kind of values it produces, e.g. a per-molecule vector. Some computes produce more than one kind of a single style, e.g. a global scalar and a global vector.

When a compute quantity is accessed, as in many of the output commands discussed below, it can be referenced via the following bracket notation, where ID is the ID of the compute:

c_ID	entire scalar, vector, or array
c_ID[I]	one element of vector, one column of array
c_ID[I][J]	one element of array

In other words, using one bracket reduces the dimension of the quantity once (vector -> scalar, array -> vector). Using two brackets reduces the dimension twice (array -> scalar). Thus a command that uses scalar compute values as input can also process elements of a vector or array.

Note that commands and variables which use compute quantities typically do not allow for all kinds, e.g. a command may require a vector of values, not a scalar. This means there is no ambiguity about referring to a compute quantity as c_ID even if it produces, for example, both a scalar and vector. The doc pages for various commands explain the details.

In SPARTA, the values generated by a compute can be used in several ways:

- Global values can be output via the stats_style command. They can be time-averaged via the fix ave/time command. Or the values can be referenced in a variable equal or variable atom command.
- Per-molecule values can be output via the dump particle command. They can be reduced to global values via the compute reduce command. Or the per-molecule values can be referenced in an molecule-style variable.
- Per-grid values can be output via the dump grid command. They can be time-averaged via the fix ave/grid command. They can be reduced to global values via the compute reduce command. Or the per-grid values can be referenced in a grid-style variable.

- Per-surf values can be output via the dump surf command. They can be time-averaged via the fix ave/surf command. They can be reduced to global values via the compute reduce command. Or the per-surf values can be referenced in a surf-style variable.

Each compute style has its own doc page which describes its arguments and what it does. Here is an alphabetic list of compute styles available in SPARTA:

- grid - varioud per grid cell quantities
- ke/molecule - temperature per molecule
- sonine/grid - Sonine moments per grid cell
- temp - temperature of all molecules

Restrictions: none

Related commands:

uncompute

Default: none

compute grid command

Syntax:

```
compute ID grid mix-ID value ...
```

- ID is documented in compute command
- grid = style name of this compute command
- mix-ID = mixture ID to perform calculation on
- one or more values may be appended
- values = *n* or *u* or *v* or *w* or *usq* or *vsq* or *wsq*

```
n = molecule count
  u = x component of velocity
  v = y component of velocity
  w = z component of velocity
  usq = x component of velocity squared
  vsq = y component of velocity squared
  wsq = z component of velocity squared
  ke = kinetic energy
  temp = temperature
```

Examples:

```
compute 1 grid species n u v w usq vsq wsq
compute 1 grid air n u v w
```

These commands will dump standard per grid cell averages for each species to a file every 1000 steps:

```
compute 1 grid species n u v w usq vsq wsq
fix 1 ave/grid 10 100 1000 c_1
dump 1 grid 1000 tmp.grid id f_1
```

Description:

Define a computation that calculates one or more values for the molecules in each grid cell. The values are summed for each group of species in the specified mixture. See the mixture command for how a set of species can be partitioned into groups.

Along with summing each specified value, a normalization factor is also computed. If the group has a single species, the normalization factor is typically just the count of contributing molecules. If the group has mutlipe species, the normalization factor is typically the total mass of the contributing molecules. The normalization factors are used by other commands that process values produced by this compute.

For example, for the *u* value, the dump grid command, will generate the average x component of velocity for molecules in a grid cell, averaged over molecules in the group. The fix ave/grid command will generate the average x component of velocity for molecules in a grid cell, averaged both over molecules in the group and time.

The *n* value counts the number of moleclues. There is no normalization factor associated with it.

The *u*, *v*, *w* values compute the summed velocity components for all molecules in the group. If the group has multiple species, the sum is mass-weighted and the normalization factor is the total mass.

```
U = Sum_i (mass_i Vx_i)
```

If the group has a single species, no mass weighting is done and the normalization factor is the count of molecules.

The *usq*, *vsq*, *wsq* values compute the sum of the velocity component squared for all molecules in the group. If the group has multiple species, the sum is mass-weighted and the normalization factor is the total mass.

```
Usq = Sum_i (mass_i Vx_i Vx_i)
```

If the group has a single species, no mass weighting is done and the normalization factor is the count of molecules.

The *ke* value computes the summed kinetic energy for all molecules in the group. The normalization factor is the count of molecules.

```
KE = Sum_i (1/2 mass_i Vsq_i)
Vsq = Vx*Vx + Vy*Vy + Vz*Vz
```

The *temp* value also computes the summed kinetic energy for all molecules in the group. The normalization factor is 3 times the count of molecules times the Boltzmann factor *kB*, so that the sum divided by the normalization factor is the temperature *T*.

```
KE = Sum_i (1/2 mass_i Vsq_i)
Vsq = Vx*Vx + Vy*Vy + Vz*Vz
3/2 kB T = KE
```

Output info:

This compute calculates a per-grid array, with the number of columns is equal to the number of values times the number of groups. The ordering of columns is first by values, then by groups. I.e. if the *n* and *u* values were specified as keywords, then the first two columns would be *n* and *u* for the first group, the 3rd and 4th columns would be *n* and *u* for the second group, etc.

The array can be accessed by any command that uses per-grid values from a compute as input.

The per-grid array values will be in the units appropriate to the individual values as described above. *U*, *v*, and *w* are in velocity units. *Usq*, *vsq*, and *wsq* are in velocity squared units. *Ke* is in energy units. *Temp* is in temperature units.

Restrictions: none

Related commands:

fix ave/grid, dump grid

Default: none

compute ke/grid command

Syntax:

```
compute ID ke/grid mix-ID
```

- ID is documented in compute command
- ke/grid = style name of this compute command
- mix-ID = mixture ID to perform calculation on

Examples:

```
compute 1 ke/grid all
compute 1 ke/grid air
```

Description:

Define a computation that calculates the translational kinetic energy of the molecules in each grid cell. The energy is summed for each group of species in the specified mixture. See the mixture command for how a set of species can be partitioned into groups.

The kinetic energy for a group of molecules in a cell is simply the sum of 1/2 m v^2 over all the molecules both in the cell and the group, where m is the mass and v is the velocity of each molecule.

Output info:

If the mixture has a single group, then this compute calculates a per-grid vector. If the mixture has multiple groups, this compute calcualtes a per-grid array, with the number of columns equal to the number of groups. Either can be accessed by any command that uses per-grid values from a compute as input.

The per-grid vector and array values will be in energy units.

Restrictions: none

Related commands:

fix ave/grid, dump grid

Default: none

compute ke/molecule command

Syntax:

```
compute ID ke/molecule
```

- ID is documented in compute command
- ke/molecule = style name of this compute command

Examples:

```
compute 1 ke/molecule
```

Description:

Define a computation that calculates the per-atom translational kinetic energy for each molecule.

The kinetic energy is simply 1/2 m v^2, where m is the mass and v is the velocity of each molecule.

Output info:

This compute calculates a per-molecule vector, which can be accessed by any command that uses per-molecule values from a compute as input.

The per-molecule vector values will be in energy units.

Restrictions: none

Related commands:

dump molecule

Default: none

compute sonine/grid command

Syntax:

```
compute ID sonine/grid mix-ID keyword values ...
```

- ID is documented in compute command
- sonine/grid = style name of this compute command
- mix-ID = mixture ID to perform calculation on
- one or more keywords may be appended, multiple times
- keyword = *thermal* or *a* or *b*
- values = values for specific keyword

```
thermal args = none = thermal temperature
  a args = dim order = sonine A moment
    dim = x or y or z
    order = number from 1 to 5
  b args = dim2 order = sonine B moment
    dim2 = xx or yy or zz or xy or yz or xz
    order = number from 1 to 5
```

Examples:

```
compute 1 sonine/grid species thermal a x 3 a z 3 b xy 3 b xz 3
compute 1 sonine/grid air a x 5 b xy 5
```

These commands will dump 10 averaged sonine moments for each species and each grid cell to a file every 1000 steps:

```
compute 1 sonine/grid species a x 5 b xy 5
fix 1 ave/grid 10 100 1000 c_1
dump 1 grid 1000 tmp.grid id f_1
```

Description:

Define a computation that calculates the sonine moments of the velocity distribution of the molecules in each grid cell. The moments are summed for each group of species in the specified mixture. See the mixture command for how a set of species can be partitioned into groups.

Along with summing each specified value, a normalization factor is also computed. If the group has a single species, the normalization factor is typically just the count of contributing molecules. If the group has mutlipe species, the normalization factor is typically the total mass of the contributing molecules. The normalization factors are used by other commands that process values produced by this compute.

For example, for the *a* value, the dump grid command, will generate the average sonine A moment for molecules in a grid cell, averaged over molecules in the group. The fix ave/grid command will generate the average sonine A moment for molecules in a grid cell, averaged both over molecules in the group and time.

The *thermal* keyword computes the summed thermal kinetic energy for all molecules in the group. The normalization factor is 3 times the count of molecules times the Boltzmann factor kB, so that the sum divided by the normalization factor is the temperature T.

```
thermal_KE = Sum_i (1/2 mass_i Csq_i)
Csq = Cx*Cx + Cy*Cy + Cz*Cz
Cx = Vx - COMx
Cy = Vy - COMy
Cz = Vz - COMz
COMx = Sum_i (mass_i Vx_i) / Sum_i (mass_i)
COMy = Sum_i (mass_i Vy_i) / Sum_i (mass_i)
COMz = Sum_i (mass_i Vz_i) / Sum_i (mass_i)
3/2 kB T = thermal_KE
```

Note that the thermal kinetic energy is calculated from *C* = the thermal velocity of each molecule, which is its velocity minus the center-of-mass (COM) velocity of molecules in that cell and group. The COM velocity is the mass-weighted sum of the velocities of molecules in the group divided by the total mass of the molecules.

The *a* keyword calculates the sum of one or more sonine A moments for all molecules in the group. If the group has multiple species, the sum is mass-weighted and the normalization factor is the total mass.

```
A1 = Sum_i (mass_i * Vdim * pow(Csq,1))
A2 = Sum_i (mass_i * Vdim * pow(Csq,2))
A3 = Sum_i (mass_i * Vdim * pow(Csq,3))
A4 = Sum_i (mass_i * Vdim * pow(Csq,4))
A5 = Sum_i (mass_i * Vdim * pow(Csq,5))
```

Vdim is Vx or Vy or Vz as specified by the *dim* value. *Csq* is the squared thermal velocity of the molecule and has the same meaning as above in the *thermal* equations. The number of moments computed is specified by the *order* value. If the group has a single species, no mass weighting is done and the normalization factor is the count of molecules.

The *b* keyword calculates the sum of one or more sonine B moments for all molecules in the group. If the group has multiple species, the sum is mass-weighted and the normalization factor is the total mass.

```
B1 = Sum_i (mass_i * Vdim1 * Vdim2 * pow(Csq,1))
B2 = Sum_i (mass_i * Vdim1 * Vdim2 * pow(Csq,2))
B3 = Sum_i (mass_i * Vdim1 * Vdim2 * pow(Csq,3))
B4 = Sum_i (mass_i * Vdim1 * Vdim2 * pow(Csq,4))
B5 = Sum_i (mass_i * Vdim1 * Vdim2 * pow(Csq,5))
```

Vdim is Vx or Vy or Vz as specified by the *dim* value. *Csq* is the squared thermal velocity of the molecule and has the same meaning as above in the *thermal* equations. The number of moments computed is specified by the *order* value. If the group has a single species, no mass weighting is done and the normalization factor is the count of molecules.

Output info:

This compute calculates a per-grid array, with the number of columns is equal to the number of values times the number of groups. The ordering of columns is first by values, then by groups. I.e. if the *thermal* and *b xy 2* moments were specified as keywords, then the 1st thru 3rd columns would be the *thermal* and B1 and B2 moments of the first group, the 4th thru 6th columns would be the *thermal* and B1 and B2 moments of the second group, etc.

The array can be accessed by any command that uses per-grid values from a compute as input.

The per-grid array values will be in the units appropriate to the individual values as described above. *Thermal* is in temperature units. A and B are in units like velocity cubed or velocity to the 6th power.

Restrictions: none

Related commands:

fix ave/grid, dump grid

Default: none

compute temp command

Syntax:

```
compute ID temp
```

- ID is documented in compute command
- temp = style name of this compute command

Examples:

```
compute 1 temp
compute myTemp temp
```

Description:

Define a computation that calculates the temperature of all particles.

The temperature is calculated by the formula KE = dim/2 N kB T, where KE = total kinetic energy of the particles (sum of 1/2 m v^2), dim = dimensionality of the simulation, N = number of particles, kB = Boltzmann constant, and T = temperature.

Output info:

This compute calculates a global scalar (the temperature). This value can be used by any command that uses global scalar values from a compute as input. See this section for an overview of SPARTA output options.

The scalar value will be in temperature units.

Restrictions: none

Related commands: none

Default: none

create_box command

Syntax:

```
create_box xlo xhi ylo yhi zlo zhi

xlo,xhi = box bounds in the x dimension (distance units)
ylo,yhi = box bounds in the y dimension (distance units)
zlo,zhi = box bounds in the z dimension (distance units)
```

Examples:

```
create_box 0 1 0 1 0 1
create_box 0 1 0 1 -0.5 0.5
create_box 0 10.0 0 5.0 -4.0 0.0
```

Description:

Set the size of the simulation box.

For a 2d simulation, as specifed by the dimension command, *zlo* < 0.0 and *zhi* > 0.0 is required. This is so that the z dimensions straddle 0.0.

Restrictions: none

Related commands: none

Default: none

create_grid command

Syntax:

```
create_grid Nx Ny Nz keyword args ...
```

NOTE: should add option to specify dx dy dz

- Nx,Ny,Nz = size of grid in each dimension

- zero or more keyword/arg pairs may be appended
- keyword = *stride* or *block* or *random*

```
stride args = d1 d2 d3
    d1 = x or y or z
    d2 = x or y or z, not same as d1
    d3 = x or y or z, not same as d1 or d2
  block args = Px Py Pz
    Px,Py,Pz = # of processors in each dimension
  random args = none
```

Examples:

```
create_grid 10 10 10
create_grid 10 10 10 block 2 * *
create_grid 100 100 1 stride y x z
```

Description:

Overlay a regular *Nx* by *Ny* by *Nz* grid over the simulation domain defined by the create_box command.

For 2d simulations, *Nz* must be set to 1.

The optional *stride* and *block* and *random* keywords determine how grid cells are assigned to processors. If none are specified, then the default *block* setting is used, as listed below.

The *stride* keyword means that every Pth cell is assigned to the same processor, where P is the number of processors. E.g. if there are 100 cells and 10 processors, then the 1st processor (proc 0) will be assigned cells 1,11,21, ..., 91. The 2nd processor (proc 1) will be assigned cells 2,12,22 ..., 92. And The 10th processor (proc 9) will be assigned cells 10,20,30, ..., 100.

The arguments *d1*, *d2*, *d3* are some permutation of *x*, *y*, and *z* and determine how the N grid cells are ordered. Each of the N cells has 3 indices (I,J,K) to describe its location in the 3d grid. If *d1 d2 d3* = *x z y*, then the cells will be ordered from 1 to N with the I index varying fastest, the K index next, and the J index slowest.

The *block* keyword maps the P processors to a *Px* by *Py* by *Pz* logical grid that overlays the actual *Nx* by *Ny* by *Nz* grid. This effectively assigns a contiguous 3d sub-block of cells to each processor.

Any of the *Px*, *Py*, *Pz* parameters can be specified with an asterisk "*", in which case SPARTA will choose the number of processors in that dimension. It will do this based on the size and shape of the global grid so as to minimize the surface-to-volume ratio of each processor's sub-block of cells.

The product of Px, Py, Pz must equal P, the total # of processors SPARTA is running on. For a 2d simulation, Pz must equal 1. If multiple partitions are being used then P is the number of processors in this partition; see this section for an explanation of the -partition command-line switch.

Note that if you run on a large, prime number of processors P, then a grid such as 1 x P x 1 will be required, which may incur extra communication costs.

The *random* keyword means that eacn grid cell will be assigned randomly to one of the processors. Note that in this case every processor will typically not be assigned exactly the same number of cells.

Restrictions:

This command can only be used after the simulation box is defined by the create_box command.

Related commands:

create_box

Default:

The option defaults are "block * * *",

create_molecules command

Syntax:

```
create_molecules mix-ID keyword value ...
```

- mix-ID = ID of mixture to use when creating molecules
- zero or more keyword/value pairs may be appended
- keyword = *n* or *single*

```
  n value = Np = number of molecules to create
    single values = species-ID x y z vx vy vz
      species-ID = ID of species of single molecule
      x,y,z = position of molecule (distance units)
      vx,vy,vz = velocity of molecule (velocity units)
```

Examples:

```
create_molecules background
create_molecules air n 100000
create_molecules air single N 5 5 5 10 -1 0
```

Description:

Create or add additional molecules to the simulation domain. The number of molecules created and their individual attributes, such species and velocity, are determined by the mixture attributes, as specied by the *mix-ID*. See the mixture command for more details.

Unless the *n* keyword is used with a *Np* value > 0, the number of molecules created is a function of the global *Fnum* value, the mixture number density, and the volume of the simulation domain. The *Fnum* value is set by the global fnum command. The mixture *nrho* is set by the mixture command. The volume of the simulation is set by the create_box command.

Based on these settings, each grid cell will have a target number of molecules *M* to insert. If *M* has a fractional value, e.g. 12.5, then 12 molecules will be inserted, and a 13th depending on the outcome of a random number generation. Each molecule will be inserted at a random location within the grid cell. The molecule species will be chosen randomly in accord with the *frac* settings of the collection of species in the mixture, as set by the mixture command. The velocity of the molecule will be set to the sum of the streaming velocity of the mixture and a thermal velocity sampled from the thermal temperature of the mixture. Both the streaming velocity and thermal temperature are also set by the mixture command.

The *n* keyword can be useful for debugging purposes. If *Np* is set to 0, then the count of created particles is a function of the *Fnum* and *nrho* and other mixture settings, as described above. If *Np* is set to a value > 0, then the *Fnum* and *nrho* settings are ignored, and *Np* molecules are created. The assignment of species and velocity to each molecules is done the same as described above.

The *single* keyword can also be useful for debugging purposes, e.g. to advect a single particle towards a surface. A single particle of the specified species is inserted at the specified position and with the specified velocity. In this case the *mix-ID* is ignored.

NOTE: the *loop* keyword is not yet implemented.

The *loop* keyword only applies when the *n* keyword is also used, and controls how the molecules are generated in parallel.

If the setting is *all*, then every processor loops over all *N* molecules. As the coordinates of each is created, each processor checks what grid cell it is in, and only stores the molecule if it owns that grid cell. Thus an identical set of molecules are created, no matter how many processors are running the simulation

If the setting is *local*, then each of the *P* processors generates a *N/P* subset of molecules, using its own random number generation. It only adds molecules to grid cells that it owns. This is a faster way to generate a large number of molecules, but means that the individual attributes of molecules will depend on the number of processors and the mapping of grid cells to procesors. Statistically, the overall set of created molecules should be the same as with the *all* setting.

Restrictions:

A non-zero *Np* value for the *n* keyword cannot be used together with the *single* keyword.

Related commands:

mixture, fix inflow

Default:

The keyword default is n = 0.

dimension command

Syntax:

```
dimension N
```

- N = 2 or 3

Examples:

```
dimension 2
```

Description:

Set the dimensionality of the simulation. By default SPARTA runs 3d simulations, but 2d simulations can also be specified.

Restrictions:

This command must be used before the simulation box is defined by a create_box command.

Related commands: none

Default:

```
dimension 3
```

dump command

dump image command

Syntax:

```
dump ID style N file args
```

- ID = user-assigned name for the dump
- style = *molecule* or *grid* or *surf* or *image*
- N = dump every this many timesteps
- file = name of file to write dump info to
- args = list of arguments for a particular style

```
molecule args = list of molecule attributes
    possible attributes = id, type, x, y, z, xs, ys, zs, vx, vy, vz,
                          c_ID, c_ID[N], f_ID, f_ID[N], v_name

      id = molecule ID
      type = molecule species
      x,y,z = unscaled molecule coordinates
      xs,ys,zs = scaled molecule coordinates
      vx,vy,vz = molecule velocities
      c_ID = per-molecule vector (or array) calculated by a compute with ID
      c_ID[N] = Nth column of per-molecule array calculated by a compute with ID
      f_ID = per-molecule vector (or array) calculated by a fix with ID
      f_ID[N] = Nth column of per-molecule array calculated by a fix with ID
      v_name = per-molecule vector calculated by a molecule-style variable with name

grid args = list of grid attributes
    possible attributes = id, proc, xlo, ylo, zlo, xhi, yhi, zhi,
                          c_ID, c_ID[N], f_ID, f_ID[N], v_name

      id = grid cell ID
      proc = processor that owns grid cell
      xlo,ylo,zlo = coords of lower left corner of grid cell
      xhi,yhi,zhi = coords of lower left corner of grid cell
      c_ID = per-grid vector (or array) calculated by a compute with ID
      c_ID[N] = Nth column of per-grid array calculated by a compute with ID
      f_ID = per-grid vector (or array) calculated by a fix with ID
      f_ID[N] = Nth column of per-grid array calculated by a fix with ID
      v_name = per-grid vector calculated by a grid-style variable with name

surf args = list of surf attributes
    possible attributes = id, v1x, v1y, v1z, v2x, v2y, v2z, v3x, v3y, v3z,
                          c_ID, c_ID[N], f_ID, f_ID[N], v_name

      id = surf element ID
      v1x,v1y,v1z = coords of 1st vertex in surface element
      v1x,v1y,v1z = coords of 2nd vertex in surface element
      v1x,v1y,v1z = coords of 3rd vertex in surface element
      c_ID = per-surf vector (or array) calculated by a compute with ID
      c_ID[N] = Nth column of per-surf array calculated by a compute with ID
      f_ID = per-surf vector (or array) calculated by a fix with ID
      f_ID[N] = Nth column of per-surf array calculated by a fix with ID
      v_name = per-surf vector calculated by a surf-style variable with name
```

image args = discussed on dump image doc page

Examples:

```
dump 1 molecule 100 dump.myforce.* id type x y vx fx
dump 2 molecule 100 dump.%.myforce id type c_myF[3] v_ke
dump 3 grid 1000 tmp.grid id proc xlo ylo zlo xhi yhi zhi
```

Description:

Dump a snapshot of simulation quantities to one or more files every N timesteps in one of several styles. The *image* style is the exception; it creates a JPG or PPM image file of the simulation configuration every N timesteps, as discussed on the dump image doc page.

The *style* keyword determines what quantities are written to the file and in what format. Settings made via the dump_modify command can also alter what info is included in the file and the format of individual values.

As described below, the filename determines the kind of output (text or binary or gzipped, one big file or one per timestep, one big file or one per processor).

The *molecule* and *grid* and *surf* styles create files in a simple text format that is self-explanatory when viewing a dump file. Many of the SPARTA post-processing tools, including Pizza.py, work with this format.

For post-processing purposes the text files are self-describing in the following sense.

The dimensions of the simulation box are included in each snapshot. This information is is formatted as:

```
ITEM: BOX BOUNDS xx yy zz
xlo xhi
ylo yhi
zlo zhi
```

where xlo,xhi are the maximum extents of the simulation box in the x-dimension, and similarly for y and z. The "xx yy zz" represent 6 characters that encode the style of boundary for each of the 6 simulation box boundaries (xlo,xhi and ylo,yhi and zlo,zhi). Each of the 6 characters is either o = outflow, p = periodic, or s = specular. See the boundary command for details.

The "ITEM: ATOMS" or "ITEM: CELLS" or "ITEM: SURFS" line in each snapshot lists column descriptors for the per-molecule or per-grid or per-surf lines that follow. The descriptors will be the attributes you specify in the dump command for the style.

Styles *molecule* and *grid* and *surf* allow you to specify a list of attributes to be written to the dump file for each molecule or grid cell or surface. Possible attributes are listed above and will appear in the order specified. An explanation of the possible attributes is given below.

Dumps are performed on timesteps that are a multiple of N (including timestep 0). Note that this means a dump will not be performed on the initial timestep after the dump command is invoked, if the current timestep is not a multiple of N. This behavior can be changed via the dump_modify first command. N can be changed between runs by using the dump_modify every command.

The specified filename determines how the dump file(s) is written. The default is to write one large text file, which is opened when the dump command is invoked and closed when an undump command is used or when SPARTA exits.

Dump filenames can contain two wildcard characters. If a "*" character appears in the filename, then one file per snapshot is written and the "*" character is replaced with the timestep value. For example, tmp.dump.* becomes tmp.dump.0, tmp.dump.10000, tmp.dump.20000, etc. Note that the dump_modify pad command can be used to insure all timestep numbers are the same length (e.g. 00010), which can make it easier to read a series of dump files in order by some post-processing tools.

If a "%" character appears in the filename, then one file is written for each processor and the "%" character is replaced with the processor ID from 0 to P-1. For example, tmp.dump.% becomes tmp.dump.0, tmp.dump.1, ... tmp.dump.P-1, etc. This creates smaller files and can be a fast mode of output on parallel machines that support parallel I/O for output.

Note that the "*" and "%" characters can be used together to produce a large number of small dump files!

If the filename ends with ".bin", the dump file (or files, if "*" or "%" is also used) is written in binary format. A binary dump file will be about the same size as a text version, but will typically write out much faster. Of course, when post-processing, you will need to convert it back to text format (see the binary2txt tool) or write your own code to read the binary file. The format of the binary file can be understood by looking at the tools/binary2txt.cpp file. This option is only available for the *molecule* style.

If the filename ends with ".gz", the dump file (or files, if "*" or "%" is also used) is written in gzipped format. A gzipped dump file will be about 3x smaller than the text version, but will also take longer to write.

This section explains the molecule attributes that can be specified as part of the *molecule* style.

Id is the molecule ID. *Type* is an integer index representing the molecule species.

The *x*, *y*, *z* attributes write molecule coordinates "unscaled", in the appropriate distance units. Use *xs*, *ys*, *zs* if you want the coordinates "scaled" to the box size, so that each value is 0.0 to 1.0.

Vx, *vy*, *vz* are components of molecule velocity.

The *c_ID* and *c_ID[N]* attributes allow per-molecule vectors or arrays calculated by a compute to be output. The ID in the attribute should be replaced by the actual ID of the compute that has been defined previously in the input script. See the compute command for details.

If *c_ID* is used as a attribute, then the per-molecule vector calculated by the compute is printed. If *c_ID[N]* is used, then N must be in the range from 1-M, which will print the Nth column of the M-length per-molecule array calculated by the compute.

The *f_ID* and *f_ID[N]* attributes allow vector or array per-molecule quantities calculated by a fix to be output. The ID in the attribute should be replaced by the actual ID of the fix that has been defined previously in the input script.

If *f_ID* is used as a attribute, then the per-molecule vector calculated by the fix is printed. If *f_ID[N]* is used, then N must be in the range from 1-M, which will print the Nth column of the M-length per-molecule array calculated by the fix.

The *v_name* attribute allows per-molecule vectors calculated by a variable to be output. The name in the attribute should be replaced by the actual name of the variable that has been defined previously in the input script. Only a molecule-style variable can be referenced, since it is the only style that generates per-molecule values. Variables of style *molecule* can reference per-molecule attributes, stats keywords, or invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of creating quantities to output to a dump file.

See Section_modify of the manual for information on how to add new compute and fix styles to SPARTA to calculate per-molecule quantities which could then be output into dump files.

This section explains the grid cell attributes that can be specified as part of the *grid* style.

Id is the grid cell ID.

Proc is the ID of the processor which owns the grid cell.

The *xlo*, *ylo*, *zlo* attributes write the coordinates of the lower-left corner of the grid cell in the appropriate distance units. The *xhi*, *yhi*, *zhi* attributes write the coordinates of the upper-right corner of the grid cell. The *zlo* and *zhi* attributes cannot be used for a 2d simulation.

The *c_ID* and *c_ID[N]* attributes allow per-grid vectors or arrays calculated by a compute to be output. The ID in the attribute should be replaced by the actual ID of the compute that has been defined previously in the input script. See the compute command for details.

If *c_ID* is used as a attribute, and the compute calculates a per-grid vector, then the per-grid vector is printed. If *c_ID[N]* is used, then N must be in the range from 1-M, which will print the Nth column of the M-column per-grid array calculated by the compute. If *c_ID* is used, and the compute calculates a per-grid array, then it is the same as if the individual columns of the array had been listed one by one. See the example below for fixes.

The *f_ID* and *f_ID[N]* attributes allow per-grid vectors or arrays calculated by a fix to be output. The ID in the attribute should be replaced by the actual ID of the fix that has been defined previously in the input script.

If *f_ID* is used as a attribute, and the fix calculates a per-grid vector, then the per-grid vector is printed. If *f_ID[N]* is used, then N must be in the range from 1-M, which will print the Nth column of the M-columne per-grid array calculated by the fix. If *f_ID* is used, and the fix calculates a per-grid array, then it is the same as if the individual columns of the array had been listed one by one. E.g. these 2 dump commands are equivalent for a simulation with 1 species, since the fix ave/grid standard command produces a 7-column per-grid array:

```
fix 1 ave/grid 10 100 1000 standard
dump 1 grid 1000 tmp.grid id f_1
dump 1 grid 1000 tmp.grid id f_1[1] f_1[2] f_1[3] f_1[4] f_1[5] f_1[6] f_1[7]
```

The *v_name* attribute allows per-grid vectors calculated by a variable to be output. The name in the attribute should be replaced by the actual name of the variable that has been defined previously in the input script. Only a grid-style variable can be referenced, since it is the only style that generates per-grid values. Variables of style *grid* can reference per-grid attributes, stats keywords, or invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of creating quantities to output to a dump file.

See Section_modify of the manual for information on how to add new compute and fix styles to SPARTA to calculate per-grid quantities which could then be output into dump files.

This section explains the surface element attributes that can be specified as part of the *surf* style. For 2d simulations, a surface element is a line segment with 2 end points. Crossing the unit +z vector into the vector (v2-v1) determines the outward normal of the line segment. For 3d simulations, a surface element is a triangle with 3 corner points. Crossing (v2-v1) into (v3-v1) determines the outward normal of the triangle.

Id is the surface element ID.

The *v1x*, *v1y*, *v1z*, *v2x*, *v2y*, *v2z*, *v3x*, *v3y*, *v3z* attributes write the coordinates of the vertices of the end or corner points of the surface element. The *v1z*, *v2z*, *v3x*, *v3y*, and *v3z* attributes cannot be used for a 2d simulation.

The *c_ID* and *c_ID[N]* attributes allow per-surf vectors or arrays calculated by a compute to be output. The ID in the attribute should be replaced by the actual ID of the compute that has been defined previously in the input script. See the compute command for details.

If *c_ID* is used as a attribute, and the compute calculates a per-srf vector, then the per-surf vector is printed. If *c_ID[N]* is used, then N must be in the range from 1-M, which will print the Nth column of the M-column per-surf array calculated by the compute. If *c_ID* is used, and the compute calculates a per-surf array, then it is the same as if the individual columns of the array had been listed one by one. See the example below for fixes.

The *f_ID* and *f_ID[N]* attributes allow per-surf vectors or arrays calculated by a fix to be output. The ID in the attribute should be replaced by the actual ID of the fix that has been defined previously in the input script.

If *f_ID* is used as a attribute, and the fix calculates a per-surf vector, then the per-surf vector is printed. If *f_ID[N]* is used, then N must be in the range from 1-M, which will print the Nth column of the M-columne per-surf array calculated by the fix. If *f_ID* is used, and the fix calculates a per-surf array, then it is the same as if the individual columns of the array had been listed one by one. E.g. these 2 dump commands are equivalent for a simulation with 1 species, since the fix ave/surf standard command produces a 7-column per-surf array:

```
fix 1 ave/surf 10 100 1000 standard
dump 1 surf 1000 tmp.surf id f_1
dump 1 surf 1000 tmp.surf id f_1[1] f_1[2] f_1[3] f_1[4] f_1[5] f_1[6] f_1[7]
```

The *v_name* attribute allows per-surf vectors calculated by a variable to be output. The name in the attribute should be replaced by the actual name of the variable that has been defined previously in the input script. Only a surf-style variable can be referenced, since it is the only style that generates per-surf values. Variables of style *surf* can reference per-surf attributes, stats keywords, or invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of creating quantities to output to a dump file.

See Section_modify of the manual for information on how to add new compute and fix styles to SPARTA to calculate per-surf quantities which could then be output into dump files.

Restrictions:

To write gzipped dump files, you must compile SPARTA with the -DSPARTA_GZIP option - see the Making SPARTA section of the documentation.

Related commands:

dump image, dump_modify, undump

Default:

The defaults for the image style are listed on the dump image doc page.

dump image command

Syntax:

```
dump ID image N file color diameter keyword value ...
```

- ID = user-assigned name for the dump
- image = style of dump command (other styles *particle* are discussed on the dump doc page)
- N = dump every this many timesteps
- file = name of file to write image to
- color = atom attribute that determines color of each atom
- diameter = atom attribute that determines size of each atom
- zero or more keyword/value pairs may be appended
- keyword = *adiam* or *atom* or *size* or *view* or *center* or *up* or *zoom* or *persp* or *box* or *axes* or *shiny* or *ssao*

```
adiam value = number = numeric value for atom diameter (distance units)
  atom = yes/no = do or do not draw atoms
  size values = width height = size of images
    width = width of image in # of pixels
    height = height of image in # of pixels
  view values = theta phi = view of simulation box
    theta = view angle from +z axis (degrees)
    phi = azimuthal view angle (degrees)
    theta or phi can be a variable (see below)
  center values = flag Cx Cy Cz = center point of image
    flag = "s" for static, "d" for dynamic
    Cx,Cy,Cz = center point of image as fraction of box dimension (0.5 = center of box)
    Cx,Cy,Cz can be variables (see below)
  up values = Ux Uy Uz = direction that is "up" in image
    Ux,Uy,Uz = components of up vector
    Ux,Uy,Uz can be variables (see below)
  zoom value = zfactor = size that simulation box appears in image
    zfactor = scale image size by factor > 1 to enlarge, factor <1 to shrink
    zfactor can be a variable (see below)
  persp value = pfactor = amount of "perspective" in image
    pfactor = amount of perspective (0 = none,  <1 = some, > 1 = highly skewed)
    pfactor can be a variable (see below)
  box values = yes/no diam = draw outline of simulation box
    yes/no = do or do not draw simulation box lines
    diam = diameter of box lines as fraction of shortest box length
  axes values = yes/no length diam = draw xyz axes
    yes/no = do or do not draw xyz axes lines next to simulation box
    length = length of axes lines as fraction of respective box lengths
    diam = diameter of axes lines as fraction of shortest box length
  shiny value = sfactor = shinyness of spheres and cylinders
    sfactor = shinyness of spheres and cylinders from 0.0 to 1.0
  ssao value = yes/no dfactor = SSAO depth shading
    yes/no = turn depth shading on/off
    dfactor = strength of shading from 0.0 to 1.0
```

Examples:

```
dump myDump all image 100 dump.*.jpg type type
```

Description:

Dump a high-quality ray-traced image of the simulation every N timesteps as either a JPG or PPM file. A series of such images can easily be converted into an animated movie of your simulation; see further details below. Other dump styles store snapshots of numerical data asociated with particles in various formats, as discussed on the dump doc page.

Here are two sample images, rendered as 1024x1024 JPG files. Click to see the full-size images:

..

The filename suffix determines whether a JPG or PPM file is created. If the suffix is ".jpg" or ".jpeg", then a JPG file is created, else a PPM file is created, which is a text-based format. To write out JPG files, you must build SPARTA with a JPEG library. See this section of the manual for instructions on how to do this.

Dumps are performed on timesteps that are a multiple of N (including timestep 0) and on the last timestep of a minimization if the minimization converges. Note that this means a dump will not be performed on the initial timestep after the dump command is invoked, if the current timestep is not a multiple of N. This behavior can be changed via the dump_modify first command, which can be useful if the dump command is invoked after a minimization ended on an arbitrary timestep. N can be changed between runs by using the dump_modify every command.

Dump image filenames must contain a wildcard character "*", so that one image file per snapshot is written. The "*" character is replaced with the timestep value. For example, tmp.dump.*.jpg becomes tmp.dump.0.jpg, tmp.dump.10000.jpg, tmp.dump.20000.jpg, etc. Note that the dump_modify pad command can be used to insure all timestep numbers are the same length (e.g. 00010), which can make it easier to convert a series of images into a movie in the correct ordering.

The *color* and *diameter* settings determine the color and size of particles rendered in the image. They can be any particle attribute defined for the dump paticle command, including *type*. Note that the *diameter* setting can be overridden with a numeric value by the optional *adiam* keyword, in which case you can specify the *diameter* setting with any valid particle attribute.

If *type* is specified for the *color* setting, then the color of each particle is determined by its type = species index. By default the mapping of types to colors is as follows:

- type 1 = red
- type 2 = green
- type 3 = blue
- type 4 = yellow
- type 5 = aqua
- type 6 = cyan

and repeats itself for types > 6. This mapping can be changed by the dump_modify acolor command.

If *type* is specified for the *diameter* setting then the diameter of each particle is determined by its type = species index. By default all types have diameter 1.0. This mapping can be changed by the dump_modify adiam command.

If other particle attributes are used for the *color* or *diameter* settings, they are interpreted in the following way.

If "vx", for example, is used as the *color* setting, then the color of the particle will depend on the x-component of its velocity. The association of a per-molecule value with a specific color is determined by a "color map", which can be specified via the dump_modify command. The basic idea is that the particle-attribute will be within a range of values, and every value within the range is mapped to a specific color. Depending on how the color map is

defined, that mapping can take place via interpolation so that a value of -3.2 is halfway between "red" and "blue", or discretely so that the value of -3.2 is "orange".

If "vx", for example, is used as the *diameter* setting, then the particle will be rendered using the x-component of its velocity as the diameter. If the per-molecule value <= 0.0, them the atom will not be drawn.

The various kewords listed above control how the image is rendered. As listed below, all of the keywords have defaults, most of which you will likely not need to change. The dump modify also has options specific to the dump image style, particularly for assigning colors to particles and other image features.

The *adiam* keyword allows you to override the *diameter* setting to a per-atom attribute with a specified numeric value. All particles will be drawn with that diameter, e.g. 1.5, which is in whatever distance units the input script defines.

The *atom* keyword allow you to turn off the drawing of all particles, if the specified value is *no*.

The *size* keyword sets the width and height of the created images, i.e. the number of pixels in each direction.

The *view*, *center*, *up*, *zoom*, and *persp* values determine how 3d simulation space is mapped to the 2d plane of the image. Basically they control how the simulation box appears in the image.

All of the *view*, *center*, *up*, *zoom*, and *persp* values can be specified as numeric quantities, whose meaning is explained below. Any of them can also be specified as an equal-style variable, by using v_name as the value, where "name" is the variable name. In this case the variable will be evaluated on the timestep each image is created to create a new value. If the equal-style variable is time-dependent, this is a means of changing the way the simulation box appears from image to image, effectively doing a pan or fly-by view of your simulation.

The *view* keyword determines the viewpoint from which the simulation box is viewed, looking towards the *center* point. The *theta* value is the vertical angle from the +z axis, and must be an angle from 0 to 180 degrees. The *phi* value is an azimuthal angle around the z axis and can be positive or negative. A value of 0.0 is a view along the +x axis, towards the *center* point. If *theta* or *phi* are specified via variables, then the variable values should be in degrees.

The *center* keyword determines the point in simulation space that will be at the center of the image. *Cx*, *Cy*, and *Cz* are speficied as fractions of the box dimensions, so that (0.5,0.5,0.5) is the center of the simulation box. These values do not have to be between 0.0 and 1.0, if you want the simulation box to be offset from the center of the image. Note, however, that if you choose strange values for *Cx*, *Cy*, or *Cz* you may get a blank image. Internally, *Cx*, *Cy*, and *Cz* are converted into a point in simulation space. If *flag* is set to "s" for static, then this conversion is done once, at the time the dump command is issued. If *flag* is set to "d" for dynamic then the conversion is performed every time a new image is created. If the box size or shape is changing, this will adjust the center point in simulation space.

The *up* keyword determines what direction in simulation space will be "up" in the image. Internally it is stored as a vector that is in the plane perpendicular to the view vector implied by the *theta* and *pni* values, and which is also in the plane defined by the view vector and user-specified up vector. Thus this internal vector is computed from the user-specified *up* vector as

```
up_internal = view cross (up cross view)
```

This means the only restriction on the specified *up* vector is that it cannot be parallel to the *view* vector, implied by the *theta* and *phi* values.

The *zoom* keyword scales the size of the simulation box as it appears in the image. The default *zfactor* value of 1 should display an image mostly filled by the atoms in the simulation box. A *zfactor* > 1 will make the simulation box larger; a *zfactor* < 1 will make it smaller. *Zfactor* must be a value > 0.0.

The *persp* keyword determines how much depth perspective is present in the image. Depth perspective makes lines that are parallel in simulation space appear non-parallel in the image. A *pfactor* value of 0.0 means that parallel lines will meet at infininty (1.0/pfactor), which is an orthographic rendering with no persepctive. A *pfactor* value between 0.0 and 1.0 will introduce more perspective. A *pfactor* value > 1 will create a highly skewed image with a large amount of perspective.

IMPORTANT NOTE: The *persp* keyword is not yet supported as an option.

The *box* keyword determines how the simulation box boundaries are rendered as thin cylinders in the image. If *no* is set, then the box boundaries are not drawn and the *diam* setting is ignored. If *yes* is set, the 12 edges of the box are drawn, with a diameter that is a fraction of the shortest box length in x,y,z (for 3d) or x,y (for 2d). The color of the box boundaries can be set with the dump_modify boxcolor command.

The *axes* keyword determines how the coordinate axes are rendered as thin cylinders in the image. If *no* is set, then the axes are not drawn and the *length* and *diam* settings are ignored. If *yes* is set, 3 thin cylinders are drawn to represent the x,y,z axes in colors red,green,blue. The origin of these cylinders will be offset from the lower left corner of the box by 10%. The *length* setting determines how long the cylinders will be as a fraction of the respective box lengths. The *diam* setting determines their thickness as a fraction of the shortest box length in x,y,z (for 3d) or x,y (for 2d).

The *shiny* keyword determines how shiny the objects rendered in the image will appear. The *sfactor* value must be a value 0.0 <= *sfactor* <= 1.0, where *sfactor* = 1 is a highly reflective surface and *sfactor* = 0 is a rough non-shiny surface.

The *ssao* keyword turns on/off a screen space ambient occlusion (SSAO) model for depth shading. If *yes* is set, then particles further away from the viewer are darkened via a randomized process, which is perceived as depth. The calculation of this effect can increase the cost of computing the image by roughly 2x. The strength of the effect can be scaled by the *dfactor* parameter. If *no* is set, no depth shading is performed.

A series of JPG or PPM images can be converted into a movie file and then played as a movie using commonly available tools.

Convert JPG or PPM files into an animated GIF or MPEG or other movie file:

- a) Use the ImageMagick convert program.

```
% convert *.jpg foo.gif
% convert *.ppm foo.mpg
```

- b) Use QuickTime.

  Select "Open Image Sequence" under the File menu Load the images into QuickTime to animate them Select "Export" under the File menu Save the movie as a QuickTime movie (*.mov) or in another format
- c) Windows-based tool.

If someone tells us how to do this via a common Windows-based tool, we'll post the instructions here.

Play the movie:

- a) Use your browser to view an animated GIF movie.

  Select "Open File" under the File menu Load the animated GIF file
- b) Use the freely available mplayer tool to view an MPEG movie.

```
% mplayer foo.mpg
```
- c) Use the Pizza.py animate tool, which works directly on a series of image files.

```
a = animate("foo*.jpg")
```
- d) QuickTime and other Windows-based media players can obviously play movie files directly.

Restrictions:

To write JPG images, you must use a -DSPARTA_JPEG switch when building SPARTA and link with a JPEG library. See the Making SPARTA section of the documentation for details.

Related commands:

dump, dump_modify, undump

Default:

The defaults for the keywords are as follows:

- adiam = not specified (use diameter setting)
- atom = yes
- size = 512 512
- view = 60 30 (for 3d)
- view = 0 0 (for 2d)
- center = s 0.5 0.5 0.5
- up = 0 0 1 (for 3d)
- up = 0 1 0 (for 2d)
- zoom = 1.0
- persp = 0.0
- box = yes 0.02
- axes = no 0.0 0.0
- shiny = 1.0
- ssao = no

dump_modify command

Syntax:

```
dump_modify dump-ID keyword values ...
```

- dump-ID = ID of dump to modify
- one or more keyword/value pairs may be appended
- keyword = *acolor* or *adiam* or *amap* or *append* or *bcolor* or *bdiam* or *backcolor* or *boxcolor* or *color* or *every* or *first* or *flush* or *format* or *thresh*

```
acolor args = type color
    type = atom type or range of types (see below)
    color = name of color or color1/color2/...
  adiam args = type diam
    type = atom type or range of types (see below)
    diam = diameter of atoms of that type (distance units)
  amap args = lo hi style delta N entry1 entry2 ... entryN
    lo = number or min = lower bound of range of color map
    hi = number or max = upper bound of range of color map
    style = 2 letters = "c" or "d" or "s" plus "a" or "f"
      "c" for continuous
      "d" for discrete
      "s" for sequential
      "a" for absolute
      "f" for fractional
    delta = binsize (only used for style "s", otherwise ignored)
      binsize = range is divided into bins of this width
    N = # of subsequent entries
    entry = value color (for continuous style)
      value = number or min or max = single value within range
      color = name of color used for that value
    entry = lo hi color (for discrete style)
      lo/hi = number or min or max = lower/upper bound of subset of range
      color = name of color used for that subset of values
    entry = color (for sequential style)
      color = name of color used for a bin of values
  append arg = yes or no
  backcolor arg = color
    color = name of color for background
  boxcolor arg = color
    color = name of color for box lines
  color args = name R G B
    name = name of color
    R,G,B = red/green/blue numeric values from 0.0 to 1.0
  every arg = N
    N = dump every this many timesteps
    N can be a variable (see below)
  first arg = yes or no
  format arg = C-style format string for one line of output
  flush arg = yes or no
  pad arg = Nchar = # of characters to convert timestep to
  thresh args = attribute operation value
    attribute = same attributes (x,fy,etotal,sxx,etc) used by dump custom style
    operation = "" or ">=" or "==" or "!="
    value = numeric value to compare to
    these 3 args can be replaced by the word "none" to turn off thresholding
```

Examples:

```
dump_modify 1 format "%d %d %20.15g %g %g"
dump_modify myDump thresh x <0.0 thresh vx >= 3.0
dump_modify 1 every 1000
dump_modify 1 every v_myVar
dump_modify 1 amap min max cf 0.0 3 min green 0.5 yellow max blue boxcolor red
```

Description:

Modify the parameters of a previously defined dump command. Not all parameters are relevant to all dump styles.

The *acolor* keyword applies only to the dump *image* style. It can be used with the dump image command, when its atom color setting is *type*, to set the color that atoms of each type will be drawn in the image.

The specified *type* should be an integer from 1 to Ntypes = the number of atom types. A wildcard asterisk can be used in place of or in conjunction with the *type* argument to specify a range of atom types. This takes the form "*" or "*n" or "n*" or "m*n". If N = the number of atom types, then an asterisk with no numeric values means all types from 1 to N. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to N (inclusive). A middle asterisk means all types from m to n (inclusive).

The specified *color* can be a single color which is any of the 140 pre-defined colors (see below) or a color name defined by the dump_modify color option. Or it can be two or more colors separated by a "/" character, e.g. red/green/blue. In the former case, that color is assigned to all the specified atom types. In the latter case, the list of colors are assigned in a round-robin fashion to each of the specified atom types.

The *adiam* keyword applies only to the dump *image* style. It can be used with the dump image command, when its atom diameter setting is *type*, to set the size that atoms of each type will be drawn in the image. The specified *type* should be an integer from 1 to Ntypes. As with the *acolor* keyword, a wildcard asterisk can be used as part of the *type* argument to specify a range of atomt types. The specified *diam* is the size in whatever distance units the input script is using, e.g. Angstroms.

The *amap* keyword applies only to the dump *image* style. It can be used with the dump image command, with its *atom* keyword, when its atom setting is an atom-attribute, to setup a color map. The color map is used to assign a specific RGB (red/green/blue) color value to an individual atom when it is drawn, based on the atom's attribute, which is a numeric value, e.g. its x-component of velocity if the atom-attribute "vx" was specified.

The basic idea of a color map is that the atom-attribute will be within a range of values, and that range is associated with a a series of colors (e.g. red, blue, green). An atom's specific value (vx = -3.2) can then mapped to the series of colors (e.g. halfway between red and blue), and a specific color is determined via an interpolation procedure.

There are many possible options for the color map, enabled by the *amap* keyword. Here are the details.

The *lo* and *hi* settings determine the range of values allowed for the atom attribute. If numeric values are used for *lo* and/or *hi*, then values that are lower/higher than that value are set to the value. I.e. the range is static. If *lo* is specified as *min* or *hi* as *max* then the range is dynamic, and the lower and/or upper bound will be calculated each time an image is drawn, based on the set of atoms being visualized.

The *style* setting is two letters, such as "ca". The first letter is either "c" for continuous, "d" for discrete, or "s" for sequential. The second letter is either "a" for absolute, or "f" for fractional.

A continuous color map is one in which the color changes continuously from value to value within the range. A discrete color map is one in which discrete colors are assigned to sub-ranges of values within the range. A sequential color map is one in which discrete colors are assigned to a sequence of sub-ranges of values covering

the entire range.

An absolute color map is one in which the values to which colors are assigned are specified explicitly as values within the range. A fractional color map is one in which the values to which colors are assigned are specified as a fractional portion of the range. For example if the range is from -10.0 to 10.0, and the color red is to be assigned to atoms with a value of 5.0, then for an absolute color map the number 5.0 would be used. But for a fractional map, the number 0.75 would be used since 5.0 is 3/4 of the way from -10.0 to 10.0.

The *delta* setting is only specified if the style is sequential. It specifies the bin size to use within the range for assigning consecutive colors to. For example, if the range is from -10.0 to 10.0 and a *delta* of 1.0 is used, then 20 colors will be assigned to the range. The first will be from -10.0 <= color1 < -9.0, then 2nd from -9.0 <= color2 < -8.0, etc.

The *N* setting is how many entries follow. The format of the entries depends on whether the color map style is continuous, discrete or sequential. In all cases the *color* setting can be any of the 140 pre-defined colors (see below) or a color name defined by the dump_modify color option.

For continuous color maps, each entry has a *value* and a *color*. The *value* is either a number within the range of values or *min* or *max*. The *value* of the first entry must be *min* and the *value* of the last entry must be *max*. Any entries in between must have increasing values. Note that numeric values can be specified either as absolute numbers or as fractions (0.0 to 1.0) of the range, depending on the "a" or "f" in the style setting for the color map.

Here is how the entries are used to determine the color of an individual atom, given the value X of its atom attribute. X will fall between 2 of the entry values. The color of the atom is linearly interpolated (in each of the RGB values) between the 2 colors associated with those entries. For example, if X = -5.0 and the 2 surrounding entries are "red" at -10.0 and "blue" at 0.0, then the atom's color will be halfway between "red" and "blue", which happens to be "purple".

For discrete color maps, each entry has a *lo* and *hi* value and a *color*. The *lo* and *hi* settings are either numbers within the range of values or *lo* can be *min* or *hi* can be *max*. The *lo* and *hi* settings of the last entry must be *min* and *max*. Other entries can have any *lo* and *hi* values and the sub-ranges of different values can overlap. Note that numeric *lo* and *hi* values can be specified either as absolute numbers or as fractions (0.0 to 1.0) of the range, depending on the "a" or "f" in the style setting for the color map.

Here is how the entries are used to determine the color of an individual atom, given the value X of its atom attribute. The entries are scanned from first to last. The first time that *lo* <= X <= *hi*, X is assigned the color associated with that entry. You can think of the last entry as assigning a default color (since it will always be matched by X), and the earlier entries as colors that override the default. Also note that no interpolation of a color RGB is done. All atoms will be drawn with one of the colors in the list of entries.

For sequential color maps, each entry has only a *color*. Here is how the entries are used to determine the color of an individual atom, given the value X of its atom attribute. The range is partitioned into N bins of width *binsize*. Thus X will fall in a specific bin from 1 to N, say the Mth bin. If it falls on a boundary between 2 bins, it is considered to be in the higher of the 2 bins. Each bin is assigned a color from the E entries. If E < N, then the colors are repeated. For example if 2 entries with colors red and green are specified, then the odd numbered bins will be red and the even bins green. The color of the atom is the color of its bin. Note that the sequential color map is really a shorthand way of defining a discrete color map without having to specify where all the bin boundaries are.

The *append* keyword applies to all dump styles except *image*. It also applies only to text output files, not to binary or gzipped files. If specified as *yes*, then dump snapshots are appended to the end of an existing dump file. If specified as *no*, then a new dump file will be created which will overwrite an existing file with the same name.

This keyword can only take effect if the dump_modify command is used after the dump command, but before the first command that causes dump snapshots to be output, e.g. a run command. Once the dump file has been opened, this keyword has no further effect.

The *backcolor* keyword applies only to the dump *image* style. It sets the background color of the images. The color name can be any of the 140 pre-defined colors (see below) or a color name defined by the dump_modify color option.

The *boxcolor* keyword applies only to the dump *image* style. It sets the color of the simulation box drawn around the atoms in each image. See the "dump image box" command for how to specify that a box be drawn. The color name can be any of the 140 pre-defined colors (see below) or a color name defined by the dump_modify color option.

The *color* keyword applies only to the dump *image* style. It allows definition of a new color name, in addition to the 140-predefined colors (see below), and associates 3 red/green/blue RGB values with that color name. The color name can then be used with any other dump_modify keyword that takes a color name as a value. The RGB values should each be floating point values between 0.0 and 1.0 inclusive.

When a color name is converted to RGB values, the user-defined color names are searched first, then the 140 pre-defined color names. This means you can also use the *color* keyword to overwrite one of the pre-defined color names with new RBG values.

The *every* keyword changes the dump frequency originally specified by the dump command to a new value. The every keyword can be specified in one of two ways. It can be a numeric value in which case it must be > 0. Or it can be an equal-style variable, which should be specified as v_name, where name is the variable name. In this case, the variable is evaluated at the beginning of a run to determine the next timestep at which a dump snapshot will be written out. On that timestep, the variable will be evaluated again to determine the next timestep, etc. Thus the variable should return timestep values. See the stagger() and logfreq() math functions for equal-style variables, as examples of useful functions to use in this context. Other similar math functions could easily be added as options for equal-style variables. When using the variable option with the *every* keyword, you also need to use the *first* option if you want an initial snapshot written to the dump file. The *every* keyword cannot be used with the dump *dcd* style.

For example, the following commands will write snapshots at timesteps 0,10,20,30,100,200,300,1000,2000,etc:

```
variable        s equal logfreq(10,3,10)
dump            1 all atom 100 tmp.dump
dump_modify     1 every v_s first yes
```

The *first* keyword determines whether a dump snapshot is written on the very first timestep after the dump command is invoked. This will always occur if the current timestep is a multiple of N, the frequency specified in the dump command, including timestep 0. But if this is not the case, a dump snapshot will only be written if the setting of this keyword is *yes*. If it is *no*, which is the default, then it will not be written.

The *flush* keyword determines whether a flush operation is invoked after a dump snapshot is written to the dump file. A flush insures the output in that file is current (no buffering by the OS), even if SPARTA halts before the simulation completes. Flushes cannot be performed with dump style *xtc*.

The text-based dump styles have a default C-style format string which simply specifies %d for integers and %g for real values. The *format* keyword can be used to override the default with a new C-style format string. Do not include a trailing "\n" newline character in the format string.

The *pad* keyword only applies when the dump filename is specified with a wildcard "*" character which becomes the timestep. If *pad* is 0, which is the default, the timestep is converted into a string of unpadded length, e.g. 100 or 12000 or 2000000. When *pad* is specified with *Nchar* > 0, the string is padded with leading zeroes so they are all the same length = *Nchar*. For example, pad 7 would yield 0000100, 0012000, 2000000. This can be useful so that post-processing programs can easily read the files in ascending timestep order.

The *thresh* keyword only applies to the dump *custom* and *image* styles. Multiple thresholds can be specified. Specifying "none" turns off all threshold criteria. If thresholds are specified, only atoms whose attributes meet all the threshold criteria are written to the dump file or included in the image. The possible attributes that can be tested for are the same as those that can be specified in the dump particle command. Note that different attributes can be output by the dump particle command than are used as threshold criteria by the dump_modify command. E.g. you can output the coordinates of atoms whose velocity components are above some threshold.

Restrictions: none

Related commands:

dump, dump image, undump

Default:

The option defaults are

- acolor = * red/green/blue/yellow/aqua/cyan
- adiam = * 1.0
- amap = min max cf 2 min blue max red
- append = no
- bcolor = * red/green/blue/yellow/aqua/cyan
- bdiam = * 0.5
- backcolor = black
- boxcolor = yellow
- color = 140 color names are pre-defined as listed below
- every = whatever it was set to via the dump command
- first = no
- flush = yes
- format = %d and %g for each integer or floating point value
- pad = 0
- thresh = none

These are the 140 colors that SPARTA pre-defines for use with the dump image and dump_modify commands. Additional colors can be defined with the dump_modify color command. The 3 numbers listed for each name are the RGB (red/green/blue) values. Divide each value by 255 to get the equivalent 0.0 to 1.0 value.

aliceblue = 240, 248, 255	antiquewhite = 250, 235, 215	aqua = 0, 255, 255	aquamarine = 127, 255, 212	azure = 240, 255, 255
beige = 245, 245, 220	bisque = 255, 228, 196	black = 0, 0, 0	blanchedalmond = 255, 255, 205	blue = 0, 0, 255
blueviolet = 138, 43, 226	brown = 165, 42, 42	burlywood = 222, 184, 135	cadetblue = 95, 158, 160	chartreuse = 127, 255, 0
chocolate = 210, 105, 30	coral = 255, 127, 80	cornflowerblue = 100, 149, 237	cornsilk = 255, 248, 220	crimson = 220, 20, 60

cyan = 0, 255, 255	darkblue = 0, 0, 139	darkcyan = 0, 139, 139	darkgoldenrod = 184, 134, 11	darkgray = 169, 169, 169
darkgreen = 0, 100, 0	darkkhaki = 189, 183, 107	darkmagenta = 139, 0, 139	darkolivegreen = 85, 107, 47	darkorange = 255, 140, 0
darkorchid = 153, 50, 204	darkred = 139, 0, 0	darksalmon = 233, 150, 122	darkseagreen = 143, 188, 143	darkslateblue = 72, 61, 139
darkslategray = 47, 79, 79	darkturquoise = 0, 206, 209	darkviolet = 148, 0, 211	deeppink = 255, 20, 147	deepskyblue = 0, 191, 255
dimgray = 105, 105, 105	dodgerblue = 30, 144, 255	firebrick = 178, 34, 34	floralwhite = 255, 250, 240	forestgreen = 34, 139, 34
fuchsia = 255, 0, 255	gainsboro = 220, 220, 220	ghostwhite = 248, 248, 255	gold = 255, 215, 0	goldenrod = 218, 165, 32
gray = 128, 128, 128	green = 0, 128, 0	greenyellow = 173, 255, 47	honeydew = 240, 255, 240	hotpink = 255, 105, 180
indianred = 205, 92, 92	indigo = 75, 0, 130	ivory = 255, 240, 240	khaki = 240, 230, 140	lavender = 230, 230, 250
lavenderblush = 255, 240, 245	lawngreen = 124, 252, 0	lemonchiffon = 255, 250, 205	lightblue = 173, 216, 230	lightcoral = 240, 128, 128
lightcyan = 224, 255, 255	lightgoldenrodyellow = 250, 250, 210	lightgreen = 144, 238, 144	lightgrey = 211, 211, 211	lightpink = 255, 182, 193
lightsalmon = 255, 160, 122	lightseagreen = 32, 178, 170	lightskyblue = 135, 206, 250	lightslategray = 119, 136, 153	lightsteelblue = 176, 196, 222
lightyellow = 255, 255, 224	lime = 0, 255, 0	limegreen = 50, 205, 50	linen = 250, 240, 230	magenta = 255, 0, 255
maroon = 128, 0, 0	mediumaquamarine = 102, 205, 170	mediumblue = 0, 0, 205	mediumorchid = 186, 85, 211	mediumpurple = 147, 112, 219
mediumseagreen = 60, 179, 113	mediumslateblue = 123, 104, 238	mediumspringgreen = 0, 250, 154	mediumturquoise = 72, 209, 204	mediumvioletred = 199, 21, 133
midnightblue = 25, 25, 112	mintcream = 245, 255, 250	mistyrose = 255, 228, 225	moccasin = 255, 228, 181	navajowhite = 255, 222, 173
navy = 0, 0, 128	oldlace = 253, 245, 230	olive = 128, 128, 0	olivedrab = 107, 142, 35	orange = 255, 165, 0
orangered = 255, 69, 0	orchid = 218, 112, 214	palegoldenrod = 238, 232, 170	palegreen = 152, 251, 152	paleturquoise = 175, 238, 238
palevioletred = 219, 112, 147	papayawhip = 255, 239, 213	peachpuff = 255, 239, 213	peru = 205, 133, 63	pink = 255, 192, 203
plum = 221, 160, 221	powderblue = 176, 224, 230	purple = 128, 0, 128	red = 255, 0, 0	rosybrown = 188, 143, 143
royalblue = 65, 105, 225	saddlebrown = 139, 69, 19	salmon = 250, 128, 114	sandybrown = 244, 164, 96	seagreen = 46, 139, 87
seashell = 255, 245, 238	sienna = 160, 82, 45	silver = 192, 192, 192	skyblue = 135, 206, 235	slateblue = 106, 90, 205
slategray = 112, 128, 144	snow = 255, 250, 250	springgreen = 0, 255, 127	steelblue = 70, 130, 180	tan = 210, 180, 140
teal = 0, 128, 128	thistle = 216, 191, 216	tomato = 253, 99, 71	turquoise = 64, 224, 208	violet = 238, 130, 238
	white = 255, 255, 255		yellow = 255, 255, 0	

wheat = 245, 222, 179		whitesmoke = 245, 245, 245		yellowgreen = 154, 205, 50

echo command

Syntax:

```
echo style
```

- style = *none* or *screen* or *log* or *both*

Examples:

```
echo both
echo log
```

Description:

This command determines whether SPARTA echoes each input script command to the screen and/or log file as it is read and processed. If an input script has errors, it can be useful to look at echoed output to see the last command processed.

The command-line switch -echo can be used in place of this command.

Restrictions: none

Related commands: none

Default:

```
echo log
```

fix command

Syntax:

```
fix ID style args
```

- ID = user-assigned name for the fix
- style = one of a long list of possible style names (see below)
- args = arguments used by a particular style

Examples:

```
fix 1 grid/check 100
```

Description:

Set a fix that will be applied to the system. In SPARTA, a "fix" is any user-specified operation that is applied to the system during timestepping. Examples include adding particles via inlet boundary conditions, enforcing boundary conditions, or computing diagnostics. There are dozens of fixes defined in SPARTA and new ones can be added; see this section for a discussion.

Fixes perform their operations at different stages of the timestep. If 2 or more fixes operate at the same stage of the timestep, they are invoked in the order they were specified in the input script.

The ID of a fix can only contain alphanumeric characters and underscores.

Fixes can be deleted with the unfix command.

IMPORTANT NOTE: The unfix command is the only way to turn off a fix; simply specifying a new fix with a similar style will not turn off the first one.

If you specify a new fix with the same ID and style as an existing fix, the old fix is deleted and the new one is created (presumably with new settings). This is the same as if an "unfix" command were first performed on the old fix, except that the new fix is kept in the same order relative to the existing fixes as the old one originally was. Note that this operation also wipes out any additional changes made to the old fix via the fix_modify command.

The fix modify command allows settings for some fixes to be reset. See the doc page for individual fixes for details.

Some fixes store an internal "state" which is written to binary restart files via the restart or write_restart commands. This allows the fix to continue on with its calculations in a restarted simulation. See the read_restart command for info on how to re-specify a fix in an input script that reads a restart file. See the doc pages for individual fixes for info on which ones can be restarted.

Some fixes calculate one of three styles of quantities: global, per-molecule, or per-grid, which can be used by other commands or output as described below. A global quantity is one or more system-wide values, e.g. the energy of a wall interacting with particles. A per-molecule quantity is one or more values per particle, e.g. the displacement vector for each particle since time 0. Per-cell quantities are calculated by each processor based on the grid cells it owns.

Note that a single fix may produces either global or per-atom or per-grid quantities (or none at all), but never more than one of these.

Global, per-atom, and local quantities each come in three kinds: a single scalar value, a vector of values, or a 2d array of values. The doc page for each fix describes the style and kind of values it produces, e.g. a per-atom vector. Some fixes produce more than one kind of a single style, e.g. a global scalar and a global vector.

When a fix quantity is accessed, as in many of the output commands discussed below, it can be referenced via the following bracket notation, where ID is the ID of the fix:

f_ID	entire scalar, vector, or array
f_ID[I]	one element of vector, one column of array
f_ID[I][J]	one element of array

In other words, using one bracket reduces the dimension of the quantity once (vector -> scalar, array -> vector). Using two brackets reduces the dimension twice (array -> scalar). Thus a command that uses scalar fix values as input can also process elements of a vector or array.

Note that commands and variables which use fix quantities typically do not allow for all kinds, e.g. a command may require a vector of values, not a scalar. This means there is no ambiguity about referring to a fix quantity as f_ID even if it produces, for example, both a scalar and vector. The doc pages for various commands explain the details.

In SPARTA, the values generated by a fix can be used in several ways:

- Global values can be output via the stats_style command. Or the values can be referenced in a variable equal or variable atom command.
- Per-molecule values can be output via the dump particle command. Or the per-molecule values can be referenced in an molecule-style variable.
- Per-grid values can be output via the dump grid command. Or the per-grid values can be referenced in a grid-style variable.

Each fix style has its own documentation page which describes its arguments and what it does, as listed below. Here is an alphabetic list of fix styles available in SPARTA:

- ave/grid - compute per-grid time-averaged quantities
- ave/time - compute/output global time-averaged quantities
- grid/check - check if particles are in the correct grid cell
- inflow - inject molecules at global boundaries
- print - print text and variables during a simulation

Restrictions: none

Related commands:

unfix

Default: none

fix ave/grid command

Syntax:

```
fix ID ave/grid Nevery Nrepeat Nfreq value1 value2 ... keyword args ...
```

- ID is documented in fix command
- ave/grid = style name of this fix command
- Nevery = use input values every this many timesteps
- Nrepeat = # of times to use input values for calculating averages
- Nfreq = calculate averages every this many timesteps zero or more input values can be listed
- value = c_ID, c_ID[i], f_ID, f_ID[i], v_name

```
c_ID = per-grid vector (or array) calculated by a compute with ID
c_ID[I] = Ith column of per-grid array calculated by a compute with ID
f_ID = per-grid vector (or array) calculated by a fix with ID
f_ID[I] = Ith column of per-grid array calculated by a fix with ID
v_name = per-grid vector calculated by an grid-style variable with name
```

- zero or more keyword/arg pairs may be appended

```
keyword = ave
  ave args = one or running
    one = output a new average value every Nfreq steps
    running = accumulate average continuously
```

Examples:

```
fix 1 all ave/grid 10 20 1000 c_mine
fix 1 all ave/grid 1 100 100 c_2 ave running
```

These commands will dump standard averages for each species and each grid cell to a file every 1000 steps:

```
compute 1 grid species n u v w usq vsq wsq
fix 1 ave/grid 10 100 1000 c_1
dump 1 grid 1000 tmp.grid id f_1
```

Description:

Use one or more per-grid vectors as inputs every few timesteps, and average them grid cell by grid cell over longer timescales, applying appropriate normalization factors. The resulting per grid cell averages can be used by other output commands such as the dump grid command.

Each input value can be the result of a compute or fix or the evaluation of a grid-style variable. The compute, fix, or variable must produce a per-grid vector or array, not a global quantity or per-molecule quantity. If you wish to time-average global quantities from a compute, fix, or variable, then see the fix ave/time command.

Computes that produce per-grid vectors or arrays are those which have the word *grid* in their style name. See the doc pages for individual fixes to determine which ones produce per-grid vectors or arrays. Variables of style *grid* are the only ones that can be used with this fix since they produce per-grid vectors.

Each per-grid value of each input vector is averaged independently.

The *Nevery*, *Nrepeat*, and *Nfreq* arguments specify on what timesteps the input values will be used in order to contribute to the average. The final averaged quantities are generated on timesteps that are a multiple of *Nfreq*. The average is over *Nrepeat* quantities, computed in the preceding portion of the simulation every *Nevery* timesteps. *Nfreq* must be a multiple of *Nevery* and *Nevery* must be non-zero even if *Nrepeat* is 1. Also, the timesteps contributing to the average value cannot overlap, i.e. Nfreq > (Nrepeat-1)*Nevery is required.

For example, if Nevery=2, Nrepeat=6, and Nfreq=100, then values on timesteps 90,92,94,96,98,100 will be used to compute the final average on timestep 100. Similarly for timesteps 190,192,194,196,198,200 on timestep 200, etc.

If a value begins with "c_", a compute ID must follow which has been previously defined in the input script. If no bracketed term is appended, and the compute calculates a per-grid vector, then the per-grid vector is used. If *c_ID[N]* is used, then N must be in the range from 1-M, which will use the Nth column of the M-column per-grid array calculated by the compute. If *c_ID* is used, and the compute calculates a per-grid array, then it is the same as if the individual columns of the array had been listed one by one.

E.g. these 2 fix ave/grid commands are equivalent, since the compute grid command produces 7 columns of output using the *all* mixture-ID which has a single group:

```
compute 1 grid all n u v w usq vsq wsq
fix 1 ave/grid 10 100 1000 c_1
fix 1 ave/grid 10 100 1000 c_1[1] c_1[2] c_1[3] c_1[4] c_1[5] c_1[6] c_1[7]
```

Users can also write code for their own compute styles and add them to SPARTA.

If a value begins with "f_", a fix ID must follow which has been previously defined in the input script. If no bracketed term is appended, and the fix calculates a per-grid vector, then the per-grid vector is used. If *f_ID[N]* is used, then N must be in the range from 1-M, which will use the Nth column of the M-column per-grid array calculated by the fix. If *f_ID* is used, and the fix calculates a per-grid array, then it is the same as if the individual columns of the array had been listed one by one. See the example above for computes.

Note that some fixes only produce their values on certain timesteps, which must be compatible with *Nevery*, else an error will result. Users can also write code for their own fix styles and add them to SPARTA.

If a value begins with "v_", a variable name must follow which has been previously defined in the input script as a grid-style variable Variables of style *grid* can reference thermodynamic keywords, or invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of generating per-grid quantities to time average.

Additional optional keywords also affect the operation of this fix.

The *ave* keyword determines what happens to the accumulation of statistics every *Nfreq* timesteps.

If the *ave* setting is *one*, then the values produced on timesteps that are multiples of Nfreq are independent of each other. Normalization (as described above) is performed, and all tallies are zeroed before accumulating over the next *Nfreq* steps.

If the *ave* setting is *running*, then tallies are never zeroed. Thus the output at any *Nfreq* timestep is normalized over all previously accumulated samples since the fix was defined. The tallies can only be zeroed by deleting the fix via the unfix command, or by re-defining the fix, or by re-specifying it.

Restart, fix_modify, output info:

No information about this fix is written to binary restart files. None of the fix_modify options are relevant to this fix.

This fix produces a per-grid vector or array which can be accessed by various output commands. A vector is produced if only a single quantity is averaged by this fix. If two or more quantities are averaged, then an array of values is produced, where the number of columns is the number of quantities averaged. The per-grid values can only be accessed on timesteps that are multiples of *Nfreq* since that is when averaging is performed.

Restrictions: none

Related commands:

compute, fix ave/time, variable

Default:

The option defaults are ave = one.

fix ave/surf command

Syntax:

```
fix ID ave/surf Nevery Nrepeat Nfreq value1 value2 ... keyword args ...
```

- ID is documented in fix command
- ave/surf = style name of this fix command
- Nevery = use input values every this many timesteps
- Nrepeat = # of times to use input values for calculating averages
- Nfreq = calculate averages every this many timesteps zero or more input values can be listed
- value = standard, c_ID, c_ID[i], f_ID, f_ID[i], v_name

```
standard = pre-defined set of per-surf values = n u v w usq vsq wsq
NOTE: the following value options are not yet supported
c_ID = per-surf vector calculated by a compute with ID
c_ID[I] = Ith column of per-surf array calculated by a compute with ID
f_ID = per-surf vector calculated by a fix with ID
f_ID[I] = Ith column of per-surf array calculated by a fix with ID
v_name = per-surf vector calculated by a surf-style variable with name
```

- zero or more keyword/arg pairs may be appended

```
keyword = ave
  ave args = one or running
    one = output a new average value every Nfreq steps
    running = accumulate average continuously
```

Examples:

```
fix 1 all ave/surf 1 100 100 standard ave running
fix 1 all ave/surf 10 20 1000 c_mine
```

Description:

Use one or more per-surf vectors as inputs every few timesteps, and average them surface element by surface element by over longer timescales. The resulting per-surf averages can be used by other output commands such as the dump surf command.

Each input value can be a standard surface element attribute or can be the result of a compute or fix or the evaluation of a surf-style variable. In the latter cases, the compute, fix, or variable must produce a per-surf vector, not a global quantity or per-molecule quantity. If you wish to time-average global quantities from a compute, fix, or variable, then see the fix ave/time command.

Computes that produce per-surf vectors or arrays are those which have the word *surf* in their style name. See the doc pages for individual fixes to determine which ones produce per-surf vectors or arrays. Variables of style *surf* are the only ones that can be used with this fix since they produce per-surf vectors.

Each per-surf value of each input vector is averaged independently.

The *Nevery*, *Nrepeat*, and *Nfreq* arguments specify on what timesteps the input values will be used in order to contribute to the average. The final averaged quantities are generated on timesteps that are a multiple of *Nfreq*. The average is over *Nrepeat* quantities, computed in the preceding portion of the simulation every *Nevery*

timesteps. *Nfreq* must be a multiple of *Nevery* and *Nevery* must be non-zero even if *Nrepeat* is 1. Also, the timesteps contributing to the average value cannot overlap, i.e. Nfreq > (Nrepeat-1)*Nevery is required.

For example, if Nevery=2, Nrepeat=6, and Nfreq=100, then values on timesteps 90,92,94,96,98,100 will be used to compute the final average on timestep 100. Similarly for timesteps 190,192,194,196,198,200 on timestep 200, etc.

If the single value *standard* is listed then statistics will be tallied on 7 quantities per species:

```
n u v w usq vsq wsq
```

for a total of 7*Nsp values, where Nsp = # of species.

N is the count of molecules (of a species) which collided with a surface element in the current timestep. *U,v,w* are the velocities of the same molecules in the x,y,z directions. *Usq,vsq,wsq* are the squared velocities of the same molecules in the x,y,z directions. These quanties are continuously tallied until a timestep that is a multiple of *Nfreq*, then normalized for output. *N* is normalized by the number of contributing samples, so that the output value is average molecule count for a grid cell. The other 6 quantities are normalized by the number of contributing molecules, so the output values are average velocity or velocity-squared for a grid cell. See the *ave* keyword for a setting that affects whether the tallies are zeroed after each *Nfreq* output.

If a value begins with "c_", a compute ID must follow which has been previously defined in the input script. If no bracketed term is appended, the per-grid vector calculated by the compute is used. If a bracketed term containing an index I is appended, the Ith column of the per-grid array calculated by the compute is used. Users can also write code for their own compute styles and add them to SPARTA.

If a value begins with "f_", a fix ID must follow which has been previously defined in the input script. If no bracketed term is appended, the per-grid vector calculated by the fix is used. If a bracketed term containing an index I is appended, the Ith column of the per-grid array calculated by the fix is used. Note that some fixes only produce their values on certain timesteps, which must be compatible with *Nevery*, else an error will result. Users can also write code for their own fix styles and add them to SPARTA.

If a value begins with "v_", a variable name must follow which has been previously defined in the input script as a grid-style variable Variables of style *grid* can reference thermodynamic keywords, or invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of generating per-grid quantities to time average.

NOTE: say something about how compute, fix, variable quantities are normalized for output.

Additional optional keywords also affect the operation of this fix.

The *ave* keyword determines what happens to the accumulation of statistics every *Nfreq* timesteps.

If the *ave* setting is *one*, then the values produced on timesteps that are multiples of Nfreq are independent of each other. Normalization (as described above) is performed, and all tallies are zeroed before accumulating over the next *Nfreq* steps.

If the *ave* setting is *running*, then tallies are never zeroed. Thus the output at any *Nfreq* timestep is normalized over all previously accumulated samples since the fix was defined. The tallies can only be zeroed by deleting the fix via the unfix command, or by re-defining the fix, or by re-specifying it.

Restart, fix_modify, output info:

No information about this fix is written to binary restart files. None of the fix_modify options are relevant to this fix.

This fix produces a per-surf vector or array which can be accessed by various output commands. A vector is produced if only a single quantity is averaged by this fix. If two or more quantities are averaged, then an array of values is produced, where the number of columns is the number of quantities averaged. The per-surf values can only be accessed on timesteps that are multiples of *Nfreq* since that is when averaging is performed.

Restrictions: none

Related commands:

compute, fix ave/histo, fix ave/spatial, fix ave/time, variable

Default:

The option defaults are ave = one.

fix ave/time command

Syntax:

```
fix ID ave/time Nevery Nrepeat Nfreq value1 value2 ... keyword args ...
```

- ID is documented in fix command
- ave/time = style name of this fix command
- Nevery = use input values every this many timesteps
- Nrepeat = # of times to use input values for calculating averages
- Nfreq = calculate averages every this many timesteps
- one or more input values can be listed
- value = c_ID, c_ID[N], f_ID, f_ID[N], v_name

```
c_ID = global scalar or vector calculated by a compute with ID
c_ID[I] = Ith component of global vector or Ith column of global array calculated by a compu
f_ID = global scalar or vector calculated by a fix with ID
f_ID[I] = Ith component of global vector or Ith column of global array calculated by a fix v
v_name = global value calculated by an equal-style variable with name
```

- zero or more keyword/arg pairs may be appended
- keyword = *mode* or *file* or *ave* or *start* or *off* or *title1* or *title2* or *title3*

```
mode arg = scalar or vector
    scalar = all input values are global scalars
    vector = all input values are global vectors or global arrays
  ave args = one or running or window M
    one = output a new average value every Nfreq steps
    running = output cummulative average of all previous Nfreq steps
    window M = output average of M most recent Nfreq steps
  start args = Nstart
    Nstart = start averaging on this timestep
  off arg = M = do not average this value
    M = value # from 1 to Nvalues
  file arg = filename
    filename = name of file to output time averages to
  title1 arg = string
    string = text to print as 1st line of output file
  title2 arg = string
    string = text to print as 2nd line of output file
  title3 arg = string
    string = text to print as 3rd line of output file, only for vector mode
```

Examples:

```
fix 1 all ave/time 100 5 1000 c_myTemp c_thermo_temp file temp.profile
fix 1 all ave/time 100 5 1000 c_thermo_press[2] ave window 20 &
                              title1 "My output values"
fix 1 all ave/time 1 100 1000 f_indent f_indent[1] file temp.indent off 1
```

Description:

Use one or more global values as inputs every few timesteps, and average them over longer timescales. The resulting averages can be used by other output commands such as stats_style custom, and can also be written to a file. Note that if no time averaging is done, this command can be used as a convenient way to simply output one or more global values to a file.

Each listed value can be the result of a compute or fix or the evaluation of an equal-style variable. In each case, the compute, fix, or variable must produce a global quantity, not a per-molecule or per-grid quantity. If you wish to spatial- or time-average or histogram per-molecule quantities from a compute, fix, or variable, then see the fix ave/spatial, fix ave/atom, or fix ave/histo commands. If you wish to sum a per-molecule or per-grid quantity into a single global quantity, see the compute reduce command.

Computes that produce global quantities are those which do not have the word *praticle* or *cell* in their style name. Only a few fixes produce global quantities. See the doc pages for individual fixes for info on which ones produce such values. Variables of style *equal* are the only ones that can be used with this fix. Variables of style *particle* or *cell* cannot be used, since they produce per-molecule or per-grid values.

The input values must either be all scalars or all vectors (or arrays), depending on the setting of the *mode* keyword. In both cases, the averaging is performed independently on each input value. I.e. each input scalar is averaged independently and each element of each input vector (or array) is averaged independently.

If *mode* = vector, then the input values may either be vectors or arrays and all must be the same "length", which is the length of the vector or number of rows in the array. If a global array is listed, then it is the same as if the individual columns of the array had been listed one by one. E.g. these 2 fix ave/time commands are equivalent, since the compute rdf command creates, in this case, a global array with 3 columns, each of length 50:

```
compute myRDF all rdf 50 1 2
fix 1 all ave/time 100 1 100 c_myRDF file tmp1.rdf mode vector
fix 2 all ave/time 100 1 100 c_myRDF[1] c_myRDF[2] c_myRDF[3] file tmp2.rdf mode vector
```

The *Nevery*, *Nrepeat*, and *Nfreq* arguments specify on what timesteps the input values will be used in order to contribute to the average. The final averaged quantities are generated on timesteps that are a mlutiple of *Nfreq*. The average is over *Nrepeat* quantities, computed in the preceding portion of the simulation every *Nevery* timesteps. *Nfreq* must be a multiple of *Nevery* and *Nevery* must be non-zero even if *Nrepeat* is 1. Also, the timesteps contributing to the average value cannot overlap, i.e. Nfreq > (Nrepeat-1)*Nevery is required.

For example, if Nevery=2, Nrepeat=6, and Nfreq=100, then values on timesteps 90,92,94,96,98,100 will be used to compute the final average on timestep 100. Similarly for timesteps 190,192,194,196,198,200 on timestep 200, etc. If Nrepeat=1 and Nfreq = 100, then no time averaging is done; values are simply generated on timesteps 100,200,etc.

If a value begins with "c_", a compute ID must follow which has been previously defined in the input script. If *mode* = scalar, then if no bracketed term is appended, the global scalar calculated by the compute is used. If a bracketed term is appended, the Ith element of the global vector calculated by the compute is used. If *mode* = vector, then if no bracketed term is appended, the global vector calculated by the compute is used. Or if the compute calculates an array, all of the columns of the global array are used as if they had been specified as individual vectors (see description above). If a bracketed term is appended, the Ith column of the global array calculated by the compute is used.

Note that there is a compute reduce command which can sum per-atom quantities into a global scalar or vector which can thus be accessed by fix ave/time. Or it can be a compute defined not in your input script, but by thermodynamic output or other fixes such as fix nvt or fix temp/rescale. See the doc pages for these commands which give the IDs of these computes. Users can also write code for their own compute styles and add them to SPARTA.

If a value begins with "f_", a fix ID must follow which has been previously defined in the input script. If *mode* = scalar, then if no bracketed term is appended, the global scalar calculated by the fix is used. If a bracketed term is appended, the Ith element of the global vector calculated by the fix is used. If *mode* = vector, then if no bracketed term is appended, the global vector calculated by the fix is used. Or if the fix calculates an array, all of the

columns of the global array are used as if they had been specified as individual vectors (see description above). If a bracketed term is appended, the Ith column of the global array calculated by the fix is used.

Note that some fixes only produce their values on certain timesteps, which must be compatible with *Nevery*, else an error will result. Users can also write code for their own fix styles and add them to SPARTA.

If a value begins with "v_", a variable name must follow which has been previously defined in the input script. Variables can only be used as input for *mode* = scalar. Only equal-style variables can be referenced. See the variable command for details. Note that variables of style *equal* define a formula which can reference individual atom properties or thermodynamic keywords, or they can invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of specifying quantities to time average.

Additional optional keywords also affect the operation of this fix.

If the *mode* keyword is set to *scalar*, then all input values must be global scalars, or elements of global vectors. If the *mode* keyword is set to *vector*, then all input values must be global vectors, or columns of global arrays. They can also be global arrays, which are converted into a series of global vectors (one per column), as explained above.

The *ave* keyword determines how the values produced every *Nfreq* steps are averaged with values produced on previous steps that were multiples of *Nfreq*, before they are accessed by another output command or written to a file.

If the *ave* setting is *one*, then the values produced on timesteps that are multiples of *Nfreq* are independent of each other; they are output as-is without further averaging.

If the *ave* setting is *running*, then the values produced on timesteps that are multiples of *Nfreq* are summed and averaged in a cummulative sense before being output. Each output value is thus the average of the value produced on that timestep with all preceding values. This running average begins when the fix is defined; it can only be restarted by deleting the fix via the unfix command, or by re-defining the fix by re-specifying it.

If the *ave* setting is *window*, then the values produced on timesteps that are multiples of *Nfreq* are summed and averaged within a moving "window" of time, so that the last M values are used to produce the output. E.g. if M = 3 and Nfreq = 1000, then the output on step 10000 will be the average of the individual values on steps 8000,9000,10000. Outputs on early steps will average over less than M values if they are not available.

The *start* keyword specifies what timestep averaging will begin on. The default is step 0. Often input values can be 0.0 at time 0, so setting *start* to a larger value can avoid including a 0.0 in a running or windowed average.

The *off* keyword can be used to flag any of the input values. If a value is flagged, it will not be time averaged. Instead the most recent input value will always be stored and output. This is useful if one of more of the inputs produced by a compute or fix or variable are effectively constant or are simply current values. E.g. they are being written to a file with other time-averaged values for purposes of creating well-formatted output.

The *file* keyword allows a filename to be specified. Every *Nfreq* steps, one quantity or vector of quantities is written to the file for each input value specified in the fix ave/time command. For *mode* = scalar, this means a single line is written each time output is performed. Thus the file ends up to be a series of lines, i.e. one column of numbers for each input value. For *mode* = vector, an array of numbers is written each time output is performed. The number of rows is the length of the input vectors, and the number of columns is the number of values. Thus the file ends up to be a series of these array sections.

The *title1* and *title2* and *title3* keywords allow specification of the strings that will be printed as the first 2 or 3 lines of the output file, assuming the *file* keyword was used. SPARTA uses default values for each of these, so they do not need to be specified.

By default, these header lines are as follows for *mode* = scalar:

```
# Time-averaged data for fix ID
# TimeStep value1 value2 ...
```

In the first line, ID is replaced with the fix-ID. In the second line the values are replaced with the appropriate fields from the fix ave/time command. There is no third line in the header of the file, so the *title3* setting is ignored when *mode* = scalar.

By default, these header lines are as follows for *mode* = vector:

```
# Time-averaged data for fix ID
# TimeStep Number-of-rows
# Row value1 value2 ...
```

In the first line, ID is replaced with the fix-ID. The second line describes the two values that are printed at the first of each section of output. In the third line the values are replaced with the appropriate fields from the fix ave/time command.

Restart, fix_modify, output info:

No information about this fix is written to binary restart files. None of the fix_modify options are relevant to this fix.

This fix produces a global scalar or global vector or global array which can be accessed by various output commands. The values can only be accessed on timesteps that are multiples of *Nfreq* since that is when averaging is performed.

A scalar is produced if only a single input value is averaged and *mode* = scalar. A vector is produced if multiple input values are averaged for *mode* = scalar, or a single input value for *mode* = vector. In the first case, the length of the vector is the number of inputs. In the second case, the length of the vector is the same as the length of the input vector. An array is produced if multiple input values are averaged and *mode* = vector. The global array has # of rows = length of the input vectors and # of columns = number of inputs.

Restrictions: none

Related commands:

compute, fix ave/atom, fix ave/spatial, fix ave/histo, variable

Default: none

The option defaults are mode = scalar, ave = one, start = 0, no file output, title 1,2,3 = strings as described above, and no off settings for any input values.

fix grid/check command

Syntax:

```
fix ID grid/check N
```

- ID is documented in fix command
- grid/check = style name of this fix command
- N = check every N timesteps

Examples:

```
fix 1 grid/check 100
```

Description:

Check if particles are inside the grid cell they are supposed to be, based on their current coordinates. This is useful as a debugging check to insure that no particles have been assigned to the incorrect grid cell during the particle move stage of the SPARTA algorithm.

The check is performed every *N* timesteps. Particles not inside the correct grid cell are counted and the value of the count can be monitored (see below). A value of 0 is "correct", meaning that no particle was found to be outside its assigned grid cell.

Restart, fix_modify, output info:

No information about this fix is written to binary restart files. None of the fix_modify options are relevant to this fix.

This fix computes a global scalar which can be accessed by various output commands. The scalar is the count of how many particles were not in the correct grid cell. The count is cummulative over all the timesteps the check was performed since the start of the run. It is initialized to zero each time a run is performed.

Restrictions: none

Related commands: none

Default: none

fix inflow command

Syntax:

```
fix ID inflow mix-ID face1 face2 ... keyword value ...
```

- ID is documented in fix command
- inflow = style name of this fix command
- mix-ID = ID of mixture to use when creating particles
- face1,face2,... = one or more of *all* or *xlo* or *xhi* or *ylo* or *yhi* or *zlo* or *zhi* zero or more keyword/value pairs may be appended
- keyword = *n* or *nevery* or *perspecies*

```
  n value = Np = number of particles to create
    nevery value = Nstep = insert every this many timesteps
    perspecies value = yes or no
```

Examples:

```
fix in inflow air all
fix in inflow mymix xlo yhi n 1000 nevery 10
```

Description:

Insert particles into the simulation box continuously during a simulation. The particles are inserted at the specified faces using properties of the specified mixture. If done every timestep, this has the effect of a continuous influx of particles thru the face(s).

The properties of the inserted particles are from the mixture with ID *mix-ID*. This determines what particle species are in the mixture at what relative fractions, and its number density, streaming velocity, and temperature. See the mixture command for details.

One or more faces of the simulation box can be specified via the *face1*, *face2*, etc arguments. There are 6 individual faces that can be specified as *xlo*, *xhi*, *ylo*, *yhi*, *zlo*, or *zhi*. Specifying *all* is the same as specifying all 6 individual faces.

On each insertion timestep, each cell with a face(s) adjacent to an inflow boundary performs the following computation at the beginning of the SPARTA timestep.

The number of particles *M* to be inserted on the face is calculated, which is a function of the global *Fnum* value, the mixture number density, the mixture streaming velocity and thermal temperature, the area of the face, and its orientation relative to the streaming velocity.

If *M* has a fractional value, e.g. 12.5, then 12 particles will be inserted, and a 13th depending on the outcome of a random number generation. Each particle will be inserted at a random location on the face. The particle species will be chosen randomly in accord with the *frac* settings of the collection of species in the mixture, as set by the mixture command. The velocity of the particle will be set to the sum of the streaming velocity of the mixture and a thermal velocity sampled from the thermal temperature of the mixture. Both the streaming velocity and thermal temperature are also set by the mixture command.

If the final particle velocity is not "into" the grid cell, then the velocity sampling is repeated until it is. This insures that *M* particles enter the simulation domain as desired.

The first timestep that inserted particles are advected, they move for a random fraction of the timestep. This insures a continuous flow field of particles enters the simulation box through that face.

The *n* keyword can be useful for debugging purposes. If *Np* is set to 0, then the count of inserted particles is a function of the *Fnum* and *nrho* and other mixture settings, as described above. If *Np* is set to a value > 0, then the *Fnum* and *nrho* settings are ignored, and *Np* particles are inserted. This is done by dividing *Np* by the total number of grid cells that are adjacent to the specified box faces and inserting an equal number of particles per grid cell.

The *nevery* keyword determines how often particles are inserted. If *Nstep* > 1, this will give non-continuous, clumpy behavior in the inlet flow field.

The *perspecies* keyword determines how the species of each inserted particle is randomly determined. It has an effect on the statistical properties of the inserted particles.

If *perspecies* is set to *yes*, then a target insertion number *M* is calculated for each species in a grid cell, which is a function of the relative number fraction of the species, as set by the mixture nfrac command. If *M* has a fractional value, e.g. 12.5, then 12 particles of that species will always be inserted, and a 13th depending on the outcome of a randomly generated number.

If *perspecies* is set to *no*, then a single target insertion number *M* is calculated for all the species in a grid cell. Each time a particle is inserted, a random number is used to choose the species of the particle, based on the relative number fractions of all the species in the mixture. As before, if *M* has a fractional value, e.g. 12.5, then 12 particles will always be inserted, and a 13th depending on the outcome of a randomly generated number.

Here is a simple example, illustrating the difference between the two options. Assume a mixture with 2 species, each with a relative number fraction of 0.5. Assume a particular grid cell inserts 10 particles from that mixture. If *perspecies* is set to *yes*, then exactly 5 particles of each species will be inserted on every timestep insertions take place. If *perspecies* is set to *no*, then exactly 10 particles will be inserted every time and on average the insertion will be 5 particles of each of the two species. But on one timestep it might be 6 of the first and 4 of the second. On another timestep it might be 3 of the first and 7 of the second.

Restart, fix_modify, output info:

No information about this fix is written to binary restart files. None of the fix_modify options are relevant to this fix.

This fix computes a global vector of length 2 which can be accessed by various output commands. The first element of the vector is the total number of particles inserted on the most recent insertion step. The second element is the cummulative total number inserted since the beginning of the run. The 2nd value is initialized to zero each time a run is performed.

Restrictions:

Particles cannot be inserted on *z* faces of the simluation box for a 2d simulation. Particles cannot be inserted on periodic faces of the simulation box.

A *n* setting of *Np* > 0 can only be used with a *perspecies* setting of *no*.

Related commands:

mixture, create_molecules

Default:

The keyword defaults are n = 0, nevery = 1, perspecies = yes.

fix print command

Syntax:

```
fix ID print N string keyword value ...
```

- ID is documented in fix command
- print = style name of this fix command
- N = print every N steps
- string = text string to print with optional variable names
- zero or more keyword/value pairs may be appended
- keyword = *file* or *append* or *screen* or *title*

```
file value = filename
  append value = filename
  screen value = yes or no
  title value = string
    string =  text to print as 1st line of output file
```

Examples:

```
fix extra print 100 "Coords of marker particle = $x $y $z"
fix extra print 100 "Coords of marker particle = $x $y $z" file coord.txt
```

Description:

Print a text string every N steps during a simulation run. This can be used for diagnostic purposes or as a debugging tool to monitor some quantity during a run. The text string must be a single argument, so it should be enclosed in double quotes if it is more than one word. If it contains variables it must be enclosed in double quotes to insure they are not evaluated when the input script line is read, but will instead be evaluated each time the string is printed.

See the variable command for a description of *equal* style variables which are the most useful ones to use with the fix print command, since they are evaluated afresh each timestep that the fix print line is output. Equal-style variables calculate formulas involving mathematical operations, statistical properties, global values calculated by a compute or fix, or references to other variables.

If the *file* or *append* keyword is used, a filename is specified to which the output generated by this fix will be written. If *file* is used, then the filename is overwritten if it already exists. If *append* is used, then the filename is appended to if it already exists, or created if it does not exist.

If the *screen* keyword is used, output by this fix to the screen and logfile can be turned on or off as desired.

The *title* keyword allow specification of the string that will be printed as the first line of the output file, assuming the *file* keyword was used. By default, the title line is as follows:

```
# Fix print output for fix ID
```

where ID is replaced with the fix-ID.

Restart, fix_modify, output info:

No information about this fix is written to binary restart files. None of the fix_modify options are relevant to this fix. No global or per-molecule or per-grid quantities are stored by this fix for access by various output commands.

Restrictions: none

Related commands:

variable, print

Default:

The option defaults are no file output, screen = yes, and title string as described above.

global command

Syntax:

```
global keyword values
```

- keyword = *fnum* or *nrho* or *vstream* or *temp*

```
fnum value = ratio
    ratio = Fnum ratio of physical particles to simulation particles
  nrho value = density
    density = number density of background gas (# per length^3 units)
  vstream values = Vx Vy Vz
    Vx,Vy,Vz = streaming velocity of background gas (velocity units)
  temp values = thermal
    thermal = temperature of background gas (temperature units)
```

Examples:

```
background fnum 1.0e20
background vstream 100.0 0 0
background temp 1000
```

Description:

Define global properties of the system.

The *fnum* keyword sets the ratio of real, physical particles (i.e. molecules) to simulation particles.

The *nrho* keyword sets the number density of the background gas. For 3d simulations the units are #/volume. For 2d, the units are effectively #/area.

Assuming your simulation is populated by particles from the background gas, the *fnum* and *nrho* settings determine how many particles will be present in your simulation.

The *vstream* keyword sets the streaming velocity of the background gas.

The *temp* keyword sets the thermal temperature of the background gas. This is a Gaussian velocity distribution superposed on top of the streaming velocity.

Restrictions: none

Related commands:

mixture

Default:

The keyword defaults are fnum = 1.0, nrho = 1.0, vstream = 0.0 0.0 0.0, temp = 300.

if command

Syntax:

```
if boolean then t1 t2 ... elif boolean f1 f2 ... elif boolean f1 f2 ... else e1 e2 ...
```

- boolean = a Boolean expression evaluated as TRUE or FALSE (see below)
- then = required word
- t1,t2,...,tN = one or more SPARTA commands to execute if condition is met, each enclosed in quotes
- elif = optional word, can appear multiple times
- f1,f2,...,fN = one or more SPARTA commands to execute if elif condition is met, each enclosed in quotes (optional arguments)
- else = optional argument
- e1,e2,...,eN = one or more SPARTA commands to execute if no condition is met, each enclosed in quotes (optional arguments)

Examples:

```
if "${steps} > 1000" then exit
if "$x <= $y" then "print X is smaller = $x" else "print Y is smaller = $y"
if "(${eng} > 0.0) || ($n <1000)" then &
  "timestep 0.005" &
elif $n ${eng_previous}" then "jump file1" else "jump file2"
```

Description:

This command provides an in-then-else capability within an input script. A Boolean expression is evaluted and the result is TRUE or FALSE. Note that as in the examples above, the expression can contain variables, as defined by the variable command, which will be evaluated as part of the expression. Thus a user-defined formula that reflects the current state of the simulation can be used to issue one or more new commands.

If the result of the Boolean expression is TRUE, then one or more commands (t1, t2, ..., tN) are executed. If it is FALSE, then Boolean expressions associated with successive elif keywords are evaluated until one is found to be true, in which case its commands (f1, f2, ..., fN) are executed. If no Boolean expression is TRUE, then the commands associated witht the else keyword, namely (e1, e2, ..., eN), are executed. The elif and else keywords and their associated commands are optional. If they aren't specified and the initial Boolean expression is FALSE, then no commands are executed.

The syntax for Boolean expressions is described below.

Each command (t1, f1, e1, etc) can be any valid SPARTA input script command. If the command is more than one word, it must enclosed in quotes, so it will be treated as a single argument, as in the examples above.

IMPORTANT NOTE: If a command itself requires a quoted argument (e.g. a print command), then double and single quotes can be used and nested in the usual manner, as in the examples above and below. See this section of the manual for more details on using quotes in arguments. Only one of level of nesting is allowed, but that should be sufficient for most use cases.

Note that by using the line continuation character "&", the if command can be spread across many lines, though it is still a single command:

```
if "$a <$b" then &
  "print 'Minimum value = $a'" &
  "run 1000" &
else &
  'print "Minimum value = $b"' &
  "minimize 0.001 0.001 1000 10000"
```

Note that if one of the commands to execute is an invalid SPARTA command, such as "exit" in the first example above, then executing the command will cause SPARTA to halt.

Note that by jumping to a label in the same input script, the if command can be used to break out of a loop. See the variable delete command for info on how to delete the associated loop variable, so that it can be re-used later in the input script.

Here is an example of a double loop which uses the if and jump commands to break out of the inner loop when a condition is met, then continues iterating thru the outer loop.

```
label       loopa
variable    a loop 5
  label     loopb
  variable  b loop 5
  print     "A,B = $a,$b"
  run       10000
  if        '$b > 2' then "print 'Jumping to another script'" "jump in.script break"
  next      b
  jump      in.script loopb
label       break
variable    b delete

next        a
jump        in.script loopa
```

The Boolean expressions for the if and elif keywords have a C-like syntax. Note that each expression is a single argument within the if command. Thus if you want to include spaces in the expression for clarity, you must enclose the entire expression in quotes.

An expression is built out of numbers:

```
0.2, 100, 1.0e20, -15.4, etc
```

and Boolean operators:

```
A == B, A != B, A <B, A  <= B, A > B, A >= B, A && B, A || B, !A
```

Each A and B is a number or a variable reference like $a or ${abc}, or another Boolean expression.

If a variable is used it must produce a number when evaluated and substituted for in the expression, else an error will be generated.

Expressions are evaluated left to right and have the usual C-style precedence: the unary logical NOT operator "!" has the highest precedence, the 4 relational operators "", and ">=" are next; the two remaining relational operators "==" and "!=" are next; then the logical AND operator "&&"; and finally the logical OR operator "||" has the lowest precedence. Parenthesis can be used to group one or more portions of an expression and/or enforce a different order of evaluation than what would occur with the default precedence.

The 6 relational operators return either a 1.0 or 0.0 depending on whether the relationship between x and y is TRUE or FALSE. The logical AND operator will return 1.0 if both its arguments are non-zero, else it returns 0.0. The logical OR operator will return 1.0 if either of its arguments is non-zero, else it returns 0.0. The logical NOT operator returns 1.0 if its argument is 0.0, else it returns 0.0.

The overall Boolean expression produces a TRUE result if the result is non-zero. If the result is zero, the expression result is FALSE.

Restrictions: none

Related commands:

variable, print

Default: none

include command

Syntax:

```
include file
```

- file = filename of new input script to switch to

Examples:

```
include newfile
include in.run2
```

Description:

This command opens a new input script file and begins reading SPARTA commands from that file. When the new file is finished, the original file is returned to. Include files can be nested as deeply as desired. If input script A includes script B, and B includes A, then SPARTA could run for a long time.

If the filename is a variable (see the variable command), different processor partitions can run different input scripts.

Restrictions: none

Related commands:

variable, jump

Default: none

jump command

Syntax:

```
jump file label
```

- file = filename of new input script to switch to
- label = optional label within file to jump to

Examples:

```
jump newfile
jump in.run2 runloop
jump SELF runloop
```

Description:

This command closes the current input script file, opens the file with the specified name, and begins reading SPARTA commands from that file. Unlike the include command, the original file is not returned to, although by using multiple jump commands it is possible to chain from file to file or back to the original file.

If the word "SELF" is used for the filename, then the current input script is re-opened and read again.

IMPORTANT NOTE: The SELF option is not guaranteed to work when the current input script is being read through stdin (standard input), e.g.

```
lmp_g++ <in.script
```

since the SELF option invokes the C-library rewind() call, which may not be supported for stdin on some systems. This can be worked around by using the -in command-line argument or the -var command-line argument to pass the script name as a variable to the input script In the latter case, the "fname" variable could be used in place of SELF. E.g.

```
lmp_g++ -in in.script

lmp_g++ -var fname n.script <in.script
```

The 2nd argument to the jump command is optional. If specified, it is treated as a label and the new file is scanned (without executing commands) until the label is found, and commands are executed from that point forward. This can be used to loop over a portion of the input script, as in this example. These commands perform 10 runs, each of 10000 steps, and create 10 dump files named file.1, file.2, etc. The next command is used to exit the loop after 10 iterations. When the "a" variable has been incremented for the tenth time, it will cause the next jump command to be skipped.

```
variable a loop 10
label loop
dump 1 all atom 100 file.$a
run 10000
undump 1
next a
jump in.lj loop
```

If the jump *file* argument is a variable, the jump command can be used to cause different processor partitions to run different input scripts. In this example, SPARTA is run on 40 processors, with 4 partitions of 10 procs each. An in.file containing the example variable and jump command will cause each partition to run a different simulation.

```
mpirun -np 40 lmp_ibm -partition 4x10 -in in.file

variable f world script.1 script.2 script.3 script.4
jump $f
```

Here is an example of a double loop which uses the if and jump commands to break out of the inner loop when a condition is met, then continues iterating thru the outer loop.

```
label       loopa
variable    a loop 5
  label     loopb
  variable  b loop 5
  print     "A,B = $a,$b"
  run       10000
  if        $b > 2 then "jump in.script break"
  next      b
  jump      in.script loopb
label       break
variable    b delete

next        a
jump        in.script loopa
```

Restrictions:

If you jump to a file and it does not contain the specified label, SPARTA will come to the end of the file and exit.

Related commands:

variable, include, label, next

Default: none

label command

Syntax:

```
label ID
```

- ID = string used as label name

Examples:

```
label xyz
label loop
```

Description:

Label this line of the input script with the chosen ID. Unless a jump command was used previously, this does nothing. But if a jump command was used with a label argument to begin invoking this script file, then all command lines in the script prior to this line will be ignored. I.e. execution of the script will begin at this line. This is useful for looping over a section of the input script as discussed in the jump command.

Restrictions: none

Related commands: none

Default: none

log command

Syntax:

```
log file
```

- file = name of new logfile

Examples:

```
log log.equil
```

Description:

This command closes the current SPARTA log file, opens a new file with the specified name, and begins logging information to it. If the specified file name is *none*, then no new log file is opened.

If multiple processor partitions are being used, the file name should be a variable, so that different processors do not attempt to write to the same log file.

The file "log.sparta" is the default log file for a SPARTA run. The name of the initial log file can also be set by the command-line switch -log. See this section for details.

Restrictions: none

Related commands: none

Default:

The default SPARTA log file is named log.sparta

mixture command

Syntax:

```
mixture ID species1 species2 ... keyword args ...
```

- ID = user-defined name of the mixture
- species1, species2, ... = zero or more species IDs to include in the mixture
- zero or more keyword/arg pairs may be appended
- keyword = *nrho* or *vstream* or *temp* or *frac* or *group*

```
nrho arg = density
    density = number density of entire mixture (# per length^3 units)
  vstream args = Vx Vy Vz
    Vx,Vy,Vz = streaming velocity of entire mixture (velocity units)
  temp arg = thermal
    thermal = temperature of entire mixture (temperature units)
  frac arg = fraction
    fraction = number fraction for each listed species (0 to 1)
  group arg = SELF or group-ID
    SELF = put each listed species (or all species if none listed) in its own group
    group-ID = put the listed species (or all species if none listed) in a group with this ID
```

Examples:

```
mixture air N O NO group lite
mixture air N O NO vstream 250.0 0.0 0.0 group species
mixture air N frac 0.8
mixture air O frac 0.2
mixture background N O
```

Description:

Define a gas mixture and its properties. A mixture can be referenced by its ID in several other SPARTA commands such as create_molecules or pre-grid computes. Any number of mixtures can be defined and used in a simulation.

The ID of a mixture can only contain alphanumeric characters and underscores.

A mixture is a collection of one or more particle species as defined by the species command. Each species belongs to a named group within the mixture so that particles of all species in the group can be acted on together by other commands. The mixture has global attributes and per-species attributes. All attributes have default values unless they are explicitly specified.

Note that the mixture command can be used multiple times with the same ID, to add species to the mixture, define groups within the mixture, or change its attributes.

There are 2 default mixtures defined by SPARTA that always exist.

The first default mixture has an ID = "all", and contains all species that have been defined. When new species are created via the "species" command, they are automatically added to this mixture. This mixture has only a single group, also named "all", which all species belong to.

The second default mixture has an ID = "species", and also contains all species that have been defined. When new species are created via the "species" command, they are also automatically added to this mixture. This mixture has one group per species, each nanmed with the species name, so that each species that belongs to its own group.

Zero or more species can be specified in the mixture command. If a listed species is not already in the mixture, due to a previous mixture command with the same ID, then that species is added to the mixture. As discussed below, it will be assigned to a default group and assigned default per-species attributes, unless the appropriate keywords are also specified.

Species can be specified which are already part of the mixture, to change their group assignemnt or their per-species proerties, as disussed below.

Zero species can be specified, if other keywords are used which alter group assignments or change global attributes of the mixture, as discussed below.

The *nrho* keyword sets a global attribute of the mixture, namely its density. For 3d simulations the units of the specified *density* are #/volume. For 2d, the units are effectively #/area, since the z-dimension thickness of the simulation box = 1.0.

The *vstream* keyword sets a global attribute of the mixture, namely the streaming velocity. Particles created using the mixture will have the specified *Vx,Vy,Vz* values.

The *temp* keyword sets a global attribute of the mixture, namely the thermal temperature of its particles. When particles are created, this value is used to sample a Gaussian velocity distribution, which is superposed on the streaming velocity, when a particle's velocity is initialized.

The *frac* keyword sets a per-species attribute for individual species in the mixture. Each species has a relative fractional density, such as 0.2, meaning one out of 5 particles is that species. The sum of this value across all species in the mixture must equal 1.0. The *frac* keyword sets this value for the listed species. If this value has never been set for M species out of the total N species in the mixture, then when a simulation is run, the *frac* value for each of the M species is set to (1 - sum)/M, where sum is the sum of the *frac* values for the N-M assigned species.

Each species in the mixture is assigned to exactly one group. The *group* keyword can be used to set or change these assignments.

As described by the collide command, mixture groups are used when performing collisions so that collisions attempts, partners, and parameters can be treated on a per-group basis for accuracy and efficiency. Per-grid computes also use mixture groups to calculate grid-based quantities on subsets of particles within each grid cell.

The specified group ID can be any string you choose. Similar to the mixture ID, it can only contain alphanumeric characters and underscores. Using SELF for the group ID has a special meaning as discussed below.

The operation of the *group* keyword depends on whether zero species or some species are specified explicitly in the mixture command. It also depends on whether the group ID is SELF or a user-defined name. In each case, after the operation is done, any group IDs for the mixture that have no species assigned to them are deleted.

- If zero species are listed and the group ID is SELF, then each species already in the mixture is assigned to a group with its species ID as the group ID. I.e. there will now be one species per group.
- If one or more species are listed and the group ID is SELF, then each listed species is assigned to a group with its species ID as the group ID.

- If zero species are listed and the group ID is not SELF, then all species already in the mixture are assigned to a group with the specified ID.
- If one or more species are listed and the group ID is not SELF, then the listed species are all assigned to a group with the specified ID.

Note that if the *group* keyword is not used, no changes to group assignements are made. If one or more new species are specified, which are not already in the mixture, then each is assigned to a group with the species ID as the group ID.

Restrictions: none

Related commands:

global, create_molecules

Default:

The *nrho*, *vstream*, and *temp* defaults are those defined for the background gas density, as set by the global command. The *frac* default is described above. The *group* keyword has no default; if it is not used, new species not already in the mixture are each assigned to a group with the species ID as the group ID.

next command

Syntax:

```
next variables
```

- variables = one or more variable names

Examples:

```
next x
next a t x myTemp
```

Description:

This command is used with variables defined by the variable command. It assigns the next value to the variable from the list of values defined for that variable by the variable command. Thus when that variable is subsequently substituted for in an input script command, the new value is used.

See the variable command for info on how to define and use different kinds of variables in SPARTA input scripts. If a variable name is a single lower-case character from "a" to "z", it can be used in an input script command as $a or $z. If it is multiple letters, it can be used as ${myTemp}.

If multiple variables are used as arguments to the *next* command, then all must be of the same variable style: *index*, *loop*, *universe*, or *uloop*. An exception is that *universe*- and *uloop*-style variables can be mixed in the same *next* command.

All the variables specified with the next command are incremented by one value from their respective list or values. *String*- or *atom*- or *equal*- or *world*-style variables cannot be used with the the next command, since they only store a single value.

When any of the variables in the next command has no more values, a flag is set that causes the input script to skip the next jump command encountered. This enables a loop containing a next command to exit. As explained in the variable command, the variable that has exhausted its values is also deleted. This allows it to be used and re-defined later in the input script.

When the next command is used with *index*- or *loop*-style variables, the next value is assigned to the variable for all processors. When the next command is used with *universe*- or *uloop*-style variables, the next value is assigned to whichever processor partition executes the command first. All processors in the partition are assigned the same value. Running SPARTA on multiple partitions of processors via the "-partition" command-line switch is described in this section of the manual. *Universe*- and *uloop*-style variables are incremented using the files "tmp.lammps.variable" and "tmp.lammps.variable.lock" which you will see in your directory during such a SPARTA run.

Here is an example of running a series of simulations using the next command with an *index*-style variable. If this input script is named in.polymer, 8 simulations would be run using data files from directories run1 thru run8.

```
variable d index run1 run2 run3 run4 run5 run6 run7 run8
shell cd $d
read_data data.polymer
run 10000
shell cd ..
```

```
clear
next d
jump in.polymer
```

If the variable "d" were of style *universe*, and the same in.polymer input script were run on 3 partitions of processors, then the first 3 simulations would begin, one on each set of processors. Whichever partition finished first, it would assign variable "d" the 4th value and run another simulation, and so forth until all 8 simulations were finished.

Jump and next commands can also be nested to enable multi-level loops. For example, this script will run 15 simulations in a double loop.

```
variable i loop 3
  variable j loop 5
  clear
  ...
  read_data data.polymer.$i$j
  print Running simulation $i.$j
  run 10000
  next j
  jump in.script
next i
jump in.script
```

Here is an example of a double loop which uses the if and jump commands to break out of the inner loop when a condition is met, then continues iterating thru the outer loop.

```
label       loopa
variable    a loop 5
  label     loopb
  variable  b loop 5
  print     "A,B = $a,$b"
  run       10000
  if        $b > 2 then "jump in.script break"
  next      b
  jump      in.script loopb
label       break
variable    b delete

next        a
jump        in.script loopa
```

Restrictions: none

Related commands:

jump, include, shell, variable,

Default: none

partition command

Syntax:

```
partition style N command ...
```

- style = *yes* or *no*
- N = partition number (see asterisk form below)
- command = any SPARTA command

Examples:

```
partition yes 1 processors 4 10 6
partition no 5 print "Active partition"
partition yes *5 fix all nve
partition yes 6* fix all nvt temp 1.0 1.0 0.1
```

Description:

This command invokes the specified command on a subset of the partitions of processors you have defined via the -partition command-line switch. See Section_start 6 for an explanation of the switch.

Normally, every input script command in your script is invoked by every partition. This behavior can be modified by defining world- or universe-style variables that have different values for each partition. This mechanism can be used to cause your script to jump to different input script files on different partitions, if such a variable is used in a jump command.

The "partition" command is another mechanism for having as input script operate differently on different partitions. It is basically a prefix on any SPARTA command. The commmand will only be invoked on the partition(s) specified by the *style* and *N* arguments.

If the *style* is *yes*, the command will be invoked on any partition which matches the *N* argument. If the *style* is *no* the command will be invoked on all the partitions which do not match the Np argument.

Partitions are numbered from 1 to Np, where Np is the number of partitions specified by the -partition command-line switch.

N can be specified in one of two ways. An explicit numeric value can be used, as in the 1st example above. Or a wild-card asterisk can be used to span a range of partition numbers. This takes the form "*" or "*n" or "n*" or "m*n". An asterisk with no numeric values means all partitions from 1 to Np. A leading asterisk means all partitions from 1 to n (inclusive). A trailing asterisk means all partitions from n to Np (inclusive). A middle asterisk means all partitions from m to n (inclusive).

This command can be useful for the "run_style verlet/split" command which imposed requirements on how the processors command lays out a 3d grid of processors in each of 2 partitions.

Restrictions: none

Related commands: none

Default: none

print command

Syntax:

```
print str
```

- str = text string to print, which may contain variables

Examples:

```
print "Done with equilibration"
print Vol=$v
print "The system volume is now $v"
print 'The system volume is now $v'
```

Description:

Print a text string to the screen and logfile. One line of output is generated. If the string has white space in it (spaces, tabs, etc), then you must enclose it in quotes so that it is treated as a single argument. If variables are included in the string, they will be evaluated and their current values printed.

If you want the print command to be executed multiple times (with changing variable values), there are 3 options. First, consider using the fix print command, which will print a string periodically during a simulation. Second, the print command can be used as an argument to the *every* option of the run command. Third, the print command could appear in a section of the input script that is looped over (see the jump and next commands).

See the variable command for a description of *equal* style variables which are typically the most useful ones to use with the print command. Equal-style variables can calculate formulas involving mathematical operations, atom properties, group properties, thermodynamic properties, global values calculated by a compute or fix, or references to other variables.

Restrictions: none

Related commands:

fix print, variable

Default: none

read_surf command

Syntax:

```
read_surf filename surfcoll-ID keyword args ...
```

- filename = name of surface file
- surfcoll-ID = ID of surface collision model
- zero or more keyword/args pairs may be appended
- keyword = *origin* or *trans* or *atrans* or *ftrans* or *scale* or *rotate* or *invert* or *clip*

```
origin args = Ox Oy Oz
  Ox,Oy,Oz = set origin of surface to this point (distance units)
trans args = Dx Dy Dz
  Dx,Dy,Dz = translate origin by this displacement (distance units)
atrans args = Ax Ay Az
  Ax,Ax,Az = translate origin to this absolute point (distance units)
ftrans args = Fx Fy Fz
  Fx,Fy,Fz = translate origin to this fractional point in simulation box
scale args = Sx Sy Sz
  Sx,Sy,Sz = scale surface by these factors around origin
rotate args = theta Rx Ry Rz
  theta = rotate surface by this angle in counter-clockwise direction (degrees)
  Rx,Ry,Rz = rotate around vector starting at origin pointing in this direction
invert args = none
clip args = none
```

Examples:

```
read_surf sphere surf.file
read_surf sphere surf.file trans 10 5 0 scale 3 3 3 invert clip
```

Description:

Read the geometry of one or more surfaces from a file. In SPARTA, a "surface" is a grid of surface elements on the surface of a physical object embedded in the global simulation box. The surface elements are triangles in 3d or line segments in 2d. Particles collide with surface elements as they advect, and optionally perform surface chemistry during the collision. Collision statistics for each surface element are tallied, which can be ouput via the dump surface command.

The surface file contains a list of triangles or line segments; SPARTA does not know or care whether they represent one or more physical objects.

NOTE: doc that file can be gzipped

All of the surface elements in the file are assigned a surfcoll-ID, which is the ID of a surface collision model previously defined by the surf_collide command. This model will be used to perform collisions when particles collide with the surface elements in the surface.

The format of the surface file is discussed below. The optional keywords allow the vertices in the file to be translated, scaled, and rotated in various ways. These allow a single input file, e.g. for a unit sphere, to be used in a variety of simulations.

A surface file has a header and a body. The header appears first. The first line of the header is always skipped; it typically contains a description of the file. Then lines are read one at a time. Lines can have a trailing comment starting with '#' that is ignored. If the line is blank (only whitespace after comment is deleted), it is skipped. If the line contains a header keyword, the corresponding value is read from the line. If it doesn't contain a header keyword, the line begins the body of the file.

The body of the file contains two sections. The first line of a section has only a keyword. The next line is skipped. The remaining lines of the section contain values. The number of lines in a section depends on the section keyword as described below. Zero or more blank lines can be used between sections. Sections can appear in any order.

The formatting of individual lines in the data file (indentation, spacing between words and numbers) is not important except that header and section keywords must be capitalized as shown and can't have extra white space between their words.

These are the recognized header keywords. Header lines can come in any order. The value(s) are read from the beginning of the line. Thus the keyword *points* should be in a line like "1000 points".

- *points* = # of points in surface
- *lines* = # of line segments in surface (only allowed for 2d)
- *triangles* = # of triangles in surface (only allowed for 3d)

These are the recognized section keywords for the body of the file.

- *Points, Lines, Triangles*

The *Points* section consists of N consecutive entries, where N = # of points, each of this form:

```
point-ID x y z     (for 3d)
point-ID x y       (for 2d)
```

The point-ID is ignored; it is only added to assist in examining the file. The point-IDs should run consecutively from 1 to N. *X,y,z* are the coordinates of the point in distance units. Note that for 2d simulations, *z* should be omitted.

The *Lines* section is only allowed for 2d simulations and consists of N entries, where N = # of lines, each of this form:

```
line-ID p1 p2
```

The line-ID is ignored; it is only added to assist in examining the file. *P1* and *p2* are the point-IDs of the 2 end points of the line segment. Each is a value from 1 to Npoints, as described above.

The ordering of *p1* and *p2* is important as it defines the direction of the outward normal for the line segment when a molecule collides with it. Molecules only collide with the "outer" edge of a line segment. This is defined by a right-hand rule. The outward normal N = (0,0,1) x (p2-p1). In other words, a unit z-direction vector is crossed into the vector from *p1* to *p2* to determine the normal.

The *Triangles* section is only allowed for 3d simulations and consists of N entries, where N = # of triangles, each of this form:

```
tri-ID p1 p2 p3
```

The tri-ID is ignored; it is only added to assist in examining the file. *P1* and *p2* and *p3* are the point-IDs of the 3 corner points of the triangle. Each is a value from 1 to Npoints, as described above.

The ordering of *p1* and *p2* and *p3* is important as it defines the direction of the outward normal for the triangle when a molecule collides with it. Molecules only collide with the "outer" face of a triangle. This is defined by a right-hand rule. The outward normal N = (p2-p1) x (p3-p1). In other words, the edge from *p1* to *p2* is crossed into the edge from *p1* to *p3* to determine the normal.

The following optional keywords can be specified. Their geometric transformations are performed in the order they are listed, which gives flexibility in how surfaces can be manipulated. Note that the order may be important; e.g. performing a *scale* followed by a *rotate* may not be the same as a *rotate* followed by a *scale*.

Most of the keywords perform a geometric transformation on all the vertices in the surface file with respect to an origin point. By default the origin is (0.0,0.0,0.0), regardless of the position of individual vertices in the file.

The *origin* keyword resets the origin to the specified *Ox,Oy,Oz*. This operation has no effect on the vertices.

The *trans* keyword shifts or displaces the origin by the vector (Dx,Dy,Dz). It also displaces each vertex by (Dx,Dy,Dz).

The *atrans* keyword resets the origin to an absolute point (Ax,Ay,Az) which implies a displacement (Dx,Dy,Dz) from the current origin. It also displaces each vertex by (Dx,Dy,Dz).

The *ftrans* keyword resets the origin to a fractional point (Fx,Fy,Fz). Fractional means that Fx = 0.0 is the lower edge/face in the x-dimension and Fx = 1.0 is the upper edge/face in the x-dimension, and similarly for Fy and Fz. This change of origin implies a displacement (Dx,Dy,Dz) from the current origin. This operation also displaces each vertex by (Dx,Dy,Dz).

The *scale* keyword does not change the origin. It computes the displacement vector of each vertex from the origin (delx,dely,delz) and scales that vector by (Sx,Sy,Sz), so that the new vertex coordinate is (Ox + Sx*delx,Oy + Sy*dely,Oz + Sz*delz).

The *rotate* keyword does not change the origin. It rotates the coordinates of all vertices by an angle *theta* in a counter-clockwise direction, around the vector starting at the origin and pointing in the direction *Rx,Ry,Rz*. Any rotation can be represented by an appropriate choice of origin, *theta* and (Rx,Ry,Rz).

The *invert* keyword does not change the origin or any vertex coordinates. It flips the direction of the outward surface normal of each surface element by changing the ordering of its vertices. Since particles only collide with the outer surface of a surface element, this is a mechanism for using a surface files containing a single sphere (for example) as either a sphere to embed in a flow field, or a spherical outer boundary containing the flow.

The *clip* keyword does not change the origin. It truncates or "clips" a surface that extends outside the simulation box in the following manner. In 3d, each of the 6 clip planes represented by faces of the global simulation box are considered in turn. Any triangle that straddles the face (with points on both sides of the clip plane), is truncated at the plane. New points along the edges that cross the plane are created. A triangle may also become a trapezoid, in which case it turned into 2 triangles. Then all the points on the side of the clip plane that is outside the box, are projected onto the clip plane. Finally, all triangles that lie in the clip plane are removed, as are any points that are unused after the triangle removal. After this operation is repeated for all 6 faces, the remaining surface is entirely inside the simulation box, though some of its triangles may include points on the faces of the simulation box. A similar operation is performed in 2d with the 4 clip edges represented by the edges of the global simulation box.

If you use the *clip* keyword, you should check the resulting statistics of the clipped surface printed out by this command, including the minimum size of line and triangle edge lengths. It is possible that very short lines or very small triangles will result near the box surface due to the clipping opertion, depending on the coordinates of the initial unclipped points.

Restrictions:

This command can only be used after the simulation box is defined by the create_box command, and after a grid has been created by the create_grid command.

Every vertex in the final surface (after translation, rotation, scaling, etc) must be inside or on the surface of the global simulation box. Note that using the *clip* operation guarantees that this will be the case.

The surface elements in a single surface file must represent a "watertight" surface. For a 2d simulation this means that every point is part of exactly 2 or 4 line segments. For a 3d simulation it means that every triangle edge is part of exactly 2 or 4 triangles. Thus if you want to model an infinitely thin object (e.g. a plane), it must be gridded on both sides. If you use the same vertices on both sides, this is why 4 line segments or triangles can border the same point or triangle edge. An exception to this rule is made in 2d if a point is on the global face of the simulation box. In that case it may only be part of one line segment. Likewise in 3d, a triangle edge which lies entirely in the global face of the simulation box, can be part of only one triangle.

Note that this definition of watertight does not require that the surface elements in a file represent a single physical object; multiple objects can be represented, provided each is watertight.

When running a simulation with multiple surfaces, read from one or more surface files, you should insure they do not overlap/intersect each other. SPARTA does not check for this, but it will typically lead to unphysical particle dynamics.

Related commands: none

Default:

The default origin for the vertices in the surface file is (0,0,0).

restart command

NOTE: this command is not yet implemented

Syntax:

```
restart 0
restart N root
restart N file1 file2
```

- N = write a restart file every this many timesteps
- root = filename to which timestep # is appended
- file1,file2 = two full filenames, toggle between them when writing file

Examples:

```
restart 0
restart 1000 poly.restart
restart 1000 restart.*.equil
restart 10000 poly.%.1 poly.%.2
```

Description:

Write out a binary restart file every so many timesteps as a run proceeds. A value of 0 means do not write out restart files. Using one filename as an argument will create a series of filenames which include the timestep in the filename. Using two filenames will produce only 2 restart files. SPARTA will toggle between the 2 names as it writes successive restart files.

Similar to dump files, the restart filename(s) can contain two wild-card characters. If a "*" appears in the filename, it is replaced with the current timestep value. This is only recognized when a single filename is used (not when toggling back and forth). Thus, the 3rd example above creates restart files as follows: restart.1000.equil, restart.2000.equil, etc. If a single filename is used with no "*", then the timestep value is appended. E.g. the 2nd example above creates restart files as follows: poly.restart.1000, poly.restart.2000, etc.

If a "%" character appears in the restart filename(s), then one file is written for each processor and the "%" character is replaced with the processor ID from 0 to P-1. An additional file with the "%" replaced by "base" is also written, which contains global information. For example, the files written on step 1000 for filename restart.% would be restart.base.1000, restart.0.1000, restart.1.1000, ..., restart.P-1.1000. This creates smaller files and can be a fast mode of output and subsequent input on parallel machines that support parallel I/O.

Restart files are written on timesteps that are a multiple of N but not on the first timestep of a run or minimization. A restart file is not written on the last timestep of a run unless it is a multiple of N. A restart file is written on the last timestep of a minimization if N > 0 and the minimization converges.

See the read_restart command for information about what is stored in a restart file.

Restart files can be read by a read_restart command to restart a simulation from a particular state. Because the file is binary (to enable exact restarts), it may not be readable on another machine. In this case, the restart2data program in the tools directory can be used to convert a restart file to an ASCII data file. Both the read_restart command and restart2data tool can read in a restart file that was written with the "%" character so that multiple files were created.

Restrictions: none

Related commands:

write_restart, read_restart

Default:

```
restart 0
```

run command

Syntax:

```
run N
```

- N = # of timesteps

Examples:

```
run 10000
run 1000000
```

Description:

Run or continue a simulation for a specified number of timesteps.

Restrictions: none

Related commands: none

Default: none

seed command

Syntax:

```
seed Nvalue
```

- Nvalue = seed for a random number generator (positive integer)

Examples:

```
seed 5838959
```

Description:

This command sets the random number seed for a master random number generator which is used by SPARTA to initialize auxiliary random number generators which in turn are used for all operations in the code requiring random numbers. Thus this command is required to perform any simulation with SPARTA.

Restrictions: none

Related commands: none

Default: none

shell command

Syntax:

```
shell cmd args
```

- cmd = *cd* or *mkdir* or *mv* or *rm* or *rmdir* or arbitrary command

```
cd arg = dir
    dir = directory to change to
  mkdir args = dir1 dir2 ...
    dir1,dir2 = one or more directories to create
  mv args = old new
    old = old filename
    new = new filename
  rm args = file1 file2 ...
    file1,file2 = one or more filenames to delete
  rmdir args = dir1 dir2 ...
    dir1,dir2 = one or more directories to delete
  anything else is passed as a command to the shell for direct execution
```

Examples:

```
shell cd sub1
shell cd ..
shell mkdir tmp1 tmp2 tmp3
shell rmdir tmp1
shell mv log.lammps hold/log.1
shell rm TMP/file1 TMP/file2
shell my_setup file1 10 file2
shell my_post_process 100 dump.out
```

Description:

Execute a shell command. A few simple file-based shell commands are supported directly, in Unix-style syntax. Any command not listed above is passed as-is to the C-library system() call, which invokes the command in a shell.

This is means to invoke other commands from your input script. For example, you can move files around in preparation for the next section of the input script. Or you can run a program that pre-processes data for input into SPARTA. Or you can run a program that post-processes SPARTA output data.

With the exception of *cd*, all commands, including ones invoked via a system() call, are executed by only a single processor, so that files/directories are not being manipulated by multiple processors.

The *cd* cmd executes the Unix "cd" command to change the working directory. All subsequent SPARTA commands that read/write files will use the new directory. All processors execute this command.

The *mkdir* cmd executes the Unix "mkdir" command to create one or more directories.

The *mv* cmd executes the Unix "mv" command to rename a file and/or move it to a new directory.

The *rm* cmd executes the Unix "rm" command to remove one or more files.

The *rmdir* cmd executes the Unix "rmdir" command to remove one or more directories. A directory must be empty to be successfully removed.

Any other cmd is passed as-is to the shell along with its arguments as one string, invoked by the C-library system() call. For example, these lines in your input script:

```
variable n equal 10
variable foo string file2
shell my_setup file1 $n ${foo}
```

would be the same as invoking

```
% my_setup file1 10 file2
```

from a command-line prompt. The executable program "my_setup" is run with 3 arguments: file1 10 file2.

Restrictions:

SPARTA does not detect errors or print warnings when any of these commands execute. E.g. if the specified directory does not exist, executing the *cd* command will silently do nothing.

Related commands: none

Default: none

species command

Syntax:

```
species file ID1 ID2 ...
```

- file = filename that lists species
- ID1, ID2, ... = one or more species names listed in file
- multi-species abbreviations can also be used (see below)

Examples:

```
species species.data air
species species.data air Cl
species species.data O2 N2 NO
species myfile H+ Cl- HCl
```

Description:

Define one or more particle species to use in the simulation. This command can be used as many times as desired to add species to a list of species that the simulation recognizes and has parameters for.

The *file* is a file that contains definitions for some number of species, not all of which need to used by this simulation. Only those requested with specific IDs will be extracted from the file and they must be present in the file. The format of the species file is discussed below.

Each *ID* is a character string used to identify the species, such as N or O2 or NO or D or Fe-, which can be any combination of alphanumeric characters, or "=", "-", or underscore.

Instead of listing single-species IDs, one of several pre-defined multi-species names can be used, each of which expands into a list of several individual species IDs. The list of recognized abbreviations is as follows:

- air = N, O, NO

These abbreviations can be used in combination with single-species IDs as in the 2nd example above.

The format of the species file is as follows. Comments or blank lines are allowed in the file. Comment lines start with a "#" character. All other lines must have format like this with values separated by whitespace:

species-ID prop1 prop2 ... prop9

The species-ID is a string that will be matched against the requested species-ID, as described above. The properties are as follows:

- prop1 = molwt = molecular weight
- prop2 = mass = mass (mass units)
- prop3 = rotdof = rotational degrees of freedom
- prop4 = rotrel = rotational ???
- prop5 = vibdof = vibrational degrees of freedom
- prop6 = vibrel = vibrational ??
- prop7 = vibtemp = vibrational temperature (temperature units)

- prop8 = specwt = ???
- prop9 = charge = ???

NOTE: give correct definitions and their units

Restrictions: none

Related commands: none

Default: none

stats command

Syntax:

```
stats N
```

- N = output statistics every N timesteps

Examples:

```
stats 100
```

Description:

Compute and print statistical info (e.g. particle count, temperature) on timesteps that are a multiple of N and at the beginning and end of a simulation. A value of 0 will only print statistics at the beginning and end.

The content and format of what is printed is controlled by the stats_style and stats_modify commands.

The timesteps on which statistical output is written can also be controlled by a variable. See the stats_modify every command.

Restrictions: none

Related commands:

stats_style, stats_modify

Default:

```
stats 0
```

stats_modify command

Syntax:

```
stats_modify keyword value ...
```

- one or more keyword/value pairs may be listed
- keyword = *flush* or *format* or *every*

```
  flush value = yes or no
    format values = int string or float string or M string
      M = integer from 1 to N, where N = # of quantities being printed
      string = C-style format string
    every value = v_name
      v_name = an equal-style variable name
```

Examples:

```
stats_modify flush yes
stats_modify temp myTemp format 3 %15.8g
stats_modify line multi format float %g
```

Description:

Set options for how statistical information is computed and printed by SPARTA.

The *flush* keyword invokes a flush operation after statistical info is written to the log file. This insures the output in that file is current (no buffering by the OS), even if SPARTA halts before the simulation completes.

The *format* keyword sets the numeric format of individual printed quantities. The *int* and *float* keywords set the format for all integer or floating-point quantities printed. The setting with a numeric value M (e.g. format 5 %10.4g) sets the format of the Mth value printed in each output line, e.g. the 5th column of output in this case. If the format for a specific column has been set, it will take precedent over the *int* or *float* setting.

IMPORTANT NOTE: The statistical output values *step* and *npart* are stored internally as 8-byte signed integers, rather than the usual 4-byte signed integers. When specifying the "format int" keyword you can use a "%d"-style format identifier in the format string and SPARTA will convert this to the corresponding "%lu" form when it is applied to those keywords. However, when specifying the "format M string" keyword for *step* and *natoms*, you should specify a string appropriate for an 8-byte signed integer, e.g. one with "%ld".

The *every* keyword allows a variable to be specified which will determine which timesteps statistical output is generated. It must be an equal-style variable, and is specified as v_name, where name is the variable name. The variable is evaluated at the beginning of a run to determine the next timestep at which a dump snapshot will be written out. On that timestep, the variable will be evaluated again to determine the next timestep, etc. Thus the variable should return timestep values. See the stagger() and logfreq() math functions for equal-style variables, as examples of useful functions to use in this context. Other similar math functions could easily be added as options for equal-style variables. In addition, statistical output will always occur on the first and last timestep of each run.

For example, the following commands will output statistical info at timesteps 0,10,20,30,100,200,300,1000,2000,etc:

```
variable        s equal logfreq(10,3,10)
```

```
stats_modify    1 every v_s
```

Note that the *every* keyword overrides the output frequency setting made by the stats command, by setting it to 0. If the stats command is later used to set the output frequency to a non-zero value, then the var,iable setting of the stats_modify every command will be overridden.

Restrictions: none

Related commands:

stats, stats_style

Default:

The option defaults are flush = no, format int = "%8d", format float = "%12.8g", and every = non-variable setting provided by the stats command.

stats_style command

Syntax:

```
stats_style arg1 arg2 ...
```

- arg1,arg2,... = list of statistical attributes

```
possible attributes = step, elapsed, dt, cpu, tpcpu, spcpu,
                      nmol, ntouch, ncomm, nbound, nexit,
                      nscoll, nscheck, ncoll, nattempt,
                      nmolave, ntouchave, ncommave, nboundave, nexitave,
                      nscollave, nscheckave, ncollave, nattemptave,
                      vol, lx, ly, lz,
                      xlo, xhi, ylo, yhi, zlo, zhi,
                      c_ID, c_ID[I], c_ID[I][J],
                      f_ID, f_ID[I], f_ID[I][J],
                      v_name

    step = timestep
    elapsed = timesteps since start of this run
    dt = timestep size
    cpu = elapsed CPU time in seconds
    tpcpu = time per CPU second
    spcpu = timesteps per CPU second
    nmol,nmolave = # of molecules (this step, per-step)
    ntouch,ntouchave = # of cell touches by molecules (this step, per-step)
    ncomm,ncommave = # of molecules communicated (this step, per-step)
    nbound,nboundave = # of boundary collisions (this step, per-step)
    nexit,nexitave = # of boundary exits (this step, per-step)
    nscoll,nscollave = # of surface collisions (this step, per-step)
    nscheck,nscheckave = # of surface checks (this step, per-step)
    ncoll,ncollave = # of molecule/molecule collisions (this step, per-step)
    nattempt,nattemptave = # of attempted collisions (this step, per-step)
    vol = volume of simulation box
    lx,ly,lz = simulation box lengths
    xlo,xhi,ylo,yhi,zlo,zhi = box boundaries,
    c_ID = global scalar value calculated by a compute with ID
    c_ID[I] = Ith component of global vector calculated by a compute with ID
    c_ID[I][J] = I,J component of global array calculated by a compute with ID
    f_ID = global scalar value calculated by a fix with ID
    f_ID[I] = Ith component of global vector calculated by a fix with ID
    f_ID[I][J] = I,J component of global array calculated by a fix with ID
    v_name = scalar value calculated by an equal-style variable with name
```

Examples:

```
stats_style step cpu nmol temp
stats_style step temp cpu spcpu nmol xlo xhi c_myTemp
```

Description:

Determine what statistical data is printed to the screen and log file.

The values printed by the various keywords are instantaneous values, calculated on the current timestep. The exception is the keywords suffixed by "ave", which print a running total divided by the number of timesteps.

Options invoked by the stats_modify command can be used to set the numeric precision of each printed value, as well as other attributes of the statistics.

The *step* and *elapsed* keywords refer to timestep count. *Step* is the current timestep, or iteration count when a minimization is being performed. *Elapsed* is the number of timesteps elapsed since the beginning of this run.

The *cpu* keyword is elapsed CPU seconds since the beginning of this run. The *tpcpu* and *spcpu* keywords are measures of how fast your simulation is currently running. The *tpcpu* keyword is simulation time per CPU second, where simulation time is in time units. The *spcpu* keyword is the number of timesteps per CPU second. Both quantities are on-the-fly metrics, measured relative to the last time they were invoked. Thus if you are printing out statistical output every 100 timesteps, the two keywords will continually output the time and timestep rate for the last 100 steps.

The *nmol*, *ntouch*, *ncomm*, *nbound*, *nexit*, *nscoll*, *nscheck*, *ncoll*, and *nattempt* keywords all generate counts for the current timestep.

The *nmolave*, *ntouchave*, *ncommave*, *nboundave*, *nexitave*, *nscollave*, *nscheckave*, *ncollave*, and *nattemptave* keywords all generate values that are the cummulative total of the corresponding count divided by *elapsed* = the number of timesteps since the start of the run.

The *nmol* keyword is the number of molecules.

The *ntouch* keyword is the number of cells touched by the molecules during the move portion of the timestep. E.g. if a molecule moves from cell A to adjacent cell B, it touches 2 cells.

The *ncomm* keyword is the number of molecules communicated to other processors.

The *nbound* keyword is the number of molecules that collided with a global boundary. Crossing a periodic boundary or exiting an outflow boundary is not counted.

The *nexit* keyword is the number of molecules that exited the simulation box through an outflow boundary.

The *nscoll* keyword is the number of molecule/surface collisions that occured, where a molecule collided with a geometric surface.

The *nscheck* keyword is the number of molecule/surface collisions that were checked for. If a cell is overlapped by N surface elements, all N must be checked for collisions each time a molecule in that cell moves.

The *ncoll* keyword is the number of molecule/molecule collisions that occured.

The *nattempt* keyword is the number of molecule/molecule collisions that were attempted.

The *vol* keyword is the volume (or area in 2d) of the simulation box.

The *lx*, *ly*, *lz* keywords are the dimensions of the simulation box.

The *xlo*, *xhi*, *ylo*, *yhi*, *zlo*, *zhi* keywords are the boundaries of the simulation box.

The *c_ID* and *c_ID[I]* and *c_ID[I][J]* keywords allow global values calculated by a compute to be output. As discussed on the compute doc page, computes can calculate global, per-molecule, or per-grid values. Only global values can be referenced by this command. However, per-molecule or per-grid compute values can be referenced in a variable and the variable referenced, as discussed below.

The ID in the keyword should be replaced by the actual ID of a compute that has been defined elsewhere in the input script. See the compute command for details. If the compute calculates a global scalar, vector, or array, then the keyword formats with 0, 1, or 2 brackets will reference a scalar value from the compute.

The *f_ID* and *f_ID[I]* and *f_ID[I][J]* keywords allow global values calculated by a fix to be output. As discussed on the fix doc page, fixes can calculate global, per-molecule, or per-grid values. Only global values can be referenced by this command. However, per-molecule or per-grid fix values can be referenced in a variable and the variable referenced, as discussed below.

The ID in the keyword should be replaced by the actual ID of a fix that has been defined elsewhere in the input script. See the fix command for details. If the fix calculates a global scalar, vector, or array, then the keyword formats with 0, 1, or 2 brackets will reference a scalar value from the fix.

The *v_name* keyword allow the current value of a variable to be output. The name in the keyword should be replaced by the variable name that has been defined elsewhere in the input script. Only equal-style variables can be referenced. See the variable command for details. Variables of style *equal* can reference per-molecule or per-grid properties or stats keywords, or they can invoke other computes, fixes, or variables when evaluated, so this is a very general means of creating statistical output.

See Section_modify for information on how to add new compute and fix styles to SPARTA to calculate quantities that can then be referenced with these keywords to generate statistical output.

Restrictions: none

Related commands:

stats, stats_modify

Default:

```
stats_style step cpu nmol
```

surf_collide command

Syntax:

```
surf_collide ID style args keyword values ...
```

- ID = user-assigned name for the surface collision model style = *specular* or *diffuse*
- args = arguments for specific style

```
specular args = none
  diffuse args = Tsurf acc
    Tsurf = temperature of surface (temperature units)
            Tsurf can be a variable (see below)
    acc = accommodation coefficient
```

- zero or more keyword/arg pairs may be appended
- keyword = *translate* or *rotate*
- values = values for specific keyword

```
translate args = Vx Vy Vz
    Vx,Vy,Vz = translational velocity of surface (velocity units)
  rotate args = Pz Py Pz Wx Wy Wz
    Px,Py,Pz = point to rotate surface around (distance units)
    Wx,Wy,Wz = angular velocity of surface around point (radians/time)
```

Examples:

```
surf_collide 1 specular
surf_collide 1 diffuse 273.15 0.9
surf_collide heatwall diffuse v_ramp 0.8
surf_collide heatwall diffuse v_ramp 0.8 translate 5.0 0.0 0.0
```

Description:

Define a style of particle-surface collisions to be performed by SPARTA. One or more styles can be defined and assigned to different surfaces or simulation box boundaries via the surface or bound_modify commands. See the read_surf command for a description of how SPARTA defines surfaces as collections of geometric elements, triangles in 3d and line segments in 2d.

The *specular* style computes a simple specular reflection model. It requires no arguments. Specular reflection means that a particle reflects off the surrace element with its incident velocity vector reversed with respect to the outward normal of the surface element. The particle's speed is unchanged.

The *diffuse* style computes a simple diffusive reflection model.

The model has 2 parameters set by the *Tsurf* and *acc* arguments. *Tsurf* is the temperature of the surface. *Acc* is an accommodation coefficient.

Diffuse reflection is where the particle is emitted from the surface with no dependence on its incident velocity. A new velocity is assigned to the particle, sampled from a Gaussian distribution consistent with the surface temperature. The new velocity will have a component in the direction of the outward surface normal given by:

and 2 components in the plane tangent to the surface normal given by:

The *Twall* value can be specified as an equal-style variable. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the current surface temperature.

Equal-style variables can specify formulas with various mathematical functions, and include thermo_style command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent temperature.

The keyword *translate* can only be applied to the *diffuse* style. It models the surface as if it were translating with a constant velocity, specified by the vector (Vx,Vy,Vz). This velocity is added to the final post-collisional velocity of each particle that collides with the surface.

Note that you should define the velocity vector to be tangential to the surface. Normally this keyword should only be used for flat surfaces, such as the simulation box wall. SPARTA does not check for these conditions.

The keyword *rotate* can only be applied to the *diffuse* style. It models the surface as if it were rotating with a constant angular velocity, specified by the vector W = (Wx,Wy,Wz), around the specified point P = (Px,Py,Pz). When a particle collides with the surface at a point X = (x,y,z), the collision point has a velocity given by V = (Vx,Vy,Vz) = W cross (X-P). This velocity is added to the final post-collisional velocity of the particle.

This keyword can be used to model a rotating wall, e.g. the end cap of a cylinder. Or to model a rotating object consisting of surface elements, e.g. a sphere. In either case, the surface itself does not move or rotate, e.g. the surface elements do not change position. They are simply modeled as having a tangential velocity, as if the entire object were rotating.

Restrictions:

The *translate* and *rotate* keywords cannot be used together.

Related commands:

read_surf, bound_modify

Default: none

timestep command

Syntax:

```
timestep dt
```

- dt = timestep size (time units)

Examples:

```
timestep 2.0
timestep 0.003
```

Description:

Set the timestep size for subsequent simulations.

Restrictions: none

Related commands:

run

Default:

```
timestep 1.0
```

uncompute command

Syntax:

```
uncompute compute-ID
```

- compute-ID = ID of a previously defined compute

Examples:

```
uncompute 2
uncompute lower-boundary
```

Description:

Delete a compute that was previously defined with a compute command.

Restrictions: none

Related commands:

compute

Default: none

undump command

Syntax:

```
undump dump-ID
```

- dump-ID = ID of previously defined dump

Examples:

```
undump mine
undump 2
```

Description:

Turn off a previously defined dump so that it is no longer active. This closes the file associated with the dump.

Restrictions: none

Related commands:

dump

Default: none

unfix command

Syntax:

```
unfix fix-ID
```

- fix-ID = ID of a previously defined fix

Examples:

```
unfix 2
unfix lower-boundary
```

Description:

Delete a fix that was previously defined with a fix command.

Restrictions: none

Related commands:

fix

Default: none

units command

Syntax:

```
units style
```

- style = *cgs* or *si*

Examples:

```
units cgs
```

Description:

This command sets the style of units used for a simulation. It determines the units of all quantities specified in the input script and various input files read by SPARTA, as well as the units of all quantities output to the screen, log file, dump files, and other output files. Typically, this command is used at the very beginning of an input script.

For style *cgs*, these are the units:

- mass = grams
- distance = centimeters
- area = cm^2
- volume = cm^3
- time = seconds
- energy = ergs
- velocity = centimeters/second
- temperature = degrees K

For style *si*, these are the units:

- mass = kilograms
- distance = meters
- area = m^2
- volume = m^3
- time = seconds
- energy = Joules
- velocity = meters/second
- temperature = degrees K

The units command also sets the timestep size for each style: distance to default values for each style:

- For style *cgs* this is dt = 1.0 sec.
- For style *si* this is dt = 1.0 sec.

Restrictions:

This command cannot be used after the simulation box is defined by the create_box command.

Related commands: none

Default:

```
units si
```

variable command

Syntax:

```
variable name style args ...
```

- name = name of variable to define
- style = *delete* or *index* or *loop* or *world* or *universe* or *uloop* or *string* or *equal* or *particle*

```
  delete = no args
    index args = one or more strings
    loop args = N
      N = integer size of loop, loop from 1 to N inclusive
    loop args = N pad
      N = integer size of loop, loop from 1 to N inclusive
      pad = all values will be same length, e.g. 001, 002, ..., 100
    loop args = N1 N2
      N1,N2 = loop from N1 to N2 inclusive
    loop args = N1 N2 pad
      N1,N2 = loop from N1 to N2 inclusive
      pad = all values will be same length, e.g. 050, 051, ..., 100
    world args = one string for each partition of processors
    universe args = one or more strings
    uloop args = N
      N = integer size of loop
    uloop args = N pad
      N = integer size of loop
      pad = all values will be same length, e.g. 001, 002, ..., 100
    string arg = one string
    equal or particle args = one formula containing numbers, stats keywords, math operations, pa
      numbers = 0.0, 100, -5.4, 2.8e-4, etc
      constants = PI
      stats keywords = step, npart, vol, etc from stats_style
      math operators = (), -x, x+y, x-y, x*y, x/y, x^y,
                       x==y, x!=y, xy, x>=y, x&&y, x||y, !x
      math functions = sqrt(x), exp(x), ln(x), log(x),
                       sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), atan2(y,x),
                       random(x,y), normal(x,y), ceil(x), floor(x), round(x)
                       ramp(x,y), stagger(x,y), logfreq(x,y,z), vdisplace(x,y), swiggle(x,y,z),
      special functions = sum(x), min(x), max(x), ave(x), trap(x)
      particle vector = mass, type, x, y, z, vx, vy, vz
      compute references = c_ID, c_ID[i], c_ID[i][j]
      fix references = f_ID, f_ID[i], f_ID[i][j]
      variable references = v_name
```

Examples:

```
variable x index run1 run2 run3 run4 run5 run6 run7 run8
variable LoopVar loop $n
variable beta equal temp/3.0
variable b1 equal x[234]+0.5*vol
variable b1 equal "x[234] + 0.5*vol"
variable b equal xcm(mol1,x)/2.0
variable b equal c_myTemp
variable b particle x*y/vol
variable foo string myfile
variable temp world 300.0 310.0 320.0 ${Tfinal}
variable x universe 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
variable x uloop 15 pad
```

```
variable x delete
```

Description:

This command assigns one or more strings to a variable name for evaluation later in the input script or during a simulation.

Variables can be used in several ways in SPARTA. A variable can be referenced elsewhere in an input script to become part of a new input command. For variable styles that store multiple strings, the next command can be used to increment which string is assigned to the variable. Variables of style *equal* store a formula which when evaluated produces a single numeric value which can be output either directly (see the print, fix print, and run every commands) or as part of statistical output (see the stats_style command). Variables of style *particle* store a formula which when evaluated produces one numeric value per particle which can be output to a dump file (see the dump custom command).

In the discussion that follows, the "name" of the variable is the arbitrary string that is the 1st argument in the variable command. This name can only contain alphanumeric characters and underscores. The "string" is one or more of the subsequent arguments. The "string" can be simple text as in the 1st example above, it can contain other variables as in the 2nd example, or it can be a formula as in the 3rd example. The "value" is the numeric quantity resulting from evaluation of the string. Note that the same string can generate different values when it is evaluated at different times during a simulation.

IMPORTANT NOTE: When the input script line that defines a variable of style *equal* or *particle* that contain a formula is encountered, the formula is NOT immediately evaluated and the result stored. See the discussion below about "Immediate Evaluation of Variables" if you want to do this.

IMPORTANT NOTE: When a variable command is encountered in the input script and the variable name has already been specified, the command is ignored. This means variables can NOT be re-defined in an input script (with 2 exceptions, read further). This is to allow an input script to be processed multiple times without resetting the variables; see the jump or include commands. It also means that using the command-line switch -var will override a corresponding index variable setting in the input script.

There are two exceptions to this rule. First, variables of style *string* and *equal* and *particle* ARE redefined each time the command is encountered. This allows these style of variables to be redefined multiple times in an input script. In a loop, this means the formula associated with an *equal* or *particle* style variable can change if it contains a substitution for another variable, e.g. $x.

Second, as described below, if a variable is iterated on to the end of its list of strings via the next command, it is removed from the list of active variables, and is thus available to be re-defined in a subsequent variable command. The *delete* style does the same thing.

This section of the manual explains how occurrences of a variable name in an input script line are replaced by the variable's string. The variable name can be referenced as $x if the name "x" is a single character, or as ${LoopVar} if the name "LoopVar" is one or more characters.

As described below, for variable styles *index*, *loop*, *universe*, and *uloop*, which string is assigned to a variable can be incremented via the next command. When there are no more strings to assign, the variable is exhausted and a flag is set that causes the next jump command encountered in the input script to be skipped. This enables the construction of simple loops in the input script that are iterated over and then exited from.

As explained above, an exhausted variable can be re-used in an input script. The *delete* style also removes the variable, the same as if it were exhausted, allowing it to be redefined later in the input script or when the input

script is looped over. This can be useful when breaking out of a loop via the if and jump commands before the variable would become exhausted. For example,

```
label       loop
variable    a loop 5
print       "A = $a"
if          "$a > 2" then "jump in.script break"
next        a
jump        in.script loop
label       break
variable    a delete
```

For the *index* style, one or more strings are specified. Initially, the 1st string is assigned to the variable. Each time a next command is used with the variable name, the next string is assigned. All processors assign the same string to the variable.

Index style variables with a single string value can also be set by using the command-line switch -var; see this section for details.

The *loop* style is identical to the *index* style except that the strings are the integers from 1 to N inclusive, if only one argument N is specified. This allows generation of a long list of runs (e.g. 1000) without having to list N strings in the input script. Initially, the string "1" is assigned to the variable. Each time a next command is used with the variable name, the next string ("2", "3", etc) is assigned. All processors assign the same string to the variable. The *loop* style can also be specified with two arguments N1 and N2. In this case the loop runs from N1 to N2 inclusive, and the string N1 is initially assigned to the variable.

For the *world* style, one or more strings are specified. There must be one string for each processor partition or "world". See this section of the manual for information on running SPARTA with multiple partitions via the "-partition" command-line switch. This variable command assigns one string to each world. All processors in the world are assigned the same string. The next command cannot be used with *equal* style variables, since there is only one value per world. This style of variable is useful when you wish to run different simulations on different partitions, or when performing a parallel tempering simulation (see the temper command), to assign different temperatures to different partitions.

For the *universe* style, one or more strings are specified. There must be at least as many strings as there are processor partitions or "worlds". See this page for information on running SPARTA with multiple partitions via the "-partition" command-line switch. This variable command initially assigns one string to each world. When a next command is encountered using this variable, the first processor partition to encounter it, is assigned the next available string. This continues until all the variable strings are consumed. Thus, this command can be used to run 50 simulations on 8 processor partitions. The simulations will be run one after the other on whatever partition becomes available, until they are all finished. *Universe* style variables are incremented using the files "tmp.lammps.variable" and "tmp.lammps.variable.lock" which you will see in your directory during such a SPARTA run.

The *uloop* style is identical to the *universe* style except that the strings are the integers from 1 to N. This allows generation of long list of runs (e.g. 1000) without having to list N strings in the input script.

For the *equal* and *particle* styles, a single string is specified which represents a formula that will be evaluated afresh each time the variable is used. If you want spaces in the string, enclose it in double quotes so the parser will treat it as a single argument. For *equal* style variables the formula computes a scalar quantity, which becomes the value of the variable whenever it is evaluated. For *particle* style variables the formula computes one quantity for each particle whenever it is evaluated.

Note that *equal* and *particle* variables can produce different values at different stages of the input script or at different times during a run. For example, if an *equal* variable is used in a fix print command, different values could be printed each timestep it was invoked. If you want a variable to be evaluated immediately, so that the result is stored by the variable instead of the string, see the section below on "Immediate Evaluation of Variables".

The next command cannot be used with *equal* or *particle* style variables, since there is only one string.

The formula for an *equal* or *particle* variable can contain a variety of quantities. The syntax for each kind of quantity is simple, but multiple quantities can be nested and combined in various ways to build up formulas of arbitrary complexity. For example, this is a valid (though strange) variable formula:

```
variable x equal "npart + c_MyTemp / vol^(1/3)"
```

Specifically, a formula can contain numbers, stats keywords, math operators, math functions, particle vectors, compute references, fix references, and references to other variables.

Number	0.2, 100, 1.0e20, -15.4, etc
Constant	PI
Stats keywords	step, npart, vol, etc
Math operators	(), -x, x+y, x-y, x*y, x/y, x^y, x==y, x!=y, xy, x>=y, x&&y, x||y, !x
Math functions	sqrt(x), exp(x), ln(x), log(x), sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), atan2(y,x), random(x,y,z), normal(x,y,z), ceil(x), floor(x), round(x), ramp(x,y), stagger(x,y), logfreq(x,y,z), vdisplace(x,y), swiggle(x,y,z), cwiggle(x,y,z)
Special functions	sum(x), min(x), max(x), ave(x), trap(x)
Particle vectors	mass, type, x, y, z, vx, vy, vz
Compute references	c_ID, c_ID[i], c_ID[i][j]
Fix references	f_ID, f_ID[i], f_ID[i][j]
Other variables	v_name

Most of the formula elements produce a scalar value. A few produce a per-molecule vector of values. These are the particle vectors, compute references that represent a per-molecule vector, fix references that represent a per-molecules vector, and variables that are particle-style variables. Math functions that operate on scalar values produce a scalar value; math function that operate on per-molecule vectors do so element-by-element and produce a per-molecule vector.

A formula for equal-style variables cannot use any formula element that produces a per-molecule vector. A formula for an particle-style variable can use formula elements that produce either a scalar value or a per-molecule vector.

The stats keywords allowed in a formula are those defined by the stats_style custom command. If a variable is evaluated directly in an input script (not during a run), then the values accessed by the thermo keyword must be current. See the discussion below about "Variable Accuracy".

Math Operators

Math operators are written in the usual way, where the "x" and "y" in the examples can themselves be arbitrarily complex formulas, as in the examples above. In this syntax, "x" and "y" can be scalar values or per-molecule vectors. For example, "vol/npart" is the division of two scalars, where "vy+vz" is the element-by-element sum of two per-molecule vectors of y and z velocities.

Operators are evaluated left to right and have the usual C-style precedence: unary minus and unary logical NOT operator "!" have the highest precedence, exponentiation "^" is next; multiplication and division are next; addition and subtraction are next; the 4 relational operators "", and ">=" are next; the two remaining relational operators "==" and "!=" are next; then the logical AND operator "&&"; and finally the logical OR operator "||" has the lowest precedence. Parenthesis can be used to group one or more portions of a formula and/or enforce a different order of evaluation than what would occur with the default precedence.

The 6 relational operators return either a 1.0 or 0.0 depending on whether the relationship between x and y is TRUE or FALSE. For example the expression x

These relational and logical operators can be used as a masking or selection operation in a formula. For example, the number of particles whose properties satifsy one or more criteria could be calculated by taking the returned per-molecule vector of ones and zeroes and passing it to the compute reduce command.

Math Functions

Math functions are specified as keywords followed by one or more parenthesized arguments "x", "y", "z", each of which can themselves be arbitrarily complex formulas. In this syntax, the arguments can represent scalar values or per-molecule vectors. In the latter case, the math operation is performed on each element of the vector. For example, "sqrt(npart)" is the sqrt() of a scalar, where "sqrt(y*z)" yields a per-molecule vector with each element being the sqrt() of the product of one particle's y and z coordinates.

Most of the math functions perform obvious operations. The ln() is the natural log; log() is the base 10 log.

The random(x,y) function takes 2 arguments: x = lo and y = hi. It generates a uniform random number between lo and hi. The normal(x,y) function also takes 2 arguments: x = mu and y = sigma. It generates a Gaussian variate centered on mu with variance sigma^2. For equal-style variables, every processor uses the same random number seed so that they each generate the same sequence of random numbers. For particle-style variables, a unique seed is created for each processor. This effectively generates a different random number for each particle being looped over in the particle-style variable.

IMPORTANT NOTE: Internally, there is just one random number generator for all equal-style variables and one for all particle-style variables. If you define multiple variables (of each style) which use the random() or normal() math functions, then the internal random number generators will only be initialized once.

The ceil(), floor(), and round() functions are those in the C math library. Ceil() is the smallest integer not less than its argument. Floor() if the largest integer not greater than its argument. Round() is the nearest integer to its argument.

The ramp(x,y) function uses the current timestep to generate a value linearly intepolated between the specified x,y values over the course of a run, according to this formula:

```
value = x + (y-x) * (timestep-startstep) / (stopstep-startstep)
```

The run begins on startstep and ends on stopstep. Startstep and stopstep can span multiple runs, using the *start* and

stop keywords of the run command. See the run command for details of how to do this.

The stagger(x,y) function uses the current timestep to generate a new timestep. X,y > 0 and x > y is required. The generated timesteps increase in a staggered fashion, as the sequence x,x+y,2x,2x+y,3x,3x+y,etc. For any current timestep, the next timestep in the sequence is returned. Thus if stagger(1000,100) is used in a variable by the dump_modify every command, it will generate the sequence of output timesteps:

```
100,1000,1100,2000,2100,3000,etc
```

The logfreq(x,y,z) function uses the current timestep to generate a new timestep. X,y,z > 0 and y < z is required. The generated timesteps increase in a logarithmic fashion, as the sequence x,2x,3x,...y*x,z*x,2*z*x,3*z*x,...y*z*x,z*z*x,2*z*x*x,etc. For any current timestep, the next timestep in the sequence is returned. Thus if logfreq(100,4,10) is used in a variable by the dump_modify every command, it will generate the sequence of output timesteps:

```
100,200,300,400,1000,2000,3000,4000,10000,20000,etc
```

The vdisplace(x,y) function takes 2 arguments: x = coord0 and y = velocity, and uses the elapsed time to change the coordinate value by a linear displacement due to the applied velocity over the course of a run, according to this formula:

```
value = coord0 + velocity*(timestep-startstep)*dt
```

where dt = the timestep size.

The run begins on startstep. Startstep can span multiple runs, using the *start* keyword of the run command. See the run command for details of how to do this. Note that the thermo_style keyword elaplong = timestep-startstep.

The swiggle(x,y,z) and cwiggle(x,y,z) functions each take 3 arguments: x = coord0, y = amplitude, z = period. They use the elapsed time to oscillate the coordinate value by a sin() or cos() function over the course of a run, according to one of these formulas, where omega = 2 PI / period:

```
value = coord0 + Amplitude * sin(omega*(timestep-startstep)*dt)
value = coord0 + Amplitude * (1 - cos(omega*(timestep-startstep)*dt))
```

where dt = the timestep size.

The run begins on startstep. Startstep can span multiple runs, using the *start* keyword of the run command. See the run command for details of how to do this. Note that the thermo_style keyword elaplong = timestep-startstep.

Special Functions

Special functions take specific kinds of arguments, meaning their arguments cannot be formulas themselves.

The sum(x), min(x), max(x), ave(x), and trap(x) functions each take 1 argument which is of the form "c_ID" or "c_ID[N]" or "f_ID" or "f_ID[N]". The first two are computes and the second two are fixes; the ID in the reference should be replaced by the ID of a compute or fix defined elsewhere in the input script. The compute or fix must produce either a global vector or array. If it produces a global vector, then the notation without "[N]" should be used. If it produces a global array, then the notation with "[N]" should be used, when N is an integer, to specify which column of the global array is being referenced.

These functions operate on the global vector of inputs and reduce it to a single scalar value. This is analagous to the operation of the compute reduce command, which invokes the same functions on per-molecule vectors.

The sum() function calculates the sum of all the vector elements. The min() and max() functions find the minimum and maximum element respectively. The ave() function is the same as sum() except that it divides the result by the length of the vector. The trap() function is the same as sum() except the first and last elements are multiplied by a weighting factor of 1/2 when performing the sum. This effectively implements an integratiion via the trapezoidal rule on the global vector of data. I.e. consider a set of points, equally spaced by 1 in their x coordinate: (1,V1), (2,V2), ..., (N,VN), where the Vi are the values in the global vector of length N. The integral from 1 to N of these points is trap(). When appropriately normalized by the timestep size, this function is useful for calculating integrals of time-series data, like that generated by the fix ave/correlate command.

Particle Vectors

Particle vectors generate one value per particle, so that a reference like "vx" means the x-component of each particles's velocity will be used when evaluating the variable.

Compute References

Compute references access quantities calculated by a compute. The ID in the reference should be replaced by the ID of a compute defined elsewhere in the input script. As discussed in the doc page for the compute command, computes can produce global, per-molecule, or per-grid values. Only global and per-molecule values can be used in a variable. Computes can also produce a scalar, vector, or array. An equal-style variable can only use scalar values, which means a global scalar, or an element of a global vector or array. Particle-style variables can use the same scalar values. They can also use per-molecule vector values. A vector value can be a per-molecule vector itself, or a column of an per-molecule array. See the doc pages for individual computes to see what kind of values they produce.

Examples of different kinds of compute references are as follows. There is no ambiguity as to what a reference means, since computes only produce global or per-molecule quantities, never both.

c_ID	global scalar, or per-molecule vector
c_ID[I]	Ith element of global vector, or Ith column from per-molecule array
c_ID[I][J]	I,J element of global array

If a variable containing a compute is evaluated directly in an input script (not during a run), then the values accessed by the compute must be current. See the discussion below about "Variable Accuracy".

Fix References

Fix references access quantities calculated by a fix. The ID in the reference should be replaced by the ID of a fix defined elsewhere in the input script. As discussed in the doc page for the fix command, fixes can produce global, per-molecule, or per-grid values. Only global and per-molecule values can be used in a variable. Fixes can also produce a scalar, vector, or array. An equal-style variable can only use scalar values, which means a global scalar, or an element of a global vector or array. Particle-style variables can use the same scalar values. They can also use per-molecule vector values. A vector value can be a per-molecule vector itself, or a column of an per-molecule array. See the doc pages for individual fixes to see what kind of values they produce.

The different kinds of fix references are exactly the same as the compute references listed in the above table, where "c_" is replaced by "f_". Again, there is no ambiguity as to what a reference means, since fixes only produce global or per-molecule quantities, never both.

f_ID	global scalar, or per-molecule vector
f_ID[I]	Ith element of global vector, or Ith column from per-molecule array
f_ID[I][J]	I,J element of global array

If a variable containing a fix is evaluated directly in an input script (not during a run), then the values accessed by the fix should be current. See the discussion below about "Variable Accuracy".

Note that some fixes only generate quantities on certain timesteps. If a variable attempts to access the fix on non-allowed timesteps, an error is generated. For example, the fix ave/time command may only generate averaged quantities every 100 steps. See the doc pages for individual fix commands for details.

Variable References

Variable references access quantities calulated by other variables, which will cause those variables to be evaluated. The name in the reference should be replaced by the name of a variable defined elsewhere in the input script. As discussed on this doc page, particle-style variables generate a per-molecule vector of values; all other variable styles generate a global scalar value. An equal-style variable can only use scalar values, which means another equal-style variable. Particle-style variables can use either other equal-style or particle-style variables.

Examples of different kinds of variable references are as follows. There is no ambiguity as to what a reference means, since variables produce only a global scalar or a per-molecule vectors, never both.

v_name	scalar, or per-molecule vector

IMPORTANT NOTE: If you define variables in circular manner like this:

```
variable a equal v_b
variable b equal v_a
print $a
```

then SPARTA may run for a while when the print statement is invoked!

Immediate Evaluation of Variables:

There is a difference between referencing a variable with a leading $ sign (e.g. $x or ${abc}) versus with a leading "v_" (e.g. v_x or v_abc). The former can be used in any command, including a variable command, to force the immediate evaluation of the referenced variable and the substitution of its value into the command. The latter is a required kind of argument to some commands (e.g. the fix ave/spatial or dump custom or thermo_style commands) if you wish it to evaluate a variable periodically during a run. It can also be used in a variable formula if you wish to reference a second variable. The second variable will be evaluated whenever the first variable is evaluated.

As an example, suppose you use this command in your input script to define the variable "v" as

```
variable v equal vol
```

before a run where the simulation box size changes. You might think this will assign the initial volume to the variable "v". That is not the case. Rather it assigns a formula which evaluates the volume (using the thermo_style keyword "vol") to the variable "v". If you use the variable "v" in some other command like fix ave/time then the current volume of the box will be evaluated continuously during the run.

If you want to store the initial volume of the system, you can do it this way:

```
variable v equal vol
variable v0 equal $v
```

The second command will force "v" to be evaluated (yielding the initial volume) and assign that value to the variable "v0". Thus the command

```
thermo_style custom step v_v v_v0
```

would print out both the current and initial volume periodically during the run.

Note that it is a mistake to enclose a variable formula in double quotes if it contains variables preceeded by $ signs. For example,

```
variable vratio equal "${vfinal}/${v0}"
```

This is because the quotes prevent variable substitution (see this section on parsing input script commands), and thus an error will occur when the formula for "vratio" is evaluated later.

Variable Accuracy:

Obviously, SPARTA attempts to evaluate variables containing formulas (*equal* and *particle* style variables) accurately whenever the evaluation is performed. Depending on what is included in the formula, this may require invoking a compute, either directly or indirectly via a thermo keyword, or accessing a value previously calculated by a compute, or accessing a value calculated and stored by a fix.

SPARTA keeps track of all of this during a run or energy minimization. An error will be generated if you attempt to evaluate a variable on timesteps when it cannot produce accurate values. For example, if a thermo_style custom command prints a variable which accesses values stored by a fix ave/time command and the timesteps on which thermo output is generated are not multiples of the averaging frequency used in the fix command, then an error will occur.

An input script can also request variables be evaluated before or after or in between runs, e.g. by including them in a print command. In this case, if a compute is needed to evaluate a variable (either directly or indirectly), SPARTA will not invoke the compute, but it will use a value previously calculated by the compute, and can do this only if it is current. Fixes will always provide a quantity needed by a variable, but the quantity may or may not be current. This leads to one of three kinds of behavior:

(1) The variable may be evaluated accurately. If it contains references to a compute or fix, and these values were calculated on the last timestep of a preceeding run, then they will be accessed and used by the variable and the result will be accurate.

(2) SPARTA may not be able to evaluate the variable and generate an error. For example, if the variable requires a quantity from a compute that is not current, SPARTA will generate an error. This means, for example, that such a variable cannot be evaluated before the first run has occurred. Likewise, in between runs, such a variable cannot be accessed unless it was evaluated on the last timestep of the preceding run, e.g. by thermodynamic output.

One way to get around this problem is to perform a 0-timestep run before using the variable. For example, these commands

```
variable t equal temp
print "Initial temperature = $t"
run 1000
```

will generate an error if the run is the first run specified in the input script, because generating a value for the "t" variable requires a compute for calculating the temperature to be invoked.

However, this sequence of commands would be fine:

```
run 0
variable t equal temp
```

```
print "Initial temperature = $t"
run 1000
```

The 0-timestep run initializes and invokes various computes, including the one for temperature, so that the value it stores is current and can be accessed by the variable "t" after the run has completed. Note that a 0-timestep run does not alter the state of the system, so it does not change the input state for the 1000-timestep run that follows. Also note that the 0-timestep run must actually use and invoke the compute in question (e.g. via thermo or dump output) in order for it to enable the compute to be used in a variable after the run. Thus if you are trying to print a variable that uses a compute you have defined, you could insure it was invoked on the last timestep of the preceding run by including it in thermodynamic output.

Unlike computes, fixes will never generate an error if their values are accessed by a variable in between runs. They always return some value to the variable. However, the value may not be what you expect if the fix has not yet calculated the quantity of interest or it is not current. For example, the fix indent command stores the force on the indenter. But this is not computed until a run is performed. Thus if a variable attempts to print this value before the first run, zeroes will be output. Again, performing a 0-timestep run before printing the variable has the desired effect.

(3) The variable may be evaluated incorrectly. And SPARTA may have no way to detect this has occurred. Consider the following sequence of commands:

```
pair_coeff 1 1 1.0 1.0
run 1000
pair_coeff 1 1 1.5 1.0
variable e equal pe
print "Final potential energy = $e"
```

The first run is performed using one setting for the pairwise potential defined by the pair_style and pair_coeff commands. The potential energy is evaluated on the final timestep and stored by the compute pe compute (this is done by the thermo_style command). Then a pair coefficient is changed, altering the potential energy of the system. When the potential energy is printed via the "e" variable, SPARTA will use the potential energy value stored by the compute pe compute, thinking it is current. There are many other commands which could alter the state of the system between runs, causing a variable to evaluate incorrectly.

The solution to this issue is the same as for case (2) above, namely perform a 0-timestep run before the variable is evaluated to insure the system is up-to-date. For example, this sequence of commands would print a potential energy that reflected the changed pairwise coefficient:

```
pair_coeff 1 1 1.0 1.0
run 1000
pair_coeff 1 1 1.5 1.0
run 0
variable e equal pe
print "Final potential energy = $e"
```

Restrictions:

All *universe*- and *uloop*-style variables defined in an input script must have the same number of values.

Related commands:

next, jump, include, temper, fix print, print

Default: none

write_restart command

NOTE: this command is not yet implemented

Syntax:

```
write_restart file
```

- file = name of file to write restart information to

Examples:

```
write_restart restart.equil
write_restart poly.%.*
```

Description:

Write a binary restart file of the current state of the simulation. See the read_restart command for information about what is stored in a restart file.

During a long simulation, the restart command is typically used to dump restart files periodically. The write_restart command is useful after a run or whenever you wish to write out a single current restart file.

Similar to dump files, the restart filename can contain two wild-card characters. If a "*" appears in the filename, it is replaced with the current timestep value. If a "%" character appears in the filename, then one file is written by each processor and the "%" character is replaced with the processor ID from 0 to P-1. An additional file with the "%" replaced by "base" is also written, which contains global information. For example, the files written for filename restart.% would be restart.base, restart.0, restart.1, ... restart.P-1. This creates smaller files and can be a fast mode of output and subsequent input on parallel machines that support parallel I/O.

Restart files can be read by a read_restart command to restart a simulation from a particular state. Because the file is binary (to enable exact restarts), it may not be readable on another machine. In this case, the restart2data program in the tools directory can be used to convert a restart file to an ASCII data file. Both the read_restart command and restart2data tool can read in a restart file that was written with the "%" character so that multiple files were created.

Restrictions:

This command requires inter-processor communication to migrate particles before the restart file is written. This means that your system must be ready to perform a simulation before using this command.

Related commands:

restart, read_restart

Default: none