SPARTA

Release 6 Jul 2020

Local SPARTA

23/07/2020

CONTENTS:

CHAPTER

ONE

VERSION INFO

The SPARTA "version" is the date when it was released, such as 3 Mar 2014. 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, and at the top of the first page of the manual (this page).

- 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.
- At some point, there also will be a Developer.pdf file in the doc directory, which describes the internal structure and algorithms of SPARTA. NOTE: as of 21 Apr 2015, this file is not yet available.

Note:

- The source for this version of the manual is in the docs branch of this fork.
- The PDF and EPUB files for this reformatted version are also available.

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

SPARTA is a Direct Simulation Monte 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), or sometimes by request under the terms of the GNU Lesser General Public License (LGPL).

The primary developers of SPARTA are Steve Plimpton, and Michael Gallis 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* since it gives quick access to documentation for all SPARTA commands.

1.1 Introduction

These sections provide an overview of what SPARTA can do, describe what it means for SPARTA to be an open-source code, and acknowledge the funding and people who have contributed to SPARTA.

Contents

1.1.1 What is SPARTA

SPARTA is a Direct Simulation Montel Carlo code that models rarefied gases, using collision, chemistry, and boundary condition models. It uses a hierarchical Cartesian grid to track and group particles for 3d or 2d or axisymmetric models. Objects emedded in the gas are represented as triangulated surfaces and cut through grid cells.

For examples of SPARTA simulations, see 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 *performance* 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, or sometimes by request under the terms of the GNU Lesser General Public License (LGPL), which means you can use or modify the code however you wish. The only restrictions imposed by the GPL or LGPL are on how you distribute the code further. See *open-source* below 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 *Section 10* 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 *Section 1.5* below for more information on SPARTA funding and individuals who have contributed to SPARTA.

1.1.2 SPARTA features

This section highlights SPARTA features, with links to specific commands which give more details. The *next section* <*grids*> illustrates the kinds of grid geometries and surface definitions which SPARTA supports.

If SPARTA doesn't have your favorite collision model, boundary condition, or diagnostic, see *Section 10 <modify>* of the manual, which describes how it can be added 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 <howto-other-code>*: SPARTA calls other code, other code calls SPARTA, umbrella code calls both

Models

- *3d or 2d* or *2d-axisymmetric* domains
- variety of *global boundary conditions*
- *create particles* within flow volume
- emit particles from simulation box faces due to *flow properties*
- emit particles from simulation box faces due to *profile defined in file*
- emit particles from surface elements due to *normal and flow properties*
- *ambipolar* approximation for ionized plasmas

Geometry

- *Cartesian, hierarchical grids* with multiple levels of local refinement
- *create grid from input script* or *read from file <command-read-grid>*
- embed :triangulated (3d) or line-segmented (2d) surfaces in grid, *read in from file*

Gas-phase collisions and chemistry

- collisions between all particles or pairs of species groups within grid cells
- *collision models:* VSS (variable soft sphere), VHS (variable hard sphere), HS (hard sphere)
- *chemistry models:* TCE, QK

Surface collisions and chemistry

- for surface elements or global simulation box *boundaries*
- *collisions:* specular or diffuse
- *reactions*

Performance

- *grid cell weighting* of particles
- *adaptation* of the grid cells between runs
- *on-the-fly adaptation* of the grid cells
- *static* load-balancing of grid cells or particles
- *dynamic* load-balancing of grid cells or particles

Diagnostics

- *global boundary statistics*
- *per grid cell statistics*
- *per surface element statistics*
- time-averaging of *global grid*, *surface* statistics

Output

- *log file of statistical info*
- *dump files* (text or binary) of per particle, per grid cell, per surface element values
- binary *restart files*
- on-the-fly *rendered images and movies* of particles, grid cells, surface elements

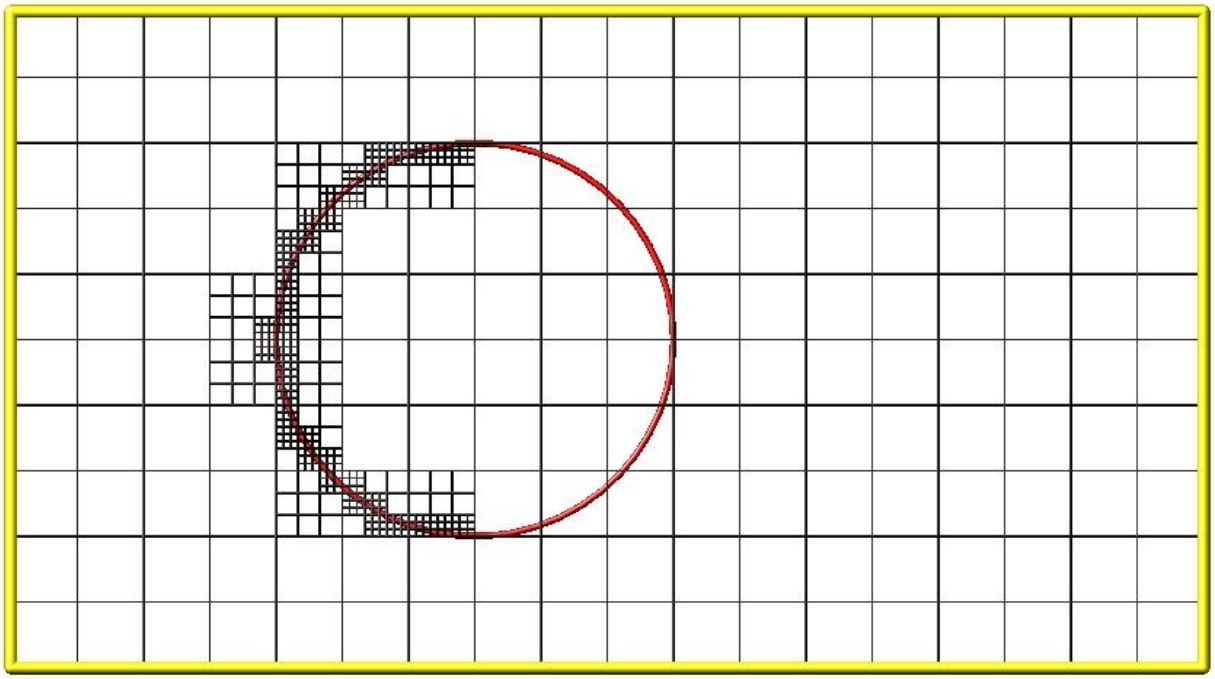
Pre- and post-processing

- Various pre- and post-processing serial tools are packaged with SPARTA; see *Section 9* of the manual.
- 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.

1.1.3 Grids and surfaces in SPARTA

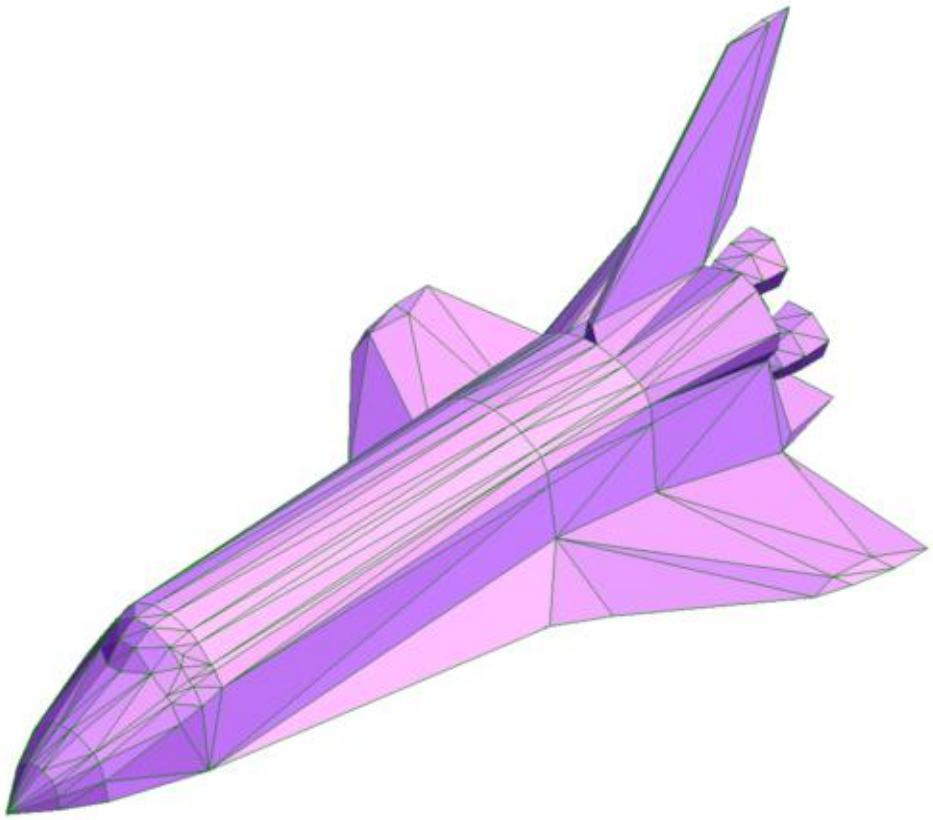
SPARTA overlays a grid over the simulation domain which is used to track particles and to co-locate particles in the same grid cell for performing collision and chemistry operations. SPARTA uses a Cartesian hierarchical grid. Cartesian means that the faces of a grid cell are aligned with the Cartesian xyz axes. Hierarchical means that individual grid cells can be sub-divided into smaller cells, recursively. This allows for flexible grid cell refinement in any region of the simulation domain. E.g. around a surface, or in a high-density region of the gas flow.

An example 2d hierarchical grid is shown in the diagram, for a circular surface object (in red) with the grid refined on the upwind side of the object (flow from left to right).



Objects represented with a surface triangulation (line segments in 2d) can also be read in to define objects which particles flow around. Individual surface elements are assigned to grid cells they intersect with, so that particle/surface collisions can be efficiently computed.

As an example, here is coarsely triangulated representation of the space shuttle (only 616 triangles!), which could be embedded in a simulation box. Click on the image for a larger picture.



See *Details of grid geometry in SPARTA* and *Details of surfaces in SPARTA* for more details of both the grids and surface objects that SPARTA supports and how to define them.

1.1.4 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 topics.

- 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, *error-common* describes how to report it.
- If you publish a paper using SPARTA results, send the citation (and any cool pictures or movies) to add to the Publications, Pictures, and Movies pages of the SPARTA WWW Site, with links and attributions back to you.
- 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 *Additional tools*. If you write a new tool that others 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 boundary conditions, collision or chemistry models, diagnostic computations, etc. *Modifying & extending SPARTA* of the manual 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.
- Cash. Small denominations, unmarked bills preferred. Paper sack OK. Leave on desk. VISA also accepted. Chocolate chip cookies encouraged.

1.1.5 Acknowledgments and citations

SPARTA development has been funded by the US Department of Energy (DOE).

If you use SPARTA results in your published work, please cite the paper(s) listed under the Citing SPARTA link of the SPARTA WWW page, and include a pointer to the SPARTA WWW Site (http://sparta.sandia.gov):

The Publications link on the SPARTA WWW page lists papers that have cited SPARTA. If your paper is not listed there, 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
- Michael Gallis, magalli at sandia.gov

1.2 Getting Started

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

Contents

1.2.1 What's in the SPARTA distribution

When you download SPARTA you will need to unzip and untar the downloaded file with the following commands:

```
gunzip sparta*.tar.gz
tar xvf sparta*.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
data	files with species, collision, and reaction parameters
doc	documentation
examples	simple test problems
python	Python wrapper
src	source files
tools	pre- and post-processing tools

1.2.2 Making SPARTA

This section has the following sub-sections:

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, JPEG).

Please read this section carefully. If you are not comfortable with cmake, makefiles, or building codes on a Linux platform, or running an MPI job on your machine, please find a local expert to help you.

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 in the src/MAKE directory or .cmake file in cmake/presets, send it to the developers and we'll include it in future SPARTA releases.

Steps to build a SPARTA executable using make:

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 g++
or
gmake mac
```

Note that on a 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 spa_g++ or spa_mac is produced, you're done; it's your lucky day.

Note that by default none of the SPARTA optional packages 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. Copy 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 Linux 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. 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 SPA_INC variable is used to include options that turn on ifdefs within the SPARTA code. The options that are currently recognized are:

- -DSPARTA_GZIP
- -DSPARTA_JPEG
- -DSPARTA_PNG
- -DSPARTA_FFMPEG
- -DSPARTA_MAP
- -DSPARTA_UNORDERED_MAP
- -DSPARTA_SMALL
- -DSPARTA_BIG
- -DSPARTA_BIGBIG
- -DSPARTA_LONGLONG_TO_LONG

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

If you use -DSPARTA_JPEG and/or -DSPARTA_PNG, the *dump command* will be able to write out JPEG and/or PNG image files respectively. If not, it will only be able to write out PPM image files. For JPEG files, you must also link SPARTA with a JPEG library, as described below. For PNG files, you must also link SPARTA with a PNG library, as described below.

If you use -DSPARTA_FFMPEG, the *dump movie* command will be available to support on-the-fly generation of rendered movies the need to store intermediate image files. It requires that your machines supports the "popen" function in the standard runtime library and that an FFmpeg executable can be found by SPARTA during the run.

If you use -DSPARTA_MAP, SPARTA will use the STL map class for hash tables. This is less efficient than the unordered map class which is not yet supported by all C++ compilers. If you use -DSPARTA_UNORDERED_MAP, SPARTA will use the unordered_map class for hash tables and will assume it is part of the STL (e.g. this works for Clang++). The default is to use the unordered map class from the "tr1" extension to the STL which is supported by most compilers. So only use either of these options if the build complains that unordered maps are not recognized.

Use at most one of the -DSPARTA_SMALL, -DSPARTA_BIG, -DSPARTA_BIGBIG settings. The default is -DSPARTA_BIG. These refer to use of 4-byte (small) vs 8-byte (big) integers within SPARTA, as described in src/spatype.h. The only reason to use the BIGBIG setting is if you have a regular grid with more than ~2 billion grid cells or a hierarchical grid with enough levels that grid cell IDs cannot fit in a 32-bit integer. In either case,

SPARTA will generate an error message for "Cell ID has too many bits". See *Details of grid geometry in SPARTA* of the manual for details on how cell IDs are formatted. The only reason to use the SMALL setting is if your machine does not support 64-bit integers.

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 particles per processor (2^31), which should not normally be a restriction since such a problem would have a huge per-processor memory and would run very slowly in terms of CPU secs/timestep.

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 use that data type.

Using one of the -DPACK_ARRAY, -DPACK_POINTER, and -DPACK_MEMCPY options can make for faster parallel FFTs on some platforms. The -DPACK_ARRAY setting is the default. See the *compute fft/grid command* for info about FFTs. See Step 6 below for info about building SPPARKS with an FFT library.

Step 5

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

If you 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, 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 is found (via MPI_INC), and the MPI library file is found (via MPI_PATH), and the name of the library file (via MPI_LIB). See Makefile.serial for an example of how this can be done.

If you are installing MPI yourself, we recommend MPICH 1.2 or 2.0 or OpenMPI. MPICH can be downloaded from the Argonne MPI site. OpenMPI can be downloaded the OpenMPI site. If you are running on a big parallel platform, your system admins or the vendor should have already installed a version of MPI, which will be faster than MPICH or OpenMPI, so find out how to build and link with it. If you use MPICH or OpenMPI, 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 use 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. You will also need to build the STUBS library for your platform before making SPARTA itself. From the src directory, type "make mpi-stubs", or from within 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() function 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 built with its FFT package, which contains commands that invoke FFTs.

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 FFT 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 or FFTW3. FFTW2 and NONE are supported as legacy options. Selecting -DFFT_FFTW will use the FFTW3 library and -DFFT_NONE will use the KISS library described above. 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 -DFFTW_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, which can speed-up the calculation, 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.

Step 7

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

A standard JPEG library usually goes by the name libjpeg.a or libjpeg.so 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.

A standard PNG library usually goes by the name libpng.a or libpng.so and has an associated header file png.h. Whichever PNG 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 none of the SPARTA optional packages 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, and you have pre-built any other needed libraries (e.g. MPI), all you need to do from the src directory is type one of the following:

```
make foo
make -j N foo
gmake foo
gmake -j N foo
```

The -j or -j N switches perform a parallel build which can be much faster, depending on how many cores your compilation machine has. N is the number of cores the build runs on.

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

Steps to build a SPARTA executable using CMake:

Step 0

Please review https://github.com/sparta/sparta/blob/master/BUILD_CMAKE.md and ensure that CMake version 3.10.0 or greater is installed:

```
which cmake
which cmake3
cmake --version
```

On clusters and supercomputers one can use modules to load cmake:

```
module avail cmake
module load <CMAKE>
```

On Linux one may use apt, yum, or pacman to install cmake.

On Mac one may use brew or macports to install cmake.

Step 1

The cmake directory contains the CMake source files for SPARTA. Create a build directory and from within the build directory, run cmake:

```
mkdir build
cd build
cmake -LH -DSPARTA_MACHINE=tutorial /path/to/sparta/cmake
```

This will generate the default Makefiles and print the SPARTA CMake options. To list the generated targets, do:

```
make help
```

Now you can try to build the SPARTA binaries with:

```
make
```

If everything works, an executable named spa_tutorial and a library named libsparta.a will be produced in build/src.

Step 2

If Step 1 did not work, see if you can use any system presets from /path/to/sparta/cmake/presets. To select a preset:

```
cd build

# Clear the CMake files
rm -rf CMake*

cmake -C /path/to/sparta/cmake/presets/<NAME>.cmake -DSPARTA_MACHINE=tutorial /path/
↪to/sparta/cmake
make
```

Step 3

If Step 2 did not work, look at cmake -LH for a list of SPARTA CMake options and their meaning, then modify one or more of those options by doing:

```
cd build
rm -rf CMake*
cmake -C /path/to/sparta/cmake/presets/<NAME>.cmake -D<OPTION_NAME>=<VALUE> /path/to/
↪sparta/cmake
make
```

where <OPTION_NAME> and <VALUE> correspond to valid option value pairs listed by cmake -LH. For the SPARTA_DEFAULT_CXX_COMPILE_FLAGS option, see *Step 4*.

For a full list of CMake option value pairs, see cmake -LAH. The most relevant CMake options (with example values) for our purposes here are:

```
-DCMAKE_C_COMPILER=gcc
-DCMAKE_CXX_COMPILER=/usr/local/bin/g++
-DCMAKE_CXX_FLAGS=-O3
```

If your cmake command line is getting too long, consider placing it in a bash script and escaping newlines. For example:

```
cmake \
-C  /path/to/sparta/cmake/presets/<NAME>.cmake \
-D  -D<OPTION_NAME>=<VALUE> \
/path/to/sparta/cmake :pre
```

Step 4

The SPARTA_DEFAULT_CXX_COMPILE_FLAGS option passes flags to the compiler when building object files. Note that if you change any -D setting in this section, you should do a full re-compile, after typing "make clean".

The SPARTA_DEFAULT_CXX_COMPILE_FLAGS option is typically used to include options that turn on ifdefs within the SPARTA code. The options that are currently recogized are:

```
-DSPARTA_GZIP
-DSPARTA_JPEG
-DSPARTA_PNG
-DSPARTA_FFMPEG
```

(continues on next page)

(continued from previous page)

```
-DSPARTA_MAP
-DSPARTA_UNORDERED_MAP
-DSPARTA_SMALL
-DSPARTA_BIG
-DSPARTA_BIGBIG
-DSPARTA_LONGLONG_TO_LONG :ul
```

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

If you use -DSPARTA_JPEG and/or -DSPARTA_PNG, the *dump image command* will be able to write out JPEG and/or PNG image files respectively. If not, it will only be able to write out PPM image files. For JPEG files, you must also link SPARTA with a JPEG library, as described below. For PNG files, you must also link SPARTA with a PNG library, as described below.

If you use -DSPARTA_FFMPEG, the *dump movie* command will be available to support on-the-fly generation of rendered movies the need to store intermediate image files. It requires that your machines supports the "popen" function in the standard runtime library and that an FFmpeg executable can be found by SPARTA during the run.

If you use -DSPARTA_MAP, SPARTA will use the STL map class for hash tables. This is less efficient than the unordered map class which is not yet supported by all C++ compilers. If you use -DSPARTA_UNORDERED_MAP, SPARTA will use the unordered_map class for hash tables and will assume it is part of the STL (e.g. this works for Clang++). The default is to use the unordered map class from the "tri1" extension to the STL which is supported by most compilers. So only use either of these options if the build complains that unordered maps are not recognized.

Use at most one of the -DSPARTA_SMALL, -DSPARTA_BIG, -DSPARTA_BIGBIG settings. The default is -DSPARTA_BIG. These refer to use of 4-byte (small) vs 8-byte (big) integers within SPARTA, as described in src/spatype.h. The only reason to use the BIGBIG setting is if you have a regular grid with more than ~2 billion grid cells or a hierarchical grid with enough levels that grid cell IDs cannot fit in a 32-bit integer. In either case, SPARTA will generate an error message for "Cell ID has too many bits". See "Section 4.8"_Section_howto.html#howto_8 of the manual for details on how cell IDs are formatted. The only reason to use the SMALL setting is if your machine does not support 64-bit integers.

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 particles per processor (2^31), which should not normally be a restriction since such a problem would have a huge per-processor memory and would run very slowly in terms of CPU secs/timestep.

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 use that data type.

Using one of the -DPACK_ARRAY, -DPACK_POINTER, and -DPACK_MEMCPY options can make for faster parallel FFTs on some platforms. The -DPACK_ARRAY setting is the default. See the "compute fft/grid"_compute_fft_grid.html command for info about FFTs. See STEP ??? below for info about building SPARKS with an FFT library.

Step 5

This step is optional. Once you get *Step 3* and *Step 4* working by modifying the options to the cmake command, try setting the same options in /path/to/sparta/cmake/presets/<NEW>.cmake by copying /path/to/sparta/cmake/presets/<NAME>.cmake and modifying the cmake source code. Note that the CMake cache is sticky and will only evict a cached option value pair if you use -D or the FORCE argument to CMake's set routine.

Now just do:

```
cd build
rm -rf CMake*
cmake -C /path/to/sparta/cmake/presets/<NEW>.cmake /path/to/sparta/cmake
make
```

consider sharing and vetting <NEW>.cmake by opening a pull request at https://github.com/sparta/sparta/.

Step 6

This step explains how to enable and select MPI in the SPARTA CMake configuration. There may already be a preset in /path/to/sparta/cmake/presets that selects the correct MPI installation.

By default, SPARTA configures with MPI enabled and cmake will print which MPI was selected. To build serial binaries, use SPARTA's MPI_STUBS package:

```
cmake -DPKG_MPI_STUBS=ON /path/to/sparta/cmake
```

You may want a different MPI installation than CMake finds. CMake uses module files such as FindMPI.cmake to handle wiring in a given installation of a library and its headers. If you're on a cluster or supercomputer, use module before running cmake so that cmake finds the MPI installation you'd like to use:

```
# Show which modules are loaded
module list

# Show which modules are available
module avail

module load <MPI> :pre
```

On Linux one may use apt, yum, or pacman to install MPI.

On Mac one may use brew or macports to install MPI.

Verify that cmake found the correct MPI installation:

```
cd build
rm -rf CMake*

# cmake should print "Found MPI*" strings
cmake [options] /path/to/sparta/cmake :pre
```

Note that if the preset file you're using enables PKG_MPI_STUBS, MPI will not be searched for unless you explicitly disable PKG_MPI_STUBS in the preset file.

If you'd like to use a custom MPI installation or cmake is not locating the MPI installation you've selected via the module command or package manager, try export MPI_ROOT=/path/to/mpi/install before running cmake. Otherwise, please see https://cmake.org/cmake/help/v3.10/module/FindMPI.html#variables-for-locating-mpi. Note that this documentation link is for CMake version 3.10.

Step 7

You may select between three thiry party libraries (TPL) for FFT which SPARTA uses when configured with cmake -DFFT={FFTW2,FFTW3,MKL}. SPARTA also provides a FFT package which can be selected with cmake -DPKG_FFT=ON.

You may need to install the FFT TPL you're interested in using. If you're on a cluster or supercomputer, use module before running cmake so that cmake finds the FFT installation you'd like to use:

```
# Show which modules are loaded
module list

# Show which modules are available
module avail

module load <FFT> :pre
```

On Linux one may use apt, yum, or pacman to install FFT.

On Mac one may use brew or macports to install FFT.

Verify that cmake found the correct MPI installation:

```
cd build
rm -rf CMake*

# cmake should print "Found FFT*" strings
cmake [options] /path/to/sparta/cmake :pre
```

Note that if the preset file you're using enables PKG_FFT, FFT will not be searched for unless you explicitly disable PKG_FFT in the preset file.

If you'd like to use a custom FFT installation or cmake is not locating the FFT installation you've selected via the module command or package manager, try export FFT_ROOT=/path/to/fft/install before running cmake. Otherwise, please open an issue at https://github.com/sparta/sparta/issues.

Step 8

You may select between 2 TPLs, JPEG or PNG, for writing out JPEG or PNG files via the "dump image"_dump_image.html command. To select a TPL, use:

```
cmake -DBUILD_JPEG=ON /path/to/sparta/cmake
```

or:

```
cmake -DBUILD_PNG=ON /path/to/sparta/cmake
```

If you'd like to use a custom jpeg or png installation, please see https://cmake.org/cmake/help/v3.10/module/FindJPEG.html or https://cmake.org/cmake/help/v3.10/module/FindPNG.html. Note that these documentation links are for CMake version 3.10.

Step 9

By default, none of the SPARTA optional packages are installed. To build SPARTA with optional packages, use:

```
cmake -DPKG_XXX=ON /path/to/sparta/cmake
```

Where XXX is the package to enable. For a full list of optional packages, see:

```
cmake -LH /path/to/sparta/cmake
```

Step 10

Once you have a correct cmake command line or the <NAME>.cmake preset file, just do:

```
cd build
cmake [OPTIONS] /path/to/sparta/cmake
```

or:

```
cd build
cmake -C /path/to/sparta/cmake/presets/<NAME>.cmake -DSPARTA_MACHINE=tutorial /path/
↪to/sparta/cmake
```

```
make -j N
```

The -j or -j N switches perform a parallel build which can be much faster, depending on how many cores your compilation machine has. N is the number of cores the build runs on.

You should get build/src/spa_tutorial and build/src/libsparta.a.

Errors that can occur when making SPARTA:

Important: 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 g++
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.

(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 using make:

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.

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.

Additional build tips using CMake:

Building SPARTA for multiple platforms. It's best to build SPARTA for multiple platforms from different build directories. However, each target creates its own spa_TARGET binary and multiple targets can be built from the same build directory. Note that the *.o object files in build/src will reflective of the most recent build configuration. Also note that if BUILD_SHARED_LIBS was enabled, libsparta will be reflective of the most recent build configuration.

Cleaning up. Typing "make clean" will delete all binary files for the most recent build configuration.

Building for a Mac:

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

Building for Windows:

At some point we may provide a pre-built Windows executable for SPARTA. Until then you will need to build an executable from source files.

One way to do this is install and use cygwin to build SPARTA with a standard Linux make or CMake, just as you would on any Linux box.

You can also import the *.cpp and *.h files into Microsoft Visual Studio. If someone does this and wants to provide project files or other Windows build tips, please send them to the developers and we will include them in the distribution.

1.2.3 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, the FFT package which includes a *compute fft/grid command* and a 2d and 3d FFT library.

For make:

You can see the list of all packages by typing "make package" from within the src directory of the SPARTA distribution. This also lists various make commands that can be used to manipulate packages.

For CMake:

You can see the list of all packages by typing "cmake -DSPARTA_LIST_PKGS=ON" from within the build directory.

If you use a command in a SPARTA input script that is part of a 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.

Including/excluding packages with make:

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

```
make yes-fft
make g++
```

or

```
make no-fft
make g++
```

Note: You should NOT include/exclude packages and build SPARTA in a single make command using multiple targets, e.g. `make yes-fft g++`. This is because the make procedure creates a list of source files that will be out-of-date for the build if the package configuration changes within the same command.

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.

If you will never run simulations that use the features in a particular packages, there is no reason to include it in your build.

When you download a SPARTA tarball, no packages are pre-installed in the src directory.

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 = fft for the FFT package. You can also type "make yes-all", or "make no-all" to include/exclude all packages. Type "make package" to see all of the package-related make options.

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/FFT or src/KOKKOS), 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.

Typing "make package-update" or "make pu" 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" or "make ps" will show which packages are currently included. For those that are included, it will list any 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.

Including/excluding packages with CMake:

To use (or not use) a package you must include it (or exclude it) before building SPARTA. From the build directory, do:

```
cmake -DPKG_FFT=ON /path/to/sparta/cmake
make -j
```

or

```
cmake -DPKG_FFT=OFF /path/to/sparta/cmake
make -j :pre
```

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.

If you will never run simulations that use the features in a particular packages, there is no reason to include it in your build.

When you download a SPARTA tarball, no packages are pre-installed in the build/src directory.

Packages are included or excluded by typing "cmake -DPKG_NAME=ON" or "cmake -DPKG_NAME=OFF", where "NAME" is the name of the package in upper-case, e.g. name = FFT for the FFT package. You can also type "cmake -DSPARTA_ENABLE_ALL_PKGS=ON", or "cmake -DSPARTA_DISABLE_ALL_PKGS=ON" to include or exclude all packages. Type "cmake -DSPARTA_LIST_PKGS=ON" to see all of the package-related CMake options.

NOTE: Inclusion or exclusion of a package works by setting CMake boolean variables to generate the correct Makefile targets and dependencies. After you have included or excluded a package, you must re-build SPARTA.

If a SPARTA package has source code changes, simply run "make" to rebuild SPARTA with these changes.

Typing "cmake" from the build directory will show which packages are currently included.

1.2.4 Building SPARTA as a library

SPARTA can be built as either a static or shared library, which can then be called from another application or a scripting language. See *Coupling SPARTA to other codes* for more info on coupling SPARTA to other codes. See *Python interface to SPARTA* for more info on wrapping and running SPARTA from Python.

The CMake build system will produce the library static of dynamic libsparta library in build/src.

Static library:

CMake builds sparta as a static library in libsparta.a, by default.

To build SPARTA as a static library ("*.a" file on Linux), type

```
make foo mode=lib
```

where foo is the machine name. This kind of library is typically used to statically link a driver application to SPARTA, so that you can insure all dependencies are satisfied at compile time. This will use the ARCHIVE and ARFLAGS settings in src/MAKE/Makefile.foo. The build will create the file libsparta_foo.a which another application can link to. It will also create a soft link libsparta.a, which will point to the most recently built static library.

Shared library:

To build SPARTA as a shared library ("*.so" file on Linux), which can be dynamically loaded, e.g. from Python, type

```
make foo mode=shlib
```

or:

```
cmake -C /path/to/sparta/cmake/presets/foo.cmake -DBUILD_SHARED_LIBS=ON /path/to/
↪sparta/cmake
make
```

where foo is the machine name. This kind of library is required when wrapping SPARTA with Python; see *Python interface to SPARTA* for details. This will use the SHFLAGS and SHLIBFLAGS settings in src/MAKE/Makefile.foo and perform the build in the directory Obj_shared_foo. This is so that each file can be compiled with the -fPIC flag which is required for inclusion in a shared library. The build will create the file libsparta_foo.so which another application can link to dyamically. It will also create a soft link libsparta.so, which will point to the most recently built shared library. This is the file the Python wrapper loads by default.

Note that for a shared library to be usable by a calling program, all the auxiliary libraries it depends on must also exist as shared libraries. This will be the case for libraries included with SPARTA, such as the dummy MPI library in src/STUBS or any package libraries in lib/packages, since they are always built as shared libraries using the -fPIC switch. However, if a library like MPI or FFTW does not exist as a shared library, the shared library build will generate an error. This means you will need to install a shared library version of the auxiliary library. The build instructions for the library should tell you how to do this.

Here is an example of such errors when the system FFTW or provided lib/colvars library have not been built as shared libraries:

```
/usr/bin/ld: /usr/local/lib/libfftw3.a(mapflags.o): relocation
R_X86_64_32 against :ref:`.rodata' can not be used when making a shared
object; recompile with -fPIC
/usr/local/lib/libfftw3.a: could not read symbols: Bad value
```

```
/usr/bin/ld: ../../lib/colvars/libcolvars.a(colvarmodule.o):
relocation R_X86_64_32 against`pthread_key_create' can not be used
when making a shared object; recompile with -fPIC
../../lib/colvars/libcolvars.a: error adding symbols: Bad value
```

As an example, here is how to build and install the MPICH library, a popular open-source version of MPI, distributed by Argonne National Labs, as a shared library in the default /usr/local/lib location:

```
./configure --enable-shared
make
make install
```

You may need to use `sudo make install` in place of the last line if you do not have write privileges for /usr/local/lib. The end result should be the file /usr/local/lib/libmpich.so.

Additional requirement for using a shared library:

The operating system finds shared libraries to load at run-time using the environment variable LD_LIBRARY_PATH.

Using CMake, ensure that CMAKE_INSTALL_PREFIX is set properly and then run "make -j install" or add build/src to LD_LIBRARY_PATH in your shell's environment.

Using make, you may wish to copy the file src/libsparta.so or src/libsparta_g++.so (for example) to a place the system can find it by default, such as /usr/local/lib, or you may wish to add the SPARTA src directory to LD_LIBRARY_PATH, so that the current version of the shared library is always available to programs that use it.

For the csh or tcsh shells, you would add something like this to your ~/.cshrc file:

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/home/sjplimp/sparta/src
```

Calling the SPARTA library:

Either flavor of library (static or shared) allows one or more SPARTA objects to be instantiated from the calling program.

When used from a C++ program, all of SPARTA is wrapped in a SPARTA_NS namespace; you can safely use any of its classes and methods from within the calling code, as needed.

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

See *How-to discussions* of the manual for ideas on how to couple SPARTA to other codes via its library interface. See *Python interface to SPARTA* 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 define the C-style API for using SPARTA as a library. See *Library interface to SPARTA* of the manual for a description of the interface and how to extend it for your needs.

1.2.5 Running SPARTA

By default, SPARTA runs by reading commands from standard input. Thus if you run the SPARTA executable by itself, e.g.

```
spa_g++
```

it will simply wait, expecting commands from the keyboard. Typically you should put commands in an input script and use I/O redirection, e.g.

```
spa_g++ < in.file
```

For parallel environments this should also work. If it does not, use the '-in' command-line switch, e.g.

```
spa_g++ -in in.file
```

Commands 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 one of the benchmarks on a Linux box, using mpirun to launch a parallel job:

```
cd src
make g++
cp spa_g++ ../bench
cd ../bench
mpirun -np 4 spa_g++ < in.free
```

or:

```
cd build
cmake -DCMAKE_CXX_COMPILER=g++ -DSPARTA_MACHINE=g++ /path/to/sparta/cmake
cp src/spa_g++ /path/to/bench
cd /path/to/bench
mpirun -np 4 spa_g++ < in.free
```

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

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

Note that this sequence of commands copies the SPARTA executable (spa_g++) 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 spa_g++ 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 *Errors* 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. The random numbers used by each processor will be different so you should only expect statistical consistency if the same problem is run on different numbers of processors.

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.

1.2.6 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:

- -e or -echo
- -i or -in
- -h or -help
- -k or -kokkos
- -l or -log
- -p or -partition
- -pk or -package
- -pl or -plog
- -ps or -pscreen
- -sc or -screen
- -sf or -suffix
- -v or -var

For example, spa_g++ might be launched as follows:

```
mpirun -np 16 spa_g++ -v f tmp.out -l my.log -sc none < in.sphere
mpirun -np 16 spa_g++ -var f tmp.out -log my.log -screen none < in.sphere
```

Here are the details on the options:

```
-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. *spa_g++ < 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 (fix, compute, collide, etc). SPARTA will print the info and immediately exit if this switch is used.

```
-kokkos on/off keyword/value ...
```

Explicitly enable or disable KOKKOS support, as provided by the KOKKOS package. Even if SPARTA is built with this package, as described above in *Making SPARTA with optional packages*, this switch must be set to enable running with the KOKKOS-enabled styles the package provides. If the switch is not set (the default), SPARTA will operate as if the KOKKOS package were not installed; i.e. you can run standard SPARTA for testing or benchmarking purposes.

Additional optional keyword/value pairs can be specified which determine how Kokkos will use the underlying hardware on your platform. These settings apply to each MPI task you launch via the "mpirun" or "mpiexec" command. You may choose to run one or more MPI tasks per physical node. Note that if you are running on a desktop machine, you typically have one physical node. On a cluster or supercomputer there may be dozens or 1000s of physical nodes.

Either the full word or an abbreviation can be used for the keywords. Note that the keywords do not use a leading minus sign. I.e. the keyword is "t", not "-t". Also note that each of the keywords has a default setting. Example of when to use these options and what settings to use on different platforms is given in *Accelerating SPARTA performance*.

- d or device
- g or gpus
- t or threads
- n or numa

```
device Nd
```

This option is only relevant if you built SPARTA with KOKKOS_DEVICES=Cuda, you have more than one GPU per node, and if you are running with only one MPI task per node. The Nd setting is the ID of the GPU on the node to run on. By default Nd = 0. If you have multiple GPUs per node, they have consecutive IDs numbered as 0,1,2,etc. This setting allows you to launch multiple independent jobs on the node, each with a single MPI task per node, and assign each job to run on a different GPU.

```
gpus Ng Ns
```

This option is only relevant if you built SPARTA with KOKKOS_DEVICES=Cuda, you have more than one GPU per node, and you are running with multiple MPI tasks per node. The Ng setting is how many GPUs you will use per node. The Ns setting is optional. If set, it is the ID of a GPU to skip when assigning MPI tasks to GPUs. This may be useful if your desktop system reserves one GPU to drive the screen and the rest are intended for computational work like running SPARTA. By default Ng = 1 and Ns is not set.

Depending on which flavor of MPI you are running, SPARTA will look for one of these 4 environment variables

```
SLURM_LOCALID (various MPI variants compiled with SLURM support)
MPT_LRANK (HPE MPI)
MV2_COMM_WORLD_LOCAL_RANK (Mvapich)
OMPI_COMM_WORLD_LOCAL_RANK (OpenMPI)
```

which are initialized by the "srun", "mpirun" or "mpiexec" commands. The environment variable setting for each MPI rank is used to assign a unique GPU ID to the MPI task.

```
threads Nt
```

This option assigns Nt number of threads to each MPI task for performing work when Kokkos is executing in OpenMP or pthreads mode. The default is Nt = 1, which essentially runs in MPI-only mode. If there are Np MPI tasks per physical node, you generally want Np*Nt = the number of physical cores per node, to use your available hardware optimally. If SPARTA is compiled with KOKKOS_DEVICES=Cuda, this setting has no effect.

```
-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.sparta. If this switch is used, SPARTA writes to the specified file. In multi-partition mode, if the switch is not used, a log.sparta file is created with hi-level status information. Each partition also writes to a log.sparta.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.

To run multiple independent simulations from one input script, using multiple partitions, see *Running multiple simulations from one input script* of the manual. World- and universe-style variables are useful in this context.

```
-package style args ....
```

Invoke the *package command* with style and args. The syntax is the same as if the command appeared at the top of the input script. For example "-package kokkos on gpus 2" or "-pk kokkos g 2" is the same as *package kokkos g 2* in the input script. The possible styles and args are documented on the *package command* doc page. This switch can be used multiple times.

Along with the *-suffix command-line switch*, this is a convenient mechanism for invoking the KOKKOS accelerator package and its options without having to edit an input script.

```
-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.sparta) If this option is not used the log file for partition N is log.sparta.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 args
```

Use variants of various styles if they exist. The specified style can be *kk*. This refers to optional KOKKOS package that SPARTA can be built with, as described above in *Making SPARTA with optional packages*.

Along with the "-package" command-line switch, this is a convenient mechanism for invoking the KOKKOS accelerator package and its options without having to edit an input script.

As an example, the KOKKOS package provides a *compute temp command* variant, with style name temp/kk. A variant style can be specified explicitly in your input script, e.g. compute temp/kk. If the suffix command is used with the appropriate style, you do not need to modify your input script. The specified suffix (kk) is automatically appended whenever your input script command creates a new *fix command*, *compute command*, etc. If the variant version does not exist, the standard version is created.

For the KOKKOS package, using this command-line switch also invokes the default KOKKOS settings, as if the command "package kokkos" were used at the top of your input script. These settings can be changed by using the "-package kokkos" command-line switch or the *package command* in your script.

The *suffix command* can also be used within an input script to set a suffix, or to turn off or back 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 Section *Parsing rules* for more info on using variables in input scripts.

Important: Currently, the command-line parser looks for arguments that start with "-" to indicate new switches. Thus you cannot specify multiple variable values if any of they start with a "-", e.g. a negative numeric value. It is OK if the first value1 starts with a "-", since it is automatically skipped.

1.2.7 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 state of the system. During the run itself, statistical information is printed periodically, every few timesteps. When the run concludes, SPARTA prints the final state and a total run time for the simulation. It then appends statistics about the CPU time and size of information stored for the simulation. An example set of statistics is shown here:

- The first line gives the total CPU run time for the simulation, in seconds.

```
Loop time of 0.639973 on 4 procs for 1000 steps with 45792 particles
```

- The next section gives a breakdown of the CPU timing (in seconds) in 7 categories. The first four are timings for particles moves, which includes interaction with surface elements, then particle collisions, then sorting of particles (required to perform collisions), and communication of particles between processors. The Modify section is time for operations invoked by fixes and computes. The Output section is for dump command and statistical output. The Other category is typically for load-imbalance, as some MPI tasks wait for others MPI tasks to complete. In each category the min,ave,max time across processors is shown, as well as a variation, and the percentage of total time.

```
MPI task timing breakdown:
Section |  min time  |  avg time  |  max time  |%varavg| %total
---------------------------------------------------------------
Move    | 0.10948    | 0.26191    | 0.42049    |  27.6 | 40.92
Coll    | 0.013711   | 0.041659   | 0.070985   |  13.5 |  6.51
Sort    | 0.01733    | 0.040286   | 0.063573   |  10.6 |  6.29
Comm    | 0.02276    | 0.023555   | 0.02493    |   0.6 |  3.68
Modify  | 0.00018167 | 0.024758   | 0.051345   |  15.6 |  3.87
Output  | 0.0002172  | 0.0007354  | 0.0012152  |   0.0 |  0.11
Other   |            | 0.2471     |            |       | 38.61
```

- The next section gives some statistics about the run. These are total counts of particle moves, grid cells touched by particles, the number of particles communicated between processors, collisions of particles with the global boundary and with surface elements (none in this problem), as well as collision and reaction statistics.

```
Particle moves    = 38096354 (38.1M)
Cells touched     = 43236871 (43.2M)
Particle comms    = 146623 (0.147M)
Boundary collides = 182782 (0.183M)
Boundary exits    = 181792 (0.182M)
SurfColl checks   = 7670863 (7.67M)
SurfColl occurs   = 177740 (0.178M)
Surf reactions    = 124169 (0.124M)
Collide attempts  = 1232 (1K)
Collide occurs    = 553 (0.553K)
Gas reactions     = 23 (0.023K)
Particles stuck   = 0
```

- The next section gives additional statistics, normalized by timestep or processor count.

```
Particle-moves/CPUsec/proc: 1.4882e+07
Particle-moves/step: 38096.4
Cell-touches/particle/step: 1.13493
Particle comm iterations/step: 1.999
Particle fraction communicated: 0.00384874
Particle fraction colliding with boundary: 0.00479789
Particle fraction exiting boundary: 0.0047719
Surface-checks/particle/step: 0.201354
Surface-collisions/particle/step: 0.00466554
Surface-reactions/particle/step: 0.00325934
Collision-attempts/particle/step: 1.232
Collisions/particle/step: 0.553
Gas-reactions/particle/step: 0.023
```

- The next 2 sections are optional. The "Gas reaction tallies" section is only output if the *react command* is used. For each reaction with a non-zero tally, the number of those reactions that occurred during the run is printed. The "Surface reaction tallies" section is only output if the *surf_react command* was used one or more times, to assign reaction models to individual surface elements or the box boundaries. For each of the commands, and each of its reactions with a non-zero tally, the number of those reactions that occurred during the run is printed. Note that this is effectively a summation over all the surface elements and/or box boundaries the *surf_react command* was used to assign a reaction model to.

```
Gas reaction tallies: style tce #-of-reactions 45 \
reaction O2 + N --> O + O + N: 10 \
reaction O2 + O --> O + O + O: 5 \
reaction N2 + O --> N + N + O: 8

Surface reaction tallies: id 1 style global #-of-reactions 2 \
reaction all: 124025 \
reaction delete: 53525 \
reaction create: 70500
```

- The last section is a histogramming across processors of various per-processor statistics: particle count, owned grid cells, processor, ghost grid cells which are copies of cells owned by other processors, and empty cells which are ghost cells without surface information (only used to pass particles to neighboring processors). The ave value is the average across all processors. The max and min values are for any processor. The 10-bin histogram shows the distribution of the value across processors. The total number of histogram counts is equal to the number of processors.

```
Particles: 11448 ave 17655 max 5306 min
Histogram: 2 0 0 0 0 0 0 0 0 2
Cells:     100 ave 100 max 100 min
Histogram: 4 0 0 0 0 0 0 0 0 0
GhostCell: 21 ave 21 max 21 min
Histogram: 4 0 0 0 0 0 0 0 0 0
EmptyCell: 21 ave 21 max 21 min
Histogram: 4 0 0 0 0 0 0 0 0 0
Surfs:     50 ave 50 max 50 min
Histogram: 4 0 0 0 0 0 0 0 0 0
GhostSurf: 0 ave 0 max 0 min
Histogram: 4 0 0 0 0 0 0 0 0 0
```

1.3 Commands

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

Contents

1.3.1 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 secs) is used for two simulations of 100 timesteps each. In the 2nd case, the default timestep (1.0 sec 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 define the grid overlaying the simulation box until the box itself has been defined. Likewise you cannot read in triangulated surfaces until a grid has been defined to store them.

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

1.3.2 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".

   How the variable is converted to a text string depends on what style of variable it is; see the *variable command* doc page for details. It can be a variable that stores multiple text strings, and return one of them. The returned text string can be multiple "words" (space separated) which will then be interpreted as multiple arguments in the input command. The variable can also store a numeric formula which will be evaluated and its numeric result returned as a string.

   As a special case, if the $ is followed by parenthesis, then the text inside the parenthesis is treated as an "immediate" variable and evaluated as an *equal-style variable*. This is a way to use numeric formulas in an input script without having to assign them to variable names. For example, these 3 input script lines:

```
variable X equal (xlo+xhi)/2+sqrt(v_area)
region 1 block $X 2 INF INF EDGE EDGE
variable X delete
```

   can be replaced by

```
region 1 block $((xlo+xhi)/2+sqrt(v_area)) 2 INF INF EDGE EDGE
```

   so that you do not have to define (or discard) a temporary variable X.

   Note that neither the curly-bracket or immediate form of variables can contain nested $ characters for other variables to substitute for. Thus you cannot do this:

```
variable        a equal 2
variable        b2 equal 4
print           "B2 = ${b$a}"
```

   Nor can you specify this `$($x-1.0)` for an immediate variable, but you could use `$(v_x-1.0)`, since the latter is valid syntax for an *equal-style variable*.

   See the *command-variable* for more details of how strings are assigned to variables and evaluated, and how they can be used 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. A long single argument enclosed in quotes can even span multiple lines if the "&" character is used, as described above. E.g.

```
print "Volume = $v"
print 'Volume = $v'
variable a string "red green blue &
                   purple orange cyan"
if "$steps > 1000" then quit
```

   The quotes are removed when the single argument is stored internally.

   See the *dump modify format* or *print command*, or *if command* 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: If the argument is itself a command that requires a quoted argument (e.g. using a *print command* as part of an *if command* 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 level of nesting is allowed, but that should be sufficient for most use cases.

1.3.3 Input script structure

This section describes the structure of a typical SPARTA input script. The "examples" directory in the SPARTA distribution contains sample input scripts; the corresponding problems are discussed in Section *Example problems*, 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 the simulation domain, particles, grid cells, and surfaces are defined.

   Relevant commands include *dimension command units command*, and *seed command*.

2. Problem definition

   These items must be defined before running a SPARTA calculation, and typically in this order:

   - *create_box* for the simulation box
   - *create_grid* or *read_grid* for grid cells
   - *read_surf* or *read_isurf* for surfaces
   - *species* for particle species properties
   - *create_particles* for particles

The first two are required. Surfaces are optional. Particles are also optional in the setup stage, since they can be added as the simulation runs.

The system can also be load-balanced after the grid and/or particles are defined in the setup stage using the *balance_grid command*. The grid can also be adapted before or betwee simulations using the *adapt_grid command*.

3. Settings

   Once the problem geometry, grid cells, surfaces, and particles are defined, a variety of settings can be specified, which include simulation parameters, output options, etc.

   Commands that do this include

   global, *timestep*, *collide* for a collision model, *react* for a chemisty model, *fix* for boundary conditions, time-averaging, load-balancing, etc. *compute* for diagnostic computations *stats_style* for screen output *dump* for snapshots of particle, grid, and surface info *dump image* for on-the-fly images of the simulation

4. Run a simulation

   A simulation is run using the *run command*.

1.3.4 Commands listed by category

This section lists many SPARTA commands, grouped by category. The *next section* lists all commands alphabetically.

Initialization: *dimension*, *package*, *seed*, *suffix*, *units*

Problem definition: *boundary*, *bound_modify*, *create_box*, *create_grid*, *create_particles*, *mixture*, *read_grid*, *read_isurf*, *read_particles*, *read_surf*, *read_restart*, *species*

Settings: *collide*, *collide_modify*, *compute*, *fix*, *global*, *react*, *react_modify*, *region*, *surf_collide*, *surf_modify*, *surf_react*, *timestep*, *uncompute*, *unfix*

Output: *dump*, *dump_image*, *dump_modify*, *restart*, *stats*, *stats_modify*, *stats_style*, *undump*, *write_grid*, *write_isurf*, *write_surf*, *write_restart*

Actions: *adapt_grid*, *balance_grid*, *run*, *scale_particles*

Miscellaneous: *clear*, *echo*, *if*, *include*, *jump*, *label*, *log*, *next*, *partition*, *print*, *quit*, *shell*, *variable*

1.3.5 Individual commands

This section lists all SPARTA commands alphabetically, with a separate listing below of styles within certain commands. The *previous section* lists many of the same commands, grouped by category.

adapt_grid	*balance_grid*	*boundary*	*bound_modify*	*clear*	*collide*
collide_modify	*compute*	*create_box*	*create_grid*	*create_particles*	*dimension*
dump	*dump image*	*dump_modify*	*dump movie*	*echo*	*fix*
global	*group*	*if*	*include*	*jump*	*label*
log	*mixture*	*move_surf*	*next*	*package*	*partition*
print	*quit*	*react*	*react_modify*	*read_grid*	*read_isurf*
read_particles	*read_restart*	*read_surf*	*region*	*remove_surf*	*reset_timestep*
restart	*run*	*scale_particles*	*seed*	*shell*	*species*
stats	*stats_modify*	*stats_style*	*suffix*	*surf_collide*	*surf_react*
surf_modify	*timestep*	*uncompute*	*undump*	*unfix*	*units*
variable	*write_grid*	*write_isurf*	*write_restart*	*write_surf*	

Fix styles

See the *fix command* for one-line descriptions of each style or click on the style itself for a full description. Some of the styles have accelerated versions, which can be used if SPARTA is built with the *appropriate accelerated package*. This is indicated by additional letters in parenthesis: k = KOKKOS.

ablate	*adapt (k)*	*ambipolar*	*ave/grid (k)*	ave/histo (k)	ave/histo/weight (k)
ave/surf	*ave/time*	*balance (k)*	*emit/face (k)*	*emit/face/file*	*emit/surf*
grid/check (k)	*move/surf (k)*	*print*	*vibmode*		

Compute styles

See the *compute command* for one-line descriptions of each style or click on the style itself for a full description. Some of the styles have accelerated versions, which can be used if SPARTA is built with the *appropriate accelerated package*. This is indicated by additional letters in parenthesis: k = KOKKOS.

boundary (k)	*count (k)*	*distsurf/grid (k)*	*eflux/grid (k)*	*fft/grid*	*grid (k)*
isurf/grid	*ke/particle (k)*	*lambda/grid (k)*	*pflux/grid (k)*	*property/grid (k)*	*react/boundary*
react/surf	*react/isurf/grid*	*reduce*	*sonine/grid (k)*	*surf (k)*	*thermal/grid (k)*
temp (k)	*tvib/grid*				

Collide styles

See the *collide command* for details of each style. Some of the styles have accelerated versions, which can be used if SPARTA is built with the *appropriate accelerated package*. This is indicated by additional letters in parenthesis: k = KOKKOS.

vss (k)

Surface collide styles

See the *surf_collide command* for details of each style. Some of the styles have accelerated versions, which can be used if SPARTA is built with the *appropriate accelerated package*. This is indicated by additional letters in parenthesis: k = KOKKOS.

cll	*diffuse (k)*	*impulsive*
piston (k)	*specular (k)*	*td*
vanish (k)		

Surface reaction styles

See the *surf_react command* for details of each style.

global	*prob*

1.4 Packages

This section gives an overview of the optional packages that extend SPARTA functionality with instructions on how to build SPARTA with each of them. Packages are groups of files that enable a specific set of features. For example, the KOKKOS package provides styles that can run on different hardware such as GPUs. You can see the list of all packages and "make" commands to manage them by typing "make package" from within the src directory of the SPARTA distribution or "cmake -DSPARTA_LIST_PKGS" from within a build directory. *Getting Started* gives general info on how to install and un-install packages as part of the SPARTA build process.

Packages may require some additional code compiled located in the lib folder, or may require an external library to be downloaded, compiled, installed, and SPARTA configured to know about its location and additional compiler flags.

Following the next two tables is a sub-section for each package. It lists authors (if applicable) and summarizes the package contents. It has specific instructions on how to install the package, including (if necessary) downloading or building any extra library it requires. It also gives links to documentation, example scripts, and pictures/movies (if available) that illustrate use of the package.

NOTE: To see the complete list of commands a package adds to SPARTA, just look at the files in its src directory, e.g. "ls src/KOKKOS". Files with names that start with fix, compute, etc correspond to commands with the same style names.

In these two tables, the "Example" column is a sub-directory in the examples directory of the distribution which has an input script that uses the package. E.g. "fft" refers to the examples/fft directory; The "Library" column indicates whether an extra library is needed to build and use the package:

- dash = no library
- sys = system library: you likely have it on your machine
- int = internal library: provided with SPARTA, but you may need to build it
- ext = external library: you will need to download and install it on your machine

Table 1: SPARTA packages

Package	Description	Doc page	Example	Library
package-fft	fast Fourier transforms	*compute_style compute/fft/grid*	fft	int or ext
package-kokkos	Kokkos-enabled styles	Section 5.3.3	Benchmarks	•

1.4.1 FFT package

Contents:

Apply Fast Fourier Transforms (FFTs) to simulation data. The FFT library is specified in the Makefile.machine using the FFT_INC, FFT_PATH, and FFT_LIB variables. Supported external FFT libraries that can be specified include FFTW2, FFTW3, and MKL.If no FFT library is specified in the Makefile, SPARTA will use the internal KISS FFT library that is included with SPARTA. See the see discussion in doc/Section_start.html#2_2 (step 6).

Install or un-install with make:

```
make yes-fft
make machine
```

```
make no-fft
make machine
```

Install or un-install with CMake:

```
cd build
cmake -C /path/to/sparta/cmake/presets/machine.cmake -DPKG_FFT=ON /path/to/sparta/
↪cmake
make
```

```
cmake -C /path/to/sparta/cmake/presets/machine.cmake -DPKG_FFT=OFF /path/to/sparta/
↪cmake
make
```

Supporting info:

- *compute fft/grid*
- examples/fft

1.4.2 KOKKOS package

Contents:

Styles adapted to compile using the Kokkos library which can convert them to OpenMP or CUDA code so that they run efficiently on multicore CPUs, KNLs, or GPUs. All the styles have a "kk" as a suffix in their style name. Section accelerate-kokkos gives details of what hardware and software is required on your system, and how to build and use this package. Its styles can be invoked at run time via the "-sf kk" or "-suffix kk" *Command-line options*.

You must have a C++11 compatible compiler to use this package.

Authors: The KOKKOS package was created primarily by Stan Moore (Sandia), with contributions from other folks as well. It uses the open-source Kokkos library which was developed by Carter Edwards, Christian Trott, and others at Sandia, and which is included in the SPARTA distribution in lib/kokkos.

Install or un-install:

For the KOKKOS package, you have 3 choices when building. You can build with either CPU or KNL or GPU support. Each choice requires additional settings in your Makefile.machine or machine.cmake file for the KOKKOS_DEVICES and KOKKOS_ARCH settings. See the src/MAKE/OPTIONS/Makefile.kokkos* or cmake/presets/*kokkos*.cmake files for examples. For CMake, it's best to start by copying cmake/presets/kokkos_cuda.cmake to cmake/presets/machine.cmake.

For multicore CPUs using OpenMP:

Using Makefiles:

```
KOKKOS_DEVICES = OpenMP
KOKKOS_ARCH = HSW            # HSW = Haswell, SNB = SandyBridge, BDW = Broadwell, etc
```

Using CMake:

```
-DKokkos_ENABLE_OPENMP=ON
-DKokkos_ARCH_HSW=ON
```

For Intel KNLs using OpenMP:

Using Makefiles:

```
KOKKOS_DEVICES = OpenMP
KOKKOS_ARCH = KNL
```

For NVIDIA GPUs using CUDA:

```
KOKKOS_DEVICES = Cuda
KOKKOS_ARCH = PASCAL60,POWER8      # P100 hosted by an IBM Power8, etc
KOKKOS_ARCH = KEPLER37,POWER8      # K80 hosted by an IBM Power8, etc
```

Using CMake:

```
-DKokkos_ENABLE_CUDA=ON
-DKokkos_ARCH_PASCAL60=ON -DKokkos_ARCH_POWER8=ON :pre
```

For make with GPUs, the following 2 lines define a nvcc wrapper compiler, which will use nvcc for compiling CUDA files or use a C++ compiler for non-Kokkos, non-CUDA files.

```
KOKKOS_ABSOLUTE_PATH = $(shell cd $(KOKKOS_PATH); pwd)
export OMPI_CXX = $(KOKKOS_ABSOLUTE_PATH)/bin/nvcc_wrapper
CC =        mpicxx
```

For CMake, copy cmake/presets/kokkos_cuda.cmake so OMPI_CXX and CC are set properly.

Once you have an appropriate Makefile.machine or machine.cmake, you can install/un-install the package and build SPARTA in the usual manner. Note that you cannot build one executable to run on multiple hardware targets (CPU or KNL or GPU). You need to build SPARTA once for each hardware target, to produce a separate executable.

Using make:

```
make yes-kokkos
make machine
```

```
make no-kokkos
make machine
```

Using CMake:

```
cmake -C /path/to/sparta/cmake/presets/machine.cmake /path/to/sparta/cmake
make
```

```
cmake -C /path/to/sparta/cmake/presets/machine.cmake -DPKG_KOKKOS=OFF /path/to/sparta/
↪cmake
make
```

Supporting info:

- src/KOKKOS: filenames -> commands
- src/KOKKOS/README
- lib/kokkos/README
- the *Accelerating SPARTA <accelerate>* section
- *Section 5.3.3 <accelerate-kokkos>*
- *Section 2.6 -k on . . . <start-command-line-options>*
- *Section 2.6 -sf kk <start-command-line-options>*
- *Section 2.6 -pk kokkos <start-command-line-options>*
- *package kokkos <command-package>*
- Benchmarks page of web site

1.5 Accelerating SPARTA performance

This section describes various methods for improving SPARTA performance for different classes of problems running on different kinds of machines.

Currently the only option is to use the KOKKOS accelerator packages provided with SPARTA that contains code optimized for certain kinds of hardware, including multi-core CPUs, GPUs, and Intel Xeon Phi coprocessors.

- 5.1 *Measuring performance*
- 5.2 *Accelerator packages with optimized styles*
- 5.2.1 KOKKOS package

The Benchmark page of the SPARTA web site gives performance results for the various accelerator packages discussed in Section 5.2, for several of the standard SPARTA benchmark problems, as a function of problem size and number of compute nodes, on different hardware platforms.

1.5.1 Measuring performance

Before trying to make your simulation run faster, you should understand how it currently performs and where the bottlenecks are.

The best way to do this is run the your system (actual number of particles) for a modest number of timesteps (say 100 steps) on several different processor counts, including a single processor if possible. Do this for an equilibrium version of your system, so that the 100-step timings are representative of a much longer run. There is typically no need to run for 1000s of timesteps to get accurate timings; you can simply extrapolate from short runs.

For the set of runs, look at the timing data printed to the screen and log file at the end of each SPARTA run. *This section* of the manual has an overview.

Running on one (or a few processors) should give a good estimate of the serial performance and what portions of the timestep are taking the most time. Running the same problem on a few different processor counts should give an estimate of parallel scalability. I.e. if the simulation runs 16x faster on 16 processors, its 100% parallel efficient; if it runs 8x faster on 16 processors, it's 50% efficient.

The most important data to look at in the timing info is the timing breakdown and relative percentages. For example, trying different options for speeding up the FFTs will have little impact if they only consume 10% of the run time. If the collide time is dominating, you may want to look at the KOKKOS package, as discussed below. Comparing how the percentages change as you increase the processor count gives you a sense of how different operations within the timestep are scaling.

Another important detail in the timing info are the histograms of particles counts and neighbor counts. If these vary widely across processors, you have a load-imbalance issue. This often results in inaccurate relative timing data, because processors have to wait when communication occurs for other processors to catch up. Thus the reported times for "Communication" or "Other" may be higher than they really are, due to load-imbalance. If this is an issue, you can uncomment the MPI_Barrier() lines in src/timer.cpp, and recompile SPARTA, to obtain synchronized timings.

1.5.2 Packages with optimized styles

Accelerated versions of various *collide_style*, *fixes*, *computes*, and other commands have been added to SPARTA via the KOKKOS package, which may run faster than the standard non-accelerated versions.

All of these commands are in the KOKKOS package provided with SPARTA. An overview of packages is give in Section *Packages*

SPARTA currently has acceleration support for three kinds of hardware, via the KOKKOS package: Many-core CPUs, NVIDIA GPUs, and Intel Xeon Phi.

Whether you will see speedup for your hardware may depend on the size problem you are running and what commands (accelerated and non-accelerated) are invoked by your input script. While these doc pages include performance guidelines, there is no substitute for trying out the KOKKOS package.

Any accelerated style has the same name as the corresponding standard style, except that a suffix is appended. Otherwise, the syntax for the command that uses the style is identical, their functionality is the same, and the numerical results it produces should also be the same, except for precision and round-off effects, and differences in random numbers.

For example, the KOKKOS package provides an accelerated variant of the Temperature Compute *compute temp command*, namely *compute temp/kk*

To see what accelerate styles are currently available, see Section *Individual commands* of the manual. The doc pages for individual commands (e.g. *compute temp command*) also list any accelerated variants available for that style.

To use an accelerator package in SPARTA, and one or more of the styles it provides, follow these general steps:

Action	Steps
Using make:	
install the accelerator package	`make yes-fft, make yes-kokkos, etc`
add compile/link flags to Makefile.machine in src/MAKE	KOKKOS_ARCH=PASCAL60
re-build SPARTA	`make kokkos_cuda`
or using CMake from a build directory:	
install the accelerator package	`cmake -DPKG_FFT=ON -DPKG_KOKKOS=ON`, etc
add compile/link flags	`cmake -C /path/to/sparta/cmake/presets/kokkos_cuda.cmake -DKokkos_ARCH_PASCAL60=ON`
re-build SPARTA	`make`
Then do the following:	
prepare and test a regular SPARTA simulation	`lmp_kokkos_cuda -in in.script; mpirun -np 32 lmp_kokkos_cuda -in in.script`
enable specific accelerator support via '-k on' *command-line switch*	k on g 1
set any needed options for the package via "-pk" *command-line switch* or *package command*	only if defaults need to be changed, -pk kokkos reduction atomic
use accelerated styles in your input via "-sf" *command-line switch* or *suffix command*	`lmp_kokkos_cuda -in in.script -sf kk`

Note that the first 3 steps can be done as a single command with suitable make command invocations. This is discussed in *Packages* of the manual, and its use is illustrated in the individual accelerator sections. Typically these steps only need to be done once, to create an executable that uses one or more accelerator packages.

The last 4 steps can all be done from the command-line when SPARTA is launched, without changing your input script, as illustrated in the individual accelerator sections. Or you can add *package command* and *suffix command* to your input script.

The Benchmark page of the SPARTA web site gives performance results for the various accelerator packages for several of the standard SPARTA benchmark problems, as a function of problem size and number of compute nodes, on different hardware platforms.

Here is a brief summary of what the KOKKOS package provides.

Styles with a "kk" suffix are part of the KOKKOS package, and can be run using OpenMP on multicore CPUs, on an NVIDIA GPU, or on an Intel Xeon Phi in "native" mode. The speed-up depends on a variety of factors, as discussed on the KOKKOS accelerator page.

The KOKKOS accelerator package doc page explains:

- what hardware and software the accelerated package requires
- how to build SPARTA with the accelerated package
- how to run with the accelerated package either via command-line switches or modifying the input script
- speed-ups to expect
- guidelines for best performance
- restrictions

1.6 How-to discussions

The following sections describe how to perform common tasks using SPARTA, as well as provide some techinical details about how SPARTA works.

The example input scripts included in the SPARTA distribution and highlighted in *Example problems* of the manual also show how to setup and run various kinds of simulations.

1.6.1 2d simulations

In SPARTA, as in other DSMC codes, a 2d simulation means that particles move only in the xy plane, but still have all 3 xyz components of velocity. Only the xy components of velocity are used to advect the particles, so that they stay in the xy plane, but all 3 components are used to compute collision parameters, temperatures, etc. Here are the steps to take in an input script to setup a 2d model.

- Use the *dimension command* to specify a 2d simulation.
- Make the simulation box periodic in z via the *boundary command*. This is the default.
- Using the *create_box command*, set the z boundaries of the box to values that straddle the `z = 0.0` plane. I.e. `zlo < 0.0 and zhi > 0.0`. Typical values are -0.5 and 0.5, but regardless of the actual values, SPARTA computes the "volume" of 2d grid cells as if their z-dimension length is 1.0, in whatever *units* are defined. This volume is used with the *global nrho* setting to calculate numbers of particles to create or insert. It is also used to compute collision frequencies.
- If surfaces are defined via the *read_surf command*, use 2d objects defined by line segements.

Many of the example input scripts included in the SPARTA distribution are for 2d models.

1.6.2 Axisymmetric simulations

In SPARTA, an axi-symmetric model is a 2d model. An example input script is provided in the examples/axisymm directory.

An axi-symmetric problem can be setup using the following commands:

- Set `dimension = 2` via the *dimension command*.
- Set the y-dimension lower boundary to "a" via the *boundary command*.
- The y-dimension upper boundary can be anything except "a" or "p" for periodic.
- Use the *create_box command* to define a 2d simulation box with `ylo = 0.0`.

If desired, grid cell weighting can be enabled via the *global weight* command. The *volume* or *radial* setting can be used for axi-symmetric models.

Grid cell weighting affects how many particles per grid cell are created when using the *create_particles command* and *fix emit/face command* variants.

During a run, it also triggers particle cloning and destruction as particles move from grid cell to grid cell. This can be important for inducing every grid cell to contain roughly the same number of particles, even if cells are of varying volume, as they often are in axi-symmetric models. Note that the effective volume of an axi-symmetric grid cell is the volume its 2d area sweeps out when rotated around the y=0 axis of symmetry.

1.6.3 Running multiple simulations from one input script

This can be done in several ways. See the documentation for individual commands for more details on how these examples work.

If "multiple simulations" means continue a previous simulation for more timesteps, then you simply use the *run command* multiple times. For example, this script

```
read_grid data.grid
create_particles 1000000
run 10000
run 10000
run 10000
run 10000
run 10000
```

would run 5 successive simulations of the same system for a total of 50,000 timesteps.

If you wish to run totally different simulations, one after the other, the *clear command* can be used in between them to re-initialize SPARTA. For example, this script

```
read_grid data.grid
create_particles 1000000
run 10000
clear
read_grid data.grid2
create_particles 500000
run 10000
```

would run 2 independent simulations, one after the other.

For large numbers of independent simulations, you can use *variable command*, and *next command*, and *jump command* to loop over the same input script multiple times with different settings. For example, this script, named `in.flow`

```
variable d index run1 run2 run3 run4 run5 run6 run7 run8
shell cd $d
read_grid data.grid
create_particles 1000000
run 10000
shell cd ..
clear
next d
jump in.flow
```

would run 8 simulations in different directories, using a data.grid file in each directory. The same concept could be used to run the same system at 8 different gas densities, using a density variable and storing the output in different log and dump files, for example

```
variable a loop 8
variable rho index 1.0e18 4.0e18 1.0e19 4.0e19 1.0e20 4.0e20 1.0e21 4.0e21
log log.$a
read data.grid
global nrho ${rho}

# other commands ...

compute myGrid grid all all n temp
dump 1 grid all 1000 dump.$a id c_myGrid
run 100000
clear  # Restore all settings
next rho
next a
jump in.flow
```

All of the above examples work whether you are running on 1 or multiple processors, but assumed you are running SPARTA on a single partition of processors. SPARTA can be run on multiple partitions via the "-partition" command-line switch as described in *Command-line options* of the manual.

In the last 2 examples, if SPARTA were run on 3 partitions, the same scripts could be used if the "index" and "loop" variables were replaced with *universe*-style variables, as described in the *variable command*. Also, the `next rho` and `next a` commands would need to be replaced with a single `next a rho` command. With these modifications, the 8 simulations of each script would run on the 3 partitions one after the other until all were finished. Initially, 3 simulations would be started simultaneously, one on each partition. When one finished, that partition would then start the 4th simulation, and so forth, until all 8 were completed.

1.6.4 Output from SPARTA (stats, dumps, computes, fixes, variables)

There are four basic kinds of SPARTA output:

- *Statistical output*, which is a list of quantities printed every few timesteps to the screen and logfile.
- *Dump files*, which contain snapshots of particle, grid cell, or surface element quantities and are written at a specified frequency.
- Certain fixes can output user-specified quantities directly to files: *fix ave/time* for time averaging, and *fix print* for single-line output of *variables*. Fix print can also output to the screen.
- *Restart files*.

A simulation prints one set of statistical output and (optionally) restart files. It can generate any number of dump files and fix output files, depending on what *dump command* and *fix command* you specify.

As discussed below, SPARTA gives you a variety of ways to determine what quantities are computed and printed when the statistics, dump, or fix commands listed above perform output. Throughout this discussion, note that users can also add their own computes and fixes to SPARTA (see *Modifying & extending SPARTA*) which can generate values that can then be output with these commands.

The following sub-sections discuss different SPARTA commands related to output and the kind of data they operate on and produce:

Global/per-particle/per-grid/per-surf data

Various output-related commands work with four different styles of data: global, per particle, per grid, or per surf. A global datum is one or more system-wide values, e.g. the temperature of the system. A per particle datum is one or more values per partice, e.g. the kinetic energy of each particle. A per grid datum is one or more values per grid cell, e.g. the temperature of the particles in the grid cell. A per surf datum is one or more values per surface element, e.g. the count of particles that collided with the surface element.

Scalar/vector/array data

Global, per particle, per grid, and per surf datums can each come in three kinds: a single scalar value, a vector of values, or a 2d array of values. The doc page for a "compute" or "fix" or "variable" that generates data will specify both the style and kind of data it produces, e.g. a per grid vector.

When a quantity is accessed, as in many of the output commands discussed below, it can be referenced via the following bracket notation, where ID in this case is the ID of a compute. The leading `c_` would be replaced by `f_` for a fix, or `v_` for a variable:

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 data once (vector -> scalar, array -> vector). Using two brackets reduces the dimension twice (array -> scalar). Thus a command that uses scalar values as input can typically also process elements of a vector or array.

Statistical output

The frequency and format of statistical output is set by the *stats*, *stats_style command*, and *stats_modify command*. The *stats_style command* also specifies what values are calculated and written out. Pre-defined keywords can be specified (e.g. np, ncoll, etc). Three additional kinds of keywords can also be specified (c_ID, f_ID, v_name), where a *compute command* or *fix command* or *variable command* provides the value to be output. In each case, the compute, fix, or variable must generate global values to be used as an argument of the *stats_style command*.

Dump file output

Dump file output is specified by the *dump command* and *dump_modify command*. There are several pre-defined formats: dump particle, dump grid, dump surf, etc.

Each of these allows specification of what values are output with each particle, grid cell, or surface element. Pre-defined attributes can be specified (e.g. id, x, y, z for particles or id, vol for grid cells, etc). Three additional kinds of keywords can also be specified (c_ID, f_ID, v_name), where a *compute command* or *fix command* or *variable command* provides the values to be output. In each case, the compute, fix, or variable must generate per particle, per grid, or per surf values for input to the corresponding *dump command*.

Fixes that write output files

Two fixes take various quantities as input and can write output files: *fix ave/time* and *fix print*.

The *fix ave/time command* enables direct output to a file and/or time-averaging of global scalars or vectors. The user specifies one or more quantities as input. These can be global *compute* values, global *fix* values, or *variables* of any style except the particle style which does not produce single values. Since a variable can refer to keywords used by the *stats_style command* (like particle count), a wide variety of quantities can be time averaged and/or output in this way. If the inputs are one or more scalar values, then the fix generates a global scalar or vector of output. If the inputs are one or more vector values, then the fix generates a global vector or array of output. The time-averaged output of this fix can also be used as input to other output commands.

The *fix print command* can generate a line of output written to the screen and log file or to a separate file, periodically during a running simulation. The line can contain one or more *variable* values for any style variable except the particle style. As explained above, variables themselves can contain references to global values generated by *stats keywords*, *computes*, *fixes*, or other *variables*. Thus the *fix print command* is a means to output a wide variety of quantities separate from normal statistical or dump file output.

Computes that process output quantities

The *compute reduce command* takes one or more per particle or per grid or per surf vector quantities as inputs and "reduces" them (sum, min, max, ave) to scalar quantities. These are produced as output values which can be used as input to other output commands.

Computes that generate values to output

Every *compute* in SPARTA produces either global or per particle or per grid or per surf values. The values can be scalars or vectors or arrays of data. These values can be output using the other commands described in this section. The doc page for each compute command describes what it produces. Computes that produce per particle or per grid or per surf values have the word "particle" or "grid" or "surf" in their style name. Computes without those words produce global values.

Fixes that generate values to output

Some *fixes* in SPARTA produces either global or per particle or per grid or per surf values which can be accessed by other commands. The values can be scalars or vectors or arrays of data. These values can be output using the other commands described in this section. The doc page for each fix command tells whether it produces any output quantities and describes them.

Two fixes of particular interest for output are the *fix ave/grid command* and *fix ave/surf command*.

The *fix ave/grid command* enables time-averaging of per grid vectors. The user specifies one or more quantities as input. These can be per grid vectors or arrays from *compute command* or *fix command*. If the input is a single vector, then the fix generates a per grid vector. If the input is multiple vectors or array, the fix generates a per grid array. The time-averaged output of this fix can also be used as input to other output commands.

The *fix ave/surf command* enables time-averaging of per surf vectors. The user specifies one or more quantities as input. These can be per surf vectors or arrays from *compute command* or *fix command*. If the input is a single vector, then the fix generates a per surf vector. If the input is multiple vectors or array, the fix generates a per surf array. The time-averaged output of this fix can also be used as input to other output commands.

Variables that generate values to output

Variables defined in an input script generate either a global scalar value or a per particle vector (only particle-style variables) when it is accessed. The formulas used to define equal- and particle-style variables can contain references to the *stats_style* keywords and to global and per particle data generated by computes, fixes, and other variables. The values generated by variables can be output using the other commands described in this section.

Summary table of output options and data flow between commands

This table summarizes the various commands that can be used for generating output from SPARTA. Each command produces output data of some kind and/or writes data to a file. Most of the commands can take data from other commands as input. Thus you can link many of these commands together in pipeline form, where data produced by one command is used as input to another command and eventually written to the screen or to a file. Note that to hook two commands together the output and input data types must match, e.g. global/per atom/local data and scalar/vector/array data.

Also note that, as described above, when a command takes a scalar as input, that could be an element of a vector or array. Likewise a vector input could be a column of an array.

Command	Input	Output
stats_style	global scalars	screen, log file
dump particle	per particle vectors	dump file
dump grid	per grid vectors	dump file
dump surf	per surf vectors	dump file
fix print <command-fix-print>	global scalar from variable	screen, file
print	global scalar from variable	screen
computes	N/A	global or per particle/grid/surf scalar/vector/array
fixes	N/A	global or per particle/grid/surf scalar/vector/array
variables	global scalars, per particle vectors	global scalar, per particle vector
compute reduce	per particle/grid/surf vectors	global scalar/vector
fix ave/time	global scalars/vectors	global scalar/vector/array, file
fix ave/grid	per grid vectors/arrays	per grid vector/array
fix ave/surf	per surf vectors/arrays	per surf vector/array

1.6.5 Visualizing SPARTA snapshots

The *dump image command* can be used to do on-the-fly visualization as a simulation proceeds. It works by creating a series of JPG or PNG or PPM files on specified timesteps, as well as movies. The images can include particles, grid cell quantities, and/or surface element quantities. This is not a substitute for using an interactive visualization package in post-processing mode, but on-the-fly visualization can be useful for debugging or making a high-quality image of a particular snapshot of the simulation.

The *dump command* can be used to create snapshots of particle, grid cell, or surface element data as a simulation runs. These can be post-processed and read in to other visualization packages.

A Python-based toolkit distributed by our group can read SPARTA particle dump files with columns of user-specified particle information, and convert them to various formats or pipe them into visualization software directly. See the Pizza.py WWW site for details. Specifically, Pizza.py can convert SPARTA particle dump files into PDB, XYZ, Ensight, and VTK formats. Pizza.py can pipe SPARTA dump files directly into the Raster3d and RasMol visualization programs. Pizza.py has tools that do interactive 3d OpenGL visualization and one that creates SVG images of dump file snapshots.

Additional Pizza.py tools may be added that allow visualization of surface and grid cell information as output by SPARTA.

1.6.6 Library interface to SPARTA

As described in *build-library*, SPARTA can be built as a library, so that it can be called by another code, used in a *coupled manner* with other codes, or driven through a *Python interface*.

All of these methodologies use a C-style interface to SPARTA that is provided in the files src/library.cpp and src/library.h. The functions therein have a C-style argument list, but contain C++ code you could write yourself in a C++ application that was invoking SPARTA directly. The C++ code in the functions illustrates how to invoke internal SPARTA operations. Note that SPARTA classes are defined within a SPARTA namespace (SPARTA_NS) if you use them from another C++ application.

Library.cpp contains these 4 functions:

```
void sparta_open(int, char **, MPI_Comm, void **);
void sparta_close(void *);
void sparta_file(void *, char *);
char *sparta_command(void *, char *);
```

The sparta_open() function is used to initialize SPARTA, passing in a list of strings as if they were *Command-line options* when SPARTA is run in stand-alone mode from the command line, and a MPI communicator for SPARTA to run under. It returns a ptr to the SPARTA object that is created, and which is used in subsequent library calls. The sparta_open() function can be called multiple times, to create multiple instances of SPARTA.

SPARTA will run on the set of processors in the communicator. This means the calling code can run SPARTA on all or a subset of processors. For example, a wrapper script might decide to alternate between SPARTA and another code, allowing them both to run on all the processors. Or it might allocate half the processors to SPARTA and half to the other code and run both codes simultaneously before syncing them up periodically. Or it might instantiate multiple instances of SPARTA to perform different calculations.

The sparta_close() function is used to shut down an instance of SPARTA and free all its memory.

The sparta_file() and sparta_command() functions are used to pass a file or string to SPARTA as if it were an input script or single command in an input script. Thus the calling code can read or generate a series of SPARTA commands one line at a time and pass it thru the library interface to setup a problem and then run it, interleaving the sparta_command() calls with other calls to extract information from SPARTA, perform its own operations, or call another code's library.

Other useful functions are also included in library.cpp. For example:

```
void *sparta_extract_global(void *, char *);
void *sparta_extract_compute(void *, char *, int, int);
void *sparta_extract_variable(void *, char *, char *);
```

This can extract various global quantities from SPARTA as well as values calculated by a compute or variable. See the library.cpp file and its associated header file library.h for details.

Other functions may be added to the library interface as needed to allow reading from or writing to internal SPARTA data structures.

The key idea of the library interface is that you can write any functions you wish to define how your code talks to SPARTA and add them to src/library.cpp and src/library.h, as well as to the *Python interface*. The routines you add can in principle access or change any SPARTA data you wish. The examples/COUPLE and python directories have example C++ and C and Python codes which show how a driver code can link to SPARTA as a library, run SPARTA on a subset of processors, grab data from SPARTA, change it, and put it back into SPARTA.

Important: The examples/COUPLE dir has not been added to the distribution yet.

1.6.7 Coupling SPARTA to other codes

SPARTA is designed to allow it to be coupled to other codes. For example, a continuum finite element (FE) simulation might use SPARTA grid cell quantities as boundary conditions on FE nodal points, compute a FE solution, and return continuum flow conditions as boundary conditions for SPARTA to use.

SPARTA can be coupled to other codes in at least 3 ways. Each has advantages and disadvantages, which you'll have to think about in the context of your application.

(1) Define a new *fix command* that calls the other code. In this scenario, SPARTA is the driver code. During its timestepping, the fix is invoked, and can make library calls to the other code, which has been linked to SPARTA

as a library. See *Modifying & extending SPARTA* of the documentation for info on how to add a new fix to SPARTA.

(2) Define a new SPARTA command that calls the other code. This is conceptually similar to method (1), but in this case SPARTA and the other code are on a more equal footing. Note that now the other code is not called during the timestepping of a SPARTA run, but between runs. The SPARTA input script can be used to alternate SPARTA runs with calls to the other code, invoked via the new command. The *run command* facilitates this with its *every* option, which makes it easy to run a few steps, invoke the command, run a few steps, invoke the command, etc.

In this scenario, the other code can be called as a library, as in (1), or it could be a stand-alone code, invoked by a system() call made by the command (assuming your parallel machine allows one or more processors to start up another program). In the latter case the stand-alone code could communicate with SPARTA thru files that the command writes and reads.

See *Modifying & extending SPARTA* of the documentation for how to add a new command to SPARTA.

(3) Use SPARTA as a library called by another code. In this case the other code is the driver and calls SPARTA as needed. Or a wrapper code could link and call both SPARTA and another code as libraries. Again, the *run command* has options that allow it to be invoked with minimal overhead (no setup or clean-up) if you wish to do multiple short runs, driven by another program.

Examples of driver codes that call SPARTA as a library are included in the examples/COUPLE directory of the SPARTA distribution; see examples/COUPLE/README for more details.

Important: The examples/COUPLE dir has not been added to the distribution yet.

Section 2.3 of the manual describes how to build SPARTA as a library. Once this is done, you can interface with SPARTA either via C++, C, Fortran, or Python (or any other language that supports a vanilla C-like interface). For example, from C++ you could create one (or more) "instances" of SPARTA, pass it an input script to process, or execute individual commands, all by invoking the correct class methods in SPARTA. From C or Fortran you can make function calls to do the same things. See *Section_9* 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 *Section 6.6* of the manual for a description of the interface and how to extend it for your needs.

Note that the `sparta_open()` function that creates an instance of SPARTA takes an MPI communicator as an argument. This means that instance of SPARTA will run on the set of processors in the communicator. Thus the calling code can run SPARTA on all or a subset of processors. For example, a wrapper script might decide to alternate between SPARTA and another code, allowing them both to run on all the processors. Or it might allocate half the processors to SPARTA and half to the other code and run both codes simultaneously before syncing them up periodically. Or it might instantiate multiple instances of SPARTA to perform different calculations.

1.6.8 Details of grid geometry in SPARTA

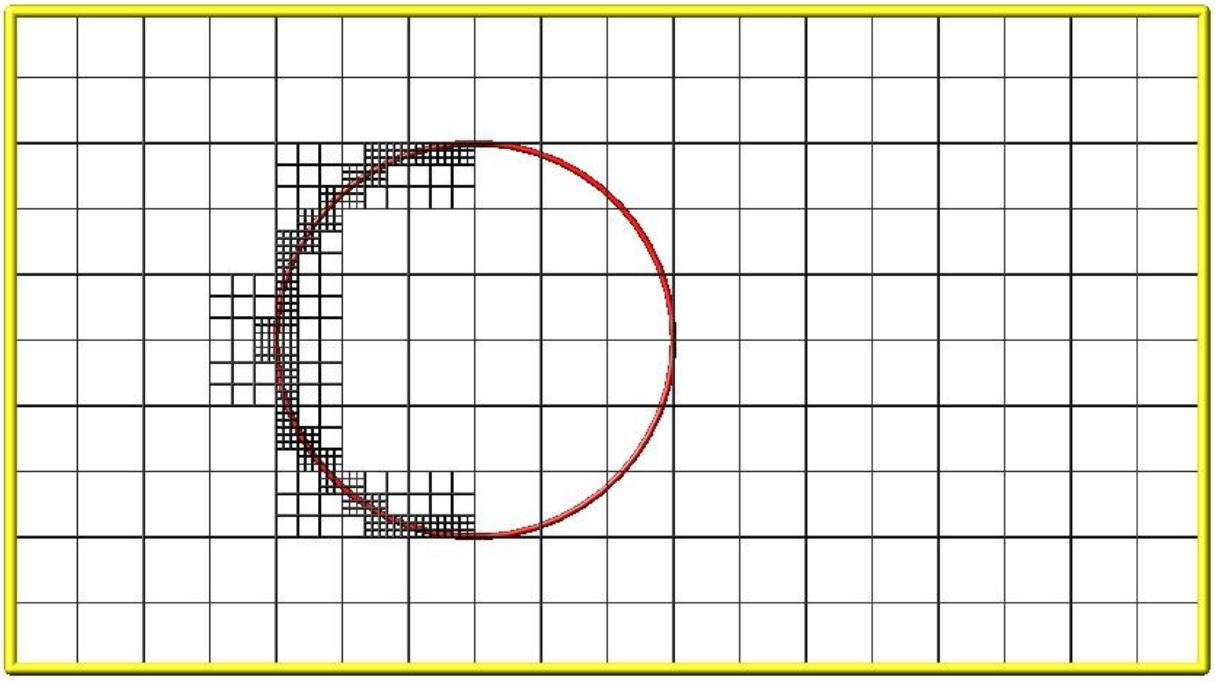
SPARTA overlays a grid over the simulation domain which is used to track particles and to co-locate particles in the same grid cell for performing collision and chemistry operations. Surface elements are also assigned to grid cells they intersect with, so that particle/surface collisions can be efficiently computed.

SPARTA uses a Cartesian hierarchical grid. Cartesian means that the faces of a grid cell, at any level of the hierarchy, are aligned with the Cartesian xyz axes. I.e. each grid cell is an axis-aligned pallelpiped or rectangular box. The hierarchy of grid cells is defined in the following manner. The entire simulation box is a single "root" grid cell at level 0 of the hierarchy. It is sub-divided into a regular Nx by Ny by Nz grid of cells, all at level 1 of the hierarchy. "Regular" means all the Nx*Ny*Nz sub-divided cells within a parent cell are the same size. Each Nx,Ny,Nz value >= 1 (although if Nx = Ny = Nz = 1 then obviously there is no sub-division). Any of the cells at level 1 can be further

sub-divided in the same manner to create cells at level 2, and recursively for levels 3, 4, etc. The Nx,Ny,Nz values for sub-dividing an individual parent cell can be uniquely chosen. All level 2 cells do not need to be sub-divided using the same Nx,Ny,Nz values. Grids for 2d and 3d simulations (see the *dimension*) follow the same rules, except that `Nz = 1` is required at every level of sub-division for 2d grids.

Note that this manner of defining a hierarchy allows for flexible grid cell refinement in any region of the simulation domain. E.g. around a surface, or in a high-density region of the gas flow. Also note that a 3d oct-tree (quad-tree in 2d) is a special case of the SPARTA hierarchical grid, where Nx = Ny = Nz = 2 is always used to sub-divide a grid cell.

An example 2d hierarchical grid is shown in the diagram, for a circular surface object (in red) with the grid refined on the upwind side of the object (flow from left to right). The first level coarse grid is 18x10. 2nd level grid cells are defined in a subset of those cells with a 3x3 sub-division. A subset of the 2nd level cells contain 3rd level grid cells via a further 3x3 sub-division.



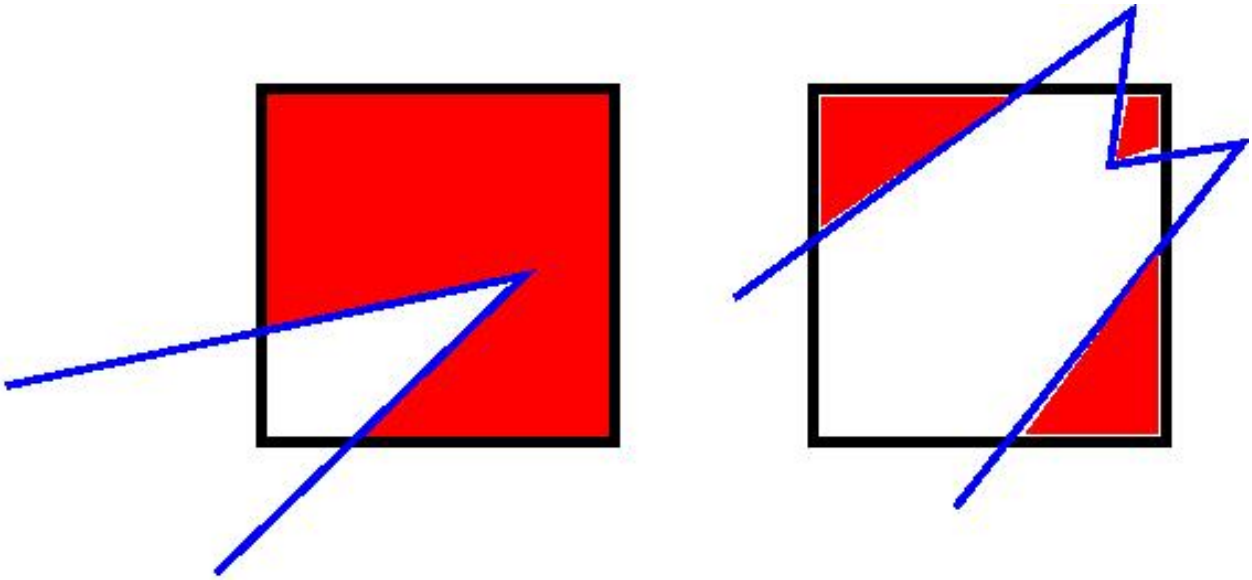
In the rest of the SPARTA manual, the following terminology is used to refer to the cells of the hierarchical grid. The flow region is the portion of the simulation domain that is "outside" any surface objects and is typically filled with particles.

- root cell = the simulation box itself
- parent cell = a grid cell at any level that is sub-divided further
- child cell = a grid cell that is not sub-divided further
- unsplit cell = a child cell not intersected by any surface elements
- cut cell = a child cell intersected by one or more surface elements, one resulting flow region
- split cell = a child cell intersected by two or more surface elemments, two or more resulting disjoint flow regions
- sub cell = one disjoint flow region portion of a split cell

The list of parent cells in a simulation is stored by every processor and is read in by the *read_grid command*, or defined by the *create_grid command*. Child cells are inferred by the same 2 commands and the union of all child cells is the

entire simulation domain. Child cells are distributed across processors, so that each child cell is owned by exactly one processor, as discussed below.

When surface objects are defined via the *read_surf command*, they intersect child cells. In this contex "intersection" by a surface element means a geometric overlap between the area of the surface element and the volume of the grid cell (or length of element and area of grid cell in 2d). Thus an intersection includes a surface triangle that only touches a grid cell on its face, edge, or at its corner point. When intersected by one or more surface elements, a child cell becomes one of 3 flavors: unsplit, cut, or split. A child cell not intersected by any surface elements is an unsplit cell. It can be entirely in the flow region or entirely inside a surface object. If a child cell is intersected so that it is partitioned into two contiguous volumes, one in the flow region, the other inside a surface object, then it is a cut cell. This is the usual case. Note that either the flow volume or inside volume can be of size zero, if the surface only "touches" the grid cell, i.e. the intersection is only on a face, edge, or corner point of the grid cell. The left side of the diagram below is an example, where red represents the flow region. Sometimes a child cell can be partitioned by surface elements so that more than one contiguous flow region is created. Then it is a split cell. Additionally, each of the two or more contiguous flow regions is a sub cell of the split cell. The right side of the diagram shows a split cell with 3 sub cells.



The union of (1) unsplit cells that are in the flow region (not entirely interior to a surface object) and (2) flow region portions of cut cells and (3) sub cells is the entire flow region of the simulation domain. These are the only kinds of child cells that store particles. Split cells and unsplit cells interior to surface objects have no particles.

Every parent and child cell is assigned an ID by SPARTA. These IDs can be output in integer or string form by the *dump command*, using its *id* and *idstr* attributes. The integer form can also be output by the *compute property/grid* <*command-compute-property-grid*>.

Here is how the grid cell ID is computed and stored by SPARTA. Say the 1st level grid is a 10x10x20 sub-division (2000 cells) of the root cell. The 1st level cells are numbered from 1 to 2000 with the x-dimension varying fastest, then y, and finally the z-dimension slowest. Now say the 374th (out of 2000, 14 in x, 19 in y, 1 in z) 1st-level cell has a 2x2x2 sub-division (8 cells), and consider the 4th 2nd-level cell (2 in x, 2 in y, 1 in z) within the 374th cell. It could be a parent cell if it is further sub-divided, or a child cell if not. In either case its ID is the same. The rightmost 11 bits of the integer ID are encoded with 374. This is because it requires 11 bits to represent 2000 cells (1 to 2000) at level 1. The next 4 bits are used to encode 1 to 8, specifically 4 in the case of this cell. Thus the cell ID in integer format is 4*2048 + 374 = 8566. In string format it will be printed as 4-374, with dashes separating the levels.

Note that a child cell has the same ID whether it is unsplit, cut, or split. Currently, sub cells of a split cell also have the same ID, though that may change in the future.

> The number of hierarchical levels a SPARTA grid can contain is limited to whether all cell IDs can be encoded in an integer. By default cell IDs are stored in 32-bit integers. 64-bit integers can be used if

SPARTA is compiled with the -DSPARTA_BIGBIG option, as explained in *Steps to build a SPARTA executable using make:*.

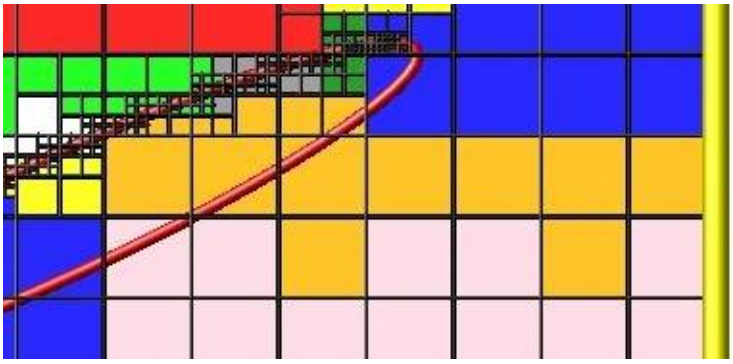
For 32-bit cell IDs if every level uses a 2x2x2 sub-division which requires 4 bits (to store values from 1 to 8), then a grid can have 7 levels. For 64-bit cell IDs, 15 levels could be defined.

The *create_grid command*, and *balance_grid command*, and *fix balance command - fix balance/kk command* determine the assignment of child cells to processors. If a child cell is assigned to a processor, that processor owns the cell whether it is an unsplit, cut, or split cell. It also owns any sub cells that are part of a split cell.

Depending on how they the commands are used, the child cells assigned to each processor will either be "clumped" or "dispersed".

Clumped means each processor's cells will be geometrically compact. Dispersed means the processor's cells will be geometrically dispersed across the simulation domain and so they cannot be enclosed in a small bounding box.

An example of a clumped assignment is shown in this zoom-in of a 2d hierarchical grid with 5 levels, refined around a tilted ellipsoidal surface object (outlined in pink). One processor owns the grid cells colored orange. A compact bounding rectangle can be drawn around the orange cells which will contain only a few grid cells owned by other processors. By contrast a dispersed assignment could scatter orange grid cells throughout the entire simulation domain.



It is important to understand the difference between the two kinds of assignments and the effects they can have on performance of a simulation. For example the create_grid and read_grid commands may produce dispersed assignments, depending on the options used, which can be converted to a clumped assignment by the balance_grid command.

Simulations typically run faster with clumped grid cell assignments. This is because the cost of communicating particles is reduced if particles that move to a neighboring grid cell often stay on-processor. Similarly, some stages of simulation setup may run faster with a clumped assignment. Examples are the finding of nearby ghost grid cells and the computation of surface element intersections with grid cells. The latter operation is invoked when the *read_surf command* is used.

If the spatial distribution of particles is highly irregular and/or dynamically changing, or if the computational work per grid cell is otherwise highly imbalanced, a clumped assignment of grid cells to processors may not lead to optimal balancing. In these scenarios a dispersed assignment of grid cells to processsors may run faster even with the overhead of increased particle communication. This is because randomly assigning grid cells to processors can balance the computational load in a statistical sense.

1.6.9 Details of surfaces in SPARTA

A SPARTA simulation can define one or more surface objects, each of which are read in via the *read_surf*. For 2d simulations a surface object is a collection of connected line segments. For 3d simulations it is a collection of connected triangles. The outward normal of lines or triangles, as defined in the surface file, points into the flow region of the simulation box which is typically filled with particles. Depending on the orientation, surface objects can thus be obstacles that particles flow around, or they can represent the outer boundary of an irregular shaped region which particles are inside of.

See the *read_surf command* doc page for a discussion of these topics:

- Requirement that a surface object be "watertight", so that particles do not enter inside the surface or escape it if used as an outer boundary.
- Surface objects (one per file) that contain more than one physical object, e.g. two or more spheres in a single file.
- Use of geometric transformations (translation, rotation, scaling, inversion) to convert the surface object in a file into different forms for use in different simulations.
- Clipping a surface object to the simulation box to effectively use a portion of the object in a simulation, e.g. a half sphere instead of a full sphere.
- The kinds of surface objects that are illegal, including infinitely thin objects, ones with duplicate points, or multiple surface or physical objects that touch or overlap.

The *read_surf command* assigns an ID to the surface object in a file. This can be used to reference the surface elements in the object in other commands. For example, every surface object must have a collision model assigned to it so that particle bounces off the surface can be computed. This is done via the *surf_modify command* and *surf_collide command*.

As described in the previous Section *Details of grid geometry in SPARTA*, SPARTA overlays a grid over the simulation domain to track particles. Surface elements are also assigned to grid cells they intersect with, so that particle/surface collisions can be efficiently computed. Typically a grid cell size larger than the surface elements that intersect it may not desirable since it means flow around the surface object will not be well resolved. The size of the smallest surface element in the system is printed when the surface file is read. Note that if the surface object is clipped to the simulation box, small lines or triangles can result near the box boundary due to the clipping operation.

The maximum number of surface elements that can intersect a single child grid cell is set by the *global surfmax* command. The default limit is 100. The actual maximum number in any grid cell is also printed when the surface file is read. Values this large or larger may cause particle moves to become expensive, since each time a particle moves within that grid cell, possible collisions with all its overlapping surface elements must be computed.

1.6.10 Restarting a simulation

There are two ways to continue a long SPARTA simulation. Multiple *run* commands can be used in the same input script. Each run will continue from where the previous run left off. Or binary restart files can be saved to disk using the *restart command*. At a later time, these binary files can be read via a *read_restart command* in a new script.

Here is an example of a script that reads a binary restart file and then issues a new run command to continue where the previous run left off. It illustrates what settings must be made in the new script. Details are discussed in the documentation for the *read_restart command* and *write_restart command*.

Look at the *in.collide* input script provided in the *bench* directory of the SPARTA distribution to see the original script that this script is based on. If that script had the line

```
restart            50 tmp.restart
```

added to it, it would produce 2 binary restart files (tmp.restart.50 and tmp.restart.100) as it ran for 130 steps, one at step 50, and one at step 100.

This script could be used to read the first restart file and re-run the last 80 timesteps:

```
read_restart   tmp.restart.50

seed           12345
collide        vss air ar.vss
```

(continues on next page)

(continued from previous page)

```
stats           10
compute         temp temp
stats_style     step cpu np nattempt ncoll c_temp

timestep         7.00E-9
run              80
```

Note that the following commands do not need to be repeated because their settings are included in the restart file: *dimension, global, boundary, create_box, create_grid, species, mixture*. However these commands do need to be used, since their settings are not in the restart file: *seed, collide, compute, fix, stats_style, timestep*. The *read_restart* doc page gives details.

If you actually use this script to perform a restarted run, you will notice that the statistics output does not match exactly. On step 50, the collision counts are 0 in the restarted run, because the line is printed before the restarted simulation begins. The collision counts in subsequent steps are similar but not identical. This is because new random numbers are used for collisions in the restarted run. This affects all the randomized operations in a simulation, so in general you should only expect a restarted run to be statistically similar to the original run.

1.6.11 Using the ambipolar approximation

The ambipolar approximation is a computationally efficient way to model low-density plasmas which contain positively-charged ions and negatively-charged electrons. In this model, electrons are not free particles which move independently. This would require a simulation with a very small timestep due to electon's small mass and high speed (1000x that of an ion or neutral particle).

Instead each ambipolar electron is assumed to stay "close" to its parent ion, so that the plasma gas appears macroscopically neutral. Each pair of particles thus moves together through the simulation domain, as if they were a single particle, which is how they are stored within SPARTA. This means a normal timestep can be used.

There are two stages during a timestep when the coupled particles are broken apart and treated as an independent ion and electron.

The first is during gas-phase collisions and chemistry. The ionized ambipolar particles in a grid cell are each split into two particles (ion and electron) and each can participate in two-body collisions with any other particle in the cell. Electron/electron collisions are actually not performed, but are tallied in the overall collision count (if using a collision mixture with a single group, not when using multiple groups). If gas-phase chemistry is turned on, reactions involving ions and electrons can be specified, which include dissociation, ionization, exchange, and recombination reactions. At the end of the collision/chemsitry operations for the grid cell, there is still a one-to-one pairing between ambipolar ions and electrons. Each pair is recombined into a single particle.

The second is during collisions with surface (or the boundaries of the simulation box) if a surface reaction model is defined for the surface element or boundary. Just as with gas-phase chemistry, surface reactions involving ambipolar species can be defined. For example, an ambipolar ion/electron pair can re-combine into a neutral species during the collision.

Here are the SPARTA commands you can use to run a simulation using the ambipolar approximation. See the input scripts in examples/ambi for an example.

Note that you will likely need to use two (or more mixtures) as arguments to various commands, one which includes the ambipolar electron species, and one which does not. Example *mixture command* for doing this are shown below.

Use the *fix ambipolar command* to specify which species is the ambipolar electron and what (multiple) species are ambipolar ions. This is required for all the other options listed here to work. The fix defines two custom per-particles attributes, an integer vector called "ionambi" which stores a 1 for a particle if it is an ambipolar ion, and a 0 otherwise. And a floating-point array called "velambi" which stores a 3-vector with the velocity of the associated electron for each

ambipolar ion or zeroes otherwise. Note that no particles should ever exist in the simulation with a species matching ambipolar electrons. Such particles are only generated (and destroyed) internally, as described above.

Use the *collide_modify ambipolar yes* command if you want to perform gas-phase collisions using the ambipolar model. This is not required. If you do this, you may also want to specify a mixture for the collide command which has two or more groups. If this is the case, the ambipolar electron species must be in a group by itself. The other group(s) can contain any combination of ion or neutral species. Note that putting the ambipolar electron species in its own group should improve the efficiency of the code due to the large disparity in electron versus ion/neutral velocities.

If you do this, DO use a mixture which includes the ambipolar electron species, so that electrons will participate in the collisions and reactions (if defined). You probably also want to specify a mixture for the collide command which has two or more groups. One group is for the ambipolar electron species, the other for ambipolar ions. Additional groups could exist for other species (e.g. neutrals), or those species could be part of the ion group. Putting the ambipolar electron species in its own group should improve the efficiency of the code due to the large disparity in electron versus ion/neutral velocities.

If you want to perform gas-phase chemistry for reactions involving ambipolar ions and electrons, use the *react command* with an input file of reactions that include the ambipolar electron and ion species defined by the fix ambipolar commmand. See the *react command* doc page for info the syntax required for ambipolar reactions. Their reactants and products must be listed in specific order.

When creating particles, either by the *create_particles command* or *fix emit/face command* variants, do NOT use a mixture that includes the ambipolar electron species. If you do this, you will create "free" electrons which are not coupled to an ambipolar ion. You can include ambipolar ions in the mixture. This will create ambipolar ions along with their associated electron. The electron will be assigned a velocity consistent with its mass and the temperature of the created particles. You can use the *mixture copy* and *mixture delete* commands to create a mixture that excludes only the ambipolar electron species, e.g.

```
mixture all copy noElectron
mixture noElectron delete e
```

If you want ambipolar ions to re-combine with their electrons when they collide with surfaces, use the *surf_react command* with an input file of surface reactions that includes recombination reactions like:

$$N^+ + e \rightarrow N$$

See the *surf_react command* doc page for syntax details. A sample surface reaction data file is provided in data/air.surf. You assign the surface reaction model to surface or the simulation box boundaries via the *surf_modify command* and *bound_modify command*.

For diagnositics and output, you can use the *compute count command* and *command-dump-particle*. The *compute count command* generate counts of individual species, entire mixtures, and groups within mixtures. For example these commands will include counts of ambipolar ions in statistical output:

```
compute myCount O+ N+ NO+ e
stats_style step nsreact nsreactave cpu np c_myCount
```

Note that the count for species "e" = ambipolar electrons should alwas be zero, since those particles only exist during gas and surface collisions. The *stats_style nsreact* and *nsreactave* keywords print tallies of surface reactions taking place.

The *dump particle* command can output the custom particle attributes defined by the *fix ambipolar command*. E.g. this command

```
dump 1 particle 1000 tmp.dump id type x y z p_ionambi p_velambi[2]
```

will output the ionambi flag = 1 for ambipolar ions, along with the vy of their associated ambipolar electrons.

The *read_restart* doc page explains how to restart ambipolar simulations where a fix like *fix ambipolar* has been used to store extra per-particle properties.

1.6.12 Using multiple vibrational energy levels

DSMC models for collisions between one or more polyatomic species can include the effect of multiple discrete vibrational levels, where a collision transfers vibrational energy not just between the two particles in aggregate but between the various levels defined for each particle species.

This kind of model can be enabled in SPARTA using the following commands:

- *species ... vibfile ...*
- *collide_modify vibrate discrete*
- *fix vibmode*
- *dump particle p_vibmode*

The *species command* with its *vibfile* option allows a separate file with per-species vibrational information to be read. See data/air.species.vib for an example of such a file.

Only species with 4,6,8 vibrational degrees of freedom, as defined in the species file read by the *species command*, need to be listed in the *vibfile*. These species have N modes, where N = degrees of freedom / 2. For each mode, a vibrational temperature, relaxation number, and degeneracy is defined in the *vibfile*. These quantities are used in the energy exchange formulas for each collision.

The *collide_modify vibrate discrete* command is used to enable the discrete model. Other allowed settings are *none* and *smooth*. The former turns off vibrational energy effects altogether. The latter uses a single continuous value to represent vibrational energy; no per-mode information is used.

The *fix vibmode* command is used to allocate per-particle storage for the population of levels appropriate to the particle's species. This will be from 1 to 4 values for each species. Note that this command must be used before particles are created via the *create_particles command* to allow the level populations for new particles to be set appropriately. The *fix vibmode command* doc page has more details.

The *dump particle* command can output the custom particle attributes defined by the *fix vibmode* command. E.g. this command

```
dump 1 particle 1000 tmp.dump id type x y z evib p_vibmode[1] p_vibmode[2] p_
↪vibmode[3]
```

will output for each particle evib = total vibrational energy (summed across all levels), and the population counts for the first 3 vibrational energy levels. The vibmode count will be 0 for vibrational levels that do not exist for particles of a particular species.

The *read_restart* doc page explains how to restart simulations where a fix like *fix vibmode* has been used to store extra per-particle properties.

1.6.13 Surface elements: explicit, implicit, distributed

SPARTA can work with two kinds of surface elements: explicit and implicit. Explicit surfaces are lines (2d) or triangles (3d) defined in surface data files read by the *read_surf command*. An individual element can be any size; a single surface element can intersect many grid cells. Implicit surfaces are lines (2d) or triangles (3d) defined by grid corner point data files read by the *read_isurf command*. The corner point values define lines or triangles that are wholly contained with single grid cells.

Note that you cannot mix explicit and implicit surfaces in the same simulation.

The data and attributes of explicit surface elements can be stored in one of two ways. The default is for each processor to store a copy of all the elements. Memory-wise, this is fine for most models. The other option is distributed, where each processor only stores copies of surface elements assigned to grid cells it owns or has a ghost copy of. For models

with huge numbers of surface elements, distributing them will use much less memory per processor. Note that a surface element requires about 150 bytes of storage, so storing a million requires about 150 MBytes.

Implicit surfaces are always stored in a distributed fashion. Each processor only stores a copy of surface elements assigned to grid cells it owns or has a ghost copy of. Note that 3d implicit surfs are not yet fully implemented. Specifically, the *read_isurf command* will not yet read and create them.

The *global surfs* command is used to specify the use of explicit versus implicit, and distributed versus non-distributed surface elements.

Unless noted, the following surface-related commands work with either explict or implicit surfaces, whether they are distributed or not. For large data sets, the read and write surf and isurf commands have options to use multiple files and/or operate in parallel which can reduce I/O times.

- *adapt_grid*
- *compute_isurf/grid* # for implicit surfs
- *compute_surf* # for explicit surfs
- *dump surf*
- *dump image*
- *fix adapt/grid*
- *fix emit/surf*
- *group surf*
- *read_isurf* # for implicit surfs
- *read_surf* # for explicit surfs
- *surf_modify*
- *write_isurf* # for implicit surfs
- *write_surf*

These command do not yet support distributed surfaces:

- *move_surf*
- *fix move/surf*
- *remove_surf*

1.6.14 Implicit surface ablation

The implicit surfaces described in the previous section can be used to perform ablation simulations, where the set of implicit surface elements evolve over time to model a receding surface. These are the relevant commands:

- *global surfs implicit*
- *read isurf*
- *fix ablate*
- *compute isurf/grid*
- *compute react/isurf/grid*
- *fix ave/grid*
- *write isurf*

- *write_surf*

The *read_isurf command* takes a binary file as an argument which contains a pixelated (2d) or voxelated (3d) representation of the surface (e.g. a porous heat shield material). It reads the file and assigns the pixel/voxel values to corner points of a region of the SPARTA grid.

The *read_isurf command* also takes the ID of a *fix ablate command* as an argument. This fix is invoked to perform a Marching Squares (2d) or Marching Cubes (3d) algorithm to convert the corner point values to a set of line segments (2d) or triangles (3d) each of which is wholly contained in a grid cell. It also stores the per grid cell corner point values.

If the *Nevery* argument of the *fix ablate command* is 0, ablation is never performed, the implicit surfaces are static. If it is non-zero, an ablation operation is performed every *Nevery* steps. A per-grid cell value is used to decrement the corner point values in each grid cell. The values can be (1) from a compute such as *compute isurf/grid* which tallies statistics about gas particle collisions with surfaces within each grid cell. Or *compute react/isurf/grid* which tallies the number of surface reactions that take place. Or values can be (2) from a fix such as *fix ave/grid* <*command-fix-ave-grid*> which time averages these statistics over many timesteps. Or they can be (3) generated randomly, which is useful for debugging.

The decrement of grid corner point values is done in a manner that models recession of the surface elements within in each grid cell. All the current implicit surface elements are then discarded, and new ones are generated from the new corner point values via the Marching Squares or Marching Cubes algorithm.

Important: Ideally these algorithms should preserve the gas flow volume inferred by the previous surfaces and only add to it with the new surfaces. However there are a few cases for the 3d Marching Cubes algorithm where the gas flow volume is not strictly preserved. This can trap existing particles inside the new surfaces. Currently SPARTA checks for this condition and deletes the trapped particles. In the future, we plan to modify the standard Marching Cubes algorithm to prevent this from happening. In our testing, the fraction of trapped particles in an ablation operation is tiny (around 0.005% or 5 in 100000). The number of deleted particles can be monitored as an output option by the *fix ablate command*.

The *write_isurf command* can be used to periodically write out a pixelated/voxelated file of corner point values, in the same format that the *read_isurf command* reads. Note that after ablation, corner point values are typically no longer integers, but floating point values. The *read_isurf command* and *write_isurf command* have options to work with both kinds of files. The *write_surf command* can also output implicit surface elements for visualization by tools such as ParaView which can read SPARTA surface element files after suitable post-processing. See the *Section tools paraview* doc page for more details.

1.6.15 Transparent surface elements

Transparent surfaces are useful for tallying flow statistics. Particles pass through them unaffected. However the flux of particles through those surface elements can be tallied and output.

Transparent surfaces are treated differently than regular surfaces. They do not need to be watertight. E.g. you can define a set of line segments that form a straight (or curved) line in 2d. Or a set of triangle that form a plane (or curved surface) in 3d. You can define multiple such surfaces, e.g. multiple disjoint planes, and tally flow statistics through each of them. To tally or sum the statistics separately, you may want to assign the triangles in each plane to a different surface group via the *read_surf group* or *group surf* commands.

Note that for purposes of collisions, transparent surface elements are one-sided. A collision is only tallied for particles passing through the outward face of the element. If you want to tally particles passing through in both directions, then define 2 transparent surfaces, with opposite orientation. Again, you may want to put the 2 surfaces in separate groups.

There also should be no restriction on transparent surfaces intersecting each other or intersecting regular surfaces. Though there may be some corner cases we haven't thought about or tested.

These are the relevant commands. See their doc pages for details:

- *read_surf transparent*
- *surf_collide transparent*
- *compute surf*

The *read_surf command* with its *transparent* keyword is used to flag all the read-in surface elements as transparent. This means they must be in a file separate from regular non-transparent elements.

The *surf_collide command* must be used with its *transparent* model and assigned to all transparent surface elements via the *surf_modify command*.

The *compute surf command* can be used to tally the count, mass flux, and energy flux of particles that pass through transparent surface elements. These quantities can then be time averaged via the *fix ave/surf* command or output via the *dump surf* command in the usual ways, as described in *Section Output from SPARTA*.

The examples/circle/in.circle.transparent script shows how to use these commands when modeling flow around a 2d circle. Two additional transparent line segments are placed in front of the circle to tally particle count and kinetic energy flux in both directions in front of the object. These are defined in the data.plane1 and data.plane2 files. The resulting tallies are output with the *stats_style command*. They could also be output with a *dump command* for more resolution if the 2 lines were each defined as multiple line segments.

1.7 Example problems

The SPARTA distribution includes an examples sub-directory with several sample problems. Each problem is in a sub-directory of its own. They are all small problems that run quickly, requiring at most a couple of minutes to run on a desktop machine. Many are 2d so that they run more quickly and can be easily visualized. Each problem has an input script (`in.*`) and produces a log file (`log.*`) when it runs. The data files they use for chemical species or reaction parameters are copied from the data directory so the problems are self-contained.

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.free.date.foo.P means it ran on P processors of machine "foo", using the dated SPARTA version.

If the "dump image" lines in each script are uncommented, a series of image snapshots will be produced. Animations of several of the examples can be viewed on the Movies section of the SPARTA WWW Site.

These are the sample problems in the examples sub-directories. See the examples/README file for more details.

- chem = chemistry in a 3d box
- circle = 2d flow around a circular object
- collide = collisional motion in a 3d box
- free = free molecular motion in a 3d box
- sphere = 3d flow around a sphere
- spiky = 2d flow around a spiky circle
- step = 2d flow around a staircase of steps

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

```
cd free
cp ../../src/spa_g++ .          # copy SPARTA executable to this dir
spa_g++ < in.free               # run the problem
```

Running the simulation produces the file log.sparta and optional `image.*.jpg`. If you have the freely available ImageMagick toolkit on your machine, you can run its "convert" command to create an animated GIF, and visualize it from the FireFox browser as follows:

```
convert image*ppm movie.gif
firefox ./movie.gif
```

A similar command should work with other browsers. Or you can select "Open File" under the File menu of your browser and load the animated GIF file directly.

1.8 Performance & scalability

The SPARTA distribution includes a bench sub-directory with several sample problems. The Benchmarks page of the SPARTA WWW Site gives timing data for these problems run on different machines, for both strong and weak scaling scenarioes:

- free = free molecular flow in a box
- collide = collisional molecular flow in a box
- sphere = flow around a sphere

For each problem there is an input script and sample log file outputs on different machines and different numbers of processors. E.g. a log file like log.free.foo.1M.P means the the free molecular problem with 1 million grid cells ran on P processors of machine "foo".

Each can be run as a serial benchmark (on one processor) or in parallel. In parallel, all the benchmarks can be run as a fixed-size problem, meaning the same problem is run on various numbers of processors (strong scaling). They can also be run as scaled-size problem, if the problem size is increased with the number of processors (weak scaling).

Here is an example of how to run the benchmark problems. See the bench/README file for more details.

- 1-processor runs:

```
spa_g++ -v x 100 -v y 100 -v z 100 < in.free
spa_g++ -v x 100 -v y 100 -v z 100 < in.collide
spa_g++ -v x 50 -v y 50 -v z 50 < in.sphere
```

- 32-processor runs:

```
mpirun -np 32 spa_g++ -v x 100 -v y 100 -v z 100 < in.free
mpirun -np 32 spa_g++ -v x 100 -v y 100 -v z 100 < in.collide
mpirun -np 32 spa_g++ -v x 50 -v y 50 -v z 50 < in.sphere
```

Note that the benchmark scripts define variables that can be set from the command line that determine the size of problem that is run. Specifically, the x,y,z variables specify the grid size (e.g. 100x100x100) that is used, and variable n specifies the number of particles (10 per grid cell in this case).

1.9 Additional tools

SPARTA is designed to be a computational kernel for performing DSMC 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 in the tools directory and are described briefly below.

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 web site.

Some of the Pizza.py tools relevant to SPARTA are as follows:

- dump - read, write, manipulate particle dump files
- gl - 3d interactive visualization via OpenGL of dump or surface files
- sdata - read, write, manipulate surface files
- olog - read log files and extract columns of data
- vcr - VCR-style GUI for 3d interactive OpenGL visualization of dump or surface files

The dump, sdata, and olog tools are included in the SPARTA distribution in the tools/pizza directory, and are used by some of the scripts discussed below.

This is the list of tools included in the tools directory of the SPARTA distribution. Each is described in more detail below.

- *dump2cfg tool* - convert a particle dump file to CFG format
- *dump2xyz tool* - convert a particle dump file to XYZ format
- *grid_refine* - refine a grid around a surface
- *implicit_grid* - create a random porous region with implicit surfaces
- *jagged* - create jagged 2d/3d surfaces with explicit surfaces
- *log2txt* - extract columns of info from a log file
- *logplot* - plot columns of info from a log file via GnuPlot
- *paraview* - converters of SPARTA data to ParaView format
- *stl2surf* - convert an STL text file into a SPARTA surface file
- *surf_create* - create a surface file with simple objects
- *surf_transform* - transform surface via tranlate/scale/rotate operations

1.9.1 dump2cfg tool

This is a Python script that converts a SPARTA particle dump file into extended CFG format so that it can be visualized by the AtomEye visualization program. AtomEye is a very fast particle visualizer, capable of interactive visualizations of millions of particles on a desktop machine. It is commonly used in the materials modeling community.

See the header of the script for the syntax used to run it.

This script uses one or more of the "Pizza.py" tools provided in the tools/pizza directory. See the tools/README file for info on how to set an environment variable so that the Pizza.py tool files can be found by Python, as well as instructions on various ways to run a Python script.

1.9.2 dump2xyz tool

This is a Python script that converts a SPARTA particle dump file into XYZ format so that it can be visualized by various visualization packages that read XYZ formatted files. An example is VMD package, commonly used in the molecular dynamics modeling community.

See the header of the script for the syntax used to run it.

This script uses one or more of the "Pizza.py" tools provided in the tools/pizza directory. See the tools/README file for info on how to set an environment variable so that the Pizza.py tool files can be found by Python, as well as instructions on various ways to run a Python script.

1.9.3 grid_refine tool

This is a Python script that creates a SPARTA grid file adapted around the lines or triangles in a SPARTA surface file. The resulting grid file can be read by the *read_grid* command. The surface file can be read by the *read_surf* command.

See the header of the script for the various adaptivity options that are supported, and the syntax used to run it.

1.9.4 implicit_grid tool

This is a Python script which can be used to generate binary files representing porous media samples, as read by the *read_isurf* command. The output files contain randomized grid corner point values which induce implicit surfaces which can contain huge numbers of surface elements. They are useful for stress testing the implicit surface options in SPARTA, as selected by the *global surfs* command.

See the header of the script for the syntax used to run it.

The examples/implicit directory uses these files as input.

1.9.5 jagged tools

These are 2 Python scripts (jagged2d.py and jagged3d.py) which can be used to generate SPARTA surface files in a pattern that can be very jagged. The surfaces can contain huge numbers of surface elements and be read by the *read_surf* command. They are useful for stress testing the explict surface options in SPARTA, including distributed or non-distributed storage, as selected by the *global surfs* command.

See the header of the scripts for the syntax used to run them.

The examples/jagged directory uses these files as input.

1.9.6 log2txt tool

This is a Python script that reads a SPARTA log file, extracts selected columns of statistical output, and writes them to a text file. It knows how to concatenate log file info across multiple successive runs. The columnar output can then be read by various plotting packages.

See the header of the script for the syntax used to run it.

This script uses one or more of the "Pizza.py" tools provided in the tools/pizza directory. See the tools/README file for info on how to set an environment variable so that the Pizza.py tool files can be found by Python, as well as instructions on various ways to run a Python script.

1.9.7 logplot tool

This is a Python script that reads a SPARTA log file, extracts the selected columns of statistical output, and plots them via the GnuPlot program. It knows how to concatenate log file info across multiple successive runs.

See the header of the script for the syntax used to run it. You must have GnuPlot installed on your system to use this script. If you can type "gnuplot" from the command line to start GnuPlot, it should work. If not (e.g. because you need a path name), then edit these 2 lines as needed in pizza/gnu.py:

```
except: PIZZA_GNUPLOT = "gnuplot"
except: PIZZA_GNUTERM = "x11"
```

For example, the first could become "/home/smith/bin/gnuplot". The second should only need changing if GnuPlot requires a different setting to plot to your screen.

This script uses one or more of the "Pizza.py" tools provided in the tools/pizza directory. See the tools/README file for info on how to set an environment variable so that the Pizza.py tool files can be found by Python, as well as instructions on various ways to run a Python script.

1.9.8 paraview tools

The tools/paraview directory has scripts which convert SPARTA grid and surface data (input and output) to ParaView format.

ParaView is a popular, powerful, freely-available visualization package. You must have ParaView installed to use the Python scripts. See tools/paraview/README for more details.

The scripts were developed by Tom Otahal (Sandia).

1.9.9 stl2surf tool

This is a Python script that reads a stereolithography (STL) text file and converts it to a SPARTA surface file. STL files contain a collection of triangles and can be created by various mesh-generation programs. The format for SPARTA surface files is described on the *read_surf* command doc page.

See the header of the script for the syntax used to run it, e.g.

```
% python stl2surf.py stlfile surffile
```

The script also checks the triangulated object to see if it is "watertight" and issues a warning if it is not, since SPARTA will perform the same check. The *read_surf* command doc page explains what watertight means for 3d objects.

1.9.10 surf_create tool

This is a Python script that creates a SPARTA surface file containing one or more simple objects whose surface is represented as triangules (3d) or line segments (2d). Such files can be read by the *read_surf* command. The 3d objects it supports are a sphere, box, and spikysphere (randomized radius at each point). The 2d objects it supports are a circle, rectangle, triangle, and spikycircly (randomized radius at each point).

See the header of the script for the syntax used to run it.

1.9.11 surf_transform tool

This is a Python script that transforms a SPARTA surface file into a new surface file using various operations supported by the *read_surf* command. These operations include translation, scaling, rotation, and inversion (changing which side of the surface is inside vs outside).

See the header of the script for the syntax used to run it.

1.10 Modifying & extending SPARTA

This section describes how to extend SPARTA by modifying its source code.

SPARTA is designed in a modular fashion so as to be easy to modify and extend with new functionality.

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 general interest to users, please submit it to the developers for inclusion in the released version of SPARTA.

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, arrays, structs).

The new features described in this section require you to write a new C++ derived class. 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 *collide_foo.cpp* and *collide_foo.h* that define a new class CollideFoo that computes inter-particle collisions 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

collide foo mix-ID params.foo 3.0

then your *collide_foo.h* file should be structured as follows:

```
#ifdef COLLIDE_CLASS
  CollideStyle(foo,CollideFoo)
#else
  ... (class definition for CollideFoo) ...
#endif
```

where "foo" is the style keyword in the collid command, and CollideFoo is the class name defined in your *collide_foo.cpp* and *collide_foo.h* files.

When you re-build SPARTA, your new collision model becomes part of the executable and can be invoked with a *collide* command like the example above. Arguments like a mixture ID, params.foo (a file with collision parameters), and 3.0 can be defined and processed by your new class.

As illustrated by this 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 that must be defined in the new derived class to give it the functionality SPARTA expects. Virtual functions that are not set to 0 are functions that can be optionally defined.

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 large volume of data on a single processor and analyze it. This runs the risk of seriously degrading the parallel efficiency.

  If you have a question about how to compute something or about internal SPARTA data structures or algorithms, feel free to send an email to the developers.
- If you add something you think is generally useful, also send an email to the developers so we can consider adding it to the SPARTA distribution.

1.10.1 Compute styles

Compute style commands calculate instantaneous properties of the simulated system. They can be global properties, or per particle or per grid cell or per surface element properties. The result can be single value or multiple values (global or per particle or per grid or per surf).

Here is a brief description of methods to define in a new derived class. See compute.h for details. All of these methods are optional.

init	initialization before a run
compute_scalar	compute a global scalar quantity
compute_vector	compute a global vector of quantities
compute_per_particle	compute one or more quantities per particle
compute_per_grid	compute one or more quantities per grid cell
compute_per_surf	compute one or more quantities per surface element
surf_tally	call when a particle hits a surface element
boundary_tally	call when a particle hits a simulation box boundary
memory_usage	tally memory usage

Note that computes with "/particle" in their style name calculate per particle quantities, with "/grid" in their name calculate per grid cell quantities, and with "/surf" in their name calculate per surface element properties. All others calcuulate global quantities.

Flags may also need to be set by a compute to enable specific properties. See the compute.h header file for one-line descriptions.

1.10.2 Fix styles

Fix style commands perform operations during the timestepping loop of a simulation. They can define methods which are invoked at different points within the timestep. They can be used to insert particles, perform load-balancing, or perform time-averaging of various quantities. They can also define and maintain new per-particle vectors and arrays

that define quantities that move with particles when they migrate from processor to processor or when the grid is rebalanced or adapated. They can also produce output of various kinds, similar to *compute command*.

Here is a brief description of methods to define in a new derived class. See fix.h for details. All of these methods are optional, except `setmask()`.

setmask	set flags that determine when the fix is called within a timestep
init	initialization before a run
start_of_step	called at beginning of timestep
end_of_step	called at end of timestep
add_particle	called when a particle is created
surf_react	called when a surface reaction occurs
memory_usage	tally memory usage

Flags may also need to be set by a fix to enable specific properties. See the fix.h header file for one-line descriptions.

Fixes can interact with the Particle class to create new per-particle vectors and arrays and access and update their values. These are the relevant Particle class methods:

add_custom	add a new custom vector or array
find_custom	find a previously defined custom vector or array
remove_custom	remove a custom vector or array

See *fix ambipolar* for an example of how these are used. It defines an integer vector called "ionambi" to flag particles as ambipolar ions, and a floatin-point array called "velambi" to store the velocity vector for the associated electron.

1.10.3 Region styles

Region style commands define geometric regions within the simulation box. Other commands use regions to limit their computational scope.

Here is a brief description of methods to define in a new derived class. See region.h for details. The `inside()` method is required.

inside: determine whether a point is inside/outside the region

1.10.4 Collision styles

Collision style commands define collision models that calculate interactions between particles in the same grid cell.

Here is a brief description of methods to define in a new derived class. See collide.h for details. All of these methods are required except `init()` and `modify_params()`.

init	initialization before a run
modify_params	process style-specific options of the *collide_modify command*
vremax_init	estimate vremax settings
attempt_collision	compute # of collisions to attempt for entire cell
attempt_collision	compute # of collisions to attempt between 2 species groups
test_collision	determine if a collision bewteen 2 particles occurs
setup_collision	pre-computation before a 2-particle collision
perform_collision	calculate the outcome of a 2-particle collision

1.10.5 Surface collision styles

Surface collision style commands define collision models that calculate interactions between a particle and surface element.

Here is a brief description of methods to define in a new derived class. See surf_collide.h for details. All of these methods are required except dynamic().

init	initialization before a run
collide	perform a particle/surface-element collision
dynamic	allow surface property to change during a simulation

1.10.6 Chemistry styles

Particle/particle chemistry models in SPARTA are specified by *reaction style commands* which define lists of possible reactions and their parameters.

Here is a brief description of methods to define in a new derived class. See react.h for details. The init() method is optional; the attempt() method is required.

init	initialization before a run
attempt	attempt a chemical reaction between two particles

1.10.7 Dump styles

Dump commands output snapshots of simulation data to a file periodically during a simulation, in a particular file format. Per particle, per grid cell, or per surface element data can be output.

Here is a brief description of methods to define in a new derived class. See dump.h for details. The `init_style()`, `modify_param()`, and `memory_usage()` methods are optional; all the others are required.

init_style	style-specific initialization before a run
modify_param	process style-specific options of the *dump_modify command*
write_header	write the header of a snapshot to a file
count	# of entities this processor will output
pack	pack a processor's data into a buffer
write_data	write a buffer of data to a file
memory_usage	tally memory usage

1.10.8 Input script commands

New commands can be added to SPARTA that will be recognized in input scripts. For example, the *create_particles command read_surf command*, and *run command* are all implemented in this fashion. When such a command is encountered in an input script, SPARTA simply creates a class with the corresponding name, invokes the "command" method of the class, and passes it the arguments from the input script. The `command()` method can perform whatever operations it wishes on SPARTA data structures.

The single method the new class must define is as follows:

command	operations performed by the input script command

Of course, the new class can define other methods and variables as needed.

1.11 Python interface to SPARTA

This section describes how to build and use SPARTA via a Python interface.

The SPARTA distribution includes the file python/sparta.py which wraps the library interface to SPARTA. This file makes it is possible to run SPARTA, invoke SPARTA commands or give it an input script, extract SPARTA results, and modify internal SPARTA variables, either from a Python script or interactively from a Python prompt. You can do the former in serial or parallel. Running Python interactively in parallel does not generally work, unless you have a package installed that extends your Python to enable multiple instances of Python to read what you type.

Python is a powerful scripting and programming language which can be used to wrap software like SPARTA and many other packages. It can be used to glue multiple pieces of software together, e.g. to run a coupled or multiscale model. See *Coupling SPARTA to other codes* of the manual and the examples/COUPLE directory of the distribution for more ideas about coupling SPARTA to other codes. See *build-library* about how to build SPARTA as a library, and *Library interface to SPARTA* 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 can extend the Python inteface as well. See details below.

Important: The examples/COUPLE dir has not been added to the distribution yet.

By using the Python interface, SPARTA can also be coupled with a GUI or other visualization tools that display graphs or animations in real time as SPARTA runs. Examples of such scripts are included 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, you need to do two things. You need to build SPARTA as a dynamic shared library, so it can be loaded by Python. And you need to tell Python how to find the library and the Python wrapper file python/sparta.py. Both these steps are discussed below. If you wish to run SPARTA in parallel from Python, you also need to extend your Python with MPI. This is also discussed below.

The 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.

1.11.1 Building SPARTA as a shared library

Instructions on how to build SPARTA as a shared library are given in *build-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".

For make, from the src directory, type

```
make makeshlib
make -f Makefile.shlib foo
```

For CMake, from the build directory, tyoe

```
cmake -C /path/to/sparta/cmake/presets/foo.cmake -DBUILD_SHARED_LIBS=ON /path/to/
↪sparta/cmake
make
```

where foo is the machine target name, such as icc or g++ or serial. This should create the file libsparta_foo.so in the src directory, as well as a soft link libsparta.so, which is what the Python wrapper will load by default. Note that if you are building multiple machine versions of the shared library, the soft link is always set to the most recently built version.

If this fails, see *Making SPARTA with optional packages* for more details, especially if your SPARTA build uses auxiliary libraries like MPI which may not be built as shared libraries on your system.

1.11.2 Installing the Python wrapper into Python

For Python to invoke SPARTA, there are 2 files it needs to know about:

- python/sparta.py
- src/libsparta.so

Sparta.py is the Python wrapper on the SPARTA library interface. Libsparta.so is the shared SPARTA library that Python loads, as described above.

You can insure Python can find these files in one of two ways:

- set two environment variables
- run the python/install.py script

If you set the paths to these files as environment variables, you only have to do it once. For the csh or tcsh shells, add something like this to your ~/.cshrc file, one line for each of the two files:

```
setenv PYTHONPATH $PYTHONPATH:/home/sjplimp/sparta/python
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/home/sjplimp/sparta/src
```

If you use the python/install.py script, you need to invoke it every time you rebuild SPARTA (as a shared library) or make changes to the python/sparta.py file.

You can invoke install.py from the python directory as

```
% python install.py [libdir] [pydir]
```

The optional libdir is where to copy the SPARTA shared library to; the default is /usr/local/lib. The optional pydir is where to copy the sparta.py file to; the default is the site-packages directory of the version of Python that is running the install script.

Note that libdir must be a location that is in your default LD_LIBRARY_PATH, like /usr/local/lib or /usr/lib. And pydir must be a location that Python looks in by default for imported modules, like its site-packages dir. If you want to

copy these files to non-standard locations, such as within your own user space, you will need to set your PYTHONPATH and LD_LIBRARY_PATH environment variables accordingly, as above.

If the install.py script does not allow you to copy files into system directories, prefix the python command with "sudo". If you do this, make sure that the Python that root runs is the same as the Python you run. E.g. you may need to do something like

```
% sudo /usr/local/bin/python install.py [libdir] [pydir]
```

You can also invoke install.py from the make command in the src directory as

```
% make install-python
```

In this mode you cannot append optional arguments. Again, you may need to prefix this with "sudo". In this mode you cannot control which Python is invoked by root.

Note that if you want Python to be able to load different versions of the SPARTA shared library (see *this section* below), you will need to manually copy files like libsparta_g++.so into the appropriate system directory. This is not needed if you set the LD_LIBRARY_PATH environment variable as described above.

1.11.3 Extending Python with MPI to run in parallel

If you wish to run SPARTA in parallel from Python, you need to extend your Python with an interface to MPI. This also allows you to make MPI calls directly from Python in your script, if you desire.

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 and exposing (some portion of) its interface to your Python script. This means Python 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 SPARTA in parallel and MPI calls themselves from a Python script which is itself running in parallel. However, when I downloaded and looked at a few of them, their documentation 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-2.1.4_94 as of Aug 2012), 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 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.py
```

where test.py 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 run on.

Important: To use Pypar and SPARTA in parallel from Python, you must insure both are using the same version of MPI. If you only have one MPI installed on your system, this is not an issue, but it can be if you have multiple MPIs. Your SPARTA build is explicit about which MPI it is using, since you specify the details in your lo-level src/MAKE/Makefile.foo file. Pypar uses the "mpicc" command to find information about the MPI it uses to build against. And it tries to load "libmpi.so" from the LD_LIBRARY_PATH. This may or may not find the MPI library that SPARTA is using. If you have problems running both Pypar and SPARTA together, this is an issue you may need to address, e.g. by moving other MPI installations so that Pypar finds the right one.

1.11.4 Testing the Python-SPARTA interface

To test if SPARTA is callable from Python, launch Python interactively and type:

```
>>> from sparta import sparta
>>> spa = sparta()
```

If you get no errors, you're ready to use SPARTA from Python. If the 2nd command fails, the most common error to see is

```
OSError: Could not load SPARTA dynamic library
```

which means Python was unable to load the SPARTA shared library. This typically occurs if the system can't find the SPARTA shared library or one of the auxiliary shared libraries it depends on, or if something about the library is incompatible with your Python. The error message should give you an indication of what went wrong.

You can also test the load directly in Python as follows, without first importing from the sparta.py file:

```
>>> from ctypes import CDLL
>>> CDLL("libsparta.so")
```

If an error occurs, carefully go thru the steps in *build-library* and above about building a shared library and about insuring Python can find the necessary two files it needs.

Test SPARTA and Python in serial:

To run a SPARTA test in serial, type these lines into Python interactively from the bench directory:

```
>>> from sparta import sparta
>>> spa = sparta()
>>> spa.file("in.free")
```

Or put the same lines in the file test.py and run it as

```
% python test.py
```

Either way, you should see the results of running the `in.free` benchmark on a single processor appear on the screen, the same as if you had typed something like:

```
spa_g++ < in.free
```

You can also pass command-line switches, e.g. to set input script variables, through the Python interface.

Replacing the "spa = sparta()" line above with

```
spa = sparta("","-v","x","100","-v","y","100","-v","z","100")
```

is the same as typing

```
spa_g++ -v x 100 -v y 100 -v z 100 < in.free
```

from the command line.

Test SPARTA and Python in parallel:

To run SPARTA in parallel, assuming you have installed the Pypar package as discussed above, create a test.py file containing these lines:

```
import pypar
from sparta import sparta
spa = sparta()
spa.file("in.free")
print "Proc %d out of %d procs has" % (pypar.rank(),pypar.size()),lmp
pypar.finalize()
```

You can then run it in parallel as:

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

and you should see the same output as if you had typed

```
% mpirun -np 4 spa_g++ < in.lj
```

Note that if you leave out the 3 lines from test.py that specify Pypar commands you will instantiate and run SPARTA independently on each of the P processors specified in the mpirun command. In this case you should get 4 sets of output, each showing that a SPARTA run was made on a single processor, instead of one set of output showing that SPARTA ran on 4 processors. If the 1-processor outputs occur, it means that Pypar is not working correctly.

Also note that once you import the PyPar module, Pypar initializes MPI for you, and you can use MPI calls directly in your Python script, as described in the Pypar documentation. The last line of your Python script should be pypar.finalize(), to insure MPI is shut down correctly.

Running Python scripts:

Note that any Python script (not just for SPARTA) 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 requires 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.

1.11.5 Using SPARTA from Python

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

```
from sparta import sparta
```

These are the methods defined by the sparta 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.

```
spa = sparta()            # create a SPARTA object using the default libsparta.so␣
↪library
spa = sparta("g++")       # create a SPARTA object using the libsparta_g++.so library
spa = sparta("",list)     # ditto, with command-line args, e.g. list = ["-echo","screen
↪"]
spa = sparta("g++",list)


spa.close()               # destroy a SPARTA object


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


fnum = spa.extract_global(name,type) # extract a global quantity
                                     # name = "dt", "fnum", etc
                      # type = 0 = int
                      #        1 = double
```

(continues on next page)

(continued from previous page)

```
temp = spa.extract_compute(id,style,type) # extract value(s) from a compute
                                          # id = ID of compute
                          # style = 0 = global data
                          #     1 = per particle data
                          #     2 = per grid cell data
                          #     3 = per surf element data
                          # type = 0 = scalar
                          #    1 = vector
                          #        2 = array


var = spa.extract_variable(name,flag)  # extract value(s) from a variable
                                       # name = name of variable
                           # flag = 0 = equal-style variable
                           #        1 = particle-style variable
```

Important: Currently, the creation of a SPARTA object from within sparta.py 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 know how to do it from Pypar. So for now, it runs with MPI_COMM_WORLD, which is all the processors. If someone figures out how to do this with one or more of the Python wrappers for MPI, like Pypar, please let us know and we will amend these doc pages.

Note that you can create multiple SPARTA objects in your Python script, and coordinate and run multiple simulations, e.g.

```
from sparta import sparta
spa1 = sparta()
spa2 = sparta()
spa1.file("in.file1")
spa2.file("in.file2")
```

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

The `extract_global()`, `extract_compute()`, 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 can easily be added. A double or integer is returned. You need to specify the appropriate data type via the type argument.

For `extract_compute()`, the global, per particle, per grid cell, or per surface element results calulated by the compute can be accessed. What is returned depends on whether the compute 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. See *Output from SPARTA (stats, dumps, computes, fixes, variables)* of the manual for a discussion of global, per particle, per grid, and per surf data, and of scalar, vector, and array data types. See the doc pages for individual *computes* for a description of what they calculate and store.

For `extract_variable()`, an *equal-style or particle-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 `particle-style` variables, a vector of doubles is returned, one value per particle, which you can use via normal Python subscripting.

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.
- Rebuild SPARTA as a shared library.
- Add a wrapper method to python/sparta.py for this interface function.
- You should now be able to invoke the new interface function from a Python script. Isn't ctypes amazing?

1.11.6 Example Python scripts that use SPARTA

There are demonstration Python scripts included in the python/examples directory of the SPARTA distribution, to illustrate what is possible when Python wraps SPARTA.

See the python/README file for more details.

1.12 Errors

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

1.12.1 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. On different machines, there can be numerical round-off in the computations which causes slight differences in particle trajectories or the number of particles, which will lead to numerical divergence of the particle trajectores and averaged statistical quantities within a few 100s or few 1000s of timesteps. When running on different numbers of processors, random numbers are used in different ways, so two simulations can be immediately different. However, the statistical properties (e.g. overall particle temperature or per grid cell temperature or surface energy flux) for the two runs on different machines or on different numbers of processors should still be similar.

A SPARTA simulation typically has two stages, setup and run. Most SPARTA errors are detected at setup time; others like running out of memory 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 physics or numerical mistakes, like choosing too big a timestep or specifying erroneous collision parameters. 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.sparta file, or using the *echo command* in your script or "-echo screen" as a *command-line argument* 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.

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 statistical output. If SPARTA crashes or hangs without spitting out an error message first then it could be a bug (see the *next section*) or one of the following cases:

SPARTA runs in the available memory a processor allows to be allocated. Most reasonable 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 statistical 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 statistical info frequently (e.g. every timestep) via the *stats command* so you can monitor what is happening. Visualizing the particle motion 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.

1.12.2 Reporting bugs

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

Check the New features and bug fixes section of the SPARTA web site to see if the bug has already been fixed.

If not, please email a description of the problem to the developers.

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 particles and grid cells and fewest number of processors and with the simplest and quick-to-run input script that reproduces the bug. And try to identify what command or combination of commands is causing the problem.

1.12.3 Error & warning messages

These are two alphabetic lists of the *Errors* and *Warnings* 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 create_particles command (create_particles.cpp:68)

means that line #68 in the file src/create_particles.cpp generated the error. Looking in the source code may help you figure out what went wrong.

Errors

%d read_surf point pairs are too close A pair of points is very close together, relative to grid size, inidicating the grid is too large, or an ill-formed surface.

%d read_surf points are not inside simulation box If clipping was not performed, all points in surf file must be inside (or on surface of) simulation box.

%d surface elements not assigned to a collision model All surface elements must be assigned to a surface collision model via the surf_modify command before a simulation is perforemd.

All universe/uloop variables must have same # of values Self-explanatory.

All variables in next command must be same style Self-explanatory.

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.

Axi-symmetry is not yet supported in SPARTA This error condition will be removed after axi-symmetry is fully implemented.

Axi-symmetry only allowed for 2d simulation Self-explanatory.

BPG edge on more than 2 faces This is an error when calculating how a 3d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

Bad grid of processors for balance_grid block Product of Px,Py,Pz must equal total number of processors.

Bad grid of processors for create_grid For block style, product of Px,Py,Pz must equal total number of processors.

Bigint setting in spatype.h is invalid Size of bigint is less than size of smallint.

Bigint setting in spatype.h is not compatible Bigint size stored in restart file is not consistent with SPARTA version you are running.

Both restart files must use % or neither Self-explanatory.

Both sides of boundary must be periodic Cannot specify a boundary as periodic only on the lo or hi side. Must be periodic on both sides.

Bound_modify surf requires wall be a surface The box boundary must be of style "s" to be assigned a surface collision model.

Bound_modify surf_collide ID is unknown Self-explanatory.

Boundary command after simulation box is defined The boundary command cannot be used after a read_data, read_restart, or create_box command.

Box boundary not assigned a surf_collide ID Any box boundary of style "s" must be assigned to a surface collision model via the bound_modify command, before a simulation is performed.

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.

Box ylo must be 0.0 for axi-symmetric model Self-explanatory.

Can only use -plog with multiple partitions Self-explanatory. See doc page discussion of command-line switches.

Can only use -pscreen with multiple partitions Self-explanatory. See doc page discussion of command-line switches.

Cannot add new species to mixture all or species This is done automatically for these 2 mixtures when each species is defined by the species command.

Cannot balance grid before grid is defined Self-explanatory.

Cannot create grid before simulation box is defined Self-explanatory.

Cannot create grid when grid is already defined Self-explanatory.

Cannot create particles before grid is defined Self-explanatory.

Cannot create particles before simulation box is defined Self-explanatory.

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 A simulation box can only be defined once.

Cannot open VSS parameter file %s Self-explantory.

Cannot open dir to search for restart file Using a "*" in the name of the restart file will open the current directory to search for matching file names.

Cannot open dump file The output file for the dump command cannot be opened. Check that the path and name are correct.

Cannot open file %s The specified file cannot be opened. Check that the path and name are correct. If the file is a compressed file, also check that the gzip executable can be found and run.

Cannot open file variable file %s The specified file cannot be opened. Check that the path and name are correct.

Cannot open fix ave/time file %s The specified file cannot be opened. Check that the path and name are correct.

Cannot open fix print file %s The output file generated by the fix print command cannot be opened

Cannot open gzipped file SPARTA was compiled without support for reading and writing gzipped files through a pipeline to the gzip program with -DSPARTA_GZIP.

Cannot open input script %s Self-explanatory.

Cannot open log.sparta The default SPARTA log file cannot be opened. Check that the directory you are running in allows for files to be created.

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 print file %s Self-explanatory.

Cannot open reaction file %s Self-explanatory.

Cannot open restart file %s The specified file cannot be opened. Check that the path and name are correct. If the file is a compressed file, also check that the gzip executable can be found and run.

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 Self-explanatory.

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 read grid before simulation box is defined Self-explanatory.

Cannot read grid when grid is already defined Self-explanatory.

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

Cannot read_surf after particles are defined This is because the newly read surface objects may enclose particles.

Cannot read_surf before grid ghost cells are defined This needs to be documented if keep this restriction.

Cannot read_surf before grid is defined Self-explantory.

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 reset timestep with a time-dependent fix defined The timestep cannot be reset when a fix that keeps track of elapsed time is in place.

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 global surfmax when surfaces already exist This setting must be made before any surfac elements are read via the read_surf command.

Cannot use collide_modify with no collisions defined A collision style must be specified first.

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

Cannot use dump_modify fileper without % in dump file name Self-explanatory.

Cannot use dump_modify nfile without % in dump file name Self-explanatory.

Cannot use fix inflow in y dimension for axisymmetric This is because the y dimension boundaries cannot be inflow boundaries for an axisymmetric model.

Cannot use fix inflow in z dimension for 2d simulation Self-explanatory.

Cannot use fix inflow n > 0 with perspecies yes This is because the perspecies option calculates the number of particles to insert itself.

Cannot use fix inflow on periodic boundary Self-explanatory.

Cannot use group keyword with mixture all or species This is because the groups for these 2 mixtures are predefined.

Cannot use include command within an if command Self-explanatory.

Cannot use non-rcb fix balance with a grid cutoff This is because the load-balancing will generate a partitioning of cells to processors that is dispersed and which will not work with a grid cutoff >= 0.0.

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

Cannot use specified create_grid options with more than one level When defining a grid with more than one level, the other create_grid keywords (stride, clump, block, etc) cannot be used. The child grid cells will be assigned to processors in round-robin order as explained on the create_grid doc page.

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.

Cannot use weight cell radius unless axisymmetric An axisymmetric model is required for this style of cell weighting.

Cannot use write_restart fileper without % in restart file name Self-explanatory.

Cannot use write_restart nfile without % in restart file name Self-explanatory.

Cannot weight cells before grid is defined Self-explanatory.

Cannot write grid when grid is not defined Self-explanatory.

Cannot write restart file before grid is defined Self-explanatory.

Cell ID has too many bits Cell IDs must fit in 32 bits (SPARTA small integer) or 64 bits (SPARTA big integer), as specified by the -DSPARTA_SMALL, -DSPARTA_BIG, or -DSPARTA_BIGBIG options in the low-level Makefile used to build SPARTA. See Section 2.2 of the manual for details. And see Section 4.8 for details on how cell IDs are formatted.

Cell type mis-match when marking on neigh proc Grid cell marking as inside, outside, or overlapping with surface elements failed. Please report the issue to the SPARTA developers.

Cell type mis-match when marking on self Grid cell marking as inside, outside, or overlapping with surface elements failed. Please report the issue to the SPARTA developers.

Cellint setting in spatype.h is not compatible Cellint size stored in restart file is not consistent with SPARTA version you are running.

Collision mixture does not contain all species The specified mixture must contain all species in the simulation so that they can be assigned to collision groups.

Collision mixture does not exist Self-explantory.

Compute ID for compute reduce does not exist Self-explanatory.

Compute ID for fix ave/grid does not exist Self-explanatory.

Compute ID for fix ave/surf does not exist Self-explanatory.

Compute ID for fix ave/time does not exist Self-explanatory.

Compute ID must be alphanumeric or underscore characters Self-explanatory.

Compute boundary mixture ID does not exist Self-explanatory.

Compute grid mixture ID does not exist Self-explanatory.

Compute reduce compute array is accessed out-of-range An index for the array is out of bounds.

Compute reduce compute calculates global or surf values The compute reduce command does not operate on this kind of values. The variable command has special functions that can reduce global values.

Compute reduce compute does not calculate a per-grid array This is necessary if a column index is used to specify the compute.

Compute reduce compute does not calculate a per-grid vector This is necessary if no column index is used to specify the compute.

Compute reduce compute does not calculate a per-particle array This is necessary if a column index is used to specify the compute.

Compute reduce compute does not calculate a per-particle vector This is necessary if no column index is used to specify the compute.

Compute reduce fix array is accessed out-of-range An index for the array is out of bounds.

Compute reduce fix calculates global values A fix that calculates peratom or local values is required.

Compute reduce fix does not calculate a per-grid array This is necessary if a column index is used to specify the fix.

Compute reduce fix does not calculate a per-grid vector This is necessary if no column index is used to specify the fix.

Compute reduce fix does not calculate a per-particle array This is necessary if a column index is used to specify the fix.

Compute reduce fix does not calculate a per-particle vector This is necessary if no column index is used to specify the fix.

Compute reduce fix does not calculate a per-surf array This is necessary if a column index is used to specify the fix.

Compute reduce fix does not calculate a per-surf vector This is necessary if no column index is used to specify the fix.

Compute reduce replace requires min or max mode Self-explanatory.

Compute reduce variable is not particle-style variable This is the only style of variable that can be reduced.

Compute sonine/grid mixture ID does not exist Self-explanatory.

Compute surf mixture ID does not exist Self-explanatory.

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.

Could not create a single particle The specified position was either not inside the simulation domain or not inside a grid cell with no intersections with any defined surface elements.

Could not find compute ID to delete Self-explanatory.

Could not find dump grid compute ID Self-explanatory.

Could not find dump grid fix ID Self-explanatory.

Could not find dump grid variable name Self-explanatory.

Could not find dump image compute ID Self-explanatory.

Could not find dump image fix ID Self-explanatory.

Could not find dump modify compute ID Self-explanatory.

Could not find dump modify fix ID Self-explanatory.

Could not find dump modify variable name Self-explanatory.

Could not find dump particle compute ID Self-explanatory.

Could not find dump particle fix ID Self-explanatory.

Could not find dump particle variable name Self-explanatory.

Could not find dump surf compute ID Self-explanatory.

Could not find dump surf fix ID Self-explanatory.

Could not find dump surf variable name Self-explanatory.

Could not find fix ID to delete Self-explanatory.

Could not find split point in split cell This is an error when calculating how a grid cell is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

Could not find stats compute ID Compute ID specified in stats_style command does not exist.

Could not find stats fix ID Fix ID specified in stats_style command does not exist.

Could not find stats variable name Self-explanatory.

Could not find surf_modify sc-ID Self-explanatory.

Could not find surf_modify surf-ID Self-explanatory.

Could not find undump ID A dump ID used in the undump command does not exist.

Cound not find dump_modify ID Self-explanatory.

Create_box z box bounds must straddle 0.0 for 2d simulations Self-explanatory.

Create_grid nz value must be 1 for a 2d simulation Self-explanatory.

Create_particles global option not yet implemented Self-explantory.

Create_particles mixture ID does not exist Self-explanatory.

Create_particles single requires z = 0 for 2d simulation Self-explanatory.

Create_particles species ID does not exist Self-explanatory.

Created incorrect # of particles: %ld versus %ld The create_particles command did not function properly.

Delete region ID does not exist Self-explanatory.

Did not assign all restart particles correctly One or more particles in the restart file were not assigned to a processor. Please report the issue to the SPARTA developers.

Did not assign all restart split grid cells correctly One or more split grid cells in the restart file were not assigned to a processor. Please report the issue to the SPARTA developers.

Did not assign all restart sub grid cells correctly One or more sub grid cells in the restart file were not assigned to a processor. Please report the issue to the SPARTA developers.

Did not assign all restart unsplit grid cells correctly One or more unsplit grid cells in the restart file were not assigned to a processor. Please report the issue to the SPARTA developers.

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.

Dump every variable returned a bad timestep The variable must return a timestep greater than the current timestep.

Dump grid and fix not computed at compatible times Fixes generate values on specific timesteps. The dump grid output does not match these timesteps.

Dump grid compute does not calculate per-grid array Self-explanatory.

Dump grid compute does not compute per-grid info Self-explanatory.

Dump grid compute vector is accessed out-of-range Self-explanatory.

Dump grid fix does not compute per-grid array Self-explanatory.

Dump grid fix does not compute per-grid info Self-explanatory.

Dump grid fix vector is accessed out-of-range Self-explanatory.

Dump grid variable is not grid-style variable Self-explanatory.

Dump image and fix not computed at compatible times Fixes generate values on specific timesteps. The dump image output does not match these timesteps.

Dump image cannot use grid and gridx/gridy/gridz Can only use grid option or one or more of grid x,y,z options by themselves, not together.

Dump image compute does not have requested column Self-explanatory.

Dump image compute does not produce a vector Self-explanatory.

Dump image compute is not a per-grid compute Self-explanatory.

Dump image compute is not a per-surf compute Self-explanatory.

Dump image fix does not have requested column Self-explanatory.

Dump image fix does not produce a vector Self-explanatory.

Dump image fix does not produce per-grid values Self-explanatory.

Dump image fix does not produce per-surf values Self-explanatory.

Dump image persp option is not yet supported Self-explanatory.

Dump image requires one snapshot per file Use a "*" in the filename.

Dump modify compute ID does not compute per-particle array Self-explanatory.

Dump modify compute ID does not compute per-particle info Self-explanatory.

Dump modify compute ID does not compute per-particle vector Self-explanatory.

Dump modify compute ID vector is not large enough Self-explanatory.

Dump modify fix ID does not compute per-particle array Self-explanatory.

Dump modify fix ID does not compute per-particle info Self-explanatory.

Dump modify fix ID does not compute per-particle vector Self-explanatory.

Dump modify fix ID vector is not large enough Self-explanatory.

Dump modify variable is not particle-style variable Self-explanatory.

Dump particle and fix not computed at compatible times Fixes generate values on specific timesteps. The dump particle output does not match these timesteps.

Dump particle compute does not calculate per-particle array Self-explanatory.

Dump particle compute does not calculate per-particle vector Self-explanatory.

Dump particle compute does not compute per-particle info Self-explanatory.

Dump particle compute vector is accessed out-of-range Self-explanatory.

Dump particle fix does not compute per-particle array Self-explanatory.

Dump particle fix does not compute per-particle info Self-explanatory.

Dump particle fix does not compute per-particle vector Self-explanatory.

Dump particle fix vector is accessed out-of-range Self-explanatory.

Dump particle variable is not particle-style variable Self-explanatory.

Dump surf and fix not computed at compatible times Fixes generate values on specific timesteps. The dump surf output does not match these timesteps.

Dump surf compute does not calculate per-surf array Self-explanatory.

Dump surf compute does not compute per-surf info Self-explanatory.

Dump surf compute vector is accessed out-of-range Self-explanatory.

Dump surf fix does not compute per-surf array Self-explanatory.

Dump surf fix does not compute per-surf info Self-explanatory.

Dump surf fix vector is accessed out-of-range Self-explanatory.

Dump surf variable is not surf-style variable Self-explanatory.

Dump_modify buffer yes not allowed for this style Not all dump styles allow dump_modify buffer yes. See the dump_modify doc page.

Dump_modify region ID does not exist Self-explanatory.

Duplicate cell ID in grid file Parent cell IDs must be unique.

Edge not part of 2 vertices This is an error when calculating how a 3d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

Edge part of invalid vertex This is an error when calculating how a 3d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

Edge part of same vertex twice This is an error when calculating how a 3d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

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

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

Failed to open FFmpeg pipeline to file %s The specified file cannot be opened. Check that the path and name are correct and writable and that the FFmpeg executable can be found and run.

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

File variable could not read value Check the file assigned to the variable.

Fix ID for compute reduce does not exist Self-explanatory.

Fix ID for fix ave/grid does not exist Self-explanatory.

Fix ID for fix ave/surf does not exist Self-explanatory.

Fix ID for fix ave/time does not exist Self-explanatory.

Fix ID must be alphanumeric or underscore characters Self-explanatory.

Fix ave/grid compute array is accessed out-of-range Self-explanatory.

Fix ave/grid compute does not calculate a per-grid array Self-explanatory.

Fix ave/grid compute does not calculate a per-grid vector Self-explanatory.

Fix ave/grid compute does not calculate per-grid values Self-explanatory.

Fix ave/grid fix array is accessed out-of-range Self-explanatory.

Fix ave/grid fix does not calculate a per-grid array Self-explanatory.

Fix ave/grid fix does not calculate a per-grid vector Self-explanatory.

Fix ave/grid fix does not calculate per-grid values Self-explanatory.

Fix ave/grid variable is not grid-style variable Self-explanatory.

Fix ave/surf compute array is accessed out-of-range Self-explanatory.

Fix ave/surf compute does not calculate a per-surf array Self-explanatory.

Fix ave/surf compute does not calculate a per-surf vector Self-explanatory.

Fix ave/surf compute does not calculate per-surf values Self-explanatory.

Fix ave/surf fix array is accessed out-of-range Self-explanatory.

Fix ave/surf fix does not calculate a per-surf array Self-explanatory.

Fix ave/surf fix does not calculate a per-surf vector Self-explanatory.

Fix ave/surf fix does not calculate per-surf values Self-explanatory.

Fix ave/surf variable is not surf-style variable Self-explanatory.

Fix ave/time cannot use variable with vector mode Variables produce scalar values.

Fix ave/time columns are inconsistent lengths Self-explanatory.

Fix ave/time compute array is accessed out-of-range An index for the array is out of bounds.

Fix ave/time compute does not calculate a scalar Self-explantory.

Fix ave/time compute does not calculate a vector Self-explantory.

Fix ave/time compute does not calculate an array Self-explanatory.

Fix ave/time compute vector is accessed out-of-range The index for the vector is out of bounds.

Fix ave/time fix array is accessed out-of-range An index for the array is out of bounds.

Fix ave/time fix does not calculate a scalar Self-explanatory.

Fix ave/time fix does not calculate a vector Self-explanatory.

Fix ave/time fix does not calculate an array Self-explanatory.

Fix ave/time fix vector is accessed out-of-range The index for the vector is out of bounds.

Fix ave/time variable is not equal-style variable Self-explanatory.

Fix command before simulation box is defined The fix command cannot be used before a read_data, read_restart, or create_box command.

Fix for fix ave/grid not computed at compatible time Fixes generate values on specific timesteps. Fix ave/grid is requesting a value on a non-allowed timestep.

Fix for fix ave/surf not computed at compatible time Fixes generate their values on specific timesteps. Fix ave/surf is requesting a value on a non-allowed timestep.

Fix for fix ave/time not computed at compatible time Fixes generate their values on specific timesteps. Fix ave/time is requesting a value on a non-allowed timestep.

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.

Fix inflow mixture ID does not exist Self-explanatory.

Fix inflow used on outflow boundary Self-explanatory.

Fix used in compute reduce not computed at compatible time Fixes generate their values on specific timesteps. Compute reduce is requesting a value on a non-allowed timestep.

Found edge in same direction This is an error when calculating how a 3d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

Found no restart file matching pattern When using a "*" in the restart file name, no matching file was found.

Gravity in y not allowed for axi-symmetric model Self-explanatory.

Gravity in z not allowed for 2d Self-explanatory.

Grid cell corner points on boundary marked as unknown = %d Corner points of grid cells on the boundary of the simulation domain were not all marked successfully as inside, outside, or overlapping with surface elements. Please report the issue to the SPARTA developers.

Grid cells marked as unknown = %d Grid cell marking as inside, outside, or overlapping with surface elements did not successfully mark all cells. Please report the issue to the SPARTA developers.

Grid cutoff is longer than box length in a periodic dimension This is not allowed. Reduce the size of the cutoff specified by the global gridcut command.

Grid file does not contain parents No parent cells appeared in the grid file.

Grid in/out other-mark error %dn Grid cell marking as inside, outside, or overlapping with surface elements failed. Please report the issue to the SPARTA developers.

Grid in/out self-mark error %d for icell %d, icorner %d, connect %d %d, other cell %d, other corner %d, values %d %dn A grid cell was incorrectly marked as inside, outside, or overlapping with surface elements. Please report the issue to the SPARTA developers.

Grid-style variables are not yet implemented 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.

Inconsistent surface to grid mapping in read_restart When surface elements were mapped to grid cells after reading a restart file, an inconsitent count of elements in a grid cell was found, as compared to the original simulation, which should not happen. Please report the issue to the SPARTA developers.

Incorrect format of parent cell in grid file Number of words in a parent cell line was not the expected number.

Incorrect line format in VSS parameter file Number of parameters in a line read from file is not valid.

Incorrect line format in species file Line read did not have expected number of fields.

Incorrect line format in surf file Self-explanatory.

Incorrect point format in surf file Self-explanatory.

Incorrect triangle format in surf file Self-explanatory.

Index between variable brackets must be positive Self-explanatory.

Input line quote not followed by whitespace An end quote must be followed by whitespace.

Invalid Boolean syntax in if command Self-explanatory.

Invalid Nx,Ny,Nz values in grid file A Nx or Ny or Nz value for a parent cell is <= 0.

Invalid SPARTA restart file The file does not appear to be a SPARTA restart file since it does not have the expected magic string at the beginning.

Invalid attribute in dump grid command Self-explanatory.

Invalid attribute in dump modify command Self-explantory.

Invalid attribute in dump particle command Self-explanatory.

Invalid attribute in dump surf command Self-explanatory.

Invalid balance_grid style for non-uniform grid Some balance styles can only be used when the grid is uniform. See the command doc page for details.

Invalid call to ComputeGrid::post_process_grid() This indicates a coding error. Please report the issue to the SPARTA developers.

Invalid call to ComputeSonineGrid::post_process_grid() This indicates a coding error. Please report the issue to the SPARTA developers.

Invalid cell ID in grid file A cell ID could not be converted into numeric format.

Invalid character in species ID The only allowed characters are alphanumeric, an underscore, a plus sign, or a minus sign.

Invalid collide style The choice of collision style is unknown.

Invalid color in dump_modify command The specified color name was not in the list of recognized colors. See the dump_modify doc page.

Invalid color map min/max values The min/max values are not consistent with either each other or with values in the color map.

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 compute property/grid field for 2d simulation Fields that reference z-dimension properties cannot be used in a 2d simulation.

Invalid compute style Self-explanatory.

Invalid dump frequency Dump frequency must be 1 or greater.

Invalid dump grid field for 2d simulation Self-explanatory.

Invalid dump image filename The file produced by dump image cannot be binary and must be for a single processor.

Invalid dump image persp value Persp value must be >= 0.0.

Invalid dump image theta value Theta must be between 0.0 and 180.0 inclusive.

Invalid dump image zoom value Zoom value must be > 0.0.

Invalid dump movie filename The file produced by dump movie cannot be binary or compressed and must be a single file for a single processor.

Invalid dump style The choice of dump style is unknown.

Invalid dump surf field for 2d simulation Self-explanatory.

Invalid dump_modify threshhold operator Operator keyword used for threshold specification in not recognized.

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

Invalid fix ave/time off column Self-explantory.

Invalid fix style The choice of fix style is unknown.

Invalid flag in grid section of restart file Unrecognized entry in restart file.

Invalid flag in header section of restart file Unrecognized entry in restart file.

Invalid flag in layout section of restart file Unrecognized entry in restart file.

Invalid flag in particle section of restart file Unrecognized entry in restart file.

Invalid flag in peratom section of restart file The format of this section of the file is not correct.

Invalid flag in surf section of restart file Unrecognized entry in restart file.

Invalid image up vector Up vector cannot be (0,0,0).

Invalid immediate variable Syntax of immediate value is incorrect.

Invalid keyword in compute property/grid command Self-explantory.

Invalid keyword in stats_style command One or more specified keywords are not recognized.

Invalid math function in variable formula Self-explanatory.

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

Invalid point index in line Self-explanatory.

Invalid point index in triangle Self-explanatory.

Invalid react style The choice of reaction style is unknown.

Invalid reaction coefficients in file Self-explanatory.

Invalid reaction formula in file Self-explanatory.

Invalid reaction style in file Self-explanatory.

Invalid reaction type in file Self-explanatory.

Invalid read_surf command Self-explanatory.

Invalid read_surf geometry transformation for 2d simulation Cannot perform a transformation that changes z cooridinates of points for a 2d simulation.

Invalid region style The choice of region style is unknown.

Invalid replace values in compute reduce Self-explanatory.

Invalid reuse of surface ID in read_surf command Surface IDs must be unique.

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 run command start/stop value Self-explanatory.

Invalid run command upto value Self-explanatory.

Invalid special function in variable formula Self-explanatory.

Invalid species ID in species file Species IDs are limited to 15 characters.

Invalid stats keyword in variable formula The keyword is not recognized.

Invalid surf_collide style Self-explanatory.

Invalid syntax in variable formula Self-explanatory.

Invalid use of library file() function This function is called thru the library interface. This error should not occur. Contact the developers if it does.

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 in special function next Only file-style or atomfile-style variables can be used with next().

Invalid variable style with next command Variable styles *equal* and *world* cannot be used in a next command.

Ionization and recombination reactions are not yet implemented This error conditions will be removed after those reaction styles are fully implemented.

Irregular comm recv buffer exceeds 2 GB MPI does not support a communication buffer that exceeds a 4-byte integer in size.

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 spatype.h are not compatible The size of the MPI datatype does not match the size of a bigint.

Migrate cells send buffer exceeds 2 GB MPI does not support a communication buffer that exceeds a 4-byte integer in size.

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.

Mixture %s fractions exceed 1.0 The sum of fractions must not be > 1.0.

Mixture ID must be alphanumeric or underscore characters Self-explanatory.

Mixture group ID must be alphanumeric or underscore characters Self-explanatory.

Mixture species is not defined One or more of the species ID is unknown.

Modulo 0 in variable formula Self-explanatory.

More than one positive area with a negative area SPARTA cannot determine which positive area the negative area is inside of, if a cell is so large that it includes both positive and negative areas.

More than one positive volume with a negative volume SPARTA cannot determine which positive volume the negative volume is inside of, if a cell is so large that it includes both positive and negative volumes.

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.

Next command must list all universe and uloop variables This is to insure they stay in sync.

No dump grid attributes specified Self-explanatory.

No dump particle attributes specified Self-explanatory.

No dump surf attributes specified Self-explanatory.

No positive areas in cell This is an error when calculating how a 2d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

No positive volumes in cell This is an error when calculating how a 3d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

Non digit character between brackets in variable Self-explantory.

Number of groups in compute boundary mixture has changed This mixture property cannot be changed after this compute command is issued.

Number of groups in compute grid mixture has changed This mixture property cannot be changed after this compute command is issued.

Number of groups in compute sonine/grid mixture has changed This mixture property cannot be changed after this compute command is issued.

Number of groups in compute surf mixture has changed This mixture property cannot be changed after this compute command is issued.

Number of groups in compute tvib/grid mixture has changed This mixture property cannot be changed after this compute command is issued.

Number of species in compute tvib/grid mixture has changed This mixture property cannot be changed after this compute command is issued.

Numeric index is out of bounds A command with an argument that specifies an integer or range of integers is using a value that is less than 1 or greater than the maximum allowed limit.

Nz value in read_grid file must be 1 for a 2d simulation Self-explanatory.

Only ylo boundary can be axi-symmetric Self-explanatory. See the boundary doc page for more details.

Owned cells with unknown neighbors = %d One or more grid cells have unknown neighbors which will prevent particles from moving correctly. Please report the issue to the SPARTA developers.

Parent cell child missing Hierarchical grid traversal failed. Please report the issue to the SPARTA developers.

Parent cell's parent does not exist in grid file Parent cells must be listed in order such that each cell's parents have already appeared in the list.

Particle %d on proc %d hit inside of surf %d on step %ld This error should not happen if particles start outside of physical objects. Please report the issue to the SPARTA developers.

Particle %d,%d on proc %d is in invalid cell on timestep %ld The particle is in a cell indexed by a value that is out-of-bounds for the cells owned by this processor.

Particle %d,%d on proc %d is in split cell on timestep %ld This should not happend. The particle should be in one of the sub-cells of the split cell.

Particle %d,%d on proc %d is outside cell on timestep %ld The particle's coordinates are not within the grid cell it is supposed to be in.

Particle vector in equal-style variable formula Equal-style variables cannot use per-particle quantities.

Particle-style variable in equal-style variable formula Equal-style variables cannot use per-particle quantities.

Partition numeric index is out of bounds It must be an integer from 1 to the number of partitions.

Per-particle compute in equal-style variable formula Equal-style variables cannot use per-particle quantities.

Per-particle fix in equal-style variable formula Equal-style variables cannot use per-particle quantities.

Per-processor particle count is too big No processor can have more particle than fit in a 32-bit integer, approximately 2 billion.

Point appears first in more than one CLINE This is an error when calculating how a 2d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

Point appears last in more than one CLINE This is an error when calculating how a 2d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

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.

React tce can only be used with collide vss Self-explanatory.

Read_grid did not find parents section of grid file Expected Parents section but did not find keyword.

Read_surf did not find lines section of surf file Expected Lines section but did not find keyword.

Read_surf did not find points section of surf file Expected Parents section but did not find keyword.

Read_surf did not find triangles section of surf file Expected Triangles section but did not find keyword.

Region ID for dump custom does not exist Self-explanatory.

Region intersect region ID does not exist One or more of the region IDs specified by the region intersect command does not exist.

Region union region ID does not exist One or more of the region IDs specified by the region union command does not exist.

Replacing a fix, but new style != old style A fix ID can be used a 2nd time, but only if the style matches the previous fix. In this case it is assumed you with to reset a fix's parameters. This error may mean you are mistakenly re-using a fix ID when you do not intend to.

Request for unknown parameter from collide VSS model does not have the parameter being requested.

Restart file byte ordering is not recognized The file does not appear to be a SPARTA restart file since it doesn't contain a recognized byte-ordering flag at the beginning.

Restart file byte ordering is swapped The file was written on a machine with different byte-ordering than the machine you are reading it on.

Restart file incompatible with current version This is probably because you are trying to read a file created with a version of SPARTA that is too old compared to the current version.

Restart file is a multi-proc file The file is inconsistent with the filename specified for it.

Restart file is not a multi-proc file The file is inconsistent with the filename specified for it.

Restart variable returned a bad timestep The variable must return a timestep greater than the current timestep.

Reuse of compute ID A compute ID cannot be used twice.

Reuse of dump ID A dump ID cannot be used twice.

Reuse of region ID A region ID cannot be used twice.

Reuse of surf_collide ID A surface collision model ID cannot be used more than once.

Run command before grid ghost cells are defined Normally, ghost cells will be defined when the grid is created via the create_grid or read_grid commands. However, if the global gridcut cutoff is set to a value >= 0.0, then ghost cells can only be defined if the partiioning of cells to processors is clumped, not dispersed. See the fix balance command for an explanation. Invoking the fix balance command with a clumped option will trigger ghost cells to be defined.

Run command before grid is defined Self-explanatory.

Run command start value is after start of run Self-explanatory.

Run command stop value is before end of run Self-explanatory.

Seed command has not been used This command should appear near the beginning of your input script, before any random numbers are needed by other commands.

Sending particle to self This error should not occur. Please report the issue to the SPARTA developers.

Single area is negative, inverse donut An inverse donut is a surface with a flow region interior to the donut hole and also exterior to the entire donut. This means the flow regions are disconnected. SPARTA cannot correctly compute the flow area of this kind of object.

Single volume is negative, inverse donut An inverse donut is a surface with a flow region interior to the donut hole and also exterior to the entire donut. This means the flow regions are disconnected. SPARTA cannot correctly compute the flow volume of this kind of object.

Singlet BPG edge not on cell face This is an error when calculating how a 3d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

Singlet CLINES point not on cell border This is an error when calculating how a 2d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

Small,big integers are not sized correctly This error occurs whenthe sizes of smallint and bigint as defined in src/spatype.h are not what is expected. Please report the issue to the SPARTA developers.

Smallint setting in spatype.h is invalid It has to be the size of an integer.

Smallint setting in spatype.h is not compatible Smallint size stored in restart file is not consistent with SPARTA version you are running.

Species %s did not appear in VSS parameter file Self-explanatory.

Species ID does not appear in species file Could not find the requested species in the specified file.

Species ID is already defined Species IDs must be unique.

Sqrt of negative value in variable formula Self-explanatory.

Stats and fix not computed at compatible times Fixes generate values on specific timesteps. The stats output does not match these timesteps.

Stats compute array is accessed out-of-range Self-explanatory.

Stats compute does not compute array Self-explanatory.

Stats compute does not compute scalar Self-explanatory.

Stats compute does not compute vector Self-explanatory.

Stats compute vector is accessed out-of-range Self-explanatory.

Stats every variable returned a bad timestep The variable must return a timestep greater than the current timestep.

Stats fix array is accessed out-of-range Self-explanatory.

Stats fix does not compute array Self-explanatory.

Stats fix does not compute scalar Self-explanatory.

Stats fix does not compute vector Self-explanatory.

Stats fix vector is accessed out-of-range Self-explanatory.

Stats variable cannot be indexed A variable used as a stats keyword cannot be indexed. E.g. v_foo must be used, not v_foo**100**.

Stats variable is not equal-style variable Only equal-style variables can be output with stats output, not particle-style or grid-style or surf-style variables.

Stats_modify every variable returned a bad timestep The variable must return a timestep greater than the current timestep.

Stats_modify int format does not contain d character Self-explanatory.

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

Support for writing images in JPEG format not included SPARTA was not built with the -DSPARTA_JPEG switch in the Makefile.

Support for writing images in PNG format not included SPARTA was not built with the -DSPARTA_PNG switch in the Makefile.

Support for writing movies not included SPARTA was not built with the -DSPARTA_FFMPEG switch in the Makefile

Surf file cannot contain lines for 3d simulation Self-explanatory.

Surf file cannot contain triangles for 2d simulation Self-explanatory.

Surf file does not contain lines Required for a 2d simulation.

Surf file does not contain points Self-explanatory.

Surf file does not contain triangles Required for a 3d simulation.

Surf-style variables are not yet implemented Self-explanatory.

Surf_collide ID must be alphanumeric or underscore characters Self-explanatory.

Surf_collide diffuse rotation invalid for 2d Specified rotation vector must be in z-direction.

Surf_collide diffuse variable is invalid style It must be an equal-style variable.

Surf_collide diffuse variable name does not exist Self-explanatory.

Surface check failed with %d duplicate edges One or more edges appeared in more than 2 triangles.

Surface check failed with %d duplicate points One or more points appeared in more than 2 lines.

Surface check failed with %d infinitely thin line pairs Two adjacent lines have normals in opposite directions indicating the lines overlay each other.

Surface check failed with %d infinitely thin triangle pairs Two adjacent triangles have normals in opposite directions indicating the triangles overlay each other.

Surface check failed with %d points on lines One or more points are on a line they are not an end point of, which indicates an ill-formed surface.

Surface check failed with %d points on triangles One or more points are on a triangle they are not an end point of, which indicates an ill-formed surface.

Surface check failed with %d unmatched edges One or more edges did not appear in a triangle, or appeared only once and edge is not on surface of simulation box.

Surface check failed with %d unmatched points One or more points did not appear in a line, or appeared only once and point is not on surface of simulation box.

Timestep must be >= 0 Reset_timestep cannot be used to set a negative timestep.

Too big a timestep Reset_timestep timestep value must fit in a SPARTA big integer, as specified by the -DSPARTA_SMALL, -DSPARTA_BIG, or -DSPARTA_BIGBIG options in the low-level Makefile used to build SPARTA. See Section 2.2 of the manual for details.

Too many surfs in one cell Use the global surfmax command to increase this max allowed number of surfs per grid cell.

Too many timesteps The cummulative timesteps must fit in a SPARTA big integer, as as specified by the -DSPARTA_SMALL, -DSPARTA_BIG, or -DSPARTA_BIGBIG options in the low-level Makefile used to build SPARTA. See Section 2.2 of the manual for details.

Too much buffered per-proc info for dump Number of dumped values per processor cannot exceed a small integer (~2 billion values).

Too much per-proc info for dump Number of local atoms times number of columns must fit in a 32-bit integer for dump.

Unbalanced quotes in input line No matching end double quote was found following a leading double quote.

Unexpected end of data file SPARTA hit the end of the data file while attempting to read a section. Something is wrong with the format of the data file.

Unexpected end of grid file Self-explantory.

Unexpected end of surf file Self-explanatory.

Units command after simulation box is defined The units command cannot be used after a read_data, read_restart, or create_box command.

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.

Unknown outcome in reaction The specified type of the reaction is not encoded in the reaction style.

VSS parameters do not match current species Species cannot be added after VSS colision file is read.

Variable ID in variable formula does not exist Self-explanatory.

Variable evaluation before simulation box is defined Cannot evaluate a compute or fix or atom-based value in a variable before the simulation has been setup.

Variable for dump every is invalid style Only equal-style variables can be used.

Variable for dump image center is invalid style Must be an equal-style variable.

Variable for dump image persp is invalid style Must be an equal-style variable.

Variable for dump image phi is invalid style Must be an equal-style variable.

Variable for dump image theta is invalid style Must be an equal-style variable.

Variable for dump image zoom is invalid style Must be an equal-style variable.

Variable for restart is invalid style It must be an equal-style variable.

Variable for stats every is invalid style It must be an equal-style variable.

Variable formula compute array is accessed out-of-range Self-explanatory.

Variable formula compute vector is accessed out-of-range Self-explanatory.

Variable formula fix array is accessed out-of-range Self-explanatory.

Variable formula fix vector is accessed out-of-range Self-explanatory.

Variable has circular dependency A circular dependency is when variable "a" in used by variable "b" and variable "b" is also used by varaible "a". Circular dependencies with longer chains of dependence are also not allowed.

Variable name between brackets must be alphanumeric or underscore characters Self-explanatory.

Variable name for compute reduce does not exist Self-explanatory.

Variable name for dump every does not exist Self-explanatory.

Variable name for dump image center does not exist Self-explanatory.

Variable name for dump image persp does not exist Self-explanatory.

Variable name for dump image phi does not exist Self-explanatory.

Variable name for dump image theta does not exist Self-explanatory.

Variable name for dump image zoom does not exist Self-explanatory.

Variable name for fix ave/grid does not exist Self-explanatory.

Variable name for fix ave/surf does not exist Self-explanatory.

Variable name for fix ave/time does not exist Self-explanatory.

Variable name for restart does not exist Self-explanatory.

Variable name for stats every does not exist Self-explanatory.

Variable name must be alphanumeric or underscore characters Self-explanatory.

Variable stats keyword cannot be used between runs Stats keywords that refer to time (such as cpu, elapsed) do not make sense in between runs.

Vertex contains duplicate edge This is an error when calculating how a 3d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

Vertex contains edge that doesn't point to it This is an error when calculating how a 3d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

Vertex contains invalid edge This is an error when calculating how a 3d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

Vertex has less than 3 edges This is an error when calculating how a 3d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

Vertex pointers to last edge are invalid This is an error when calculating how a 3d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

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.

Y cannot be periodic for axi-symmetric Self-explanatory. See the boundary doc page for more details.

Z dimension must be periodic for 2d simulation Self-explanatory.

Warnings

%d particles were in wrong cells on timestep %ld This is the total number of particles that are incorrectly matched to their grid cell.

Grid cell interior corner points marked as unknown = %d Corner points of grid cells interior to the simulation domain were not all marked successfully as inside, outside, or overlapping with surface elements. This should normally not happen, but does not affect simulations.

More than one compute ke/particle This may be inefficient since each such compute stores a vector of length equal to the number of particles.

Restart file used different # of processors The restart file was written out by a SPARTA simulation running on a different number of processors. This means you will likely want to re-balance the grid cells and particles across processors. This can be done using the balance or fix balance commands.

Surface check found %d nearly infinitely thin line pairs Two adjacent lines have normals in nearly opposite directions indicating the lines nearly overlay each other.

Surface check found %d nearly infinitely thin triangle pairs Two adjacent triangles have normals in nearly opposite directions indicating the triangles nearly overlay each other.

Surface check found %d points nearly on lines One or more points are nearly on a line they are not an end point of, which indicates an ill-formed surface.

Surface check found %d points nearly on triangles One or more points are nearly on a triangle they are not an end point of, which indicates an ill-formed surface.

1.13 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.

1.13.1 Coming attractions

The developers wish list link" on the SPARTA web page gives a list of features we are planning to add to SPARTA in the future. Please contact the you are interested in contributing to the those developments 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.

1.13.2 Past versions

Sandia's predecessor to SPARTA is a DSMC code called ICARUS. It was developed in the early 1990s by Tim Bartel and Steve Plimpton. It was later modified and extended by Michael Gallis.

ICARUS is a 2d code, written in Fortran, which models the flow geometry around bodies with a collection of adjoining body-fitted grid blocks. The geometry of the grid cells within in a single block is represented with analytic equations, which allows for fast particle tracking.

Some details about ICARUS, including simulation snapshots and papers, are discussed on this page

Performance-wise ICARUS scaled quite well on several generations of parallel machines, and is still used by Sandia researchers today. ICARUS was export-controlled software, and so was not distributed widely outside of Sandia.

SPARTA development began in late 2011. In contrast to ICARUS, it is a 3d code, written in C++, and uses a hierarchical Cartesian grid to track particles. Surfaces are embedded in the grid, which cuts and splits their flow volumes.

The Authors link on the SPARTA web page gives a timeline of features added to the code since its initial open-source release.

1.14 List of Commands

1.14.1 adapt_grid command

Syntax:

```
adapt_grid group-ID action1 action2 style args ... keyword args ...
```

- group-ID = group ID for which grid cell adaptation will be attempted
- action1 = `refine` or `coarsen`
- action2 = `coarsen` or `refine`, optional
- style = `particle` or `surf` or `value` or `random`
  - particle args = rthresh cthresh
    - rcount = threshold in particle count for refinment
    - ccount = threshold in particle count for coarsening
  - surf arg = surfID ssize
    - surfID = group ID for which surface elements to consider
    - ssize = do not refine to create cells smaller than ssize (dist units) coarsen only if child cells are smaller than ssize (dist units)
  - value args = c_ID/c_ID[N]/f_ID/f_ID[N] rthresh cthresh
    - c_ID = ID of a compute that calculates a per grid vector, use values from vector
    - c_ID[N] = ID of a compute that calculates a per grid array, use values from Nth column of array
    - f_ID = ID of a fix that calculates a per grid vector, use vector
    - f_ID[N] = ID of a fix that calculates a per grid array, use Nth column of array
    - rvalue = threshold in value for refinement
    - cvalue = threshold in value for coarsening
  - random args = rfrac cfrac
    - rfrac = fraction of child cells to refine
    - cfrac = fraction of parent cells to coarsen
- zero or more keyword/args pairs may be appended

  keyword = `iterate` or `maxlevel` or `minlevel` or `thresh` or `combine`

or `cells` or `region` or `dir`

- iterate arg = niterate
  - niterate = number of iterations of action loop
- maxlevel arg = Nmax
  - Nmax = do not refine to create child cells at a level > Nmax
- minlevel arg = Nmin
  - Nmin = do not coarsen to create child cells at a level < Nmin
- thresh args = rdecide cdecide
  - rdecide = less or more = refine when value is less or more than rvalue
  - cdecide = less or more = coarsen when value is less or more than cvalue
- combine arg = sum or min or max = how to combine child values into parent value
- cells args = Nx Ny Nz
  - Nx,Ny,Nz = refine a cell into Nx by Ny by Nz child cells
- region args = regID rflag
  - regID = ID of region that cells must be inside to be eligible for adaptation
  - rflag = all or one or center = what portion of grid cell must be inside
- dir args = Sx Sy Sz
  - Sx,Sy,Sz = vector components used with style surf to test surf elements
- file arg = filename
  - filename = name of file to write out with new parent grid info

Examples:

```
adapt_grid all refine particle 10 50
adapt_grid all coarsen particle 10 50
adapt_grid all refine coarsen particle 10 50
adapt_grid all refine surf all 0.15 iterate 1 dir 1 0 0
adapt_grid all refine coarsen value c_11 5.0 10.0 iterate 2
```

Description:

This command perform a one-time adaptation of grid cells within a grid cell group, either by refinement or coarsening or both. This command can be invoked as many times as desired, before or between simulation runs. Grid adaptation can also be performed on-the-fly during a simulation by using the *fix adapt* command.

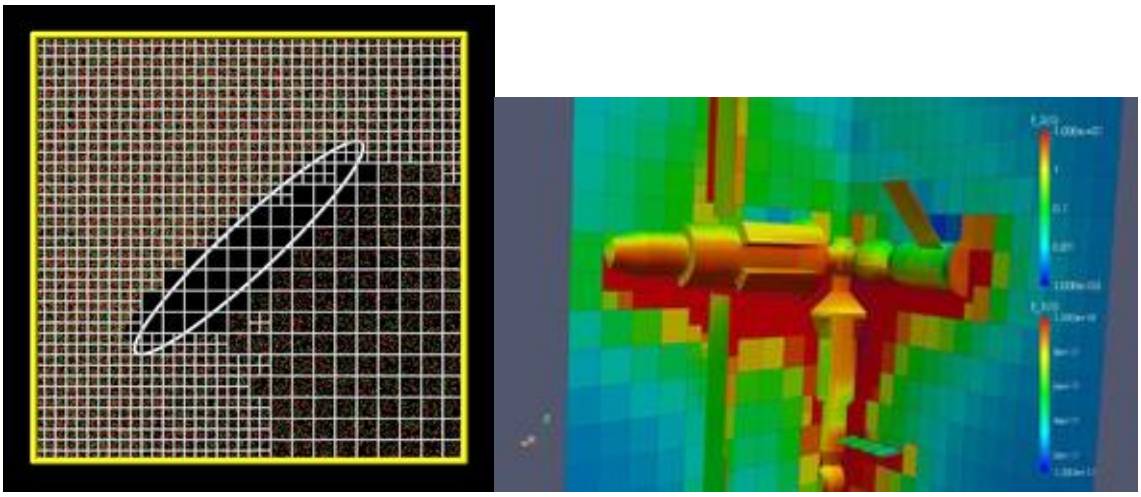
Refinement means splitting one child cell into multiple new child cells; the original child cell becomes a parent cell. Coarsening means combining all the child cells of a parent cell, so that the child cells are deleted and the parent cell becomes a single new child cell. See *Section howto 4.8* for a description of the hierarchical grid used by SPARTA and a defintion of child and parent cells.

Grid adaptation can be useful for adjusting the grid cell sizes to the current density distribution, or mean-free-path of particles, or to other simulation attributes such as the presence of surface elements. A well-adapted grid can improve accuracy of the simulation and/or reduce a simulation's computational cost.

Only grid cells in the grid group specified by *group-ID* are eligible for refinement. A parent grid cell is only eligible for coarsening if all its child cells are in the specified grid group. See the *group grid* command for info on how grid cells can be assigned to grid groups. Note that the grid group assignment is transferred to new refined or coarsened cells, so that new cells remain eligible for adaptation if the adapt_grid command is invoked again or successive adaptations are performed via the *fix adapt* command.

The *action1* and *action2* parameters determine whether refinement or coarsening is performed and in what order. *Action2* is optional. If not specified, only *action1* is performed. Note that cells which are refined by *action1* are not eligible for subsequent coarsening by *action2*, during a single invocation of this command. Likewise cells that are coarsened by *action1* are not eligible for subsequent refinement by *action2*. This is also true if the *iterate* keyword is used to loop over the two actions multiple times. Cells can be successivly refined on each iteration, but will never be coarsened. Likewise cells can be successivly coarsensed, but will never be refined. Of course any cell may be refined or coarsened later if the adapt_grid command is used again, including on later timesteps via the *fix adapt* command.

Examples of 2d and 3d refined grids are shown here. The 3d simulation shows 2d planar cuts through the 3d grid. Click on either image for a larger version.



The first step in a refinement action is to determine what child cells are eligible for refinement. Child cells that are wholly inside a closed surface are not eligible. The *maxlevel* and *region* keywords also affect eligibility. They are described below.

The first step in a coarsening action is to determine what parent cells are eligible for coarsening. Only parent cells whose children are all child cells are eligible. If one or more of their children are also parent cells, then the parent cell is a "grandparent" and is not eligible for coarsening. The *minlevel* and *region* keywords also affect eligibility. They are described below.

The *style* parameter is then used to decide whether to refine or coarsen each eligible grid cell. The operation of the differnt styles is described in the next section. Note that for refinement, the number of new child cells created withing a single cell is set by the *cells* keyword which defaults to 2x2x2 for 3d models and 2x2x1 for 2d models.

Note that many of the style take an argument for both refinement and coarsening, e.g. *rcount* and *ccount* for style *particle*. Both arguments must be specified, though one or the other will be ignored if the specified actions do not include refinement or coarsening.

The *particle* style adapts based on the number of particles in a grid cell. For refinement, if the current number (on this timestep) is more than *rcount*, the cell is refined. For coarsening, if the sum of the current number of particles in all child cells of the parent cell is less than *ccount*, the parent cell is coarsened. Note that if you wish to use time-averaged counts of particles in each cell you should use the *value* style with the ID of a *fix ave/grid* command that time-averages particle counts from the *compute grid* command.

The *surf* style adapts only if a grid cell contains one or more surface elements in the specified *surfID* group. The *dir* keyword can be used to exclude additional surface elements. For refinement, the cell is refined unless the refinement will create child cells with any of their dimensions smaller than the specified *ssize*. For coarsening, the parent cell is coarsened only if any of the child cell dimensions is smaller than the specified *ssize*.

The *value* style uses values calculated by a *compute* or *fix* to decide whether to adapt each cell. The fix or compute must calculate per-grid values as described in *Section howto 4.4*. If the compute or fix calculates a vector of such values, it is specified as c_ID or f_ID. If it calculates an array of such values, it is specified as c_ID[N] or f_ID[N] when N is the column of values to use, from 1 to Ncolumns.

For refinement, if the compute or fix value for the grid cell is "more" than *rvalue*, the cell is refined. For coarsening, if the "sum" of the compute or fix values in all child cells of the parent cell is "less" than *cvalue*, the parent cell is coarsened. The *thresh* keyword can be used to change the refinment or coarsening criteria to "less" versus "more". Likewise the *combine* keyword can be used to change the "sum" of child cell values to be a "min" or "max" operation.

Here is an example using particle count as calculated by the *compute grid* command as an adaptation criterion. A cell will be refined if its count > 25, and a parent cell coarsened if the sum of its children cell counts < 10.

```
compute 1 grid all n nrho
adapt_grid refine coarsen value c_11 25 10
```

The same thing could be accomplished with this command:

```
adapt_grid refine coarsen particle 25 10
```

These commands use a time-averaged particle count as an adaptation criterion in the same manner:

```
compute 1 grid all n nrho
fix 1 ave/grid 10 100 1000 c_11
run 1000    # run to accumulate time averages
adapt_grid refine coarsen value f_11 25 10
```

Here is an example using mean-free path (MFP) as calculated by the *compute lambda/grid* command as an adaptation criterion. Note the use of "thresh less more" to refine when MFP is less than the specified threshold (0.05).

```
compute 1 lambda/grid c_12 NULL N2 kall
adapt_grid refine coarsen value c_12 0.05 0.1 &
           combine min thresh less more
```

The *random* style is provided for test and debugging purposes. For each cell eligible for adaptation, a uniform random number RN bewteen 0.0 and 1.0 is generated. For refinement, the cell is refined if RN < *rfrac*, so that approximately an *rfrac* fraction of the child cells are refined. Similarly, for coarsening, the parent cell is coarsened if RN < *cfrac*, so that approximately a *cfrac* fraction of the parent cells are coarsened.

Various optional keywords can also be specified.

The *iterate* keyword determines how many times the *action1* and *action2* operations are looped over. The default is once. If multiple iterations are used, cells can be recursively refined or coarsened. If no further refinement or coarsening occurs on an iteration, the loop ends. Note that the compute used with style *value* will be recalculated at each iteration to accurately reflect per grid values for the current grid.

The *maxlevel* keyword limits how far a grid cell can be refined. See *Section howto 4.8* for a definition of the level assigned to each parent and child cell. Child cells with a level >= *Nmax* are not eligible for refinement. The default setting of *Nmax* = 0 means there is no limit on refinement.

The *minlevel* keyword limits how far a grid cell can be coarsened. See *Section howto 4.8* for a definition of the level assigned to each parent and child cell. Parent cells with a level < *Nmin* are not eligible for coarsening. The default

setting of *Nmin* = 1 means the only limit on coarsening is that the first level grid is preserved (never coarsened to a single root cell). The specified *Nmin* must be >= 1.

The *thresh* keyword is only used by style *value*. It sets the comparison criterion for refinement as *rdecide* = *less* or *more*. This means a child cell is refined if its compute or fix value is *less* or *more* than *rvalue*. Similarly, it sets the comparison criterion for coarsening as *cdecide* = *less* or *more*. This means a parent cell is coarsened if the compute or fix value accumulated from the compute or fix values of its children is *less* or *more* than *cvalue*.

The *combine* keyword is only used by style *value*. It determines how the compute or fix value for a parent cell is accumulated from the compute or fix values of all its children. If the setting is *sum*, the child values are summed. If it is *min* or *max*, the parent value is the minimum or maximum of all the child values.

The *cells* keyword determines how many new child cells are created when a single grid cell is refined. Nx by Ny by Nz new child cells are created. Nz must be 1 for 2d simulations. In the future we plan to allow for variable refinement by allowing wild cards to be used for Nx, Ny, and Nz.

The *region* keyword can be used to limit which grid cells are eligible for adapation. It applies to both child cells for refinment and parent cells for coarsening. The ID of the geometric region is speficied as *regID*. See the *region* command for details on what kind of geometric regions can be defined. Note that the *side* option for the *region* command can be used to define whether the inside or outside of the geometric region is considered to be "in" the region.

The grid cell must be in the region to be eligible for adaptation. The *rflag* setting determines how a grid cell is judged to be in the region or not. For *rflag* = *one*, it is in the region if any of its corner points (4 for 2d, 8 for 3d) is in the region. For *rflag* = *all*, all its corner points must be in the region. For *rflag* = *center*, the center point of the grid cell must be in the region.

The *dir* keyword is only used by the style *surf*. The Sx,Sy,Sz settings are components of a vector. It's length does not matter, just its direction. Only surface elements whose normal is opposed to the vector direction (in a dot product sense) are eligible surfaces for the adapation procedure described above for the *surf* style. This can be useful to exclude refinement around surface elements that are not facing "upwind" with respect to the flow direction of the particles. This is accomplished by setting Sx,Sy,Sz to the flow direction. If Sy,Sy,Sz = (0,0,0), which is the default, then no surface elements are excluded.

The *file* keyword triggers output of the adapted grid to the specified *filename*. The format of the file is the same as that created by the *write_grid* command, which is a list of parent cells. The file can be read in by a subsequent simulation to define a grid, or used by visualization or other post-procesing tools. Note that no file is written if no grid cells are refined or coarsened.

If the filename contains a "*" wildcard character, then the "*" is replaced by the current timestep. This is useful for the *fix adapt* command, if you wish to write out multiple grid files, each time the grid iadapts.

If the grid is partitioned across processors in a "clumped" manner before this command is invoked, it will still be clumped by processor after the adaptation. Likewise if it is not clumped before, it will remain un-clumped after adaptation. You can use the *balance_grid* command after this command to re-balance the new adapted grid cells and their particles across processors. See *Section howto 4.8* for a description clumped and unclumped grids.

Restrictions:

This command can only be used after the grid has been created by the *create_grid*, *read_grid*, or *read_restart* commands.

Currently a fix cannot be used with style *value* for *iterate* > 1. This is because the per-grid cell values accumulated by the fix are not interpolated to new grid cells so that the fix can be re-evaluated multiple times. In the future we may revove this restriction.

Related commands:

fix adapt command, *balance_grid command*

Default:

The keyword defaults are iterate = 1, minlevel = 1, maxlevel = 0, thresh = more for rdecide and less for cdecide, combine = sum, cells = 2 2 2 for 3d and 2 2 1 for 2d, no region, dir = 0 0 0, and no file.

1.14.2 balance_grid command

Syntax:

```
balance_grid style args ...
```

- style = `none` or `stride` or `clump` or `block` or `random` or `proc` or `rcb`
  - none args = none
  - stride args = xyz or xzy or yxz or yzx or zxy or zyx
  - clump args = xyz or xzy or yxz or yzx or zxy or zyx
  - block args = Px Py Pz: # of processors in each dimension
  - random args = none
  - proc args = none
  - rcb args = weight: or part or time
- zero or more keyword/value(s) pairs may be appended

  keyword = `axes` or `flip`
  - axes value = dims: string with any of "x", "y", or "z" characters in it
  - flip value = yes or no

Examples:

```
balance_grid block * * *
balance_grid block * 4 *
balance_grid clump yxz
balance_grid random
balance_grid rcb part
balance_grid rcb part axes xz
```

Description:

This command adjusts the assignment of grid cells and their particles to processors, to attempt to balance the computational cost (load) evenly across processors. The load balancing is "static" in the sense that this command performs the balancing once, before or between simulations. The assignments will remain static during the subsequent run. To perform "dynamic" balancing, see the *fix balance* command, which can adjust the assignemt of grid cells to processors on-the-fly during a run.

After grid cells have been assigned, they are migrated to new owning processors, along with any particles they own or other per-cell attributes stored by fixes. The internal data structures within SPARTA for grid cells and particles are re-initialized with the new decomposition.

This command can be used immediately after the grid is created, via the *create_grid* or *read_restart* commands. In the former case balance_grid can be used to partition the grid in a more desirable manner than the default creation options allow for. In the latter case, balance grid can be used to change the somewhat random assignment of grid cells to processors that will be made if the restart file is read by a different number of processors than it was written by.

This command can also be used once particles have been created, or a simulation has come to equilibrium with a spatially varying density distribution of particles, so that the computational load is more evenly balanced across processors.

The details of how child cells are assigned to processors by the various options of this command are described below. The cells assigned to each processor will either be "clumped" or "dispersed".

The *clump* and *block* and *rcb* styles will produce clumped assignments of child cells to each processor. This means each processor's cells will be geometrically compact. The *stride* and *random* and *proc* styles will produce dispersed assignments of child cells to each processor.

Important: See *Section 6.8* of the manual for an explanation of clumped and dispersed grid cell assignments and their relative performance trade-offs.

The *none* style will not change the assignment of grid cells to processors. However it will update the internal data structures within SPARTA that store ghost cell information on each processor for cells owned by other processors. This is useful if the *global gridcut* command was used after grid cells were already defined. That command erases ghost cell information stored by processors, which then needs to be re-generated before a simulation is run. Using the balance_grid none command will re-generate the ghost cell information.

The *stride*, *clump*, and *block* styles can only be used if the grid is "uniform". The grid in SPARTA is hierarchical with one or more levels, as defined by the *create_grid* or *read_grid* commlands. If the parent cell of every grid cell is at the same level of the hierarchy, then for puposed os this command the grid is uniform, meaning the collection of grid cells effectively form a uniform fine grid overlaying the entire simulation domain.

The meaning of the *stride*, *clump*, and *block* styles is exactly the same as when they are used as keywords with the *create_grid* command. See its doc page for details.

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

The *proc* style means that each processor will choose a random processor to assign its first grid cell to. It will then loop over its grid cells and assign each to consecutive processors, wrapping around the enumeration of processors if necessary. Note that in this case every processor will typically not be assigned exactly the same number of cells.

The *rcb* style uses a recursive coordinate bisectioning (RCB) algorithm to assign spatially-compact clumps of grid cells to processors. Each grid cell has a "weight" in this algorithm so that each processor is assigned an equal total weight of grid cells, as nearly as possible.

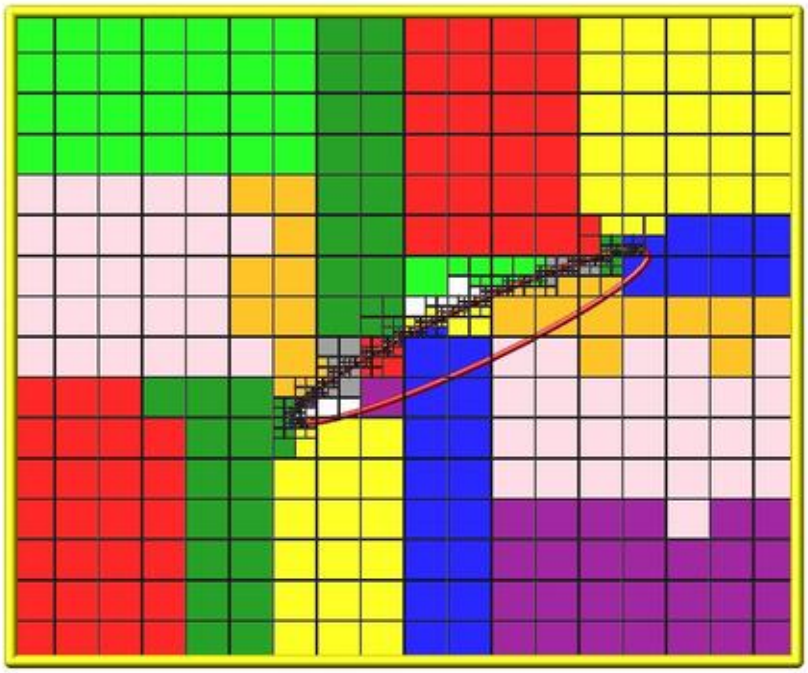
If the *weight* argument is specified as *cell*, then the weight for each grid cell is 1.0, so that each processor will end up with an equal number of grid cells.

If the *weight* argument is specified as *part*, then the weight for each grid cell is the number of particles it currently owns, so that each processor will end up with an equal number of particles.

If the *weight* argument is specified as *time*, then timers are used to estimate the cost of each grid cell. The cost from the timers is given on a per processor basis, and then assigned to grid cells by weighting by the relative number of particles in the grid cells. If no timing data has yet been collected at the point in a script where this command is issued,

a *cell* style weight will be used instead of *time*. A small warmup run (for example 100 timesteps) can be used before the balance command so that timer data is available. The timers used for balancing tally time from the move, sort, collide, and modify portions of each timestep.

Here is an example of an RCB partitioning for 24 processors, of a 2d hierarchical grid with 5 levels, refined around a tilted ellipsoidal surface object (outlined in pink). This is for a *weight cell* setting, yielding an equal number of grid cells per processor. Each processor is assigned a different color of grid cells. (Note that less colors than processors were used, so the disjoint yellow cells actually belong to three different processors). This is an example of a clumped distribution where each processor's assigned cells can be compactly bounded by a rectangle. Click for a larger version of the image.



The optional keywords *axes* and *flip* only apply to the *rcb* style. Otherwise they are ignored.

The *axes* keyword allows limiting the partitioning created by the RCB algorithm to a subset of dimensions. The default is to allow cuts in all dimension, e.g. x,y,z for 3d simulations. The dims value is a string with 1, 2, or 3 characters. The characters must be one of "x", "y", or "z". They can be in any order and must be unique. For example, in 3d, a dims = xz would only partition the 3d grid only in the x and z dimensions.

The *flip* keyword is useful for debugging. If it is set to *yes* then each time an RCB partitioning is done, the coordinates of grid cells will (internally only) undergo a sign flip to insure that the new owner of each grid cell is a different processor than the previous owner, at least when more than a few processors are used. This will insure all particle and grid data moves to new processors, fully exercising the rebalancing code.

Restrictions:

This command can only be used after the grid has been created by the *create_grid*, *read_grid*, or *read_restart* <*command-read-restart*> commands.

This command also initializes various options in SPARTA before performing the balancing. This is so that grid cells are ready to migrate to new processors. Thus if an error is flagged, e.g. that a simulation box boundary condition is not yet assigned, that operation needs to be performed in the input script before balancing can be performed.

Related commands:

fix balance command - fix balance/kk command

Default:

The default settings for the optional keywords are axes = xyz, flip = no.

1.14.3 bound_modify command

Syntax:

```
bound_modify wall1 wall2 ... keyword value ...
```

- wall1,wall2,... = *xlo* or *xhi* or *ylo* or *yhi* or *zlo* or *zhi*
- one or more keyword/value pairs may be listed

  keywords = collide or react

  - collide value = sc-ID: ID of a surface collision model
  - react value = sr-ID: ID of a surface reaction model or none

Examples:

```
bound_modify yhi collide 1 react 2
bound_modify zlo zhi collide hotwall
```

Description:

Set parameters for one or more of the boundaries of the global simulation box. Any of the 6 faces can be selected via the list of *wall* settings.

The *collide* 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. The ID of the surface collision model is specified as *sc-ID*, which is the ID used in the *surf_collide* command.

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

The *react* 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 reaction model to the boundary, as defined by the *surf_react* command. The ID of the surface reaction model is specified as *sr-ID*, which is the ID used in the *surf_react* command. If an sr-ID of *none* is used then surface reactions are turned off.

The effect of this keyword is that particle collisions with the specified boundaries will induce reactions which are computed by the specified surface reaction model.

Restrictions:

For 2d simulations, the *zlo* and *zhi* boundaries cannot be modified by this command, since they are always periodic.

All boundaries of type "s" must be assigned to a surface collision model via the *collide* keyword before a simlulation can be performed. Using a surface reaction model is optional.

Related commands:

boundary command surf_modify command

Default:

The default for boundary reactions is none.

1.14.4 boundary command

Syntax:

```
boundary x y z
```

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

```
o is outflow
p is periodic
r is specular reflection
a is axi-symmetric
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 in that dimension. 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 *a* means an axi-symmetric boundary, which can only be used for the lower y-dimension boundary in a 2d simulation. The simulation box must also have a value of 0.0 for *ylo*; see the *create_box* command. This effectively means that the x-axis is the axis of symmetry. The upper y-dimension boundary cannot be periodic.

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 the boundary to the model is done via the *bound_modify* command.

Restrictions:

This command must be used before the grid is defined, e.g. by a *create_grid* command.

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

Related commands:

bound_modify command surf_collide command

Default:

```
boundary p p p
```

1.14.5 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 almost 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

1.14.6 collide command

Syntax:

```
collide style args keyword value ...
```

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

  none args = none No argument is passed

  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

  vss/kk args = mix-ID file

  - mix-ID = ID of mixture to use for group definitions
  - file = filename that lists species with their VSS model parameters

- zero or more keyword/value pairs may be appended
- keyword = *relax*

  ```
  relax value = constant or variable
  ```

Examples:

```
collide none
collide vss all ../data/air.vss
collide vss species all.vss relax variable
```

Description:

Define what style of particle-particle collisions will be performed by SPARTA each timestep. 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. Collisions alter the velocity of participating particles as well as their rotational and vibrational energies. The rotational and vibrational properties of each species are set in the file read by the *species* command.

The collision style determines how many pairs of particles are considered for collisions, the criteria for which collisions actually occurs, and the outcome of individual collision, which alters the velocities of the two particles. If chemistry is enabled, via the *react* command, particles involved in collisions may also change species, or a particle may be

deleted, or a new particle created. The *collide_modify* command can also be used to alter aspects of how collisions are performed. For example, it can be used to turn on/off the tracking of vibrational energy and its exchange in collisions.

A *mix-ID* argument is specified for each collision style. It must contain all the species defined for use by the simulation, via the *species* command. The group definitions in the mixture assign one or more particle species to each group. These groupings are used to determine how pairs of particles are chosen to collide with each other, in the following manner.

Consider a cell with N particles and a mixture with M groups. Based on its species, each particle is assigned to one of the M groups. Each unique pair of groups is considered, including each group paired with itself. For each pair of groups a value *Nattempt* (see equation 11.3 in [Bird94]) is calculated which is the number of collisions to attempt. This is a function of N1 and N2 (the number of particles in each group), the grid cell volume, and other parameters of the collision style.

For each collision attempt, a random pair of particles is selected, with one particle from each group. Whether the collision occurs or not is a function of the relative velocities of the two particles, their respectives species, and other parameters of the collision style; see equation 11.4 in [Bird94].

Note: If you are using the ambipolar approximation with charged species, as described in *Using the ambipolar approximation*, and you have used the *collide_modify ambipolar yes* command to enable ambipolar collisions (not required), and you are using a mixture ID with multiple groups, then the ambipolar electron species must be in a group by itself.

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

The *vss* style implements the Variable Soft Sphere (VSS) model for collisions. As discussed below, with appropriate parameter choices, it can also compute the Variable Hard Sphere (VHS) model and the Hard Sphere (HS) model. See chapters 2.6 and 2.7 in [Bird94] for details.

In DSMC, the variable-soft-sphere (VSS) interaction of Koura and Matsumoto [Koura92] and the variable-hard-sphere (VHS) interaction of [Bird94] are used to approximate molecular interactions. Both models yield transport properties proportional to a power (omega) of the gas temperature. This temperature dependence of the transport properties is similar to the Inverse Power Law model (IPL) for which Chapman-Enskog theory provides closed form solutions for the transport properties.

Both VSS and VHS interactions define parameters *diam* = molecular diameter, which is a function of the molecular speed, and *alpha* = angular-scattering parameter, which relates the scattering angle to the impact parameter. Setting *alpha* = 1 produces isotropic (hard sphere) interactions, which converts the VSS model into a VHS model.

The *file* argument is for a collision data file which contains definitions of VSS model parameters for some number of species. Example files are included in the data directory of the SPARTA distribution, with a "*.css" suffix. The file can contain species not used by this simulation; they will simply be ignored. All species currently defined by the simulation must be present in the file.

The format of the file depends of the setting of the optional *relax* keyword, as explained below. Comments or blank lines are allowed in the file. Comment lines start with a "#" character. All other lines must have the following format with parameters separated by whitespace.

If the *relax* keyword is specified as *constant*, which is the default, then each line has 4 parameters following the species ID:

```
species-ID diam omega tref alpha
```

The species-ID is a string that will be matched to one of the species defined by the simulation, via the *species* command. The meaning of additional properties is as follows:

- diam = VHS or VSS diameter of particle (distance units)
- omega = temperature-dependence of viscosity (unitless)
- tref = reference temperature (temperature units)
- alpha = angular scattering parameter (unitless)

The methodolgy for deriving VSS/VHS parameters from these properties is explained in Chapter 3 of [Bird94]. Parameter values for the most common gases are given in Appendix A of the same book. These values are based on the first-order approximation of the Chapman-Enskog theory. Infinite-order parameters are described in [Gallis04].

In the *constant* case rotational and vibrational relaxation during a collision is treated in the same constant manner for every collision, using the rotational and vibrational relaxation numbers from the species data file, as read by the *species* command.

If the *relax* keyword is specified as *variable*, then each line has 8 parameters following the species ID:

```
species-ID diam omega tref alpha Zrotinf T* C1 C2
```

The first 4 parameters are the same as above. Parameters 5 and 6 affect rotational relaxation; parameters 7 and 8 affect vibrational relaxation. In this case the rotational and vibrational relaxation during a collision is treated as a variable and is computed for each collision. This calculation is only performed for polyatomic species, using equations A5 and A6 on pages 413 and 414 in [Bird94]. Zrotinf and T* are parameters in the numerator and denominator of eq A5. C1 and C2 are in eq A6. The units of these parameters is as follows:

- Zrotinf (unitless)
- T* (temperature units)
- C1 (temperature units)
- C2 (temperature^(1/3) units)

Note that a collision data file with the 4 extra relaxation parameters (per species) can be used when the *relax* keyword is specified as *constant*. In that case, the extra parameters are simply ignored.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

none

Related commands:

collide_modify command, *mixture command*, *react command*

Default:

Style = none is the default (no collisions). If the vss style is specified, then relax = constant is the default.

1.14.7 collide_modify command

Syntax:

```
collide_modify keyword values ...
```

one or more keyword/value pairs may be listed

- keywords = *vremax* or *remain* or *ambipolar* or *nearcp* or *rotate* or *vibrate*
  - vremax values = Nevery startflag
    - Nevery = zero vremax every this many timesteps
    - startflag = yes or no = zero vremax at start of every run
  - remain value = yes or no = hold remaining fraction of collisions over to next timestep
  - nearcp values = choice Nlimit
    - choice = yes or no to turn on/off near collision partners
    - Nlimit = max # of attempts made to find a collision partner
  - ambipolar value = no or yes
  - rotate value = no or smooth
  - vibrate value = no or smooth or discrete

Examples:

```
collide_modify vremax 1000 yes
collide_modify vremax 0 no remain no
collide_modify ambipolar yes
```

Description:

Set parameters that affect how collisions are performed.

The *vremax* keyword affects how often the Vremax parameter, for collision frequency is re-zeroed during the simulation. This parameter is stored for each grid cell and each pair of collision groups (groups are described by the *collide* command).

The value of Vremax affects how many events are attempted in each grid cell for a pair of groups, and thus the overall time spent performing collisions. Vremax is continuously set to the largest difference in velocity between a pair of colliding particles. The larger Vremax grows, the more collisions are attempted for the grid cell on each timestep, though this does not affect the number of collisions actually performed. Thus if Vremax grows large, collisions become less efficient, though still accurate.

For non-equilibrium flows, it is typically desirable to reset Vremax to zero fairly frequently (e.g. every 1000 steps) so that it does not become large, due to anomolously fast moving particles. In contrast, when a system is at equilibrium, it is typically desirable to not reset Vremax to zero since it will also stay roughly constant.

If *Nevery* is specified as 0, Vremax is not zeroed during a run. Otherwise Vremax is zeroed on timesteps that are a multiple of *Nevery*. Additionally, if *startflag* is set to *yes*, Vremax is zeroed at the start of every run. If it is set to *no*, it is not.

The *remain* keyword affects how the number of attempted collisions for each grid cell is calculated each timestep. If the value is set to *yes*, then any fractional collision count (for each grid cell and pair of grgroups) is carried over to the next timestep. E.g. if the computed collision count is 7.3, then 7 attempts are made on this timestep, and 0.3 are carried over to the next timestep, to be added to the computed collision count for that step. If the value is set to *no*, then no carry-over is made. Instead, in this example, 7 attempts are made and an 8th attempt is made conditionally with a probability of 0.3, using a random number.

The *nearcp* keyword stands for "near collision partner" and affects how collision partners are selected. If *no* is specified, which is the default, then collision partner pairs are selected randomly from all particles in the grid cell. In this case the *Nlimit* parameter is ignored, though it must still be specfied.

If *yes* is specified, then up to *Nlimit* collision partners are considered for each collision. The first partner I is chosen randomly from all particles in the grid cell. A distance R that particle I moves in that timestep is calculated, based on its velocity. *Nlimit* possible collision partners J are examined, starting at a random J. If one of them is within a distance R of particle I, it is immediately selected as the collision partner. If none of the *Nlimit* particles are within a distance R, the closest J particle to I is selected. An exception to these rules is that a particle J is not considered for a collision if the I,J pair were the most recent collision partners (in the current timestep) for each other. The convergence properties of this near-neighbor algorithm are described in [Gallis11]. Note that choosing *Nlimit* judiciously will avoid costly searches when there are large numbers of particles in some or all grid cells.

If the *ambipolar* keyword is set to *yes*, then collisions within a grid cell with use the ambipolar approximation. This requires use of the *fix ambipolar* command to define which species is an electron and which species are ions. There can be many of the latter. When collisions within a single grid cell are performed, each ambipolar ion is split into two particles, the ion and an associated electron. Collisions between the augmented set of particles are calculated. Ion/electron chemistry can also occur if the *react* command has been used to read a file of reactions that include such reactions. See the *react* command doc page. After all collisions in the grid cell have been computed, there is still a one-to-one correspondence between ambipolar ions and electron, and each pair is recombined into a single ambipolar particle.

The *rotate* keyword determines how rotational energy is treated in particle collisions and stored by particles. If the value is set to *no*, then rotational energy is not tracked; every particle's rotational energy is 0.0. If the value is set to *smooth*, a particle's rotational energy is a single continuous value.

The *vibrate* keyword determines how vibrational energy is treated in particle collisions and stored by particles. If the value is set to *no*, then vibrational energy is not tracked; every particle's vibrational energy is 0.0. If the value is set to *smooth*, a particle's vibrational energy is a single continuous value. If the value is set to *discrete*, each particle's vibrational energy is set to discrete values, namely multiples of kT where k = the Boltzmann constant and T is one or more characteristic vibrational temperatures set for the particle species.

Note that in the *discrete* case, if any species are defined that have 4,6,8 vibrational degrees of freedom, which correspond to 2,3,4 vibrational modes, then the *species* command must be used with its optional *vibfile* keyword to set the vibrational info (temperature, relaxation number, degeneracy) for those species.

Also note that if any such species are defined (with more than one vibrational mode, then use of the *discrete* option also requires the *fix vibmode* command be used to allocate storage for the per-particle mode values.

Restrictions:

none

Related commands:

collide command

Default:

The option defaults are vremax = (0,yes), remain = yes, ambipolar no, nearcp no, rotate smooth, and vibrate = no.

1.14.8 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 ke/particle
compute myGrid all n mass u usq 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. Examples include calculation of the system temperature or counting collisions of particles with surface elements. Code for new computes can be added to SPARTA; see *Section 10* of the manual for details.

Note that defining a compute does not perform a computation. Instead computes are invoked by other SPARTA commands as needed, e.g. to generate statistics or dump file output. See *Section 4.4* for a summary of various SPARTA output options, many of which involve computes.

The ID for a compute is used to identify the compute in other commands. Each compute ID must be unique. The ID can only contain alphanumeric characters and underscores. You can specify multiple computees of the same style so long as they have different IDs. A compute can be deleted with the *uncompute* command, after which its ID can be re-used.

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:

- *boundary* - various quantities on each global boundary

- *count* - particle counts for species and mixtures and mixture groups
- *distsurf/grid* - distance from grid cells to surface
- *eflux/grid* - energy flux density per grid cell
- *fft/grid* - FFTs across grid cells
- *grid* - various per grid cell quantities
- *isurf/grid* - various implicit surface element quantities
- *ke/particle* - temperature per particle
- *lambda/grid* - mean-free path per grid cell
- *pflux/grid* - momentum flux density per grid cell
- *property/grid* - per grid cell properties
- *react/boundary* - reaction stats on global boundary
- *react/surf* = reaction stats for explicit surfs
- *react/isurf/grid* - reactions stats for implicit surfs
- *reduce* - reduce vectors to scalars
- *sonine/grid* - Sonine moments per grid cell
- *surf* - various explicit surface element quantities
- *thermal/grid* - thermal temperature per grid cell
- *temp* - temperature of particles
- *tvib/grid* - vibrational temperature per grid cell

There are also additional accelerated compute styles included in the SPARTA distribution for faster performance on specific hardware. The list of these with links to the individual styles are given in the pair section of this page.

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

Global, per-particle, per-grid, and per-surf quantities each come in two forms: a single scalar value or a vector of values. Additionaly, global quantities can also be a 2d array of values. The doc page for each compute describes the style and kind of values it produces, e.g. a per-particle vector. Some computes can produce more than one form 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 f_ID even if it produces, for example, both a scalar and vector. The doc pages for various commands explain the details.

The values generated by a compute 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-particle values can be output via the *dump particle* command. Or the values can be referenced in a *particle-style variable*.
- Per-grid values can be output via the *dump grid* command. They can be time-averaged via the *fix ave/grid* command.
- Per-surf values can be output via the *dump surf* command. They can be time-averaged via the *fix ave/surf* command.

Restrictions:

none

Related commands:

uncompute command

Default:

none

1.14.9 compute boundary command

Syntax:

```
compute ID boundary mix-ID value1 value2 ...
```

```
compute ID boundary/kk mix-ID value1 value2 ...
```

- `ID` is documented in *compute* command
- `boundary` = style name of this compute command
- `mix-ID` = mixture ID to perform calculation on
- one or more values can be appended

  `value` = `n` or `nwt` or `mflux` or `press` or `shx` or `shy` or `shz` or `ke` or `erot` or `evib` or `etot`

  - `n` = count of particles hitting boundary
  - `nwt` = weighted count of particles hitting boundary

- `mflux` = flux of mass on boundary
- `press` = magnitude of normal pressure on boundary
- `shx,shy,shz` = components of shear stress on boundary
- `ke` = flux of particle kinetic energy on boundary
- `erot` = flux of particle rotational energy on boundary
- `evib` = flux of particle vibrational energy on boundary
- `etot` = flux of particle total energy on boundary

Examples:

```
compute 1 boundary all n press eng
compute mine boundary species press shx shy shz
```

These commands will print values for the current timestep for the xlo and xhi boundaries, as part of statistical output:

```
compute 1 boundary all n press
stats_style step np c_1[1][1] c_1[1][2] c_1[2][1] c_1[2][2]
```

These commands will dump time averages for each species and each boundary to a file every 1000 steps:

```
compute 1 boundary species n press shx shy shz
fix 1 ave/time 10 100 1000 c_1[*] mode vector file tmp.boundary
```

Description:

Define a computation that calculates one or more values for each boundary (i.e. face) of the simulation box, based on the particles that cross or collide with the boundary. 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.

Note that depending on the settings for the *boundary* command, when a particle collides with a boundary, it can exit the simulation box (outflow), re-enter from the other side (periodic), reflect specularly from the boundary, or interact with it as if it were a surface. In the surface case, the incident particle may bounce off (possibly as a different species), be captured by the boundary (vanish), or a 2nd particle can also be emitted. The formulas below account for all these possible scenarios. As an example, the pressure exerted on an outflow boundary versus a specularly reflecting boundary is different, since in the former case there is no net momentum flux back into the simulation box by reflected particles.

Also note that all values for a boundary collision are tallied based on the species group of the incident particle. Quantities associated with outgoing particles are part of the same tally, even if they are in different species groups.

The results of this compute can be used by different commands in different ways. The values for a single timestep can be output by the *stats_style* command.

The values over many sampling timesteps can be averaged by the *fix ave/time* command. It does its averaging as if the particles striking the boundary at each sampling timestep were combined together into one large set to compute the formulas below. The answer is then divided by the number of sampling timesteps if it is not otherwise normalized by the number of particles. Note that in general this is a different normalization than taking the values produced by the formulas below for a single timestep, summing them over the sampling timesteps, and then dividing by the number of sampling steps. However for the current values listed below, the two normalization methods are the same.

Note: If particle weighting is enabled via the *global weight* command, then all of the values below are scaled by the weight assigned to the grid cell in which the particle collision with the boundary occurs. The only exception is the the *n* value, which is NOT scaled by the weight; it is a simple count of particle crossings or collisions with the boundary.

The *n* value counts the number of particles in the group crossing or colliding with the boundary.

The *nwt* value counts the number of particles in the group crossing or colliding with the boundary and weights the count by the weight assigned to the grid cell in which the particle collision with the boundary occurs. The *nwt* quantity will only be different than *n* if particle weighting is enabled via the *global weight* command.

The *mflux* value calculates the mass flux imparted to the boundary by particles in the group. This is computed as

```
Mflux = Sum_i (mass_i) / (A * dt / fnum)
```

where the sum is over all contributing particle masses, normalized by A = the area of the surface element, dt = the timestep, and fnum = the real/simulated particle ratio set by the *global fnum* command.

The *press* value calculates the pressure *P* exerted on the boundary in the normal direction by particles in the group, such that outward pressure is positive. This is computed as

```
p_delta = mass * (V_post - V_pre)
P = Sum_i (p_delta_i dot N) / (A * dt / fnum)
```

where A, dt, fnum are defined as before. P_delta is the change in momentum of a particle, whose velocity changes from V_pre to V_post when colliding with the boundary. The pressure exerted on the boundary is the sum over all contributing p_delta dotted into the normal N of the boundary which is directed into the box, normalized by A = the area of the boundary face and dt = the timestep and fnum = the real/simulated particle ratio set by the *global fnum* command.

The *shx*, *shy*, *shz* values calculate the shear pressure components Sx, Sy, Sz extered on the boundary in the tangential direction to its normal by particles in the group, with respect to the x, y, z coordinate axes. These are computed as

```
p_delta = mass * (V_post - V_pre)
p_delta_t = p_delta - (p_delta dot N) N
Sx = - Sum_i (p_delta_t_x) / (A * dt / fnum)
Sy = - Sum_i (p_delta_t_y) / (A * dt / fnum)
Sz = - Sum_i (p_delta_t_z) / (A * dt / fnum)
```

where p_delta, V_pre, V_post, N, A, dt, and fnum are defined as before. P_delta_t is the tangential component of the change in momentum vector p_delta of a particle. P_delta_t_x (and y,z) are its x, y, z components.

The *ke* value calculates the kinetic energy flux *Eflux* imparted to the boundary by particles in the group, such that energy lost by a particle is a positive flux. This is computed as

```
e_delta = 1/2 mass (V_post^2 - V_pre^2)
Eflux = - Sum_i (e_delta) / (A * dt / fnum)
```

where e_delta is the kinetic energy change in a particle, whose velocity changes from V_pre to V_post when colliding with the boundary. The energy flux imparted to the boundary is the sum over all contributing e_delta, normalized by A = the area of the boundary face and dt = the timestep and fnum = the real/simulated particle ratio set by the *global fnum* command.

The *erot* value calculates the rotational energy flux *Eflux* imparted to the boundary by particles in the group, such that energy lost by a particle is a positive flux. This is computed as

```
e_delta = Erot_post - Erot_pre
Eflux = - Sum_i (e_delta) / (A * dt / fnum)
```

where e_delta is the rotational energy change in a particle, whose internal rotational energy changes from Erot_pre to Erot_post when colliding with the boundary. The flux equation is the same as for the *ke* value.

The *evib* value calculates the vibrational energy flux *Eflux* imparted to the boundary by particles in the group, such that energy lost by a particle is a positive flux. This is computed as

```
e_delta = Evib_post - Evib_pre
Eflux = - Sum_i (e_delta) / (A * dt / fnum)
```

where e_delta is the vibrational energy change in a particle, whose internal vibrational energy changes from Evib_pre to Evib_post when colliding with the boundary. The flux equation is the same as for the *ke* value.

The *etot* value calculates the total energy flux imparted to the boundary by particles in the group, such that energy lost by a particle is a positive flux. This is simply the sum of kinetic, rotational, and vibrational energies. Thus the total energy flux is the sum of what is computed by the *ke*, *erot*, and *evib* values.

Output info:

This compute calculates a global array, with the number of columns 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 number of rows is 4 for a 2d simulation for the 4 faces (xlo, xhi, ylo, yhi), and it is 6 for a 3d simulation (xlo, xhi, ylo, yhi, zlo, zhi).

The array can be accessed by any command that uses global array values from a compute as input. See *Section 6.4* for an overview of SPARTA output options.

The array values will be in the *units* appropriate to the individual values as described above. *N* is unitless. *Press*, *shx*, *shy*, *shz* are in pressure units. *Ke*, *erot*, *evib*, and *etot* are in energy/area-time units for 3d simulations and energy/length-time units for 2d simulations.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

If specified with a *kk* suffix, this compute can be used no more than twice in the same input script (active at the same time).

Related commands:

fix ave/time command

Default:

none

1.14.10 compute count command

1.14.11 compute count/kk command

Syntax:

```
compute ID count id1 id2 ...
```

- ID is documented in *compute* command
- count = style name of this compute command
- id1,id2,... = species ID or mixture ID or mixture/group
  - species ID = ID used with the species command
  - mixture ID = ID used with the mixture command, expands to all groups in mixture
  - mixture/group = ID of mixture followed by name of a group within mixture

Examples:

```
compute 1 count species
compute Ncounts count N N2 N+ air/O
```

Description:

Define a computation that counts the number of particles currently in the simulation for various species or groups within mixtures. Groups are collections of one or more species within a mixture. See the "mixture" command for an explanation of how species are added to a mixture and how groups of species within the mixture are defined.

Each of the listed *ids* (id1, id2, etc) can be in one of three formats. Any of the ids can be in any of the formats.

An *id* can be a species ID, in which case the count is for particles of that species.

An *id* can be a mixture ID, in which case one count is performed for each of the groups within the mixture. In the first example above, "species" is the name of a default mixture which assigns every species defined for the simulation to its own group. If there are 10 species in the simulation, there will thus be 10 counts calculated, the same as if the command had been specified with explicit names for all 10 species, e.g.

```
compute 1 count O2 N2 O N NO O2+ N2+ O+ N+ NO+
```

An *id* can also be of the form mix-ID/name where mix-ID is a mixture ID and name is the name of a group in that mixture.

Output info:

If there is a single count accumulated, this compute calculates a global scalar. If there are multiple counts accumulated, it calculates a global vector with a length = number of counts. These results can be used by any command that uses global scalar or vector values from a compute as input. See *Section 4.4* for an overview of SPARTA output options.

The values will all be unitless counts.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

It is an error if a listed *id* is both a species ID and a mixture ID, since this command cannot distinguish between them.

Related commands:

none

Default:

none

1.14.12 compute distsurf/grid command

1.14.13 compute distsurf/grid/kk command

Syntax:

```
compute ID distsurf/grid group-ID surf-ID keyword args ...
```

- ID is documented in *compute* command
- distsurf/grid = style name of this compute command
- group-ID = group ID for which grid cells to perform calculation on
- surf-ID = group ID for which surface elements to consider
- zero or more keyword/args pairs may be appended

- keyword = *dir*
  - dir args = Sx Sy Sz: direction vector used to test surf elements

Examples:

```
compute 1 distsurf/grid all all
compute 1 distsurf/grid subset sphere2 dir 1 0 0
```

:line

Styles with a {kk} suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the "Accelerating SPARTA"_Section_accelerate.html section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the "Making SPARTA"_Section_start.html#start_3 section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the "-suffix command-line switch"_Section_start.html#start_6 when you invoke SPARTA, or you can use the "suffix"_suffix.html command in your input script.

See the "Accelerating SPARTA"_Section_accelerate.html section of the manual for more instructions on how to use the accelerated styles effectively.

Description:

Define a computation that calculates the minimum distance from each grid cell in a grid cell group to any surface element in a surface element group. This is useful for grid adaptation; the *adapt_grid* command can use the compute as a criterion for refining or coarsening individual grid cells.

Only grid cells in the grid group specified by *group-ID* are included in the calculation. See the *group grid* command for info on how grid cells can be assigned to grid groups. Only surface elements in the surface element group specified by *surf-ID* are included in the distance calculations. See the *group surf* command for info on how surface elements can be assigned to surface element groups.

If the *dir* keyword is specified it can exclude additional surface elements. The Sx,Sy,Sz settings are components of a vector. It's length does not matter, just its direction. Only surface elements whose normal is opposed to the vector direction (in a dot product sense) are eligible surfaces for the distance calculations. This can be useful to exclude surface elements that are not facing "upwind" with respect to the flow direction of the particles. I.e. by setting Sx,Sy,Sz to the flow direction. If Sy,Sy,Sz = (0,0,0), which is the default, then no surface elements are excluded by this criterion.

Each grid cell also only considers a subset of eligible surfaces in its distance calculations. A vector from the grid cell center to the center of each surface element is calculated. If that vector is opposed to the normal vector of the surface element (in a dot product sense), the distance from the grid cell to the surface is calculated. This means that for an individual grid cell, only surface elements that are "facing" the grid cell are considered.

The "distance" between a grid cell and a surface element is the minimum distance between the two geometric entities. If the surface element overlaps with the grid cell, the distance is 0.0. Otherwise the distance is the minimum distance between the perimeter of the grid cell and the line segment (in 2d) or the perimeter of the triangle (in 3d).

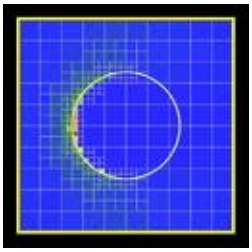
Here is an example of using this compute with the *adapt_grid* command to adapt the grid around the upwind side of a circular object (flow is from the left boundary of the box). The first adapt_grid command uses a threshold distance value of 0.5 to create refine grid cells once. The second adapt_grid command uses a threshold distance value of 0.1 to create some of the grid cells closer to the surface a second time.

Note: include pic

Here is an example of how to use this compute with two successive "adapt_grid" commands. The first refines once for grid cells within a distance of 0.3 from surface elements facing upwind. The second refines again for grid cells within a distance of 0.1 from the surface elements.

```
compute 5 distsurf/grid all dir 1 0 0
adapt_grid refine value c_5 0.3 0.0 thresh less more
adapt_grid refine value c_5 0.1 0.0 thresh less more
```

For a 2d simulation of flow around a circle (flow from right to left), these commands produce this kind of adapted grid (click for a larger image):



Output info:

This compute calculates a per-grid vector whose values are the distances of each grid cell from any of the surface elements.

This compute performs calculations for all flavors of child grid cells in the simulation, which includes unsplit, cut, split, and sub cells. See *Section 6.8* of the manual gives details of how SPARTA defines child, unsplit, split, and sub cells.

Grid cells not in the specified *group-ID* will output zeroes for all their values.

The vector can be accessed by any command that uses per-grid values from a compute as input. See *Section 6.4* for an overview of SPARTA output options.

The per-grid array values for the vector will be in distance *units*.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix command* in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

None

Related commands:

adapt_grid command

Default:

The keyword default is dir = 0,0,0.

1.14.14 compute eflux/grid command

1.14.15 compute eflux/grid/kk command

Syntax:

```
compute ID eflux/grid group-ID mix-ID value1 value2 ...
```

- ID is documented in *compute* command
- eflux/grid = style name of this compute command
- group-ID = group ID for which grid cells to perform calculation on
- mix-ID = mixture ID to perform calculation on
- one or more values can be appended
- values = `heatx` or `heaty` or `heatz`
  - heatx,heaty,heatz = xyz components of energy flux density tensor

Examples:

```
compute 1 eflux/grid all species heatx heaty heatz
compute 1 eflux/grid subset species heaty
```

These commands will dump 10 time averaged energy flux densities for each species and each grid cell to a dump file every 1000 steps:

```
compute 1 eflux/grid all species heatx heaty heatz
fix 1 ave/grid 10 100 1000 c_1[*]
dump 1 grid all 1000 tmp.grid id f_1[*]
```

Description:

Define a computation that calculates components of the energy flux density vector for each grid cell in a grid cell group. This is also called the heat flux density vector, and is based on the thermal velocity of the particles in each grid cell. The values are tallied separately for each group of species in the specified mixture, as described in the Output section below. See the mixture command for how a set of species can be partitioned into groups.

Only grid cells in the grid group specified by *group-ID* are included in the calculations. See the *group grid* command for info on how grid cells can be assigned to grid groups.

The values listed above rely on first computing and subtracting the center-of-mass (COM) velocity for all particles in the group and grid cell from each particle to yield a thermal velocity. This thermal velocity is used to compute the components of the energy flux density vector, as described below. This is in contrast to some of the values tallied by the *compute grid temp* command which simply uses the full velocity of each particle to compute a momentum or kinetic energy density. For non-streaming simulations, the two results should be similar, but for streaming flows, they will be different.

The results of this compute can be used by different commands in different ways. The values for a single timestep can be output by the *dump grid* command.

The values over many sampling timesteps can be averaged by the *fix ave/grid* command. It does its averaging as if the particles in the cell at each sampling timestep were combined together into one large set of particles to compute the formulas below.

Note that the center-of-mass (COM) velocity that is subtracted from each particle to yield a thermal velocity for each particle, as described below, is also computed over one large set of particles (across all timesteps), in contrast to using a COM velocity computed only for particles in the current timestep, which is what the *compute sonine/grid* command does.

Note that this is a different form of averaging than taking the values produced by the formulas below for a single timestep, summing those values over the sampling timesteps, and then dividing by the number of sampling steps.

Calculation of the energy flux density is done by first calcuating the center-of-mass (COM) velocity of particles for each group with a grid cell. This is done as follows:

```
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)
Cx = Vx - COMx
Cy = Vy - COMy
Cz = Vz - COMz
Csq = Cx*Cx + Cy*Cy + Cz*Cz
```

The COM velocity is (COMx,COMy,COMz). The thermal velocity of each particle is (Cx,Cy,Cz), i.e. its velocity minus the COM velocity of particles in its group and cell.

The *heatx*, *heaty*, *heatz* values compute the components of the energy flux density vector due to particles in the group as follows:

```
heatx = 0.5 * fnum/volume Sum_i (mass_i Cx Csq)
heaty = 0.5 * fnum/volume Sum_i (mass_i Cy Csq)
heatz = 0.5 * fnum/volume Sum_i (mass_i Cz Csq)
```

Note that if particle weighting is enabled via the *global weight* command, then the volume used in the formula is divided by the weight assigned to the grid cell.

Output info:

This compute calculates a per-grid array, with the number of columns 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 *momxx* and *momxy* values were specified as keywords, then the first two columns would be *momxx* and *momxy* for the first group, the 3rd and 4th columns would be *momxx* and *momxy* for the second group, etc.

This compute performs calculations for all flavors of child grid cells in the simulation, which includes unsplit, cut, split, and sub cells. See *Section 6.8* of the manual gives details of how SPARTA defines child, unsplit, split, and sub cells. Note that cells inside closed surfaces contain no particles. These could be unsplit or cut cells (if they have zero flow volume). Both of these kinds of cells will compute a zero result for all their values. Likewise, split cells store no particles and will produce a zero result. This is because their sub-cells actually contain the particles that are geometrically inside the split cell.

Grid cells not in the specified *group-ID* will output zeroes for all their values.

The array can be accessed by any command that uses per-grid values from a compute as input. See *Section 6.4* for an overview of SPARTA output options.

The per-grid array values will be in the *units* of energy flux density = energy-velocity/volume units.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

none

Related commands:

compute grid compute thermal/grid, *compute pflux/grid command*, *fix ave/grid command*, *dump grid*

Default:

none

1.14.16 compute fft/grid command

Syntax:

```
compute ID fft/grid value1 value2 ... keyword args ...
```

- ID is documented in *compute command*
- fft/grid = style name of this compute command
- one or more values can be appended

- value = c_ID, c_ID[N], f_ID, f_ID[N], v_name
  - c_ID = per-grid vector 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 calculated by a fix with ID
  - f_ID[I] = Ith column of per-grid or array calculated by a fix with ID
  - v_name = per-grid vector calculated by a grid-style variable with name
- zero or more keyword/arg pairs can be appended

  keyword = `sum` or `scale` or `conjugate` or `kmag`
- sum = yes or no to sum all FFTs into a single output
- scale = sfactor = numeric value to scale results by
- conjugate = yes or no = perform complex conjugate multiply or not
- kx = yes or no = calculate x-component of wavelength or not
- kx = yes or no = calculate y-component of wavelength or not
- kx = yes or no = calculate z-component of wavelength or not
- kmag = yes or no = calculate wavelength magnitude or not

Examples:

```
compute 1 fft/grid c_1
```

These commands will dump FFTs of instantaneous and time-averaged velocity components in each grid cell to a dump file every 1000 steps:

```
compute 1 grid all u v w
fix 1 ave/grid 10 100 1000 c_1
compute 2 fft/grid f_11 f_12 f_13
dump 1 grid all 1000 tmp.grid id c_2 f_1
```

Description:

Define a computation that performs FFTs on per-grid values. This can be useful, for example, in calculating the energy spectrum of a turbulent flow.

The defined grid must be a regular one-level grid (not hierarchical) with an even number of grid cells in each dimension. Depending on the *dimension* of the simulation, either 2d or 3d FFTs will be performed. Because FFTs assume a periodic field, the simulation domain should be periodic in all dimensions, as set by the *boundary* command, though SPARTA does not check for that.

The results of this compute can be used by different commands in different ways. The values for a single timestep can be output by the *dump grid* command. The values over many sampling timesteps can be averaged by the *fix ave/grid* command.

An FFT is performed on each input value independently.

Each listed input can be the result of a *compute* or *fix* or the evaluation of a *variable*, all of which must generate per-grid quantities.

If a value begins with c_, a compute ID must follow which has been previously defined in the input script. The compute must generate a per-grid vector or array. See the individual *compute* doc page for details. If no bracketed integer is appended, the vector calculated by the compute is used. If a bracketed integer is appended, the Ith column of the 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. The fix must generate a per-grid vector or array. See the individual *fix command* doc page for details. Note that some fixes only produce their values on certain timesteps, which must be compatible with when this compute references the values, else an error results. If no bracketed integer is appended, the vector calculated by the fix is used. If a bracketed integer is appended, the Ith column of the array calculated by the fix is used. Users can also write code for their own fix style 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. It must be a *grid-style variable*. Such a variable defines a formula which can reference stats keywords or invoke other computes, fixes, or variables when they are evaluated. So this is a very general means of creating a per-grid input to perform an FFT on.

If the *sum* keyword is set to *yes*, the results of all FFTs will be summed together, grid value by grid value, to create a single output.

The result of each FFT is scaled by the *sfactor* value of the *scale* keyword, whose default is 1.0.

If the *conjugate* keyword is set to *no*, the result of each FFT is 2 values for each grid point, the real and imaginary parts of a complex number. If the *conjugate* keyword is set to *yes*, the complex value for each grid point is multiplied by its complex conjugate to yield a single real-valued number for each grid point. Note that this value is effectively the squared length of the complex 2-vector with real and imaginary components.

If one or more of the *kx*, *ky*, *kz*, or *kmag* keywords are set to *yes*, then one or moer extra columns of per-grid output is generated. For *kx* the x-component of the K-space wavevector is generated. Similarly for *ky* and *kz*. For *kmag* the length of each K-space wavevector is generated. These values can be useful, for example, for histogramming an energy spectrum computed from the FFT of a velocity field, as a function of wavelength or a component of the wavelength.

Note that the wavevector for each grid cell is indexed as (Kx,Ky,Kz). Those indices are the x,y,z components output by the *kx*, *ky*, *kz* keywords. The total wavelength, which is output by the *kmag* keyword, is sqrt(Kx^2 + Ky^2 + Kz^2) for 3d models and sqrt(Kx^2 + Ky^2) for 2d models. For all keywords, the Kx,Ky,Kz represent distance from the origin in a periodic sense. Thus for a grid that is NxMxP, the Kx values associated with the x-dimension and used in those formulas are not Kx = 0,1,2 ... N-2,N-1. Rather they are Kx = 0,1,2, ... N/2-1, N/2, N/2-1, ... 2,1. Similary for Ky in the y-dimension with a max index of M/2, and Kz in the z-dimension with a max index of P/2.

Output info:

The number of per-grid values ouptut by this compute depends on the optional keyword settings. The number of FFTs is equal to the number of specified input values.

There are 2 columns of output per FFT if *sum* = no and *conjugate* = no, with real and imaginary components for each FFT. There is 1 column of output per FFT if *sum* = no and *conjugate* = yes. There are 2 columns of output if *sum* = yes and *conjugate* = no, with real and imaginary components for the sum of all the FFTs. There is one column of output for *sum* = yes and *conjugate* = yes. For all these cases, there is one extra column of output for each of the *kx*, *ky*, *kz*, *kmag* keywords if they are set to *yes*. The extra columns come before the FFT columns, in the order *kx*, *ky*, *kz*, *kmag*. Thus is only *ky* and *kmag* are set to yes, there will be 2 extra columns, the first for *ky* and the 2nd for *kmag*.

If the total number of output columns = 1, then this compute produces a per-grid vector as output. Otherwise it produces a per-grid array.

This compute performs calculations for all flavors of child grid cells in the simulation, which includes unsplit, cut, split, and sub cells. See *Section* of the manual gives details of how SPARTA defines child, unsplit, split, and sub cells. Note that cells inside closed surfaces contain no particles. These could be unsplit or cut cells (if they have zero flow volume). Both of these kinds of cells will compute a zero result for all their values. Likewise, split cells store no particles and will produce a zero result. This is because their sub-cells actually contain the particles that are geometrically inside the split cell.

The array can be accessed by any command that uses per-grid values from a compute as input. See *Section <howto-output>* for an overview of SPARTA output options.

The per-grid vector or array values will be in the *units* appropriate to the FFT operations as described above. The K-space wavevector magnitudes are effectively unitless, e.g. `sqrt(Kx^2 + Ky^2 + Kz^2)` where Kx,Ky,Kz are integers. The FFT values can be real or imaginary or squared values in K-space resulting from FFTs of per-grid quantities in whatever units the specified input values represent.

Restrictions:

This style is part of the FFT package. It is only enabled if SPARTA was built with that package. See the *Getting Started* section for more info.

Related commands:

fix ave/grid command, *dump command*, *compute grid command*

Default:

The option defaults are sum = no, scale = 1.0, conjugate = no, kmag = no.

1.14.17 compute grid command

1.14.18 compute grid/kk command

Syntax:

```
compute ID grid group-ID mix-ID value1 value2 ...
```

- `ID` is documented in *compute* command
- `grid` = style name of this compute command
- `group-ID` = group ID for which grid cells to perform calculation on
- `mix-ID` = mixture ID to perform calculation on
- one or more values can be appended:

  value = `n` or `nrho` or `nfrac` or `mass` or `massrho` or `massfrac` or `u` or `v` or `w` or `usq` or `vsq` or `wsq` of `ke` or `temp` or `erot` or `trot` or `evib` or `tvib` or `pxrho` or `pyrho` or `pzrho` or `kerho`

  - `n` = *particle count*
  - `nrho` = *number density*
  - `nfrac` = *number fraction*
  - `mass` = *mass*

- massrho = *mass density*
- massfrac = *mass fraction*
- 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*
- erot = *rotational energy*
- trot = *rotational temperature*
- evib = *vibrational energy*
- tvib = *vibrational temperature (classical definition)*
- pxrho = *x component of momentum density*
- pyrho = *y component of momentum density*
- pzrho = *z component of momentum density*
- kerho = *kinetic energy density*

Examples:

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

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

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

Description:

Define a computation that calculates one or more values for each grid cell in a grid cell group, based on the particles in the cell. The values are tallied separately for each group of species in the specified mixture, as described in the Ouput section below. See the *mixture* command for how a set of species can be partitioned into groups. Only grid cells in the grid group specified by *group-ID* are included in the calculations. See the *group grid* command for info on how grid cells can be assigned to grid groups.

The results of this compute can be used by different commands in different ways. The values for a single timestep can be output by the *dump grid* command.

The values over many sampling timesteps can be averaged by the *fix ave/grid* command. It does its averaging as if the particles in the cell at each sampling timestep were combined together into one large set of particles to compute the formulas below.

Note that for most of the values, this is a different form of averaging than taking the values produced by the formulas below for a single timestep, summing those values over the sampling timesteps, and then dividing by the number of sampling steps.

The `n` value counts the number of particles in each group. When accumulated over multiple sampling steps, this value is normalized by the number of sampling steps.

The `nrho` value computes the number density for the grid cell volume due to particles in each group:

```
Nrho = fnum/volume * N
```

N is the number of particles (same as the *n* keyword), fnum is the real/simulated particle ratio set by the *global fnum* command, and volume is the flow volume of the grid cell. When accumulated over multiple sampling steps, this value is normalized by the number of sampling steps. Note that if particle weighting is enabled via the *global weight* command, then the volume used in the formula is divided by the weight assigned to the grid cell.

The `nfrac` value computes the number fraction of particles in each group:

Nfrac = Ngroup / Ntotal

Ngroup is the count of particles in the group and Ntotal is the total number of particles in all groups in the mixture. Note that this total is not (necessarily) all particles in the cell.

The `mass` value computes the average mass of particles in each group:

```
Mass = Sum_i (mass_i) / N
```

where Sum_i is a sum over particles in the group.

The `massrho` value computes the mass density for the grid cell volume due to particles in each group:

```
Massrho = fnum/volume * Sum_i (mass_i)
```

where Sum_i is a sum over particles in the group, fnum is the real/simulated particle ratio set by the *global fnum* command, and volume is the flow volume of the grid cell. When accumulated over multiple sampling steps, this value is normalized by the number of sampling steps. Note that if particle weighting is enabled via the *global weight* command, then the volume used in the formula is divided by the weight assigned to the grid cell.

The `massfrac` value computes the mass fraction of particles in each group:

```
Massfrac = Sum_i (mass_i) / Masstotal
```

where Sum_i is a sum over particles in the group and Masstotal is the total mass of particles in all groups in the mixture. Note that this total is not (necessarily) the mass of all particles in the cell.

The `u`, `v`, `w` values compute the components of the mass-weighted average velocity of particles in each group:

```
U = Sum_i (mass_i Vx_i) / Sum_i (mass_i)
V = Sum_i (mass_i Vy_i) / Sum_i (mass_i)
W = Sum_i (mass_i Vz_i) / Sum_i (mass_i)
```

This is the same as the center-of-mass velocity of particles in each group.

The `usq`, `vsq`, `wsq` values compute the average mass-weighted squared components of the velocity of particles in each group:

```
Usq = Sum_i (mass_i Vx_i Vx_i) / Sum_i (mass_i)
Vsq = Sum_i (mass_i Vy_i Vy_i) / Sum_i (mass_i)
Wsq = Sum_i (mass_i Vz_i Vz_i) / Sum_i (mass_i)
```

The `ke` value computes the average kinetic energy of particles in each group:

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

Note that this is different than the group's contribution to the average kinetic energy of entire grid cells. That can be calculated by multiplying the *ke* quantity by the *n* quantity.

The `temp` value first computes the average kinetic energy of particles in each group, as for the *ke* value. This is then converted to a temperature *T* by the following formula where *kB* is the Boltzmann factor:

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

Note that this definition of temperature does not subtract out a net streaming velocity for particles in the grid cell, so it is not a thermal temperature when the particles have a non-zero streaming velocity. See the *compute thermal/grid* command to calculate thermal temperatures after subtracting out streaming components of velocity.

The `erot` value computes the average rotational energy of particles in each group:

```
Erot = Sum_i (erot_i) / N
```

Note that this is different than the group's contribution to the average rotational energy of entire grid cells. That can be calculated by multiplying the *erot* quantity by the *n* quantity.

The `trot` value computes a rotational temperature by the following formula where *kB* is the Boltzmann factor:

```
Trot = (2/kB) Sum_i (erot_i) / Sum_i (dof_i)
```

Dof_i is the number of rotational degrees of freedom for particle i.

The `evib` value computes the average vibrational energy of particles in each group:

```
Evib = Sum_i (evib_i) / N
```

Note that this is different than the group's contribution to the average vibrational energy of entire grid cells. That can be calculated by multiplying the *evib* quantity by the *n* quantity.

The `tvib` value computes a classical definition of vibrational temperature, valid for continous distributions of vibrational energy, by the following formula where *kB* is the Boltzmann factor:

```
Tvib = (2/kB) Sum_i (evib_i) / Sum_i (dof_i)
```

Dof_i is the number of vibrational degrees of freedom for particle i.

The `pxrho`, `pyrho`, `pzrho` values compute components of momentum density for the grid cell volume due to particles in each group:

```
Pxrho = fnum/volume * Sum_i (mass_i * Vx_i)
Pyrho = fnum/volume * Sum_i (mass_i * Vy_i)
Pzrho = fnum/volume * Sum_i (mass_i * Vz_i)
```

where Sum_i is a sum over particles in the group, fnum is the real/simulated particle ratio set by the *global fnum* command, and volume is the flow volume of the grid cell. When accumulated over multiple sampling steps, this value is normalized by the number of sampling steps. Note that if particle weighting is enabled via the *global weight* command, then the volume used in the formula is divided by the weight assigned to the grid cell.

The `kerho` value computes the kinetic energy density for the grid cell volume due to particles in each group:

```
Vsq = Vx*Vx + Vy*Vy + Vz*Vz
KErho = fnum/volume * Sum_i (mass_i * Vsq_i)
```

where Sum_i is a sum over particles in the group, fnum is the real/simulated particle ratio set by the *global fnum* command, and volume is the flow volume of the grid cell. When accumulated over multiple sampling steps, this value is normalized by the number of sampling steps. Note that if particle weighting is enabled via the *global weight* command, then the volume used in the formula is divided by the weight assigned to the grid cell.

Output info:

This compute calculates a per-grid array, with the number of columns 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.

This compute performs calculations for all flavors of child grid cells in the simulation, which includes unsplit, cut, split, and sub cells. See *Section 6.8* of the manual gives details of how SPARTA defines child, unsplit, split, and sub cells. Note that cells inside closed surfaces contain no particles. These could be unsplit or cut cells (if they have zero flow volume). Both of these kinds of cells will compute a zero result for all their values. Likewise, split cells store no particles and will produce a zero result. This is because their sub-cells actually contain the particles that are geometrically inside the split cell.

Grid cells not in the specified *group-ID* will output zeroes for all their values.

The array can be accessed by any command that uses per-grid values from a compute as input. See *Section 6.4* for an overview of SPARTA output options.

The per-grid array values will be in the *units* appropriate to the individual values as described above. *N* is unitless. *Nrho* is in 1/distance^3 units for 3d simulations and 1/distance^2 units for 2d simulations. *Mass* is in mass units. *Massrho* is in is in mass/distance^3 units for 3d simulations and mass/distance^2 units for 2d simulations. *U*, *v*, and *w* are in velocity units. *Usq*, *vsq*, and *wsq* are in velocity squared units. *Ke*, *erot*, and *evib* are in energy units. *Temp* and *trot* and *tvib* are in temperature units. *Pxrho*, *pyrho*, *pzrho* are in momentum/distance^3 units for 3d simulations and momentum/distance^2 units for 2d simulations, where momentum is in units of mass*velocity. *Kerho* is in units of energy/distance^3 units for 3d simulations and energy/distance^2 units for 2d simulations.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

none

Related commands:

fix ave/grid command, *command-dump grid*, *compute thermal/grid command*

Default:

none

1.14.19 compute isurf/grid command

Syntax:

```
compute ID isurf/grid group-ID mix-ID value1 value2 ...
```

- ID is documented in *compute* command
- isurf/grid = style name of this compute command
- group-ID = group ID for which grid cells to perform calculation on
- mix-ID = mixture ID for particles to perform calculation on
- one or more values can be appended
- value = `n` or `nwt` or `mflux` or `fx` or `fy` or `fz` or `press` or `px` or `py` or `pz` or `shx` or `shy` or `shz` or `ke`
  - n = count of particles hitting surface elements in a grid cell
  - nwt = weighted count of particles hitting surface elements in a grid cell
  - mflux = flux of mass on surface elements in a grid cell
  - fx,fy,fz = components of force on surface elements in a grid cell
  - press = magnitude of normal pressure on surface elements in a grid cell
  - px,py,pz = components of normal pressure on surface elements in a grid cell
  - shx,shy,shz = components of shear stress on surface elements in a grid cell
  - ke = flux of particle kinetic energy on surface elements in a grid cell
  - erot = flux of particle rotational energy on surface elements in a grid cell
  - evib = flux of particle vibrational energy on surface elements in a grid cell
  - etot = flux of particle total energy on surface elements in a grid cell

Examples:

```
compute 1 isurf/grid all all n press eng
compute mine isurf/grid sphere species press shx shy shz
```

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

```
compute 1 isurfgrid all species n press shx shy shz
fix 1 ave/grid all 10 100 1000 c_1[*]
dump 1 grid all 1000 tmp.grid id f_1[*]
```

These commands will time-average the force surface elements in each grid cell, then sum them across grid cells to compute drag (fx) and lift (fy) on the set of implicit surfs:

```
compute 1 isurf/grid all all fx fy
fix 1 ave/grid all 10 100 1000 c_1[*]
compute 2 reduce sum f_1[1] f_1[2]
stats 1000
stats_style step cpu np c_2[1] c_2[2]
```

Description:

Define a computation that calculates one or more values for each grid cell in a grid cell group, based on the particles that collide with the implicit surfaces in that 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. Only grid cells in the grid group specified by *group-ID* are included in the calculations. See the *group grid* command for info on how grid cells can be assigned to grid groups.

Implicit surface elements are triangles for 3d simulations and line segments for 2d simulations. Unlike explicit surface elements, each triangle or line segment is wholly contained within a single grid cell. See the *read_isurf* command for details.

This command can only be used for simulations with implicit surface elements. See the similar *compute surf* command for use with simulations with explicit surface elements.

Note that when a particle collides with a surface element, it can bounce off (possibly as a different species), be captured by the surface (vanish), or a 2nd particle can also be emitted. The formulas below account for all the possible outcomes. For example, the kinetic energy flux *ke* onto a suface element for a single collision includes a positive contribution from the incoming particle and negative contributions from 0, 1, or 2 outgoing particles. The exception is the *n* and *nwt* values which simply tally counts of particles colliding with the surface element.

Also note that all values for a collision are tallied based on the species group of the incident particle. Quantities associated with outgoing particles are part of the same tally, even if they are in different species groups.

The results of this compute can be used by different commands in different ways. The values for a single timestep can be output by the *dump grid* command.

The values over many sampling timesteps can be averaged by the *fix ave/grid* command. It does its averaging as if the particles striking the surface elements within the grid cell at each sampling timestep were combined together into one large set to compute the formulas below. The answer is then divided by the number of sampling timesteps if it is not otherwise normalized by the number of particles. Note that in general this is a different normalization than taking the values produced by the formulas below for a single timestep, summing them over the sampling timesteps, and then dividing by the number of sampling steps. However for the current values listed below, the two normalization methods are the same.

NOTE: If particle weighting is enabled via the *global weight* command, then all of the values below are scaled by the weight assigned to the grid cell in which the particle collision with the surface element occurs. The only exception is the the *n* value, which is NOT scaled by the weight; it is a simple count of particle collisions with surface elements in the grid cell.

The meaning of all the value keywords and the formulas for calculating these quantities is exactly the same as described by the *compute surf* command.

The only difference is that the quantities are calculated on a per grid cell basis, summing over all the surface elements in that grid cell.

Output info:

This compute calculates a per-grid array, with the number of columns 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.

Grid cells not in the specified *group-ID* will output zeroes for all their values.

The array can be accessed by any command that uses per-grid values from a compute as input. See *Section 6.4* for an overview of SPARTA output options.

The per-grid array values will be in the *units* appropriate to the individual values as described above. *N* is unitless. *Press*, *px*, *py*, *pz*, *shx*, *shy*, *shz* are in in pressure units. *Ke*, *erot*, *evib*, and *etot* are in energy/area-time units for 3d simulations and energy/length-time units for 2d simulations.

Restrictions:

none

Related commands:

fix ave/grid command dump grid, *compute surf command*

Default:

none

1.14.20 compute ke/particle command

1.14.21 compute ke/particle/kk command

Syntax:

```
compute ID ke/particle
```

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

Examples:

```
compute 1 ke/particle
```

Description:

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

The results of this compute can be used by different commands in different ways. The values for a single timestep can be output by the *dump particle* command.

The kinetic energy is

```
Vsq = Vx*Vx + Vy*Vy + Vz*Vz
KE = 1/2 m Vsq
```

where m is the mass and (Vx,Vy,Vz) are the velocity components of the particle.

Output info:

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

The vector can be accessed by any command that uses per-particle values from a compute as input. See *Output from SPARTA (stats, dumps, computes, fixes, variables)* for an overview of SPARTA output options.

The per-particle vector values will be in energy *units*.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA with optional packages* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix command* in your input script.

See the *Accelerating SPARTA performance* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

none

Related commands:

dump particle

Default:

none

1.14.22 compute lambda/grid command

1.14.23 compute lambda/grid/kk command

Syntax:

```
compute ID lambda/grid nrho temp species extra
```

- ID is documented in *compute* command
- lambda/grid = style name of this compute command
- nrho = compute or fix column for number density, prefaced by `c_` or `f_`
- temp = NULL or compute or fix column for temperature, prefaced by `c_` or `f_`
- species = species name used for reference properties
- extra = *kall* or *kx* or *ky* or *kz* (optional)

```
kall = also calculate Knudsen number based on cell size in all dimensions
kx = also calculate Knudsen number based on cell size in x dimension
ky = also calculate Knudsen number based on cell size in y dimension
kz = also calculate Knudsen number based on cell size in z dimension
```

Examples:

```
compute 1 lambda/grid c_GR[1] NULL Ar
compute 1 lambda/grid f_ave[2] f_ave[3] N2 kall
```

These commands will dump time averages for the mean free path for each grid cell to a dump file every 1000 steps:

```
compute 1 grid species nrho temp
fix 1 ave/grid 10 100 1000 c_1[*]
compute 2 lambda/grid f_1[1] f_1[2] Ar
dump 1 grid all 1000 tmp.grid id c_2
```

Description:

Define a computation that calculates the mean free path (lambda) between molecular collisions for each grid cell, based on the particles in that cell. Optionally, a Knudsen number for each cell can also be calculated, which is the mean free path divided by the cell size. These quantities can be useful for estimating the optimal grid cell size when adapting the grid, e.g. via the *adapt_grid* or *fix adapt/grid* commands.

Unlike other computes that calculate per grid cell values, this compute does not take a "group-ID" for a grid cell group as an argument, nor a particle *mixture* ID as an argument. This is because it uses the number density and temperature calculated by other computes or fixes as input, and those computes or fixes use grid group IDs or mixture IDs as part of their computations.

The results of this compute can be used by different commands in different ways. For example, the values can be output by the *dump grid* command.

The formula used to calculate the mean free path (lambda) is given in [Bird94] as equation 4.65:

$$\lambda = \{\sqrt{2}\pi D_{\mathrm{ref}}^2 n (T_{\mathrm{ref}}/T)^{\omega-1/2}\}^{-1}$$

This is an approximate mean free path for a multi-species mixture, suitable for estimating optimal grid cell sizes as explained above. It is a simplied version of formulas 4.76 and 4.77 from the same reference.

Dref and Tref and omega are collision properties for a reference species in the flow. The reference species is specified by the *species* argument. It must be a species defined by the *species* command and listed in the file of per-species collision properties read in by the *collide* command.

Specifically, Dref is the diameter of molecules of the species, Tref is the reference temperature, and omega is the viscosity temperature-dependence for the species.

In the formula above, n is the number density and T is the thermal temperature of particles in a grid cell. This compute does not calculate these quantities itself; instead it uses another compute or fix to perform the calculation. This is done by specifying the *nrho* and *temp* arguments like this:

- c_ID = compute with ID that calculates nrho/temp as a vector output
- c_ID[m] = compute with ID that calculates nrho/temp as its Mth column of array output
- f_ID[m] = fix with ID that calculates a time-averaged nrho/temp as a vector output
- f_ID[m] = fix with ID that calculates a time-averaged nrho/temp as its Mth column of array output

The *temp* argument can also be specified as NULL, which drops the (Tref/T) ratio term from the formula above. That is also effectively the case if the reference species defines omega = 1/2. In that case, the *temp* argument is ignored, whether it is NULL or not.

Note that if the value of n is 0.0 for a grid cell, its mean-free-path will be set to 1.0e20 (infinite length).

The *compute_grid* command can calculate a number density, using its *nrho* value. It can also calculate a temperature using its *temp* value. Note that this temperature is inferred from the translational kinetic energy of the particles, which is only appopriate for a mean free path calculation for systems with zero or small streaming velocities. For systems with streaming flow, an appropriate temperature can be calculated by the *compute thermal/grid* command. The formulas on its doc page show that the the center-of-mass velocity from the particles in each grid cell is subtracted from each particle's velocity to yield a translational thermal velocity, from which a thermal temperature is calculated.

The *fix ave/grid* command can calculate the same values in a time-averaged sense, assuming it uses these same computes as input. Using this fix as input to this compute will thus yield less noisy values, due to the time averaging.

Note that the compute or fix (via the compute(s) it uses as input) can perform its number density or temperature calculation for a subset of the particles based on the "mixture" it uses. See the *mixture* command for how a set of species can be partitioned into groups.

IMPORTANT NOTE: If the ID of a *fix ave/grid* command is used as the *nrho* or *temp* argument, it only produces output on timesteps that are multiples of its *Nfreq* argument. Thus this compute can only be invoked on those timesteps. For example, if a *dump grid* command invokes this compute to write values to a dump file, it must do so on timesteps that are multiples of *Nfreq*.

One of the *kall* or *kx* or *ky* or *kz* extra arguments can be optionally appended. If specified, this calculates an additional value per grid cell, namely the dimensionless Knudsen number which is the ratio of the mean free path to the cell size. For *kall*, the cell size is taken to be the average of the three grid cell side lengths (or two cell lengths for a 2d simulation). For *kx*, *ky*, or *kz*, the cell size is the single cell side length in the corresponding x,y,z dimension.

Output info:

This compute calculates a per-grid vector or array. If one of *kall*, *kx*, *ky*, or *kz* is not specified, then it is a vector. If one extra argument is specified, it is an array with two columns. The vector or first column of the array is the mean free path; the second column is the Knudsen number.

This compute performs calculations for all flavors of child grid cells in the simulation, which includes unsplit, cut, split, and sub cells. See *Section 6.8* of the manual gives details of how SPARTA defines child, unsplit, split, and sub cells. Note that cells inside closed surfaces contain no particles. These could be unsplit or cut cells (if they have zero flow volume). Both of these kinds of cells will compute a zero result for all the individual values. Likewise, split cells store no particles and will produce a zero result. This is because their sub-cells actually contain the particles that are geometrically inside the split cell.

The vector or array can be accessed by any command that uses per-grid values from a compute as input. See *Section 4.4* for an overview of SPARTA output options.

The per-grid array values for the vector or first column of the array will be in distance *units*. The second column of the array will be dimensionless.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

To use this compute, a collision style must be defined via the *collide* command, which defines properties for the reference *species*.

As explained above, to use this compute with *nrho* or *temp* defined as input from a *fix ave/grid* command, this compute must only be invoked on timesteps that are multiples of the *Nfreq* argument used by the fix, since those are the steps when it produces output.

Related commands:

compute grid command, *compute thermal/grid command*, *fix ave/grid command*, *dump grid*

Default:

none

1.14.24 compute pflux/grid command

1.14.25 compute pflux/grid/kk command

Syntax:

```
compute ID pflux/grid group-ID mix-ID value1 value2 ...
```

- `ID` is documented in *compute* command
- `pflux/grid` = style name of this compute command
- `group-ID` = group ID for which grid cells to perform calculation on
- `mix-ID` = mixture ID to perform calculation on
- one or more values can be appended

  `values` = *momxx* or *momyy* or *momzz* or *momxy* or *momyz* or *momxz*

  - `momxx,momyy,momzz` = diagonal components of momentum flux density tensor
  - `momxy,momyz,momxz` = off-diagonal components of momentum flux density tensor

Examples:

```
compute 1 pflux/grid all species momxx momyy momzz
compute 1 pflux/grid subset species momxx momxy
```

These commands will dump 10 time averaged momentum flux densities for each species and each grid cell to a dump file every 1000 steps:

```
compute 1 pflux/grid all species momxx momyy momzz
fix 1 ave/grid 10 100 1000 c_1[*]
dump 1 grid all 1000 tmp.grid id f_1[*]
```

Description:

Define a computation that calculates components of the momemtum flux density tensor for each grid cell in a grid cell group. This is equivalent to the kinetic energy density tensor, and is based on the thermal velocity of the particles in each grid cell. The values are tallied separately for each group of species in the specified mixture, as described in the Output section below. See the mixture command for how a set of species can be partitioned into groups.

Only grid cells in the grid group specified by *group-ID* are included in the calculations. See the *group grid* command for info on how grid cells can be assigned to grid groups.

The values listed above rely on first computing and subtracting the center-of-mass (COM) velocity for all particles in the group and grid cell from each particle to yield a thermal velocity. This thermal velocity is used to compute the components of the momentum flux density tensor, as described below. This is in contrast to some of the values tallied by the *compute grid temp* command which simply uses the full velocity of each particle to compute a momentum or kinetic energy density. For non-streaming simulations, the two results should be similar, but for streaming flows, they will be different.

The results of this compute can be used by different commands in different ways. The values for a single timestep can be output by the *dump grid* command.

The values over many sampling timesteps can be averaged by the *fix ave/grid* command. It does its averaging as if the particles in the cell at each sampling timestep were combined together into one large set of particles to compute the formulas below.

Note that the center-of-mass (COM) velocity that is subtracted from each particle to yield a thermal velocity for each particle, as described below, is also computed over one large set of particles (across all timesteps), in contrast to using

a COM velocity computed only for particles in the current timestep, which is what the *compute sonine/grid* command does.

Note that this is a different form of averaging than taking the values produced by the formulas below for a single timestep, summing those values over the sampling timesteps, and then dividing by the number of sampling steps.

Calculation of the momentum flux density is done by first calculating the center-of-mass (COM) velocity of particles for each group within a grid cell. This is done as follows:

```
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)
Cx = Vx - COMx
Cy = Vy - COMy
Cz = Vz - COMz
```

The COM velocity is (COMx,COMy,COMz). The thermal velocity of each particle is (Cx,Cy,Cz), i.e. its velocity minus the COM velocity of particles in its group and cell.

The *momxx*, *momyy*, *momzz* values compute the diagonal components of the momentum flux density tensor due to particles in the group as follows:

```
momxx = fnum/volume Sum_i (mass_i Cx^2)
momyy = fnum/volume Sum_i (mass_i Cy^2)
momzz = fnum/volume Sum_i (mass_i Cz^2)
```

The *momxy*, *momyz*, *momxz* values compute the off-diagonal components of the momentum flux density tensor due to particles in the group as follows:

```
momxy = fnum/volume Sum_i (mass_i Cx Cy)
momyz = fnum/volume Sum_i (mass_i Cy Cz)
momxz = fnum/volume Sum_i (mass_i Cx Cz)
```

Note that if particle weighting is enabled via the *global weight* command, then the volume used in the formula is divided by the weight assigned to the grid cell.

Output info:

This compute calculates a per-grid array, with the number of columns 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 *momxx* and *momxy* values were specified as keywords, then the first two columns would be *momxx* and *momxy* for the first group, the 3rd and 4th columns would be *momxx* and *momxy* for the second group, etc.

This compute performs calculations for all flavors of child grid cells in the simulation, which includes unsplit, cut, split, and sub cells. See *Section 6.8* of the manual gives details of how SPARTA defines child, unsplit, split, and sub cells. Note that cells inside closed surfaces contain no particles. These could be unsplit or cut cells (if they have zero flow volume). Both of these kinds of cells will compute a zero result for all their values. Likewise, split cells store no particles and will produce a zero result. This is because their sub-cells actually contain the particles that are geometrically inside the split cell.

Grid cells not in the specified *group-ID* will output zeroes for all their values.

The array can be accessed by any command that uses per-grid values from a compute as input. See *Section 6.4* for an overview of SPARTA output options.

The per-grid array values will be in the *units* of momentum flux density = energy density = energy/volume units.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

none

Related commands:

compute grid command, *compute thermal/grid command*, *compute eflux/grid command*, *fix ave/grid command*, *dump grid*

Default:

none

1.14.26 compute property/grid command

1.14.27 compute property/grid/kk command

Syntax:

```
compute ID property/grid group-ID input1 input2 ...
```

- ID is documented in *compute* command
- property/grid = style name of this compute command
- group-ID = group ID for which grid cells to perform calculation on
- input = one or more grid attributes

  possible attributes = id, proc, xlo, ylo, zlo, xhi, yhi, zhi, xc, yc, zc

  - id = integer form of 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
- xc,yc,zc = coords of center of grid cell
- vol = flow volume of grid cell (area in 2d)

Examples:

```
compute 1 property/grid all id xc yc zc
```

Description:

Define a computation that simply stores grid attributes for each grid cell in a grid cell group. This is useful so that the values can be used by other *output commands* that take computes as inputs. See for example, the *compute reduce*, *fix ave/grid*, and *dump grid* commands.

Only grid cells in the grid group specified by *group-ID* are included in the calculation. See the *group grid* command for info on how grid cells can be assigned to grid groups.

The values are stored in a per-grid vector or array as discussed below.

Id is the grid cell ID. In SPARTA each grid cell is assigned a unique ID which represents its location, in a topological sense, within the hierarchical grid. This ID is stored as an integer such as 5774983, but can also be decoded into a string such as 33-4-6, which makes it easier to understand the grid hierarchy. In this case it means the grid cell is at the 3rd level of the hierarchy. Its grandparent cell was 33 at the 1st level, its parent was cell 4 (at level 2) within cell 33, and the cell itself is cell 6 (at level 3) within cell 4 within cell 33. If you specify *id*, the ID is printed directly as an integer. The ID in string format can be accessed by the *dump grid* command and its *idstr* argument.

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

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

The *vol* attribute is the flow volume of the grid cell (or area in 2d). Flow volume is the portion of the grid cell that is accessible to particles, i.e. outside any closed surface that may intersect the cell.

Output info:

This compute calculates a per-grid vector or per-grid array depending on the number of input values. If a single input is specified, a per-grid vector is produced. If two or more inputs are specified, a per-grid array is produced where the number of columns = the number of inputs.

This compute performs calculations for all flavors of child grid cells in the simulation, which includes unsplit, cut, split, and sub cells. See *Section 6.8* of the manual gives details of how SPARTA defines child, unsplit, split, and sub cells. The *id* and *xlo,ylo,zlo* and *xhi,yhi,zhi* values for a split cell and its sub cells are all the same. The *vol* of a cut cell is the portion of the cell in the flow. The *vol* of a split cell is the same as if it were unsplit. The *vol* of each sub cell within a split cell is its portion of the flow volume.

Grid cells not in the specified *group-ID* will output zeroes for all their values.

The vector or array can be accessed by any command that uses per-atom values from a compute as input. See *Section 4.4* for an overview of SPARTA output options.

The vector or array values will be in whatever *units* the corresponding attribute is in, e.g. distance units for xlo or xc.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix command* in your input script.

See the "Accelerating SPARTA"_Section_accelerate.html section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

none

Related commands:

dump grid compute reduce command fix ave/grid command

Default:

none

1.14.28 compute react/boundary command

Syntax:

```
compute ID react/boundary reaction-ID value1 value2 ...
```

- ID is documented in *compute* command
- react/boundary = style name of this compute command
- reaction-ID = surface reaction ID which defines surface reactions
- zero or more values can be appended
- value = *r:s1/s2/s3* ... or *p:s1/s2/s3* ...
  - r: or p: = list of reactant species or product species
  - s1,s2,s3 = one or more species IDs, separated by "/" character

Examples:

```
surf_react air prob air.surf
compute 1 react/boundary air
compute 2 react/boundary air r:N/O/N2/O2 p:N/O/NO
```

These commands will time average the reaction tallies for each face and output the results as part of statistical output:

```
compute 2 react/boundary air r:N/O/N2/O2 p:N/O/NO
```

```
fix 1 ave/time all 10 100 1000 c_2[*]
stats_style step np f_1[1][*] f_1[2][*] f_1[3][*] f_1[4][*]
```

Description:

Define a computation that tallies counts of reactions for each boundary (i.e. face) of the simulation box, based on the particles that collide with the boundary. Only faces assigned to the surface reaction model specified by *reaction-ID* are included in the tallying.

Note that when a particle collides with a face, it can bounce off (possibly as a different species), be captured by the surface (vanish), or a 2nd particle can also be emitted.

The doc page for the *surf_react* command explains the different reactions that can occur for each specified style.

If no values are specified each reaction specified by the *surf_react* style is tallied individually for each boundary.

If M values are specified, then M tallies are made for each face, one per value. If the value starts with "r:" then any reaction which occurs with one (or more) of the listed species as a reactant is counted as part of that tally. If the value starts with "p:" then any reaction which occurs with one (or more) of the listed species as a product is counted as part of that tally. Note that these rules mean that a single reaction may be tallied multiple times depending on which values it matches.

The results of this compute can be used by different commands in different ways. The values for a single timestep can be output by the *stats_style* command. The values over many sampling timesteps can be averaged by the *fix ave/time* command.

Output info:

This compute calculates a global array, with the number of columns either equal to the number of reactions defined by the *surf_react* style (if no values are specified) or equal to M = the # of values specified. The number of rows is 4 for a 2d simulation for the 4 faces (xlo, xhi, ylo, yhi), and it is 6 for a 3d simulation (xlo, xhi, ylo, yhi, zlo, zhi).

The array can be accessed by any command that uses global array values from a compute as input. See *Section 6.4* for an overview of SPARTA output options.

The array values are counts of the number of reactions that occurred on each face.

Restrictions:

none

Related commands:

fix ave/time command, *compute react/surf command*

Default:

none

1.14.29 compute react/isurf/grid command

Syntax:

```
compute ID react/isurf/grid group-ID reaction-ID value1 value2 ...
```

- ID is documented in *compute* command
- react/isurf/grid = style name of this compute command
- group-ID = group ID for which grid cells to perform calculation on
- reaction-ID = surface reaction ID which defines surface reactions
- zero or more values can be appended
- value = *r:s1/s2/s3* . . . or *p:s1/s2/s3* . . .
  - r: or p: = list of reactant species or product species
  - s1,s2,s3 = one or more species IDs, separated by "/" character

Examples:

```
surf_react air prob air.surf
compute 1 react/isurf/grid all air
compute 2 react/isurf/grid all air r:N/O/N2/O2 p:N/O/NO
```

These commands will dump time averages for each surface element to a dump file every 1000 steps:

```
compute 2 react/isurf/grid all air r:N/O/N2/O2 p:N/O/NO
fix 1 ave/grid all 10 100 1000 c_2[*]
dump 1 grid all 1000 tmp.surgrid id f_1[*]
```

Description:

Define a computation that tallies counts of reactions for each grid cell containing implicit surface elements in a grid group, based on the particles that collide with those elements. Only grid cells elements in the grid group specified by *group-ID* are included in the tallying. See the *group grid* command for info on how grid cells can be assigned to grid groups. Likewise only grid cells with surface elements assigned to the surface reaction model specified by *reaction-ID* are included in the tallying.

Implicit surface elements are triangles for 3d simulations and line segments for 2d simulations. Unlike explicit surface elements, each triangle or line segment is wholly contained within a single grid cell. See the *read_isurf* command for details.

This command can only be used for simulations with implicit surface elements. See the similar *compute react/surf* command for use with simulations with explicit surface elements.

Note that when a particle collides with a surface element, it can bounce off (possibly as a different species), be captured by the surface (vanish), or a 2nd particle can also be emitted.

The doc page for the *surf_react* command explains the different reactions that can occur for each specified style.

If no values are specified each reaction specified by the *surf_react* style is tallied individually for each grid cell.

If M values are specified, then M tallies are made for each grid cell, one per value. If the value starts with "r:" then any reaction which occurs with one (or more) of the listed species as a reactant is counted as part of that tally. If the value starts with "p:" then any reaction which occurs with one (or more) of the listed species as a product is counted as part of that tally. Note that these rules mean that a single reaction may be tallied multiple times depending on which values it matches.

The results of this compute can be used by different commands in different ways. The values for a single timestep can be output by the *dump grid* command.

The values over many sampling timesteps can be averaged by the *fix ave/grid* command.

Output info:

This compute calculates a per-grid array, with the number of columns either equal to the number of reactions defined by the *surf_react* style (if no values are specified) or equal to M = the # of values specified.

Grid cells not in the specified *group-ID* or whose implicit surfaces are not assigned to the specified *reaction-ID* will output zeroes for all their values.

The array can be accessed by any command that uses per-grid values from a compute as input. See *Section 6.4* for an overview of SPARTA output options.

The per-grid array values are counts of the number of reactions that occurred on surface elements in that grid cell.

Restrictions:

none

Related commands:

fix ave/grid command dump grid, *compute react/surf command*

Default:

none

1.14.30 compute react/surf command

Syntax:

```
compute ID react/surf group-ID reaction-ID value1 value2 ...
```

- ID is documented in *compute command*
- react/surf = style name of this compute command
- group-ID = group ID for which surface elements to perform calculation on
- reaction-ID = surface reaction ID which defines surface reactions

- zero or more values can be appended
- value = *r:s1/s2/s3 ...* or *p:s1/s2/s3 ...*
  - r: or p: = list of reactant species or product species
  - s1,s2,s3 = one or more species IDs, separated by "/" character

Examples:

```
surf_react air prob air.surf
compute 1 react/surf all air
compute 2 react/surf all air r:N/O/N2/O2 p:N/O/NO
```

These commands will dump time averages for each surface element to a dump file every 1000 steps:

```
compute 2 react/surf all air r:N/O/N2/O2 p:N/O/NO
fix 1 ave/surf all 10 100 1000 c_2[*]
dump 1 surf all 1000 tmp.surf id f_1[*]
```

Description:

Define a computation that tallies counts of reactions for each explicit surface element in a surface element group, based on the particles that collide with that element. Only surface elements in the surface group specified by *group-ID* are included in the tallying. See the *group surf* for info on how surface elements can be assigned to surface groups. Likewise only surface elements assigned to the surface reaction model specified by *reaction-ID* are included in the tallying.

Explicit surface elements are triangles for 3d simulations and line segments for 2d simulations. Unlike implicit surface elements, each explicit triangle or line segment may span multiple grid cells. See the *read_surf command* for details.

This command can only be used for simulations with explicit surface elements. See the similar *compute react/isurf/grid* for use with simulations with implicit surface elements.

Note that when a particle collides with a surface element, it can bounce off (possibly as a different species), be captured by the surface (vanish), or a 2nd particle can also be emitted.

The doc page for the *surf_react command* explains the different reactions that can occur for each specified style.

If no values are specified each reaction specified by the *surf_react* style is tallied individually for each surface element.

If M values are specified, then M tallies are made for each surface element, one per value. If the value starts with "r:" then any reaction which occurs with one (or more) of the listed species as a reactant is counted as part of that tally. If the value starts with "p:" then any reaction which occurs with one (or more) of the listed species as a product is counted as part of that tally. Note that these rules mean that a single reaction may be tallied multiple times depending on which values it matches.

The results of this compute can be used by different commands in different ways. The values for a single timestep can be output by the *dump surf* command.

The values over many sampling timesteps can be averaged by the *fix ave/surf command*.

Output info:

This compute calculates a per-surf array, with the number of columns either equal to the number of reactions defined by the *surf_react* style (if no values are specified) or equal to M = the # of values specified.

Surface elements not in the specified *group-ID* or not assigned to the specified *reaction-ID* will output zeroes for all their values.

The array can be accessed by any command that uses per-surf values from a compute as input. See *Output from SPARTA (stats, dumps, computes, fixes, variables)* for an overview of SPARTA output options.

The per-surf array values are counts of the number of reactions that occurred.

Restrictions:

none

Related commands:

fix ave/surf command, *dump surf* , *compute react/isurf/grid*

Default:

none

1.14.31 compute reduce command

Syntax:

```
compute ID reduce mode input1 input2 ... keyword args ...
```

- ID is documented in *compute* command
- reduce = style name of this compute command
- mode = *sum* or *min* or *max* or *ave* or *sumsq* or *avesq*
- one or more inputs can be listed
- input = x, y, z, vx, vy, vz, ke, erot, evib, c_ID, c_ID[N], f_ID, f_ID[N], v_name
  - x,y,z,vx,vy,vz = particle position or velocity component
  - ke,erot,evib = particle energy component
  - c_ID = per-particle or per-grid vector calculated by a compute with ID
  - c_ID[I] = Ith column of per-particle or per-grid array calculated by a compute with ID, I can include wildcard (see below)
  - f_ID = per-particle or per-grid or per-surf vector calculated by a fix with ID
  - f_ID[I] = Ith column of per-particle or per-grid or per-surf array calculated by a fix with ID, I can include wildcard (see below)
  - v_name = per-particle or per-grid vector calculated by a particle-style or grid-style variable with name

- zero or more keyword/args pairs may be appended

  keyword = `replace`

  - replace args = vec1 vec2
    - vec1 = reduced value from this input vector will be replaced
    - vec2 = replace it with vec1[N] where N is index of max/min value from vec2

Examples:

```
compute 1 reduce sum c_grid[*]
compute 2 reduce min f_ave v_myKE
compute 3 reduce max c_mine[1] c_mine[2] c_temp replace 1 3 replace 2 3
```

These commands will include the average grid cell temperature, across all grid cells, in the stats output:

```
compute 1 temp
compute 2 grid all all temp
compute 3 reduce ave c_2[1]
stats_style step c_temp c_3
```

Description:

Define a calculation that "reduces" one or more vector inputs into scalar values, one per listed input. The inputs can be per-particle or per-grid or per-surf quantities; they cannot be global quantities. Particle attributes are per-particle quantities, *computes* may generate per-particle or per-grid quantities, *fixes* may generate any of the three kinds of quantities, and *particle-style or grid-style variables* generate per-particle or per-grid quantities. See the *variable command* and its special functions which can perform the same operations as the compute reduce command on global vectors.

The reduction operation is specified by the *mode* setting. The *sum* option adds the values in the vector into a global total. The *min* or *max* options find the minimum or maximum value across all vector values. The *ave* setting adds the vector values into a global total, then divides by the number of values in the vector. The *sumsq* option sums the square of the values in the vector into a global total. The *avesq* setting does the same as *sumsq*, then divdes the sum of squares by the number of values. The last two options can be useful for calculating the variance of some quantity, e.g. variance = sumsq - ave^2.

Each listed input is operated on independently.

Each listed input can be a particle attribute or can be the result of a *compute* or *fix* or the evaluation of a *variable*.

Note that for values from a compute or fix, the bracketed index I can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "n*" or "m*n". If N = the size of the vector (for *mode* = scalar) or the number of columns in the array (for *mode* = vector), then an asterisk with no numeric values means all indices from 1 to N. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from n to N (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual columns of the array had been listed one by one. E.g. these 2 compute reduce commands are equivalent, since the *compute grid* command creates a per-grid array with 3 columns:

```
compute myGrid grid all all u v w
compute 2 all reduce min c_myGrid[*]
compute 2 all reduce min c_myGrid[1] c_myGrid[2] c_myGrid[3]
```

The particle attributes x,y,z,vx,vy,vz are position and velocity components. The ke,erot,evib attributes are for kinetic, rotational, and vibrational energy of particles.

If a value begins with `c_`, a compute ID must follow which has been previously defined in the input script. Computes can generate per-particle or per-grid quantities. See the individual *compute* doc page for details. If no bracketed integer is appended, the vector calculated by the compute is used. If a bracketed integer is appended, the Ith column of the array calculated by the compute is used. Users can also write code for their own compute styles and *add them to SPARTA*. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

Important: A compute which generates per-surf quantities cannot be used as input. This is because its values have not yet been combined across processors to sum the contributions from all processors whose particles collide with the same surface element. The combining is performed by the *fix ave/surf* command, at each of its *Nfreq* timesteps. Thus to use this compute on per-surf values, specify a fix ID for a *fix ave/surf* and insure the fix outputs its values when they are needed.

If a value begins with `f_`, a fix ID must follow which has been previously defined in the input script. Fixes can generate per-particle or per-grid or per-surf quantities. See the individual *fix* doc page for details. Note that some fixes only produce their values on certain timesteps, which must be compatible with when this compute references the values, else an error results. If no bracketed integer is appended, the vector calculated by the fix is used. If a bracketed integer is appended, the Ith column of the array calculated by the fix is used. Users can also write code for their own fix style and *add them to SPARTA*. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

If a value begins with `v_`, a variable name must follow which has been previously defined in the input script. It must be a *particle-style or grid-style variable*. Both styles define formulas which can reference stats keywords or invoke other computes, fixes, or variables when they are evaluated. Particle-style variables can also reference various per-particle attributes (position, velocity, etc). So these variables are a very general means of creating per-particle or per-grid quantities to reduce.

If the *replace* keyword is used, two indices *vec1* and *vec2* are specified, where each index ranges from 1 to the # of input values. The replace keyword can only be used if the *mode* is *min* or *max*. It works as follows. A min/max is computed as usual on the *vec2* input vector. The index N of that value within *vec2* is also stored. Then, instead of performing a min/max on the *vec1* input vector, the stored index is used to select the Nth element of the *vec1* vector.

Here is an example which prints out both the grid cell ID and number of particles for the grid cell with the maximum number of particles:

```
compute 1 property/grid id
compute 2 grid all n
compute 3 reduce max c_1 c_2[1] replace 1 2
stats_style step c_temp c_3[1] c_3[2]
```

The first two input values in the compute reduce command are vectors with the ID and particle count of each grid cell. Instead of taking the max of the ID vector, which does not yield useful information in this context, the *replace* keyword will extract the ID for the grid cell which has the maximum number of particles. This ID and the cell's particle count will be printed with the statistical output.

If a single input is specified this compute produces a global scalar value. If multiple inputs are specified, this compute produces a global vector of values, the length of which is equal to the number of inputs specified.

Output info:

This compute calculates a global scalar if a single input value is specified or a global vector of length N where N is the number of inputs, and which can be accessed by indices 1 to N. These values can be used by any command that uses global scalar or vector values from a compute as input. See *Section 6.4* for an overview of SPARTA output options.

The scalar or vector values will be in whatever *units* the quantities being reduced are in.

Restrictions:

none

Related commands:

compute command, *fix command*, *variable command*

Default:

none

1.14.32 compute sonine/grid command

1.14.33 compute sonine/grid/kk command

Syntax:

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

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

```
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 all air a x 5 b xy 5
compute 1 sonine/grid subset air a x 5
```

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

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

Description:

Define a computation that calculates the sonine moments of the velocity distribution of the particles in each grid cell in a grid cell group. The values are tallied separately for each group of species in the specified mixture, as described in the Output section below. See the mixture command for how a set of species can be partitioned into groups.

Only grid cells in the grid group specified by *group-ID* are included in the calculations. See the *group grid* command for info on how grid cells can be assigned to grid groups.

The results of this compute can be used by different commands in different ways. The values for a single timestep can be output by the *dump grid* command.

The values over many sampling timesteps can be averaged by the *fix ave/grid* command. It does its averaging as if the particles in the cell at each sampling timestep were combined together into one large set of particles to compute the A,B formulas below.

Note however that the center-of-mass (COM) velocity that is subtracted from each particle to yield a squared thermal velocity Csq for each particle, as described below, is the COM velocity for only the particles in the current timestep. When time-averaging it is NOT the COM velocity for all particles across all timesteps.

Note that this is a different form of averaging than taking the values produced by the formulas below for a single timestep, summing those values over the sampling timesteps, and then dividing by the number of sampling steps.

Calculation of both the A and B sonine moments is done by first calcuating the center-of-mass (COM) velocity of particles for each group within a grid cell. This is done as follows:

```
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)
Cx = Vx - COMx
Cy = Vy - COMy
Cz = Vz - COMz
Csq = Cx*Cx + Cy*Cy + Cz*Cz
```

The COM velocity is (COMx,COMy,COMz). The thermal velocity of each particle is (Cx,Cy,Cz), i.e. its velocity minus the COM velocity of particles in its group and cell. This allows computation of Csq for each particle which is used in the formulas below to calculate the sonine moments.

The *a* keyword calculates the average of one or more sonine A moments for all particles in each group:

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

Vdim is Vx or Vy or Vz as specified by the *dim* value. *Csq* is the squared thermal velocity of the particle, as in the COM equations above. The number of moments computed is specified by the *order* value, e.g. for order = 3, the first 3 moments are computed, which leads to 3 columns of output as explained below.

The *b* keyword calculates the average of one or more sonine B moments for all particles in each group:

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

Vdim is Vx or Vy or Vz as specified by the *dim* value. *Csq* is the squared thermal velocity of the particle, as in the COM equations above. The number of moments computed is specified by the *order* value, e.g. for order = 2, the first 2 moments are computed, which leads to 2 columns of output as explained below.

Output info:

This compute calculates a per-grid array, with the number of columns 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 *a z 3* and *b xy 2* moments were specified as keywords, then the 1st thru 3rd columns would be the A1, A2, A3 moments of the first group, the 4th and 5th columns would be the B1 and B2 moments of the first group, the 6th thru 8th columns would be the A1, A2, A3 moments of the 2nd group, etc.

This compute performs calculations for all flavors of child grid cells in the simulation, which includes unsplit, cut, split, and sub cells. See *Section 6.8* of the manual gives details of how SPARTA defines child, unsplit, split, and sub cells. Note that cells inside closed surfaces contain no particles. These could be unsplit or cut cells (if they have zero flow volume). Both of these kinds of cells will compute a zero result for all their values. Likewise, split cells store no particles and will produce a zero result. This is because their sub-cells actually contain the particles that are geometrically inside the split cell.

Grid cells not in the specified *group-ID* will have zeroes for all their values.

The array can be accessed by any command that uses per-grid values from a compute as input. See *Section 6.4* for an overview of SPARTA output options.

The per-grid array values will be in the *units* appropriate to the individual values as described above. These are units like velocity cubed or velocity to the 6th power.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

none

Related commands:

fix ave/grid command, *dump grid*

Default:

none

1.14.34 compute surf command

1.14.35 compute surf/kk command

Syntax:

```
compute ID surf group-ID mix-ID value1 value2 ...
```

- `ID` is documented in *compute command*
- `surf` = style name of this compute command
- `group-ID` = group ID for which surface elements to perform calculation on
- `mix-ID` = mixture ID for particles to perform calculation on

  one or more values can be appended
- `value` = `n` or `nwt` or `mflux` or `fx` or `fy` or `fz` or `press` or `px` or `py` or `pz` or `shx` or `shy` or `shz` or `ke`
  - `n` = *count of particles hitting surface element*
  - `nwt` = *weighted count of particles hitting surface element*
  - `mflux` = *flux of mass on surface element*
  - `fx,fy,fz` = *components of force on surface element*
  - `press` = *magnitude of normal pressure on surface element*
  - `px,py,pz` = *components of normal pressure on surface element*
  - `shx,shy,shz` = *components of shear stress on surface element*
  - `ke` = *flux of particle kinetic energy on surface element*
  - `erot` = *flux of particle rotational energy on surface element*
  - `evib` = *flux of particle vibrational energy on surface element*
  - `etot` = *flux of particle total energy on surface element*

Examples:

```
compute 1 surf all all n press eng
compute mine surf sphere species press shx shy shz
```

These commands will dump time averages for each species and each surface element to a dump file every 1000 steps:

```
compute 1 surf all species n press shx shy shz
fix 1 ave/surf all 10 100 1000 c_1[*]
dump 1 surf all 1000 tmp.surf id f_1[*]
```

These commands will time-average the force on each surface element then sum them across element to compute drag (fx) and lift (fy) on the body:

```
compute 1 surf all all fx fy
fix 1 ave/surf all 10 100 1000 c_1[*]
compute 2 reduce sum f_1[1] f_1[2]
stats 1000
stats_style step cpu np c_2[1] c_2[2]
```

Description:

Define a computation that calculates one or more values for each explicit surface element in a surface element group, based on the particles that collide with that element. 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. Only surface elements in the surface group specified by *group-ID* are included in the calculations. See the *group surf* command for info on how surface elements can be assigned to surface groups.

Explicit surface elements are triangles for 3d simulations and line segments for 2d simulations. Unlike implicit surface elements, each explicit triangle or line segment may span multiple grid cells. See the *read_surf* command for details.

This command can only be used for simulations with explicit surface elements. See the similar *compute isurf/grid* command for use with simulations with implicit surface elements.

Note that when a particle collides with a surface element, it can bounce off (possibly as a different species), be captured by the surface (vanish), or a 2nd particle can also be emitted. The formulas below account for all the possible outcomes. For example, the kinetic energy flux *ke* onto a suface element for a single collision includes a positive contribution from the incoming particle and negative contributions from 0, 1, or 2 outgoing particles. The exception is the *n* and *nwt* values which simply tally counts of particles colliding with the surface element.

If the surface element is transparent, the particle will pass through the surface unaltered. The flux of particle count, mass, or energy to the surface can still be tallied by this compute. See details on transparent surface elements below.

Also note that all values for a collision are tallied based on the species group of the incident particle. Quantities associated with outgoing particles are part of the same tally, even if they are in different species groups.

The results of this compute can be used by different commands in different ways. The values for a single timestep can be output by the *dump surf* command.

The values over many sampling timesteps can be averaged by the *fix ave/surf* command. It does its averaging as if the particles striking the surface element at each sampling timestep were combined together into one large set to compute the formulas below. The answer is then divided by the number of sampling timesteps if it is not otherwise normalized by the number of particles. Note that in general this is a different normalization than taking the values produced by the formulas below for a single timestep, summing them over the sampling timesteps, and then dividing by the number of sampling steps. However for the current values listed below, the two normalization methods are the same.

Note: If particle weighting is enabled via the *global weight* command, then all of the values below are scaled by the weight assigned to the grid cell in which the particle collision with the surface element occurs. The only exception is the the `n` value, which is **NOT** scaled by the weight; it is a simple count of particle collisions with the surface element.

The `n` value counts the number of particles in the group striking the surface element.

The `nwt` value counts the number of particles in the group striking the surface element and weights the count by the weight assigned to the grid cell in which the particle collision with the surface element occurs. The *nwt* quantity will only be different than `n` if particle weighting is enabled via the *global weight* command.

The `mflux` value calculates the mass flux imparted to the surface element by particles in the group. This is computed as

```
Mflux = Sum_i (mass_i) / (A * dt / fnum)
```

where the sum is over all contributing particle masses, normalized by A = the area of the surface element, dt = the timestep, and fnum = the real/simulated particle ratio set by the *global fnum* command.

The `fx`, `fy`, `fz` values calculate the components of force extered on the surface element by particles in the group, with respect to the x, y, z coordinate axes. These are computed as

```
p_delta = mass * (V_post - V_pre)
Px = - Sum_i (p_delta_x) / (dt / fnum)
Py = - Sum_i (p_delta_y) / (dt / fnum)
Pz = - Sum_i (p_delta_z) / (dt / fnum)
```

where p_delta is the change in momentum of a particle, whose velocity changes from V_pre to V_post when colliding with the surface element. The force exerted on the surface element is the sum over all contributing p_delta, normalized by dt and fnum as defined before.

The `press` value calculates the pressure *P* exerted on the surface element in the normal direction by particles in the group, such that outward pressure is positive. This is computed as

```
p_delta = mass * (V_post - V_pre)
P = Sum_i (p_delta_i dot N) / (A * dt / fnum)
```

where p_delta, V_pre, V_post, dt, fnum are defined as before. The pressure exerted on the surface element is the sum over all contributing p_delta dotted into the outward normal N of the surface element, also normalized by A = the area of the surface element.

The `px`, `py`, `pz` values calculate the normal pressure Px, Py, Pz extered on the surface element in the direction of its normal by particles in the group, with respect to the x, y, z coordinate axes. These are computed as

```
p_delta = mass * (V_post - V_pre)
p_delta_n = (p_delta dot N) N
Px = - Sum_i (p_delta_n_x) / (A * dt / fnum)
Py = - Sum_i (p_delta_n_y) / (A * dt / fnum)
Pz = - Sum_i (p_delta_n_z) / (A * dt / fnum)
```

where p_delta, V_pre, V_post, N, A, and dt are defined as before. P_delta_n is the normal component of the change in momentum vector p_delta of a particle. P_delta_n_x (and y,z) are its x, y, z components.

The `shx`, `shy`, `shz` values calculate the shear pressure Sx, Sy, Sz extered on the surface element in the tangential direction to its normal by particles in the group, with respect to the x, y, z coordinate axes. These are computed as

```
p_delta = mass * (V_post - V_pre)
p_delta_t = p_delta - (p_delta dot N) N
Sx = - Sum_i (p_delta_t_x) / (A * dt / fnum)
Sy = - Sum_i (p_delta_t_y) / (A * dt / fnum)
Sz = - Sum_i (p_delta_t_z) / (A * dt / fnum)
```

where p_delta, V_pre, V_post, N, A, and dt are defined as before. P_delta_t is the tangential component of the change in momentum vector p_delta of a particle. P_delta_t_x (and y,z) are its x, y, z components.

The `ke` value calculates the kinetic energy flux *Eflux* imparted to the surface element by particles in the group, such that energy lost by a particle is a positive flux. This is computed as

```
e_delta = 1/2 mass (V_post^2 - V_pre^2)
Eflux = - Sum_i (e_delta) / (A * dt / fnum)
```

where e_delta is the kinetic energy change in a particle, whose velocity changes from V_pre to V_post when colliding with the surface element. The energy flux imparted to the surface element is the sum over all contributing e_delta, normalized by A = the area of the surface element and dt = the timestep and fnum = the real/simulated particle ratio set by the *global fnum* command.

The `erot` value calculates the rotational energy flux *Eflux* imparted to the surface element by particles in the group, such that energy lost by a particle is a positive flux. This is computed as

```
e_delta = Erot_post - Erot_pre
Eflux = - Sum_i (e_delta) / (A * dt / fnum)
```

where e_delta is the rotational energy change in a particle, whose internal rotational energy changes from Erot_pre to Erot_post when colliding with the surface element. The flux equation is the same as for the *ke* value.

The `evib` value calculates the vibrational energy flux *Eflux* imparted to the surface element by particles in the group, such that energy lost by a particle is a positive flux. This is computed as

```
e_delta = Evib_post - Evib_pre
Eflux = - Sum_i (e_delta) / (A * dt / fnum)
```

where e_delta is the vibrational energy change in a particle, whose internal vibrational energy changes from Evib_pre to Evib_post when colliding with the surface element. The flux equation is the same as for the *ke* value.

The `etot` value calculates the total energy flux imparted to the surface element by particles in the group, such that energy lost by a particle is a positive flux. This is simply the sum of kinetic, rotational, and vibrational energies. Thus the total energy flux is the sum of what is computed by the *ke*, *erot*, and *evib* values.

Transparent surface elements:

This compute will tally information on particles that pass through transparent surface elements. The section *Transparent surface elements* in *How-to discussions* page provides an overview of transparent surfaces and how to create them.

The `n` and `nwt` value are calculated the same for transparent surfaces. I.e. they are the count and weighted count of particles passing through the surface.

The `mflux`, `ke`, `erot`. `evib`, and `etot` values are fluxes. For transparent surfaces, they are calculated for the incident particle as if they had struck the surface. The outgoing particle is ignored. This means the tally quantity is the flux of particles onto the outward face of the surface. No tallying is done for particles hitting the inward face of the surface. See *Section 6.15* for how to do tallying in both directions.

All the other values are calculated as described above. This means they will be zero, since the incident particle and outgoing particle have the same mass and velocity.

Output info:

This compute calculates a per-surf array, with the number of columns 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.

Surface elements not in the specified *group-ID* will output zeroes for all their values.

The array can be accessed by any command that uses per-surf values from a compute as input. See *Section 6.4* for an overview of SPARTA output options.

The per-surf array values will be in the *units* appropriate to the individual values as described above. *N* is unitless. *Press*, *px*, *py*, *pz*, *shx*, *shy*, *shz* are in in pressure units. *Ke*, *erot*, *evib*, and *etot* are in energy/area-time units for 3d simulations and energy/length-time units for 2d simulations.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix command* in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

none

Related commands:

fix ave/surf command, *dump surf* , *compute isurf/grid command*

Default:

none

1.14.36 compute temp command

1.14.37 compute temp/kk 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.

Note that this definition of temperature does not subtract out a net streaming velocity for particles, so it is not a thermal temperature when the particles have a non-zero streaming velocity. See the *compute thermal/grid* command for calculation of thermal temperatures on a per grid cell basis.

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 *Section 6.4* for an overview of SPARTA output options.

The scalar value will be in temperature *units*.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

none

Related commands:

none

Default:

none

1.14.38 compute thermal/grid command

1.14.39 compute thermal/grid/kk command

Syntax:

```
compute ID thermal/grid group-ID mix-ID value1 value2 ...
```

- ID is documented in *compute* command
- thermal/grid = style name of this compute command
- group-ID = group ID for which grid cells to perform calculation on
- mix-ID = mixture ID to perform calculation on
- one or more values can be appended
- value = *temp* or *press*

```
temp = temperature
press = pressure
```

Examples:

```
compute 1 thermal/grid all species temp
compute 1 thermal/grid subset air temp press
```

These commands will dump 10 time averaged thermal temperatures for each species and each grid cell to a dump file every 1000 steps:

```
compute 1 thermal/grid species temp
fix 1 ave/grid 10 100 1000 c_1[*]
dump 1 grid all 1000 tmp.grid id f_1[*]
```

Description:

Define a computation that calculates one or more values for each grid cell in a grid cell group, which are based on the thermal temperature of the particles in each grid cell. The values are tallied separately for each group of species in the specified mixture, as described in the Output section below. See the mixture command for how a set of species can be partitioned into groups.

Only grid cells in the grid group specified by *group-ID* are included in the calculation. See the *group grid* command for info on how grid cells can be assigned to grid groups.

The values listed above rely on first computing a thermal temperature which subtracts the center-of-mass (COM) velocity for all particles in the group and grid cell from each particle to yield a thermal velocity. This thermal velocity is used to compute the temperature, as described below. This is in contrast to some of the values tallied by the *compute grid temp* command which simply uses the full velocity of each particle to compute a temperature. For non-streaming simulations, the two results should be similar, but for streaming flows, they will be different.

The results of this compute can be used by different commands in different ways. The values for a single timestep can be output by the *dump grid* command.

The values over many sampling timesteps can be averaged by the *fix ave/grid* command. It does its averaging as if the particles in the cell at each sampling timestep were combined together into one large set of particles to compute the formulas below.

Note that this is a different form of averaging than taking the values produced by the formulas below for a single timestep, summing those values over the sampling timesteps, and then dividing by the number of sampling steps.

Also note that the center-of-mass (COM) velocity that is subtracted from each particle to yield a squared thermal velocity Csq for each particle, as described below, is also computed over one large set of particles (across all timesteps). This is in contrast to using a COM velocity computed only for particles in the current timestep, which is what the *compute sonine/grid* command does.

Calculation of the thermal temperature is done by first calcuating the center-of-mass (COM) velocity of particles for each group within a grid cell. This is done as follows:

```
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)
Cx = Vx - COMx
Cy = Vy - COMy
Cz = Vz - COMz
Csq = Cx*Cx + Cy*Cy + Cz*Cz
```

The COM velocity is (COMx,COMy,COMz). The thermal velocity of each particle is (Cx,Cy,Cz), i.e. its velocity minus the COM velocity of particles in its group and cell. This allows computation of Csq for each particle which is used to calculate the total kinetic energy due to particles in the group as follows:

```
thermal_KE = Sum_i (1/2 mass_i Csq_i)
```

The *temp* value computes the thermal temperature T, due to particles in each group:

```
T = thermal_KE / (3/2 N kB)
```

The *press* value uses the thermal_KE to compute a pressure P for the grid cell due to particles in the group:

```
P = 2/3 fnum/volume * thermal_KE
```

Note that if multiple groups are defined in the mixture, one group's value is effectively a partial pressure due to particles in the group. When accumulated over multiple sampling steps, this value is normalized by the number of sampling steps. Note that if particle weighting is enabled via the *global weight* command, then the volume used in the formula is divided by the weight assigned to the grid cell.

Output info:

This compute calculates a per-grid array, with the number of columns 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 *temp* and *press* values were specified as keywords, then the first two columns would be *temp* and *press* for the first group, the 3rd and 4th columns would be *temp* and *press* for the second group, etc.

This compute performs calculations for all flavors of child grid cells in the simulation, which includes unsplit, cut, split, and sub cells. See *Section 6.8* of the manual gives details of how SPARTA defines child, unsplit, split, and sub cells. Note that cells inside closed surfaces contain no particles. These could be unsplit or cut cells (if they have zero flow volume). Both of these kinds of cells will compute a zero result for all their values. Likewise, split cells store no particles and will produce a zero result. This is because their sub-cells actually contain the particles that are geometrically inside the split cell.

Grid cells not in the specified *group-ID* will output zeroes for all their values.

The array can be accessed by any command that uses per-grid values from a compute as input. See *Section 6.4* for an overview of SPARTA output options.

The per-grid array values will be in the *units* appropriate to the individual values as described above. *Temp* is in temperature units. *Press* is in prsesure units.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

none

Related commands:

compute grid command fix ave/grid command dump grid

Default:

none

1.14.40 compute tvib/grid command

Syntax:

```
compute ID tvib/grid group-ID mix-ID keyword ...
```

- ID is documented in *compute* command
- tvib/grid = style name of this compute command
- group-ID = group ID for which grid cells to perform calculation on
- mix-ID = mixture ID to perform calculation on
- zero or more keywords can follow

```
possible keywords = mode
mode = output one temperature per vibrational mode
```

Examples:

```
compute 1 tvib/grid all species
compute 1 tvib/grid subset all
compute 1 tvib/grid all species mode
```

Description:

Define a computation that calculates the vibrational temperature for each grid cell in a grid cell group, based on the particles in the cell. How the vibrational temperature is computed is explained below. The temperature is calculated separately for each group of species in the specified mixture, as described in the Output section below. See the mixture command for how a set of species can be partitioned into groups.

Only grid cells in the grid group specified by *group-ID* are included in the calculations. See the *group grid* command for info on how grid cells can be assigned to grid groups.

The results of this compute can be used by different commands in different ways. The values for a single timestep can be output by the *dump grid* command.

The values over many sampling timesteps can be averaged by the *fix ave/grid* command. It does its averaging as if the particles in the cell at each sampling timestep were combined together into one large set to compute the formulas below. Note that this is a different normalization than taking the values produced by the formulas below for a single timestep, summing them over the sampling timesteps, and then dividing by the number of sampling steps.

If the *mode* keyword is specified, then temperatures for each vibrational mode of each polyatomic species are calculated and output as explained below. To use this option, the *collide_modify vibrate discrete* option must be set, and the "fix vibmode" command must be used to store info about individual vibrational modes with each particle.

The vibrational temperature in a grid cell for a group of particles comprised of different species and (optionally) different vibrational modes is defined as a weighted average as follows:

```
T_group = (T1*N1 + T2*N2 + ...) / (N1 + N2 + ...)
```

What is summed over in the numerator and denominator depends on several settings.

If the *collide_modify vibrate* setting is *no*, then no vibrational energy is assigned to particles. All the output temperatures will be 0.0.

If the *collide_modify vibrate* setting is *smooth*, then the sums in the numerator and denominator are over the different species in the group. T1, T2, ... are the vibrational temperatures of each species. N1, N2, ... are the counts of particles of each species.

The vibrational temperature Tsp for particles of a single species is defined as follows:

```
Ibar = Sum_i (e_vib_i) / (N kB Theta)
Tsp = Theta / ln(1 + 1/Ibar))
```

where e_vib is the continuous (smooth) vibrational energy of a single particle I, N is the total # of particles of that species, and *kB* is the Boltzmann factor. Theta is the characteristic vibrational temperature for the species, as defined in the file read by the *species* command.

If the *collide_modify vibrate* setting is *discrete*, but no species has a vibrational DOF setting that implies multiple vibrational modes (vibdof = 4,6,8), then the calulation of vibrational temeperatures is the same as for *collide_modify vibrate smooth*. See the *species* command and its description of the per-species "vibdof" setting in the species file.

If the *collide_modify vibrate* setting is *discrete*, and one or more species have vibrational DOF settings that imply multiple vibrational modes (vibdof = 4,6,8), as defined by the *species* command, then the sums in the numerator and denominator are over the different species in the group and the modes for each species. For example if species CO2 has vibdof=6, then it has 3 modes. Three terms in the numerator and demoninator are included when CO2 is a species in the group.

The vibrational temperature Tsp_m for particles of a single species and single mode M is defined as follows:

```
Ibar_m = Sum_i (level_im) / (N)
Tsp_m = Theta_m / ln(1 + 1/Ibar_m))
```

where level_im is the integer level for mode M of a single particle I, and N is the total # of particles of that species. Theta_m is the characteristic vibrational temperature for the species and its mode M, as defined in the vibfile read by the *species* command.

Finally, if the *mode* keyword is used, then the output of this compute is not Ngroup vibrational temperatures, but rather Ngroup*Nmode vibrational temperatures, where Nmode is the maximum # of vibrational modes associated with any species in the system (not just in the mixture). Thus the sums in the numerator and denominator are over the different species in the group but for only a single modes of each of those species. If the species does not define that mode, then its contribution is zero. For example if species CO2 has vibdof=6, then it has 3 modes. For the group it is in, it will contribute to 3 output temperature values, one for mode 1, another for mode 2, another for mode 3.

The vibrational temperature Tsp_m for particles of a single species and single mode M is calculated the same as explained above.

Output info:

This compute calculates a per-grid array. If the *mode* keyword is not specified, the number of columns is equal to the number of groups in the specified mixture. If is is specified, the number of columns is equal to the number of groups in the specified mixture times the maximum number of vibrational modes defined for any species in the system (not just in the mixture). The ordering of the columns is as follows: T11, T12, T13, T21, T22, T23, T31, ... TN1, TN2, TN3. Where the first index is the group from 1 to N, and the second index is the vibrational mode (1 to 3 in this example).

This compute performs calculations for all flavors of child grid cells in the simulation, which includes unsplit, cut, split, and sub cells. See *Section 4.8* of the manual gives details of how SPARTA defines child, unsplit, split, and sub cells. Note that cells inside closed surfaces contain no particles. These could be unsplit or cut cells (if they have zero flow volume). Both of these kinds of cells will compute a zero result for all their values. Likewise, split cells store no particles and will produce a zero result. This is because their sub-cells actually contain the particles that are geometrically inside the split cell.

Grid cells not in the specified *group-ID* will output zeroes for all their values.

The array can be accessed by any command that uses per-grid values from a compute as input. See *Section 4.4* for an overview of SPARTA output options.

The per-grid array values will be in temperature *units*.

Restrictions:

none

Related commands:

compute grid command

Default:

none

1.14.41 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 means the z dimensions straddle 0.0. Typical values are -0.5 and 0.5, but this is not required. See *Section 6.1* of the manual for more information about 2d simulations.

For 2d axisymmetric simulations, as set by the *dimension* and *boundary* commands, the ylo setting must be 0.0. See *Section 6.2* of the manual for more information about axisymmetric simulations.

Restrictions:

none

Related commands:

none

Default:

none

1.14.42 create_grid command

Syntax:

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

- Nx,Ny,Nz = size of 1st-level grid in each dimension
- zero or more keywords/args pairs may be appended
- keyword = *level* or *region* or *stride* or *clump* or *block* or *random* or *inside*
  - level args = Nlevel Px Py Pz Cx Cy Cz
    - Nlevel = level from 2 to M, must be in ascending order
    - Px Py Pz = range of parent cells in each dimension in which to create child cells
    - Cx Cy Cz = size of child grid in each dimension within parent cells
  - region args = Nlevel reg-ID Cx Cy Cz
    - Nlevel = level from 2 to M, must be in ascending order
    - reg-ID = ID of region which parent cells must be in to create child cells
    - Cx Cy Cz = size of child grid in each dimension within parent cells
  - stride arg = xyz or xzy or yxz or yzx or zxy or zyx
  - clump arg = xyz or xzy or yxz or yzx or zxy or zyx
  - block args = Px Py Pz
    - Px,Py,Pz = # of processors in each dimension
  - random args = none
  - inside args = any or all

Examples:

```
create_grid 10 10 10
create_grid 10 10 10 block * * *
create_grid 10 10 10 block 4 2 5
create_grid 10 10 10 level 2 * * * 2 2 3
create_grid 20 10 1 level 2 10*15 3*7 1 2 2 1
create_grid 20 10 1 region 2 b2 2 2 1 region 3 b3 2 3 1 inside any
create_grid 20 10 1 level 2 10*15 3*7 1 2 2 1 region 3 b3 2 3 1
create_grid 8 8 10 level 2 5* * * 4 4 4 level 3 1 2*3 3* 2 2 1
```

Description:

Overlay a grid over the simulation domain defined by the *create_box* command. The grid can also be defined by the *read_grid* command.

The grid in SPARTA is hierarchical, as described in *Section howto*. The entire simulation box is a single parent grid cell at level 0. It is subdivided into Nx by Ny by Nz cells at level 1. Each of those cells can be a child cell (no further sub-division) or can be a parent cell which is further subdivided into Nx by Ny by Nz cells at level 2. This can recurse to as many levels as desired. Different cells can stop recursing at different levels. Each parent cell can define its own unique Nx, Ny, Nz values for subdivision. Note that a grid with a single level is simply a uniform grid with Nx by Ny by Nz cells in each dimension.

In the current SPARTA implementation, all processors own a copy of all parent cells. Each child cell is owned by a unique processor. The details of how child cells are assigned to processors by the various options of this command are described below. The cells assigned to each processor will either be "clumped" or "dispersed".

The *clump* and *block* keywords will produce clumped assignments of child cells to each processor. This means each processor's cells will be geometrically compact. The *stride* and *random* keywords, as well as the round-robin assignment scheme for grids with multiple levels (described below), will produce dispersed assignments of child cells to each processor.

Important: See *Section 6.8* of the manual for an explanation of clumped and dispersed grid cell assignments and their relative performance trade-offs. The *balance_grid* command can be used after the grid is created, to assign child cells to processors in different ways. The "fix balance" command can be used to re-assign them in a load-balanced manner periodically during a running simulation.

A single-level grid is defined by specifying only the arguments *Nx*, *Ny*, *Nz*, with no additional *level* or *region* keywords. This will create a uniform Nx by Ny by Nz grid of child cells. For 2d simulations, *Nz* must equal 1.

For single-level grids, one of the keywords *stride*, *clump*, *block*, or *random* can be used to determine which processors are assigned which cells in the grid. The *inside* keyword is ignored for single-level grids. If no keyword is used, the cells are assigned in round-robin fashion, so that each processor is assigned every Pth grid cell, where P = the number of processors. This is the same as "stride xyz" in the discussion 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. The 10th processor (proc 9) will be assigned cells 10,20,30, ..., 100.

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

The argument for *stride* and *clump* determines how the N grid cells are ordered and is some permutation of the letters *x*, *y*, and *z*. Each of the N cells has 3 indices (I,J,K) to describe its location in the 3d grid. If the stride argument is yxz, then the cells will be ordered from 1 to N with the y dimension (J index) varying fastest, the x dimension next (I index), and the z dimension slowest (K index).

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 *Section 2.6* 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 each grid cell will be assigned randomly to one of the processors. Note that in this case different processors will typically not be assigned exactly the same number of cells.

A hierarchical grid with more than one level can be defined using the *level* or *region* keywords one or more times with *Nlevel* in ascending order, starting with *Nlevel* = 2. At each level the *level* or *region* keyword can be used interchangeably. Child cells (at any level) are assigned to processors in round-robin fashion, so that each processor is assigned every Pth grid cell, where P = the number of processors.

Note that the keywords *stride*, *clump*, *block*, or *random* cannot be used with a hierarchical grid. The keyword *inside* can be used, but it must come after all the *level* or *region* keywords.

For the *level* keyword, the Px, Py, Pz arguments specify which cells in the previous level are flagged as parents and sub-divided to create cells at the new level. For example, if the level 1 grid is 100x100x100, then Px, Py, Pz for level 2 could select any contiguous range of cells from 1 to 100 in x, y, or z. If the level 2 grid is 4x4x2 within any level 1 cell (as set by Cx, Cy, Cz), then Px, Py, Pz for level 3 could select any contiguous range of cells from 1 to 4 in x, y and 1 to 2 in z.

Each of the Px, Py, Pz arguments can be a single number or be specified with a wildcard asterisk, as in the examples above. For example, Px can be specified as "*" or "*n" or "n*" or "m*n". If N = the number of grid cells in the x-direction in the previous level as defined by Nx (or Cx), then an asterisk with no numeric values means all cells with indices from 1 to N. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from n to N (inclusive). A middle asterisk means all indices from m to n (inclusive).

The Cx, Cy, Cz arguments are the number of new cells (in each dimension) to partition each selected parent cell into. For 2d simulations, *Cz* must equal 1. Note that for each new level, only grid cells that exist in the previous level are partitioned further. E.g. level 3 cells are only added to level 2 cells that exist, since some level 1 cells may not have been partitioned into level 2 cells.

This command creates a two-level grid:

```
create_grid 10 10 10 level 2 * * * 2 2 3
```

The 1st level is 10x10x10. Each of the 1000 level 1 cells is further partitioned into 2x2x3 cells. This means the total number of level 2 cells is 1000 * 12 = 12000. The resulting grid thus has 1001 parent cells (the simulation box plus the 1000 level 1 cells), and 12000 child cells.

This command creates a 3-level grid:

```
create_grid 8 8 10 level 2 5* * * 4 4 4 level 3 1 2*3 3* 2 2 1
```

The last example above creates a 3-level grid. The first level is 8x8x10. The second level is 4x4x4 within each 1st level cell, but only half or 320 of the 640 level 1 cells are partitioned, namely those with x indices from 5 to 8. Those with x indices from 1 to 4 remain as level 1 cells. Some of the level 2 cells are further partitioned into 2x2x1 level 3 cells. For the 4x4x4 level 2 grid within 320 or the level 1 cells, only the level 2 cells with x index = 1, y index = 2-3, and z-index = 3-4 are further partitioned into level 3 cells, which is just 4 of the 64 level 2 cells.

The resulting grid thus has 1601 parent cells: 1 for the simulation box, 320 level 1 cells, and 1280 level 2 cells. It has 24640 child cells: 320 level 1 cells, 19200 level 2 cells, and 5120 level 3 cells.

For the *region* keyword, the subset of cells in the previous level which are flagged as parents and sub-divided is determined by which of them are in the geometric region specified by *reg-ID*.

The *region* command can define volumes for simple geometric objects such as a sphere or rectangular block. It can also define unions or intersections of simple objects or other union or intersection objects. by defining an appropriate region, a complex portion of the simulation domain can be refined to a new level.

Each grid cell at the previous level is tested to see whether it is "in" the region. The *inside* keyword determines how this is done. If *inside* is set to *any* which is the default, then the grid cell is in the region if any of its corner points (4 in 2d, 8 in 3d) is in the region. If *inside* is set to *all*, then all 4 or 8 corner points must be in the region for the grid cell itself to be in the region. Note that the *side* option for the *region* command can be used to define whether the inside or outside of the geometric region is considered to be "in" the region.

If the grid cell is in the region, then it is refined using the Cx, Cy, Cz arguments in the same manner that the *level* keyword uses them. Examples for the use of the *region* keyword are given above.

Restrictions:

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

The hierarchical grid used by SPARTA is encoded in a 32-bit or 64-bit integer ID. The precision is set by the -DSPARTA_BIG or -DSPARTA_SMALL or -DSPARTA_BIGBIG compiler switch, as described in *Section 2.2*. The number of grid levels that can be used depends on the resolution of the grid at each level. For a minimal refinement of 2x2x2, a level uses 4 bits of the integer ID. Thus for this style of refinement a maximum of 7 levels can be used for 32-bit IDs and 15 levels for 64-bit IDs.

Related commands:

create_box command, *read_grid command*

Default:

The only keyword with a default setting is inside = any.

1.14.43 create_particles command

1.14.44 create_particles/kk command

Syntax:

```
create_particles mix-ID style args keyword value ...
```

- mix-ID = ID of mixture to use when creating particles

- style = *n* or *single*
  - n args = Np =
    - Np = 0 or number of particles to create
  - single args = species-ID x y z vx vy vz
    - species-ID = ID of species of single particle
    - x,y,z = position of particle (distance units)
    - vx,vy,vz = velocity of particle (velocity units)
- zero or more keyword/value pairs may be appended
- keyword = *global* or *region* or *species* or *density* or *temperature* or *velocity* or *twopass*
  - global value = yes or no
  - region value = region-ID
  - species values = svar xvar yvar zvar
  - density values = dvar xvar yvar zvar
  - temperature values = tvar xvar yvar zvar
  - velocity values = vxvar vyvar vzvar xvar yvar zvar
  - twopass values = none

Examples:

```
create_particles background n 0
create_particles air n 100000 region sphere
create_particles air n 100000 global yes
create_particles air single 3 5.0 6.0 5.4 10.0 -1.0 0.0
create_particles air n 0 species mySpecies xpos NULL zpos
create_particles air n 0 density myDens xgrid ygrid NULL
create_particles air n 0 temperature myTemp xgrid ygrid zgrid
create_particles air n 0 velocity myVx NULL myVz xpos ypos NULL twopass
```

Description:

Create particles and add them to the simulation domain. The attributes of individual particles, such as species and velocity, are determined by the mixture attributes, as specied by the *mix-ID*. In particular the *temp*, *trot*, *tvib*, and *vstream* attributes of the mixture affect create particle velocities and internal energy modes. See the *mixture* command for more details. Note that this command can be used multiple times to add more and more particles.

Particles are only created in grid cells which are entirely external to surfaces. Particles are not created in grid cells cut by surfaces.

Important: When a particle is created at a specified temperature (as set by the *mixture* command), it's rotational and vibrational energy will also be initialized, consistent with the mixture temperatures. The *rotate* and *vibrate* options of the *collide_modify* command determine how internal energy modes are initialized. If the *collide* command has not yet been specified, then no rotational or vibrational energy will be assigned to created particles. Thus if you wish to create particles with non-zero internal energy, the *collide* and (optionally) *collide_modify* commands must be used before this command.

If the *n* style is used with *Np* = 0, then the number of created particles is calculated by SPARTA as a function of the global *fnum* value, the mixture number density, and the flow 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 flow volume of the simulation is the total volume of the simulation domain as specified by the *create_box* command, minus any volume that is interior to surfaces defined by the *read_surf* command. Note that the flow volume includes volume contributions from grid cells cut by surfaces. However particles are only created in grid cells entirely external to surfaces. This means that particles may be created in external cells at a (slightly) higher density to compensate for no particles being created in cut cells that still contribute to the overall flow volume.

If the *n* style is used with a non-zero *Np*, then exactly *Np* particles are created, which can be useful for debugging or benchmarking purposes.

Based on the value of *Np*, each grid cell will have a target number of particles *M* to insert, which is a function of the cell's volume as compared to the total system flow volume. 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. As grid cells are looped over, the remainder fraction is accumulated, so that exactly *Np* particles are created across all the processors.

Important: The preceeding calculation is actually done using *weighted* cell volumes. Grid cells can be weighted using the *global weight* command.

Each particle is inserted at a random location within the grid cell. The particle species is 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 is 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 internal rotational and vibrational energies of the particle are also set based on the *trot* and *tvib* settings for the mixture, as explained above.

The *single* style creates a single particle. This can 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.

This is the meaning of the other allowed keywords.

The *global* keyword only applies when the *n* style is used, and controls how particles are generated in parallel.

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

---

Important: The *global* yes option is not yet implemented.

If the value is *no*, then each of the *P* processors generates a *N/P* subset of particles, using its own random number generation. It only adds particles to grid cells that it owns, as described above. This is a faster way to generate a large number of particles, but means that the individual attributes of particles will depend on the number of processors and the mapping of grid cells to procesors. The overall set of created particles should have the same statistical properties as with the *yes* setting.

If the *region* keyword is used, then a particle will only added if its position is within the specified *region-ID*. This can be used to only allow particle insertion within a subset of the simulation domain. Note that the *side* option for the *region* command can be used to define whether the inside or outside of the geometric region is considered to be "in" the region.

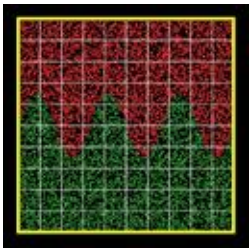
Important: If the *region* and *n* keywords are used together, less than N particles may be added. This is because grid cells will be candidates for particle insertion, unless they are entirely outside the bounding box that encloses the region. Particles those grid cells attempt to add are included in the count for N, even if some or all of the particle insertions are rejected due to not being inside the region.

The *species* keyword can be used to create particles with a spatially-dependent separation of species. The specified *svar* is the name of an *equal-style variable* whose formula should evaluate to a species number, i.e. an integer from 1 to Nsp, where Nsp is the number of species in the mixture with mix-ID. Since equal-style variables evaluate to floating-point values, this value is truncated to an integer value. The formula for the species variable can use one or two or three variables which will store the x, y, or z coordinates of the particle that is being created. If used, these variables must be *internal-style variables* defined in the input script; their initial numeric values can be anything. They must be internal-style variables, because this command resets their values directly. Their names are specified as *xvar*, *yvar*, and *zvar*. If any of them is not used in the *svar* formula, it can be specified as NULL.

When a particle is added, its coordinates are stored in the *xvar*, *yvar*, *zvar* variables if they are specified. The *svar* variable is then evaluated. The returned value is used to set the species of that particle, based on the list of species defined for the mixture. If the returned value is <= 0 or greater than Nsp = the number of species in the mixture, then no particle is created.

As an example, these commands can be used in a 2d simulation, to create a particle distribution with species 1 on top of species 2 with a sinudoidal interface between the two species, as illustrated in the snapshot of the initial particle distribution. Click on the image for a larger version. Note that when using this option less than the requested N particles can be created if the species variable returns values <= 0 or greater than Nsp = the number of species in the mixture.

```
variable x internal 0
variable y internal 0
variable n equal 3
variable s equal "(v_y < 0.5*(ylo+yhi) + 0.15*yhi*sin(2*PI*v_n*v_x/xhi)) + 1"
create_particles species n 10000 species s x y NULL
```



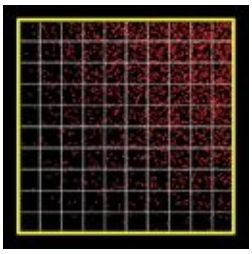
The *density* keyword can be used to create particles with a spatially-dependent density variation. The specified *dvar* is the name of an *equal-style variable* whose formula should evaluate to a positive value. The formula for *dvar* can use one or two or three variables which will store the x, y, or z coordinates of the geometric center point of a grid cell. If used, these other variables must be *internal-style variables* defined in the input script; their initial numeric values can by anything. Their names are specified as *xvar*, *yvar*, and *zvar*. If any of them is not used in the *dvar* formula, it can be specified as NULL.

When particles are added to a grid cell, its center point coordinates are stored in *xvar*, *yvar*, *zvar* if they are defined. The *dvar* variable is then evaluated. The returned value is used as a scale factor on the number of particles to create in that grid cell. Thus a value of 0.5 would create half as many particles in that grid cell as would otherwise be the case, due to the global *fnum* and mixture *nrho* settings that define the density, as explained above. A value of 1.2 would create 20% more particles in that grid cell.

As an example, these commands can be used in a 2d simulation, to create more particles towards the upper right corner of the domain and less towards the lower left corner, as illustrated in the snapshot of the initial particle distribution.

Click on the image for a larger version. Note that less than requested N particles will be created in this case because all the scale factors generated by the variable *d* are less than 1.0.

```
variable x internal 0
variable y internal 0
variable d equal "v_x/xhi * v_y/yhi"
create_particles air n 10000 density d x y NULL
```



The *temperature* keyword can be used to create particles with a spatially-dependent thermal temperature variation. The specified *tvar* is the name of an *equal-style variable* whose formula should evaluate to a positive value. The formula for the *tvar* variable can use one or two or three variables which will store the x, y, or z coordinates of the geometric center point of a grid cell. If used, these other variables must be *internal-style variables* defined in the input script; their initial numeric values can by anything. Their names are specified as *xvar*, *yvar*, and *zvar*. If any of them is not used in the *tvar* formula, it can be specified as NULL.

When particles are added to a grid cell, its center point coordinates are stored in *xvar*, *yvar*, *zvar* if they are defined. The *tvar* variable is then evaluated. The returned value is used as a scale factor on the thermal temperature number for particles created in that grid cell. Thus a value of 0.5 would create particles with a thermal temperature half of what would otherwise be the case, due to the mixture *temp* setting which defines the thermal temperature, as explained above. A value of 1.2 would create particles with a 20% higher thermal temperature.

As an example, these commands can be used in a 2d simulation, to create a thermal temperature gradient in x, where the temperature on the left side of the box is the default value, and the temperature on the right side is 3x larger.

```
variable x internal 0
variable t equal "1.0 + 2.0*(v_x-xlo)/(xhi-xlo)"
create_particles air n 10000 temperature t x NULL NULL
```

The *velocity* keyword can be used to create particles with a spatially-dependent streaming velocity. The specified *vxvar*, *vyvar*, *vzvar* are the names of *equal-style variables* whose formulas should evaluate to the corresponding component of the streaming velocity. If any of them are specified as NULL, then that streaming velocity component is set by the corresponding global or mixture streaming velocity component, the same as if the *velocity* keyword were not used.

The formulas for the *vxvar*, *vyvar*, *vzvar* variables can use one or two or three variables which will store the x, y, or z coordinates of the particle that is being created. If used, these other variables must be *internal-style variables* defined in the input script; their initial numerica values can by anything. Their names are specified as *xvar*, *yvar*, and *zvar*. If any of them is not used in the *vxvar*, *vyvar*, *vzvar* formulas, it can be specified as NULL.

When a particle is added, its coordinates are stored in *xvar*, *yvar*, *zvar* if they are defined. The *vxvar*, *vyvar*, *vzvar* variables are then evaluated. The returned values are used to set the streaming velocity of that particle. A thermal velocity is also added to the particle, using the the global or mixture temperature, as described above.

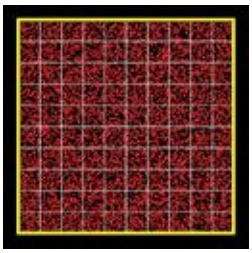
As an example, these commands can be used in a 2d simulation, to give particles an initial velocity pointing towards the upper right corner of the domain with a magnitude that makes them all reach that point at the same time (assuming their thermal velocity is small and it is not a collisional flow). Click on the image to play an animation of the effect.

```
variable x internal 0
variable y internal 0
variable vx equal (xhi-v_x)/(1000*7.0e-9)  # timesteps and timestep-size
```

(continues on next page)

(continued from previous page)

```
variable vy equal (yhi-v_y)/(1000*7.0e-9)
create_particles air n 10000 velocity vx vy NULL x y NULL
```



The *twopass* keyword does not require a value. If used, the creation procedure will loop over the creation grid cells twice, the same as the KOKKOS package version of this command does, so that it can reallocate memory efficiently, e.g. on a GPU. If this keyword is used the non-KOKKOS and KOKKOS version will generate exactly the same set of particles, which makes debugging easier. If the keyword is not used, the non-KOKKOS and KOKKOS runs will use random numbers differently and thus generate different particles, though they will be statistically similar.

This command (or more generically styles) can take a suffix as shown at the top of this page.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

none

Related commands:

mixture command, *fix emit/face command*

Default:

The option default is global = no.

1.14.45 dimension command

Syntax:

```
dimension N
```

- N = 2 or 3

Examples:

```
dimension 2
dimension 3
```

Description:

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

2d axi-symmetric models can be run by setting the dimension to 2, and defining the lower boundary in the y-dimension to axi-symmetric via the *boundary* command.

Restrictions:

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

Related commands:

none

Default:

```
dimension 3
```

1.14.46 dump command

Ref: *dump image command*

Syntax:

```
dump ID style select-ID N file args
```

- `ID` = user-assigned name for the dump
- `style` = `particle` or `grid` or `surf` or `image`
- `select-ID` = which particles, grid cells, surface elements to dump
  - for dump style = particle or image, use a mixture ID
  - for style = grid, use a grid group ID

  - for style = surf, use a surface group ID

- `N` = dump every this many timesteps
- `file` = name of file to write dump info to
- `args` = list of arguments for a particular style
  - `particle` args = list of particle attributes

    possible attributes = id, type, proc, x, y, z, xs, ys, zs, vx, vy, vz, ke, erot, evib, p_ID, p_ID[N], c_ID, c_ID[N], f_ID, f_ID[N], v_name

    * `id` = particle ID
    * `type` = particle species
    * `proc` = ID of owning processor
    * `x,y,z` = unscaled particle coordinates
    * `xs,ys,zs` = scaled particle coordinates
    * `vx,vy,vz` = particle velocities
    * `ke,erot,evib` = translational, rotational, and vibrational energy
    * `p_ID` = custom per-particle vector with ID
    * `p_ID[N]` = Nth column of custom per-particle array with ID
    * `c_ID` = per-particle vector calculated by a compute with ID
    * `c_ID[N]` = Nth column of per-particle array calculated by a compute, with ID, `N` can include wildcard (see below)
    * `f_ID` = per-particle vector calculated by a fix with ID
    * `f_ID[N]` = Nth column of per-particle array calculated by a fix. With ID, `N` can include wildcard (see below)
    * `v_name` = per-particle vector calculated by a particle-style variable with name
  - `grid` args = list of grid attributes

    possible attributes = id, idstr, proc, xlo, ylo, zlo, xhi, yhi, zhi, c_ID, c_ID[N], f_ID, f_ID[N], v_name

    * `id` = integer form of grid cell ID
    * `idstr` = string form of 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
    * `xc,yc,zc` = coords of center of grid cell
    * `vol` = flow volume of grid cell (area in 2d)
    * `c_ID` = per-grid vector calculated by a compute with ID
    * `c_ID[N]` = Nth column of per-grid array calculated by a compute with ID, `N` can include wildcard (see below)
    * `f_ID` = per-grid vector calculated by a fix with ID
    * `f_ID[N]` = Nth column of per-grid array calculated by a fix with ID, `N` can include wildcard (see below)

    * 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 = surface 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 calculated by a compute with ID
    * c_ID[N] = Nth column of per-surf array calculated by a compute with ID, I can include wildcard (see below)
    * f_ID = per-surf vector calculated by a fix with ID
    * f_ID[N] = Nth column of per-surf array calculated by a fix with ID, N can include wildcard (see below)
    * v_name = per-surf vector calculated by a surf-style variable with name

- image args

  Discussed on *dump image* doc page

Examples:

```
dump 1 particle all 100 dump.myforce.* id type x y vx fx
dump 2 particle inflow 100 dump.%.myforce id type c_myF[3] v_ke
dump 3 grid all 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 ID for a dump is used to identify the dump in other commands. Each dump ID must be unique. The ID can only contain alphanumeric characters and underscores. You can specify multiple dumpes of the same style so long as they have different IDs. A dump can be deleted with the *undump* command, after which its ID can be re-used.

The style setting determines what quantities are written to the file and in what format. The particle, grid, surf options are for particles, grid cells, or surface elements. Settings made via the *dump_modify* command can also alter what info is included in the file and the format of individual values.

The select-ID setting determines which particles, grid cells, or surface elements are output. For style = particle, the select-ID is a mixture ID as defined by the *mixture* command. Only particles whose species are part of the mixture are output. For style = grid, the select-ID is for a grid group, as defined by the *group grid* command. Only grid cells in the group are output. For style = surf, the select-ID is for a surface eleemnt group, as defined by the *group surf* command. Only surface elements in the group are output.

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 precision of values output to text-based dump files can be controlled by the *dump_modify format* command and its options.

The `particle` 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 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: NUMBER OF ATOMS" or "ITEM: NUMBER OF CELLS" or "ITEM: NUMBER OF SURFS" entry in each snapshot gives the number of particles, grid cells, surfaces to follow.

The "ITEM: ATOMS" or "ITEM: CELLS" or "ITEM: SURFS" entry in each snapshot lists column descriptors for the per-particle or per-grid or per-surf lines that follow. The descriptors are the attributes specied in the dump command for the style. 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 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.

Warning: The file "binary2txt.cpp" is not currently shipped with SPARTA

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.

Note that in the discussion which follows, for styles which can reference values from a compute or fix, like the particle, grid, or surf styles, the bracketed index I can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "n*" or "m*n". If N = the size of the vector (for mode = scalar) or the number of columns in the array (for mode = vector), then an asterisk with no numeric values means all indices from 1 to N. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from n to N (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard 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, since the *compute grid* command creates a per-grid array with 3 columns:

```
compute myGrid all all u v w
dump 2 grid all 100 tmp.dump id c_myGrid[*]
dump 2 grid all 100 tmp.dump id c_myGrid[1] c_myGrid[2] c_myGrid[3]
```

Particle attributes

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

Id is the particle ID. Type is an integer index representing the particle species. It is a value from 1 to Nspecies, The value corresponds to the order in which species were defined via the *species* command. Proc is the ID of the processor which currently owns the particle.

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

Vx, Vy, Vz are components of particle velocity. The ke, erot, and evib attributes are the kinetic, rotational, and vibrational energies of the particle. A particle's kinetic energy is given by 1/2 m (vx^2 + vy^2 + vz^2). The way that rotational and vibrational energy is treated in collisions and stored by particles is affected by the *collide_modify* command.

The p_ID and p_ID[N] attributes allow custom per-particle vectors or arrays defined by a *fix* command to be output. The ID in the attribute should be replaced by the actual ID of the custom particle attribute that the fix defines. See individual fix commands for details, e.g. the *fix ambipolar* command which defines the custom vector "ionambi" and the custom array "velambi".

If p_ID is used as an attribute, the custom attribute must be a vector, and it is output. If p_ID[N] is used, the custom attribute must be an array, and N must be in the range from 1-M, which will output the Nth column of the M-column array.

The c_ID and c_ID[I] attributes allow per-particle 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 an attribute, the compute must calculate a per-particle vector, and it is output. If c_ID[I] is used, the compute must calculate a per-particle array, and I must be in the range from 1-M, which will output the Ith column of the M-column array. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

The f_ID and f_ID[I] attributes allow vector or array per-particle 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 an attribute, the fix must calculate a per-particle vector, and it is output. If f_ID[I] is used, the fix must calculate a per-particle array, and I must be in the range from 1-M, which will output the Ith column of the M-column array. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

The v_name attribute allows per-particle 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 particle-style variable can be referenced, since it is the only style that generates per-particle values. Variables of style particle can reference per-particle 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 10* of the manual for information on how to add new compute and fix styles to SPARTA to calculate per-particle quantities which could then be output into dump files.

Grid Attributes

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

Note that dump grid will output one line (per snapshot) for 3 kinds of child cells: unsplit cells, cut cells, and sub cells of split cells. *Section 6.8* of the manual gives details of how SPARTA defines child, unsplit, cut, split, and sub cells. This is different than *compute* or *fix* commands that produce per grid information, which also include split cells in their output. The dump grid command discards that output since the sub cells of a split cell provide the needed information for further processing and visualization. Note that unsplit cells can be outside (in the flow) or inside surface objects, if they exist.

Id and idstr are two different forms of the grid cell ID. In SPARTA each grid cell is assigned a unique ID which represents its location, in a topological sense, within the hierarchical grid. This ID is stored as an integer such as 5774983, but can also be decoded into a string such as 33-4-6, which makes it easier to understand the grid hierarchy. In this case it means the grid cell is at the 3rd level of the hierarchy. Its grandparent cell was 33 at the 1st level, its parent was cell 4 (at level 2) within cell 33, and the cell itself is cell 6 (at level 3) within cell 4 within cell 33. If you specify id, the ID is printed directly as an integer. If you specify idstr, it is printed as a string.

Proc is the ID of the processor which currently 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 xc, yc, zc attributes write the coordinates of the center point of the grid cell. The zlo, zhi, zc attributes cannot be used for a 2d simulation.

The vol attribute is the flow volume of the grid cell (or area in 2d) for unsplit or cut or sub cells. *Section 4.8* of the manual gives details of how SPARTA defines unsplit and sub cells. Flow volume is the portion of the grid cell that is accessible to particles, i.e. outside any closed surface that may intersect the cell. Note that unsplit cells which are inside a surface object will have a flow volume of 0.0. Likewise a cut cell which is inside a suface object but which is intersected by surface element(s) which only touch a face, edge, or corner point of the grid cell, will have a flow volume of 0.0.

The c_ID and c_ID[I] 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 an attribute, and the compute calculates a per-grid vector, then the per-grid vector is output. If c_ID[I] is used, then I must be in the range from 1-M, which will output the Ith column of the M-column per-grid array calculated by the compute. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

The f_ID and f_ID[I] 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 an attribute, and the fix calculates a per-grid vector, then the per-grid vector is output. If f_ID[I] is used, then I must be in the range from 1-M, which will output the Ith column of the M-columne per-grid array calculated by the fix. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

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 10* 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.

Surface attributes

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[I] 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 an attribute, and the compute calculates a per-srf vector, then the per-surf vector is output. If c_ID[I] is used, then I must be in the range from 1-M, which will output the Ith column of the M-column per-surf array calculated by the compute. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

The f_ID and f_ID[I] 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 an attribute, and the fix calculates a per-surf vector, then the per-surf vector is output. If f_ID[I] is used, then I must be in the range from 1-M, which will output the Ith column of the M-column per-surf array calculated by the fix. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

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.

Important: Surf-style variables have not yet been implemented in SPARTA.

See *Section 10* 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 command, *dump_modify command*, *undump command*

Default:

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

1.14.47 dump image command

1.14.48 dump movie command

Syntax:

```
dump ID style mix-ID N file color diameter keyword value ...
```

- ID = user-assigned name for the dump
- style = *image* or *movie* = style of dump command (other styles *particle* or *grid* or *surf* are discussed on the *dump command* doc page)
- mix-ID = mixture ID for which particles to include in image
- N = dump every this many timesteps
- file = name of file to write image to
- color = particle attribute that determines color of each particle
- diameter = particle attribute that determines size of each particle
- zero or more keyword/value pairs may be appended
- keyword = *particle* or *pdiam* or *grid* or *gridx* or *gridy* or *gridz* or *surf* or *size* or *view* or *center* or *up* or *zoom* or *persp* or *box* or *gline* or *sline* or *axes* or *shiny* or *ssao*

  particle = yes/no do or do not draw particles

  pdiam value = number numeric value for particle diameter (distance units)

  grid values = color color = proc or per-grid compute or fix

  gridx values = xcoord color

  - xcoord = x value to dray yz plane of grid cells at
  - color = proc or per-grid compute or fix

  gridy values = ycoord color

  - ycoord = y value to dray xz plane of grid cells at
  - color = proc or per-grid compute or fix

  gridz values = zcoord color

  - zcoord = z value to dray xy plane of grid cells at
  - color = proc or per-grid compute or fix

  surf values = color diam

  - color = one or proc or per-surf compute or fix

- diam = diameter of 2d lines as fraction of shortest box length

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

gline values = yes/no diam = draw outline of each grid cell

- yes/no = do or do not draw grid cell outlines
- diam = diameter of grid outlines as fraction of shortest box length

sline values = yes/no diam = draw outline of each surface element

- yes/no = do or do not draw surf element outlines
- diam = diameter of surf element outlines 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 seed dfactor = SSAO depth shading

- yes/no = turn depth shading on/off
- seed = random # seed (positive integer)
- dfactor = strength of shading from 0.0 to 1.0

Examples:

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

These commands will dump shapshot images of all particles whose species are in the *mix-ID* to a file every 100 steps. The last two shell command will make a movie from the JPG files (once the run has finished) and play it in the Firefox browser:

```
dump  4 image all 100 tmp.*.jpg type type pdiam 0.2 view 90 -90
dump_modify 4 pad 4
```

```
% convert tmp*jpg tmp.gif
% firefox tmp.gif
```

Description:

Dump a high-quality ray-traced image of the simulation every N timesteps and save the images either as a sequence of JPEG or PNG or PPM files, or as a single movie file. The options for this command as well as the *dump_modify* command control what is included in the image and how it appears.

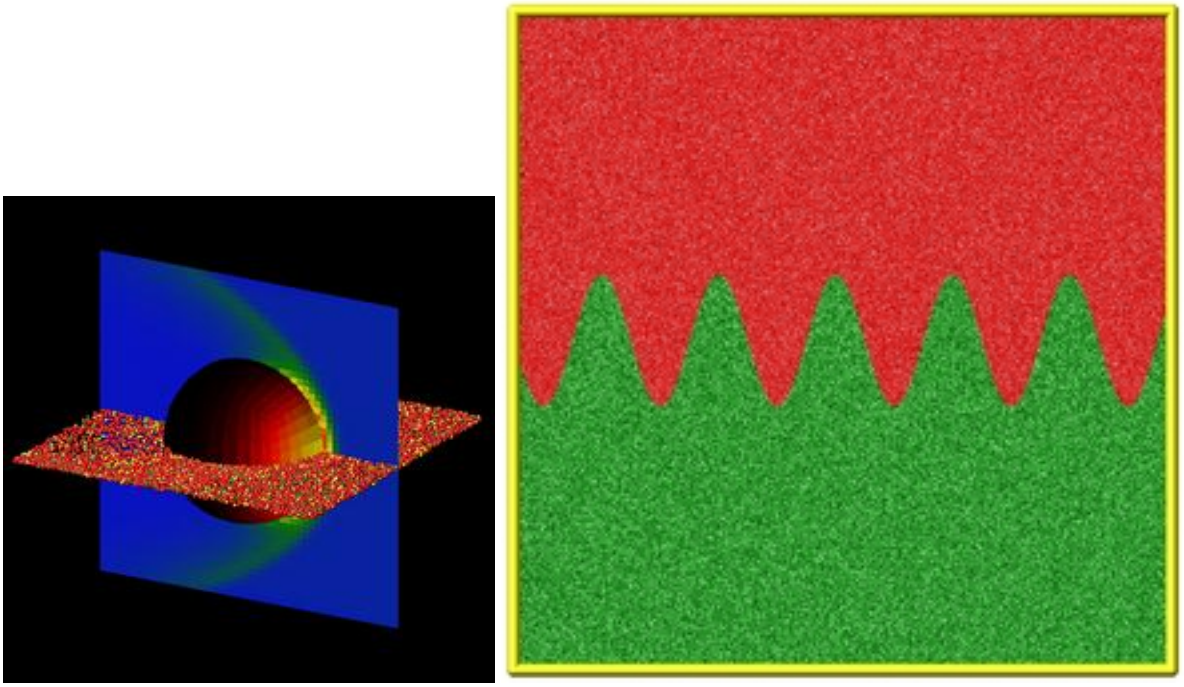
Any or all of these entities can be included in the images:

- particles (all in mixture or limited to a *region*)
- grid cells (all or limited to a *region*)
- x,y,z planes cutting through the grid
- surface elements

Particles can be colored by any attribute allowed by the *dump particle* command. Grid cells and the x,y,z cutting planes can be colored by any per-grid attribute calculated by a *compute* or *fix*. Surface elements can be colored by any per-surf attribute calculated by a *compute* or *fix*.

A series of images can easily be converted into an animated movie of your simulation (see further details below), or the process can be automated without writing the intermediate files using the dump movie command. Other dump styles store snapshots of numerical data asociated with particles, grid cells, and surfaces in various formats, as discussed on the *dump* doc page.

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



The left image is flow around a sphere with visualization of triangular surface elements on the sphere surface (colored by surface presssure), a vertical plane of grid cells (colored by particle density), and a horizontal plane of particles (colored by chemical species). The right image is the initial condition for a 2d simulation of Rayleigh-Taylor mixing as a relatively dense heavy gas (red) mixes with a light gas (green), driven by gravity in the downward direction.

The filename suffix determines whether a JPEG, PNG, or PPM file is created with the *image* dump style. If the suffix is ".jpg" or ".jpeg", then a JPEG format file is created, if the suffix is ".png", then a PNG format is created, else a PPM (aka NETPBM) format file is created. The JPEG and PNG files are binary; PPM has a text mode header followed by binary data. JPEG images have lossy compression; PNG has lossless compression; and PPM files are uncompressed but can be compressed with gzip, if SPARTA has been compiled with -DSPARTA_GZIP and a ".gz" suffix is used.

Similarly, the format of the resulting movie is chosen with the *movie* dump style. This is handled by the underlying FFmpeg converter program, which must be available on your machine, and thus details have to be looked up in the FFmpeg documentation. Typical examples are: .avi, .mpg, .m4v, .mp4, .mkv, .flv, .mov, .gif Additional settings of the movie compression like bitrate and framerate can be set using the *dump_modify* command.

To write out JPEG and PNG format files, you must build SPARTA with support for the corresponding JPEG or PNG library. To convert images into movies, SPARTA has to be compiled with the -DSPARTA_FFMPEG flag. See *Section 2.2* of the manual for instructions on how to do this.

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.

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.

Dump *movie* filenames on the other hand, must not have any wildcard character since only one file combining all images into a single movie will be written by the movie encoder.

Several of the keywords determine what objects are rendered in the image, namely particles, grid cells, or surface elements. There are additional optional kewords which 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.

Rendering of particles

Particles are drawn by default using the *color* and *diameter* settings. The *particle* keyword allow you to turn off the drawing of all particles, if the specified value is *no*. Only particles in a geometric region can be drawn using the *dump_modify region* command.

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 particle* command, including *type*.

The *diameter* setting can be overridden with a numeric value by the optional *pdiam* keyword, in which case you can specify the *diameter* setting with any valid particle attribute. The *pdiam* keyword overrides the *diameter* setting 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.

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 = purple

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

If *proc* is specified for the *color* setting, then the color of each particle is determined by the ID of the owning processor. The default mapping of proc IDs to colors is that same as in the list above, except that proc P corresponds to type P+1.

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 *proc* is specified for the *diameter* setting then the diameter of each particle will be the proc ID (0 up to Nprocs-1) in whatever *units* you are using, which is undoubtably not what you want.

Any of the particle attributes listed in the *dump custom* command can also be 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-particle value with a specific color is determined by a "color map", which can be specified via the *dump_modify cmap* 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-particle value <= 0.0, them the particle will not be drawn.

Rendering of grid cells

The *grid* keyword turns on the drawing of grid cells with the specified color attribute. For 2d, the grid cell is shaded with an rectangle that is infinitely thin in the z dimension, which allows you to still see the particles in the grid cell. For 3d, the grid cell is drawn as a solid brick, which will obscure the particles inside it.

Only grid cells in a geometric region can be drawn using the *dump_modify region* command.

The *gridx* and *gridy* and *gridz* keywords turn on the drawing of of a 2d plane of grid cells at the specified coordinate. This is a way to draw one or more slices through a 3d image.

The *dump_modify region* command does not apply to the *gridx* and *gridy* and *gridz* plane drawing.

If *proc* is specified for the *color* setting, then the color of each grid cell is determined by its owning processor ID. This is useful for visualizing the result of a load balancing of the grid cells, e.g. by the *balance_grid* or *fix balance* commands. By default the mapping of proc IDs to colors is as follows:

- proc ID 1 = red
- proc ID 2 = green
- proc ID 3 = blue
- proc ID 4 = yellow
- proc ID 5 = aqua
- proc ID 6 = purple

and repeats itself for IDs > 6. Note that for this command, processor IDs range from 1 to Nprocs inclusive, instead of the more customary 0 to Nprocs-1. This mapping can be changed by the *dump_modify gcolor* command.

The *color* setting can also be a per-grid compute or fix. In this case, it is specified as *c_ID* or *c_ID[N]* for a compute and as *f_ID* and *f_ID[N]* for a fix.

This allows per grid cell values in a vector or array to be used to color the grid cells. The ID in the attribute should be replaced by the actual ID of the compute or fix that has been defined previously in the input script. See the *compute* or *fix* command for details.

If *c_ID* is used as a attribute, then the per-grid vector calculated by the compute 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 per-grid array calculated by the compute.

If *f_ID* is used as a attribute, then the per-grid vector calculated by the fix 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 per-grid array calculated by the fix.

The manner in which values in the vector or array are mapped to color is determined by the *dump_modify cmap* command.

Rendering of surface elements

The *surf* keyword turns on the drawing of surface elements with the specified color attribute. For 2d, the surface element is a line whose diameter is specified by the *diam* setting as a fraction of the minimum simulation box length. For 3d it is a triangle and the *diam* setting is ignored. The entire surface is rendered, which in 3d will hide any grid cells (or fractions of a grid cell) that are inside the surface.

The *dump_modify region* command does not apply to surface element drawing.

If *one* is specified for the *color* setting, then the color of every surface element is drawn with the color specified by the *dump_modify scolor* keyword, which is gray by default.

If *proc* is specified for the *color* setting, then the color of each surface element is determined by its owning processor ID. Surface elements are assigned to owning processors in a round-robin fashion. By default the mapping of proc IDs to colors is as follows:

- proc ID 1 = red
- proc ID 2 = green
- proc ID 3 = blue
- proc ID 4 = yellow
- proc ID 5 = aqua
- proc ID 6 = purple

and repeats itself for IDs > 6. Note that for this command, processor IDs range from 1 to Nprocs inclusive, instead of the more customary 0 to Nprocs-1. This mapping can be changed by the *dump_modify scolor* command, which has not yet been added to SPARTA.

The *color* setting can also be a per-surf compute or fix. In this case, it is specified as *c_ID* or *c_ID[N]* for a compute and as *f_ID* and *f_ID[N]* for a fix.

This allows per-surf values in a vector or array to be used to color the surface elemtns. The ID in the attribute should be replaced by the actual ID of the compute or fix that has been defined previously in the input script. See the *compute* or *fix* command for details.

If *c_ID* is used as a attribute, then the per-surf vector calculated by the compute 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 per-surf array calculated by the compute.

If *f_ID* is used as a attribute, then the per-surf vector calculated by the fix 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 per-surf array calculated by the fix.

The manner in which values in the vector or array are mapped to color is determined by the *dump_modify cmap* command.

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 particles 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: 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 *gline* keyword determines how the outlines of grid cells are rendered as thin cylinders in the image. If the *gridx* or *gridy* or *gridz* keywords are specified to draw a plane(s) of grid cells, then outlines of all cells in the plane(s) are drawn. If the planar options are not used, then the outlines of all grid cells are drawn, whether the *grid* keyword is specified or not. In this case, the *dump_modify region* command can be used to restrict which grid cells the outlines are drawn for.

For the *gline* keywork, if *no* is set, then grid outlines are not drawn and the *diam* setting is ignored. If *yes* is set, the 12 edges of each grid cell 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 grid cell outlines can be set with the *dump_modify glinecolor* command.

The *sline* keyword determines how the outlines of surface elements are rendered as thin cylinders in the image. If *no* is set, then the surface element outlines are not drawn and the *diam* setting is ignored. If *yes* is set, a line is drawn for 2d and a triangle outline for 3d surface elements, 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 surface element outlines can be set with the *dump_modify slinecolor* 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 JPEG, PNG, or PPM images can be converted into a movie file and then played as a movie using commonly available tools. Using dump style *movie* automates this step and avoids the intermediate step of writing (many) image snapshot file.

To manually convert JPEG, PNG or PPM files into an animated GIF or MPEG or other movie file you can:

- a) Use the ImageMagick convert program.

```
% convert *.jpg foo.gif
% convert -loop 1 *.ppm foo.mpg
```

  Animated GIF files from ImageMagick are unoptimized. You can use a program like gifsicle to optimize and massively shrink them. MPEG files created by ImageMagick are in MPEG-1 format with rather inefficient compression and low quality.

- 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. QuickTime can generate very high quality and efficiently compressed movie files. Some of the supported formats require to buy a license and some are not readable on all platforms until specific runtime libraries are installed.

- c) Use FFmpeg

  FFmpeg is a command line tool that is available on many platforms and allows extremely flexible encoding and decoding of movies.

```
cat snap.*.jpg | ffmpeg -y -f image2pipe -c:v mjpeg -i - -b:v 2000k movie.m4v
cat snap.*.ppm | ffmpeg -y -f image2pipe -c:v ppm -i - -b:v 2400k movie.avi
```

  Frontends for FFmpeg exist for multiple platforms. For more information see the FFmpeg homepage

---

You can play a movie file as follows:

- 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 or ffplay tool to view a movie. Both are available for multiple OSes and support a large variety of file formats and decoders.

```
% mplayer foo.mpg
% ffplay bar.avi
```

- 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- or MacOS-based media players can obviously play movie files directly. Similarly for corresponding tools bundled with Linux desktop environments. However, due to licensing issues with some file formats, the formats may require installing additional libraries, purchasing a license, or may not be supported.

Restrictions:

To write JPEG images, you must use the -DSPARTA_JPEG switch when building SPARTA and link with a JPEG library. To write PNG images, you must use the -DSPARTA_PNG switch when building SPARTA and link with a PNG library.

To write *movie* files, you must use the -SPARTA_FFMPEG switch when building SPARTA. The FFmpeg executable must also be available on the machine where SPARTA is being run. Typically it's name is lowercase, i.e. ffmpeg.

See *Steps to build a SPARTA executable using make:* and *Steps to build a SPARTA executable using CMake:* sections of the documentation for details on how to compile with optional switches.

Note that since FFmpeg is run as an external program via a pipe, SPARTA has limited control over its execution and no knowledge about errors and warnings printed by it. Those warnings and error messages will be printed to the screen only. Due to the way image data is communicated to FFmpeg, it will often print the message + pipe:: Input/output error :pre + which can be safely ignored. Other warnings and errors have to be addressed according to the FFmpeg documentation. One known issue is that certain movie file formats (e.g. MPEG level 1 and 2 format streams) have video bandwith limits that can be crossed when rendering too large of image sizes. Typical warnings look like this:

```
[mpeg @ 0x98b5e0] packet too large, ignoring buffer limits to mux it
[mpeg @ 0x98b5e0] buffer underflow st=0 bufi=281407 size=285018
[mpeg @ 0x98b5e0] buffer underflow st=0 bufi=283448 size=285018
```

In this case it is recommended to either reduce the size of the image or encode in a different format that is also supported by your copy of FFmpeg, and which does not have this limitation (e.g. .avi, .mkv, mp4).

Related commands:

dump, *dump_modify*, *undump*

Default:

The defaults for the keywords are as follows:

- particle = yes
- pdiam = not specified (use diameter setting)
- grid = not specified (no drawing of grid cells)
- gridx = not specified (no drawing of x-plane of grid cells)
- gridy = not specified (no drawing of y-plane of grid cells)
- gridz = not specified (no drawing of z-plane of grid cells)
- surf = not specified (no drawing of surface elements)
- 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
- gline = no 0.0
- sline = no 0.0
- axes = no 0.0 0.0
- shiny = 1.0
- ssao = no

1.14.49 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
- these keywords apply to various dump styles

  keyword = *append* or *buffer* or *every* or *fileper* or *first* or *flush* or *format* or *nfile* or *pad* or *region* or *thresh*

  - append arg = yes or no
  - buffer arg = yes or no
  - every arg = N
    - N = dump every this many timesteps
    - N can be a variable (see below)
  - fileper arg = Np = write one file for every this many processors
  - first arg = yes or no
  - flush arg = yes or no
  - format args = line string, int string, float string, M string, or none
    - string = C-style format string
    - M = integer from 1 to N, where N = # of per-atom quantities being output
  - nfile arg = Nf = write this many files, one from each of Nf processors
  - pad arg = Nchar = Number of characters to convert timestep to
  - region arg = region-ID or “none”
    - Select the region to dump

  - thresh args = attribute operation value
    - attribute = same attributes (x,fy,etotal,sxx,etc) used by dump custom style
    - operation = "<" or "<=" or ">" or ">=" or "==" or "!="
    - value = numeric value to compare to
    - these 3 args can be replaced by the word "none" to turn off thresholding

- these keywords apply only to the (image and *movie styles*)

  keyword = *bcolor* or *bdiam* or *backcolor* or *bitrate* or *boxcolor* or *cmap* or *color* or *framerate* or *gcolor* or *glinecolor* or *pcolor* or *pdiam* or *scolor* or *slinecolor*

  - backcolor arg = color = name of color for background
  - bitrate arg = rate = target bitrate for movie in kbps
  - boxcolor arg = color = name of color for box lines
  - cmap args = mode lo hi style delta N entry1 entry2 ... entryN
    - mode = particle or grid or surf or xplane or yplane or zplane
    - 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
  - color args = name R G B
    - name = name of color
    - R,G,B = red/green/blue numeric values from 0.0 to 1.0

- framerate arg = fps

  fps = frames per second for movie

- gcolor args = proc color
  - proc = proc ID or range of IDs (see below)
  - color = name of color or color1/color2/...
- glinecolor arg = color

  color = name of color for grid cell outlines

- pcolor args = type color
- type = particle type or range of types or proc ID or range of IDs (see below)

  color = name of color or color1/color2/...

- pdiam args = type diam
  - type = particle type or range of types (see below)
  - diam = diameter of particles of that type (distance units)
- scolor args = proc color
  - proc = proc ID or range of IDs (see below)
  - color = name of color for surf one option
- slinecolor arg = color

  color = name of color for surface element outlines

Examples:

```
dump_modify 1 format line "%d %d %20.15g %g %g"
dump_modify 1 format float %20.15g
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 cmap particle 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.

These keywords apply to all dump styles unless otherwise noted. The descriptions give details.

The *append* keyword applies to all dump styles except *image* and *movie*. 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 *buffer* keyword applies only all dump styles except *image* and *movie*. It also applies only to text output files, not to binary or gzipped files. If specified as *yes*, which is the default, then each processor writes its output into an internal text buffer, which is then sent to the processor(s) which perform file writes, and written by those processors(s) as one large chunk of text. If specified as *no*, each processor sends its per-atom data in binary format to the processor(s) which perform file wirtes, and those processor(s) format and write it line by line into the output file.

The buffering mode is typically faster since each processor does the relatively expensive task of formatting the output for its own atoms. However it requires about twice the memory (per processor) for the extra buffering.

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 variable*, as examples of useful functions to use in this context. Other similar math functions could easily be added as options for *equal-style variable*. 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.

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 particle all 100 tmp.dump id type x y z
dump_modify 1 every v_s first yes
```

The *fileper* keyword documented below with the *nfile* keyword.

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 applies to all dump styles except *image* and *movie*. It also applies only when the styles are used to write multiple successive snapshots to the same file. It 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.

The *format* keyword can be used to change the default numeric format output by the text-based dump styles: *particle*, *grid*, *surf*.

All the specified format strings are C-style formats, e.g. as used by the C/C++ printf() command. The *line* keyword takes a single argument which is the format string for an entire line of output with N fields for each particle, grid cell, or suraface elememt, which you must enclose in quotes if it is more than one field. The *int* and *float* keywords take a single format argument and are applied to all integer or floating-point quantities output. The setting for *M string* also takes a single format argument which is used for the Mth value output in each line, e.g. the 5th column is output in high precision for "format 5 %20.15g".

The *format* keyword can be used multiple times. The precedence is that for each value in a line of output, the *M* format (if specified) is used, else the *int* or *float* setting (if specified) is used, else the *line* setting (if specified) for that value is used, else the default setting is used. A setting of *none* clears all previous settings, reverting all values to their default format.

Note: Grid cell IDs are stored internally as 4-byte or 8-byte signed integers, depending on how SPARTA was compiled. When specifying the *format int* option you can use a "%d"-style format identifier in the format string and SPARTA will convert this to the corresponding 8-byte form it it is needed when outputting those values. However, when specifying the *line* option or *format M string* option for those values, you should specify a format string appropriate for an 8-byte signed integer, e.g. one with "%ld", if SPARTA was compiled with the -DSPARTA_BIGBIG option for 8-byte IDs.

The *nfile* or *fileper* keywords apply to all dump styles except *image* and *movie*. They can be used in conjunction with the "%" wildcard character in the specified dump file name. As explained on the *dump* command doc page, the "%" character causes the dump file to be written in pieces, one piece for each of P processors. By default P = the number of processors the simulation is running on. The *nfile* or *fileper* keyword can be used to set P to a smaller value, which can be more efficient when running on a large number of processors.

The *nfile* keyword sets P to the specified Nf value. For example, if Nf = 4, and the simulation is running on 100 processors, 4 files will be written, by processors 0,25,50,75. Each will collect information from itself and the next 24 processors and write it to a dump file.

For the *fileper* keyword the specified value of Np means write one file for every Np processors. For example, if Np = 4, every 4th processor (0,4,8,12,etc) will collect information from itself and the next 3 processors and write it to a dump file.

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 *region* keyword only applies to the dump *particle* and *image* styles. If specified, only particles in the region will be written to the dump file or included in the image. Only one region can be applied as a filter (the last one specified). See the *region* command for more details. Note that a region can be defined as the "inside" or "outside" of a geometric shape, and it can be the "union" or "intersection" of a series of simpler regions.

The *thresh* keyword only applies to the dump *particle* and *image* styles. Multiple thresholds can be specified. Specifying "none" turns off all threshold criteria. If thresholds are specified, only particles 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 particles whose velocity components are above some threshold.

These keywords apply only to the *dump image command* and *command-dump-movie* styles. Any keyword that affects an image, also affects a movie, since the movie is simply a collection of images. Some of the keywords only affect the *command-dump-movie* style. The descriptions give details.

The *backcolor* keyword can be used with the *dump image command* to set 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 *bitrate* keyword can be used with the *command-dump-movie* to define the size of the resulting movie file and its quality via setting how many kbits per second are to be used for the movie file. Higher bitrates require less compression and will result in higher quality movies. The quality is also determined by the compression format and encoder. The default setting is 2000 kbit/s, which will result in average quality with older compression formats.

Important: Not all movie file formats supported by dump movie allow the bitrate to be set. If not, the setting is silently ignored.

The *boxcolor* keyword can be used with the *dump image command* to set the color of the simulation box drawn around the particles 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 *cmap* keyword can be used with the *dump image command* command to define a color map that is used to draw "objects" which can be particles, grid cells, or surface elements. The mode setting must be *particle* or *grid* or *surf* or *gridx* or *gridy* or *gridz* which correspond to the same keywords in the *dump image command*.

Color maps are used to assign a specific RGB (red/green/blue) color value to an individual object when it is drawn, based on the object's attribute, which is a numeric value, e.g. the x-component of velocity for a particle, if the particle-attribute "vx" was specified in the *dump image command*.

The basic idea of a color map is that the attribute will be within a range of values, and that range is associated with a a series of colors (e.g. red, blue, green). A 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 *cmap* keyword. Here are the details.

The *lo* and *hi* settings determine the range of values allowed for the 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 objects 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 objects 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 object, given the value X of its attribute. X will fall between 2 of the entry values. The color of the object 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 object'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 object, given the value X of its 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 objects 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 object, given the value X of its 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 object 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 *color* keyword can be used with the *dump image command* to define 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 *framerate* keyword can be used with the *command-dump-movie* to define the duration of the resulting movie file. Movie files written by the dump *movie* command have a default frame rate of 24 frames per second and the images generated will be converted at that rate. Thus a sequence of 1000 dump images will result in a movie of about 42 seconds. To make a movie run longer you can either generate images more frequently or lower the frame rate. To speed a movie up, you can do the inverse. Using a frame rate higher than 24 is not recommended, as it will result in simply dropping the rendered images. It is more efficient to dump images less frequently.

The *gcolor* keyword can be used one or more times with the *dump image command*, only when its grid color setting is *proc*, to set the color that grid cells will be drawn in the image.

The *proc* setting should be an integer from 1 to Nprocs = the number of processors. A wildcard asterisk can be used in place of or in conjunction with the *proc* argument to specify a range of processor IDs. This takes the form "*" or "*n" or "n*" or "m*n". If N = the number of processors, then an asterisk with no numeric values means all procs from 1 to N. A leading asterisk means all procs from 1 to n (inclusive). A trailing asterisk means all procs from n to N (inclusive). A middle asterisk means all procs from m to n (inclusive). Note that for this command, processor IDs range from 1 to Nprocs inclusive, instead of the more customary 0 to Nprocs-1.

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 processors. In the latter case, the list of colors are assigned in a round-robin fashion to each of the specified processors.

The *glinecolor* keyword can be used with the *dump image command* to set the color of the grid cell outlines drawn around the grid cells in each image. See the "dump image gline" command for how to specify that cell outlines 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 *pcolor* keyword can be used one or more times with the *dump image command*, only when its particle color setting is *type* or *procs*, to set the color that particles will be drawn in the image.

If the particle color setting is *type*, then the specified *type* for the *pcolor* keyword should be an integer from 1 to Ntypes = the number of particle types. A wildcard asterisk can be used in place of or in conjunction with the *type* argument to specify a range of particle types. This takes the form "*" or "*n" or "n*" or "m*n". If N = the number of particle 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).

If the particle color setting is *proc*, then the specified *type* for the *pcolor* keyword should be an integer from 1 to Nprocs = the number of processors. A wildcard asterisk can be used in place of or in conjunction with the *type* argument to specify a range of processor IDs, just as described above for particle types. Note that for this command, processor IDs range from 1 to Nprocs inclusive, instead of the more customary 0 to Nprocs-1.

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 particle types. In the latter case, the list of colors are assigned in a round-robin fashion to each of the specified particle types.

The *pdiam* keyword can be used with the *dump image command*, when its particle diameter setting is *type*, to set the size that particles of each type will be drawn in the image. The specified *type* should be an integer from 1 to Ntypes. As with the *pcolor* keyword, a wildcard asterisk can be used as part of the *type* argument to specify a range of particle types. The specified *diam* is the size in whatever distance *units command* the input script is using.

The *scolor* keyword can be used one or more times with the *dump image command*, only when its surface element color setting is *one* or *proc*, to set the color that surface elements will be drawn in the image.

When the surf color is *one*, the *proc* setting for this command is ignored.

When the surf color is *proc*, the *proc* setting for this command should be an integer from 1 to Nprocs = the number of processors. A wildcard asterisk can be used in place of or in conjunction with the *proc* argument to specify a range of processor IDs. This takes the form "*" or "*n" or "n*" or "m*n". If N = the number of processors, then an asterisk with no numeric values means all procs from 1 to N. A leading asterisk means all procs from 1 to n (inclusive). A trailing asterisk means all procs from n to N (inclusive). A middle asterisk means all procs from m to n (inclusive). Note that for this command, processor IDs range from 1 to Nprocs inclusive, instead of the more customary 0 to Nprocs-1.

When the surf color is *one*, the specified *color* setting for this command must 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.

When the surf color is *proc*, the *color* setting for this command can be one or more colors separated by a "/" character, e.g. red/green/blue. For a single color, that color is assigned to all the specified processors. For two or more colors, the list of colors are assigned in a round-robin fashion to each of the specified processors.

The *slinecolor* keyword can be used with the *dump image command* to set the color of the surface element outlines drawn around the surface elements in each image. See the "dump image sline" command for how to specify that surface element outlines 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.

Restrictions:

none

Related commands:

dump command dump image command, *undump command*

Default:

The option defaults are

- append = no
- buffer = yes for all dump styles except *image* and *movie*
- backcolor = black
- boxcolor = yellow
- cmap = mode min max cf 0.0 2 min blue max red, for all modes
- color = 140 color names are pre-defined as listed below
- every = whatever it was set to via the *dump command*
- fileper = # of processors
- first = no
- flush = yes
- format = %d and %g for each integer or floating point value
- gcolor = * red/green/blue/yellow/aqua/cyan

- glinecolor = white
- nfile = 1
- pad = 0
- pcolor = * red/green/blue/yellow/aqua/cyan
- pdiam = * 1.0
- region = none
- scolor = * gray
- slinecolor = white
- thresh = none

These are the 140 colors that SPARTA pre-defines for use with the *dump image command* and `dump_modify` command. 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.

Table 2: Pre-defined colors

aliceblue = 240, 248, 255	antiquewhite = 250, 235, 215	aqua = 0, 255, 255	aquamarine = 127, 2
azure = 240, 255, 255	beige = 245, 245, 220	bisque = 255, 228, 196	black = 0, 0, 0
blanchedalmond = 255, 255,	05 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
coral = 255, 127, 80	cornflowerblue = 100, 149, 2	7 cornsilk = 255, 248, 220	crimson = 220, 20, 6
cyan = 0, 255, 255	darkblue = 0, 0, 139	darkcyan = 0, 139, 139	darkgoldenrod = 184
darkgray = 169, 169, 169	darkgreen = 0, 100, 0	darkkhaki = 189, 183, 107	darkmagenta = 139,
darkolivegreen = 85, 107, 4	darkorange = 255, 140, 0	darkorchid = 153, 50, 204	darkred = 139, 0, 0
darksalmon = 233, 150, 122	darkseagreen = 143, 188, 14	darkslateblue = 72, 61, 139	darkslategray = 47, 7
darkturquoise = 0, 206, 209	darkviolet = 148, 0, 211	deeppink = 255, 20, 147	deepskyblue = 0, 19
dimgray = 105, 105, 105	dodgerblue = 30, 144, 255	firebrick = 178, 34, 34	floralwhite = 255, 25
forestgreen = 34, 139, 34	fuchsia = 255, 0, 255	gainsboro = 220, 220, 220	ghostwhite = 248, 24
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,
indigo = 75, 0, 130	ivory = 255, 240, 240	khaki = 240, 230, 140	lavender = 230, 230,
lavenderblush = 255, 240,	45 lawngreen = 124, 252, 0	lemonchiffon = 255, 250, 205	lightblue = 173, 216
lightcoral = 240, 128, 128	lightcyan = 224, 255, 255	lightgoldenrodyellow = 250, 250,	10 lightgreen = 144,
lightgrey = 211, 211, 211	lightpink = 255, 182, 193	lightsalmon = 255, 160, 12	lightseagreen = 32, 1
lightskyblue = 135, 206, 250	lightslategray = 119, 136,	53 lightsteelblue = 176, 196,	lightyellow = 255, 2
lime = 0, 255, 0	limegreen = 50, 205, 50	linen = 250, 240, 230	magenta = 255, 0, 25
maroon = 128, 0, 0	mediumaquamarine = 102, 205, 170	mediumblue = 0, 0, 205	mediumorchid = 186
mediumpurple = 147, 112, 21	mediumseagreen = 60, 179,	13 mediumslateblue = 123, 104, 238	mediumspringgreen
54 mediumturquoise = 72, 209,	04 mediumvioletred = 199, 21,	midnightblue = 25, 25, 112	mintcream = 245, 25
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, 17	palegreen = 152, 251
paleturquoise = 175, 238, 2	palevioletred = 219, 112,	47 papayawhip = 255, 239, 213	peachpuff = 255, 23
peru = 205, 133, 63	pink = 255, 192, 203	plum = 221, 160, 221	powderblue = 176, 2
purple = 128, 0, 128	red = 255, 0, 0	rosybrown = 188, 143, 143	royalblue = 65, 105,
saddlebrown = 139, 69, 19	salmon = 250, 128, 114	sandybrown = 244, 164, 96	seagreen = 46, 139,
seashell = 255, 245, 238	sienna = 160, 82, 45	silver = 192, 192, 192	skyblue = 135, 206,

Continued on n

Table 2 – continued from previous page

slateblue = 106, 90, 205	slategray = 112, 128, 144	snow = 255, 250, 250	springgreen = 0, 255
steelblue = 70, 130, 180	tan = 210, 180, 140	teal = 0, 128, 128	thistle = 216, 191, 2
tomato = 253, 99, 71	turquoise = 64, 224, 208	violet = 238, 130, 238	wheat = 245, 222, 1
white = 255, 255, 255	whitesmoke = 245, 245, 245	yellow = 255, 255, 0	yellowgreen = 154, 2

1.14.50 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
```

1.14.51 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 warn
fix 1 ave/time all 100 5 1000 c_myTemp c_thermo_temp file temp.profile
```

Description:

Set a fix that will be applied to the system. In SPARTA, a "fix" is an operation that is applied to the system during timestepping. Examples include adding particles via inlet boundary conditions or computing diagnostics. Code for new fixes can be added to SPARTA; see *Section 10* of the manual for details.

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 for a fix is used to identify the fix in other commands. Each fix ID must be unique; see an exception below. The ID can only contain alphanumeric characters and underscores. You can specify multiple fixes of the same style so long as they have different IDs. A fix can be deleted with the *unfix* command, after which its ID can be re-used.

Important: The *unfix* command is the only way to turn off a fix; simply specifying a new fix with the same style and a different ID 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.

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.

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

- *adapt* - on-the-fly grid adaptation
- *adapt/kk* - Kokkos version of fix adapt
- *ambipolar* - ambipolar approximation for ionized plasmas
- *ave/grid* - compute per grid cell time-averaged quantities
- *ave/grid/kk* - Kokkos version of fix ave/grid
- ave/histo - compute/output time averaged histograms
- ave/histo/weight - compute/output weighted histograms
- *ave/surf* - compute per surface element time-averaged quantities
- *ave/time* - compute/output global time-averaged quantities
- *balance* - perform dynamic load-balancing

- *balance/kk* - Kokkos version of fix balance
- *emit/face* - emit particles at global boundaries
- *emit/face/kk* - Kokkos version of fix emit/face
- *emit/face/file* - emit particles at global boundaries using a distribution defined in a file
- *emit/surf* - emit particles at surfaces
- *grid/check* - check if particles are in the correct grid cell
- *grid/check/kk* - Kokkos version of fix grid/check
- *move/surf* - move surfaces dynamically during a simulation
- *move/surf/kk* - Kokkos version of fix move/surf
- *print* - print text and variables during a simulation
- *vibmode* - discrete vibrational energy modes

There are also additional accelerated compute styles included in the SPARTA distribution for faster performance on specific hardware. The list of these with links to the individual styles are given in the pair section of this page.

In addition to the operation they perform, some fixes also produce one of four styles of quantities: global, per-particle, per-grid, or per-surf. These can be used by other commands or output as described below. A global quantity is one or more system-wide values, e.g. the temperature of the system. A per-particle quantity is one or more values per particle, e.g. the kinetic energy of each particle. A per-grid quantity is one or more values per grid cell. A per-surf quantity is one or more values per surface element.

Global, per-particle, per-grid, and per-surf quantities each come in two forms: a single scalar value or a vector of values. Additionaly, global quantities can also be a 2d array of values. The doc page for each fix describes the style and kind of values it produces, e.g. a per-particle vector. Some fixes can produce more than one form 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.

Any 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-particle values can be output via the *dump particle* command. Or the per-particle values can be referenced in an *particle-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*.

Restrictions:

none

Related commands:

unfix command

Default:

none

1.14.52 fix ablate command

Syntax:

```
fix ID ablate group-ID Nevery scale source maxrandom
```

- ID is documented in *fix* command
- ablate = style name of this fix command
- group-ID = ID of group of grid cells that contain implicit surfaces
- Nevery = perform ablation once every Nevery steps
- scale = scale factor to convert source to grid corner point value decrement
- source = computeID or fixID or random

```
computeID = c_ID or c_ID[n] for a compute that calculates per grid cell values
fixID = f_ID or f_ID[n] for a fix that calculates per grid cell values
random = perform a random decrement
```

- maxrandom = maximum per grid cell decrement as an integer (only specified if source = random)

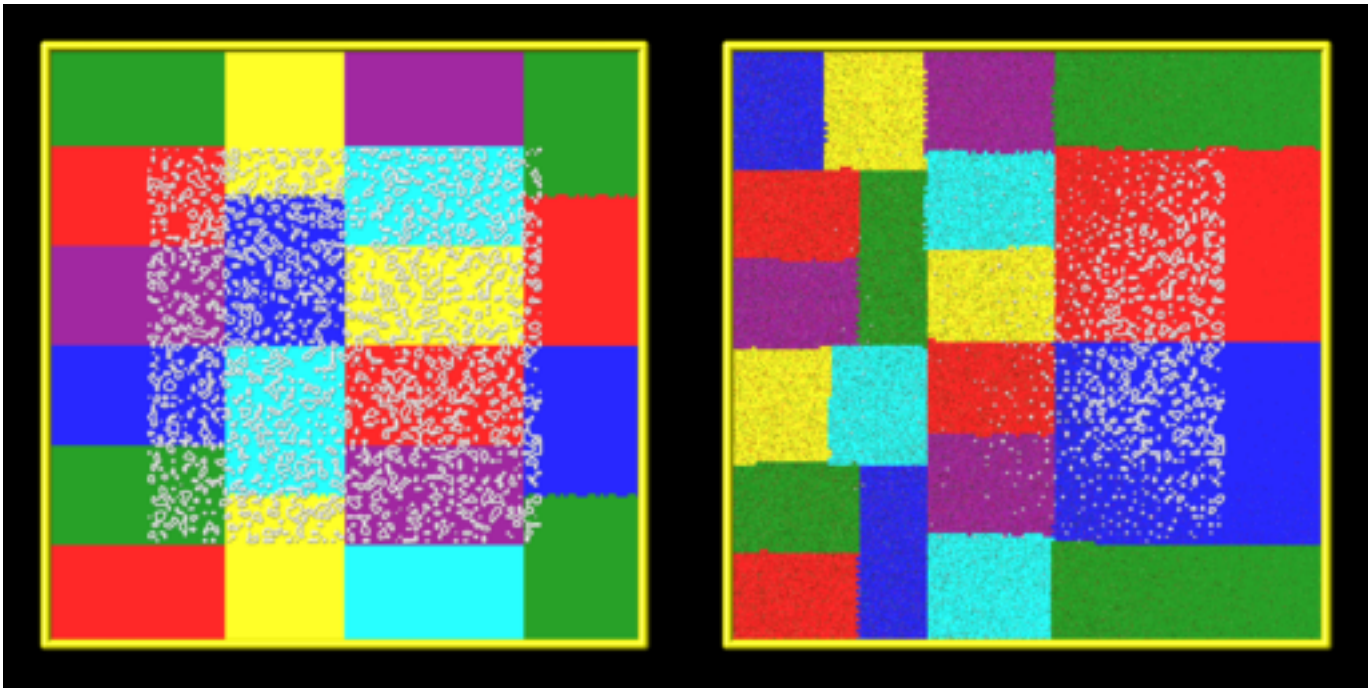
Examples:

```
fix 1 ablate surfcells 0 0.0 random 10
fix 1 ablate surfcells 1000 10.0 c_tally
```

Description:

Perform ablation once every Nevery steps on a set of grid cell corner points to induce new implicit surface elements in those grid cells. This command is also used as an argument to the *read_isurf* command so that the grid corner point values it reads from a file can be assigned to and stored by each grid cell.

Here are simulation snapshots of 2d and 3d implicit surface models through which particles flow. Click on any image for a larger image. The 1st and 3rd images are the initial states of the porous media. The 2nd and 4th images are snapshots midway through an ablation simulation. In the 2d case, the colorings are by processor for sub-domains each owns. Particles flow from left to right. The implicit triangles for the 3d case were created via Marching Cubes (discussed on the *read_isurf* command doc page) from a tomographic image of a sample of NASA FiberForm (TM) material, used as a heat shield material on spacecraft. Particles flow from top to bottom.



The specified *group-ID* must be the name of a grid cell group, as defined by the *group grid* command, which contains a set of grid cells, all of which are the same size, and which comprise a contiguous 3d array. It must be the same as group-ID used with the *read_isurf* command, which specifies its *Nx* by *Ny* by *Nz* extent. See the *read_isurf* command for more details. This command reads the initial values for grid cell corner points, which are stored by this fix.

The specfied *Nevery* determines how often an ablation operation is performed. If *Nevery* = 0, ablation is never performed. The grid cell corner point values and the surface elements they induce will remain static for the duration of subsequent simulations.

The specified *scale* is a pre-factor on the specified *source* of ablation strength. It converts the per grid cell numeric quantities produced by the *source* (which may have associated units) to a unitless decrement value for the grid cell corner points, which range from 0 to 255 inclusive. A value of 255 represents solid material and a value of 0 is void (flow volume for particles). Values in between represent partially ablated material.

The *source* can be specified as a per grid cell quantity calculated by a compute, such as *compute isurf/grid*, e.g. the number of collisions of particles with the surfaces in each grid cell or the amount of energy transferred to the surface by the collisions. It can also be specified a per grid cell quantity calculated by a fix, such as *fix ave/grid*. That fix could time average per-grid cell quantities from per grid cell computes. In that case the *scale* factor should account for applying a time-averaged quantity at an interval of *N* steps.

For debugging purposes, the *source* can also be specified as *random* with an additional integer *maxrandom* value also specified. In this case, the *scale* factor should be floating point value between 0.0 and 1.0. Each time ablation is

performed, two random numbers are generated for each grid cell. The first is a random value between 0.0 and 1.0. The second is a random integer between 1 and maxrandom. If the first random # < *scale*, then the second random integer is the decrement value for the cell. Thus *scale* is effectively the fraction of grid cells whose corner point values are decremented.

Here is an example of commands that will couple ablation to surface reaction statistics to modulate ablation of a set of implicit surfaces. These lines are taken from the examples/ablation/in.ablation.3d.reactions input script:

```
surf_collide       1 diffuse 300.0 1.0
surf_react      2 prob air.surf
```

```
compute             10 react/isurf/grid all 2
fix                 10 ave/grid all 1 100 100 c_10*
dump                10 grid all 100 tmp.grid id c_101
```

```
global              surfs implicit
fix                 ablate ablate all 100 2.0 c_101   # could be f_10
read_isurf          all 20 20 20 binary.21x21x21 99.5 ablate
```

```
surf_modify         all collide 1 react 2
```

The order of these commands matter, so here is the explanation.

The *surf_modify* command must come after the *read_isurf* command, because surfaces must exist before assigning collision and reaction models to them. The *fix ablate* command must come before the *read_isurf* command, since it uses the ID of the *fix ablate command* as an argument to create implicit surfaces. The *fix ablate command* takes a compute or fix as an argument, in this case the ID of the *compute react/isurf/grid* command. This is to specify what calculation drives the ablation. In this case, it is the *compute react/isurf/grid* command (or could be the *fix ave/grid* command) which tallies counts of surface reactions for implicit triangles in each grid cell. The *compute react/isurf/grid* react/isurf/grid command requires the ID of a surface reaction model, so that it knows the list of possible reactions to tally. In this case the reaction is set by the *surf_react* command, which must therefore comes near the beginning of this list of commands.

As explained on the *read_isurf* doc page, the marching cubes (3d) or marching squares (2d) algorithm is used to convert a set of grid corner point values to a set of implicit triangles in each grid cell which represent the current surface of porous material which is undergoing dynamic ablation. This uses a threshold value, defined by the *read_isurf* command, to set the boundary between solid material and void.

The ablation operation decrements the corner point values of each grid cell containing porous material. The marching cubes or squares algorithm is re-invoked on the new corner point values to create a new set of implicit surfaces, which effectively recess due to the decrement produced byt the ablative *source* factor.

The manner in which the per-grid source decrement value is applied to the grid corner points is as follows. Note that each grid cell has 4 (2d) or 8 (3d) corner point values. Except at the boundary of the 2d of 3d array of grid cells containing porous materials, each corner point is similarly shared by 4 (2d) or 8 (3d) grid cells.

Within each grid cell, the decrement value is subtracted from the smallest corner point value. Except that a corner point value cannot become smaller than 0.0. If this would occur, only a portion of the decrement is used to set the corner point to 0.0; the remainder is applid to the next smallest corner point value. And so forth on successive corner points until all of the decrement is used.

The amount of decrement applied to each corner point is next shared between all the grid cells (4 or 8) sharing each corner point value. The sum of those decrements is subtracted from the corner point, except that it's final value is set no smaller than 0.0. All the copies of each corner point value are now identical.

Finally, no corner point value can be nearly equal to the marching cubes/squares threshold value, else line segments or triangles of zero or epsilon size will result. So corner points with values X where thresh-epsilon < X < thresh+epsilon are reset to thresh-epsilon. Thresh is defined by the *read_isurf* command. Epsilon is set to 1.0e-4 in src/fix_ablate.cpp. Note that this is on the scale of corner point values from 0 to 255.

Restart, output info:

No information about this fix is written to *binary restart files*.

This fix computes a global scalar and a global vector of length 2. The global scalar is the current sum of unique corner point values across the entire grid (not counting duplicate values). This sum assumes that corner point values are 0.0 on the boundary of the 2d or 3d array of grid cells containing implicit surface elements.

The 2 vector values are the (1) sum of decrement values for each grid cell in the most recent ablation operation, and (2) the # of particles deleted during the most recent ablation operation that ended up "inside" the newly ablated surface. The latter quantity should be 0. A non-zero value indicates a corner case in the marching cubes or marching squares algorithm the developers still need to address.

These values can be accessed by any command that uses global values from a fix as input. See *Section 6.4* for an overview of SPARTA output options.

The scalar and vector values are unitless.

Restrictions:

This fix can only be used in simulations that define implicit surfaces.

Related commands:

read_isurf command

Default:

none

1.14.53 fix adapt command

1.14.54 fix adapt/kk command

Syntax:

```
fix ID adapt Nfreq args ...
```

- ID is documented in *fix* command
- adapt = style name of this fix command
- Nfreq = perform grid adaptation every this many steps
- args = all remaining args are identical to those defined for the *adapt_grid* command

Examples:

```
fix 1 adapt 1000 all refine particle 10 50
fix 1 adapt 1000 all coarsen particle 10 50
fix 1 adapt 500 subset refine coarsen particle 10 50
fix 1 adapt 10000 all refine surf 0.15 iterate 1 dir 1 0 0
fix 10 adapt 1000 all refine coarsen value c_11 5.0 10.0 iterate 2
```

Description:

This command performs on-the-fly adapatation of grid cells as a simulation runs, either by refinement or coarsening or both. Grid adaptation can also be performed before or between simulations by using the *adapt_grid* command.

Refinement means splitting one child cell into multiple new child cells; the original child cell becomes a parent cell. Coarsening means combining all the child cells of a parent cell, so that the child cells are deleted and the parent cell becomes a single new child cell. See *Section howto 4.8* for a description of the hierarchical grid used by SPARTA and a defintion of child and parent cells.

Grid adaptation can be useful for adjusting the grid cell sizes to the current particle density distribution, or mean-free-path of particles, or to other simulation attributes such as the presence of surface elements. A well-adapted grid can improve accuracy of the simulation and/or reduce a simulation's computational cost.

Adaptation is performed by this command once every *Nfreq* timesteps.

All of the command arguments which appear after *Nfreq*, which determine how adapation is done for both refinement and coarsening, are exactly the same as for the *adapt_grid* command.

This includes a group-ID parameter which can be used to limit adaptation to a subset of current grid cells. See the *adapt_grid* command doc page for details.

The one exception is that the *iterate* keyword cannot be used with the fix adapt command. Only a single iteration of the action1 and action2 parameters (described on the *adapt_grid* doc page) can be performed each time grid adaptation is performed.

Restart, output info:

No information about this fix is written to *binary restart files*.

This fix computes a global scalar which is a flag for whether any grid cells were adapted on the last timestep it was invoked. The value of the flag is 1 if any cells were refined or coarsened, else it is 0.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

none

Related commands:

adapt_grid command, *balance_grid command*

Default:

none

1.14.55 fix ambipolar command

Syntax:

```
fix ID ambipolar especies ion1 ion2 ...
```

- ID is documented in *fix* command
- ambipolar = style name of this fix command
- especies = species ID for ambipolar electrons
- ion1,ion2,... = species IDs for one or more ambipolar ions

Examples:

```
fix 1 ambipolar e N+ O+ NO+
```

Description:

Enable the ambipolar approximation to be used in a simulation. The ambipolar approximation is a computationally efficient way to model low-density plasmas which contain positively-charged ions and negatively-charged electrons. In this model, electrons are not free particles which move independently. This would require a simulation with a very small timestep due to electon's small mass and high speed (1000x that of an ion or neutral particle).

Instead each ambipolar electron is assumed to stay "close" to its parent ion, so that the plasma gas appears macroscopically neutral. Each pair of particles thus moves together through the simulation domain, as if they were a single particle, which is how they are stored within SPARTA. This means a normal timestep can be used.

An overview of how to run simulations with the ambipolar approximation is given in the *Section 6.11*. This includes gas-phase collisions and chemistry as well as surface chemistry when particles collide with surface elements or the global boundary of the simulation box. The section also lists all the commands that can be used in an input script to invoke various options associated with the ambipolar approximation. All of them depend on this fix ambipolar command being defined.

This command defines *especies* which is the species ID associated with the ambipolar electrons. It also specifies one or more species IDs as *ion1*, *ion2*, etc for ambipolar ions. SPARTA checks that the especies has a negative charge (as read in by the *species* command), and the ions have positive charges. An error is flagged if that is not the case.

Internally, this fix defines two custom particle attributes. The first is named "ionambi" and is an integer vector (one integer per particle). It stores a value of 1 for ambipolar ions, or 0 otherwise. The second is named "velambi" and

is a floating-point arrays (3 values per particle). It stores the velocity of the ambipolar electron associated with the ambipolar ion, or zeroes otherwise.

Restart, output info:

No information about this fix is written to *binary restart files*.

However, the values of the two custom particle attributes defined by this fix are written to the restart file. Namely the integer value "ionambi" and floating-point velocity values "velambi" for each particle. As explained on the *read_restart* doc page these values can be re-assigned to particles when a restart file is read, if a new fix ambipolar command is specified in the restart script before the first *run* command is used.

No global or per-particle or per-grid quantities are stored by this fix for access by various output commands.

However, the two custom particle attributes defined by this fix can be accessed by the *dump particle* command, as p_ionambi and p_velambi. That means those per-particle values can be written to particle dump files.

Restrictions:

none

Related commands:

collide_modify ambipolar yes

Default:

none

1.14.56 fix ave/grid command

1.14.57 fix ave/grid/kk command

Syntax:

```
fix ID ave/grid group-ID Nevery Nrepeat Nfreq value1 value2 ... keyword args ...
```

- `ID` is documented in *fix* command
- `ave/grid` = style name of this fix command
- `group-ID` = group ID for which grid cells to perform calculation on
- `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, I can include wildcard (see below)
- 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, I can include wildcard (see below)
- v_name = per-grid vector calculated by a 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 ave/grid all 10 20 1000 c_mine
fix 1 ave/grid all 1 100 100 c_2[1] ave running
fix 1 ave/grid all 1 100 100 c_2[*] ave running
fix 1 ave/grid section1 5 20 100 v_myEng
```

These commands will dump 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 all 1000 tmp.grid id f_1[*]
```

Description:

Use one or more per-grid vectors as inputs every few timesteps, and average 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. Only grid cells in the grid group specified by *group-ID* are included in the averaging. See the *group grid* command for info on how grid cells can be assigned to grid groups.

Each input value can be the result of a *compute* or *fix* or *grid-style variable*. The compute or fix must produce a per-grid vector or array, not a global or per-particle or per-surf quantity. If you wish to time-average global quantities from a compute, fix, or variable, then see the *fix ave/time* command. To time-average per-surf quantities, see the *fix ave/surf* command.

Each per-grid value of each input vector is averaged independently.

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.

Note that for values from a compute or fix, the bracketed index I can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "n*" or "m*n". If N = the size of the vector (for *mode* = scalar) or the number of columns in the array (for *mode* = vector), then an asterisk with no numeric values means all indices from 1 to N. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from n to N (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard 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 creates a per-grid array with 3 columns:

```
compute myGrid all all u v w
fix 1 ave/grid all 10 20 1000 c_myGrid[*]
fix 1 ave/grid all 10 20 1000 c_myGrid[1] c_myGrid[2] c_myGrid[3]
```

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[I]* is used, then I must be in the range from 1-M, which will use the Ith column of the M-column per-grid array calculated by the compute. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

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[I]* is used, then I must be in the range from 1-M, which will use the Ith column of the M-column per-grid array calculated by the fix. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

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. Only grid-style variables can be referenced. See the *variable* command for details. Note that grid-style variables define a formula which can reference *stats_style* 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.

For averaging of a value that comes from a compute or fix, normalization is performed as follows. Note that no normalization is performed on a value produced by a grid-style variable.

If the compute or fix is summing over particles in a grid cell to calculate a per-grid quantity (e.g. energy or temperature), this takes the form of a numerator divided by a denominator. For example, see the formulas discussed on the *compute grid* doc page, where the denominator is 1 (for keyword n), or the number of particles (ke, mass, temp), or the sum of particle masses (u, usq, etc). When this command averages over a series of timesteps, the numerator and denominator are summed separately. This means the numerator/denominator division only takes place when this fix produces output, every Nfreq timesteps.

For example, say the Nfreq output is over 2 timesteps, and the value produced by *compute grid mass* is being averaged. Say a grid cell has 10 particles on the 1st timestep with a numerator value of 10.0, and 100 particles on the 2nd timestep with a numerator value of 50.0. The output of this fix will be (10+50) / (10+100) = 0.54, not ((10/10) + (50/100)) / 2 = 0.75.

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, output info:

No information about this fix is written to *binary restart files*.

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.

This fix performs averaging for all child grid cells in the simulation, which includes unsplit, split, and sub cells. *Details of grid geometry in SPARTA* of the manual gives details of how SPARTA defines child, unsplit, split, and sub cells.

Grid cells not in the specified *group-ID* will output zeroes for all their values.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

If performing on-the-fly grid adaptation every N timesteps, using the *fix adapt* command, this fix cannot time-average across time windows > N steps, since the grid may change. This means *Nfreq* cannot be > N, and keyword *ave* = *running* is not allowed.

Related commands:

compute command, *fix ave/time command*

Default:

The option defaults are ave = one.

1.14.58 fix ave/surf command

Syntax:

```
fix ID ave/surf group-ID Nevery Nrepeat Nfreq value1 value2 ... keyword args ...
```

- `ID` is documented in *fix* command
- `ave/surf` = style name of this fix command
- `group-ID` = group ID for which surface elements to perform calculation on
- `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-surf vector (or array) calculated by a compute with ID
  - `c_ID[I]` = Ith column of per-surf array calculated by a compute with ID, I can include wildcard (see below)
  - `f_ID` = per-surf vector (or array) calculated by a fix with ID
  - `f_ID[I]` = Ith column of per-surf array calculated by a fix with ID, I can include wildcard (see below)
- 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 ave/surf all 1 100 100 c_surf ave running
fix 1 ave/surf leftcircle 10 20 1000 c_mine[2]
fix 1 ave/surf leftcircle 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, applying appropriate normalization factors. The resulting per-surf averages can be used by other output commands such as the *dump surf* command. Only surface elements in the surface group specified by *group-ID* are included in the averaging. See the *group surf* command for info on how surface elements can be assigned to surface groups.

Each input value can be the result of a *compute* or *fix*. The compute or fix must produce a per-surf vector or array, not a global or per-particle or per-grid quantity. If you wish to time-average global quantities from a compute or fix then see the *fix ave/time* command. To time-average per-grid quantities, see the *fix ave/grid* command.

Each per-surf value of each input vector is averaged independently.

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.

Note that for values from a compute or fix, the bracketed index I can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "n*" or "m*n". If N = the size of the vector (for *mode* = scalar) or the number of columns in the array (for *mode* = vector), then an asterisk with no numeric values means all indices from 1 to N. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from n to N (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual columns of the array had been listed one by one. E.g. these 2 fix ave/surf commands are equivalent, since the *compute surf* command creates a per-surf array with 4 columns:

```
compute mySurf all all n fx fy fz
fix 1 ave/surf all 10 20 1000 c_mySurf[*]
fix 1 ave/surf all 10 20 1000 c_mySurf[1] c_mySurf[2] &
      c_mySurf[3] c_mySurf[4]
```

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-surf vector, then the per-surf vector is used. If *c_ID[I]* is used, then I must be in the range from 1-M, which will use the Ith column of the M-column per-surf array calculated by the compute. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

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-surf vector, then the per-surf vector is used. If *f_ID[I]* is used, then I must be in the range from 1-M, which will use the Ith column of the M-column per-surf array calculated by the fix. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

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*.

For averaging of a value that comes from a compute or fix, normalization is performed as follows. If the compute or fix is summing over particles to calculate a per-surf quantity (e.g. pressure or energy flux), this takes the form of a numerator divided by a denominator. For example, see the formulas discussed on the *compute surf* doc page, where the denominator is 1 (for keyword n), area times dt (timestep) for the other quantities (press, shx, ke, etc). When this command averages over a series of timesteps, the numerator and denominator are summed separately. This means the numerator/denominator division only takes place when this fix produces output, every Nfreq timesteps.

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, output info:

No information about this fix is written to *binary restart files*.

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.

Surface elements not in the specified *group-ID* will output zeroes for all their values.

Restrictions:

none

Related commands:

compute command fix ave/time command

Default:

The option defaults are ave = one.

1.14.59 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 or array calculated by a compute with ID
  - `c_ID[I]` = Ith component of global vector or Ith column of global array calculated by a compute with ID, I can include wildcard (see below)
  - `f_ID` = global scalar or vector or array calculated by a fix with ID

  - f_ID[I] = Ith component of global vector or Ith column of global array calculated by a fix with ID, I can include wildcard (see below)
  - 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 = start averaging on this timestep
  - off arg = M = do not average this value

    M = value: # from 1 to Nvalues
  - file arg = filename = name of file to output time averages to
  - title1 arg = string = text to print as 1st line of output file
  - title2 arg = string = text to print as 2nd line of output file
  - title3 arg = string = text to print as 3rd line of output file, only for vector mode

Examples:

```
fix 1 ave/time all 100 5 1000 c_myTemp c_thermo_temp file temp.profile
fix 1 ave/time all 100 5 1000 c_myCount[2] c_myCount[3] ave window 20 &
                              title1 "My output values"
fix 1 ave/time all 100 5 1000 c_myCount[*] ave window 20
fix 1 ave/time all 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-grid or per-surf quantity. If you wish to time-average those quantities, see the *fix ave/grid* and *fix ave/surf* commands.

Computes that produce global quantities are those which do not have the word *particle* or *grid* or *surf* 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* cannot be used, since they produce per-particle 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* = scalar, then the input values must be scalars, or vectors with a bracketed term appended, indicating the Ith value of the vector is used.

If *mode* = vector, then the input values must be vectors, or arrays with a bracketed term appended, indicating the Ith column of the array is used. All vectors must be the same length, which is the length of the vector or number of rows in the array.

Note that for values from a compute or fix, the bracketed index I can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "n*" or "m*n". If N = the size of the vector (for *mode* = scalar) or the number of columns in the array (for *mode* = vector), then an asterisk with no numeric values means all indices from 1 to N. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from n to N (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual elements of the vector or columns of the array had been listed one by one. E.g. these 2 fix ave/time commands are equivalent, since the *compute count* command creates, in this case, a global vector with 3 values.

```
compute 1 count Ar He O
fix 1 ave/time 100 1 100 c_1 file tmp.count
fix 1 ave/time 100 1 100 c_1[1] c_1[2] c_1[3] file tmp.count
```

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. If a bracketed term is appended, the Ith column of the global array calculated by the compute is used. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

Note that there is a *compute reduce* command which can sum per-particle or per-grid or per-surf quantities into a global scalar or vector which can thus be accessed by fix ave/time. Also Note that users can also write code for their own compute styles and *add them to SPARTA*; their output can then be processed by this fix.

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. If a bracketed term is appended, the Ith column of the global array calculated by the fix is used. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

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 *stats_style* 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, output info:

No information about this fix is written to *binary restart files*.

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 command, *fix ave/surf command*, *variable command*

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.

1.14.60 fix balance command - fix balance/kk command

Syntax:

```
fix ID balance Nfreq thresh bstyle args
```

- ID is documented in *fix* command
- balance = style name of this fix command
- Nfreq = perform dynamic load balancing every this many steps
- thresh = rebalance if imbalance factor is above this threshhold
- bstyle = *random* or *proc* or *rcb*

```
random args = none
proc args = none
rcb args = weight
  weight = cell or part or time
```

- zero or more keyword/value(s) pairs may be appended
- keyword = *axes* or *flip*

```
axes value = dims
  dims = string with any of "x", "y", or "z" characters in it
flip value = yes or no
```

Examples:

```
fix 1 balance 1000 1.1 rcb cell
fix 2 balance 10000 1.0 random
```

Description:

This command dynamically adjusts the assignment of grid cells and their particles to processors as a simulation runs, to attempt to balance the computational cost (load) evenly across processors. The load balancing is "dynamic" in the sense that rebalancing is performed periodically during the simulation. To perform "static" balancing, before or between runs, see the *balance_grid* command.

This command is useful to use during simulations where the spatial distribution of particles varies with time, leading to load imbalance.

After grid cells have been assigned, they are migrated to new owning processors, along with any particles they own or other per-cell attributes stored by fixes. The internal data structures within SPARTA for grid cells and particles are re-initialized with the new decomposition.

The details of how child cells are assigned to processors by the various options of this command are described below. The cells assigned to each processor will either be "clumped" or "dispersed".

The *rcb* keyword will produce clumped assignments of child cells to each processor. This means each processor's cells will be geometrically compact. The *random* and *proc* keywords will produce dispersed assignments of child cells to each processor.

IMPORTANT NOTE: See *Section 6.8* of the manual for an explanation of clumped and dispersed grid cell assignments and their relative performance trade-offs.

Rebalancing is attempted by this command once every *Nfreq* timesteps, but only if the current imbalance factor exceeds the specified *thresh*. This factor is defined as the maximum number of particles owned by any processor, divided by the average number of particles per processor. Thus an imbalance factor of 1.0 is perfect balance. For 10000 particles running on 10 processors, if the most heavily loaded processor has 1200 particles, then the factor is 1.2, meaning there is a 20% imbalance. The *thresh* setting must be >= 1.0.

IMPORTANT NOTE: This command attempts to minimize the imbalance factor, as defined above. But computational cost is not strictly proportional to particle count, depending on the *collision* and *chemistry* models being used. Also, changing the assignment of grid cells and particles to processors may lead to additional communication overheads, e.g. when migrating particles between processors. Thus you should benchmark the run times of your simulation to judge how often balancing should be performed, and how aggressively to set the *thresh* value.

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

The *proc* keyword means that each processor will choose a random processor to assign its first grid cell to. It will then loop over its grid cells and assign each to consecutive processors, wrapping around the collection of processors if necessary. In this case every processor will typically not be assigned exactly the same number of grid cells.

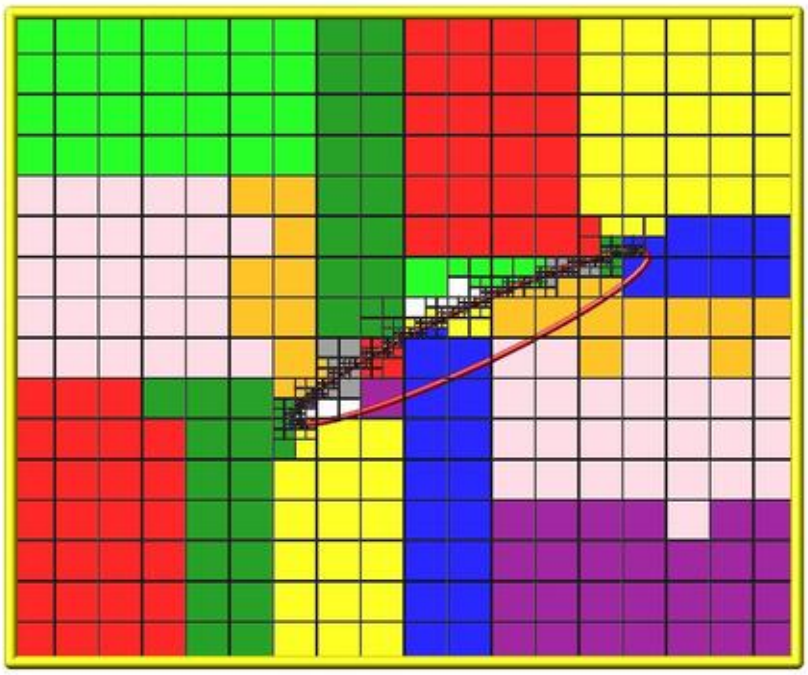
The *rcb* keyword uses a recurvise coordinate bisectioning (RCB) algorithm to assign spatially-compact clumps of grid cells to processors. Each grid cell has a "weight" in this algorithm so that each processor is assigned an equal total weight of grid cells, as nearly as possible.

If the *weight* argument is specified as *cell*, then the weight for each grid cell is 1.0, so that each processor will end up with an equal number of grid cells.

If the *weight* argument is specified as *part*, than the weight for each grid cell is the number of particles it currently owns, so that each processor will end up with an equal number of particles.

If the *weight* argument is specified as *time*, then timers are used to estimate the cost of each grid cell. The cost from the timers is given on a per processor basis, and then assigned to grid cells by weighting by the relative number of particles in the grid cells. If no timing data has yet been collected at the point in a script where this command is issued, a *cell* style weight will be used instead of *time*. A small warmup run (for example 100 timesteps) can be used before the balance command so that timer data is available. The number of timesteps *Nfreq* between balancing steps also needs to be large enough to give reliable timings. The timers used for balancing tally time from the move, sort, collide, and modify portions of each timestep.

Here is an example of an RCB partitioning for 24 processors, of a 2d hierarchical grid with 5 levels, refined around a tilted ellipsoidal surface object (outlined in pink). This is for a *weight cell* setting, yielding an equal number of grid cells per processor. Each processor is assigned a different color of grid cells. (Note that less colors than processors were used, so the disjoint yellow cells actually belong to three different processors). This is an example of a clumped distribution where each processor's assigned cells can be compactly bounded by a rectangle. Click for a larger version of the image.



The optional keywords *axes* and *flip* only apply to the *rcb* style. Otherwise they are ignored.

The *axes* keyword allows limiting the partitioning created by the RCB algorithm to a subset of dimensions. The default is to allow cuts in all dimension, e.g. x,y,z for 3d simulations. The dims value is a string with 1, 2, or 3 characters.

The characters must be one of “x”, “y”, or “z”. They can be in any order and must be unique. For example, in 3d, a dims = xz would only partition the 3d grid only in the x and z dimensions.

The *flip* keyword is useful for debugging. If it is set to *yes* then each time an RCB partitioning is done, the coordinates of grid cells will (internally only) undergo a sign flip to insure that the new owner of each grid cell is a different processor than the previous owner, at least when more than a few processors are used. This will insure all particle and grid data moves to new processors, fully exercising the rebalancing code.

Restart, output info:

No information about this fix is written to *binary restart files*.

This fix computes a global scalar which is the imbalance factor after the most recent rebalance and a global vector of length 2 with additional information about the most recent rebalancing. The 2 values in the vector are as follows:

- 1 = max # of particles per processor
- 2 = imbalance factor before the last rebalance was performed

As explained above, the imbalance factor is the ratio of the maximum number of particles on any processor to the average number of particles per processor. For the *rcb* style’s *time* option, the imbalance factor after the most recent rebalance cannot be computed and 0.0 is returned for the global scalar value.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

none

Related commands:

create_grid command, *balance_grid command*

Default:

none

1.14.61 fix emit/face command

1.14.62 fix emit/face/kk command

Syntax:

```
fix ID emit/face mix-ID face1 face2 ... keyword value(s) ...
```

- ID is documented in *fix* command
- emit/face = 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(s) pairs may be appended
- keyword = `n` or `nevery` or `perspecies` or `region` or `subsonic` or `twopass`

  n value = Np Np = number of particles to create

  nevery value = Nstep Add particles every this many timesteps

  perspecies value value = yes or no

  region value = region-ID ID of the region

  subsonic values = Psub Tsub

  - Psub = pressure setting at inflow boundary (pressure units)
  - Tsub = temperature setting at inflow boundary, can be NULL (temperature units)

  twopass values = none none is the only possible value

Examples:

```
fix in emit/face air all
fix in emit/face mymix xlo yhi n 1000 nevery 10 region circle
fix in emit/face air xlo subsonic 0.1 300
fix in emit/face air xhi subsonic 0.05 NULL twopass
```

Description:

Emit particles from one or more faces of the simulation box, continuously during a simulation. If invoked every timestep, this fix creates a continuous influx of particles thru the face(s).

The properties of the added particles are determined by the mixture with ID `mix-ID`. This sets the number and species of added particles, as well as their streaming velocity, thermal temperature, and internal energy modes. The details are explained below.

One or more faces of the simulation box can be specified via the `face1`, `face2`, etc arguments. The 6 possible faces 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 grid cell with one or more of its faces touching a specified boundary `face` performs the following computations to add particles. The particles are added at the beginning of the SPARTA timestep.

The molecular flux across a grid cell face per unit time is given by equation 4.22 of [Bird94]. The number of particles *M* to insert on a particular grid cell face is based on this flux and additional global, flow, and cell face properties:

- global property: `fnum` ratio as specified by the *global* command
- flow properties: number density, streaming velocity, and thermal temperature
- cell face properties: area of face and its orientation relative to the streaming velocity

The flow properties are defined for the specified mixture via the *mixture* command.

If *M* has a fractional value, e.g. 12.5, then 12 particles are added, and a 13th depending on the value of a random number. Each particle is added at a random location on the grid cell face. The particle species is chosen randomly in accord with the *frac* settings of the collection of species in the mixture, as set by the *mixture command* command.

Important: The preceeding calculation is actually done using face areas associated with *weighted* cell volumes. Grid cells can be weighted using the *global weight* command.

The velocity of the particle is set to the sum of the streaming velocity and a thermal velocity sampled from the thermal temperature. The internal energy modes of the particle are determined by the `trot` and `tvib` settings of the mixture and the `rotate` and `vibrate` options of the *collide_modify* command. Note that if the *collide* command has not been specified (free molecular flow), then no rotational or vibrational energy will be assigned to created particles.

If the final particle velocity is not directed "into" the grid cell, then the velocity sampling procedure is repeated until it is. This insures that all added particles enter the simulation domain, as desired.

The first timestep that added particles are advected, they move for a random fraction of the timestep. This insures a continuous flow field of particles entering the simulation box.

The `n` keyword can alter how many particles are added, which can be useful for debugging purposes. If `Np` is set to 0, then the number of added particles is a function of `fnum`, `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 exactly `Np` particles are added on each insertion timestep. This is done by dividing `Np` by the total number of grid cells that are adjacent to the specified box faces and adding an equal number of particles per grid cell.

The `nevery` keyword determines how often particles are added. If `Nstep` > 1, this may give a non-continuous, clumpy distribution in the inlet flow field.

The `perspecies` keyword determines how the species of each added particle is randomly determined. This has an effect on the statistical properties of added particles.

If `perspecies` is set to `yes`, then a target insertion number `M` in a grid cell is calculated for each species, 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 added, and a 13th depending on the value of a random number.

If `perspecies` is set to `no`, then a single target insertion number `M` in a grid cell is calculated for all the species. Each time a particle is added, 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 added, and a 13th depending on the value of a random number.

Here is a simple example that illustrates 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 adds 10 particles from that mixture. If `perspecies` is set to `yes`, then exactly 5 particles of each species will be added on every timestep insertions take place. If `perspecies` is set to `no`, then exactly 10 particles will be added every time and on average there 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.

If the `region` keyword is used, then a particle will only added if its position is within the specified `region-ID`. This can be used to only allow particle insertion on a subset of the boundary face. Note that the `side` option for the *region* command can be used to define whether the inside or outside of the geometric region is considered to be "in" the region.

Important: If the `region` and `n` keywords are used together, less than N particles may be added on an insertion timestep. This is because grid cells will be candidates for particle insertion, unless they are entirely outside the bounding box that encloses the region. Particles those grid cells attempt to add are included in the count for N, even if some or all of the particle insertions are rejected due to not being inside the region.

The `subsonic` keyword uses the method of Fang and Liou [Fang02] to determine the number of particles to insert in each grid cell on the emitting face(s). They used the method of characteristics to calculate the mean properties of the incoming molecular flux, so that the prescribed pressure condition is achieved. These properties are then applied to calculate the molecular flux across a grid cell face per unit time, as given by equation 4.22 of [Bird94].

This keyword allows specification of both the pressure and temperature at the boundary or just the pressure (by specifying the temperature as NULL). If specified, the temperature must be > 0.0. Currently, instantaneous values for the density, temperature, and stream velocity of particles in the cells adjacent to the boundary face(s) are computed and used to determine the properties of inserted particles on each timestep.

Important: Caution must be exercised when using the subsonic boundary condition without specifying an inlet temperature. In this case the code tries to estimate the temperature of the flow from the properties of the particles in the domain. If the domain contains few particles per cell it may lead to spurious results. This boundary condition is meant more for an outlet than an inlet boundary condition, and performs well in cases where the cells are adequately populated.

Important: When using this keyword, you should also use an appropriate boundary collision or chemistry model via the *boundary* or *bound_modify* or *surf_collide* or *surf_react* commands, so that particles hitting the surface disappear as if they were exiting the simulation domain. That is necessary to produce the correct subsonic conditions that the particle insertions due to this command are trying to achieve.

The `twopass` keyword does not require a value. If used, the insertion procedure will loop over the insertion grid cells twice, the same as the KOKKOS package version of this fix does, so that it can reallocate memory efficiently, e.g. on a GPU. If this keyword is used the non-KOKKOS and KOKKOS version will generate exactly the same set of particles, which makes debugging easier. If the keyword is not used, the non-KOKKOS and KOKKOS runs will use random numbers differently and thus generate different particles, though they will be statistically similar.

Restart, output info:

No information about this fix is written to *binary restart files*.

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 added on the most recent insertion step. The second element is the cummulative total number added since the beginning of the run. The 2nd value is initialized to zero each time a run is performed.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the

manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

Particles cannot be emitted from periodic faces of the simulation box. Particles cannot be emitted from `z` faces of the simluation box for a 2d simulation.

A `n` setting of `Np` > 0 can only be used with a `perspecies` setting of `no`.

A warning will be issued if a specified face has an inward normal in a direction opposing the streaming velocity. Particles will still be emitted from that face, so long as a small fraction have a thermal velocity large enough to overcome the outward streaming velocity, so that their net velocity is inward. The threshold for this is that a thermal velocity 3 sigmas from the mean thermal velocity is large enough to overcome the outward streaming velocity and produce a net velocity into the simulation box.

Related commands:

mixture, *create_particles*, *fix emit/face/file*

Default:

The keyword defaults are n = 0, nevery = 1, perspecies = yes, region = none, no subsonic settings, no twopass setting.

1.14.63 fix emit/face/file command

Syntax:

```
fix ID emit/face/file mix-ID face filename boundary-ID keyword value ...
```

- ID is documented in *fix* command
- emit/face/file = style name of this fix command
- mix-ID = ID of mixture to use when creating particles
- face = *xlo* or *xhi* or *ylo* or *yhi* or *zlo* or *zhi*
- filename = input data file with boundary values for the emission
- boundary-ID = section of data file to read
- zero or more keyword/value pairs may be appended

- keyword = `frac` or `nevery` or `perspecies` or `region`
  - frac value = fraction = 0.0 to 1.0 fraction of particles to insert
  - nevery value = Nstep = insert every this many timesteps
  - perspecies value = yes or no
  - region value = region-ID

Examples:

```
fix in emit/face/file air xlo input.data xlo
fix in emit/face/file mymix ylo file.txt oneface frac 0.1 nevery 10
```

Description:

Emit particles from a face of the simulation box, continuously during a simulation. The particles are added using properties of the specified mixture and values read from an input file that can override those properties. The input file can thus be used to create an influx of particles that varies spatially over the surface of the *face*. This can be useful, for example, to model an object inserted into a plume flow where the flow has spatially varying properties. If invoked every timestep, this fix creates a continuous influx of particles thru the face.

The properties of the added particles are determined by the mixture with ID *mix-ID* and the input file. Together they set the number and species of added particles, as well as their streaming velocity, thermal temperature, and internal energy modes. Settings for a subsonic pressure boundary condition is also allowed. The details are explained below.

Only one face of the simulation box can be specified via the *face* argument. The 6 possible faces are *xlo*, *xhi*, *ylo*, *yhi*, *zlo*, or *zhi*. This command can be used multiple times to add particles on multiple faces.

On each insertion timestep, each grid cell with a face touching the specified boundary *face* performs the following computations to add particles. The particles are added at the beginning of the SPARTA timestep.

The molecular flux across a grid cell face per unit time is given by equation 4.22 of [Bird94]. The number of particles *M* to add on a particular grid cell face is based on this flux and additional global, flow, and cell face properties:

- global property: *fnum* ratio as specified by the *global* command
- flow properties: number density, streaming velocity, and thermal temperature
- cell face properties: area of face and its orientation relative to the streaming velocity

The flow properties are defined for the specified mixture via the *mixture* command. Any or all them can be overridden by values in the input data file, which affect individual grid cells as described below.

If *M* has a fractional value, e.g. 12.5, then 12 particles are added, and a 13th depending on the value of a random number. Each particle is added at a random location on the grid cell face. The particle species is chosen randomly in accord with the *frac* settings of the collection of species in the mixture, as set by the *mixture* command. These can also be overridden by spatially varying number fraction values in the input data file, as described below.

The velocity of the particle is set to the sum of the streaming velocity and a thermal velocity sampled from the thermal temperature. The internal energy modes of the particle are determined by the *trot* and *tvib* settings and the *rotate* and *vibrate* options of the *collide_modify* command. Note that if the *collide* command has not been specified (free molecular flow), then no rotational or vibrational energy will be assigned to created particles.

If the final particle velocity is not directed "into" the grid cell, then the velocity sampling procedure is repeated until it is. This insures that all added particles enter the simulation domain, as desired.

The first timestep that added particles are advected, they move for a random fraction of the timestep. This insures a continuous flow field of particles entering the simulation box.

For 3d simulations, the input data file defines a 2d mesh of data points which conceptually overlays some portion or all of the specified face of the simulation box. For a 2d simulation, a 1d mesh is defined. The mesh is topologically regular, but can have uniform or non-uniform spacing in each of its two or one dimensions (for 3d or 2d problems). One or more values can be defined at every mesh point, which override any of the mixture settings defined by the *mixture* command. These are the flow properties discussed above (number density, streaming velocity, and thermal temperature), as well as the number fraction of any species in the mixture. Any value not defined in the input data file defaults to the mixture value.

For 3d simulations, a 2d mesh is defined in the file using I,J indices. (The 1d mesh for 2d simulations is described below). I and J map to any of the simulation box faces in this manner. A simulation box face has two varying dimensions (e.g. ylo face = x and z dimensions). The I index in the file corresponds to the "lowest" of these dimensions, where x < y < z. The J index in the file corresponds to the higher. Thus for face ylo, I = x and J = z. A low I or J value corresponds to a low x or z value, regardless of whether the mapping is to the ylo or yhi face. A 1d mesh for a 2d simulation is defined in an analogous manner, e.g. for face xlo, I = y.

For a 3d simulation, interpolation from values on the 2d mesh to any grid cell face that is on the corresponding simulation box face is done in the following manner. There are 3 cases to consider.

a. For a grid cell face that is entirely inside the area defined by the file mesh, the centroid (center point) of the grid cell face is surrounded geometrically by 4 file mesh points. The 4 values defined on those 4 file points are averaged in a weighted manner using bilinear interpolation (described below) to determine the value for the grid cell face. This value is then used for the calculation described above for M = the number of particles to add on the cell face as well as the properties of the added particles.

b. For a grid cell face that is entirely outside the area defined by the file mesh, no particles are added in that grid cell.

c. For a grid cell face that partially overlaps the area defined by the file mesh, the extent of the overlap is computed. The centroid (center point) of the overlap area is surrounded geometrically by 4 file mesh points. The values for those 4 points are used as in (a) above to determine properties of particles added in that grid cell. Note that the area of insertion, used to calculate M, is the overlap area, which is smaller than the grid cell face area. Also, particles are only added within the overlap area of the grid cell face.

For a 2d simulation, the 3 cases are similar, except for (a) and (c) the centroid is the midpoint of a line segment, the centroid is surrounded by 2 file mesh points, and linear interpolation (described below) is performed to determine the value for the grid face.

The format of the input data file is a series of one or more sections, defined as follows (without the parenthesized comments). Note that one file can contain many sections, each with a different set of tabulated values. The sections can be a mix of 2d and 3d formats. SPARTA reads the file section by section, skipping sections with non-matching boundary IDs, until it finds one that matches the specified boundary-ID. The lines that follow must be in this order:

```
# plume ABC info           (one or more comment or blank lines)

PLUME_ABC                  (boundary-ID is first word on line)
NIJ 4 10                   (mesh size: Ni by Nj)
NV 3                       (Nv = number of values per mesh point)
VALUES nrho temp Ar        (list of Nv values per mesh point)
IMESH 0.0 0.3 0.9 1.0      (mesh coordinates in I direction)
JMESH ...                  (mesh coordinates in J direction)
                           (blank)
1 1 1.0 300.0 0.5          (I, J, value1, value2, ...)
1 2 1.02 310.0 0.5
...
4 10 3.0 400.0 0.7
```

This format is for a 3d simulation. For a 2d simulation, there are 3 changes:

1. "NIJ 4 10" is replaced by "NI 6"
2. JMESH line is not included
3. "I,J,value1,..." is replaced by "I,value1,..."

A section begins with a non-blank line whose first character is not a "#". Blank lines or lines starting with "#" can be used as comments between sections. The first line begins with a boundary-ID which identifies the section. The line can contain additional text, but the initial text must match the boundary-ID specified in the fix emit/face/file command. Otherwise the section is skipped.

The VALUES line lists Nv keywords. The list of possible keywords is as follows, along with the meaning of the numeric value specified for the mesh point:

- nrho = number density
- vx,vy,vz = 3 components of streaming velocity
- temp = thermal temperature
- trot = rotational temperature
- tvib = vibrational temperature
- press = pressure for subsonic boundary condition
- species = number fraction of any species in the mixture

The IMESH and JMESH lines must list values that are monotonically increasing.

Following a blank line, the next N = Ni x Nj lines (or N = Ni lines for a 2d simulation) list the tabulated values. The format of each line is I,J followed by Nv values. The N lines can be in any order, but all unique I,J (or I for 2d) indices must be listed.

Note that if number fractions are specified for one or more species in the mixture, then they override number fraction values for the mixture itself, as set by the *mixture* command. However, for each grid cell, the rule that the number fraction of all species in the mixture must sum to 1.0 is enforced, just as it is for the mixture. This means that number fractions of species not specified in the file or in the mixture may be reset (for that grid cell) to insure the sum = 1.0, as explained on the *mixture* command doc page. If this cannot be done, an error will be generated.

If the *press* keyword is used, this means a subsonic pressure boundary condition is used for the face, similar to how the *subsonic* keyword is used for the *fix emit/face* command. If just the *press* keyword is specified, but not the *temp* keyword, then it is similar to the "subsonic press NULL" setting for the *fix emit/face* command. If both keywords are used it is similar to the "subsonic press temp" setting for the *fix emit/face* command. The difference with this command is that both the *press* and *temp* values can be vary spatially across the box face, like the other keyword values.

The subsonic pressure boundary condition is uses the method of Fang and Liou [Fang02] to determine the number of particles to insert in each grid cell on the emitting face(s). They used the method of characteristics to calculate the mean properties of the incoming molecular flux, so that the prescribed pressure condition is achieved. These properties are then applied to calculate the molecular flux across a grid cell face per unit time, as given by equation 4.22 of [Bird94].

As explained above the input data file can specify both the pressure and temperature at the boundary or just the pressure. If specified, the temperature must be > 0.0. Currently, instantaneous values for the density, temperature, and stream velocity of particles in the cells adjacent to the boundary face(s) are computed and used to determine the properties of inserted particles on each timestep.

Important: Caution must be exercised when using the subsonic boundary condition without specifying an inlet temperature. In this case the code tries to estimate the temperature of the flow from the properties of the particles in the domain. If the domain contains few particles per cell it may lead to spurious results. This boundary condition is

meant more for an outlet than an inlet boundary condition, and performs well in cases where the cells are adequately populated.

Important: When using a subsonic pressure boundary condition, you should also use an appropriate boundary collision or chemistry model via the *boundary* or *bound_modify* or *surf_collide* or *surf_react* commands, so that particles hitting the surface disappear as if they were exiting the simulation domain. That is necessary to produce the correct subsonic conditions that the particle insertions due to this command are trying to achieve.

For 3d simulations, bilinear interpolation from the 2d mesh of values specified in the file is performed using this equation to calculate the value at the centroid point (i,j) in the grid cell face:

$$f(i,j) = (1/area)(f(i1,j1)(i2-i)(j2-j) + f(i2,j1)(i-i1)(j2-j) + \\ f(i2,j2)(i-i1)(j-j1) + f(i1,j2)(i2-i)(j-j1))$$

where the 4 surrounding file mesh points are (i1,j1), (i2,j1), (i2,j2), and (i1,j2). The 4 f() values on the right-hand side are the values defined at the file mesh points. The sum is normalized by the area of the overlap between the grid cell face and file mesh.

For 2d simulations, linear interpolation from the 1d mesh of values specified in the file is performed using this equation to calculate the value at the centroid poitn (i) in the grid cell line:

$$f(i) = (1/length)(f(i1)(i2-i) + f(i2)(i-i1) \\ = f(i1) + (i-i1)/(i2-i1)(f(i2) - f(i1))$$

where the 2 surrounding file mesh points are (i1) and (i2). The 2 f() values on the right-hand side are the values defined at the file mesh points. The sum is normalized by the length of the overlap between the grid cell line and file mesh.

The *frac* keyword can alter how many particles are added, which can be useful for debugging purposes. If *frac* is set to 1.0 (the default) then the number of particles added is the sum of the *M* values computed for each grid cell that overlaps with the mesh defined in the file, as described above. If *frac* < 1.0 then *M* is scaled by frac to determine the number of particles added in each grid cell. Thus a simulation with less particles can easily be run to test if it is setup correctly.

The *nevery* keyword determines how often particles are added. If *Nstep* > 1, this may give a non-continuous, clumpy distribution in the inlet flow field.

The *perspecies* keyword determines how the species of each added particle is randomly determined. This has an effect on the statistical properties of added particles.

If *perspecies* is set to *yes*, then a target insertion number *M* in a grid cell is calculated for each species, 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 added, and a 13th depending on the value of a random number.

If *perspecies* is set to *no*, then a single target insertion number *M* in a grid cell is calculated for all the species. Each time a particle is added, 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 added, and a 13th depending on the value of a random number.

Here is a simple example that illustrates 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 adds 10 particles from that mixture. If *perspecies* is set to *yes*, then exactly 5 particles of each species will be added on every timestep insertions take place. If *perspecies* is set to *no*, then exactly 10 particles will be added every time and on average there 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.

If the *region* keyword is used, then a particle will only added if its position is within the specified *region-ID*. This can be used to only allow particle insertion on a subset of the boundary face. Note that the *side* option for the *region* command can be used to define whether the inside or outside of the geometric region is considered to be "in" the region.

Restart, output info:

No information about this fix is written to *binary restart files*.

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 added on the most recent insertion step. The second element is the cummulative total number added since the beginning of the run. The 2nd value is initialized to zero each time a run is performed.

Restrictions:

Particles cannot be added on periodic faces of the simulation box. Particles cannot be added on z faces of the simluation box for a 2d simulation.

Unlike the *fix emit/face command*, no warning is issued if the specified emission face has an inward normal in a direction opposing the streaming velocity, as defined by the mixture. This is because the streaming velocity as defined by the specified mixture may be overridden by values in the file.

For that grid cell, particles will still be emitted from that face, so long as a small fraction have a thermal velocity large enough to overcome the outward streaming velocity, so that their net velocity is inward. The threshold for this is the thermal velocity for particles 3*sigma from the mean thermal velocity.

Related commands:

mixture command, *create_particles command*, *fix emit/face command*

Default:

The keyword defaults are frac = 1.0, nevery = 1, perspecies = yes, region = none.

1.14.64 fix emit/surf command

Syntax:

```
fix ID emit/surf mix-ID group-ID keyword value ...
```

- ID is documented in *fix* command
- emit/surf = style name of this fix command
- mix-ID = ID of mixture to use when creating particles
- group-ID = ID of surface group that emits particles

- zero or more keyword/value pairs may be appended
- keyword = *n* or *normal* or *nevery* or *perspecies* or *region* or *subsonic*

```
n value = Np = number of particles to create
normal value = yes or no = emit normal to surface elements or with streaming␣
↪velocity
nevery value = Nstep = add particles every this many timesteps
perspecies value = yes or no
region value = region-ID
subsonic values = Psub Tsub
  Psub = pressure setting at inflow boundary (pressure units)
  Tsub = temperature setting at inflow boundary, can be NULL (temperature units)
```

Examples:

```
fix in emit/surf air all
fix in emit/face mymix myPatch region circle normal yes
fix in emit/surf air all subsonic 0.1 300
fix in emit/surf air all subsonic 0.05 NULL
```

Description:

Emit particles from a group of surface elements, continuously during a simulation. If invoked every timestep, this fix creates a continuous outflux of particles from the surface elements in the group.

The properties of the added particles are determined by the mixture with ID *mix-ID*. This sets the number and species of added particles, as well as their streaming velocity, thermal temperature, and internal energy modes. The details are explained below.

Which surface elements emit particles is specified by the *group-ID* for a surface group, which defines a set of surface elements. The *group surf* is used to define surface groups.

On each insertion timestep, each grid cell that overlaps with one or more emitting surface elements performs the following computations to add particles for each grid cell/surface element pairing. The particles are added at the beginning of the SPARTA timestep.

The molecular flux emitted from a surface element per unit time is given by equation 4.22 of [Bird94]. The number of particles *M* to insert on the portion of a surface element that is contained within a grid cell is based on this flux and additional global, flow, and surface element properties:

- global property: *fnum* ratio as specified by the *global* command
- flow properties: number density, streaming velocity, and thermal temperature
- surface element properties: portion of surface element area that overlaps with the grid cell and its orientation relative to the streaming velocity

The flow properties are defined for the specified mixture via the *mixture* command.

If *M* has a fractional value, e.g. 12.5, then 12 particles are added, and a 13th depending on the value of a random number. Each particle is added at a random location within the portion of the surface element that overlaps with the grid cell. The particle species is chosen randomly in accord with the *frac* settings of the collection of species in the mixture, as set by the *mixture* command.

IMPORTANT NOTE: The preceeding calculation is actually done using surface element areas associated with *weighted* cell volumes. Grid cells can be weighted using the *global weight* command.

The velocity of the particle is set to the sum of the streaming velocity and a thermal velocity sampled from the thermal temperature. The internal energy modes of the particle are determined by the *trot* and *tvib* settings of the mixture and the *rotate* and *vibrate* options of the *collide_modify* command. Note that if the *collide* command has not been specified (free molecular flow), then no rotational or vibrational energy will be assigned to created particles. See the discussion of the *normal* keyword below for a way to change the velocity assignment to be oriented in the direction normal to the surface element, rather than in the direction of the streaming velocity.

If the final particle velocity is not directed "out of" the surface element, then the velocity sampling procedure is repeated until it is. This insures that all added particles emit from the surface element, as desired.

The first timestep that added particles are advected, they move for a random fraction of the timestep. This insures a continuous flow field of particles emitting from each surface element.

The *n* keyword can alter how many particles are added, which can be useful for debugging purposes. If *Np* is set to 0, then the number of added particles is a function of *fnum*, *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 exactly *Np* particles are added on each insertion timestep. This is done by dividing *Np* by the total number of grid cell/surface element pairs and adding an equal number of particles per pair.

The *normal* keyword can be used to alter how velocities are set for added particles. If *normal* is set to *no*, then a particle's velocity is set as described above, using the mixture's streaming velocity superposed with a thermal velocity sampled from the temperature of the mixture. Note that the same streaming velocity is used for all emitting surface elements, regardless of their orientation with respect to the streaming velocity. If *normal* is set to *yes*, then each surface element is assigned its own "streaming" velocity in the following manner. The streaming velocity points in the direction of the outward normal of the surface element, and its magnitude is set to the magnitude of the mixture's streaming velocity. A velocity is then assigned to the particle in the same manner as before. It is assigned the outward streaming velocity superposed with a thermal velocity sampled fromt he temperature of the mixture. The effect is that particles effectively stream outward from each emitting surface element.

The *nevery* keyword determines how often particles are added. If *Nstep* > 1, this may give a non-continuous, clumpy distribution in the inlet flow field.

The *perspecies* keyword determines how the species of each added particle is randomly determined. This has an effect on the statistical properties of added particles.

If *perspecies* is set to *yes*, then a target insertion number *M* for a grid cell/surface element pair is calculated for each species, 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 added, and a 13th depending on the value of a random number.

If *perspecies* is set to *no*, then a single target insertion number *M* for a grid cell/surface element pair is calculated for all the species. Each time a particle is added, 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 added, and a 13th depending on the value of a random number.

Here is a simple example that illustrates 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/surface element pair adds 10 particles from that mixture. If *perspecies* is set to *yes*, then exactly 5 particles of each species will be added on every timestep insertions take place. If *perspecies* is set to *no*, then exactly 10 particles will be added every time and on average there 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.

If the *region* keyword is used, then a particle will only added if its position is within the specified *region-ID*. This can be used to only allow particle insertion on a subset of the collective area of the specified group of surface elements. Note that the *side* option for the *region* command can be used to define whether the inside or outside of the geometric region is considered to be "in" the region.

IMPORTANT NOTE: If the *region* and *n* keywords are used together, less than N particles may be added on an insertion timestep. This is because grid cell/suface element pairs will be candidates for particle insertion, unless the grid cell is entirely outside the bounding box that encloses the region. Particles those grid cell/surface element pairs will attempt to add are included in the count for N, even if some or all of the particle insertions are rejected due to not being inside the region.

The *subsonic* keyword uses the method of Fang and Liou [Fang02] to determine the number of particles to insert in each grid cell on the emitting face(s). They used the method of characteristics to calculate the mean properties of the incoming molecular flux, so that the prescribed pressure condition is achieved. These properties are then applied to calculate the molecular flux across a grid cell face per unit time, as given by equation 4.22 of [Bird94]

This keyword allows specification of both the pressure and temperature at the surface or just the pressure (by specifying the temperature as NULL). If specified, the temperature must be > 0.0. Currently, instantaneous values for the density, temperature, and stream velocity of particles in the cells containing the surface elements are computed and used to determine the properties of inserted particles on each timestep.

IMPORTANT NOTE: Caution must be exercised when using the subsonic boundary condition without specifying an inlet temperature. In this case the code tries to estimate the temperature of the flow from the properties of the particles in the domain. If the domain contains few particles per cell it may lead to spurious results. This boundary condition is meant more for an outlet than an inlet boundary condition, and performs well in cases where the cells are adequately populated.

IMPORTANT NOTE: When using this keyword, you should also use an appropriate surface collision or chemistry model via the *surf_collide* or *surf_react* commands, so that particles hitting the surface disappear as if they were exiting the simulation domain. That is necessary to produce the correct subsonic conditions that the particle insertions due to this command are trying to achieve.

Restart, output info:

No information about this fix is written to *binary restart files*.

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 added on the most recent insertion step. The second element is the cummulative total number added since the beginning of the run. The 2nd value is initialized to zero each time a run is performed.

Restrictions:

A *n* setting of $Np > 0$ can only be used with a *perspecies* setting of *no*.

If *normal* is set to *no*, which is the default, then unlike the *fix emit/face command*, no warning is issued if a surface element has an inward normal in a direction opposing the streaming velocity, as defined by the mixture.

For that surface element, particles will still be emitted, so long as a small fraction have a thermal velocity large enough to overcome the outward streaming velocity, so that their net velocity is inward. The threshold for this is the thermal velocity for particles 3*sigma from the mean thermal velocity.

Related commands:

mixture, *create_particles*, *fix emit/face*

Default:

The keyword defaults are n = 0, normal = no, nevery = 1, perspecies = yes, region = none, no subsonic settings.

1.14.65 fix grid/check command

1.14.66 fix grid/check/kk command

Syntax:

```
fix ID grid/check N outflag keyword arg ...
```

- ID is documented in *fix* command
- grid/check = style name of this fix command
- N = check every N timesteps
- outflag = *error* or *warn* or *silent*
- zero or more keyword/args pairs may be appended
- keyword = *outside*

```
outside arg = yes or no
```

Examples:

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

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 timestepping algorithm.

The check is performed once 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 outside its assigned grid cell.

If the *outside* keyword is set to *yes*, then a check for particles inside implicit surfaces is also performed. If a particle is in a grid cell with implicit surface elements and the particles is "inside" the surfaces (which have possibly ablated), then the error count is incremented.

If the outflag setting is *error*, SPARTA will print an error and stop if it finds a particle in an incorrect grid cell or inside the implicit surface elements. For *warn*, it will print a warning message and continue. For *silent*, it will print no message, but the count of such occurrences can be monitored as described below, e.g. by outputting the value with the *stats* command.

Restart, output info:

No information about this fix is written to *binary restart files*.

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.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

none

Related commands:

none

Default:

The option default is outside = no.

1.14.67 fix move/surf command

1.14.68 fix move/surf/kk command

Syntax:

```
fix ID move/surf groupID Nevery Nlarge args ...
```

- ID is documented in *fix* command
- move/surf = style name of this fix command
- group-ID = group ID for which surface elements to move
- Nevery = move surfaces incrementally every this many steps
- Nlarge = move surfaces the entire distance after this many timesteps
- args = all remaining args are identical to those defined for the *move_surf* command starting with its "style" argument

Examples:

```
fix 1 move/surf all 100 1000 trans 1 0 0
fix 1 move/surf partial 100 10000 rotate 360 0 0 1 5 5 0 connect yes
fix 1 move/surf object2 100 50000 rotate 360 0 0 1 5 5 0
```

Description:

This command performs on-the-fly movement of all the surface elements in the specfied group via one of several styles. See the *group surf* command for info on how surface elements can be assigned to surface groups. Surface element moves can also be performed before or between simulations by using the *move_surf* command.

Moving surfaces during a simulation run can be useful if you want to to track transient changes in a flow while some attribute of the surface elements change, e.g. the separation between two spheres.

All of the command arguments which appear after *Nlarge*, which determine how surface elements move, are exactly the same as for the *move_surf* command, starting with its *style* argument. This includes optional keywords it defines. See its doc page for details.

Nevery specifies how often surface elements are moved incrementally along the path towards their final position. The current timestep must be a multiple of *Nevery*.

Nlarge must be a multiple of *Nevery* and specifies how long it will take the surface elements to move to their final position.

Thus if *Nlarge* = 100*Nevery*, each surface elements will move 1/100 of its total distance every *Nevery* steps.

The same rules that the *move_surf* command follows for particle deletion after surface elements move, are followed by this command as well. The criteria are applied after every incremental move. This is to prevent particles from ending up inside surface objects.

Likewise, the *connect* option of the *move_surf* command should be used in the same manner by this command if you need to insure that moving only some elements of an object do not result in a non-watertight surface grid.

Restart, output info:

No information about this fix is written to *binary restart files*. No global or per-particle or per-grid quantities are stored by this fix for access by various output commands.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

An error will be generated if any surface element vertex is moved outside the simulation box.

Related commands:

read_surf command, *move_surf command*, *remove_surf command*

Default:

none

1.14.69 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 quotes if it is more than one word. If it contains variables it must be enclosed in 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, output info:

No information about this fix is written to *binary restart files*. No global or per-particle or per-grid quantities are stored by this fix for access by various output commands.

Restrictions:

none

Related commands:

variable command, *print command*

Default:

The option defaults are no file output, screen = yes, and title string as described above.

1.14.70 fix vibmode command

Syntax:

```
fix ID vibmode
```

- ID is documented in *fix* command
- vibmode = style name of this fix command

Examples:

```
fix 1 vibmode
```

Description:

Enable multiple vibrational energy levels, defined on a per-species basis, to be used in a simulation. This fix is meant to be used with the *collide_modify vibrate discrete* setting which means that the vibrational energy of each (non-monoatomic) particle is discretized across one or more energy modes, each with its own characteristic vibrational temperature. This fix allocates per-particle storage for the mode indices and also has code to populate the multiple levels appropriately when particles are created. Collisions between pairs of particles will then transfer energy between the different modes of the two particles.

An overview of how to run simulations with multiple vibrational energy modes is given in the *Section 4.12*. This includes use of the *species* command with its *vibfile* option, and the use of the *collide_modify vibrate discrete* command. The section also lists all the commands that can be used in an input script to invoke various options associated with the vibrational energy modes. All of them depend on this fix vibmode command being defined.

Internally, this fix defines a custom particle attribute named "vibmode". It is an integer array with N values per particle. N is the maximum number of energy modes for any species defined in the simulation. The number of energy modes is half the vibrational degrees of freedom defined for each species. See the "species" command for how the degrees of freedom and associated vibrational temperatures and other properties are defined for each mode for each species.

Each of the N values is an integer count for the

Restart, output info:

No information about this fix is written to *binary restart files*.

However, the values of the custom particle attribute defined by this fix are written to the restart file. Namely the integer values stored in "vibmode" for each particle. As explained on the *read_restart* doc page these values can be re-assigned to particles when a restart file is read, if a new fix vibmode command is specified in the restart script before the first *run* command is used.

No global or per-particle or per-grid quantities are stored by this fix for access by various output commands.

However, the custom particle attributes defined by this fix can be accessed by the *dump particle* command, as p_vibmode. That means those per-particle values can be written to particle dump files.

Restrictions:

This fix is required if "collide_modify vibrate discrete" is used and there is one or more species defined which haave multiple vibrational energy modes (2 or more). In this scenario, if it is not defined, an error will occur when a "create_particles" or *run* command is issued. Conversely, if no species has multiple vibrational modes, this fix cannot be used.

Defining this fix after particles have been created will not populate the vibrational energy modes of particles that already exist. An exception is if the *read_restart* command is used to read in particles from a previous simulation where this fix was used. In that case, defining this fix after reading the restart file will enable the particles to keep their previous vibrational energy mode values.

Related commands:

collide modify vibrate discrete

Default:

none

1.14.71 global command

Syntax:

```
global keyword values ...
```

one or more keyword/value pairs

keyword = `fnum` or `nrho` or `vstream` or `temp` or `gravity` or `surfs` or `surfgrid` or `surfmax` or `cellmax` or `splitmax` or `surftally` or `surfpush` or `gridcut` or `comm/sort` or `comm/style` or `weight` or `particle/reorder` or `mem/limit`

- 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)

- gravity values = mag ex ey ez
  - mag = magnitude of acceleration due to gravity (acceleration units)
  - ex,ey,ez = direction vector that gravity acts in
- surfs value = explicit or explicit/distributed or implicit
  - explicit = surfs defined in read_surf file, each proc owns copy of all surfs
  - explicit/distributed = surfs defined in read_surf file, each proc owns only the surfs for its owned_ghost grid cells
  - implicit = surfs defined in read_isurf file, each proc owns only the surfs for its owned+ghost grid cells surfgrid value = percell or persurf or auto
  - percell = loop over my cells and check every surf
  - persurf = loop over my surfs and cells they overlap
  - auto = choose percell or persurf based on surface element and proc count
- surfmax value = Nsurf

  Nsurf = max # of surface elements allowed in single grid cell

- cellmax value = Ncell

  Ncell = max # of grid cells a single surf can overlap

- splitmax value = Nsplit

  Nsplit = max # of sub-cells one grid cell can be split into by surface elements

- surftally value = reduce or rvous or auto
  - reduce = tally surf collision info via MPI_Allreduce operations
  - rvous = tally via a rendezvous algorithm
  - auto = choose reduce or rvous based on surface element and proc count
- surfpush value(s) = no/yes or slo shi svalue
  - no = do not push surface element points near cell surface

  - yes = push surface element points near cell surface if necessary
  - slo,shi = push points within this range
  - svalue = push points to this value
- gridcut value = cutoff

  cutoff = acquire ghost cells up to this far away (distance units)
- comm/sort value = yes or no

  yes/no = sort incoming messages by proc ID if yes, else no sort
- comm/style value = neigh or all
  - neigh = setup particle comm with subset of near-neighbor processor
  - all = allow particle comm with potentially any processor
- weight value = wstyle mode
  - wstyle = cell
  - mode = none or volume or radius
- particle/reorder value = nsteps

  nsteps = reorder the particles every this many timesteps
- mem/limit value = grid or bytes
  - grid = limit extra memory for load-balancing, particle reordering, and restart file read/write to grid cell memory
  - bytes = limit extra particle memory to this amount (in MBytes)

Examples:

```
global fnum 1.0e20
global vstream 100.0 0 0 fnum 5.0e18
global temp 1000
global weight cell radius
global mem/limit 100
```

Description:

Define global properties of the system.

The *fnum* keyword sets the ratio of real, physical molecules to simulation particles. E.g. a value of 1.0e20 means that one particle in the simulation represents 1.0e20 molecules of the particle species.

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 since the z dimension is treated as having a length of 1.0.

Assuming your simulation is populated by particles from the background gas, the *fnum* and *nrho* settings can determine how many particles will be present in your simulation, when using the *create_particles* or *fix emit* command variants.

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.

The *gravity* keyword sets an acceleration term which is included in the motion of particles. The magnitude of gravity is set by the *mag* keyword. Its direction of action is set as (ex,ex,ez). The direction does not have to be a unit vector. If the magnitude is set to 0.0, no acceleration term is included, which is the default.

The *surfs* keyword determines what kind of surface elements SPARTA uses and how they are distributed across processors. Possible values are *explicit*, *explicit/distributed*, and *implicit*.

See the *Howto 6.13* section of the manual for an explantion of explicit versus implicit surfaces. The distributed option can be important for models with huge numbers of surface elements. Each processor stores copies of only the surfaces that overlap grid cells it owns or has ghost copies of. Implicit surfaces are always distributed.

The *explicit* setting is the default and means each processor stores a copy of all the defined surface elements. Note that a surface element requires about 100 bytes of storage, so storing a million on a single processor requires about 100 MBytes.

The *surfgrid* keyword determines what algorithm is used to enumerate the overlaps (intersections) between grid cells and surface elements (lines in 2d, triangles in 3d).

The possible settings are *percell*, *persurf*, and *auto*. The *auto* setting is the default and will choose between a *percell* or *persurf* algorithm based on the number of surface elements and processor count. If there are more processors than surface elements, the *percell* algorithm is used. Otherwise the *persurf* algorithm is used. The *percell* algorithm loops over the subset of grid cells each processor owns. All the surface elements are tested for overlap with each owned grid cell. The *persurf* algorithm loops over a 1/P fraction of surface elements on each processor. The bounding box around each surface is used to find all grid cells it possibly overlaps. For large numbers of surface elements or processors, the *persurf* algorithm is generally faster.

The *surfmax* keyword determines the maximum number of surface elements (lines in 2d, triangles in 3d) that can overlap a single grid cell. The default is 100, which should be large enough for any simulation, unless you define very coarse grid cells relative to the size of surface elements they contain.

The *cellmax* keyword determines the maximum number of grid cells that a single surface element (lines in 2d, tringles in 3d) can overlap. This keyword is only used if the *persurf* algorithm defined by the *surfgrid* keyword is invoked. The default is 100, which should be large enough for most simulations, unless you define one or more very large surface elements relative to the size of grid cells they intersect.

The *splitmax* keyword determines the maximum number of sub-cells a single grid cell can be split into as a result of its intersection with multiple surface elements (lines in 2d, triangles in 3d). The default is 10, which should be large enough for any simulation, unless you embed a complex-shaped surface object into one or a very few grid cells.

The *surftally* keyword determines what algorithm is used to combine tallies of surface collisions across processors that own portions of the same surface element. The possible settings are *reduce*, *rvous*, and *auto*. The *auto* setting is the default and will choose between a *reduce* or *rvous* algorithm based on the number of surface elements and processor count. If there are more processors than surface elements, the *reduce* algorithm is used. Otherwise the *rvous* algorithm is used. The *reduce* algorithm is suitable for relatively small surface elememt counts. It creates a copy of a vector or array of length the global number of surface elements. Each processor sums its tally contributions into the vector or array. An MPI_Allreduce() is performed to sum it across all processors. Each processor than extracts values for the N/P surfaces it owns. The *rvous* algorithm is faster for large surface element counts. A rendezvous style of communication is performed where every processor sends its tally contributions directly to the processor which owns the element as one of its N/P elements.

The *surfpush* keyword is only useful to use when SPARTA is having problems embedding a surface in the simulation grid, which occurs when when surface elements are defined via the *read_surf* command. Or for debugging purposes.

In rare cases, if a surface element point is just slightly inside or outside a grid cell, but within an epsilon distance from the surface of the grid cell, a numerical round-off error can occur when computing the cut volume. The

error can be avoided if such points are shifted (pushed) to a slightly different location, which only induces a tiny change in the computed cut volume. By default the *surfpush* keyword is set to *yes*, which will perform this "push" operation on a grid cell if the numerical issue is flagged. SPARTA prints out how many grid cells needed this push operation.

If you set *surfpush* to *no*, then the push operation is not performed, which will result in an error if the numerical issue occurs.

If the default *surfpush yes* still gives an error, then setting the *slo*, *shi*, and *svalue* allows experimentation with a different mode of pushing.

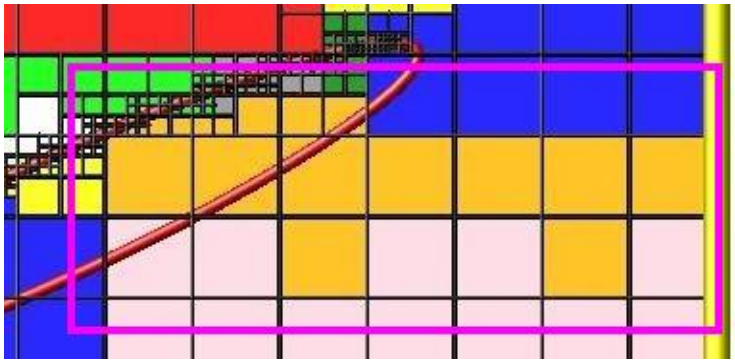
These 3 values are all multipliers on an epsilon of 1.0e-6 which is set internally in the code. Epsilon refers to a fraction of the size of a grid cell in each of its dimensions. Negative values for any of the 3 values distances inside a grid cell (inward from the cell face). Positive values are distances outside a grid cell (outward from the cell face). Zero values are exactly on the cell face. If any surface point (end points of 2d lines, corner points of 3d triangles) is between a *slo* to *shi* distance from any of the cell faces, then it is pushed to be a distance *svalue* from the face.

When *surfpush* is set to *yes*, SPARTA tries 2 kinds of pushing first, if the numerical issue is encountered for a grid cell. The first is *slo* = -1, *shi* = 1, *svalue* = 1, which means any point within a fractional distance (in each dimension) of 1.0e-6 inside the cell to 1.0e-6 outside the cell, is shifted to be a distance 1.0e-6 outside the cell. The second try is with *slo* = -1, *shi* = 1, *svalue* = 0, which puts the point on the face. If you set *slo*, *shi*, *svalue* explicitly, it will be the third option tried.

If you cannot get a surface to embed properly in a grid, meaning you get errors with the default setting of *surfpush yes*, then please contact the SPARTA developers. We will want to figure out what is unusual about your surface file!

The *gridcut* keyword determines the cutoff distance at which ghost grid cells will be stored by each processor. Assuming the processor owns a compact clump of grid cells (see below), it will also store ghost cell information from nearby grid cells, up to this distance away. If the setting is -1.0 (the default) then each processor owns a copy of ghost cells for all grid cells in the simulation. This can require too much memory for large models. If the cutoff is 0.0, processors own a minimal number of ghost cells. This saves memory but may require multiple passes of communication each timestep to move all the particles and migrate them to new owning processors. Typically a cutoff the size of 2-3 grid cell diameters is a good compromise that requires only modest memory to store ghost cells and allows all particle moves to complete in only one pass of communication.

An example of the *gridcut* cutoff applied to a clumped assignment is shown in this zoom-in of a 2d hierarchical grid with 5 levels, refined around a tilted ellipsoidal surface object (outlined in pink). One processor owns the grid cells colored orange. A bounding rectangle around the orange cells, extended by a short cutoff distance, is drawn as a purple rectangle. The rectangle contains only a few ghost grid cells owned by other processors.



Important: Using the *gridcut* keyword with a cutoff >= 0.0 is only allowed if the grid cells owned by each processor are "clumped". If each processor's grid cells are "dispersed", then ghost cells cannot be created with

a *gridcut* cutoff >= 0.0. Whenever ghost cells are generated, a warning to this effect will be triggered. At a later point when surfaces are read in or a simulation is performed, an error will result. The solution is to use the *balance_grid* command to change to a clumped grid cell assignment. See *Section 6.8* of the manual for an explanation of clumped and dispersed grid cell assignments and their relative performance trade-offs.

Important: If grid cells have already been defined via the *create_grid*, *read_grid*, or *read_restart* commands, when the *gridcut* cutoff is specified, then any ghost cell information that is currently stored will be erased. As discussed in the preceeding paragraph, a *balance_grid* command must then be invoked to regenerate ghost cell information. If this is not done before surfaces are read in or a simulation is performed, an error will result.

The *comm/sort* keyword determines whether the messages a proc receives for migrating particles (every step) and ghost grid cells (at setup and after re-balance) are sorted by processor ID. Doing this requires a bit of overhead, but can make it easier to debug in parallel, because simulations should be reproducible when run on the same number of processors. Without sorting, messages may arrive in a randomized order, which means lists of particles and grid cells end up in a different order leading to statistical differences between runs.

The *comm/style* keyword determines the style of particle communication that is performed to migrate particles every step. The most efficient method is typically for each processor to exchange messages with only the processors it has ghost cells for, which is the method used by the *neigh* setting. The *all* setting performs a relatively cheap, but global communication operation to determine the exact set of neighbors that need to be communicated with at each step.

For small processor counts there is typically little difference. On large processor counts the *neigh* setting can be significantly faster. However, if the flow is streaming in one dominant direction, there may be no particle migration needed to upwind processors, so the *all* method can generate smaller counts of neighboring processors.

Note that the *neigh* style only has an effect (at run time) when the grid is decomposed by the RCB option of the *balance* or *fix balance* commands. If that is not the case, SPARTA performs the particle communication as if the *all* setting were in place.

The *weight* keyword determines whether particle weighting is used. Currently the only style allowed, as specified by wstyle = *cell*, is per-cell weighting. This is a mechanism for inducing every grid cell to contain roughly the same number of particles (even if cells are of varying size), so as to minimize the total number of particles used in a simulation while preserving accurate time and spatial averages of flow quantities. The cell weights also affect how many particles per cell are created by the *create_particles* and *fix emit* command variants.

If the mode is set to *none*, per-cell weighting is turned off if it was previously enabled. For mode = *volume* or *radius*, per-cell weighting is enabled, which triggers two computations. First, at the time this command is issued, each grid cell is assigned a "weight" which is calculated based either on the cell *volume* or *radius*, as specified by the *mode* setting.

For the *volume* setting, the weight of a cell is its 3d volume for a 3d model, and the weight is its 2d area for a 2d model. For an axi-symmetric model, the weight is the 3d volume of the 2d axi-symmetric cell, i.e. the volume the area sweeps out when rotated around the y=0 axis of symmetry. The *radius* setting is only allowed for axisymmetric systems. The weight in this case is the distance the cell's midpoint is from the y=0 axis of symmetry. See *Section 6.2* for more details on axi-symmetric models.

Second, when a particle moves from an initial cell to a final cell, the initial/final ratio of the two cell weights is calculated. If the ratio > 1, then additional particles may be created in the final cell, by cloning the attributes of the incoming particle. E.g. if the ratio = 3.4, then two extra particle are created, and a 3rd is created with probability 0.4. If the ratio < 1, then the incoming particle may be deleted. E.g. if the ratio is 0.7, then the incoming particle is deleted with probability 0.3.

Note that the first calculation of weights is performed whenever the *global weight* command is issued. If particles already exist, they are not cloned or destroyed by the new weights. The second calculation only happens when a simulation is run.

The *particle/reorder* keyword determines how often the list of particles on each processor is reordered to store particles in the same grid cell contiguously in memory. This operation is performed every *nsteps* as specified. A value of 0 means no reordering is ever done. This option is only available when using the KOKKOS package and can improve performance on certain hardware such as GPUs, but is typically slower on CPUs except when running on thousands of nodes.

The *mem/limit* keyword limits the amount of memory allocated for several operations: load balancing, reordering of particles, and restart file read/write. This should only be necessary for very large simulations where the memory footprint for particles and grid cells is a significant fraction of available memory. In this case, these operations can trigger a memory error due to the additional memory they require. Setting a limit on the memory size will perform these operations more incrementally so that memory errors do not occur.

A load-balance operation can use as much as 3x more memory than the memory used to store particles (reported by SPARTA when a simulation begins). Particle reordering temporarily doubles the memory needed to store particles because it is performed out-of-place by default. Reading and writing restart files also requires temporary buffers to hold grid cells and particles and can double the memory required.

Specifying the value for *mem/limit* as *grid*, will allocate extra memory limited to the size of memory for storing grid cells on each processor. For most simulations this is typically much smaller than the memory used to store particles. Specifying a numeric value for *bytes* will allocate extra memory limited to that many MBytes on each processor. *Bytes* can be specified as a floating point value or an integer, e.g. 0.5 if you want to use 1/2 MByte of extra memory or 100 for a 100 MByte buffer. Specifying a value of 0 (the default) means no limit is used. The value used for *mem/limit* must not exceed 2GB or an error will occur.

For load-balancing, the communication of grid and particle data to new processors will then be performed in multiple passes (if necessary) so that only a portion of grid cells and their particles which fit into the extra memory are migrated in each pass. Similarly for particle reordering, multiple passes are performed using the extra memory to reorder the particles nearly in-place. For reading/writing restart files, multiple passes are used to read from or write to the restart file as well. For reading restart files, this option is ignored unless reading from multiple files (i.e. a "%" character was used in the command to write out the restart) and the number of MPI ranks is greater than the number of files.

Note that for these operations if the extra memory is too small, performance will suffer due to the large number of multiple passes required.

Restrictions:

The global surfmax command must be used before surface elements are defined, e.g. via the *read_surf command*.

Related commands:

mixture command

Default:

The keyword defaults are

- fnum = 1.0
- nrho = 1.0
- vstream = 0.0 0.0 0.0
- temp = 273.15
- gravity = 0.0 0.0 0.0 0.0

- surfs = explicit
- surfgrid = auto
- surfmax = 100
- cellmax = 100
- splitmax = 10
- surftally = auto
- surfpush = yes
- gridcut = -1.0
- comm/sort = no
- comm/style = neigh
- weight = cell none
- particle/reorder = 0
- mem/limit = 0.

1.14.72 group command

Syntax:

```
group ID which style args
```

- ID = user-defined name of the grid or surface group
- which = *grid* or *surf*
- style options for which = grid: *region* or *subtract* or *union* or *intersect* or *clear*
- style options for which = surf: *type* or *id* or *region* or *subtract* or *union* or *intersect* or *clear*

```
type or id args
  args = list of one or more surface element types or IDs
    any entry in list can be a range formatted as A:B
    A = starting index, B = ending index
  args = logical value
    logical = "<" or "<=" or ">" or ">=" or "==" or "!="
    value = a surface element type or ID
  args = logical value1 value2
    logical = "<>"
    value1,value2 = surface element types or IDs
region args = region-ID rflag
  region-ID = ID of region which grid cell or surface element must be in
  rflag = one or all or center
    any = one (or more) corner points of grid cell or surface element in region
    all = all corner points of grid cell or surface element in region
    any = center point of grid cell or surface element in region
subtract args = two or more group IDs
union args = one or more group IDs
intersect args = two or more group IDs
clear = no args
```

Examples:

```
group sphere surf type 1 3
group sphere surf id 50 100:150
group sphere surf id <= 1000
group sphere surf id <> 50 250
group patch grid region leftedge
group patch surf region cutout
group boundary surf subtract all a2 a3
group boundary grid union lower upper
group boundary surf union lower upper
group boundary surf intersect upper leftside
```

Description:

Assign grid cells to grid groups or surface elements to surface groups. In SPARTA, a "grid group" is a collection of one or more grid cells. A "surface" group is a collection of one or more surface elements (line segements in 2d, triangles in 3d). Other commands take group IDs as arguments so that they act on a set of grid cells or surface elements. For example, see the *compute grid*, *compute surf*, *fix ave/grid*, *fix ave/surf*, *dump grid*, or *dump surf* commands.

An individual grid cell can belong to multiple grid groups. An individual surface element can belong to multiple surface groups. Each grid or surface group has a name which is specified as the *ID* in this command. Each grid group and surface group ID must be unique, though the same ID can be used for both a grid and surface group. IDs can only contain alphanumeric characters and underscores.

If the specified group ID already exists, grid cells or surface elements are added to the group. Otherwise a new group is created. This means the group command can be used multiple times with the same group ID to incrementally add grid cells or surface elements to the group.

A grid group with the ID *all* is pre-defined. All grid cells belong to this group. Likewise, a surface group with the ID *all* is pre-defined. All surface elements belong to this group.

After this command has performed its grid cell or surface elements assignments, statistics about the group are printed to the screen, so that you can check if the command operated as you expect.

Note that this command assigns all flavors of child grid cells to groups, which includes unsplit, cut, split, and sub cells. See *Section 6.8* of the manual gives details of how SPARTA defines child, unsplit, split, and sub cells.

The following styles can be used for grid groups.

The *region* style puts all grid cells in the region volume associated with the *region-ID* into the group. See the *region* command for details on what kind of geometric regions can be defined. Note that the *side* option for the *region* command can be used to define whether the inside or outside of the geometric region is considered to be "in" the region.

The *rflag* setting determines how a grid cell is judged to be in the region or not. For *rflag* = *one*, it is in the region if any of its corner points (4 for 2d, 8 for 3d) is in the region. For *rflag* = *all*, all its corner points must be in the region. For *rflag* = *center*, the center point of the grid cell must be in the region.

The following styles can be used for surface groups.

The *type* and *id* styles put all surface elements with the specified types or surface element IDs into the group. These two styles can use arguments specified in one of two formats.

For surface elements, the "type" of each element is defined when the elements are read from a surface file, via the *read_surf* command. In the file, a positive integer type value can be optionally defined for each element (default = 1). The specified type values can also be incremented using the *typeadd* keyword of the *read_surf* commmand.

For surface elements, the "ID" of each element is simply its index from 1 to N, for all N surface elements that have been read in via the *read_surf* command. The ordering of IDs is determined by the order the elements appear in the read-in surface file. If multiple files are read (or the same file multiple times), IDs increase monotonically each time new surface elements are added.

The first format is a list of values (types or IDs). For example, the first command in the examples above puts all surface elements of type 1 and 3 into the group named sphere. Each entry in the list can optionally be a colon-separated range A:B, as in the second axample above. A "range" is a series of values (types or IDs). The second example with 100:150 adds all surface elements with IDs from 100 to 150 (inclusive) to the group named sphere, along with element 50 since it also appears in the list of values.

The second format is a logical operator followed by one or two values (type or ID). The 7 valid logicals are listed above. All the logicals except "<>" take a single argument. The third example above adds all surface elements with IDs from 1 to 1000 to the group named sphere. The logical "<>" means "between" and takes 2 arguments. The fourth example above adds all surface elements IDs from 50 to 250 (inclusive) to the group named sphere.

The *region* style puts all surface elements in the region volume associated with the *region-ID* into the group. See the *region* command for details on what kind of geometric regions can be defined. Note that the *side* option for the *region* command can be used to define whether the inside or outside of the geometric region is considered to be "in" the region.

The *rflag* setting determines how a surface element is judged to be in the region or not. For *rflag* = *one*, it is in the region if any of its corner points (3 for triangle, 2 for line) is in the region. For *rflag* = *all*, all its corner points must be in the region. For *rflag* = *center*, the center point of the line segment or centroid point of the triangle must be in the region.

The following styles can be used for either grid or surface groups.

The *subtract* style takes a list of two or more existing group names as arguments. All grid cells or surface elements that belong to the 1st group, but not to any of the other groups are added to the specified group.

The *union* style takes a list of one or more existing group names as arguments. All grid cells or surface elements that belong to any of the listed groups are added to the specified group.

The *intersect* style takes a list of two or more existing group names as arguments. Grid cells or surface elements that belong to every one of the listed groups are added to the specified group.

The *clear* style un-assigns all grid cells or surface elements that were assigned to that group. This is a way to empty a group before adding more grid cells or surface elements to it.

Restrictions:

No more than 32 grid groups and no more than 32 surface groups can be defined, including "all".

Related commands:

dump command, *region command*, *compute grid compute surf*

Default:

All grid cells belong to the "all" grid group. All surface elements belong to the "all" surface group.

1.14.73 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 "${steps} > 1000" then quit
if "${myString} == a10" then quit
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 $n<10000 &
  "timestep 0.01" &
else &
  "timestep 0.02" &
  "print 'Max step reached'"
if "${eng} > ${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, except an *include* command, which is not allowed. 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 *Section commands 2* 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"' &
  "run 50000"
```

Note that if one of the commands to execute is *quit*, as 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 (which start with a digit or period or minus sign) or strings (which start with a letter and can contain alphanumeric characters or underscores):

```
0.2, 100, 1.0e20, -15.4, etc
InP, myString, a123, ab_23_cd, 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 string or a variable reference like $a or ${abc}, or A or B can be another Boolean expression.

If a variable is used it can produce a number when evaluated, like an *equal-style variable*. Or it can produce a string, like an *index-style variable*. For an individual Boolean operator, A and B must both be numbers or must both be strings. You cannot compare a number to a string.

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.

When the 6 relational operators (first 6 in list above) compare 2 numbers, they return either a 1.0 or 0.0 depending on whether the relationship between A and B is TRUE or FALSE. When the 6 relational operators compare 2 strings, they also return a 1.0 or 0.0 for TRUE or FALSE, but the comparison is done by the C function strcmp().

When the 3 logical operators (last 3 in list above) compare 2 numbers, they also return either a 1.0 or 0.0 depending on whether the relationship between A and B is TRUE or FALSE (or just A). 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 3 logical operators can only be used to operate on numbers, not on strings.

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 command print command

Default:

none

1.14.74 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 command, *jump command*

Default:

none

1.14.75 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: The SELF option is not guaranteed to work when the current input script is being read through stdin (standard input), e.g.

```
spa_g++ < in.script
```

since the SELF option invokes the C-library rewind() call, which may not be supported for stdin on some systems or by some MPI implementations. This can be worked around by using the *-in command-line argument*, e.g.

```
spa_g++ -in in.script
```

or by using the *-var command-line argument* to pass the script name as a variable to the input script. In the latter case, a *variable* called "fname" could be used in place of SELF, e.g.

```
spa_g++ -var fname in.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 grid all 100 file.$a
run 10000
undump 1
next a
jump in.flow 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 command, *include command*, *label command*, *next command*

Default:

none

1.14.76 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

1.14.77 log command

Syntax:

```
log file keyword
```

- file = name of new logfile
- keyword = *append* if output should be appended to logfile (optional)

Examples:

```
log log.equil
log log.equil append
```

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 the optional keyword *append* is specified, then output will be appended to an existing log file, instead of overwriting it.

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 *Section 2.6* for details.

Restrictions:

none

Related commands:

none

Default:

The default SPARTA log file is named log.sparta

1.14.78 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* or *copy* or *delete*

  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)

  trot arg = Trot Trot = rotational temperature of entire mixture (temperature units)

  tvib arg = Tvib Tvib = vibrational 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

  copy arg = new-ID new-ID = ID of new mixture to create, as a copy of this one

  delete args = sp1 sp2 . . . sp1,sp2,. . . = species to delete from the mixture

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 copy myAir
mixture background N O
mixture air delete N NO
```

Description:

Define a gas mixture and its properties. A mixture can be referenced by its ID in several other SPARTA commands such as *create_particles* or *per-grid computes*. Any number of mixtures can be defined and used in a simulation.

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 both global attributes and per-species attributes. All attributes have default values unless they are explicitly specified.

The ID for a mixture is used to identify the mixture in other commands. Each mixture ID must be unique. The ID can only contain alphanumeric characters and underscores.

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. Also note that a species can belong to more than one mixture.

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 defines one group per species, each with the species name, so that each species in the mixture 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 assignment 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.

These keywords set global attributes of the mixture.

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 use 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 each particle's velocity is initialized.

The *trot* keyword sets a global attribute of the mixture, namely the rotational temperature of its particles. When particles are created, this value is used to sample a Gaussian energy distribution to define each particle's rotational energy. If this keyword is not specified, the thermal temperature is used as the default.

The *tvig* keyword sets a global attribute of the mixture, namely the vibrational temperature of its particles. When particles are created, this value is used to sample a Gaussian energy distribution to define each particle's vibrational energy. If this keyword is not specified, the thermal temperature is used as the default.

This keyword sets per-species attributes of the mixture.

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 a mixture is assigned to exactly one group. The *group* keyword can be used to set or change these assignments. Every mixture has one or more named groups.

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 per-grid quantities on a per-grid-cell, per-group basis, i.e. on subsets of particles within each grid cell.

If the *group* keyword is not used in a mixture command, no changes to group assignements are made for species that are already in the mixture. If one or more new species are specified, then all of them are assigned to a group with "default" as the group ID. Note that this means that mixtures defined with mixture commands that never use the *group* keyword will have just a single group.

If the *group* keyword is used, the 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 no 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. This includes the "default" group if it was implicitly created by a previous mixture command.

- If no species are listed in the mixture command and the group ID is SELF, then every 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 no 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.

These keywords operate on one or more mixtures.

The *copy* keyword creates a new mixture with *new-ID* which is an identical copy of the mixture with *ID*. Regardless of where the *copy* keyword appears in the command, the operation is delayed until all other keywords have been invoked.

This is useful if you wish to create a new mixture which is nearly the same as the current mixture. Subsequent mixture commands can be used to change the properties of the new mixture.

The *delete* keyword removes one or more species from the mixture, specified as *sp1*, *sp2*, etc. No other keywords can be used with *delete*. All arguments that follow it are assumed to be species IDs that are currently in the mixture. When using *delete*, no species can be defined before the keyword, i.e. *species1*, *species2*, etc cannot be defined in the comand syntax described above.

After the listed species are removed, any group IDs for the mixture that have no species assigned to them are also deleted.

Restrictions:

The streaming velocity and thermal temperature of the mixture cannot both be zero. A zero streaming velocity means a zero vector = (0,0,0).

The restrictions on use of the *delete* keyword are described above.

Related commands:

global command, *create_particles command*

Default:

The *nrho*, *vstream*, and *temp* defaults are those defined for the background gas density, as set by the *global* command. The *trot* and *tvib* defaults are to use the thermal temperature *temp*, either its default or the value specified by this 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 assigned to a group with a group ID = "default".

1.14.79 move_surf command

1.14.80 Syntax:

```
move_surf groupID style args ... keyword value ...
```

- group-ID = group ID for which surface elements to move
- style = *file* or *trans* or *rotate*

```
file args = filename entry
trans args = Dx Dy Dz
  Dx,Dy,Dz = displacement applied to all surface points (distance units)
rotate args = theta Rx Ry Rz Ox Oy Oz
  theta = rotate surface points by this angle in counter-clockwise direction␣
↪(degrees)
  Rx,Ry,Rz = rotate around vector starting at origin pointing in this direction
  Ox,Oy,Oz = origin to rotate around (distance units)
```

- zero or more keyword/value pairs may be appended
- keyword = *connect*

```
connect arg = yes or no
```

Examples:

```
move_surf all trans 1 0 0
move_surf partial rotate 360 0 0 1 5 5 0 connect yes
move_surf object2 rotate 360 0 0 1 5 5 0
```

Description:

This command performs a one-time movement of all the surface elements in the specified group via the specified style. See the *group surf* command for info on how surface elements can be assigned to surface groups.

This command can be invoked as many times as desired, before or between simulation runs. Surface points can also be moved on-the-fly during a simulation by using the *fix move/surf command*.

Moving surfaces between simulations can be useful if you want to perform a series of runs from one input script, where some attribute of the surface elements change, e.g. the separation between two spheres.

Important: The *file* style is not yet implemented. It will allow new positions of points to be listed in a file.

In 2d, surface elements are line segments with 2 vertices each. In 3d, surface elements are triangles with 3 vertices each. If a line segment or triangle belongs to the specified group, all of its vertices are moved. This effectively moves the entire surface element.

Important: Unless a vertex is on the simulation box boundary, it will be part of two surface elements (in 2d) or multiple surface elements (in 3d). If you choose a surface groupID which does not include all the elements in a gridded object, then you cannot move them without breaking apart the object in a "watertight" sense (so that particles could erroneously move inside the object). To prevent this use the optional *connect* keyword with its *yes* setting. This will insure that multiple copies of the same vertex in other elements (not in the surface group) will also be moved.

This is a way to morph the shape of a gridded object, e.g. make a sphere more oblate, by moving only a portion of its elements.

The *trans* style shifts or displaces each vertex by the vector (Dx,Dy,Dz).

The *rotate* style rotates the coordinates of all vertices by an angle *theta* in a counter-clockwise direction, around the vector starting at (Ox,Oy,Oz) and pointing in the direction *Rx,Ry,Rz*. Any desired rotation can be represented by an appropriate choice of (Ox,Oy,Oz), *theta*, and (Rx,Ry,Rz).

After the surface has been moved, then all particles in grid cells that meet either of these criteria are deleted:

- the grid cell is now inside a surface
- the grid cell overlaps with a surface element that moved

This is to prevent particles from ending up inside surface objects.

Note that in this context, "overlaps" means that any part of the surface element touches any part of the grid cell, including its surface. Also note that if a surface element object (e.g. a sphere) moved a long distance then grid cells that were inside the object in its old position and thus contained no particles, will still have no particles immediately after the move. This will effectively leave a "void" in the flow until particles re-fill the grid cells that are now outside the object.

Restrictions:

An error will be generated if any surface element vertex is moved outside the simulation box.

Related commands:

read_surf command, *fix move/surf command remove_surf command*

Default:

The option default is connect = no.

1.14.81 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*, *file*, *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 of values. A *file*-style variable reads the next line from its associated file. *String*- or *particle*- 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. *File*-style variables are exhausted when the end-of-file is reached.

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 *file*-style variables, the next line is read from its file and the string assigned to the variable.

When the next command is used with *universe*- or *uloop*-style variables, all *universe*- or *uloop*-style variables must be listed in the next command. This is because of the manner in which the incrementing is done, using a single lock file for all variables. The next value (for each variable) is assigned to whichever processor partition executes the command first. All processors in the partition are assigned the same value(s). Running SPARTA on multiple partitions of processors via the "-partition" command-line switch is described in *Section 2.6* of the manual. *Universe*- and *uloop*-style variables are incremented using the files "tmp.sparta.variable" and "tmp.sparta.variable.lock" which you will see in your directory during and after 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.flow, 8 simulations would be run using surface data files from directories run1 thru run8.

```
variable d index run1 run2 run3 run4 run5 run6 run7 run8
shell cd $d
create_box 0 10 0 10 0 10
create_grid 100 100 100
read_surf data.surf 1
...
run 10000
shell cd ..
clear
next d
jump in.flow
```

If the variable "d" were of style *universe*, and the same in.flow 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
```

(continues on next page)

(continued from previous page)

```
  clear
  ...
  read_surf data.surf.$i$j 1
  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 command, *include command*, *shell command*, *variable command*

Default:

none

1.14.82 package command

Syntax:

```
package style args
```

- style = *kokkos*
- args = arguments specific to the style

- kokkos args = keyword value ...

  zero or more keyword/value pairs may be appended

  keywords = comm or reduction

  - comm value = threaded or classic
    - threaded = perform pack/unpack using KOKKOS (e.g. on GPU or using OpenMP) (default)
    - classic = perform communication pack/unpack in non-KOKKOS mode
  - reduction = parallel/reduce or atomic
    - parallel/reduce = use parallel reduction for statistics (default)
    - atomic = use atomic reduction for statistics
  - collide/extra = factor
    - factor = increase memory used for collisions by this factor (default)
  - collide/retry = yes or no
    - yes = retry collision algorithm until successful
    - no = do not retry collision algorithm (default)
  - gpu/direct = yes or no - yes = use GPU-direct, i.e. CUDA-aware MPI (default) - no = do not use GPU-direct

Examples:

```
package kokkos comm classic
package kokkos comm threaded reduction atomic
package kokkos gpu/direct no
```

Description:

This command invokes package-specific settings for the KOKKOS accelerator package available in SPARTA.

If this command is specified in an input script, it must be near the top of the script, before the simulation box has been created. This is because it specifies settings that the accelerator package used in its initialization, before a simulation is defined.

This command can also be specified from the command-line when launching SPARTA, using the "-pk" *command-line switch*. The syntax is exactly the same as when used in an input script.

Note that the KOKKOS accelerator package requires the package command to be specified, if the package is to be used in a simulation (SPARTA can be built with the accelerator package without using it in a particular simulation). However, a default version of the command is typically invoked by other accelerator settings. For example, the KOKKOS package requires a "-k on" *command-line switch* respectively, which invokes a "package kokkos" command with default settings.

Note: A package command for a particular style can be invoked multiple times when a simulation is setup, e.g. by the "-k on", "-sf", and "-pk" *command-line switches*, and by using this command in an input script. Each time it is used all of the style options are set, either to default values or to specified settings. I.e. settings from previous invocations do not persist across multiple invocations.

See the the *Accelerating SPARTA* section of the manual for more details about using the various accelerator packages for speeding up SPARTA simulations.

The *kokkos* style invokes settings associated with the use of the KOKKOS package.

All of the settings are optional keyword/value pairs. Each has a default value as listed below.

The *reduction* keyword sets the type of reduction used to gather statistics. The *parallel/reduce* option uses a parallel reduction and is typically the preferred method when running on CPUs and Xeon Phis. The *atomic* option uses thread atomics and is typically faster when running on GPUs.

Chemical reactions can increase the number of particles in the simulation, which requires extra memory storage. It is not possible to resize Kokkos data structures during the collide routine, so two workarounds are provided. The default is to use the *collide/extra* keyword, which ensures there is extra memory allocated to store new particles. For example, if *collide/extra* is set to 1.1, then the memory is over-allocated by 10%. If this space is still not sufficient to hold new particles, the code will error out and the simulation must be restarted using a larger value for *collide/extra*. Alternatively, if the *collide/retry* option is set to *yes*, backup copies of the Kokkos data structures are created. If space is exceeded during the collide routine, the Kokkos data structures are restored from backup, their size is increased, and the collide routine is started over from the beginning. This guarantees that the collide routine will eventually succeed without producing an error, but increased memory by a factor of 2 and also has overhead from making a backup copy of the data. If the *collide/retry* option is set to *yes*, the *collide/extra* keyword will be ignored. If reactions are not defined, both these options will be ignored.

The *comm* keyword determines whether the host or device performs the packing and unpacking of data when communicating per-atom data between processors. The value options are *threaded* or *classic*.

The optimal choice for this keyword depends on the hardware used. When running on CPUs or Xeon Phi, the *classic* option is typically fastest. When using GPUs, the *threaded* value will typically be optimal. In this case data can stay on the GPU for many timesteps without being moved between the host and GPU. This requires that your MPI is able to access GPU memory directly. Currently that is true for OpenMPI 1.8 (or later versions), Mvapich2 1.9 (or later), and CrayMPI.

The *gpu/direct* keyword chooses whether GPU-direct will be used. When this keyword is set to *on*, buffers in GPU memory are passed directly through MPI send/receive calls. This can reduce overhead of first copying the data to the host CPU. However GPU-direct is not supported on all systems, which can lead to segmentation faults and would require using a value of *off*.

Restrictions:

This command cannot be used after the simulation box is defined by a *create_box* command.

The kk style of this command can only be invoked if SPARTA was built with the KOKKOS package. See the *Making SPARTA* section for more info.

Related commands:

suffix command, "-pk" *command-line setting <start-command-line-options*

Default:

For the KOKKOS package, the option defaults are comm = threaded, reduction = parallel/reduce, collide/extra = 1.1, and collide/retry = no, gpu/direct yes. These settings are made automatically by the required "-k on" *command-line switch*. You can change them by using the package kokkos command in your input script or via the "-pk kokkos" *command-line switch*.

1.14.83 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 2.6* of the manual 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 an 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).

Restrictions:

none

Related commands:

none

Default:

none

1.14.84 print command

Syntax:

print string keyword value:pre

- string = text string to print, which may contain variables
- zero or more keyword/value pairs may be appended
- keyword = *file* or *append* or *screen*

```
file value = filename
append value = filename
screen value = yes or no
```

Examples:

```
print "Done with equilibration"
print 'Done with equilibration'
print "Done with equilibration" file info.dat
```

```
compute myTemp temp
variable t equal c_myTemp
print "The system temperature is now $t"
```

Description:

Print a text string to the screen and logfile. One line of output is generated. The text string must be a single argument, so it should be enclosed in quotes if it is more than one word. If it contains variables, they will be evaluated and their current values printed.

If the *file* or *append* keyword is used, a filename is specified to which the output 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 to the screen and logfile can be turned on or off as desired.

If you want the print command to be executed multiple times (e.g. 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, global values calculated by a *compute* or *fix*, or references to other *variables*.

Restrictions:

none

Related commands:

fix print command, *variable command*

Default:

The option defaults are no file output and screen = yes.

1.14.85 quit command

Syntax:

```
quit
```

Examples:

```
quit
if "$n > 10000" then quit
```

Description:

This command causes SPARTA to exit, after shutting down all output cleanly.

It can be used as a debug statement in an input script, to terminate the script at some intermediate point.

It can also be used as an invoked command inside the "then" or "else" portion of an *if* command.

Restrictions:

none

Related commands:

if command

Default:

none

1.14.86 react command

Syntax:

```
react style args
```

- style = *none* or *tce* or *qk* or *tce/qk*
- args = arguments for that style

```
none args = none
tce args = infile
  infile = file with list of gas-phase chemistry reactions
qk args = infile
  infile = file with list of gas-phase chemistry reactions
tce/qk args = infile
  infile = file with list of gas-phase chemistry reactions
tce/kk args = infile
  infile = file with list of gas-phase chemistry reactions
```

Examples:

```
react none
react tce air.tce
react qk air.tce
```

Description:

Define chemical reactions to perform in the gas phase when particle-particle collisions occur. See the *surf_react* command for specification of surface chemistry reactions.

The *none* style means that no chemistry will be performed, which is the default.

For other styles, a file is specified which contains a list of chemical reactions, with their associated parameters. The reactions are read into SPARTA and stored in a list. Each time a simulation is run via the *run* command, the list is scanned. Only reactions for which all the reactants and all the products are currently defined as species-IDs will be active for the simulation. Thus the file can contain more reactions than are used in a particular simulation. See the *species* command for how species IDs are defined.

The reaction models for the various styles are described below. When a pair of particles collide, the list of all reactions with those two species as reactants is looped over. A probability for each reaction is calculated, using the formulas discussed below, and a random number is used to decide which reaction (if any) takes place. No check is made that the sum of probabilities for all possible reactions is <= 1.0, but that should normally be the case if reasonable reaction coefficients are defined.

The format of the reaction file is the same for all three of the currently defined styles, and is also described below. The various styles interpret and compute the specified reactions in different ways. The data directory in the SPARTA distribution contains reaction files for these reaction models, all with the suffix ".tce".

The *tce* style is Bird's Total Collision Energy (TCE) model. When this style is specified, all computed reactions will use the TCE model.

Using kinetic theory, the TCE model allows for reaction probabilities to be defined based on known, measured, reaction rates. The model is described in detail in [Bird94]; see chapter 6. The required input parameters for each reaction (discussed below) are values that permit its effective Arrhenius rate to be calculated, namely

$$K(T) = AT^b e^{-E_a/kT}$$

where K(T) is the forward reaction rate, T is the temperature of the participating molecules which is a function of their velocities and internal energy states, k the Boltzmann constant, and A,b,Ea are input parameters as discussed below.

All 5 reactions coefficients read from the reaction file (described below) are used to calculate terms in equation 6.10 of [Bird94] for the probability that a reaction takes place.

The C2, C3, C4 values are the Arrhenius activation energy Ea, prefactor A, and exponent b, used in the rate formula above.

The *qk* style is Bird's Quantum-Kinetic model (QK). When this style is specified, all computed reactions will use the QK model.

The QK model implemented is that of [Bird09] as validated [Gallis09] and modified [Gallis10].

The QK model depends solely on properties of the colliding molecules and unlike the TCE model makes no use of measured reaction rates or adjustable parameters. The macroscopic properties used in the QK model are the available collision energy, activation energies, and quantized vibrational energy levels.

According to the QK model dissociation reactions take place when the maximum obtainable vibrational energy after an inelastic energy exchange is higher than the dissociation level [Bird09].

$$int[E_c/(k\Theta_v)] > \Theta_d/\Theta_v$$

Exchange reactions take place when the vibrational energy after a trial energy exchange is above the activation energy of the exchange reaction [Gallis10].

$$i_v > int[E_a/(k\Theta_v)]$$

A new version of the QK model for exchange reactions has been proposed by [Bird11]. This will be implemented in future releases of SPARTA.

For the QK model, SPARTA reads the same 5 coefficients per reaction from the reaction file (described below) as for the TCE model. Three of the coefficients (C1,C2,C5) are used to calculate terms in equation 6.10 of [Bird94] for the probability that a reaction takes place. The Arrhenius rate parameters C3 and C4 are ignored by the QK model.

The *tce/qk* style is a hybrid model which can be used to compute reactions using both the TCE and QK models. When this style is specified, reactions from the input file that are flagged with an A = Arrhenius style will be computed using the TCE model. Reactions from the input file that are flagged with a Q = Quantum style will be computed using the QK model.

The format of the input reaction file is as follows. Comments or blank lines are allowed in the file. Comment lines start with a "#" character. All other entries must come in 2-line pairs with values separated by whitespace in the following format

```
R1 + R2 + ... --> P1 + P2 + ...
type style C1 C2 ...
```

The first line is a text-based description of a single reaction. R1, R2, etc are one or more reactants, listed as *species* IDs. P1, P2, etc are one or more products, also listed as *species* IDs. The number of allowed reactants and products depends on the reaction type, as discussed below. In most cases there is no restriction on the order or listed reactants or products on what species are listed. Exceptions are detailed below. Note that individual reactants and products must be separated by whitespace and a "+" sign. The left-hand and right-hand sides of the equation must be separated by whitespace and "–>".

The *type* of each reaction is a single character (upper or lower case) with the following meaning. The type determines how many reactants and products can be specified in the first line.

```
D = dissociation = 2 reactants and 3 products
E = exchange = 2 reactants and 2 products
I = ionization = 2 reactants and 2 or 3 products
R = recombination = 2 reactants and 1 product (see below)
```

A dissociation reaction means that R1 dissociates into P1 and P2 when it collides with R2. R2 is preserved in the collision, so P3 = R2 is required.

An exchange reaction is a collision between R1 and R2 that results in new products P1 and P2. There is no restriction on the species involved in the reaction.

An ionization reaction with 2 products is typically a collision between R1 and R2 that results in a positively charged ion and an election. See the discussion on ambipolar reactions below. However, SPARTA does not check for this, so there is no restriction on the species involved in the reaction.

An ionization reaction with 3 products is typically a collision between a neutral R1 and an electon R2 which ejects an electron from the neutral species, resulting in an ion P1 and a new electron P2. See the discussion on ambipolar reactions below. Again, SPARTA does not check for this, so there is no restriction on the species involved in the reaction. R2 is preserved in the collision, so P3 = R2 is required.

A recombination reaction is a collision between R1 and R2 that results in P1. There is no restriction on the species involved in the reaction.

Note that recombination reactions actually involve a 3rd particle whose species is not altered by the reaction but whose velocity is, in order to balance energy and momentum. So conceptually it can be thought of as both a reactant and a product. There are 3 ways you can specify recombination reactions, to include information about which species of 3rd particles are eligible to participate:

```
R1 + R2 -> P1
R1 + R2 -> P1 + atom/mol
R1 + R2 -> P1 + P2
```

In the first case, no info for a 3rd particle is listed. This means any species of 3rd particle can be used. In the second case, a non-species keyword is used, either "atom" or "mol". This means the 3rd particle must be either an atomic species, or a molecular species. This is based on the vibrational degrees of freedom listed in the *species file*. A non-zero DOF is molecular; zero DOF is atomic. In the third case, a specific species P2 is listed. This means the 3rd particle must be that species.

Note that for the same R1 and R2, multiple recombination reactions can be listed in the reaction file. When two particles R1 and R2 are selected for collision and a possible reaction, if any recombination reaction is defined for R1 and R2, then a 3rd particle in the same grid cell is randomly selected. Its species P2 is used to match at most one of the possibly multiple recombination reactions for R1 and R2. Only that recombination reaction is checked for a reaction as a possible outcome of the collision.

This matching is done from most-specific to least-specific, i.e. the reverse ordering of the 3 cases above. If there is a defined reaction that lists P2 (third case, most specific), it is used. If not, and there is a defined reaction for "atom" or "mol" that corresponds to P2 (second case, intermediate specificity), then it is used. If not, and there is a defined reaction with no P2 (first case, least specific), then it is used. If none of these matches occur, no recombination reaction is possible for that collision between R1 and R2. Note that these matching rules means that for the same R1 and R2, you can list two reactions, one with P2 = "atom", and one with P2 = "mol". And/or you can list multiple reactions of the third kind, each with a unique P2.

Important: If the ambipolar approximation is being used, via the *fix ambipolar* and *collide_modify ambipolar yes* commands, then reactions which involve either ambipolar ions or the ambipolar electron have more restricitve rules about the ordering of reactants and products. See the next section for a discussion of these requirements.

The *style* of each reaction is a single character (upper or lower case) with the following meaning:

- A = Arrhenius
- Q = Quantum

The style determines how many reaction coefficients are listed as C1, C2, etc, and how they are interpreted by SPARTA.

For both the A = Arrhenius style and Q = Quantum style, there are 5 coefficients:

- C1 = number of internal degrees of freedom (as defined by the TCE model)
- C2 = Arrhenius activation energy Ea
- C3 = Arrhenius prefactor A
- C4 = Arrhenius exponent b
- C5 = overall reaction energy (positive for exothermic)

The different reaction styles use these values in different ways, as explained above.

If the ambipolar approximation is being used, via the *fix ambipolar* command, then reactions which involve either ambipolar ions or the ambipolar electron have more restricitve rules about the ordering of reactants and products, than those described in the preceeding section.

Note that ambipolar collisions are turned on via the *collide_modify ambipolar yes* commands, which in turn requries that the *fix ambipolar* is defined in your input script. This fix defines a particular species as an ambipolar electron, written as "e" in the reactions that follow. It also defines a list of ambipolar ions, which are written as species with a trailing "+" sign in the rules that follow. Neutral species (without "+") can be any non-ambipolar species.

These rules only apply to reactions that involve ambipolar species (ions or electrons) as a reactant or product. Note that every ambipolar reaction written here conserves charge. I.e. the net charge of the reactants equals the net charge of the products.

Ambipolar dissociation reactions must list their reactants and products in one of the following orders:

```
AB + e -> A + e + B
AB+ + e -> A+ + e + B
```

Ambipolar ionization reactions with 2 or 3 products must be in one of the following orders:

```
A + B -> AB+ + e
A + e -> A+ + e + e
```

Ambipolar exchange reactions must be in one of the following orders:

```
AB+ + e -> A + B
AB+ + C -> A + BC+
C + AB+ -> A + BC+
```

Ambipolar recombination reactions must be in the following order:

```
A+ + e -> A
A + B+ -> AB+
A+ + B -> AB+
```

A third particle for recombination reactions can be specified in the same way as described above for non-ambipolar recombination.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix* command in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

none

Related commands:

collide command surf_react command

Default:

style = none

1.14.87 react_modify command

Syntax:

```
react_modify keyword values ...
```

- one or more keyword/value pairs may be listed
- keywords = *recomb* or *rboost*

```
recomb value = yes or no = enable or disable defined recombination reactions
rboost value = rfactor
  rfactor = boost probability of recombination reactions by this factor
```

Examples:

```
react_modify recomb no
react_modify rboost 100.0
```

Description:

Set parameters that affect how reactions are performed.

The *recomb* keyword turns on or off recombination reactions. It is only relevant if recombination reactions were defined in the reaction file read in by the *react* command. If the setting is *no* then they will be disabled even if they were listed in the reaction file. This is useful to turn recombination reactions off, to see if they affect simulation results.

The *rboost* keyword is a setting for recombination reactions. It is ignored if no recombination reactions exist, or the *recomb* keyword is set to *no*. The *rboost* setting does not affect the overalll statistical results of recombination

reactions, but tries to improve their computational efficiency. Recombination reactions typically occur with very low probability, which means the code spends time testing for reactions that rarely occur. If the *rfactor* is set to N > 1, then recombination reactions are skipped N-1 out of N times, when one or more such reactions is defined for a pair of colliding particles. A random number us used to select on that probability. To compensate, when a recombination reaction is actually tested for occurrence, its rate is boosted by a factor of N, making it N times more likely to occur.

The smallest value *rboost* can be set to is 1.0, which effectively applies no boost factor.

Important: Setting *rboost* too large could meant the probability of a recombination reaction becomes > 1.0, when it is does occur. SPARTA does not check for this, so you should estimate the largest boost factor that is safe to use for your model.

Restrictions:

none

Related commands:

react command

Default:

The option defaults are recomb = yes and rboost = 1000.0.

1.14.88 read_grid command

Syntax:

```
read_grid filename
```

- filename = name of grid file

Examples:

```
read_grid grid.overlay
```

Description:

Read in a grid description from a file, which will overlay the simulation domain defined by the *create_box* command. The grid can also be defined by the *create_grid* command.

The grid in SPARTA is hierarchical. The entire simulation box is a single parent grid cell at level 0. It is subdivided into Nx by Ny by Nz cells at level 1. Each of those cells can be a child cell (no further sub-division) or can be a parent cell which is further subdivided into Nx by Ny by Nz cells at level 2. This can recurse to as many levels as desired. Different cells can stop recursing at different levels. Each parent cell can define its own unique Nx, Ny, Nz values for subdivision. Note that a grid with a single level is simply a uniform grid with Nx by Ny by Nz cells in each dimension.

In the current SPARTA implementation, all processors own a copy of all parent cells. Each child cell is owned by a unique processor. They are assigned by this command to processors in a round-robin fashion, as they are created at each level when the file is read. This is a "dispersed" assignment of child cells to each processor.

Important: See *Section 6.8* of the manual for an explanation of clumped and dispersed grid cell assignments and their relative performance trade-offs. The *balance_grid* command can be used after the grid is created, to assign child cells to processors in different ways. The "fix balance" command can be used to re-assign them in a load-balanced manner periodically during a running simulation.

The specified file can be a text file or a gzipped text file (detected by a .gz suffix).

A grid file contains only a listing of parent cells. Child cells are inferred from the parent cell definitions.

A grid 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 one or more 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 grid 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 (only one for this file). Header lines can come in any order. The value(s) are read from the beginning of the line. Thus the keyword *parents* should be in a line like "1000 parents".

- *parents* = # of parent cells in file

These are the recognized section keywords for the body of the file (only one for this file).

- *Parents*

The *Parents* section consists of N consecutive entries, where N = # of parents, each of this form:

```
index parent-ID Nx Ny Nz
```

The index is ignored; it is only added to assist in examining the file. Typically, the indices should run consecutively from 1 to N.

The parent-ID is a string of numbers (one per level) separated by dashes, e.g. 12-352-65, where level 1 is the coarsest grid overlaying the simulation domain, level 2 is the refined grid within a level 1 cell, etc.

The first number in the ID string is which level 1 cell (from 1 to N1) this parent cell descends from, the second number is which level 2 cell (from 1 to N2) this parent cell descends from, etc. The final number is which cell this cell is within its own parent.

As an example, consider the parent ID 12-352-65. Assume the simulation box was partitioned with a 10x10x10 level 1 grid, or 1000 level 1 grid cells. These are numbered from 1 to 1000, with x varying fastest, then y, finally z. The parent cell with ID 12-352-65 is inside the 12th of those level 1 cells. If that cell were sub-divided into 8x6x10 cells, there would be 480 level 2 cells within the 12th level 1 cell. The parent cell with ID 12-352-65 is inside the 352nd of those level 2 cells. Likewise it is within the 65th of the level 3 cells inside the 352nd level 2 cell.

The Nx, Ny, Nz values determine how the parent cell is sub-divided into Nx by Ny by Nz cells at the next level. Each of those cells could be a child cell or yet another parent cell. Nz must be specified as 1 for 2d grids.

For example, this entry:

```
index 12-352-65 2 2 2
```

means the parent cell 12-352-65 at level 3 is further sub-divided into 2x2x2 level 4 cells. The IDs of the 8 new cells will be 12-352-65-1, 12-352-65-2, . . . , 12-352-65-8.

The lines in the *Parents* section must be ordered such that no parent cell is listed before its own parent cell appears. A simple way to insure this is to list the single level 0 cell first, all level 1 parent cells next, then level 2 parent cells, etc.

The parent cell with ID = 0 is a special case. It can be thought of as the "root" cell, or the single level 0 cell, which represents the entire simulation domain. Its specification in the grid file defines the level 1 grid that overlays the simulation domain. Thus the first line of the *Parents* section should be formatted something like this:

```
1 0 10 10 20
```

which means the level 1 grid has 10x10x20 cells.

Restrictions:

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

To read gzipped grid files, you must compile SPARTA with the -DSPARTA_GZIP option - see *Section 2.2* of the manual for details.

The hierarchical grid used by SPARTA is encoded in a 32-bit or 64-bit integer ID. The precision is set by the -DSPARTA_BIG or -DSPARTA_SMALL or -DSPARTA_BIGBIG compiler switch, as described in Section 2.2. The number of grid levels that can be used depends on the resolution of the grid at each level. For a minimal refinement of 2x2x2, a level uses 4 bits of the integer ID. Thus a maximum of 7 levels can be used for 32-bit IDs and 15 levels for 64-bit IDs.

Related commands:

create_box command, *create_grid command*

Default:

none

1.14.89 read_isurf command

Syntax:

```
read_isurf group-ID Nx Ny Nz filename thresh ablateID keyword args ...
```

- group-ID = group ID for which grid cells to perform calculation on
- Nx,Ny,Nz = grid cell extent for adding implicit surfs
- filename = binary file with grid corner point values
- thresh = threshold for surface definition, value > 0.0 and < 255.0
- ablateID = ID of a *fix ablate command*
- zero or more keyword/args pairs may be appended

- keyword = *group* or *type* or *push* or *precision* or *read*

```
group arg = group-ID
  group-ID = new or existing surface group to assign the surface elements to
type arg = tfile
  tfile = binary file with per grid cell surface type values
push arg = yes or no = whether to push corner point values to 0/255
precision arg = int or double
read arg = serial or parallel
```

Examples:

```
read_isurf portion 100 100 1 isurf.material.2d 180.5 group mesh
read_isurf subset 150 100 50 isurf.materials.3d 120.5 type isurf.type
read_isurf subset 150 100 50 isurf.materials.3d 120.5 read parallel
```

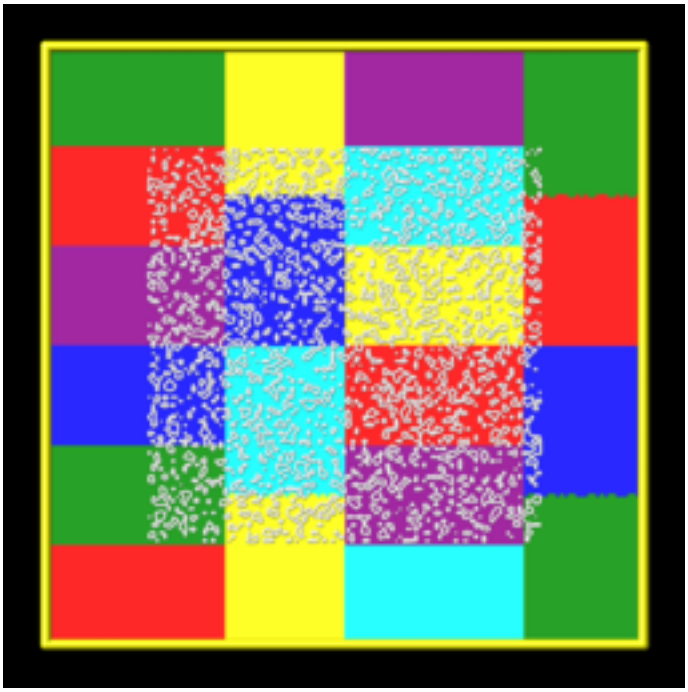
Description:

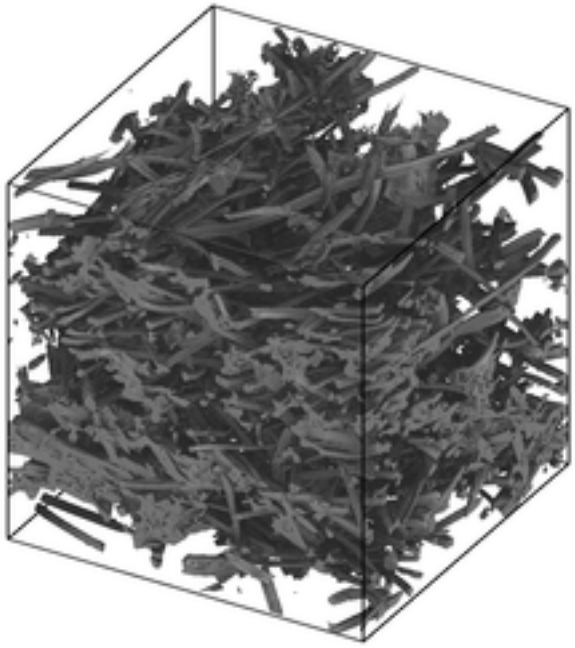
Read the geometry of a surface from the specified file. In SPARTA, a "surface" is a collection of surface elements that represent the surface of one or more physical objects which will be embedded in the global simulation box. Surfaces can be explicit or implicit.

This command reads implicit surfaces from a file containing grid corner point values which implicitly define the surface elements. See the *read_surf command* to read explicit surfaces from a different kind of file. See the *Surface elements: explicit, implicit, distributed* section of the manual for an explantion of explicit versus implicit surfaces as well as distributed versus non-distributed storage. You cannot mix explicit and implicit surfaces in the same simulation.

Surface elements are triangles in 3d or line segments in 2d. Surface elements for each physical object are required to be a complete, connected set that tile the entire surface of the object. See the discussion of watertight surfaces below. Implicit surfaces will always be watertight, due to the algorithm that defines them.

Here are simulation snapshots of 2d and 3d implicit surface models through which particles could flow. Click on either image for a larger image. In the 2d case, the colorings are by processor for sub-domains each owns. The implicit triangles for the 3d case were created via Marching Cubes (discussed below) from a tomographic image of a sample of NASA FiberForm (TM) material, used as a heat shield material on spacecraft.





Particles collide with surface elements as they advect. Each surface element is assigned to a collision model, specified by the *surf_collide command* which affects how a particle bounces off the surface. Each surface element can optionally be assigned to a reaction model, specified by the *surf_react command* which determines if any surface chemistry occurs during a collision. Statistics for each surface element due to their interactions with particles can be tallied via the *compute isurf/grid command*, time-averaged via the *fix ave/grid command*, and ouput via the *dump surface* command.

Surface elememts can be assigned to surface groups via the *group surf* command. Surface group IDs are used by other commands to operate on selected sets of elements. This command has a *type* keyword which can be used to help assign different elements to different groups.

Note that at some point, it will be possible to use the read_isurf command multiple times to read surfaces from multiple files and add them to the simulation domain, so long as the grid extent of the different commands does not overlap. However currently, that is not yet possible.

The format of a surface file for implicit surfaces is discussed below.

The tools directory contains a *implicit_grid.py tool* which can create implicit surface files in a randomized manner for different grid extents.

The specified *group-ID* must be the name of a grid cell group, as defined by the *group grid* command, which contains a set of grid cells, all of which are the same size, and which comprise a contiguous 3d array, with specified extent *Nx* by *Ny* by *Nz*. For 2d simulations, *Nz* must be specified as 1, and the group must comprise a 2d array of cells that is *Nx* by *Ny*. These are the grid cells in which implicit surfaces will be created.

The specified *filename* is for a binary file in the following format:

- first 4 bytes = Nxfile (integer)
- next 4 bytes = Nyfile (integer)
- next 4 bytes = Nzfile (integer), only for 3d simulations
- final N bytes = Nxfile by Nyfile by Nzfile grid corner point values (integer)

For 2d simulations, the first 8 bytes store 2 integers in binary format: Nxfile and Nyfile. For 3d simulations, the first 12 bytes store 3 integers in binary format: Nxfile, Nyfile, and Nzfile. These are the dimensions of the grid of corner point values in the remainder of the file.

Important: The Nxfile, Nyfile, Nzfile values are for a 2d or 3d grid of corner points, which overlay the Nx by Ny by Nz grid of cells. In each dimension there is one more corner point than cells. Thus Nxfile = Nx+1, Nyfile = Ny+1, Nzfile = Nz+1 is required. SPARTA will give an error if the read_isurf Nx,Ny,Nz arguments do not match the first 2 or 3 integers in the file.

The remaining N bytes of the file are a series of corner point values. There are N = Nxfile * Nyfile values in 2d, and N = Nxfile * Nyfile * Nzfile values in 3d.

If the *precision* keyword is set to *int*, which is the default, then the values are one-byte integers, from 0 to 255 inclusive. If the *precision* keyword is set to *double*, then they are double-precision floating point values, from 0.0 to 255.0 inclusive. The one-byte integer format is what is typically used for tomographic images. The double-precision format is what is written by the *write_isurf command*. The latter is typically used when running an ablation model via the *fix ablate command*, where material is removed incrementally (from the corner point values) due to collisions of particles with the implicit surfaces.

Important: The corner point values are a 2d or 3d regular array which must be ordered as follows. The x indices (1 to Nxfile) vary fastest, then the y indices (1 to Nyfile), and the z indices slowest (1 to Nzfile). These will be assigned as corner points to each child grid cell in the Nx by Ny by Nz simulation domain. For mapping corner points to grid cells, the ordering of the regular array of grid cells in the simulation domain is the same: their x indices vary fastest, then y, and their z indices very slowest.

The 8 corner point values (4 in 2d) for each grid cell are used with a marching cubes algorithm (marching squares in 2d) to infer a set of triangles (line segments in 2d) which are created in the grid cell.

Important: All triangles (line segments in 2d) created within the same grid cell are assigned the same surface ID, which is the grid cell ID.

A good description of the two algorithms is given on these Wikipedia webpages:

- https://en.wikipedia.org/wiki/Marching_cubes
- https://en.wikipedia.org/wiki/Marching_squares

The algorithms require a threshold value as input, which is the *thresh* value in the read_isurf command. For corner point values that bracket the threshold, it determines precisely where in the grid cell the corner points of the inferred implicit surface(s) will be.

The threshold must be specified as a floating point value such that 0 < thresh < 255. An integer value for thresh (e.g. 128 or 128.0) is not allowed, because that could induce implicit surfaces with zero length (2d line) or area (3d triangle).

Important: The aggregate set of implicit surfaces created by this procedure must represent a watertight object(s), the same as explained for the *read_surf command*, otherwise SPARTA will generate an error. The marching cube and square algorithms guarantee this. However, if the Nx by Ny by Nz array of grid cells is interior to the simulation box, the entire outer boundary of the Nxfile by Nyfile by Nzfile grid of corner points should have values = 0. This will insure no surface element touches the outer boundary (which would induce a non-watertight surface). If the array of grid cells touches the simulation box face, then this is not a requirement (the same as if a set of explicit surfs were clipped at the box boundary). However, if a boundary is periodic in a particular dimension and the array of grid cells touches that boundary, then you must insure the Nxfile by Nyfile by Nzfile grid of corner points spans that entire dimension, and its values are periodic in the same sense the simulation box is. E.g. if the y dimension is periodic, then

the corner point values at the y = 1 and y = Nyfile lines or planes of the 2d or 3d corner point array must be identical. Otherwise the aggregate set of implicit surfaces will not be consistent across the y periodic boundary.

The specified *ablateID* is the fix ID of a *fix ablate command* which has been previously specified in the input script. It stores the grid corner point values for each grid cell. It also has the code logic for converting grid corner point values to surface elements (line segments or triangles) and also optinally allows for the surface to be ablated during a simulation due to particles colliding with the surface elements.

The following optional keywords affect attributes of the read-in surface elements and how they are read.

Surface groups are collections of surface elements. Each surface element belongs to one or more surface groups; all elements belong to the "all" group, which is created by default. Surface group IDs are used by other commands to identify a group of suface elements to operate on. See the *group surf* command for more details.

Every surface element also stores a *type* which is a positive integer. *Type* values are useful for flagging subsets of elements. For example, implicit surface elemnts in different regions of the simulation box. Surface element types can be used to define surface groups. See the *group surf* command for details.

The *group* keyword specifies an extra surface *group-ID* to which all the implicit surface elements are assigned when created by the read-in corner points. All the created implicit elements are also assigned to the "all" group and to *group-ID*. If *group-ID* does not exist, a new surface group is created. If it does exist the create implicit surface elements are added to that group.

The *type* keyword triggers the reading of a per grid cell type file with the specified name *tfile*.

The specified *filename* is for a binary file in the following format:

- first 4 bytes = Nxfile (integer)
- next 4 bytes = Nyfile (integer)
- next 4 bytes = Nzfile (integer), only for 3d simulations
- final N bytes = Nxfile by Nyfile by Nzfile grid corner point values (integer)

For 2d simulations, the first 8 bytes store 2 integers in binary format: Nxfile and Nyfile. For 3d simulations, the first 12 bytes store 3 integers in binary format: Nxfile, Nyfile, and Nzfile. These are the dimensions of the grid of corner point values in the remainder of the file.

Important: The Nxfile, Nyfile, Nzfile values are for a 2d or 3d grid of per-cell values, which overlay the Nx by Ny by Nz grid of cells. Thus Nxfile = Nx, Nyfile = Ny, Nzfile = Nz is required. SPARTA will give an error if the read_isurf Nx,Ny,Nz arguments do not match the first 2 or 3 integers in the file.

The remaining N bytes of the file are a series of one-byte integer values. There are N = Nxfile * Nyfile values in 2d, and N = Nxfile * Nyfile * Nzfile values in 3d. Each value is a single byte integer from 1 to 255 inclusive, since surface element type values must be > 0.

Important: The corner point values are a 2d or 3d regular array which must be ordered as follows. The x indices (1 to Nxfile) vary fastest, then the y indices (1 to Nyfile), and the z indices slowest (1 to Nzfile). These will be assigned to each grid cell in the Nx by Ny by Nz simulation domain. For mapping type values to grid cells, the ordering of the regular array of grid cells in the simulation domain is the same: their x indices vary fastest, then y, and their z indices very slowest.

The type value for each grid cell is used to assign a type value to each surface element created in that grid cell by the marching cubes or squares algorithm.

The *push* keyword specifies whether or not (*yes* or *no*) to "push" grid corner points values to their minimum/maximum possible values, i.e. 0 or 255 respectively. Each corner point value which is below (above) the specified *thresh* value is and is also entirely surrounded by neighbor corner point values which are also below (above) the *thresh* value is reset to 0 (255). In 2d, there are 8 corner points surrouding each interior corner point, i.e. all corner points on the face of the 2x2 set of grid cells which surround the interior point. In 3d, there are 26 corner points surrouding each interior corner point, i.e. all corner points on the face of the 2x2x2 set of grid cells which surround the interior point. The purpose of this operation is to reset corner point values to 0 if they are fully exterior to the surface object(s), and likewise to 255 if they are fully interior to the surface object(s).

Note that the push is a one-time operation, performed when the corner point values are read in, before the first set of surface elements are created by the marching cubes or marching squares algorithms.

The default for the *push* keyword is *yes*.

The *read* keyword specifies how the input file of grid corner point values is read. If the value is *serial*, which is the default, then only a single proc reads the file, a chunk of values at at time. They are broadcast to other processors, and each scans them for corner point values that correspond to grid cells it owns. If the value is *parallel*, then each proc opens the input file and reads a N/P portion of the corner point values, where N is the # of corner point values, and P is the # of procs. Additional communication is then performed to communicate the corner point values where they are needed by each grid cell that owns one of the corner point values. The *parallel* option can be faster for simulations with large grid corner point files and large numbers of processors.

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*. If particles already exist in the simulation, you must insure particles do not end up inside the set of implicit surfaces.

Related commands:

read_surf command write_surf command fix ablate command

Default:

The optional keyword defaults are group = all, type = no, push = yes, precision int, and read serial.

1.14.90 read_particles command

Syntax:

```
read_particles file Nstep
```

- file = dump file to read snapshot from
- Nstep = timestep to read

Examples:

```
read_particles dump.sphere 10500
```

Description:

Read a snapshot of particles from a previously created dump file and add them to the simulation domain. This is a means of reading in particles from a previous SPARTA simulation or created as output by another code. The *create_particles*, *fix emit/face*, and *read_restart* commands are alternate ways to generate particles for a simulation.

The dump file must be in the SPARTA format created by the *dump particles* command which is described on its doc page.

Currently, each line of particle data in the file must have 8 fields in the following order. At some point we may generalize this format.

```
id, type, x, y, z, vx, vy, vz
```

The *id* is any positive integer, which can simply be set to values from 1 to Nparticles if desired. The type is the species ID from 1 to Nspecies. The value corresponds to the order in which species are defined in the current input script via the *species* command. The x,y,z values are the particle coordinates which must be inside (or on the surface of) the simulation box. If a particle is outside the box it will be skipped when the file is read. For 2d or axisymmetric simulations z = 0.0 should be used, though SPARTA does not check for this. The vx,vy,vz values are the particle velocity. The rotational and vibrational energies for the new particles are set to 0.0.

When the reading of particles is complete, the number of particles read is printed to the screen. If the number is smaller than the particles in the file, it is because some were outside the simulation box.

A check is made for any particle inside a surface object which triggers an error. However the check is only for grid cells entirely inside a surface object. Particles in grid cells which are cut by surfaces are not checked. It is your responsibility to insure particles close to surfaces are actually outside the surface object. If this is not the case, errors may be triggered once particles begin to move.

Restrictions:

none

Related commands:

create_particles command, *fix emit/face command*

Default:

none

1.14.91 read_restart command

Syntax:

```
read_restart file keyword args ...
```

- file = name of binary restart file to read in
- zero or one keyword/args pair may be listed
  - keywords = gridcut or balance

* gridcut arg = cutoff

  cutoff = acquire ghost cells up to this far away (distance units)

* balance args = same as for balance_grid command

Examples:

```
read_restart save.10000
read_restart restart.*
read_restart flow.*.%
read_restart save.10000 gridcut -1.0
read_restart save.10000 balance rcb cell
```

Description:

Read in a previously saved simulation from a restart file. This allows continuation of a previous run on the same or different number of processors. Information about what is stored in a restart file is given below. Basically this operation will re-create the simulation box with all its particles, the hierarchical grid used to track particles, and surface elements embedded in the grid, all with their attributes at the point in time the information was written to the restart file by a previous simluation.

Although restart files are saved in binary format to allow exact regeneration of information, the random numbers used in the continued run will not be identical to those used if the run had been continued. Hence the new run will not be identical to the continued original run, but should be statistically similar.

Important: Because restart files are binary, they may not be portable to other machines. SPARTA will print an error message if it cannot read a restart file for this reason.

If a restarted run is performed on the same number of processors as the original run, then the assignment of grid cells (and their particles) to processors will be the same as in the original simulation. If the processor count changes, then the assignment will necessarily be different. In particular, even if the original assignment was "clumped", meaning each processor's cells were geometrically compact, the new assignment will not be clumped; it will be "dispersed". See *Section 6.8* of the manual for an explanation of clumped and dispersed grid cell assignments and their relative performance trade-offs.

Note that the restart file contains the setting for the *global gridcut* command. If it is >= 0.0 and the assignment of grid cells to processors is "dispersed" (as described in the preceeding paragraph), and there are surface elements defined in the restart file, an error will be triggered. This is because the read_restart command needs to mark all the grid cells as inside vs outside the defined surface and cannot do this without ghost cell information. As explained on the doc page for the *global gridcut* command, ghost cells cannot be setup with gridcut >= 0.0 and "dispersed" grid cells.

The solution is to use one of the two keywords listed above, either *gridcut* or *balance*. The former allows you to reset the grid cutoff to -1.0 so that ghost cells can be setup. Note however that this means each processor will own a copy of all grid cells (at least until you change it later), which may be undesirable or even impossible for large problems if it requires too much memory. The other solution is to use the *balance* keyword to trigger a re-balance of the grid cells to processors as soon as the read_restart command reads them in. The arguments for the *balance* keyword are identical to those for the *balance_grid* command. If you choose a balancing style that results in a "clumped" assignment, then ghost cells will be setup successfully.

Similar to how restart files are written (see the *write_restart* and *restart* commands), the restart filename can contain two wild-card characters. If a "*" appears in the filename, the directory is searched for all filenames that match the pattern where "*" is replaced with a timestep value. The file with the largest timestep value is read in. Thus, this

effectively means, read the latest restart file. It's useful if you want your script to continue a run from where it left off. See the *run* command and its "upto" option for how to specify the run command so it doesn't need to be changed either.

If a "%" character appears in the restart filename, SPARTA expects a set of multiple files to exist. The *restart* and *write_restart* commands explain how such sets are created. Read_restart will first read a filename where "%" is replaced by "base". This file tells SPARTA how many processors created the set and how many files are in it. Read_restart then reads the additional files. For example, if the restart file was specified as save.% when it was written, then read_restart reads the files save.base, save.0, save.1, ... save.P-1, where P is the number of processors that created the restart file.

Note that P could be the total number of processors in the previous simulation, or some subset of those processors, if the *fileper* or *nfile* options were used when the restart file was written; see the *restart* and *write_restart* commands for details. The processors in the current SPARTA simulation share the work of reading these files; each reads a roughly equal subset of the files. The number of processors which created the set can be different than the number of processors in the current SPARTA simulation. This can be a fast mode of input on parallel machines that support parallel I/O.

A restart file stores only the following information about a simulation, as specified by the associated commands:

- *units*
- *dimension*
- *simulation box size* and *boundary conditions*
- *global settings*
- particles with their individual attributes and custom attributes defined by fixes
- *particle species info*
- *mixtures*
- geometry of the hierarchical grid that overlays the simulation domain as *created* or *read from a file*
- geometry of all defined *surface elements*
- *group definitions* for grid cells and surface elements
- current timestep number

No other information is stored in the restart file. Specifically, information about these simulation entities and their associated commands is NOT stored:

- *random number seed*
- *computes*
- *fixes*
- *collision model*
- *chemistry (reaction) model*
- *surface collision models*
- *surface reaction models*
- assignment of surfaces/boundaries to surface models
- *variables*
- *regions*
- output options for *stats*, *dump*, *restart* files

- *timestep size*

This means any information specified in the original input script by these commands needs to be re-specified in the restart input script, assuming the continued simulation needs the information.

Also note that many commands can be used after a restart file is read, to override a setting that was stored in the restart file. For example, the *global* command can be used to reset the values of its specified keywords.

In particular, take note of the following issues:

The status of time-averaging fixes, such as *fix ave/time*, *fix ave/grid*, *fix ave/surf*, does not carry over into the restarted run. E.g. if the *ave running* option is used with those commands in the original script and again specified in the restart script, the running averaged quantities do not persist into the new run.

The *surf_modify* command must be used in the restart script to assign surface collision models, specified by the *surf_collide* command, to all *global boundaries* of type "s", and to any surfaces contained in the restart file, as read in by the *read_surf* command.

If a collision model is specified in the restart script, and the *collide_modify vremax or remain* command is used to enable Vremax and fractional collision count to persist for many timesteps, no information about these quantities persists from the original simulation to the restarted simulation. The initial run in the restart script will re-initialize these data structures.

If a fix is used which defines custom attributes of particles, the vectors or arrays for these attributes are stored in the restart file. See the *fix ambipolar command* as an example; it creates a custom vector called "ionambi" and a custom array called "velambi". However, the restart script must specify the same fix before the first *run* command it uses, so that the same custom attributes are re-created, otherwise the custom attribute info from the restart file will be deleted.

Restrictions:

none

Related commands:

read_grid command, *read_surf command*, *write_restart command*, *restart command*

Default:

none

1.14.92 read_surf command

Syntax:

```
read_surf filename keyword args ...
```

- `filename` = name of surface file
- zero or more keyword/args pairs may be appended

  keyword = `origin` or `trans` or `atrans` or `ftrans` or `scale` or `rotate` or `transparent` or `invert` or `clip` or `group` or `typeadd` or `particle` or `file`

  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*

transparent args args = *none*

invert args args = *none*

clip args = none or fraction fraction = *push points close to the box boundary to the boundary (optional)*

group arg = group-ID group-ID = *new or existing surface group to assign the surface elements to*

typeadd arg = Noffset Noffset = *add Noffset to the type value of each element*

particle args = none or check or keep

- none = *allow no particles in simulation when read surfs (default)*
- check = *delete particles inside surfs or in cells intersected by surfs*
- keep = *keep all particles*

file args = *identical to those defined for the write_surf command* this keyword must be last

Examples:

```
read_surf surf.sphere
read_surf surf.sphere group sphere2 typeadd 1
read_surf surf.file trans 10 5 0 scale 3 3 3 invert clip
read_surf surf.file trans 10 5 0 scale 3 3 3 invert clip 1.0e-6
read_surf surf.file trans 10 5 0 scale 3 3 3 invert clip file tmp.surfs
read_surf surf.file trans 10 5 0 scale 3 3 3 invert clip file tmp.surfs.% points no␣
↪nfile 32
```

Description:

Read the geometry of a surface from the specified file. In SPARTA, a "surface" is a collection of surface elements that represent the surface(s) of one or more physical objects which will be embedded in the global simulation box. Surfaces can be explicit or implicit. This command reads explicit surfaces from a file containing a list of explicit surfaces. See the *read_isurf* command to read implicit surfaces from a different kind of file. See the *Howto 6.13* section of the manual for an explanation of explicit versus implicit surfaces as well as distributed versus non-distributed storage. You cannot mix explicit and implicit surfaces in the same simulation.

Surface elements are triangles in 3d or line segments in 2d. Surface elements for each physical object are required to be a complete, connected set that tile the entire surface of the object. See the discussion of watertight objects below.

Particles collide with surface elements as they advect. Each surface element is assigned to a collision model, specified by the *surf_collide command* which affects how a particle bounces off the surface. Each surface element can optionally be assigned to a reaction model, specified by the *surf_react command* which determines if any surface chemistry occurs during a collision. Statistics for each surface element due to their interactions with particles can be tallied via the *compute surf command*, time-averaged via the *fix ave/surf command*, and ouput via the *dump surface* command.

Surface elements can be assigned to surface groups via the *group surf* command. Surface group IDs are used by other commands to operate on selected sets of elements. This command has *group* and *typeadd* keywords which can be used to help assign different elements or different objects to different groups.

Explicit surface elements can be stored in a distributed fashion (each processor only stores elements which overlap grid cells it owns or has a ghost cell copy of). Or each processor can store a copy of all surface elements (the default). See the *global surfs* command to change this setting.

Note that the read_surf command can be used multiple times to read multiple objects from multiple files and add them to the simulation domain. The format of a surface file for explicit elements is discussed below. Optional keywords allow the vertices in the file to be translated, scaled, and rotated in various ways. This allows a single surface file, e.g. containing a unit sphere, to be used multiple times in a single simulation or in different simulations.

The tools directory contains tools that can create surface files with simple geometric objects (spheres, blocks, etc). It also has tools that can convert surface files in other formats to the SPARTA format for explicit surfaces, e.g. for files created by a mesh-generation program.

If all the surface elements are contained in a single file, the specified file can be a text file or a gzipped text file (detected by a .gz suffix).

If a "%" character appears in the surface filename, SPARTA expects a set of multiple files to exist. The *write_surf command* explains how such sets are created. Read_surf will first read a filename where "%" is replaced by "base". This file tells SPARTA how many total surfaces and files are in the set (i.e. just the header information described below). The read_surf command then reads the additional files. For example, if the surface file was specified as save.% when it was written, then read_surf reads the files save.base, save.0, save.1, ... save.P-1, where P is the number of processors that created the surface file.

Note that P could be the total number of processors in the previous simulation, or some subset of those processors, if the *fileper* or *nfile* options were used when the surface file was written; see the *write_surf command* for details. The processors in the current SPARTA simulation share the work of reading these files; each reads a roughly equal subset of the files. The number of processors which created the set can be different than the number of processors in the current SPARTA simulation. This can be a fast mode of input on parallel machines that support parallel I/O.

Format of a single surface file

The remainder of this section describes the format of a single surface file, whether it is the only file or one of multiple files flagged with a processor number.

A surface file for explicit surfaces 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 one or more 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 surface 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".

- *files* = # of files in set (only for base file, see below)

- *points* = # of points in surface (optional, see below)
- *lines* = # of line segments in surface (only allowed for 2d)
- *triangles* = # of triangles in surface (only allowed for 3d)

The *files* keyword only appears in the "base" file for a set of multiple files indicated by the "%" character in the filename. It tells SPARTA how many additional files exist in the set. A "base" file has no additional sections, i.e. no body.

The *points* keyword is optional (see below). For a set of multiple files, it cannot appear in the "base" file, but only in individual files in the set.

The *points*, *lines*, *triangles* keywords refer to the number of points, lines, triangles in an individual file. Except in the case of a "base" file for a set of multiple files. In that case, the *lines* and *triangles* keywords give the number of lines or triangles in the entire set.

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:

```
index x y z     (for 3d)
index x y       (for 2d)
```

The index value is ignored; it is only added to assist in examining the file. When lines and triangles reference point indices they are simply ordered from 1 to N, regardless of the actual value of the index in the file. *X,y,z* are the coordinates of the point in distance units. Note that for 2d simulations, *z* should be omitted.

Important: Unless points are on the surface of the simulation box, they will be part of multiple lines or triangles. However, there is no requirement that each point appear exactly once in the *Points* list. For example, a point that is the common corner point of M triangles, could appear 1 or 2 or up to M times. However, if the same point appears multiple times in the *Points* list, the coordinates of all copies must be numerically identical, in order for SPARTA to verify the surface is a watertight object, as discussed below.

Important: The *points* keyword and *Points* section are not required. You must either use both or neither. As explained next, an optional format for the *Lines* or *Triangles* sections includes point coordinates directly with each line or triangle.

- The *Lines* section is only allowed for 2d simulations and consists of N entries, where N = # of lines. All entries must be in the same format, either A or B. If a Points section was included, use format A. If it was not, use format B.

```
line-ID (type) p1 p2              # format A
line-ID (type) p1x p1y p2x p2y    # format B
```

The line-ID is stored internally with the line. If the read_surf commmand is reading a single file, the line-IDs should be unique values from 1 to N where N is the number of lines specified in the header of the file. For a set of multiple files, each line in the collection of all files should have a unique ID, and the IDs should range from 1 to N, where N is the number of lines specified in the base file. SPARTA does not check line-IDs for uniqueness. Note that lines in an individual file (single or multiple) do not need to be listed by ID order; they can be in any order.

Important: If the read_surf command is used when lines already exist, i.e. to add new lines, then each line-ID is incremented by Nprevious = the # of lines that already exist.

Type is an optional integer value which must be specified for all or none of the lines in the file. If used, it must be a positive integer value for each line. If not specified, the type of each line is set to 1. Line IDs and types can be used to assign lines to surface groups via the *group surf* command.

For format A, *p1* and *p2* are the indices of the 2 end points of the line segment, as found in the Points section. Each is a value from 1 to the # of points, as described above. For format B, (p1x,p1y) and (p2x,p2y) are the (x,y) coordinates of the two points (1,2) in the line.

The ordering of *p1*, *p2* is important as it defines the direction of the outward normal for the line segment when a particle 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. All entries must be in the same format, either A or B. If a Points section was included, use format A. If it was not, use format B.

```
tri-ID (type) p1 p2 p3                                  # format A
tri-ID (type) p1x p1y p1z p2x p2y p2z p3x p3y p3z        # format B
```

The tri-ID is stored internally with the line. If the read_surf commmand is reading a single file, the tri-IDs should be unique values from 1 to N where N is the number of triangles specified in the header of the file. For a set of multiple files, each triangle in the collection of all files should have a unique ID, and the IDs should range from 1 to N, where N is the number of triangles specified in the base file. SPARTA does not check tri-IDs for uniqueness. Note that triangles in an individual file (single or multiple) do not need to be listed by ID order; they can be in any order.

Important: If the read_surf command is used when triangles already exist, i.e. to add new triangles, then each tri-ID is incremented by Nprevious = the # of triangles that already exist.

Type is an optional integer value which must be specified for all or none of the triangles in the file. If used, it must be a positive integer value for each triangle. If not specified, the type of each triangle is set to 1. Triangle IDs and types can be used to assign triangles to surface groups via the *group surf* command.

For format A, *p1*, *p2*, and *p3* are the indices of the 3 corner points of the triangle, as found in the Points section. Each is a value from 1 to the # of points, as described above. For format B, (p1x,p1y,p1z), (p2x,p2y,p2z), and (p3x,p3y,p3z) are the (x,y,z) coordinates of the three corner points (1,2,3) of the triangle.

The ordering of *p1*, *p2*, *p3* is important as it defines the direction of the outward normal for the triangle when a particle 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.

Optional keywords

The following optional keywords affect the geometry of the read-in surface elements. The geometric transformations they describe 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 an *origin* operation followed by a *rotate* operation may not be the same as a *rotate* operation followed by an *origin* operation.

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 `transparent` keyword flags all the read in surface elements as transparent, meaning particles pass through them. This is useful for tallying flow statistics. The *surf_collide transparent* command must also be used to assign a transparent collision model to those surface elements. The *compute surf* will tally fluxes differently for transparent surf elements. The *Section 6.15* doc page provides an overview of transparent surfaces. See those doc pages for details.

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.

Important: If a surface you clip crosses a periodic boundary, as specified by the *boundary* command, then the clipping that takes place must be consistent on both the low and high end of the box (in the periodic dimension). This means any point on the boundary that is generated by the clip operation should be generated twice, once on the low side of the box and once on the high side. And those two points must be periodic images of each other, as implied by periodicity. If the surface you are reading does not clip in this manner, then SPARTA will likely generate an error about mis-matched or inconsistent cells when it attempts to mark all the grid cells and their corner points as inside vs outside the surface.

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 be created near the box surface due to the clipping operation, depending on the coordinates of the initial unclipped points.

If this is the case, an optional `fraction` argument can be appended to the `clip` keyword. `Fraction` is a unitless value which is converted to a distance `delta` in each dimension where delta = fraction * (boxhi - boxlo). If a point is nearer than delta to the lo or hi boundary in a dimension, the point is moved to be on the boundary, before the clipping operation takes place. This can prevent tiny surface elements from being created due to clipping. If `fraction` is not specified, the default value is 0.0, which means points are not moved. If specified, `fraction` must be a value between 0.0 and 0.5.

Note that the `clip` operation may delete some surface elements and create new ones. Likewise for the points that define the end points or corner points of surface element lines (2d) or triangles (3d). The resulting altered set of surface elements can be written out to a file by the *write_surf* command, which can then be used an input to a new simulation or for post-processing and visualization.

Important: When the `clip` operation deletes or adds surface elements, the line-IDs or tri-IDs will be renumbered to produce IDs that are consecutive values from 1 to the # of surface elements. The ID of a surface element that is unclipped may change due to this reordering.

The following optional keywords affect group and type settings for the read-in surface elements and output of the elements. Also how particles are treated when surface elements are added.

Surface groups are collections of surface elements. Each surface element belongs to one or more surface groups; all elements belong to the "all" group, which is created by default. Surface group IDs are used by other commands to identify a group of suface elements to operate on. See the *group surf* command for more details.

Every surface element also stores a `type` which is a positive integer. `Type` values are useful for flagging subsets of elements or different objects in the surface file. For example, a patch of triangles on a sphere. Or one sphere out of several that the file contains. Surface element types can be used to define surface groups. See the *group surf* command for details.

The `group` keyword specifies an extra surface `group-ID` to assign all the read-in surface elements to. All the read-in elements are assigned to the "all" group and to `group-ID`. If `group-ID` does not exist, a new surface group is created. If it does exist the read-in surface elements are added to that group.

The `typeadd` keyword defines an `Noffset` value which is added to the type of each read-in surface element. The default is Noffset = 0, which means the read-in type values are not altered. If type values are not included in the file, they default to 1 for every element, but can still be altered by the `typeadd` keyword.

Note that use of the `group` and `typeadd` keywords allow the same surface file to be read multiple times (e.g. with different origins, tranlations, rotations, etc) to define multiple objects, and assign their surface elements to different groups or different type values.

The `particle` keyword determines how particles in the simulation are affected by the new surface elements. If the setting is `none`, which is the default, then no particles can exist in the simulation. If the setting is *check*, then particles in grid cells that are inside the new watertight surface object(s) or in grid cells intersected by the new surface elements are deleted. This is to insure no particles will end up inside a surface object, which will typically generate errors when particles move. If the setting is *keep* then no particles are deleted. It is up to you to insure that no particles are inside surface object(s), else an error may occur later. This setting can be useful if a *remove_surf* was used to remove a surface object, and a new object is being read in, and you know the new object is smaller than the one it replaced. E.g. for a model of a shrinking or ablating object.

If the `file` keyword is used, the surfaces will be written out to the specified `filename` immediately after they are read in. The arguments for this keyword are identical to those used for the *write_surf* command. This includes a file name with optional "*" and "%" wildcard characters, as well as its optional keywords.

Important: The `file` keyword must be the last keyword specified with the read_isurf command. This is because all

the remaining arguments are passed to the *write_surf* command.

The format for the output file is the same as the one written by the *write_surf* command, or read by this command. Note that it can be useful to write out a new surface file after reading one if clipping was performed; the new file will contain the surface element altered by clipping and will not contain any surface elements removed by clipping.

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. If particles already exist in the simulation, you must insure particles do not end up inside the added surfaces. See the *particle* keyword for options with regard to particles.

To read gzipped surface files, you must compile SPARTA with the -DSPARTA_GZIP option - see *Section 2.2* of the manual for details.

The *clip* keyword cannot be used when the *global surfs explicit/distributed* command has been used. This is because we have not yet figured out how to clip distributed surfaces.

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 line segments. For a 3d simulation it means that every triangle edge is part of exactly 2 triangles. Exceptions to these rules allow for triangle edges (in 3d) that lie entirely in a global face of the simulation box, or for line points (in 2d) that are on a global edge of the simulation box. This can be the case after clipping, which allows for use of watertight surface object (e.g. a sphere) that is only partially inside the simulation box, but which when clipped to the box becomes non-watertight, e.g. half of a sphere.

Note that this definition of watertight does not require that the surface elements in a file represent a single physical object; multiple objects (e.g. spheres) can be represented, provided each is watertight.

Another restriction on surfaces is that they do not represent an object that is "infinitely thin", so that two sides of the same object lie in the same plane (3d) or on the same line (2d). This will not generate an error when the surface file is read, assuming the watertight rule is followed. However when particles collide with the surface, errors will be generated if a particle hits the "inside" of a surface element before hitting the "outside" of another element. This can occur for infinitely thin surfaces due to numeric round-off.

When running a simulation with multiple objects, read from one or more surface files, you should insure they do not touch or overlap with each other. SPARTA does not check for this, but it will typically lead to unphysical particle dynamics.

Related commands:

read_isurf command, *write_surf command*

Default:

The default origin for the vertices in the surface file is (0,0,0). The defaults for group = all, type = no, toffset = 0, particle = none.

1.14.93 region command

Syntax:

```
region ID style args keyword value ...
```

- ID = user-assigned name for the region
- style = *block* or *cylinder* or *plane* or *sphere* or *union* or *intersect*

```
block args = xlo xhi ylo yhi zlo zhi
  xlo,xhi,ylo,yhi,zlo,zhi = bounds of block in all dimensions (distance units)
cylinder args = dim c1 c2 radius lo hi
  dim = x or y or z = axis of cylinder
  c1,c2 = coords of cylinder axis in other 2 dimensions (distance units)
  radius = cylinder radius (distance units)
  lo,hi = bounds of cylinder in dim (distance units)
plane args = px py pz nx ny nz
  px,py,pz = point on the plane (distance units)
  nx,ny,nz = direction normal to plane (distance units)
sphere args = x y z radius
  x,y,z = center of sphere (distance units)
  radius = radius of sphere (distance units)
union args = N reg-ID1 reg-ID2 ...
  N = # of regions to follow, must be 2 or greater
  reg-ID1,reg-ID2, ... = IDs of regions to join together
intersect args = N reg-ID1 reg-ID2 ...
  N = # of regions to follow, must be 2 or greater
  reg-ID1,reg-ID2, ... = IDs of regions to intersect
```

- zero or more keyword/value pairs may be appended
- keyword = *side*

```
side value = in or out
  in = the region is inside the specified geometry
  out = the region is outside the specified geometry
```

Examples:

```
region 1 block -3.0 5.0 INF 10.0 INF INF
region 2 sphere 0.0 0.0 0.0 5 side out
region void cylinder y 2 3 5 -5.0 INF
region outside union 4 side1 side2 side3 side4
```

Description:

This command defines a geometric region of space. Various other commands use regions. See the *group grid*, *group surf*, and *dump_modify* commands for examples.

Commands which use regions typically test whether a point is contained in the region or not. For this purpose, coordinates exactly on the region boundary are considered to be interior to the region. This means, for example, for a spherical region, a point on the sphere surface would be part of the region if the sphere were defined with the *side in* keyword, but would not be part of the region if it were defined using the *side out* keyword. See more details on the *side* keyword below.

The lo/hi values for the *block* or *cylinder* styles can be specified as INF which means a large negative or positive number (1.0e20).

For style *cylinder*, the c1,c2 params are coordinates in the 2 other dimensions besides the cylinder axis dimension. For dim = x, c1/c2 = y/z; for dim = y, c1/c2 = x/z; for dim = z, c1/c2 = x/y. Thus the third example above specifies a cylinder with its axis in the y-direction located at x = 2.0 and z = 3.0, with a radius of 5.0, and extending in the y-direction from -5.0 to infinity.

The *union* style creates a region consisting of the volume of all the listed regions combined. The *intersect* style creates a region consisting of the volume that is common to all the listed regions.

Important: Regions in SPARTA are always 3d geometric objects, regardless of whether the *dimension* of the simulation 2d or 3d. Thus when using regions in a 2d simulation, for example, you should be careful to define the region so that its intersection with the 2d x-y plane of the simulation has the 2d geometric extent you want.

The *side* keyword determines whether the region is considered to be inside or outside of the specified geometry. Using this keyword in conjunction with *union* and *intersect* regions, complex geometries can be built up. For example, if the interior of two spheres were each defined as regions, and a *union* style with *side* = out was constructed listing the region-IDs of the 2 spheres, the resulting region would be all the volume in the simulation box that was outside both of the spheres.

Restrictions:

none

Related commands:

dump_modify command

Default:

The option default is side = in.

1.14.94 remove_surf command

Syntax:

```
remove_surf surfID
```

- surfID = group ID for which surface elements to remove

Examples:

```
remove_surf topsurf
```

Description:

Remove a group of surface elements that have previously been read-in via the *read_surf* command. The *group surf* or *read_surf* can be used to assign each surface element to one or more groups. This command removes all surface elements in the specified *surfID* group.

Note that the remaining surface elements must still constitute a "watertight" surface or an error will be generated. The definition of watertight is explained in the Restrictions section of the *read_surf* doc page.

After surface elements have been deleted, any surface points that are no longer part of a surface element are also deleted. The remaining surface points and elements are renumbered to create compressed, contiguous lists. The new list of surface elements can be output via the *write_surf* command.

Restrictions:

none

Related commands:

read_surf command

Default:

none

1.14.95 reset_timestep command

Syntax:

```
reset_timestep N
```

- N = timestep number

Examples:

```
reset_timestep 0
reset_timestep 4000000
```

Description:

Set the timestep counter to the specified value. This command normally comes after the timestep has been set by reading a restart file via the *read_restart* command, or a previous simulation advanced the timestep.

The *create_box* command sets the timestep to 0; the *read_restart* command sets the timestep to the value it had when the restart file was written.

Restrictions:

none

This command cannot be used when any fixes are defined that keep track of elapsed time to perform certain kinds of time-dependent operations. Examples are the *fix ave/time*, *fix ave/grid*, and *fix ave/surf* commands. Thus these fixes should be specified after the timestep has been reset.

Resetting the timestep clears flags for *computes* that may have calculated some quantity from a previous run. This means these quantity cannot be accessed by a variable in between runs until a new run is performed. See the *variable* command for more details.

Related commands:

none

Default:

none

1.14.96 restart command

Syntax:

```
restart 0
restart N root keyword value ...
restart N file1 file2 keyword value ...
```

- N = write a restart file every this many timesteps
- N can be a variable (see below)
- root = filename to which timestep # is appended
- file1,file2 = two full filenames, toggle between them when writing file
- zero or more keyword/value pairs may be appended
- keyword = *fileper* or *nfile*

```
fileper arg = Np
  Np = write one file for every this many processors
nfile arg = Nf
  Nf = write this many files, one from each of Nf processors
```

Examples:

```
restart 0
restart 1000 flow.restart
restart 1000 restart.*.equil
restart 10000 flow.%.1 flow.%.2 nfile 10
restart v_mystep flow.restart
```

Description:

Write out a binary restart file every so many timesteps, in either or both of two modes, as a run proceeds. A value of 0 means do not write out any restart files. The two modes are as follows. If one filename is specified, a series of filenames will be created which include the timestep in the filename. If two filenames are specified, only 2 restart files will be created, with those names. SPARTA will toggle between the 2 names as it writes successive restart files.

Note that you can specify the restart command twice, once with a single filename and once with two filenames. This would allow you, for example, to write out archival restart files every 100000 steps using a single filenname, and more frequent temporary restart files every 1000 steps, using two filenames. Using restart 0 will turn off both modes of output.

Similar to *dump* files, the restart filename(s) can contain two wild-card characters.

If a "*" appears in the single 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: flow.restart.1000, flow.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. The optional *fileper* and *nfile* keywords discussed below can alter the number of files written.

Restart files are written on timesteps that are a multiple of N but not on the first timestep of a run or minimization. You can use the *write_restart* command to write a restart file before a run begins. 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.

Instead of a numeric value, N can be specifed as 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 restart file 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() and stride() 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*.

For example, the following commands will write restart files every step from 1100 to 1200, and could be useful for debugging a simulation where something goes wrong at step 1163:

```
variable   s equal stride(1100,1200,1)
restart     v_s tmp.restart
```

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.

The optional *nfile* or *fileper* keywords can be used in conjunction with the "%" wildcard character in the specified restart file name(s). As explained above, the "%" character causes the restart file to be written in pieces, one piece for each of P processors. By default P = the number of processors the simulation is running on. The *nfile* or *fileper* keyword can be used to set P to a smaller value, which can be more efficient when running on a large number of processors.

The *nfile* keyword sets P to the specified Nf value. For example, if Nf = 4, and the simulation is running on 100 processors, 4 files will be written, by processors 0,25,50,75. Each will collect information from itself and the next 24 processors and write it to a restart file.

For the *fileper* keyword, the specified value of Np means write one file for every Np processors. For example, if Np = 4, every 4th processor (0,4,8,12,etc) will collect information from itself and the next 3 processors and write it to a restart file.

Restrictions:

none

Related commands:

write_restart command, *read_restart command*

Default:

```
restart 0
```

1.14.97 run command

Syntax:

```
run N keyword values ...
```

- N = # of timesteps
- zero or more keyword/value pairs may be appended
- keyword = `upto` or `start` or `stop` or `pre` or `post` or `every`
  - *upto value* = none
  - *start value* = N1: timestep at which 1st run started
  - *stop value* = N2: timestep at which last run will end
  - pre value = no or yes
  - post value = no or yes
  - every values = M c1 c2 . . .
    - M = break the run into M-timestep segments and invoke one or more commands between each segment
    - c1,c2,. . . ,cN = one or more SPARTA commands, each enclosed in quotes

      c1 = NULL means no command will be invoked

Examples:

```
run 10000
run 1000000 upto
run 100 start 0 stop 1000
run 1000 pre no post yes
run 100000 start 0 stop 1000000 every 1000 "print 'Temp = $t'"
run 100000 every 1000 NULL
```

Description:

Run or continue a simulation for a specified number of timesteps.

A value of N = 0 is acceptable; only the statistics of the system are computed and printed without taking a timestep.

The `upto` keyword means to perform a run starting at the current timestep up to the specified timestep. E.g. if the current timestep is 10,000 and "run 100000 upto" is used, then an additional 90,000 timesteps will be run. This can be useful for very long runs on a machine that allocates chunks of time and terminate your job when time is exceeded. If you need to restart your script multiple times (reading in the last restart file), you can keep restarting your script with the same run command until the simulation finally completes.

The `start` or `stop` keywords can be used if multiple runs are being performed and you want a *variable* or *fix* command that changes some value over time (e.g. target temperature) to make the change across the entire set of runs and not just a single run.

For example, consider these commands followed by 10 run commands:

```
variable     myTemp equal ramp(300,500)
surf_collide 1 diffuse v_myTemp 0.5
run      1000 start 0 stop 10000
run      1000 start 0 stop 10000
...
run      1000 start 0 stop 10000
```

The `ramp()` function in the *variable* and its use in the "surf_collide" command will ramp the target temperature from 300 to 500 during a run. If the run commands did not have the start/stop keywords (just "run 1000"), then the temperature would ramp from 300 to 500 during the 1000 steps of each run. With the start/stop keywords, the ramping takes place smoothly over the 10000 steps of all the runs together.

The `pre` and `post` keywords can be used to streamline the setup, clean-up, and associated output to the screen that happens before and after a run. This can be useful if you wish to do many short runs in succession (e.g. SPARTA is being called as a library which is doing other computations between successive short SPARTA runs).

By default (pre and post = yes), SPARTA zeroes statistical counts before every run and initializes other *fixes* and *computes* as needed. And after every run it gathers and prints timings statistics. If a run is just a continuation of a previous run (i.e. no settings are changed), the initial computation is not necessary. So if *pre* is specified as "no" then the initial setup is skipped, except for printing statistical info. Note that if *pre* is set to "no" for the very 1st run SPARTA performs, then it is overridden, since the initial setup computations must be done.

Important: If your input script changes settings between 2 runs (e.g. adds a *fix* or *compute*), then the initial setup must be performed. SPARTA does not check for this, but it would be an error to use the *pre no* option in this case.

If `post` is specified as "no", the full timing and statistical output is skipped; only a one-line summary timing is printed.

The `every` keyword provides a means of breaking a SPARTA run into a series of shorter runs. Optionally, one or more SPARTA commands (c1, c2,..., cN) will be executed in between the short runs. If used, the `every` keyword must be the last keyword, since it has a variable number of arguments. Each of the trailing arguments is a single SPARTA command, and each command should be enclosed in quotes, so that the entire command will be treated as a

single argument. This will also prevent any variables in the command from being evaluated until it is executed multiple times during the run. Note that if a command itself needs one of its arguments quoted (e.g. the *print* command), then you can use a combination of single and double quotes, as in the example above or below.

The `every` keyword is a means to avoid listing a long series of runs and interleaving commands in your input script. For example, a *print* command could be invoked or a *fix* could be redefined, e.g. to reset a load balancing parameter. Or this could be useful for invoking a command you have added to SPARTA that wraps some other code (e.g. as a library) to perform a computation periodically during a long SPARTA run. See *Section 8* of the manual for info about how to add new commands to SPARTA. See *Section 6.7* of the manual for ideas about how to couple SPARTA to other codes.

With the `every` option, N total steps are simulated, in shorter runs of M steps each. After each M-length run, the specified commands are invoked. If only a single command is specified as NULL, then no command is invoked. Thus these lines:

```
compute t temp
variable myT equal c_t
run 6000 every 2000 "print 'Temp = $myT'"
```

are the equivalent of:

```
compute t temp
variable myT equal c_t
run 2000
print "Temp = $myT"
run 2000
print "Temp = $myT"
run 2000
print "Temp = $myT"
```

which does 3 runs of 2000 steps and prints the x-coordinate of a particular atom between runs. Note that the variable "$q" will be evaluated afresh each time the print command is executed.

Note that by using the line continuation character "&", the run every command can be spread across many lines, though it is still a single command:

```
run 100000 every 1000 &
  "print 'Minimum value = $a'" &
  "print 'Maximum value = $b'" &
  "print 'Temp = $c'"
```

If the *pre* and *post* options are set to "no" when used with the *every* keyword, then the 1st run will do the full setup and the last run will print the full timing summary, but these operations will be skipped for intermediate runs.

IMPORTANT NOTE: You might hope to specify a command that exits the run by jumping out of the loop, e.g.

```
compute t temp
variable T equal c_t
run 10000 every 100 "if '$T < 300.0' then 'jump SELF afterrun'"
```

Unfortunately this will not currently work. The run command simply executes each command one at a time each time it pauses, then continues the run. You can replace the jump command with a simple *quit* command and cause SPARTA to exit during the middle of a run when the condition is met.

Restrictions:

The number of specified timesteps N must fit in a signed 32-bit integer, so you are limited to slightly more than 2 billion steps (2^31) in a single run. However, you can perform successive runs to run a simulation for any number of

steps (ok, up to 2^63 steps).

Related commands:

none

Default:

The option defaults are start = the current timestep, stop = current timestep + N, pre = yes, and post = yes.

1.14.98 scale_particles command

Syntax:

```
scale_particles mix-ID factor
```

- mix-ID = ID of mixture to use when scaling particles
- factor = scale factor

Examples:

```
scale_particles air 0.5
scale_particles air 4.0
```

Description:

Scale the number of particles in the simulation by cloning or deleting individual particles. This can be useful between runs, or after reading a restart file, to increase or decrease the particle count before a new *run* command is issued, as if the *global fnum* value had been changed. For example, an initial coarse simulation can be performed, followed by a simulation at higher resolution.

Only particles of species in the specified mixture are considered for cloning/deleting. See the *mixture* command for how it defines a collection of species.

The specified *factor* can be any value >= 0.0.

If *factor* < 1.0, then for each particle, a random number *R* is generated. If R > factor, the particle is deleted.

If *factor* > 1.0, then for each particle additional particles may be created, by cloning all attributes of the original particle, except for a new random particle ID assigned to each new particle. E.g. if *factor* = 3.4, then two extra particles are created, and a 3rd is created with probability 0.4.

Restrictions:

none

Related commands:

create_particles command

Default:

none

1.14.99 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. This generator is used by SPARTA to initialize auxiliary random number generators, which in turn are used for all operations in the code requiring random numbers. This means you can effectively run a statistically-independent simulation by simply changing this single seed.

The various random number generators used in SPARTA are portable, which means they produce the same random number streams on any machine.

This command is required to perform a SPARTA simulation.

Restrictions:

none

Related commands:

none

Default:

none

1.14.100 shell command

Syntax:

```
shell cmd args
```

- cmd = *cd* or *mkdir* or *mv* or *rm* or *rmdir* or *putenv* 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
putenv args = var1=value1 var2=value2
  var=value = one of more definitions of environment variables
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.sparta hold/log.1
shell rm TMP/file1 TMP/file2
shell putenv SPARTA_DATA=../../data
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.

The *putenv* cmd defines or updates an environment variable directly. Since this command does not pass through the shell, no shell variable expansion or globbing is performed, only the usual substitution for SPARTA variables defined with the *variable* command is performed. The resulting string is then used literally.

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

1.14.101 species command

Syntax:

```
species file ID1 ID2 ... keyword value ...
```

- file = filename with species info
- ID1, ID2, . . . = one or more species names listed in file
- multi-species abbreviations can also be used (see below)
- zero or more keyword/value pairs may be appended
- keyword = *vibfile*

```
vibfile value = vfile = filename for extra vibrational info
```

Examples:

```
species air.species air
species ar.species Ar
species air.species air CO2 CO vibfile co2.species.vib
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 the list of species that the simulation recognizes.

The specified *file* is the name of a file containing definitions for a list of species, not all of which need to specified in this command, or used in a simulation. Only those requested by ID will be extracted from the file and they must be present in the file. The format of the species file is discussed below. The data directory in the SPARTA distribution contains several species files, all with the suffix ".species".

Each *ID* is a character string used to identify the species, such as N or O2 or NO or D or Fe-. The string can be any combination of alphanumeric characters or "+", "-", or underscore.

Instead of specifying IDs for single species, one of several pre-defined multi-species names can be used, each of which is expanded into a list of several individual species IDs. The list of currently recognized abbreviations is as follows:

- air = N, O, NO

These abbreviations can be used in combination with single-species IDs as in the 3rd example above.

The format of a 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 the following format with values separated by whitespace:

```
species-ID prop1 prop2 ... prop9 prop10
```

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

- prop1 = molecular weight (atomic mass units, e.g. 16 for oxygen)
- prop2 = molecular mass (mass units)
- prop3 = rotational degrees of freedom (integer, unitless)
- prop4 = inverse rotational relaxtion number (unitless)
- prop5 = vibrational degrees of freedom (integer, unitless)
- prop6 = inverse vibrational relaxation number (unitless)
- prop7 = vibrational temperature (temperature units)
- prop8 = species weight (unitless)
- prop9 = multiple of electon charge (1 for a proton)

The allowed values for rotational degrees of freedom (rotdof = prop3) are 0,2,3. Typically, 0 = monatomic species, 2 = diatomic, 3 = anything else.

The allowed values for vibrational degrees of freedom (vibdof = prop5) are 0,2,4,6,8. The associated number of vibrational modes is vibdof divided by 2. Typically, 0 modes = monatomic species, 1 mode = diatomic, 2/3/4 modes = triatomic or larger molecules.

Note that all the listed rotational and vibrational values must be specified for each species, but in cases where they are not used by SPARTA, they can simply be specified as 0.0. Whether or not the values are used for a species depends on the value of rotdof and vibdof. Whether the values are used in a simulation also depends on the settings specified for the *rotation* and *vibration* keywords of the *collide_modify* command.

Specifically, if prop3 for rotdof = 0, then prop4 is ignored. If prop5 for vibdof = 0, then prop6 and prop7 are ignored.

If vibdof = 4,6,8, then information for 2,3,4 vibrational modes can be specified for the species in a separate file using the optional *vibfile* keyword, as discussed below. If the *collide_modify vibration* command is used with a setting of *discrete*, then this vibrational mode info must be specified for each species with a vibdof = 4,6,8. Note that the *fix vibmode* command must also be used to allocate per-particle storage for these additional modes.

The optional *vibfile* keyword can be used to specify additional vibrational information in the specified *vfile*. If this option is used, then an entry must appear in *vfile* for every species in this command with a vibdof value = 4,6,8. Note that even if this vibrational info is read, it is ignored by SPARTA unless the *collide_modify vibrate* setting is specified as *discrete*.

The format of a species vibrational file is as follows. See data/co2.species.vib for an example. Comments or blank lines are allowed in the file. Comment lines start with a "#" character. All other lines must have the following format with values separated by whitespace:

```
species-ID N temp1 relax1 degen1 temp2 relax2 degen2 ... tempN relaxN degenN
```

The species-ID is a string that will be matched against the requested species-ID, as described above. N is the number of vibrational modes that follow, which must be either 2,3,4, and must match the corresponding vibdof value = 4,6,8 (divided by two) used in the species file.

For each of the N modes, 3 values are listed:

- tempI = vibrational temperature of mode I (temperature units)
- relaxI = inverse vibrational relaxation number of mode I (unitless)
- degenI = degeneracy of mode I (integer, unitless)

These quantities are used during collisions if vibrational energy is modeled in discrete levels.

Note that the values for temp1 and relax1 override the same values defined in the species file (prop7 and prop6) when they are listed for the same species in the *vibfile*.

Restrictions:

none

Related commands:

none

Default:

none

1.14.102 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 run. 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 command, *stats_modify command*

Default:

```
stats 0
```

1.14.103 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 = line string, int string, float string, M string, or none
  string = C-style format string
  M = integer from 1 to N, where N = # of quantities being output
every value = v_name
  v_name = an equal-style variable name
```

Examples:

```
stats_modify flush yes
stats_modify format 3 %15.8g
stas_modify format line "%ld %g %g %15.8g"
```

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 can be used to change the default numeric format of any of quantities the *stats_style* command outputs. All the specified format strings are C-style formats, e.g. as used by the C/C++ printf() command. The *line* keyword takes a single argument which is the format string for the entire line of stats output, with N fields, which you must enclose in quotes if it is more than one field. The *int* and *float* keywords take a single format argument and are applied to all integer or floating-point quantities output. The setting for *M string* also takes a single format argument which is used for the Mth value output in each line, e.g. the 5th column is output in high precision for "format 5 %20.15g".

The *format* keyword can be used multiple times. The precedence is that for each value in a line of output, the *M* format (if specified) is used, else the *int* or *float* setting (if specified) is used, else the *line* setting (if specified) for that value is used, else the default setting is used. A setting of *none* clears all previous settings, reverting all values to their default format.

Note: The stats output values *step* and *atoms* are stored internally as 8-byte signed integers, rather than the usual 4-byte signed integers. When specifying the *format int* option you can use a "%d"-style format identifier in the format string and SPARTA will convert this to the corresponding 8-byte form when it is applied to those keywords. However, when specifying the *line* option or *format M string* option for *step* and *natoms*, you should specify a format 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 the timesteps on which 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 variable setting of the stats_modify every command will be overridden.

Restrictions:

none

Related commands:

stats command, *stats_style command*

Default:

The option defaults are flush = no, format int = "%8d", format float = "%12.8g", and every = non-variable setting provided by the *stats* command.

1.14.104 stats_style command

Syntax:

```
stats_style arg1 arg2 ...
```

- arg1,arg2,... = list of keywords

```
possible keywords = step, elapsed, elaplong, dt, cpu, tpcpu, spcpu, wall,
                    np, npave, ntouch, ntouchave, ncomm, ncommave,
                    nbound, nboundave, nexit, nexitave,
            nscoll, nscollave, nscheck, nscheckave,
                    ncoll, ncollave, nattempt, nattemptave,
                    nreact, nreactave, nsreact, nsreactave,
                    nparent, nchild, nsplit, maxlevel,
            vol, lx, ly, lz,
            xlo, xhi, ylo, yhi, zlo, zhi,
            s_ID[I], r_ID[I],
            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
  - elaplong = timesteps since start of initial run in a series of runs
  - dt = timestep size
  - cpu = elapsed CPU time in seconds within a run
  - tpcpu = time per CPU second
  - spcpu = timesteps per CPU second
  - wall = wallclock time in seconds
  - np,npave = # of particles (this step, per-step)
  - ntouch,ntouchave = # of cell touches by particles (this step, per-step)
  - ncomm,ncommave = # of particles 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 particle/particle collisions (this step, per-step)
- nattempt,nattemptave = # of attempted collisions (this step, per-step)
- nreact,nreactave = # of chemical reactions (this step, per-step)
- nsreact,nsreactave = # of chemical reactions on surfs and boundaries (this step, per-step)
- nparent,nchild,nsplit = # of parent, child, split cells
- maxlevel = max # of grid refinement levels
- vol = volume of simulation box
- lx,ly,lz = simulation box lengths
- xlo,xhi,ylo,yhi,zlo,zhi = box boundaries,
- s_ID[I] = Ith component of global vector calculated by a surface collision model with ID
- r_ID[I] = Ith component of global vector calculated by a surface reaction model with ID
- 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, I can include wildcard (see below)
- 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, I can include wildcard (see below)
- 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 np
stats_style step cpu spcpu np xlo xhi c_myCount[2]
stats_style step cpu spcpu np xlo xhi c_myCount[*]
```

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. *Elapsed* is the number of timesteps elapsed since the beginning of this run. *Elaplong* is the number of timesteps elapsed since the beginning of an initial run in a series of runs. See the *start* and *stop* keywords for the *run* command for info on how to invoke a series of runs that keep track of an initial starting time. If these keywords are not used, then *elapsed* and *elaplong* are the same value.

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 *wall* keyword is elapsed time in seconds since SPARTA was launched. This can be used to time portions of the input script in the following manner:

```
variable            t equal wall
variable            t1 equal $t
portion of input script
variable            t2 equal $t
variable            delta equal v_2-v_1
print               "Delta time = $delta"
```

The *np*, *ntouch*, *ncomm*, *nbound*, *nexit*, *nscoll*, *nscheck*, *ncoll*, *nattempt*, *nreact*, and *nsreact* keywords all generate counts for the current timestep.

The *npave*, *ntouchave*, *ncommave*, *nboundave*, *nexitave*, *nscollave*, *nscheckave*, *ncollave*, *nattemptave*, *nreactave*, and *nsreactave* 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 current run.

The *np* keyword is the number of particles.

The *ntouch* keyword is the number of cells touched by the particles during the move portion of the timestep. E.g. if a particle moves from cell A to adjacent cell B, it touches 2 cells.

The *ncomm* keyword is the number of particles communicated to other processors.

The *nbound* keyword is the number of particles 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 particles that exited the simulation box through an outflow boundary.

The *nscoll* keyword is the number of particle/surface collisions that occurred, where a particle collided with a geometric surface.

The *nscheck* keyword is the number of particle/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 particle in that cell moves.

The *ncoll* keyword is the number of particle/particle collisions that occurred.

The *nattempt* keyword is the number of particle/particle collisions that were attempted.

The *nreact* keyword is the number of chemical reactions that occurred.

The *nsreact* keyword is the number of chemical reactions on surfaces that occurred, including the global boundaries if they are treated as reacting surfaces, via the *bound_modify* command.

The *nparent* keyword is the number of parent cells, including the root cell. The *nchild* keyword is the number of child cells, which includes both unsplit and split cells. The *nsplit* keyword is the number of split cells. See *Section howto 4.8* for a description of the hierarchical grid used by SPARTA and a definition of these various kinds of grid cells.

The *maxlevel* keyword is the maximum number of levels for grid refinement currently in the simulation. This may change due to dynamic grid adaptation.

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.

For output values from a compute or fix, the bracketed index *I* used to index a vector, as in *c_ID[I]* or *f_ID[I]*, can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "n*" or "m*n". If N = the size of the vector (for *mode* = scalar) or the number of columns in the array (for *mode* = vector), then an asterisk with no numeric values means all indices from 1 to N. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from n to N (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual elements of the vector had been listed one by one. E.g. these 2 stats_style commands are equivalent, since the *compute reduce* command creates a global vector with 6 values.

```
compute myCount reduce max x y z vx vy vz
stats_style step np c_myCount[*]
stats_style step np c_myCount[1] c_myCount[2] c_myCount[3] &
                    c_myCount[4] c_myCount[5] c_myCount[6]
```

For the following keywords, the ID in the keyword should be replaced by the actual ID of a surface collision model, surface reaction model, compute, fix, or variable name that has been defined elsewhere in the input script. See those commands for details. If the entity calculates a global scalar, vector, or array, then the keyword formats with 0, 1, or 2 brackets will reference a scalar value from the entity.

The *s_ID[I]* and *r_ID[I]* keywords allow global values calculated by a surface collision model or surface reaction model to be output. As discussed on the *surf_collide* and *surf_react* doc pages, these models both calculate a global vector of quantities.

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-particle, per-grid, or per-surf values. Only global values can be referenced by this command. However, per-particle, per-grid, or per-surf compute values can be referenced in a *variable* and the variable referenced, as discussed below. See the discussion above for how the I in *c_ID[I]* can be specified with a wildcard asterisk to effectively specify multiple values from a global compute vector.

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-particle, per-grid, or per-surf values. Only global values can be referenced by this command. However, per-particle or per-grid or per-surf fix values can be referenced in a *variable* and the variable referenced, as discussed below. See the discussion above for how the I in *f_ID[I]* can be specified with a wildcard asterisk to effectively specify multiple values from a global fix vector.

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-particle or per-grid or per-surf 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 command stats_modify command

Default:

```
stats_style step cpu np
```

1.14.105 suffix command

Syntax:

```
suffix style args
```

- style = *off* or *on* or *kk*

Examples:

```
suffix off
suffix on
suffix kk
```

Description:

This command allows you to use variants of various styles if they exist. In that respect it operates the same as the *-suffix command-line switch*. It also has options to turn off or back on any suffix setting made via the command line.

The specified style *kk* refers to the optional KOKKOS package that SPARTA can be built with, as described in *this section of the manual*. The KOKKOS package is a collection of styles optimized to run using the Kokkos library on various kinds of hardware, including GPUs via CUDA and many-core chips via OpenMP multi-threading.

As an example, the KOKKOS package provides a *compute_style temp* variant, with style name temp/kk. A variant style can be specified explicitly in your input script, e.g. compute temp/kk. If the suffix command is used with the appropriate style, you do not need to modify your input script. The specified suffix (kk) is automatically appended whenever your input script command creates a new *fix*, *compute*, etc. If the variant version does not exist, the standard version is created.

If the specified style is *off*, then any previously specified suffix is temporarily disabled, whether it was specified by a command-line switch or a previous suffix command. If the specified style is *on*, a disabled suffix is turned back on. The use of these 2 commands lets your input script use a standard SPARTA style (i.e. a non-accelerated variant), which can be useful for testing or benchmarking purposes. Of course this is also possible by not using any suffix commands, and explicitly appending or not appending the suffix to the relevant commands in your input script.

Restrictions:

none

Related commands:

Command-line switch -suffix

Default:

none

1.14.106 surf_collide command

Syntax:

```
surf_collide ID style args keyword values ...
```

- ID = user-assigned name for the surface collision model
- style = specular or diffuse or cll or impulsive or td or piston or transparent or vanish or specular/kk or diffuse/kk or piston/kk or vanish/kk
- args = arguments for specific style

  specular or specular/kk args = none

  diffuse or diffuse/kk args = Tsurf acc

  - Tsurf = temperature of surface (temperature units). Tsurf can be a variable (see below)
  - acc = accommodation coefficient

  cll args = Tsurf acc_n acc_t acc_rot acc_vib

  - Tsurf = temperature of surface (temperature units). Tsurf can be a variable (see below)
  - acc_n = accommodation coefficient in the surface normal direction
  - acc_t = accommodation coefficient in the surface tangential direction
  - acc_rot = accommodation coefficient for the rotational modes
  - acc_vib = accommodation coefficient for the vibrational modes

  impulsive args = Tsurf model param1 param2 var theta_peak pol_pow azi_pow

  - Tsurf = temperature of surface (temperature units). Tsurf can be a variable (see below)
  - **model** can be softsphere or tempvar
    - softsphere args = en_ratio eff_mass
      - param1 = en_ratio = fraction of energy lost in the collision
      - param2 = eff_mass = effective mass of the surface atom
    - tempvar args = a1 a0
      - param1 = a1 = linear term in the variation with temperature
      - param2 = a0 = constant term in the variation with temperature
  - var = variance of the scattered particle velocity distribution
  - theta_peak = peak location of the polar angle distribution
  - pol_pow = cosine power represeting the polar angular distribution
  - azi_pow = cosine power represeting the azimuthal angular distribution

  td arg = Tsurf Tsurf = temperature of surface (temperature units). Tsurf can be a variable (see below)

  piston or piston/kk args = Vwall Vwall = velocity of boundary wall (velocity units)

transparent args = none

vanish or vanish/kk args = none

- zero or more keyword/arg pairs may be appended

  keyword = `translate` or `rotate` or `partial`

  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)
  - `partial` args = *eccen (only for cll style)*
    - eccen = eccentricity parameter
  - `barrier` args = *bar_val (only for td style)*
    - bar_val = value of the desorption barrier in temperature units
  - `bond` args = *bond_trans bond_rot bond_vib (only for td style)*
    - bond_trans = amount of bond dissociation energy (in temperature units) going into translational mode
    - bond_rot = amount of bond dissociation energy (in temperature units) going into rotational mode
    - bond_vib = amount of bond dissociation energy (in temperature units) going into vibrational mode
  - `init_energy` = *IE_trans IE_rot IE_vib (only for td style)*
    - IE_trans = fraction of initial translational energy going into translational mode
    - IE_rot = fraction of initial translational energy going into rotational mode
    - IE_vib = fraction of initial translational energy going into vibrational mode
  - `step` args = *epsilon (only for impulsive style)*
    - epsilon = ratio of the height to the width of the step
  - `double` args = *polar_pow_2 (only for impulsive style)*
    - polar_pow_2 = cosine power for the polar angular distribution between peak and surface
  - `intenergy` args = *frac_rot frac_vib (only for impulsive style)*
    - frac_rot = fraction of lost translational energy going into the rotational mode
    - frac_vib = fraction of lost translational energy going into the vibrational mode

Examples:

```
surf_collide 1 specular
surf_collide 1 transparent
surf_collide 1 diffuse 273.15 0.9
surf_collide 1 cll 273.15 0.8 0.8 0.5 0.1
surf_collide 1 cll 273.15 1.0 1.0 0.1 0.1 partial 0.5
surf_collide 1 impulsive 1000.0 softsphere 0.2 50 2000 60 5 75
surf_collide 1 impulsive 1000.0 tempvar 3 500 2000 60 5 75
```

(continues on next page)

(continued from previous page)

```
surf_collide 1 impulsive 1000.0 softsphere 0.2 50 2000 60 5 75 double 10
surf_collide 1 impulsive 1000.0 tempvar 3 500 2000 60 5 75 step 0.1
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 model for particle-surface collisions. One or more models can be defined and assigned to different surfaces or simulation box boundaries via the *surf_modify command* or *bound_modify command*. See *Details of surfaces in SPARTA* for more details of how SPARTA defines surfaces as collections of geometric elements, triangles in 3d and line segments in 2d. Chemical reactions can also be part of a particle-surface interaction model. See the *surf_react command* for details. All of the collision styles listed here support optional reactions, except the *vanish* style.

The ID for a surface collision model is used to identify it in other commands. Each surface collision model ID must be unique. The ID can only contain alphanumeric characters and underscores.

The *specular* style computes a simple specular reflection model. It requires no arguments. Specular reflection means that a particle reflects off a surface 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 from 0.0 to 1.0, which determines what fraction of surface collisions are diffusive. The rest are specular. Thus a setting of *acc* = 0.0 means all collisions are specular and a setting of *acc* = 1.0 means all collisions are diffusive.

Note that setting *acc* = 0.0, is a way to perform surface reactions with specular reflection, via the *surf_react command*, which cannot be done in conjunction with the `surf_collide` specular command. See the *surf_react command* doc page for details.

Diffuse reflection emits the particle 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 thermal components in the direction of the outward surface normal and the plane tangent to the surface given by:

$$u = \{-ln(R_f)\}^{1/2}/\beta$$

The *Tsurf* 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 *stats_style command* keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent temperature.

The *cll* style computes the surface collision model proposed by Cercignani, Lampis and Lord. The model has 5 parameters set by the *Tsurf*, *acc_n*, *acc_t*, *acc_rot*, and *acc_vib* arguments. *Tsurf* is the temperature of the surface. *acc_n*, *acc_t*, *acc_rot*, and *acc_vib* are the accommodation coefficient for the surface normal direction, surface tangential directions, rotational energy mode, and vibrational energy mode respectively. The rotational and vibrational energy accommodation values must be specified even for an atomic species; however these values are simply ignored.

The theoretical scattering kernel was proposed by Cercignani and Lampis [Cercignani71]. In this original model, two accommodation coefficients for the normal and tangential directions are employed. Each of these quantities can take a value between 0 and 1. Specular reflection is achieved by using the values (0,0), while complete thermal accommodation with the surface and cosine angular distributions is obtained using (1,1). There is smooth variation of both the energy and angular distribution for values in between these limits leading to lobular distributions similar to those observed in experiments. The implementation details of this model within DSMC was given by Lord [Lord90], along with extension to rotational and vibrational modes with both continuous and discrete levels [Lord91].

The *Tsurf* 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 *stats_style command* keywords for the simulation box parameters and timestep and elapsed time. Thus, it is easy to specify a time-dependent temperature.

The *td* style computes the thermal desorption surface collision model proposed by Swaminathan Gopalan *et al.* [SG18]. The model has 1 parameter set by *Tsurf* argument, which is the temperature of the surface. This is similar to *diffuse* style with an accommodation coefficient *acc* = 1.0.

The particles are scattered thermally based on the Maxwell Boltzmann distribution conisstent with the surface temperture. The new velocity will have thermal components in the direction of the outward surface normal and the plane tangent to the surface given by:

$$u = \{-ln(R_f)\}^{1/2}/\beta$$

The *Tsurf* 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 *stats_style command* keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent temperature.

The *impulsive* style computes the surface collision model proposed by Swaminathan Gopalan *et al.* [SG18]. The model has 8 parameters. Within impulsive scattering, two different models are available, namely *softsphere* and *tempvar*. The *softsphere* argument uses the soft sphere model and has two parameters: *en_ratio* which represents the fraction of energy lost during the collision, and *eff_mass* specifying the effective mass of the surface atom. The *tempvar* argument directly provides the peak value of the scattered particle velocity distribution as a linear function of temperature. It has two parameters: the linear term *a1* and constant term *a0*. The other five parameters *Tsurf*, *var*, *pol_peak*, *pol_pow*, *azi_pow* are set for both the models. *Tsurf* is the surface temperature. *var* is the variance of the scattered particle velocity distribution. *pol_peak* is the peak of the polar angle distribution. *pol_pow* and *azi_pow* are the cosine power representing the polar and azimuthal angle distribution respectively.

The *impulsive* model is used to represent the scattering of particles having super or hyperthermal translational energies and very low internal energies, like in a beam. This type of scattering falls under the structural regime, whose scattering physics and distributions are very different from the thermal regime. The velocity distribution of the impulsive scattering model can be represented using a Gaussian distribution with a mean u_0 and a variance α following Rettner [Rettner94a]

$$f_{\text{impulsive}}(u) \propto u^2 \exp\left(-\frac{(u-u_0)^2}{2\alpha^2}\right)$$

The variance parameter is directly specified by the user. The value of u_0 can be provided directly using the *tempvar* model in which it is represented as a linear function of temperature. The linear term *a1* and constant term *a0* are given

as inputs.

$$\langle E_f \rangle = E_i \left(1 - \frac{2\mu}{(\mu+1)^2} \left[1 + \mu \sin^2 \chi + \frac{E_{\text{int}}}{E_i} \left(\frac{\mu+1}{2\mu} \right) - \cos\chi \sqrt{1 - \mu^2 \sin^2 \chi - \frac{E_{\text{int}}}{E_i} (\mu+1)} \right] \right)$$

The u_0 parameter can also be specified by a more physical model such as the soft sphere scattering model [Alexander12]. This model uses the parameters *en_ratio*, the fraction of energy lost in the collision and *eff_mass*, the effective mass of the surface atom to determine the average final energy and then the average final velocity *u0*. Within the soft sphere model, the average final velocity will vary as a function of the final polar angle.

$$u_0 = a_1 \cdot T + a_0$$

Both the polar and azimuthal angular distribution are lobular in nature and sharply peaked. These distributions can be represented using the cosine power law distribution [Glatzer97]. The peak of the azimuthal distribution remains at zero, while the peak of the polar angle distribution is usually higher than the incident angle (away from the normal). Hence the peak location (theta_peak) and cosine power (n) of the polar angle distribution and the cosine power (m) of the azimuthal angular distribution are taken as input parameters. A factor of 2 is present in the azimuthal distribution to ensure the function remians positive within the range of the azimuthal angle: (-180, 180)

$$N(\theta) \propto \cos^n (\theta - \theta_{\text{peak}})$$

$$N(\phi) \propto \cos^m \left(\frac{\phi}{2} \right)$$

The internal (rotational and vibrational) energy of an incident molecule remains unchanged within the *impulsive* model unless the optional keyword *intenergy* is specified (see below).

The *Tsurf* 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 *stats_style command* keywords for the simulation box parameters and timestep and elapsed time. Thus, it is easy to specify a time-dependent temperature.

The *piston* style models a subsonic pressure boundary condition. It can only be assigned to the simulation box boundaries via the *bound_modify command* or to surface elements which are parallel to one of the box boundaries (via the *surf_modify command*).

It treats collisions of particles with the surface as if the surface were moving with specified velocity *Vwall* away from the incident particle. Thus the "collision" actually occurs later in the timestep and the reflected velocity is less than it would be for reflection from a stationary surface. This calculation is performed using equations 12.30 and 12.31 in [Bird94] to compute the reflected velocity and final position of the particle. If the particle does not return within the timestep to a position inside the simulation box (for a boundary surface) or to the same side of the initial surface that it started from (for a surface element collision), the particle is deleted. This effectively induces particles at the boundary to have a velocity distribution consistent with a subsonic pressure boundary condition, as explained in [Bird94].

Vwall should be chosen to correspond to a desired pressure condition for the density of particles in the system.

NOTE: give more details on how to do this?

Note that *Vwall* must always be input as a value >= 0.0, meaning the surface is moving away from the incident particle. For example, in the z-dimension, if the upper box face is assigned *Vwall*, it is moving upward. Similarly if the lower box face is assigned *Vwall*, it is moving downward.

The *transparent* style simply allows particles to pass through the surface without altering the particle properties.

This is useful for tallying flow statistics. The surface elements must have been flagged as transparent when they were read in, via the *read_surf command* and its transparent keyword. The *compute surf command* will tally fluxes differently for transparent surf elements. The *Transparent surface elements* doc page provides an overview of transparent surfaces. See those doc pages for details.

The *vanish* style simply deletes any particle which hits the surface.

This is useful if a surface is defined to be an inflow boundary on the simulation domain, e.g. using the *fix emit/surf command*. Using this surface collision model will also treat the surface as an outflow boundary. This is similar to using the *fix emit/face command* on a simulation box face while also setting the face to be an outflow boundary via the *boundary o* command.

Note that the *surf_react global* command can also be used to delete particles hitting a surface, by setting the *pdelete* parameter to 1.0. Using a surf_collide vanish command is simpler.

The keyword `translate` can only be applied to the *diffuse* and *cll* 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.

The keyword `rotate` can only be applied to the *diffuse* and *cll* 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). Note that W and P define the rotation axis. The magnitude of W defines the speed of rotation. I.e. if the length of W = 2*pi then the surface is rotating at one revolution per time unit, where time units are defined by the *units command*.

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.

The `rotate` keyword can be used to treat a simulation box boundary as a rotating wall, e.g. the end cap of an axisymmetric cylinder. Or to model a rotating object consisting of surface elements, e.g. a sphere. In either case, the wall or surface elements themselves do not change position due to rotation. They are simply modeled as having a tangential velocity, as if the entire object were rotating.

Important: For both the `translate` and `rotate` keywords the added velocity can only be tangential to the surface, with no normal component since the surface is not actually moving in the normal direction. SPARTA does not check that the specified translation or rotation produces a tangential velocity. However if does enforce the condition by subtracting off any component of the added velocity that is normal to the simulation box boundary or individual surface element.

The keyword `partial` can only be applied to the *cll* style. Within the CLL model, the energy and angular distribution are linked. Lord [Lord95] proposed a way to decouple the energy accommodation from the angular distribution. This case of partially diffuse scattering with incomplete energy accommodation can be activated in SPARTA using the optional keyword *partial*. It requires an additional parameter eccentricity set by the *eccen* argument. For this case, the energy accommodation is calculated using the accommodation coefficients, but the angular distribution is computed using the additional parameter eccentricity. The `eccen` parameter can vary between 0 and 1. A value of 0 represents fully diffuse scattering and gives a cosine angular distribution. Increasing value of *eccen* presents more peaked and lobular distribution [Lord95].

The keywords `barrier`, `bond`, and `initenergy` can only be applied to the `td` style. Due to the nature of the interaction between the products and the surface, the desorption of the products might have an energy barrier. For a surface desorption process, this desorption barrier exists only in the normal direction. Thus, only the products having enough energy (in the normal direction) to overcome the barrier will be able to desorb from the surface. This alters the velocity distribution of the observed products along the surface normal direction and thus leads to the distortion of the speed distribution [Goodman72]. The angular distributions, which represent the ratio of the normal to the tangential

velocities, are also altered as a result of the desorption barrier. The angular distributions are peaked more towards the normal and are often described by a cosine power law distribution.

$$T_{\text{norm}} = T_{\text{surf}} \left(1 + \frac{E_{\text{barrier}}}{k_b}\right).$$

$$f(v) \propto v^2 \exp\left(-\frac{mv^2}{2k_b}\left(\frac{\cos^2\theta}{T_{\text{norm}}} + \frac{\sin^2\theta}{T_{\text{surf}}}\right)\right)$$

In addition to the desorption energy barrier, products formed through thermal mechanisms might have energies exceeding those corresponding to the bulk surface temperature. The energy of the local surface environment where the product formation occurs might be greater than the normal surface temperature due to the formation of local hot-spots [Rettner94b].

These hot-spots might stem from the dissociation or bond energy of the intermediates or the products. The optional keyword `bond` can be used to account for this scenario. This requires three arguments: the amount of energy (in temperature units) going into the translational, rotational and vibrational mode.

$$E_{prod} = k_b T_s + k_b \sigma_2$$

The higher energy during desorption might also arise due to the energy deposited by high speed of the incoming gas-phase particles. Since the formation of the products is rapid, the product might form and desorb before this high energy dissipates from the local hot-spots [Beckerle90]. In this case, although the products are in thermal equilibrium with the surroundings, the energies of the products might not depend only on the equilibrium surface temperature, but also on the incoming velocities of the particles. This can be used within SPARTA using the optional keyword `initenergy`. It requires 3 arguments: fraction of the initial translational energy going into the translational, rotational and vibrational modes.

$$E_{prod} = k_b T_s + \sigma_1 E_{in}$$

The keywords `step`, `double`, and `intenergy` can only be applied to the `impulsive` style. In some cases, it is observed that the polar angular distribution on either side of the peak is different. Goodman [Goodman74] provided a physical reasoning for the observed faster decay rate in the polar angular distribution away from the normal with the surface assumed to consist of periodic steps of average height H and average periodicity L. The ratio of the height to periodicity is *epsilon* and the correction to the angular distribution is given by

$$f_{corr} = \begin{cases} 1 - \epsilon \tan(\theta_0), & \text{if } \tan(\theta_0) < \epsilon^{-1} \\ 0, & \text{otherwise} \end{cases}$$

This optional argument can be accessed using the keyword `step`, and `epsilon` parameter must be specified. Another optional argument to specify the angular distribution of the products is the *double* keyword. In this option, the angular distribution on either sides of the peak are represented by a different cosine power decay. It requires one argument *pol_pow_2*, which describes the distribution between the peak and the surface. The distribution between the surface normal and the peak is described using the parameter *pol_pow*.

The keyword `intenergy` can be used to modify the internal energy of an incident molecule during collision. In the case of hyperthermal collision the energy from the translational mode is transfered to the internal modes. This keyword requires two input parameters *frac_rot* and *frac_vib*. These specify the fraction of the change in translational energy (difference between the final and initial) transferred to the rotational and vibrational mode respectively.

Output info:

All the surface collide models calculate a global vector of length 2. The values can be used by the *stats_style command* and by *variable command* that define formulas. The latter means they can be used by any command that uses a variable as input, e.g. the *fix ave/time command*. See *Output from SPARTA (stats, dumps, computes, fixes, variables)* for an overview of SPARTA output options.

The first element of the vector is the count of particles that hit surface elements assigned to this collision model during the current timestep. The second element is the cummulative count of particles that have hit surface elements since the current run began.

Styles with a *kk* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed in the *Accelerating SPARTA* section of the manual. The accelerated styles take the same arguments and should produce the same results, except for different random number, round-off and precision issues.

These accelerated styles are part of the KOKKOS package. They are only enabled if SPARTA was built with that package. See the *Making SPARTA* section for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke SPARTA, or you can use the *suffix command* in your input script.

See the *Accelerating SPARTA* section of the manual for more instructions on how to use the accelerated styles effectively.

Restrictions:

The *translate* and *rotate* keywords cannot be used together.

If specified with a *kk* suffix, this command can be used no more than twice in the same input script (active at the same time).

Related commands:

read_surf command, *bound_modify command*

Default:

none

References:

1.14.107 surf_modify command

Syntax:

```
surf_modify group-ID keyword args ...
```

group-ID = ID of the surface group to operate on

- one or more keyword/arg pairs may be listed
- keyword = *collide* or (react)

  collide arg = sc-ID sc-ID = ID of a surface collision model

  react arg = sr-ID sr-ID = ID of a surface reaction model or none

Examples:

```
surf_modify sphere collide 1
surf_modify all collide sphere react sphere
```

Description:

Set parameters for a group of surface elements in the specified group-ID. Surface elements are read in by the *read_surf* command. They can be assigned to groups by that command or via the *group* command.

The *collide* keyword is used to assign a surface collision model. Surface collision models are defined by the *surf_collide* command, which assigns each a surface collision ID, specified here as *sc-ID*.

The effect of this keyword is that particle collisions with surface elements in group-ID will be computed by the surface collision model with *sc-ID*.

The *react* keyword is used to assign a surface reaction model. Surface reaction models are defined by the *surf_react* command, which assigns each a surface reaction ID, specified here as *sr-ID* or the word "none". The latter means no reaction model.

The effect of this keyword is that particle collisions with surface elements in group-ID will induce reactions which are computed by the surface reaction model with *sr-ID*. If "none" is used, no surface reactions occur.

Note: If the same surface element is assigned to multiple groups, using this command multiple times may override the effect of a previous command that assigned a different collision or reaction model to a particular surface element.

Restrictions:

All surface elements must be assigned to a surface collision model via the *collide* keyword before a simlulation can be performed. Using a surface reaction model is optional.

This command cannot be used before surfaces exist.

Related commands:

read_surf command, *bound_modify command*

Default:

The default for surface reactions is none.

1.14.108 surf_react command

Syntax:

```
surf_react ID style args
```

- ID = user-assigned name for the surface reaction model
- style = *global* or *prob*

- args = arguments for that style

```
global args = pdelete pcreate
  pdelete = probability that surface collision removes the incident particle
  pcreate = probability that surface collision clones the incident particle
prob args = infile
  infile = file with list of surface chemistry reactions
```

Examples:

```
surf_react global 0.2 0.15
surf_react prob air.surf
```

Description:

Define a model for surface chemistry reactions to perform when particles collide with surface elements or the global boundaries of the simulation box. One or more models can be defined and assigned to different surfaces or simulation box boundaries via the *surf_modify* or *bound_modify* commands. See *Section 6.9* for more details of how SPARTA defines surfaces as collections of geometric elements, triangles in 3d and line segments in 2d. Also see the *react* command for specification of a gas-phase chemistry reaction model.

The ID for a surface reaction model is used to identify it in other commands. Each surface reaction model ID must be unique. The ID can only contain alphanumeric characters and underscores.

The surface reaction models for the various styles are described below. When a a particle collides with a surface element or boundary. the list of all reactions possible with that species as the reactant is looped over. A probability for each reaction is calculated, using the formulas discussed below, and a random number is used to decide which reaction (if any) takes place. A check is made that the sum of probabilities for all possible reactions is <= 1.0, which should normally be the case if reasonable reaction coefficients are defined.

Important: A surface reaction model can not be specified for surfaces whose surface collision style does not support reactions. Currently this is only the *vanish* collision style. See the *surf_collide* doc page for details.

The *global* style is a simple model that can be used to test whether surface reactions are occurring as expected. There is no list of reactions for different species; all species are treated the same. This style thus defines two universal reactions, the first for particle deletion, the second for particle creation.

The *global* style takes two parameters, *pdelete* and *pcreate*. The first is the probability that a "deletion" reaction takes place when a collision occurs. If it does, the particle is deleted. The second is the probablity that a "creation" reaction occurs, which clones the particle, so that one particle becomes two. The two particles leave the surface according to whatever surface collision model is defined by the *surf_collide* command, and is assigned to that surface/boundary by the *surf_modify collide* command.

The sum of *pdelete* and *pcreate* must be <= 1.0.

Note that if you simply wish to delete all particles which hit the surface, you can use the *surf_collide vanish* command, which is simpler.

For the *prob* style, a file is specified which contains a list of surface chemical reactions, with their associated parameters. The reactions are read into SPARTA and stored in a list. Each time a simulation is run via the *run* command, the list is scanned. Only reactions for which all the reactants and all the products are currently defined as species-IDs

will be active for the simulation. Thus the file can contain more reactions than are used in a particular simulation. See the *species* command for how species IDs are defined. This style thus defines N reactions, where N is the number of reactions listed in the specified file.

As explained below each reaction has a specified probability between 0.0 and 1.0. That probability is used to choose which reaction (if any) is performed.

The format of the input surface reaction file is as follows. Comments or blank lines are allowed in the file. Comment lines start with a "#" character. All other entries must come in 2-line pairs with values separated by whitespace in the following format

```
R1 --> P1 + P2
type style C1 C2 ...
```

The first line is a text-based description of a single reaction. R1 is a single reactant for the particle that collides with the surface/boundary, listed as a *species* IDs. P1 and P2 are one or two products, also listed as *species* IDs. The number of reactants is always 1. The number of allowed products depends on the reaction type, as discussed below. Individual reactants and products must be separated by whitespace and a "+" sign. The left-hand and right-hand sides of the equation must be separated by whitespace and "–>".

The *type* of each reaction is a single character (upper or lower case) with the following meaning. The type determines how many reactants and products can be specified in the first line.

```
D = dissociation = 1 reactant and 2 products
E = exchange = 1 reactant and 1 product
R = recombination = 1 reactant and 1 product named NULL
```

A dissociation reaction means that R1 dissociates into P1 and P2 when it collides with the surface/boundary. There is no restriction on the species involved in the reaction.

An exchange reaction is a collision where R1 becomes a new product P1. There is no restriction on the species involved in the reaction.

A recombination reaction is a collision where R1 is absorbed by the surface, so that the particle disappears. There are no products which is indicated in the file by listing a single product as NULL. There is no restriction on the species involved in the reaction.

The *style* of each reaction is a single character (upper or lower case) with the following meaning:

- S = Surface

The style determines how many reaction coefficients are listed as C1, C2, etc, and how they are interpreted by SPARTA.

For S = Surface style, there is a single coefficient:

- C1 = probability that the reaction occurs (0.0 to 1.0)

If the ambipolar approximation is being used, via the *fix ambipolar command*, then reactions which involve either ambipolar ions or the ambipolar electron have more restricitve rules about the ordering of reactants and products, than those described in the preceeding section for the *prob* style.

The first is an "exchange" reaction which converts an ambipolar ion into a neutral species. Internally this removes the ambipolar electron associated with the ion. In the file of reactions this is done by having the reactant be an ambipolar ion, and the product not be an ambipolar ion.

The second is a "dissociation" reaction where a neutral species is ionized by colliding with the surface/boundary, creating an ambipolar ion and ambipolar electron. In the file of reactions this is done by having the reactant not be an ambipolar ion, the first product be an ambipolar ion, and the second product be an ambipolar electron. The two products must be specified in this order.

Output info:

All the surface reaction models calculate a global vector of values. The values can be used by the *stats_style* command and by *variables* that define formulas. The latter means they can be used by any command that uses a variable as input, e.g. "the *fix ave/time* command. See *Section 4.4* for an overview of SPARTA output options.

The *global* and *prob* styles each compute a vector of length 2 + 2*nlist. For the *global* style, nlist = 2, for "delete" and "create" reactions. For the *prob* style, nlist is the number of reactions listed in the file is read as input.

The first element of the vector is the count of particles that performed surface reactions for surface elements assigned to this reaction model during the current timestep. The second element is the cummulative count of particles that have performed reactions since the beginning of the current run. The next nlist elements are the count of each individual reaction that occurred during the current timestep. The final nlist elements are the cummulative count of each individual reaction since the beginning of the current run.

Restrictions:

none

Related commands:

react command surf_modify command bound_modify command

Default:

none

1.14.109 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 command

Default:

```
timestep 1.0
```

1.14.110 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 command

Default:

none

1.14.111 undump command

Syntax:

```
undump dump-ID
```

- dump-ID = ID of previously defined dump

Examples:

```
undump mine
undump 2
```

Description:

Delete a dump that was previously defined with a *dump command*. This also closes the file associated with the dump.

Restrictions:

none

Related commands:

dump command

Default:

none

1.14.112 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 command

Default:

none

1.14.113 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.

Important: Internally, this command simply sets the numeric values of conversion factors used by SPARTA, e.g. the Boltzmann constant used to convert temperature to energy. It is up to you to insure that all input values used in the input script and other input files (surface data, species files, reaction files) contain numeric values consistent with the chosen units.

For style *cgs*, these are the units:

- mass = grams
- distance = centimeters
- area = cm^2
- volume = cm^3
- time = seconds
- energy = ergs
- velocity = centimeters/second
- acceleration = centimeters/second^2
- pressure = barye (dyne/cm^2 = 0.1 pascals)
- 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
- acceleration = meters/second^2
- pressure = pascals (newton/meter^2)
- temperature = degrees K

The units command also sets a default timestep size; see the *timestep command* to change this value.

- For style *cgs* this is dt = 1.0 sec.
- For style *si* this is dt = 1.0 sec.

Restrictions:

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

Related commands:

none

Default:

```
units si
```

1.14.114 variable command

1.14.115 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 `format` or `getenv` or `file` or `internal` or `equal` or `particle` or `grid`
  - 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
- format args = vname fstr - vname = name of equal-style variable to evaluate - fstr = C-style format string
- getenv arg = one string
- file arg = filename
- internal arg = numeric value
- equal or particle or grid args = one formula containing numbers, stats keywords, math operations, particle vectors, compute/fix/variable references
  - numbers = 0.0, 100, -5.4, 2.8e-4, etc
  - constants = PI
  - stats keywords = step, np, vol, etc from stats_style
  - math operators = (), -x, x+y, 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), abs(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), stride(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), slope(x), next(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]
  - surface collision model references = s_ID[i]
  - surface reaction model references = r_ID[i]
  - 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 beta equal "temp / 3.0"
variable b equal c_myTemp
variable b particle x*y/vol
variable foo string myfile
variable foo internal 3.5
variable f file values.txt
```

(continues on next page)

(continued from previous page)

```
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 str format x %.6g
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 thus be useful in several contexts. A variable can be defined and then 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), or used as input to an averaging fix (see the *fix ave/time* 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 particle* command). Variables of style *internal* are used by a few commands which set their value directly.

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: When an input script line is encountered that defines a variable of style *equal* or *particle* or *grid* that contains a formula, 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. This is also true of the *format* style variable since it evaluates another variable when it is invoked.

Variables of style *equal* and *particle* and *grid* can be used as inputs to various other commands which evaluate their formulas as needed, e.g. at different timesteps during a *run*.

Variables of style *internal* can be used in place of an equal-style variable, except by commands that set the value stored by the internal-style variable. Thus any command that states it can use an equal-style variable as an argument, can also use an internal-style variable. This means that when the command evaluates the variable, it will use the value set (internally) by another command.

Important: 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*, *getenv*, *internal*, *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 or v_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.

Section 3.2 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
```

Styles and arguments

This section describes how various variable styles are defined and what they store. Many of the styles store one or more strings. Note that a single string can contain spaces (multiple words), if it is enclosed in quotes in the variable command. When the variable is substituted for in another input script command, its returned string will then be interpreted as multiple arguments in the expanded command.

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 *Section 2.6* of the manual 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. N1 <= N2 and N2 >= 0 is required.

For the *world* style, one or more strings are specified. There must be one string for each processor partition or "world". See *Section 2.6* 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.

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.sparta.variable" and "tmp.sparta.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 *string* style, a single string is assigned to the variable. The only difference between this and using the *index* style with a single string is that a variable with *string* style can be redefined. E.g. by another command later in the input script, or if the script is read again in a loop.

For the *format* style, an equal-style variable is specified along with a C-style format string, e.g. "%f" or "%.10g", which must be appropriate for formatting a double-precision floating-point value. This allows an equal-style variable to be formatted specifically for output as a string, e.g. by the *print* command, if the default format "%.15g" has too much precision.

For the *getenv* style, a single string is assigned to the variable which should be the name of an environment variable. When the variable is evaluated, it returns the value of the environment variable, or an empty string if it not defined. This style of variable can be used to adapt the behavior of SPARTA input scripts via environment variable settings, or to retrieve information that has been previously stored with the *shell putenv* command. Note that because environment variable settings are stored by the operating systems, they persist beyond a *clear* command.

For the *file* style, a filename is provided which contains a list of strings to assign to the variable, one per line. The strings can be numeric values if desired. See the discussion of the next() function below for equal-style variables, which will convert the string of a file-style variable into a numeric value in a formula.

When a file-style variable is defined, the file is opened and the string on the first line is read and stored with the variable. This means the variable can then be evaluated as many times as desired and will return that string. There are two ways to cause the next string from the file to be read: use the *next* command or the next() function in an equal- or particle- or grid-style variable, as discussed below.

The rules for formatting the file are as follows. A comment character "#" can be used anywhere on a line; text starting with the comment character is stripped. Blank lines are skipped. The first "word" of a non-blank line, delimited by white space, is the "string" assigned to the variable.

For the *internal* style a numeric value is provided. This value will be assigned to the variable until a SPARTA command sets it to a new value. There is currently only one command that requirew *internal* variables as inputs, because it resets them: *create_particles*. As mentioned above, an internal-style variable can be used in place of an equal-style variable anywhere else in an input script, e.g. as an argument to another command that allows for equal-style variables.

For the *equal* and *particle* and *grid* 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. For *grid* style variables the formula computes one quantity for each grid cell whenever it is evaluated. A *grid* style variable computes quantites for all flavors of child grid cells in the simulation, which includes unsplit, cut, split, and sub cells. See *Details of grid geometry in SPARTA* of the manual gives details of how SPARTA defines child, unsplit, split, and sub cells.

Note: that *equal* and *particle* and *grid* 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* or *grid* style variables, since there is only one string.

Formulas

The formula for an *equal* or *particle* or *grid* 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 "np + 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.

Table 3: Components of formulas

Number	0.2, 100, 1.0e20, -15.4, etc
Constant	PI
Stats keywords	step, np, vol, etc
Math operators	(), -x, x+y, 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), abs(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), stride(x,y,z), vdisplace(x,y), swiggle(x,y,z), cwiggle(x,y
Special functions	sum(x), min(x), max(x), ave(x), trap(x), slope(x), next(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]
Surface collision model referen	es s_ID[i]
Surface reaction model referenc	s r_ID[i]
Other variables	v_name

Most of the formula elements produce a scalar value. A few produce a per-particle vector or per-grid vector of values. These are the particle vectors, compute references that represent a per-particle or per-grid vector, fix references that represent a per-particle or per-grid vector, and variables that are particle-style or grid-style variables. Math functions that operate on scalar values produce a scalar value; math function that operate on per-particle vectors do so element-by-element and produce a per-particle vector.

A formula for equal-style variables cannot use any formula element that produces a per-particle or per-grid vector. A formula for a particle-style variable can use formula elements that produce either a scalar value or a per-particle vector, but not a per-grid vector. Likewise a particlgrid-style variable can use formula elements that produce either a scalar value or a per-grid vector, but not a per-particle 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 stats 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-particle or per-grid vectors. For example, "vol/np" is the division of two scalars, where "vy+vz" is the element-by-element sum of two per-particle 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 and the modulo operator "%" 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.

Important: Because a unary minus is higher precedence than exponentiation, the formula "-2^2" will evaluate to 4, not -4. This convention is compatible with some programming languages, but not others. As mentioned, this behavior

can be easily overridden with parenthesis; the formula "-(2^2)" will evaluate to -4.

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<10.0 in a particle-style variable formula will return 1.0 for all particles whose x-coordinate is less than 10.0, and 0.0 for the others. 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.

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-particle 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-particle or per-grid vectors. In the latter cases, the math operation is performed on each element of the vector. For example, "sqrt(np)" is the sqrt() of a scalar, where "sqrt(y*z)" yields a per-particle 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 σ^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 or grid-style variables, a unique seed is created for each processor. This effectively generates a different random number for each particle or grid cell being looped over in the particle-style or grid-style variable.

Important: Internally, there is just one random number generator for all equal-style variables and one for all particle-style and grid-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.

Important: Currently, the run command does not currently support the start/stop keywords. In the formula above startstep = 0 and stopstep = the number of timesteps being performed by the run.

The `stagger(x,y)` function uses the current timestep to generate a new timestep. X,y > 0 and x > y are 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 are 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 `stride(x,y,z)` function uses the current timestep to generate a new timestep. X,y >= 0 and z > 0 and x <= y are required. The generated timesteps increase in increments of z, from x to y, I.e. it generates the sequece x,x+z,x+2z,...,y. If y-x is not a multiple of z, then similar to the way a for loop operates, the last value will be one that does not exceed y. For any current timestep, the next timestep in the sequence is returned. Thus if `stagger(1000,2000,100)` is used in a variable by the *dump_modify every* command, it will generate the sequence of output timesteps:

```
1000,1100,1200, ... ,1900,2000
```

The `vdisplace(x,y)` function takes 2 arguments: x = value0 and y = velocity, and uses the elapsed time to change the value by a linear displacement due to the applied velocity over the course of a run, according to this formula:

```
value = value0 + 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 *stats_style* keyword *elaplong* = timestep-startstep.

The `swiggle(x,y,z)` and `cwiggle(x,y,z)` functions each take 3 arguments: x = value0, y = amplitude, z = period. They use the elapsed time to oscillate the 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 = value0 + Amplitude * sin(omega*(timestep-startstep)*dt)
value = value0 + 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 *stats_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)`, `trap(x)`, and `slope(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-particle or per-grid 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 integration 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()`.

The `slope()` function uses linear regression to fit a line to the 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 returned value is the slope of the line. If the line has a single point or is vertical, it returns 1.0e20.

The `next(x)` function takes 1 argument which is a variable ID (not "v_foo", just "foo"). It must be for a file-style variable. Each time the `next()` function is invoked (i.e. each time the equal-style or particle-style or grid-style variable is evaluated), the following steps occur.

For file-style variables, the current string value stored by the file-style variable is converted to a numeric value and returned by the function. And the next string value in the file is read and stored. Note that if the line previously read from the file was not a numeric string, then it will typically evaluate to 0.0, which is likely not what you want.

Since file-style variables read and store the first line of the file when they are defined in the input script, this is the value that will be returned the first time the `next()` function is invoked. If `next()` is invoked more times than there are lines in the file, the variable is deleted, similar to how the *next* command operates.

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.

The meaning of the different particle vectors is self-explanatory.

Particle vectors can only be used in *particle* style variables, not in *equal* or *grid* style varaibles.

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-particle, per-grid, or per-surf values. Only global and per-particle and per-grid 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-particle vector values. A vector value can be a per-particle vector itself, or a column of an per-particle array. Grid-style variables can use the same scalar values. They can also use per-grid vector values. A vector value can be a per-grid vector itself, or a column of an per-grid 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-particle or per-grid quantities, never more than one kind of quantity.

c_ID	global scalar, or per-particle or per-grid vector
c_ID[I]	Ith element of global vector, or Ith column from per-particle or per-grid array
c_ID[I][J]	I,J element of global array

For I and J, integers can be specified or a variable name, specified as v_name, where name is the name of the variable, like x[v_myIndex]. The variable can be of any style expect particle-style. The variable is evaluated and the result is expected to be numeric and is cast to an integer (i.e. 3.4 becomes 3), to use an an index, which must be a value from 1 to N. Note that a "formula" cannot be used as the argument between the brackets, e.g. x[243+10] or x[v_myIndex+1] are not allowed. To do this a single variable can be defined that contains the needed formula.

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-particle, per-grid, or per-surf values. Only global and per-particle and per-grid 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-particle vector values. A vector value can be a per-particle vector itself, or a column of an per-particle array. Grid-style variables can use the same scalar values. They can also use per-grid vector values. A vector value can be a per-grid vector itself, or a column of an per-grid 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-particle or per-grid quantities, never more than one kind of quantity.

f_ID	global scalar, or per-particle or per-grid vector
f_ID[I]	Ith element of global vector, or Ith column from per-particle or per-grid array
f_ID[I][J]	I,J element of global array

For I and J, integers can be specified or a variable name, specified as v_name, where name is the name of the variable. The rules for this syntax are the same as for the "Compute References" discussion above.

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.

Surface Collision and Surface Reaction Model References

These references access quantities calculated by a *surf_collide command* or *surf_react command*. The ID in the reference should be replaced by the ID of a surface collision or surface reaction model defined elsewhere in the input script. As discussed in the doc pages for the *surf_collide command* and *surf_react command*, they produce global vectors, the elements of which can be accessed by equal-style or particle-style or grid-style variables, e.g.

s_ID[I]	Ith element of global vector for a surface collision model
r_ID[I]	Ith element of global vector for a surface reaction model

---

Variable References

Variable references access quantities stored or calculated 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, equal-style variables generate a global scalar numeric value; particle-style variables generate a per-particle vector of numeric values; grid-style variables generate a per-grid vector of numeric values; all other variables store a string. The formula for an equal-style variable can use any style of variable except a particle- or grid-style. The formula for a particle-style variable can use any style of variable except a grid-style. The formula for a grid-style variable can use any style of variable except a particle-style. If a string-storing variable is used, the string is converted to a numeric value. Note that this will typically produce a 0.0 if the string is not a numeric string, which is likely not what you want. The formula for a particle-style variable can use any style of variable, including other 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-particle or per-grid vector, never more than one of these quantities.

v_name	scalar, or per-particle or per-grid vector

Immediate Evaluation of Variables:

There is a difference between referencing a variable with a leading `$` sign (e.g. `$x` or `${abc}`) versus with a leading `v_` (e.g. `v_x` or `v_abc`). The former can be used in any input script command, including a variable command. The input script parser evaluates the reference variable immediately and substitutes its value into the command. As explained in *Section commands 3.2* for "Parsing rules", you can also use un-named "immediate" variables for this purpose. For example, a string like this `$((xlo+xhi)/2+sqrt(v_area))` in an input script command evaluates the string between the parenthesis as an equal-style variable formula.

Referencing a variable with a leading `v_` is an optional or required kind of argument for some commands (e.g. the *fix ave/spatial* or *dump custom* or *stats_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 "n" as

```
variable n equal np
```

before a run where the particle count changes. You might think this will assign the initial count to the variable "n". That is not the case. Rather it assigns a formula which evaluates the count (using the stats_style keyword "np") to the variable "n". If you use the variable "n" in some other command like *fix ave/time* then the current particle count will be evaluated continuously during the run.

If you want to store the initial particle count of the system, it can be done in this manner:

```
variable n equal np
variable n0 equal $n
```

The second command will force "n" to be evaluated (yielding the initial count) and assign that value to the variable "n0". Thus the command

```
stats_style custom step v_n v_n0
```

would print out both the current and initial particle count periodically during the run.

Also note that it is a mistake to enclose a variable formula in quotes if it contains variables preceeded by $ signs. For example,

```
variable nratio equal "${nfinal}/${n0}"
```

This is because the quotes prevent variable substitution (see *Section 2.2* of the manual on parsing input script commands), and thus an error will occur when the formula for "nratio" is evaluated later.

Variable Accuracy:

Obviously, SPARTA attempts to evaluate variables containing formulas (*equal* and *particle* and *grid* style variables) accurately whenever the evaluation is performed. Depending on what is included in the formula, this may require invoking a *compute*, or accessing a value previously calculated by a compute, or accessing a value calculated and stored by a *fix*. If the compute is one that calculates certain properties of the system such as the pressure induced on a global boundary due to collisions, then these quantities need to be tallied during the timesteps on which the variable will need the values.

SPARTA keeps track of all of this during a *run*. An error will be generated if you attempt to evaluate a variable on timesteps when it cannot produce accurate values. For example, if a *stats_style custom* command prints a variable which accesses values stored by a *fix ave/time* command and the timesteps on which stats 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 was invoked on the current timestep. 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 will generate an error message stating so. For example, if the variable requires a quantity from a *compute* that has not been invoked on the current timestep, 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, a variable containing a compute cannot be evaluated unless the compute was invoked on the last timestep of the preceding run, e.g. by stats output.

   One way to get around this problem is to perform a 0-timestep run before using the variable. For example, these commands

   ```
   compute myTemp grid all temp
   variable t equal c_myTemp1
   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:

   ```
   compute myTemp grid all temp
   variable t equal c_myTemp1
   run 0
   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 *stats* 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 can insure it is invoked on the last timestep of the preceding run by including it in stats 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:

```
compute myTemp grid all temp
variable t equal c_myTemp1
run 1000
create_particles all n 10000
print "Final temperature = $t"
```

The first run is performed using the current set of particles. The temperature is evaluated on the final timestep and stored by the *compute grid* compute (when invoked by the *stats_style* command). Then new particles are added by the *create_particles* command, altering the temperature of the system. When the temperature is printed via the "t" variable, SPARTA will use the temperature value stored by the *compute grid command*, 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 temperature that reflected the new particles:

```
compute myTemp grid all temp
variable t equal c_myTemp1
run 1000
create_particles all n 10000
run 0
print "Final temperature = $t"
```

Restrictions:

All *universe*- and *uloop*-style variables defined in an input script must have the same number of values.

Related commands:

next command, *jump command*, *include command*, *fix print command*, *print command*

Default:

none

1.14.116 write_grid command

Syntax:

```
write_grid mode file
```

- mode = *parent* or *geom*
- file = name of file to write grid info to

Examples:

```
write_grid parent data.grid
write_grid geom viz.out
```

Description:

Write a grid file in text format describing the currently defined hierarchical grid. See the *read_grid* and *create_grid* commands for a definition of hierarchical grids and parent/child cells as used by SPARTA.

The file is written in text format in one of two modes.

If *mode* is *parent* then a list of parent cells is written in the same format as the input file used by the *read_grid* command. Thus the file can be used to start a subsequent simulation using the same grid topology.

If *mode* is *geom* then the geometric description of all the child cells is written in the following format. This file can be used in conjunction with snapshot files of per-grid properties, written by the *dump grid* command, to visualize various properties on the grid.

```
Description line

N points
M cells

Points

1 x y z
2 x y z
...
N x y z

Cells

1 p1 p2 p3 p4 ...
2 p1 p2 p3 p4 ...
...
M p1 p2 p3 p4 ...
```

The file will have N points and M grid cells. For each point the x,y,z coordinates are output. For each grid cell, the indices of the 4 (in 2d) or 8 (in 3d) points comprising the corners of the grid cell are output. Each point index is an integer from 1 to N. The ordering of the point indices is (LL,LR,UR,UL) or counter-clockwise for 2d grid cells. For 3d grid cells it is the same where the first 4 indices are the lower-Z indices, and the next 4 are the upper-Z indices.

Important: The points in the output file will not be unique. Instead there will be 4 or 8 for each grid cell, with some (x,y,z) coordinates being duplicated since they are shared by multiple grid cells. Converting the output file to one with

a unique list of points is currently a post-processing task.

Restrictions:

none

Related commands:

read_grid command create_grid command

Default:

none

1.14.117 write_isurf command

Syntax:

```
write_isurf group-ID Nx Ny Nz filename ablateID keyword args ...
```

- group-ID = group ID for which grid cells store the implicit surfs
- Nx,Ny,Nz = grid cell extent of the grid cell group
- filename = name of file to write grid corner point info to
- ablateID = ID of the *fix ablate* command which stores the corner points
- zero or more keyword/args pairs may be appended
- keyword = *precision*

```
precision arg = int or double
```

Examples:

```
write_isurf block 100 100 200 isurf.material.* ablation
```

Description:

Write a grid corner point file in binary format describing the current corner point values which define the current set of implicit surface elements. See the *read_isurf* command for a definition of implicit surface elements and how they are defined from grid conner point values. The surface file can be used for later input to a new simulation or for post-processing and visualization.

The specified *group-ID* is the name of a grid cell group, as defined by the *group grid* command, which contains a set of grid cells, all of which are the same size, and which comprise a contiguous 3d array, with specified extent *Nx* by *Ny* by *Nz*. These should be the same parameters that were used by the *read_isurf* command, when the original grid corner point values were read in and used to define a set of implicit surface elements. For 2d simulations, *Nz* must be

specified as 1, and the group must comprise a 2d array of cells that is *Nx* by *Ny*. These are the grid cells that contain implicit surfaces.

Similar to *dump* files, the *filename* can contain a "*" wildcard character. The "*" character is replaced with the current timestep value. For example isurf.material.0 or isurf.material.100000.

The specified *ablateID* is the fix ID of a *fix ablate* command which has been previously specified in the input script for use with the *read_isurf* command and (optionally) to perform ablation during a simulation. It stores the grid corner point values for each grid cell.

The output file is written in the same binary format as the *read_isurf* command reads in.

Restrictions:

none

Related commands:

read_isurf command

Default:

The optional keyword default is precision double.

1.14.118 write_restart command

Syntax:

```
write_restart file keyword value ...
```

- file = name of file to write restart information to
- zero or more keyword/value pairs may be appended
- keyword = *fileper* or *nfile*

```
fileper arg = Np
  Np = write one file for every this many processors
nfile arg = Nf
  Nf = write this many files, one from each of Nf processors
```

Examples:

```
write_restart restart.equil
write_restart restart.equil.mpiio
write_restart flow.%.* nfile 10
```

Description:

Write a binary restart file with the current state of the simulation.

During a long simulation, the *restart* command can be used to output restart files periodically. The write_restart command is useful at the end of a run or between two runs, 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. The optional *fileper* and *nfile* keywords discussed below can alter the number of files written.

Restart files can be read by a *read_restart* command to restart a simulation from a particular state. Because the file is binary, it may not be readable on another machine.

IMPORTANT NOTE: Although the purpose of restart files is to enable restarting a simulation from where it left off, not all information about a simulation is stored in the file. For example, the list of fixes that were specified during the initial run is not stored, which means the new input script must specify any fixes you want to use. See the *read_restart* command for details about what is stored in a restart file.

The optional *nfile* or *fileper* keywords can be used in conjunction with the "%" wildcard character in the specified restart file name. As explained above, the "%" character causes the restart file to be written in pieces, one piece for each of P processors. By default P = the number of processors the simulation is running on. The *nfile* or *fileper* keyword can be used to set P to a smaller value, which can be more efficient when running on a large number of processors.

The *nfile* keyword sets P to the specified Nf value. For example, if Nf = 4, and the simulation is running on 100 processors, 4 files will be written, by processors 0,25,50,75. Each will collect information from itself and the next 24 processors and write it to a restart file.

For the *fileper* keyword, the specified value of Np means write one file for every Np processors. For example, if Np = 4, every 4th processor (0,4,8,12,etc) will collect information from itself and the next 3 processors and write it to a restart file.

Restrictions:

none

Related commands:

restart command read_restart command

Default:

none

1.14.119 write_surf command

Syntax:

```
write_surf file
```

- file = name of file to write surface element info to
- zero or more keyword/args pairs may be appended
- keyword = *points* or *fileper* or *nfile*

```
points arg = yes or no to include a Points section in the file
fileper arg = Np
  Np = write one file for every this many processors
nfile arg = Nf
  Nf = write this many files, one from each of Nf processors
```

Examples:

```
write_surf data.surf
write_surf data.surf points no
write_surf data.surf.% nfile 50
```

Description:

Write a surface file in text format describing the currently defined surface elements, whether they be explicit or implicit surfaces. See the *read_surf* and *read_isurf* commands for a definition of surface elements and how they are defined and used be SPARTA. The surface file can be used for later input to a new simulation or for post-processing and visualization.

Note that if surface objects were clipped when read in by the *read_surf* command then some surface elements may have been deleted and new ones created. Likewise for the points that define the end points or corner points of surface element lines (2d) or triangles (3d). Similarly, if surface elements have been removed by the *remove_surf* command, then points may have also been deleted. In either case, surface points and elements are renumbered by these operations to create compressed, contiguous lists. These lists are what is output by this command.

The file is written as a text file in the same format as the *read_surf* command reads in. Note that a Points section is optional. If the *points* keyword is specified with a value of *yes*, then a Points section is included in the file. If the value is *no*, then point coordinates are included with individual lines or triangles.

Similar to *dump* files, the surface 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, i.e. just the header information for the number of points and lines or triangles, as described on the *read_surf* doc page.

For example, the files written for filename data.% would be data.base, data.0, data.1, . . . , data.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. The optional *fileper* and *nfile* keywords discussed below can alter the number of files written.

Note that implicit surfaces read in by the *read_isurf* command can be written out by the write_surf command, e.g. for visualization purposes. But they cannot be read back in to SPARTA via the *read_isurf* command, because write_surf creates files in an explicit surface format. See the *Howto 6.13* section of the manual for a discussion of explicit and

implicit surfaces for an explantion of explicit versus implicit surfaces as well as distributed versus non-distributed storage. You cannot mix explicit and implicit surfaces in the same simulation.

The optional *nfile* or *fileper* keywords can be used in conjunction with the "%" wildcard character in the specified surface file name. As explained above, the "%" character causes the surface file to be written in pieces, one piece for each of P processors. By default P = the number of processors the simulation is running on. The *nfile* or *fileper* keyword can be used to set P to a smaller value, which can be more efficient when running on a large number of processors.

The *nfile* keyword sets P to the specified Nf value. For example, if Nf = 4, and the simulation is running on 100 processors, 4 files will be written, by processors 0,25,50,75. Each will collect information from itself and the next 24 processors and write it to a surface file.

For the *fileper* keyword, the specified value of Np means write one file for every Np processors. For example, if Np = 4, every 4th processor (0,4,8,12,etc) will collect information from itself and the next 3 processors and write it to a surface file.

Restrictions:

none

Related commands:

read_surf command read_isurf command

Default:

The default is points = yes.

- Command index

BIBLIOGRAPHY

[Koura92] K. Koura and H. Matsumoto, "Variable soft sphere molecular model for air species," Phys Fluids A, 4, 1083 (1992).

[Gallis04] M. A. Gallis, J. R. Torczynski, and D. J. Rader, "Molecular gas dynamics observations of Chapman-Enskog behavior and departures therefrom in nonequilibrium gases," Phys Rev E, 69, 042201 (2004).

[Gallis11] M. A. Gallis, J. R. Torczynski, "Effect of Collision-Partner Selection Schemes on the Accuracy and Efficiency of the Direct Simulation Monte Carlo Method," International Journal for Numerical Methods in Fluids, 67(8):1057-1072. DOI:10.1002/fld.2409 (2011).

[Fang02] Y. Fang and W. W. Liou, Microfluid Flow Computations Using a Parallel DSMC Code, AIAA 2002-1057. (2002).

[Bird09] G. A. Bird, Chemical Reactions in DSMC Rarefied Gas Dynamics, Editor T Abe, AIP Conference Proceedings (2009).

[Bird11] G. A. Bird, "The Q-K model for gas-phase chemical reaction rates", Physics of Fluids, 23, 106101, (2011).

[Gallis09] M. A. Gallis, R. B. Bond, and J. R. Torczynski, "A Kinetic-Theory Approach for Computing Chemical-Reaction Rates in Upper-Atmosphere Hypersonic Flows", J Chem Phys, 131, 124311, (2009).

[Gallis10] M. A. Gallis, R. B. Bond, and J. R.Torczynski, "Assessment of Collision-Energy-Based Models for Atmospheric-Species Reactions in Hypersonic Flows", J Thermophysics and Heat Transfer, (2010).

[Bird94] G. A. Bird, Molecular Gas Dynamics and the Direct Simulation of Gas Flows, Clarendon Press, Oxford (1994).

[Cercignani71] Cercignani C, Lampis M, Kinetic models for gas-surface interactions, Transport theory and statistical physics, Jan (1971).

[Lord90] R. G. Lord, presented at the 17th International Symposium on Rarefied Gas Dynamics, Germany, July (1990).

[Lord91] R. G. Lord, Some extensions of the Cercignani-Lampis gas-surface interaction model, Physics of Fluids A: Fluid Dynamics, Jan (1991).

[SG18] K. Swaminathan Gopalan, Development of a detailed surface chemistry framework in DSMC, AIAA Aerospace Sciences Meeting, Jan (2018).

[Rettner94a] C. T. Rettner, Reaction of an H-atom beam with Cl/Au(111): Dynamics of concurrent EleyRideal and Langmuir-Hinshelwood mechanisms, Journal of Chemical Physics, (1994).

[Alexander12] W. A. Alexander, *et al*, Kinematics and dynamics of atomic-beam scattering on liquid and self-assembled monolayer surfaces, Faraday discussions, (2012)

[Glatzer97] D. Glatzer, *et al*, Rotationally excited NO molecules incident on a graphite surface: in- and out-of-plane angular distributions, Surface Science, (1997)

[Lord95] R. G. Lord, Some further extensions of the Cercignani-Lampis gas-surface interaction model, Physics of Fluids, May (1995).

[Goodman72] F. O. Goodman, Simple model for the velocity distribution of molecules desorbed from surfaces following recombination of atoms, Surface Science, (1972).

[Rettner94b] C. T. Rettner and J. Lee, Dynamic displacement of o2 from pt (111): A new desorption mechanism, The Journal of chemical physics, (1994).

[Beckerle90] J. Beckerle, A. Johnson, and S. Ceyer, Collision-induced desorption of physisorbed CH4 from Ni (111): Experiments and simulations, The Journal of Chemical Physics, (1990).

[Goodman74] F. O. Goodman, Determination of characteristic surface vibration temperatures by molecular beam scattering: Application to specular scattering in the H-LiF (001) system, Surface Science, (1974)

INDEX