SPARTA Users Manual

name of code

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

Table of Contents

Table of Contents

Table of Contents

Table of Contents

SPARTA Documentation

14 Feb 2014 version

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.
- There is also a Developer.pdf file in the doc directory, which describes the internal structure and algorithms of SPARTA. NOTE: this file is not yet available.

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

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

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

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

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

PDF file of the entire manual, generated by htmldoc

1. Introduction

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

1.1 What is SPARTA

SPARTA is a Direct Simulation Montel Carlo code that models rarefied gases, using 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 Section 6 for information on SPARTA performance and scalability, or the Benchmarks section of the SPARTA WWW Site.

SPARTA is a freely-available open-source code, distributed under the terms of the GNU Public License, which means you can use or modify the code however you wish. See Section 1.5 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 8 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.

1.2 SPARTA features

This section highlights SPARTA features, with links to specific commands which give more details. The next section 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 8 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: 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 or at inlet boundaries

Geometry

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

Gas-phase collisions and chemistry

- collisions between pairs of species groups
- VSS (variable soft sphere) or VHS (variable hard sphere) collisions
- TCE chemistry model

Surface collisions

- specular or diffuse

Performance

- grid cell weighting of particles
- 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

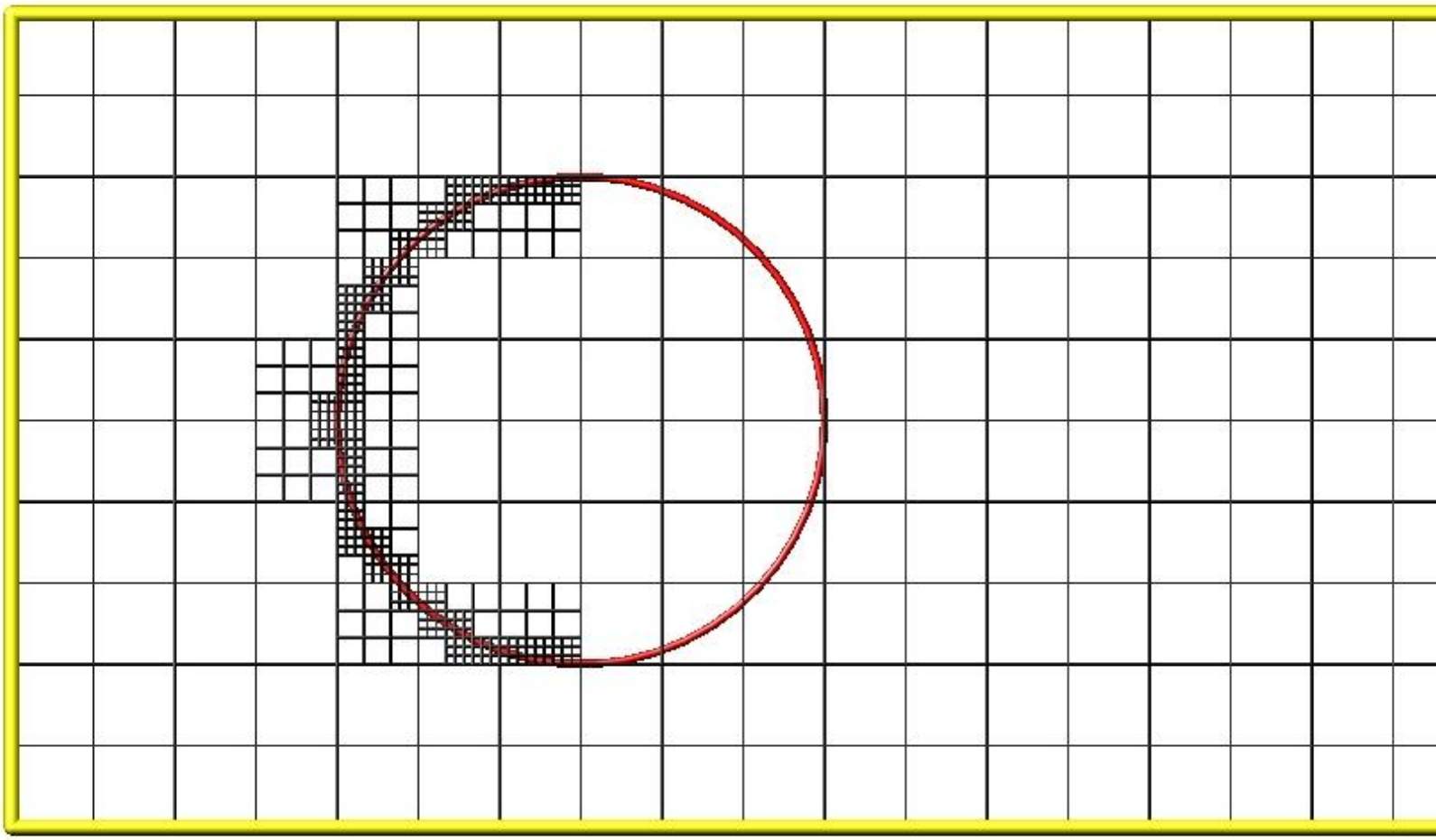
Pre- and post-processing

- Various pre- and post-processing serial tools are packaged with SPARTA; see Section 7 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.

1.3 Grids and surfaces in SPARTA

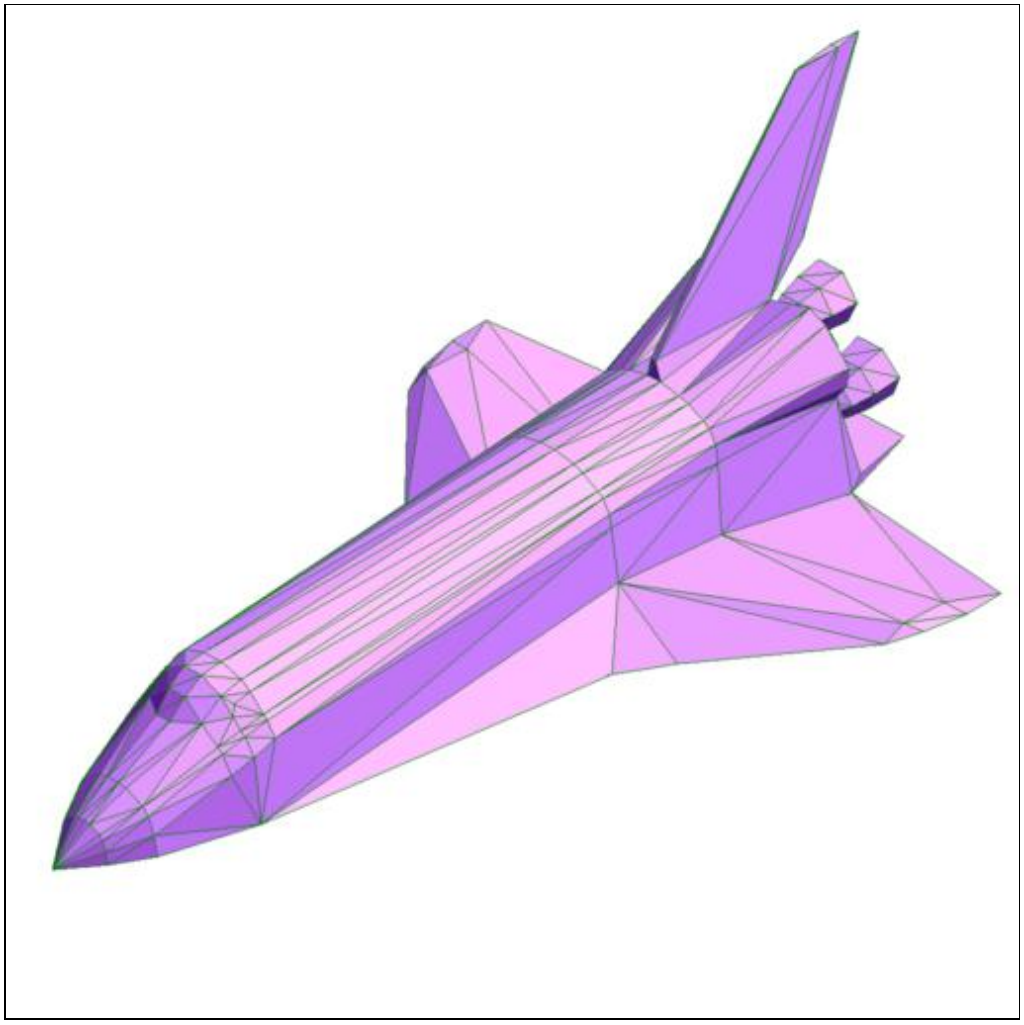
SPARTA overlays a grid over the simulation domain which is used to track particles and to co-locate particles in the same grid cell for performing collision and chemistry operations. SPARTA uses a Cartesian hierarchical grid. Cartesian means that the faces of a grid cell are aligned with the Cartesian 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 Sections 4.9 and 4.10 for more details of both the grids and surface objects that SPARTA supports and how to define them.

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, Section 10.1 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 Section 7. 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. Section 8 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.

1.5 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

2. Getting Started

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

2.1 What's in the SPARTA distribution

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

```
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

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 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, send it to the developers and we'll include it in future SPARTA releases.

Steps to build a SPARTA executable:

Step 0

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

```
make 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.

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 recogized are:

- -DSPARTA_GZIP
- -DSPARTA_JPEG
- -DSPARTA_PNG
- -DLAMMPS_FFMPEG
- -DSPARTA_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 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 STL unordered map class which is not yet supported by all C++ compilers. The default is to use the unordered map class, so only set -DSPARTA_MAP 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 to enable simulation of large models with more than ~2 billion grid cells. 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.

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 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 JPG variables allow you to specify a JPEG and/or PNG library which LAMMPS uses when writing out JPEG or PNG files via the dump image command. These can be left blank if you do not use the -DLAMMPS_JPEG or -DLAMMPS_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 7

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.

Errors that can occur when making SPARTA:

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

Here are two non-obvious errors that can occur:

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

```
make makelist
make -f Makefile.list 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:

(1) Building SPARTA for multiple platforms.

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

(2) Cleaning up.

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

Building for a Mac:

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

2.3 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 Section 4.7 for more info on coupling SPARTA to other codes. See Section 9 for more info on wrapping and running SPARTA from Python.

Static library:

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

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

where foo is the machine name. This kind of library is typically used to statically link SPARTA to a driver application, so that you can insure all dependencies are satisfied at compile time. The first "make" command will create a current Makefile.lib with all the file names in your src dir. The second "make" command will use it to build SPARTA as a static library, using 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.

Shared library:

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

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

where foo is the machine name. This kind of library is required when wrapping SPARTA with Python; see Section_python for details. The first "make" command will create a current Makefile.shlib with all the file names in your src dir. The second "make" command will use it to build SPARTA as a shared library, using the SHFLAGS and SHLIBFLAGS settings in src/MAKE/Makefile.foo. 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 the Python wrapper uses 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, since they are always built as shared libraries with the -fPIC switch. However, if a library like MPI does not exist as a shared library, the second make command will generate an error. This means you will need to install a shared library version of the package. The build instructions for the library should tell you how to do this.

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. So 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
```

2.4 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
```

Section 3 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
```

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 Section 10 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.

2.5 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
- -l or -log
- -p or -partition
- -pl or -plog
- -ps or -pscreen
- -sc or -screen
- -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.

`-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 simulatoins from one input script, using multiple partitions, see Section 4.3 of the manual. World- and universe-style variables are useful in this context.

`-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.

```
-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 3.2 for more info on using variables in input scripts.

IMPORTANT NOTE: 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.

2.6 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:

```
Loop time of 36.3787 on 8 procs for 100 steps with 10000000 particles

Particle moves    = 1000000000 (1B)
Cells touched     = 1395597726 (1.4B)
Particle comms    = 3990284 (3.99M)
Boundary collides = 4003287 (4M)
Boundary exits    = 0 (0K)
SurfColl checks   = 0 (0K)
SurfColl occurs   = 0 (0K)
Collide attempts  = 93689476 (93.7M)
Collide occurs    = 70166679 (70.2M)
Particles stuck   = 0

Particle-moves/CPUsec/proc: 3.43607e+06
Particle-moves/step: 1e+07
Cell-touches/particle/step: 1.3956
Particle comm iterations/step: 1
Particle fraction communicated: 0.00399028
Particle fraction colliding with boundary: 0.00400329
Particle fraction exiting boundary: 0
Surface-checks/particle/step: 0
Surface-collisions/particle/step: 0
Collision-attempts/particle/step: 0.0936895
Collisions/particle/step: 0.0701667

Move  time (%) = 16.6921 (45.8842)
Coll  time (%) = 11.9058 (32.7275)
```

```
Sort  time (%) = 7.24376 (19.9121)
Comm  time (%) = 0.17353 (0.47701)
Outpt time (%) = 0.363323 (0.998723)
Other time (%) = 0.000194222 (0.000533888)

Particles: 1.25e+06 ave 1.25283e+06 max 1.24835e+06 min
Histogram: 3 1 0 0 1 1 1 0 0 1
Cells:     125000 ave 125000 max 125000 min
Histogram: 8 0 0 0 0 0 0 0 0 0
GhostCell: 15608 ave 15608 max 15608 min
Histogram: 8 0 0 0 0 0 0 0 0 0
EmptyCell: 7957 ave 7957 max 7957 min
Histogram: 8 0 0 0 0 0 0 0 0 0
```

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

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 statistics.

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

The next to last section gives a breakdown of the CPU timing (in seconds) in 6 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 percentage of CPU time for each category is shown in parenthesis.

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..

what are empty cells?

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

3. Commands

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

3.1 SPARTA input script

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

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

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

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

does something different than this sequence:

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

In the first case, the specified timestep (0.5 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 10 gives more information on what errors mean. The documentation for each command lists restrictions on how the command can be used.

3.2 Parsing rules

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

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

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

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

(3) The line is searched repeatedly for $ characters, which indicate variables that are replaced with a text string. See an exception in (6).

If the $ is followed by curly brackets, then the variable name is the text inside the curly brackets. If no curly brackets follow the $, then the variable name is the single character immediately following the $. Thus ${myTemp} and $x refer to variable names "myTemp" and "x".

How the variable is converted to a text string depends on what style of variable it is; see the variable 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 variable command 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 or if commands for examples. A "#" or "$" character that is between quotes will not be treated as a comment indicator in (2) or substituted for as a variable in (3).

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

3.3 Input script structure

This section describes the structure of a typical SPARTA input script. The "examples" directory in the SPARTA distribution contains sample input scripts; the corresponding problems are discussed in Section 5, 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, units, and seed.

(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 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.

(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.

3.4 Commands listed by category

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

Initialization:

dimension, seed, units

Problem definition:

boundary, bound_modify, create_box, create_grid, create_particles, mixture, read_grid, read_surf, read_restart, species,

Settings:

collide, collide_modify, compute, fix, global, react, region, surf_collide, surf_modify, timestep, uncompute, unfix

Output:

dump, dump_image, dump_modify, restart, stats, stats_modify, stats_style, undump, write_grid, write_restart

Actions:

balance_grid, run

Miscellaneous:

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

---

3.5 Individual commands

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

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
if	include	jump	label	log	mixture
next	partition	print	quit	react	read_grid
read_restat	read_surf	region	reset_timestep	restart	run
seed	shell	species	stats	stats_modify	stats_style
surf_collide	surf_modify	timestep	uncompute	undump	unfix
units	variable	write_grid	write_restart		

Fix styles

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

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

Compute styles

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

boundary	grid	ke/particle	property/grid	reduce	sonine/grid
surf	temp	tvib/grid			

Collide styles

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

vss

Surface collide styles

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

diffuse	specular

4. 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.

NOTE: add restarting a simulation, like in LAMMPS

The example input scripts included in the SPARTA distribution and highlighted in Section 5 of the manual also show how to setup and run various kinds of simulations.

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.

4.2 Axisymmetric simulations

This section will be filled in when axisymmetry is fully implemented in SPARTA.

- create box with ylo = 0.0 and zlo/zhi = 0.5
- ylo boundary with "a" setting
- cell weighting options

4.3 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 variables and the next and jump commands 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 ${rho}
compute myGrid grid all n temp
dump 1 all grid 1000 dump.$a id c_myGrid
run 100000
clear
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 Section 2.5 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.

4.4 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 and fix commands 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 Section 10) 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
- Scalar/vector/array data
- Statistical output
- Dump file output
- Fixes that write output files
- Computes that process output quantities
- Computes that generate values to output
- Fixes that generate values to output
- Variables that generate values to output
- Summary table of output options and data flow between commands

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, and stats_modify commands. 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 or fix or variable 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 and dump_modify commands. 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 or fix or variable 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 and fix ave/surf commands.

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 ararys from compute or fix commands. 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 ararys from compute or fix commands. 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	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

4.5 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.

4.6 Library interface to SPARTA

As described in Section 2.3, 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 arguments 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 NOTE: The examples/COUPLE dir has not been added to the distribution yet.

4.7 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 Section 8 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 Section_modify 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 NOTE: 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 4.6 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.

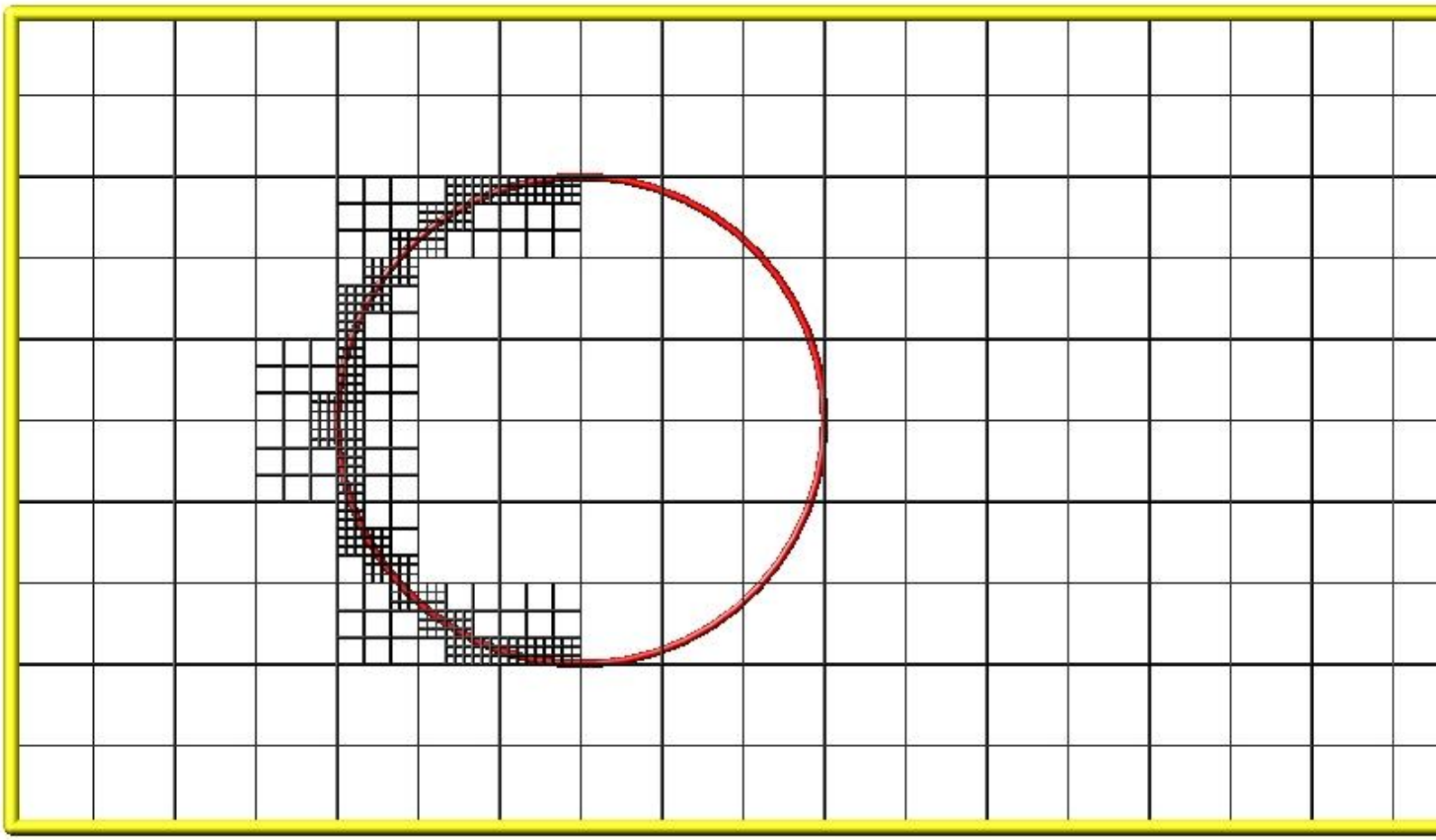
4.8 Details of grid geometry in SPARTA

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.



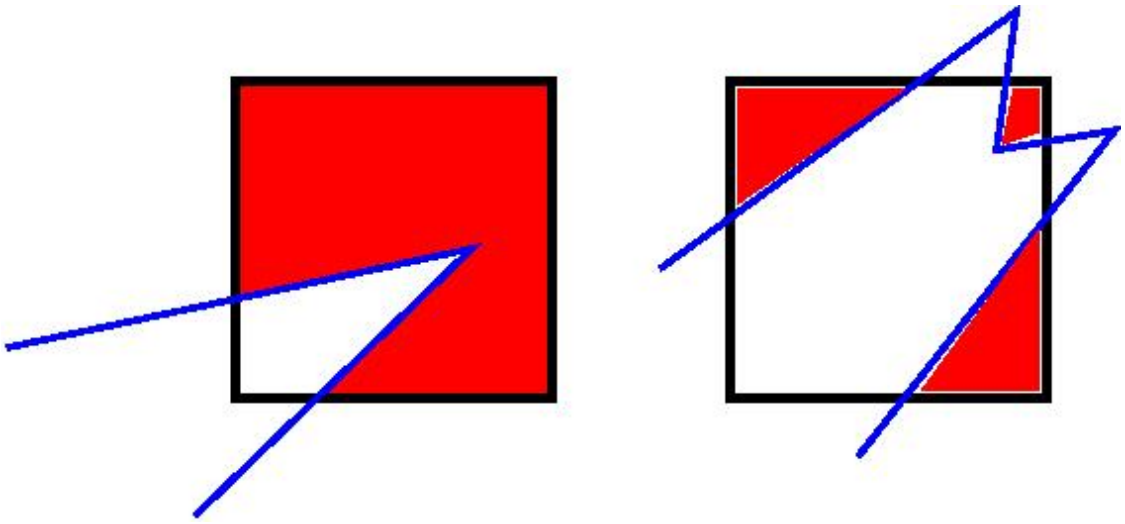
In the rest of the SPARTA manual, the following terminology is used to refer to the cells of the hierarchical grid. The flow region is the portion of the simulation domain that is "outside" any surface objects and is typically filled with particles.

- 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 so that one portion of the cell is in the flow region
- split cell = a child cell intersected by two or more surface elemments so that two or more disjoint portions of the cell are in the flow region
- sub cell = one disjoint portion of a split cell in the flow region

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. Child cells can thus become 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. 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 grid command, using its *id* and *idstr* attributes. The integer form can also be output by the 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) 1st-level cells 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 1000) 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 = 33142. 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 Section 2.2.2.

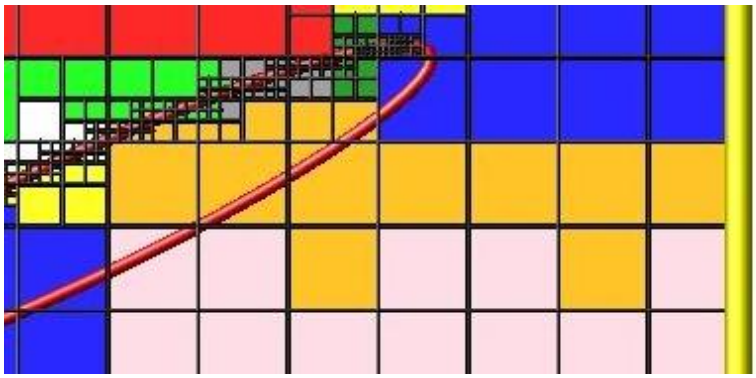
For 32-bit cell IDs if every level uses a 2x2x2 sub-division which requires 4 bits (to store values from 1 to 8), then a grid can have 7 levels. For 64-bit cell IDs, 15 levels could be defined.

The create_grid and balance and fix balance commands 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.

4.9 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 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 and surf_collide commands.

As described in the previous Section 4.8, 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.

5. 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.foo.P means it ran on P processors of machine "foo".

If the "dump image" lines in each script are uncommented, a series of JPG 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.

free: free molecular flow in a box collide: collisional flow in a box step: flow around a step circle flow around a sphere spiky: flow around a spiky circle

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 optionall 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*jpg 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.

5. 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).

7. 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
- slog - read log files and extract columns of data
- vcr - VCR-style GUI for 3d interactive OpenGL visualization of dump or surface files

The sdata and slog tools are included in the SPARTA distribution, as discussed below, in the pizza section, so that they can be used by other provided Python scripts.

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

- dump2cfg - convert a particle dump file to CFG format
- dump2xyz - convert a particle dump file to XYZ format
- log2txt - extract columns of info from a log file
- logplot - plot columns of info from a log file via GnuPlot
- pizza - Pizza.py tools used by other tools

---

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.

---

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.

---

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.

---

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.

---

pizza tool

The pizza sub-directory contains several of the tools available in the Pizza.py toolkit. This is so that other Python scripts in the tools directory can use them directly.

See the tools/README file for info on how to set an environment variable so that the Python scripts can find the Pizza.py tool files.

8. 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.

8.1 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

8.2 Fix styles

Fix style commands perform operations during the timestepping loop of a simulation. They 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 produce output of various kinds, similar to compute commands.

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
memory_usage	tally memory usage

8.3 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

---

8.4 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

---

8.5 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

8.6 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

8.7 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

8.8 Input script commands

New commands can be added to SPARTA that will be recognized in input scripts. For example, the create_particles, read_surf, and run commands 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.

9. 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 Section 4.7 of the manual and the examples/COUPLE directory of the distribution for more ideas about coupling SPARTA to other codes. See Section 2.3 about how to build SPARTA as a library, and Section 4.6 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 NOTE: 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.

9.1 Building SPARTA as a shared library

Instructions on how to build SPARTA as a shared library are given in Section 2.3. 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".

From the src directory, type

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

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 Section 2.3 for more details, especially if your SPARTA build uses auxiliary libraries like MPI which may not be built as shared libraries on your system.

9.2 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.

9.3 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 NOTE: 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.

9.4 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 Section 2.3 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.

9.5 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

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 NOTE: 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 Section 4.4 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?

9.6 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.

10. Errors

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

10.1 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.

10.2 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.

10.3 Error & warning messages

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

ERROR: Illegal 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 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 pre-defined.

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 %d\n
: 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 %d\n
: 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 cell ID in grid file

A cell ID could not be converted into numeric format.

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.

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.

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.

11. 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.

11.1 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.

11.2 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 it's initial open-source release.

balance_grid command

Syntax:

```
balance_grid keyword args ...
```

- keyword = *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
      Px,Py,Pz = # of processors in each dimension
    random args = none
    proc args = none
    rcb args = weight
      weight = cell or part
```

Examples:

```
balance_grid block * * *
balance_grid block * 4 *
balance_grid clump yxz
balance_grid random
balance_grid rcb part
```

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* 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* and *proc* keywords will

produce dispersed assignments of child cells to each processor.

IMPORTANT NOTE: See Section 5.8 of the manual for an explanation of clumped and dispersed grid cell assignments and their relative performance trade-offs.

The *none* keyword 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* keywords 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.

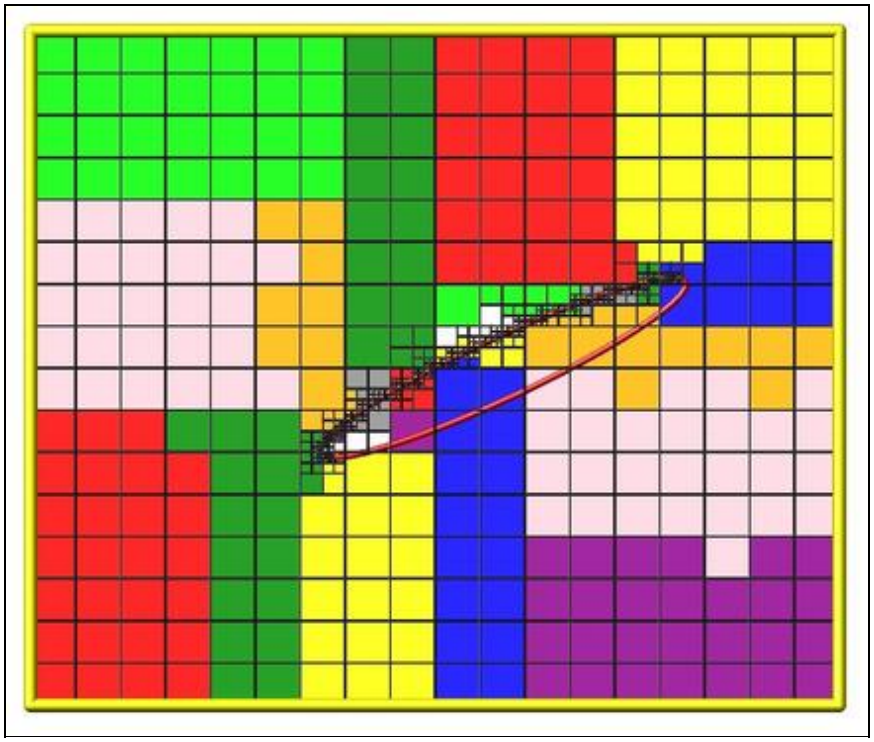
The meaning of the *stride*, *clump*, and *block* keywords is exactly the same as when they are used with the create_grid command. See its doc page for details.

The *random* keyword 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* 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 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* keyword 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*, than 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.

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.



Restrictions:

This command can only be used after the grid has been created by the create_grid, read_grid, or 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

Default: none

bound_modify command

Syntax:

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

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

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

Examples:

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

Description:

Set parameters for one of the boundaries of the global simulation box. Any of the 6 faces can be selected via the *wall* setting.

The *surf* keyword can only be used when the boundary is of type "s", for surface, as set by the boundary command. This keyword assigns a surface collision model to the boundary, as defined by the surf_collide command. 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.

Restrictions:

For 2d simulations, the *zlo* and *zhi* boundaries cannot be modified by this command, since they are always periodic.

Related commands:

boundary, surf_modify

Default: none

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.

IMPORTANT NOTE: Axi-symmetry is not yet fully implemented in SPARTA, so style *a* cannot yet be used.

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, surf_collide

Default:

```
boundary p p p
```

clear command

Syntax:

```
clear
```

Examples:

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

Description:

This command deletes all atoms, restores all settings to their default values, and frees all memory allocated by SPARTA. Once a clear command has been executed, it is 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

collide command

Syntax:

```
collide style args
```

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

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

Examples:

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

Description:

Define what style of particle-particle collisions will be performed by SPARTA 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 calcualted 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)).

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. (see chapter 2.6 and 2.7 in (Bird94)).

In DSMC, the variable-soft-sphere (VSS) interaction of Koura and Matsumoto and the variable-hard-sphere (VHS) interaction of Bird 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 file which contains definitions of VSS model parameters for some number of species. It 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 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 properties separated by whitespace:

```
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 the 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).

Restrictions: none

Related commands:

collide_modify, mixture, react

Default:

style = none

(Koura92) K. Koura and H. Matsumoto, "Variable soft sphere molecular model for air species," Phys Fluids A, 4, 1083 (1992).

(Bird94) G. A. Bird, Molecular Gas Dynamics and the Direct Simulation of Gas Flows, Clarendon Press, Oxford (1994).

(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).

collide_modify command

Syntax:

```
collide_modify keyword values ...
```

- one or more keyword/value pairs may be listed
- keywords = *vremax* or *remain*

```
  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
    rotate value = no or yes
    vibrate value = no or discrete or smooth
```

Examples:

```
collide_modify vremax 1000 yes
collide_modify vremax 0 no remain no
```

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 *rotate* keyword determines how vibrational 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 *yes*, a particle's rotational energy is a 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 *discrete*, each particle's vibrational energy is set to discrete values, namely multiples of kT where k = the Boltzmann constant and T is the characteristic vibrational temperature set for the particle's species in the file read by the species command. If the value is set to *smooth*, a particle's vibrational energy is a continuous value. Note that the *discrete* setting is only observed if the vibrational degrees of freedom for the species is 2, as set in the file read by the species command, i.e. for species that are dimer molecules. Species with more degrees of freedom (4,6,etc) are always treated as if the setting were *smooth*.

Restrictions: none

Related commands:

collide

Default:

The option defaults are vremax = (0,yes), remain = yes, and vibrate = none.

compute command

Syntax:

```
compute ID style args
```

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

Examples:

```
compute 1 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 8 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
- grid - various per grid cell quantities
- ke/particle - temperature per particle
- property/grid - per grid cell properties
- reduce - reduce vectors to scalars
- sonine/grid - Sonine moments per grid cell
- surf - various per surface element quantities
- temp - temperature of particles
- tvib/grid - vibrational temperature per grid cell

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

Default: none

compute boundary command

Syntax:

```
compute ID boundary 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 *press* or *shx* or *shy* or *shz* or *ke*

```
n = count of particles hitting boundary
press = magnitude of normal pressure on boundary
shx,shy,shz = components of shear stress on boundary
ke = flux of kinetic energy on boundary
erot = flux of rotational energy on boundary
evib = flux of vibrational energy on boundary
etot = flux of total energy on boundary
```

Examples:

```
compute 1 boundary all n press eng
compute mine boundary species press shx shy shz
```

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 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 strike that 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.

The manner in which statistics are tallied for each boundary depends on the style of boundary as specified by the boundary command. For *outflow* boundaries, only the flux of particles leaving the simulation box contributes to the pressure, stress, and energy; there is no contribution from reflected particles. For *periodic* boundaries, only the count *n* is tallied, since the particles exert no pressure or stress on the boundary. For *specular* boundaries, the particle collisions exert pressure and stress on the boundary but no energy flux, since the collisions conserve energy. For *surface* boundaries, each collision may contribute to all the values, depending on the surface collision model.

The *n* value counts the number of particles in the group striking the boundary or passing through it.

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 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, 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 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 4.4 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.

Restrictions: none

Related commands:

fix ave/time

Default: none

compute grid command

Syntax:

```
compute ID grid mix-ID value1 value2 ...
```

- ID is documented in compute command
- grid = style name of this compute command
- mix-ID = mixture ID to perform calculation on
- one or more values can be appended
- value = *n* or *ndensity* or *mass* or *u* or *v* or *w* or *usq* or *vsq* or *wsq* of *ke* or *temp* or *erot* or *trot*

```
n = particle count
  ndensity = number density
  mass = mass
  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
```

Examples:

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

Description:

Define a computation that calculates one or more values for each grid cell, based on the particles in the 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.

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. The answer is then divided by the number of sampling timesteps if it is not otherwise normalized by the number of particles. 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 timesteps.

The *n* value counts the number of particles in the group.

The *ndensity* value computes the number density of particles in the group:

Ndensity = N * fnum / volume

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.

The *mass* value computes the average mass of particles in the group:

```
Mass = Sum_i (mass_i) / N
```

where Sum_i is a sum over particles in the grid cell and in the group.

The *u*, *v*, *w* values compute the components of the average velocity of particles in the group:

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

The *usq*, *vsq*, *wsq* values compute the squared component of the average velocity of particles in the group.

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

The *ke* value computes the average kinetic energy of particles in the group:

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

The *temp* value first computes the average kinetic energy of particles in the 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 sonine/grid command to calculate thermal temperatures after subtracting out streaming components of velocity.

The *erot* value computes the average rotational energy of particles in the group:

```
Erot = Sum_i (erot_i) / N
```

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 the group:

```
Erot = Sum_i (evib_i) / N
```

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 4.8 of the manual gives details of how SPARTA defines child, unsplit, split, and sub cells. Note that split cells and unsplit cells inside closed surfaces contain no particles. Thus they will compute a zero result for all the individual 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 the units appropriate to the individual values as described above. *N* is unitless. *Ndensity* is in 1/distance^3 units for 3d simulations and 1/distance^2 units for 2d simulations. *Mass* is in mass units. *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* are in temperature units.

Restrictions: none

Related commands:

fix ave/grid, dump grid

Default: none

compute ke/particle 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 Section 4.4 for an overview of SPARTA output options.

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

Restrictions: none

Related commands:

dump particle

Default: none

compute property/grid command

Syntax:

```
compute ID property/grid input1 input2 ...
```

- ID is documented in compute command
- property/grid = style name of this compute command
- 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 all property/grid id xc yc zc
```

Description:

Define a computation that simply stores grid attributes for each grid cell. 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.

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 4.8 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.

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.

Restrictions: none

Related commands:

dump grid, compute reduce, fix ave/grid

Default: none

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*
- 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
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
v_name = per-particle vector calculated by a particle-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
```

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 variables generate per-particle 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.

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 particle-style variable.

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.

IMPORTANT NOTE: 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 compute reduce 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 compute reduce 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 particle-style variable. Particle-style variables can reference stats keywords and various per-particle attributes, or invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of generating per-particle 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 4.4 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, fix, variable

Default: none

compute sonine/grid command

Syntax:

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

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

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

Examples:

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

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

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

Description:

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

The results of this compute are 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. The answer is then divided by the number of sampling timesteps if it is not otherwise normalized by the number of particles. 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 timesteps.

The *thermal* keyword computes the average thermal kinetic energy for all N particles in the group. This is converted to a temperature T by the following formula where kB is the Boltzmann factor:

```
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
thermal_KE = Sum_i (1/2 mass_i Csq_i) / N
T = thermal_KE / (3/2 kB)
```

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

The *a* keyword calculates the aveage of one or more sonine A moments for all particles in the 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 *thermal* equations above. The number of moments computed is specified by the *order* value, e.g. for order = 3, the first 3 moments are computed.

The *b* keyword calculates the averageof one or more sonine B moments for all particles in the 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 *thermal* equations above. The number of moments computed is specified by the *order* value, , e.g. for order = 3, the first 3 moments are computed.

Output info:

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

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 split cells and unsplit cells inside closed surfaces contain no particles. Thus they will compute a zero result for all the individual 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 the units appropriate to the individual values as described above. *Thermal* is in temperature units. A and B are in units like velocity cubed or velocity to the 6th power.

Restrictions: none

Related commands:

fix ave/grid, dump grid

Default: none

compute surf command

Syntax:

```
compute ID surf 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 *press* or *px* or *py* or *pz* or *shx* or *shy* or *shz* or *ke*

```
n = count of particles hitting 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 n press eng
compute mine surf 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 boundary species n press shx shy shz fix 1 ave/surf 10 100 1000 c_1 dump 1 surf 1000 tmp.surf id f_1

Description:

Define a computation that calculates one or more values for each surface element, based on the particles that strike 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.

Surface elements are triangles for 3d simulations and line segments for 2d simulatoins. See the read_surf command for details.

The results of this compute are 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 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 timesteps. However, for the current values below, the two normalization methods are the same.

The *n* value counts the number of particles in the group striking the surface element.

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 is the change in momentum of a particle, whose velocity changes from V_pre to V_post when colliding with the surface element. 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, 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 *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.

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.

The array can be accessed by any command that uses per-surf values from a compute as input. See Section 4.4 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 presure 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/surf, dump surf

Default: none

compute temp command

Syntax:

```
compute ID temp
```

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

Examples:

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

Description:

Define a computation that calculates the temperature of all particles.

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

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.

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 4.4 for an overview of SPARTA output options.

The scalar value will be in temperature units.

Restrictions: none

Related commands: none

Default: none

compute tvib/grid command

Syntax:

```
compute ID tvib/grid mix-ID
```

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

Examples:

compute 1 tvib/grid species

Description:

Define a computation that calculates the vibrational temperature for each grid cell, based on the particles in the cell. A temperature is calculated for each group of species in the specified mixture. See the mixture command for how a set of species can be partitioned into groups. See the compute grid command for other energy and temperature related values that can be calculated on a per-grid-cell basis.

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. The answer is then divided by the number of sampling timesteps if it is not otherwise normalized by the number of particles. 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 timesteps.

The vibrational temperature for a group of particles comprised of different species is defined as a weighted average as follows:

```
T_group = (T1*N1 + T2*N2 + ...) / (N1 + N2 + ...)
```

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 in the group is defined as follows

```
Tsp = (2/kB) Sum_i (evib_i) / N (2 Ibar ln(1 + 1/Ibar))
Ibar = Sum_i (e_vib_i) / (N kB Theta)
```

where e_vib is the 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.

Note that the collide_modify vibrate command affects how vibrational energy is treated in particle collisions and stored by particles. It thus also affects the vibrational temperature calculated by this compute.

Output info:

This compute calculates a per-grid array, with the number of columns equal to the number of groups in the specified mixture.

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 split cells and unsplit cells inside closed surfaces contain no particles. Thus they will compute a zero result for all the individual 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

Default: none

create_box command

Syntax:

```
create_box xlo xhi ylo yhi zlo zhi

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

Examples:

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

Description:

Set the size of the simulation box.

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

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 *stride* or *clump* or *block* or *random*

```
  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
    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
```

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 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. 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 NOTE: See Section 5.8 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.

As mentioned above, a single-level grid is defined by specifying only the arguments *Nx*, *Ny*, *Nz*, with no additional levels. 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. 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 2.5 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.

If a hierarchical grid with more than one level is desired, it can be defined using the *level* keyword one or more times with *Nlevel* in ascending order, starting with *Nlevel* = 2. The other keywords *stride*, *clump*, *block*, or *random* cannot be used with a hierarchical grid. 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.

The Px, Py, Pz arguments specify which cells in the previous level are flagged as parents and partitioned to create cells at the new level. 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 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.

The next-to-last example above creates a 2-level grid. 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 final grid in this example thus has 1001 parent cells (the simulation box plus the 1000 level 1 cells), and 12000 child cells.

The last example above creates a 3-level grid. The 1st level is 8x8x10. The 2nd 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 final grid in the last example 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.

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 a maximum of 7 levels can be used for 32-bit IDs and 15 levels for 64-bit IDs.

Related commands:

create_box, read_grid

Default:

create_particles 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*

```
  global value = yes or no
```

Examples:

```
create_particles background
create_particles air n 100000
create_particles air n 100000 global yes
create_particles air single N 5.0 6.0 5.4 10.0 -1.0 0.0
```

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*. 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.

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.

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 *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.

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

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.

IMPORTANT NOTE: The *global* yes option is not yet implemented.

Restrictions: none

Related commands:

mixture, fix inflow

Default:

The option default is global = no.

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
```

dump command

dump image command

Syntax:

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

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

```
particle args = list of particle attributes
    possible attributes = id, type, proc, x, y, z, xs, ys, zs, vx, vy, vz,
                          ke, erot, evib,
                          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
      c_ID = per-particle vector (or array) calculated by a compute with ID
      c_ID[N] = Nth column of per-particle array calculated by a compute with ID
      f_ID = per-particle vector (or array) calculated by a fix with ID
      f_ID[N] = Nth column of per-particle array calculated by a fix with ID
      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 (or array) calculated by a compute with ID
      c_ID[N] = Nth column of per-grid array calculated by a compute with ID
      f_ID = per-grid vector (or array) calculated by a fix with ID
      f_ID[N] = Nth column of per-grid array calculated by a fix with ID
      v_name = per-grid vector calculated by a grid-style variable with name

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

      id = surf element ID
      v1x,v1y,v1z = coords of 1st vertex in surface element
      v1x,v1y,v1z = coords of 2nd vertex in surface element
      v1x,v1y,v1z = coords of 3rd vertex in surface element
```

```
            c_ID = per-surf vector (or array) calculated by a compute with ID
            c_ID[N] = Nth column of per-surf array calculated by a compute with ID
            f_ID = per-surf vector (or array) calculated by a fix with ID
            f_ID[N] = Nth column of per-surf array calculated by a fix with ID
            v_name = per-surf vector calculated by a surf-style variable with name

      image args = discussed on dump image doc page
```

Examples:

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

Description:

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

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

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

The *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 (see the binary2txt tool) or write your own code to read the binary file. The format of the binary file can be understood by looking at the tools/binary2txt.cpp file.

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

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

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

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 8 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.

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 two kinds of child cells: unsplit cells and sub cells of split cells. Section 4.8 of the manual gives details of how SPARTA defines child, unsplit, 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.

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 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.

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

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

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

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

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

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

See Section 8 of the manual for information on how to add new compute and fix styles to SPARTA to calculate per-grid quantities which could then be output into dump files.

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

Id is the surface element ID.

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

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

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

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

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

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

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

See Section 8 of the manual for information on how to add new compute and fix styles to SPARTA to calculate per-surf quantities which could then be output into dump files.

Restrictions:

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

Related commands:

dump image, dump_modify, undump

Default:

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

dump image command

dump movie command

Syntax:

```
dump ID style 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 doc page)
- 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 all image 100 dump.*.jpg type type
dump myDump all movie 100 movie.mpg type type
```

These commands will dump shapshot images of all particles 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 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.

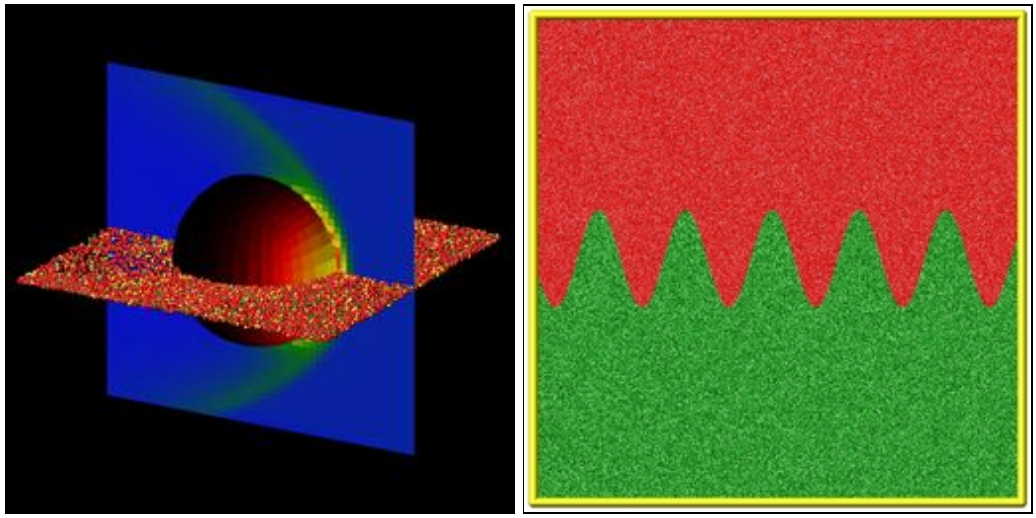
Any or all of these entities can be included in the images:

- particles (all 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 NOTE: The *persp* keyword is not yet supported as an option.

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

The *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 Section 2.2.2 section 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

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
    Np = write one file for every this many processors
  first arg = yes or no
  flush arg = yes or no
  format arg = C-style format string for one line of output
  nfile arg = Nf
    Nf = write this many files, one from each of Nf processors
  pad arg = Nchar = # of characters to convert timestep to
  region arg = region-ID or "none"
  thresh args = attribute operation value
    attribute = same attributes (x,fy,etotal,sxx,etc) used by dump custom style
    operation = "" or ">=" or "==" or "!="
    value = numeric value to compare to
    these 3 args can be replaced by the word "none" to turn off thresholding
```

- 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
    color = name of color for background
  bitrate arg = rate
    rate = target bitrate for movie in kbps
  boxcolor arg = color
    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 "%d %d %20.15g %g %g"
dump_modify myDump thresh x <0.0 thresh vx >= 3.0
dump_modify 1 every 1000
dump_modify 1 every v_myVar
dump_modify 1 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 variables, as examples of useful functions to use in this context. Other similar math functions could easily be added as options for equal-style variables. When using the variable option with the *every* keyword, you also need to use the *first* option if you want an initial snapshot written to the dump file.

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

```
variable               s equal logfreq(10,3,10)
dump            1 all particle 100 tmp.dump id type x y z
dump_modify     1 every v_s first yes
```

The *fileper* keyword is 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 text-based dump styles have a default C-style format string which simply specifies %d for integers and %g for real values, when per particle or per grid cell or per surface element info is written to the dump file. The *format* keyword can be used to override the default with a new C-style format string. The number of fields it specifies should match the number of entities being printed on each line. Do not include a trailing "\n" newline character in the format string.

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 and 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 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 dump movie command 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 NOTE: 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 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 dump movie command 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 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, dump image, undump

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 and dump_modify commands. Additional colors can be defined with the dump_modify color command. The 3 numbers listed for each name are the RGB (red/green/blue) values. Divide each value by 255 to get the equivalent 0.0 to 1.0 value.

aliceblue = 240, 248, 255	antiquewhite = 250, 235, 215	aqua = 0, 255, 255	aquamarine = 127, 255, 212	azure = 240, 255, 255
beige = 245, 245, 220	bisque = 255, 228, 196	black = 0, 0, 0	blanchedalmond = 255, 255, 205	blue = 0, 0, 255
blueviolet = 138, 43, 226	brown = 165, 42, 42	burlywood = 222, 184, 135	cadetblue = 95, 158, 160	chartreuse = 127, 255, 0
chocolate = 210, 105, 30	coral = 255, 127, 80	cornflowerblue = 100, 149, 237	cornsilk = 255, 248, 220	crimson = 220, 20, 60
cyan = 0, 255, 255	darkblue = 0, 0, 139	darkcyan = 0, 139, 139	darkgoldenrod = 184, 134, 11	darkgray = 169, 169, 169
darkgreen = 0, 100, 0	darkkhaki = 189, 183, 107	darkmagenta = 139, 0, 139	darkolivegreen = 85, 107, 47	darkorange = 255, 140, 0
darkorchid = 153, 50, 204	darkred = 139, 0, 0	darksalmon = 233, 150, 122	darkseagreen = 143, 188, 143	darkslateblue = 72, 61, 139
darkslategray = 47, 79, 79	darkturquoise = 0, 206, 209	darkviolet = 148, 0, 211	deeppink = 255, 20, 147	deepskyblue = 0, 191, 255
dimgray = 105, 105, 105	dodgerblue = 30, 144, 255	firebrick = 178, 34, 34	floralwhite = 255, 250, 240	forestgreen = 34, 139, 34
fuchsia = 255, 0, 255	gainsboro = 220, 220, 220	ghostwhite = 248, 248, 255	gold = 255, 215, 0	goldenrod = 218, 165, 32
gray = 128, 128, 128	green = 0, 128, 0	greenyellow = 173, 255, 47	honeydew = 240, 255, 240	hotpink = 255, 105, 180
indianred = 205, 92, 92	indigo = 75, 0, 130	ivory = 255, 240, 240	khaki = 240, 230, 140	lavender = 230, 230, 250
	lawngreen = 124, 252, 0			

lavenderblush = 255, 240, 245		lemonchiffon = 255, 250, 205	lightblue = 173, 216, 230	lightcoral = 240, 128, 128
lightcyan = 224, 255, 255	lightgoldenrodyellow = 250, 250, 210	lightgreen = 144, 238, 144	lightgrey = 211, 211, 211	lightpink = 255, 182, 193
lightsalmon = 255, 160, 122	lightseagreen = 32, 178, 170	lightskyblue = 135, 206, 250	lightslategray = 119, 136, 153	lightsteelblue = 176, 196, 222
lightyellow = 255, 255, 224	lime = 0, 255, 0	limegreen = 50, 205, 50	linen = 250, 240, 230	magenta = 255, 0, 255
maroon = 128, 0, 0	mediumaquamarine = 102, 205, 170	mediumblue = 0, 0, 205	mediumorchid = 186, 85, 211	mediumpurple = 147, 112, 219
mediumseagreen = 60, 179, 113	mediumslateblue = 123, 104, 238	mediumspringgreen = 0, 250, 154	mediumturquoise = 72, 209, 204	mediumvioletred = 199, 21, 133
midnightblue = 25, 25, 112	mintcream = 245, 255, 250	mistyrose = 255, 228, 225	moccasin = 255, 228, 181	navajowhite = 255, 222, 173
navy = 0, 0, 128	oldlace = 253, 245, 230	olive = 128, 128, 0	olivedrab = 107, 142, 35	orange = 255, 165, 0
orangered = 255, 69, 0	orchid = 218, 112, 214	palegoldenrod = 238, 232, 170	palegreen = 152, 251, 152	paleturquoise = 175, 238, 238
palevioletred = 219, 112, 147	papayawhip = 255, 239, 213	peachpuff = 255, 239, 213	peru = 205, 133, 63	pink = 255, 192, 203
plum = 221, 160, 221	powderblue = 176, 224, 230	purple = 128, 0, 128	red = 255, 0, 0	rosybrown = 188, 143, 143
royalblue = 65, 105, 225	saddlebrown = 139, 69, 19	salmon = 250, 128, 114	sandybrown = 244, 164, 96	seagreen = 46, 139, 87
seashell = 255, 245, 238	sienna = 160, 82, 45	silver = 192, 192, 192	skyblue = 135, 206, 235	slateblue = 106, 90, 205
slategray = 112, 128, 144	snow = 255, 250, 250	springgreen = 0, 255, 127	steelblue = 70, 130, 180	tan = 210, 180, 140
teal = 0, 128, 128	thistle = 216, 191, 216	tomato = 253, 99, 71	turquoise = 64, 224, 208	violet = 238, 130, 238
wheat = 245, 222, 179	white = 255, 255, 255	whitesmoke = 245, 245, 245	yellow = 255, 255, 0	yellowgreen = 154, 205, 50

echo command

Syntax:

```
echo style
```

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

Examples:

```
echo both
echo log
```

Description:

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

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

Restrictions: none

Related commands: none

Default:

```
echo log
```

fix command

Syntax:

```
fix ID style args
```

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

Examples:

```
fix 1 grid/check 100 warn
fix 1 all ave/time 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 8 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 NOTE: 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:

- ave/grid - compute per grid cell time-averaged quantities
- ave/surf - compute per surface element time-averaged quantities
- ave/time - compute/output global time-averaged quantities
- balance - perform dynamic load-balancing
- grid/check - check if particles are in the correct grid cell
- inflow - inject particles at global boundaries

- print - print text and variables during a simulation

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

Default: none

fix ave/grid command

Syntax:

```
fix ID ave/grid Nevery Nrepeat Nfreq value1 value2 ... keyword args ...
```

- ID is documented in fix command
- ave/grid = style name of this fix command
- Nevery = use input values every this many timesteps
- Nrepeat = # of times to use input values for calculating averages
- Nfreq = calculate averages every this many timesteps zero or more input values can be listed
- value = c_ID, c_ID[i], f_ID, f_ID[i], v_name

```
c_ID = per-grid vector (or array) calculated by a compute with ID
c_ID[I] = Ith column of per-grid array calculated by a compute with ID
f_ID = per-grid vector (or array) calculated by a fix with ID
f_ID[I] = Ith column of per-grid array calculated by a fix with ID
```

- zero or more keyword/arg pairs may be appended

```
keyword = ave
  ave args = one or running
    one = output a new average value every Nfreq steps
    running = accumulate average continuously
```

Examples:

```
fix 1 all ave/grid 10 20 1000 c_mine
fix 1 all ave/grid 1 100 100 c_2 ave running
```

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

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

Description:

Use one or more per-grid vectors as inputs every few timesteps, and average by grid cell over longer timescales, applying appropriate normalization factors. The resulting per grid cell averages can be used by other output commands such as the dump grid command.

Each input value can be the result of a compute or fix. 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.

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.

Each per-grid value of each input vector is averaged independently.

The *Nevery*, *Nrepeat*, and *Nfreq* arguments specify on what timesteps the input values will be used in order to

contribute to the average. The final averaged quantities are generated on timesteps that are a multiple of *Nfreq*. The average is over *Nrepeat* quantities, computed in the preceding portion of the simulation every *Nevery* timesteps. *Nfreq* must be a multiple of *Nevery* and *Nevery* must be non-zero even if *Nrepeat* is 1. Also, the timesteps contributing to the average value cannot overlap, i.e. Nfreq > (Nrepeat-1)*Nevery is required.

For example, if Nevery=2, Nrepeat=6, and Nfreq=100, then values on timesteps 90,92,94,96,98,100 will be used to compute the final average on timestep 100. Similarly for timesteps 190,192,194,196,198,200 on timestep 200, etc.

If a value begins with "c_", a compute ID must follow which has been previously defined in the input script. If no bracketed term is appended, and the compute calculates a per-grid vector, then the per-grid vector is used. If *c_ID[N]* is used, then N must be in the range from 1-M, which will use the Nth column of the M-column per-grid array calculated by the compute. If *c_ID* is used, and the compute calculates a per-grid array, then it is the same as if the individual columns of the array had been listed one by one.

E.g. these 2 fix ave/grid commands are equivalent, since the compute grid command produces 3 columns of output using the *all* mixture-ID which has a single group:

```
compute 1 grid all n u usq
fix 1 ave/grid 10 100 1000 c_1
fix 1 ave/grid 10 100 1000 c_1[1] c_1[2] c_1[3]
```

Users can also write code for their own compute styles and add them to SPARTA.

If a value begins with "f_", a fix ID must follow which has been previously defined in the input script. If no bracketed term is appended, and the fix calculates a per-grid vector, then the per-grid vector is used. If *f_ID[N]* is used, then N must be in the range from 1-M, which will use the Nth column of the M-column per-grid array calculated by the fix. If *f_ID* is used, and the fix calculates a per-grid array, then it is the same as if the individual columns of the array had been listed one by one. See the example above for computes.

Note that some fixes only produce their values on certain timesteps, which must be compatible with *Nevery*, else an error will result. Users can also write code for their own fix styles and add them to SPARTA.

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 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 calculations for all child grid cells in the simulation, which includes unsplit, split, and sub cells. Section 4.8 of the manual gives details of how SPARTA defines child, unsplit, split, and sub cells.

Restrictions: none

Related commands:

compute, fix ave/time

Default:

The option defaults are ave = one.

fix ave/surf command

Syntax:

```
fix ID ave/surf Nevery Nrepeat Nfreq value1 value2 ... keyword args ...
```

- ID is documented in fix command
- ave/surf = style name of this fix command
- Nevery = use input values every this many timesteps
- Nrepeat = # of times to use input values for calculating averages
- Nfreq = calculate averages every this many timesteps zero or more input values can be listed
- value = 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
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
```

- zero or more keyword/arg pairs may be appended

```
keyword = ave
  ave args = one or running
    one = output a new average value every Nfreq steps
    running = accumulate average continuously
```

Examples:

```
fix 1 all ave/surf 1 100 100 standard ave running
fix 1 all ave/surf 10 20 1000 c_mine
```

Description:

Use one or more per-surf vectors as inputs every few timesteps, and average them surface element by surface element by over longer timescales, applying appropriate normalization factors. The resulting per-surf averages can be used by other output commands such as the dump surf command.

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.

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.

Each per-surf value of each input vector is averaged independently.

The *Nevery*, *Nrepeat*, and *Nfreq* arguments specify on what timesteps the input values will be used in order to contribute to the average. The final averaged quantities are generated on timesteps that are a multiple of *Nfreq*. The average is over *Nrepeat* quantities, computed in the preceding portion of the simulation every *Nevery* timesteps. *Nfreq* must be a multiple of *Nevery* and *Nevery* must be non-zero even if *Nrepeat* is 1. Also, the timesteps contributing to the average value cannot overlap, i.e. Nfreq > (Nrepeat-1)*Nevery is required.

For example, if Nevery=2, Nrepeat=6, and Nfreq=100, then values on timesteps 90,92,94,96,98,100 will be used to compute the final average on timestep 100. Similarly for timesteps 190,192,194,196,198,200 on timestep 200,

etc.

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

E.g. these 2 fix ave/surf commands are equivalent, since the compute surf command produces 5 columns of output using the *all* mixture-ID which has a single group:

```
compute 1 surf all n press px py pz
fix 1 ave/surf 10 100 1000 c_1
fix 1 ave/surf 10 100 1000 c_1[1] c_1[2] c_1[3] c_1[4] c_1[5]
```

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[N]* is used, then N must be in the range from 1-M, which will use the Nth column of the M-column per-surf array calculated by the fix. If *f_ID* is used, and the fix calculates a per-surf array, then it is the same as if the individual columns of the array had been listed one by one. See the example above for computes.

Note that some fixes only produce their values on certain timesteps, which must be compatible with *Nevery*, else an error will result. Users can also write code for their own fix styles and add them to SPARTA.

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.

Restrictions: none

Related commands:

compute, "fix ave/time

Default:

The option defaults are ave = one.

fix ave/time command

Syntax:

```
fix ID ave/time Nevery Nrepeat Nfreq value1 value2 ... keyword args ...
```

- ID is documented in fix command
- ave/time = style name of this fix command
- Nevery = use input values every this many timesteps
- Nrepeat = # of times to use input values for calculating averages
- Nfreq = calculate averages every this many timesteps
- one or more input values can be listed
- value = c_ID, c_ID[N], f_ID, f_ID[N], v_name

```
c_ID = global scalar or vector 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 compu
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 v
v_name = global value calculated by an equal-style variable with name
```

- zero or more keyword/arg pairs may be appended
- keyword = *mode* or *file* or *ave* or *start* or *off* or *title1* or *title2* or *title3*

```
mode arg = scalar or vector
    scalar = all input values are global scalars
    vector = all input values are global vectors or global arrays
  ave args = one or running or window M
    one = output a new average value every Nfreq steps
    running = output cummulative average of all previous Nfreq steps
    window M = output average of M most recent Nfreq steps
  start args = Nstart
    Nstart = start averaging on this timestep
  off arg = M = do not average this value
    M = value # from 1 to Nvalues
  file arg = filename
    filename = name of file to output time averages to
  title1 arg = string
    string = text to print as 1st line of output file
  title2 arg = string
    string = text to print as 2nd line of output file
  title3 arg = string
    string = text to print as 3rd line of output file, only for vector mode
```

Examples:

```
fix 1 all ave/time 100 5 1000 c_myTemp c_thermo_temp file temp.profile
fix 1 all ave/time 100 5 1000 c_thermo_press[2] ave window 20 &
                              title1 "My output values"
fix 1 all ave/time 1 100 1000 f_indent f_indent[1] file temp.indent off 1
```

Description:

Use one or more global values as inputs every few timesteps, and average them over longer timescales. The resulting averages can be used by other output commands such as stats_style custom, and can also be written to a file. Note that if no time averaging is done, this command can be used as a convenient way to simply output one or more global values to a file.

Each listed value can be the result of a compute or fix or the evaluation of an equal-style variable. In each case, the compute, fix, or variable must produce a global quantity, not a per-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* = vector, then the input values may either be vectors or arrays and all must be the same "length", which is the length of the vector or number of rows in the array. If a global array is listed, then it is the same as if the individual columns of the array had been listed one by one. E.g. these two fix ave/time commands are equivalent, since the compute grid command in this case produces 3 columns of output.

```
compute 1 grid all n u usq
fix 1 ave/time 100 1 100 c_1 file tmp.grid mode vector
fix 1 ave/time 100 1 100 c_1[1] c_1[2] c_1[3] file tmp.grid mode vector
```

The *Nevery*, *Nrepeat*, and *Nfreq* arguments specify on what timesteps the input values will be used in order to contribute to the average. The final averaged quantities are generated on timesteps that are a mlutiple of *Nfreq*. The average is over *Nrepeat* quantities, computed in the preceding portion of the simulation every *Nevery* timesteps. *Nfreq* must be a multiple of *Nevery* and *Nevery* must be non-zero even if *Nrepeat* is 1. Also, the timesteps contributing to the average value cannot overlap, i.e. Nfreq > (Nrepeat-1)*Nevery is required.

For example, if Nevery=2, Nrepeat=6, and Nfreq=100, then values on timesteps 90,92,94,96,98,100 will be used to compute the final average on timestep 100. Similarly for timesteps 190,192,194,196,198,200 on timestep 200, etc. If Nrepeat=1 and Nfreq = 100, then no time averaging is done; values are simply generated on timesteps 100,200,etc.

If a value begins with "c_", a compute ID must follow which has been previously defined in the input script. If *mode* = scalar, then if no bracketed term is appended, the global scalar calculated by the compute is used. If a bracketed term is appended, the Ith element of the global vector calculated by the compute is used. If *mode* = vector, then if no bracketed term is appended, the global vector calculated by the compute is used. Or if the compute calculates an array, all of the columns of the global array are used as if they had been specified as individual vectors (see description above). If a bracketed term is appended, the Ith column of the global array calculated by the compute is used.

Note that 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. Or if the fix calculates an array, all of the columns of the global array are used as if they had been specified as individual vectors (see description above). If a bracketed term is appended, the Ith column of the global array calculated by the fix is used.

Note that some fixes only produce their values on certain timesteps, which must be compatible with *Nevery*, else an error will result. Users can also write code for their own fix styles and add them to SPARTA.

If a value begins with "v_", a variable name must follow which has been previously defined in the input script. Variables can only be used as input for *mode* = scalar. Only equal-style variables can be referenced. See the variable command for details. Note that variables of style *equal* define a formula which can reference 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, fix ave/grid, fix ave/surf, variable

Default: none

The option defaults are mode = scalar, ave = one, start = 0, no file output, title 1,2,3 = strings as described above, and no off settings for any input values.

fix balance 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
```

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 5.8 of the manual for an explanation of clumped and dispersed grid cell assignments and their relative performance trade-offs.

Rebalancing is attempted 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.

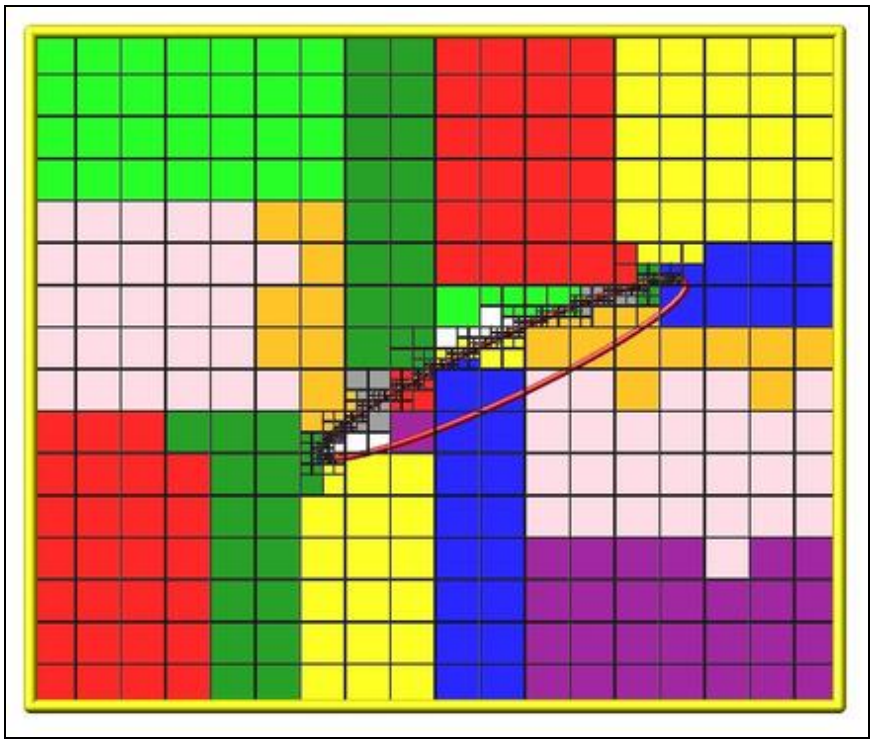
IMPORTANT NOTE: This command attempts to minimize the imbalance factor, as defined above. But computational cost is not strictly proportional to particle count, depending on the collision and chemistry models being used. Also, changing the assignment of grid cells and particles to processors may lead 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*, than 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.

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.



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.

Restrictions: none

Related commands:

create_grid, balance_grid

Default: none

fix grid/check command

Syntax:

```
fix ID grid/check N outflag
```

- 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*

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 outflag setting is *error*, SPARTA will print an error and stop if it finds a particle in an incorrect grid cell. 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.

Restrictions: none

Related commands: none

Default: none

fix inflow command

Syntax:

```
fix ID inflow mix-ID face1 face2 ... keyword value ...
```

- ID is documented in fix command
- inflow = style name of this fix command
- mix-ID = ID of mixture to use when creating particles
- face1,face2,... = one or more of *all* or *xlo* or *xhi* or *ylo* or *yhi* or *zlo* or *zhi* zero or more keyword/value pairs may be appended
- keyword = *n* or *nevery* or *perspecies*

```
  n value = Np = number of particles to create
    nevery value = Nstep = insert every this many timesteps
    perspecies value = yes or no
```

Examples:

```
fix in inflow air all
fix in inflow mymix xlo yhi n 1000 nevery 10
```

Description:

Insert particles into the simulation box continuously during a simulation. The particles are inserted on the specified faces of the box using properties of the specified mixture. If done every timestep, this induces a continuous influx of particles thru the face(s).

The properties of the inserted particles are from the mixture with ID *mix-ID*. This determines the species of the particles added at what relative fractions, and the overall number density, streaming velocity, and temperature. See the mixture command for details.

One or more faces of the simulation box can be specified via the *face1*, *face2*, etc arguments. 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 a face(s) on an inflow boundary performs the following computation at the beginning of the SPARTA timestep.

The molecular flux across a surface element 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 the flux, the global *fnum* ratio (specified by the global command, the mixture number density, the mixture streaming velocity and thermal temperature, the area of the face, and its orientation relative to the streaming velocity.

If *M* has a fractional value, e.g. 12.5, then 12 particles are inserted, and a 13th depending on the value of a random number. Each particle is inserted at a random location on the 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. The velocity of the particle is set to the sum of the mixture's streaming velocity and a thermal velocity sampled from the thermal temperature of the mixture. Both the streaming velocity and thermal temperature are also set by the mixture command.

If the final particle velocity is not directed "into" the grid cell, then the velocity sampling procedure is repeated until it is. This insures that all inserted particles enter the simulation domain, as desired.

The first timestep that inserted particles are advected, they move for a random fraction of the timestep. This insures a continuous flow field of particles entering the simulation box.

The *n* keyword can alter how many particles are inserted, which can be useful for debugging purposes. If *Np* is set to 0, then the number of inserted 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 inserted 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 inserting an equal number of particles per grid cell.

The *nevery* keyword determines how often particles are inserted. 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 inserted particle is randomly determined. This has an effect on the statistical properties of the inserted 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 inserted, 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 inserted, a random number is used to choose the species of the particle, based on the relative number fractions of all the species in the mixture. As before, if *M* has a fractional value, e.g. 12.5, then 12 particles will always be inserted, and a 13th depending on the 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 inserts 10 particles from that mixture. If *perspecies* is set to *yes*, then exactly 5 particles of each species will be inserted on every timestep insertions take place. If *perspecies* is set to *no*, then exactly 10 particles will be inserted every time and on average the insertion will be 5 particles of each of the two species. But on one timestep it might be 6 of the first and 4 of the second. On another timestep it might be 3 of the first and 7 of the second.

Restart, 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 inserted on the most recent insertion step. The second element is the cummulative total number inserted since the beginning of the run. The 2nd value is initialized to zero each time a run is performed.

Restrictions:

Particles cannot be inserted on *z* faces of the simluation box for a 2d simulation. Particles cannot be inserted on periodic faces of the simulation box.

A *n* setting of *Np* > 0 can only be used with a *perspecies* setting of *no*.

Related commands:

mixture, create_particles

Default:

The keyword defaults are n = 0, nevery = 1, perspecies = yes.

(Bird94) G. A. Bird, Molecular Gas Dynamics and the Direct Simulation of Gas Flows, Clarendon Press, Oxford (1994).

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, print

Default:

The option defaults are no file output, screen = yes, and title string as described above.

global command

Syntax:

```
global keyword values ...
```

- one or more keyword/value pairs
- keyword = *fnum* or *nrho* or *vstream* or *temp* or *gravity* or *surfmax* or *gridcut* or *comm/sort* or *comm/style* or *weight*

```
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
  surfmax value = Nsurf
    Nsurf = max # of surface elements allowed in single grid cell
  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 = grid
    mode = none or volume or radius
```

Examples:

```
global fnum 1.0e20
global vstream 100.0 0 0 fnum 5.0e18
global temp 1000
global weight cell radius
```

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 inflow commands.

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.

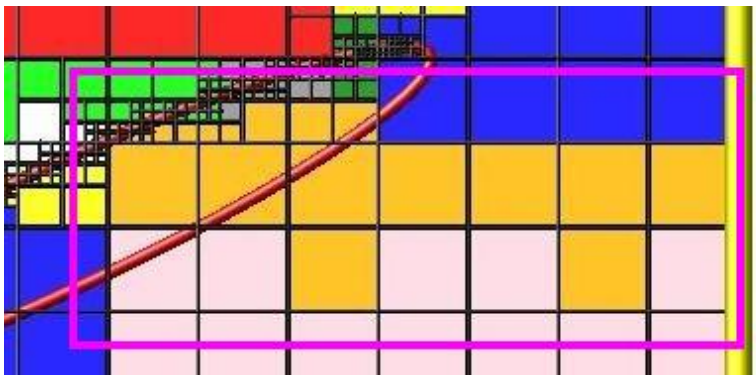
The *gravity* keyword sets an acceleration term which is included in the motion of particles. The magnitude of gravity is set by the *mag* keyword. Its direction of action is set as (ex,ex,ez). The direction does not have to be a unit vector. If the magnitude is set to 0.0, no acceleration term is included, which is the default.

The *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).

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.

IMPORTANT NOTE: Using the *gridcut* keyword with a cutoff >= 0.0 is only possible if the grid cells owned by each processor are "clumped" (by the time the next simulation is performed). If each processor's grid cells are "dispersed", then ghost cells cannot be stored for a *gridcut* cutoff >= 0.0. This will generate an error when a simulation is performed. The solution is to use the balance_grid command to change to a clumped grid cell assignment. See Section 5.8 of the manual for an explanation of clumped and dispersed grid cell assignments and their relative performance trade-offs.

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 NOTE: If grid cells have already been defined via the create_grid, read_grid, or read_restart commands, when the *gridcut* cutoff is specified, then all ghost cell information that is currently stored will be erased. In this case, a balance_grid command must then be invoked to regenerate ghost cell information before a simulation is performed. Note that the previous note applies in this scenario. If the "balance_grid" option selects a dispersed grid cell assignment and the *gridcut* cutoff = -1.0, then no ghost information will be generated, resulting in an error when a simulation is run.

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 = *grid*, is grid-based 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.

If grid-based particle weighting is enabled, two computations occur. First, each grid cell stores a "weight" which is pre-computed based either on the cell *volume* or *radius*, as specified by the *mode* setting. The *radius* setting is only allowed for axisymmetric systems; see Section 4.2 for details. The radius in this case is the distance the cell's midpoint is from the axis of symmetry.

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.

Restrictions:

The global surfmax command must be used before surface elements are defined, e.g. via the read_surf command.

Related commands:

mixture

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, surfmax = 100, gridcut = -1.0, comm/sort = no, comm/style = neigh.

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 ${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, print

Default: none

include command

Syntax:

```
include file
```

- file = filename of new input script to switch to

Examples:

```
include newfile
include in.run2
```

Description:

This command opens a new input script file and begins reading SPARTA commands from that file. When the new file is finished, the original file is returned to. Include files can be nested as deeply as desired. If input script A includes script B, and B includes A, then SPARTA could run for a long time.

If the filename is a variable (see the variable command), different processor partitions can run different input scripts.

Restrictions: none

Related commands:

variable, jump

Default: none

jump command

Syntax:

```
jump file label
```

- file = filename of new input script to switch to
- label = optional label within file to jump to

Examples:

```
jump newfile
jump in.run2 runloop
jump SELF runloop
```

Description:

This command closes the current input script file, opens the file with the specified name, and begins reading SPARTA commands from that file. Unlike the include command, the original file is not returned to, although by using multiple jump commands it is possible to chain from file to file or back to the original file.

If the word "SELF" is used for the filename, then the current input script is re-opened and read again.

IMPORTANT NOTE: The SELF option is not guaranteed to work when the current input script is being read through stdin (standard input), e.g.

```
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 all atom 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, include, label, next

Default: none

label command

Syntax:

```
label ID
```

- ID = string used as label name

Examples:

```
label xyz
label loop
```

Description:

Label this line of the input script with the chosen ID. Unless a jump command was used previously, this does nothing. But if a jump command was used with a label argument to begin invoking this script file, then all command lines in the script prior to this line will be ignored. I.e. execution of the script will begin at this line. This is useful for looping over a section of the input script as discussed in the jump command.

Restrictions: none

Related commands: none

Default: none

log command

Syntax:

```
log file 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 start 5 for details.

Restrictions: none

Related commands: none

Default:

The default SPARTA log file is named log.sparta

mixture command

Syntax:

```
mixture ID species1 species2 ... keyword args ...
```

- ID = user-defined name of the mixture
- species1, species2, ... = zero or more species IDs to include in the mixture
- zero or more keyword/arg pairs may be appended
- keyword = *nrho* or *vstream* or *temp* or *frac* or *group*

```
nrho arg = density
    density = number density of entire mixture (# per length^3 units)
  vstream args = Vx Vy Vz
    Vx,Vy,Vz = streaming velocity of entire mixture (velocity units)
  temp arg = thermal
    thermal = temperature of entire mixture (temperature units)
  frac arg = fraction
    fraction = number fraction for each listed species (0 to 1)
  group arg = SELF or group-ID
    SELF = put each listed species (or all species if none listed) in its own group
    group-ID = put the listed species (or all species if none listed) in a group with this ID
```

Examples:

```
mixture air N O NO group lite
mixture air N O NO vstream 250.0 0.0 0.0 group species
mixture air N frac 0.8
mixture air O frac 0.2
mixture background N O
```

Description:

Define a gas mixture and its properties. A mixture can be referenced by its ID in several other SPARTA commands such as create_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 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 that belongs to its own group.

Zero or more species can be specified in the mixture command. If a listed species is not already in the mixture, due to a previous mixture command with the same ID, then that species is added to the mixture. As discussed below, it will be assigned to a default group and assigned default per-species attributes, unless the appropriate keywords are also specified.

Species can be specified which are already part of the mixture, to change their group 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.

The *nrho* keyword sets a global attribute of the mixture, namely its density. For 3d simulations the units of the specified *density* are #/volume. For 2d, the units are effectively #/area, since the z-dimension thickness of the simulation box = 1.0.

The *vstream* keyword sets a global attribute of the mixture, namely the streaming velocity. Particles created using the mixture will have the specified *Vx,Vy,Vz* values.

The *temp* keyword sets a global attribute of the mixture, namely the thermal temperature of its particles. When particles are created, this value is used to sample a Gaussian velocity distribution, which is superposed on the streaming velocity, when a particle's velocity is initialized.

The *frac* keyword sets a per-species attribute for individual species in the mixture. Each species has a relative fractional density, such as 0.2, meaning one out of 5 particles is that species. The sum of this value across all species in the mixture must equal 1.0. The *frac* keyword sets this value for the listed species. If this value has never been set for M species out of the total N species in the mixture, then when a simulation is run, the *frac* value for each of the M species is set to (1 - sum)/M, where sum is the sum of the *frac* values for the N-M assigned species.

Each species in a mixture is assigned to exactly one group. The *group* keyword can be used to set or change these assignments.

As described by the collide command, mixture groups are used when performing collisions so that collisions attempts, partners, and parameters can be treated on a per-group basis for accuracy and efficiency. Per-grid computes also use mixture groups to calculate per-grid quantities on subsets of particles within each grid cell.

The specified group ID can be any string you choose. Similar to the mixture ID, it can only contain alphanumeric characters and underscores. Using SELF for the group ID has a special meaning as discussed below.

The operation of the *group* keyword depends on whether zero species or some species are specified explicitly in the mixture command. It also depends on whether the group ID is SELF or a user-defined name. In each case, after the operation is done, any group IDs for the mixture that have no species assigned to them are deleted.

- If zero species are listed and the group ID is SELF, then each species already in the mixture is assigned to a group with its species ID as the group ID. I.e. there will now be one species per group.
- If one or more species are listed and the group ID is SELF, then each listed species is assigned to a group with its species ID as the group ID.

- If zero species are listed and the group ID is not SELF, then all species already in the mixture are assigned to a group with the specified ID.
- If one or more species are listed and the group ID is not SELF, then the listed species are all assigned to a group with the specified ID.

Note that if the *group* keyword is not used, no changes to group assignements are made. If one or more new species are specified, which are not already in the mixture, then each is assigned to a group with the species ID as the group ID.

Restrictions: none

Related commands:

global, create_particles

Default:

The *nrho*, *vstream*, and *temp* defaults are those defined for the background gas density, as set by the global command. The *frac* default is described above. The *group* keyword has no default; if it is not used, new species not already in the mixture are each assigned to a group with the species ID as the group ID.

next command

Syntax:

```
next variables
```

- variables = one or more variable names

Examples:

```
next x
next a t x myTemp
```

Description:

This command is used with variables defined by the variable command. It assigns the next value to the variable from the list of values defined for that variable by the variable command. Thus when that variable is subsequently substituted for in an input script command, the new value is used.

See the variable command for info on how to define and use different kinds of variables in SPARTA input scripts. If a variable name is a single lower-case character from "a" to "z", it can be used in an input script command as $a or $z. If it is multiple letters, it can be used as ${myTemp}.

If multiple variables are used as arguments to the *next* command, then all must be of the same variable style: *index*, *loop*, *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 2.5 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
  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, include, shell, variable,

Default: none

partition command

Syntax:

```
partition style N command ...
```

- style = *yes* or *no*
- N = partition number (see asterisk form below)
- command = any SPARTA command

Examples:

```
partition yes 1 processors 4 10 6
partition no 5 print "Active partition"
partition yes *5 fix all nve
partition yes 6* fix all nvt temp 1.0 1.0 0.1
```

Description:

This command invokes the specified command on a subset of the partitions of processors you have defined via the -partition command-line switch. See Section 2.5 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

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, variable

Default:

The option defaults are no file output and screen = yes.

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

Default: none

react command

Syntax:

```
react style args
```

- style = *none* or *tce* or *qk*
- args = arguments for that style

```
none args = none
  tce args = infile
    infile = file with list of chemical reactions
  qk args = infile
    infile = file with list of chemical reactions
```

Examples:

```
react none
react tce air.tce
```

Description:

Define chemical reactions to perform when particle-particle collisions occur.

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 and the corresponding input file format are described below.

The *tce* style is Bird's Total Collision Energy Model. Using kinetic theory, it 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 is an Arrhenius-type rate for each reaction

where k_f is the forward reaction rate, T is the Temperature and k the Boltzman constant.

The format of the input 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 data directory in the SPARTA distribution contains reaction files for the TCE model, all with the suffix ".tce".

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. 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
```

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. 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 P1 and P2. R2 is preserved in the collision, so P3 = R2 is required.

IMPORTANT NOTE: Ionization and recombination reactions have not yet been implemented in SPARTA.

The *style* of each reaction is a single character (upper or lower case) with the following meaning:

- A = Arrhenius

The style determines how many reaction coefficients are listed as C1, C2, ..., and how they are interpreted by SPARTA.

For the A = Arrhenius style, there are 5 coefficients:

- C1 = number of internal degrees of freedom (as defined by the TCE model)
- C2 = activation Energy E_a
- C3 = Arrhenius Rate A
- C4 = Arrhenius rate b
- C5 = reaction energy (positive for exothermic)

The *qk* style is Bird's Quantum-Kinetic model (QK).

IMPORTANT NOTE: The QK model has not yet been implemented in SPARTA.

Restrictions: none

Related commands:

collide

Default:

style = none

(Bird94) G. A. Bird, Molecular Gas Dynamics and the Direct Simulation of Gas Flows, Clarendon Press, Oxford (1994).

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 NOTE: See Section 5.8 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. Finally, if the entry in the grid file for this parent cell is

```
index 12-352-65 2 2 2
```

then it means the parent cell is further sub-divided into 2x2x2 level 4 cells, Each of those cells could be a child cell or yet another parent cell. 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.

For 2d simulations, *Pz* and *Cz* must both equal 1, for every parent cell listed in the file.

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, create_grid

Default: none

read_restart command

Syntax:

```
read_restart file
```

- file = name of binary restart file to read in

Examples:

```
read_restart save.10000
read_restart restart.*
read_restart flow.*.%
```

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 NOTE: Because restart files are binary, they may not be portable to other machines. SPARTA will print an error message if this is the case.

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, but will be "dispersed".

IMPORTANT NOTE: See Section 5.8 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 restart file is read, 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.

IMPORTANT NOTE: As explained below, the restart file contains the setting for the global gridcut command. If you restart on a different number of processors, and the gridcut setting from the original script is >= 0.0, then SPARTA will be unable to generate ghost cell information when setting up the grid. This is because the assignment of grid cells to processors will not be clumped. This will generate an error when a simulation is performed. The solution is to use the balance_grid command after reading the restart file. Optionally, the global gridcut command can be used before the balance_grid command, to change the cutoff setting. If the cutoff is set to -1.0, then the balance_grid none command can be used, which will not change the grid cell assignments, but will generate the needed ghost cell information.

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 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:

- units
- simulation box size and boundary conditions
- all settings of the global command
- particles with their individual attributes
- particle species info
- mixtures
- geometry of the hierarchical grid that overlays the simulation domain
- geometry of all derined surface elements
- current timestep number

Basically, this means the information from the original input script specified by these commands is saved in the restart file:

- units
- dimension
- create_box
- boundary
- species
- mixture
- create_grid or read_grid
- read_surf

No other information is stored in the restart file. Specifically, information about these simulation parameters is NOT stored:

- computes
- fixes
- collision model
- chemistry (reaction) model
- surface collision models
- variables

- regions
- output options for stats, dump, restart files
- timestep size

Basically, 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:

- compute
- fix
- collide
- react
- surf_collide
- surf_modify
- variable
- region
- stats_style
- dump
- restart
- timestep

In particular, take note of these 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. The latter would have been setup in the original script via the read_surf command.
- Even 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.

Also note that many commands can be used after a restart file is read, to re-specify a setting that was stored in the restart file. file is read. For example, the global command can be used to reset the values of its specified keywords.

Restrictions: none

Related commands:

read_grid, read_surf, write_restart, restart

Default: none

read_surf command

Syntax:

```
read_surf ID filename keyword args ...
```

- ID = ID for all surface elements in the file
- 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 *invert* or *clip*

```
origin args = Ox Oy Oz
  Ox,Oy,Oz = set origin of surface to this point (distance units)
trans args = Dx Dy Dz
  Dx,Dy,Dz = translate origin by this displacement (distance units)
atrans args = Ax Ay Az
  Ax,Ax,Az = translate origin to this absolute point (distance units)
ftrans args = Fx Fy Fz
  Fx,Fy,Fz = translate origin to this fractional point in simulation box
scale args = Sx Sy Sz
  Sx,Sy,Sz = scale surface by these factors around origin
rotate args = theta Rx Ry Rz
  theta = rotate surface by this angle in counter-clockwise direction (degrees)
  Rx,Ry,Rz = rotate around vector starting at origin pointing in this direction
invert args = none
clip args = none
```

Examples:

```
read_surf 1 surf.sphere
read_surf sphere surf.file trans 10 5 0 scale 3 3 3 invert clip
```

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. The surface elements are triangles in 3d or line segments in 2d. The 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.

Particles collide with surface elements as they advect, and optionally perform surface chemistry during the collision. Collision statistics for each surface element can be tallied via the compute surf command, time-averaged via the fix ave/surf command, and ouput via the dump surface command.

IMPORTANT NOTE: Surface chemistry models have not yet been implemented in SPARTA.

The ID assigned to the surfaces read by the command is used to identify them in other commands. The ID can only contain alphanumeric characters and underscores. See the surf_collide command as an example, which defines what model is used to perform collisions when particles collide with elements in the surface.

Note that the read_surf command can be used multiple times to read multiple objects from multiple files. The surfaces in each file can be assigned to a new unique ID. Or the ID can be the same as the ID used in the last read_surf command, so that surface elements from multiple (successive) files are all assigned to the same ID. You

cannot re-use an ID from a read_surf command older than the previous read_surf command.

The format of the surface file is discussed below. The optional keywords allow the vertices in the file to be translated, scaled, and rotated in various ways. These allow a single surface file, e.g. containing a unit sphere, to be used in a variety of simulations.

The specified file can be a text file or a gzipped text file (detected by a .gz suffix).

A surface file has a header and a body. The header appears first. The first line of the header is always skipped; it typically contains a description of the file. Then lines are read one at a time. Lines can have a trailing comment starting with '#' that is ignored. If the line is blank (only whitespace after comment is deleted), it is skipped. If the line contains a header keyword, the corresponding value is read from the line. If it doesn't contain a header keyword, the line begins the body of the file.

The body of the file contains 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".

- *points* = # of points in surface
- *lines* = # of line segments in surface (only allowed for 2d)
- *triangles* = # of triangles in surface (only allowed for 3d)

These are the recognized section keywords for the body of the file.

- *Points, Lines, Triangles*

The *Points* section consists of N consecutive entries, where N = # of points, each of this form:

```
point-ID x y z     (for 3d)
point-ID x y       (for 2d)
```

The point-ID is ignored; it is only added to assist in examining the file. The point-IDs should run consecutively from 1 to N. *X,y,z* are the coordinates of the point in distance units. Note that for 2d simulations, *z* should be omitted.

The *Lines* section is only allowed for 2d simulations and consists of N entries, where N = # of lines, each of this form:

```
line-ID p1 p2
```

The line-ID is ignored; it is only added to assist in examining the file. *P1* and *p2* are the point-IDs of the 2 end points of the line segment. Each is a value from 1 to Npoints, as described above.

The ordering of *p1* and *p2* is important as it defines the direction of the outward normal for the line segment when a 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, each of this form:

```
tri-ID p1 p2 p3
```

The tri-ID is ignored; it is only added to assist in examining the file. *P1* and *p2* and *p3* are the point-IDs of the 3 corner points of the triangle. Each is a value from 1 to Npoints, as described above.

The ordering of *p1* and *p2* and *p3* is important as it defines the direction of the outward normal for the triangle when a 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.

The following optional keywords can be specified. 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* followed by a *origin*.

Most of the keywords perform a geometric transformation on all the vertices in the surface file with respect to an origin point. By default the origin is (0.0,0.0,0.0), regardless of the position of individual vertices in the file.

The *origin* keyword resets the origin to the specified *Ox,Oy,Oz*. This operation has no effect on the vertices.

The *trans* keyword shifts or displaces the origin by the vector (Dx,Dy,Dz). It also displaces each vertex by (Dx,Dy,Dz).

The *atrans* keyword resets the origin to an absolute point (Ax,Ay,Az) which implies a displacement (Dx,Dy,Dz) from the current origin. It also displaces each vertex by (Dx,Dy,Dz).

The *ftrans* keyword resets the origin to a fractional point (Fx,Fy,Fz). Fractional means that Fx = 0.0 is the lower edge/face in the x-dimension and Fx = 1.0 is the upper edge/face in the x-dimension, and similarly for Fy and Fz. This change of origin implies a displacement (Dx,Dy,Dz) from the current origin. This operation also displaces each vertex by (Dx,Dy,Dz).

The *scale* keyword does not change the origin. It computes the displacement vector of each vertex from the origin (delx,dely,delz) and scales that vector by (Sx,Sy,Sz), so that the new vertex coordinate is (Ox + Sx*delx,Oy + Sy*dely,Oz + Sz*delz).

The *rotate* keyword does not change the origin. It rotates the coordinates of all vertices by an angle *theta* in a counter-clockwise direction, around the vector starting at the origin and pointing in the direction *Rx,Ry,Rz*. Any rotation can be represented by an appropriate choice of origin, *theta* and (Rx,Ry,Rz).

The *invert* keyword does not change the origin or any vertex coordinates. It flips the direction of the outward surface normal of each surface element by changing the ordering of its vertices. Since particles only collide with the outer surface of a surface element, this is a mechanism for using a surface files containing a single sphere (for example) as either a sphere to embed in a flow field, or a spherical outer boundary containing the flow.

The *clip* keyword does not change the origin. It truncates or "clips" a surface that extends outside the simulation box in the following manner. In 3d, each of the 6 clip planes represented by faces of the global simulation box are

considered in turn. Any triangle that straddles the face (with points on both sides of the clip plane), is truncated at the plane. New points along the edges that cross the plane are created. A triangle may also become a trapezoid, in which case it turned into 2 triangles. Then all the points on the side of the clip plane that is outside the box, are projected onto the clip plane. Finally, all triangles that lie in the clip plane are removed, as are any points that are unused after the triangle removal. After this operation is repeated for all 6 faces, the remaining surface is entirely inside the simulation box, though some of its triangles may include points on the faces of the simulation box. A similar operation is performed in 2d with the 4 clip edges represented by the edges of the global simulation box.

If you use the *clip* keyword, you should check the resulting statistics of the clipped surface printed out by this command, including the minimum size of line and triangle edge lengths. It is possible that very short lines or very small triangles will be created near the box surface due to the clipping operation, depending on the coordinates of the initial unclipped points.

Restrictions:

This command can only be used after the simulation box is defined by the create_box command, and after a grid has been created by the create_grid command.

To read gzipped surface files, you must compile SPARTA with the -DSPARTA_GZIP option - see Section 2.2 of the manual for details.

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: none

Default:

The default origin for the vertices in the surface file is (0,0,0).

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.

IMPORTANT NOTE: Currently, only the dump_modify command can use a region to limit its output. Usage by other commands will be added in the future.

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 NOTE: 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

Default:

The option default is side = in.

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

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:

NOTE: Restart files and their associated commands are not yet fully implemented in SPARTA.

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, read_restart

Default:

```
restart 0
```

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
      N1 = timestep at which 1st run started
    stop value = N2
      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 se
      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 NOTE: 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 4.7 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.

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

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

species command

Syntax:

```
species file ID1 ID2 ...
```

- file = filename that lists species
- ID1, ID2, ... = one or more species names listed in file
- multi-species abbreviations can also be used (see below)

Examples:

```
species species.data air
species species.data air Cl
species species.data O2 N2 NO
species myfile H+ Cl- HCl
```

Description:

Define one or more particle species to use in the simulation. This command can be used as many times as desired to add species to the list of species that the simulation recognizes.

The specified *file* is the name of a file containing definitions for some number of species, not all of which need to used by this 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 recognized abbreviations is as follows:

- air = N, O, NO

These abbreviations can be used in combination with single-species IDs as in the 2nd example above.

The format of 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

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)
- prop4 = rotational relaxtion number (pure)
- prop5 = vibrational degrees of freedom (integer)
- prop6 = vibrational relaxation number (pure)

- prop7 = vibrational temperature (temperature units)
- prop8 = species weight (pure)
- prop9 = multiple of electon charge (1 for a proton)

Restrictions: none

Related commands: none

Default: none

stats command

Syntax:

```
stats N
```

- N = output statistics every N timesteps

Examples:

```
stats 100
```

Description:

Compute and print statistical info (e.g. particle count, temperature) on timesteps that are a multiple of N and at the beginning and end of a simulation 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, stats_modify

Default:

```
stats 0
```

stats_modify command

Syntax:

```
stats_modify keyword value ...
```

- one or more keyword/value pairs may be listed
- keyword = *flush* or *format* or *every*

```
  flush value = yes or no
    format values = int string or float string or M string
      M = integer from 1 to N, where N = # of quantities being printed
      string = C-style format string
    every value = v_name
      v_name = an equal-style variable name
```

Examples:

```
stats_modify flush yes
stats_modify temp myTemp format 3 %15.8g
stats_modify line multi format float %g
```

Description:

Set options for how statistical information is computed and printed by SPARTA.

The *flush* keyword invokes a flush operation after statistical info is written to the log file. This insures the output in that file is current (no buffering by the OS), even if SPARTA halts before the simulation completes.

The *format* keyword sets the numeric format of individual printed quantities. The *int* and *float* keywords set the format for all integer or floating-point quantities printed. The setting with a numeric value M (e.g. format 5 %10.4g) sets the format of the Mth value printed in each output line, e.g. the 5th column of output in this case. If the format for a specific column has been set, it will take precedent over the *int* or *float* setting.

IMPORTANT NOTE: The output values *step* and *np* are stored internally as 8-byte signed integers, rather than the usual 4-byte signed integers. When specifying the "format int" keyword you can use a "%d"-style format identifier in the format string and SPARTA will convert this to the corresponding "%ld" form when it is applied to those keywords. However, when specifying the "format M string" keyword for those keywords, you should specify a string appropriate for an 8-byte signed integer, e.g. one with "%ld".

The *every* keyword allows a variable to be specified which will determine 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, stats_style

Default:

The option defaults are flush = no, format int = "%8d", format float = "%12.8g", and every = non-variable setting provided by the stats command.

stats_style command

Syntax:

```
stats_style arg1 arg2 ...
```

- arg1,arg2,... = list of keywords

```
possible keywords = step, elapsed, dt, cpu, tpcpu, spcpu,
                    np, ntouch, ncomm, nbound, nexit,
                    nscoll, nscheck, ncoll, nattempt,
                    npave, ntouchave, ncommave, nboundave, nexitave,
                    nscollave, nscheckave, ncollave, nattemptave,
                    nreact, nreactave,
                    vol, lx, ly, lz,
                    xlo, xhi, ylo, yhi, zlo, zhi,
                    c_ID, c_ID[I], c_ID[I][J],
                    f_ID, f_ID[I], f_ID[I][J],
                    v_name
```

```
step = timestep
elapsed = timesteps since start of this run
elaplong = timesteps since start of initial run in a series of runs
dt = timestep size
cpu = elapsed CPU time in seconds
tpcpu = time per CPU second
spcpu = timesteps per CPU second
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)
vol = volume of simulation box
lx,ly,lz = simulation box lengths
xlo,xhi,ylo,yhi,zlo,zhi = box boundaries,
c_ID = global scalar value calculated by a compute with ID
c_ID[I] = Ith component of global vector calculated by a compute with ID
c_ID[I][J] = I,J component of global array calculated by a compute with ID
f_ID = global scalar value calculated by a fix with ID
f_ID[I] = Ith component of global vector calculated by a fix with ID
f_ID[I][J] = I,J component of global array calculated by a fix with ID
v_name = scalar value calculated by an equal-style variable with name
```

Examples:

```
stats_style step cpu np
stats_style step cpu spcpu np xlo xhi c_myTemp
```

Description:

Determine what statistical data is printed to the screen and log file.

The values printed by the various keywords are instantaneous values, calculated on the current timestep. The exception is the keywords suffixed by "ave", which print a running total divided by the number of timesteps.

Options invoked by the stats_modify command can be used to set the numeric precision of each printed value, as well as other attributes of the statistics.

The *step* and *elapsed* keywords refer to timestep count. *Step* is the current timestep. *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 *np*, *ntouch*, *ncomm*, *nbound*, *nexit*, *nscoll*, *nscheck*, *ncoll*, *nattempt*, and *nreact* keywords all generate counts for the current timestep.

The *npave*, *ntouchave*, *ncommave*, *nboundave*, *nexitave*, *nscollave*, *nscheckave*, *ncollave*, *nattemptave*, and *nreactave* keywords all generate values that are the cummulative total of the corresponding count divided by *elapsed* = the number of timesteps since the start of the run.

The *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 *vol* keyword is the volume (or area in 2d) of the simulation box.

The *lx*, *ly*, *lz* keywords are the dimensions of the simulation box.

The *xlo*, *xhi*, *ylo*, *yhi*, *zlo*, *zhi* keywords are the boundaries of the simulation box.

The *c_ID* and *c_ID[I]* and *c_ID[I][J]* keywords allow global values calculated by a compute to be output. As discussed on the compute doc page, computes can calculate global, per-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.

The ID in the keyword should be replaced by the actual ID of a compute that has been defined elsewhere in the input script. See the compute command for details. If the compute calculates a global scalar, vector, or array, then the keyword formats with 0, 1, or 2 brackets will reference a scalar value from the compute.

The *f_ID* and *f_ID[I]* and *f_ID[I][J]* keywords allow global values calculated by a fix to be output. As discussed on the fix doc page, fixes can calculate global, per-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.

The ID in the keyword should be replaced by the actual ID of a fix that has been defined elsewhere in the input script. See the fix command for details. If the fix calculates a global scalar, vector, or array, then the keyword formats with 0, 1, or 2 brackets will reference a scalar value from the fix.

The *v_name* keyword allow the current value of a variable to be output. The name in the keyword should be replaced by the variable name that has been defined elsewhere in the input script. Only equal-style variables can be referenced. See the variable command for details. Variables of style *equal* can reference per-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, stats_modify

Default:

```
stats_style step cpu np
```

surf_collide command

Syntax:

```
surf_collide ID style args keyword values ...
```

- ID = user-assigned name for the surface collision model style = *specular* or *diffuse*
- args = arguments for specific style

```
specular args = none
  diffuse args = Tsurf acc
    Tsurf = temperature of surface (temperature units)
            Tsurf can be a variable (see below)
    acc = accommodation coefficient
```

- zero or more keyword/arg pairs may be appended
- keyword = *translate* or *rotate*
- values = values for specific keyword

```
translate args = Vx Vy Vz
    Vx,Vy,Vz = translational velocity of surface (velocity units)
  rotate args = Pz Py Pz Wx Wy Wz
    Px,Py,Pz = point to rotate surface around (distance units)
    Wx,Wy,Wz = angular velocity of surface around point (radians/time)
```

Examples:

```
surf_collide 1 specular
surf_collide 1 diffuse 273.15 0.9
surf_collide heatwall diffuse v_ramp 0.8
surf_collide heatwall diffuse v_ramp 0.8 translate 5.0 0.0 0.0
```

Description:

Define a style for particle-surface collisions. One or more styles can be defined and assigned to different surfaces or simulation box boundaries via the read_surf or bound_modify commands. See Section 4.9 for more details of how SPARTA defines surfaces as collections of geometric elements, triangles in 3d and line segments in 2d.

The ID for a surface collisoin 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.

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 keyword *translate* can only be applied to the *diffuse* style. It models the surface as if it were translating with a constant velocity, specified by the vector (Vx,Vy,Vz). This velocity is added to the final post-collisional velocity of each particle that collides with the surface.

Note that this keyword is designed for use with flat surfaces, such as the simulation box wall. You should define the velocity vector to be tangential to the surface, with no normal component. SPARTA does not check for these conditions.

The keyword *rotate* can only be applied to the *diffuse* style. It models the surface as if it were rotating with a constant angular velocity, specified by the vector W = (Wx,Wy,Wz), around the specified point P = (Px,Py,Pz). When a particle collides with the surface at a point X = (x,y,z), the collision point has a velocity given by V = (Vx,Vy,Vz) = W cross (X-P). This velocity is added to the final post-collisional velocity of the particle.

This keyword can be used to 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.

Restrictions:

The *translate* and *rotate* keywords cannot be used together.

Related commands:

read_surf, bound_modify

Default: none

surf_modify command

Syntax:

```
surf_modify keyword args ...
```

- one or more keyword/arg pairs may be listed
- keyword = *collide*

```
collide args = surf-ID sc-ID
    surf-ID = ID of a surface
    sc-ID = ID of a surface collision model
```

Examples:

```
surf_modify collide 1 2
surf_modify collide sphere bounce
```

Description:

Set parameters for surface elements that have been read in by the read_surf command. For each file of objects read by a read_surf command, a surface ID is assigned. This is the *surf-ID* argument described above as an argument of various keywords.

The *collide* keyword is used to assign a surface collision model to a set of surface elements. The set of surface elements is specified by *surf-ID*. The surface collision model is defined by the surf_collide command. Which collision model to use 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 these surface elements will be computed by the specified surface collision model.

Restrictions:

All surface elements must be assigned to a surface collision model via the *collide* keyword before a simlulation can be performed.

Related commands:

bound_modify

Default: none

timestep command

Syntax:

```
timestep dt
```

- dt = timestep size (time units)

Examples:

```
timestep 2.0
timestep 0.003
```

Description:

Set the timestep size for subsequent simulations.

Restrictions: none

Related commands:

run

Default:

```
timestep 1.0
```

uncompute command

Syntax:

```
uncompute compute-ID
```

- compute-ID = ID of a previously defined compute

Examples:

```
uncompute 2
uncompute lower-boundary
```

Description:

Delete a compute that was previously defined with a compute command.

Restrictions: none

Related commands:

compute

Default: none

undump command

Syntax:

```
undump dump-ID
```

- dump-ID = ID of previously defined dump

Examples:

```
undump mine
undump 2
```

Description:

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

Default: none

unfix command

Syntax:

```
unfix fix-ID
```

- fix-ID = ID of a previously defined fix

Examples:

```
unfix 2
unfix lower-boundary
```

Description:

Delete a fix that was previously defined with a fix command.

Restrictions: none

Related commands:

fix

Default: none

units command

Syntax:

```
units style
```

- style = *cgs* or *si*

Examples:

```
units cgs
```

Description:

This command sets the style of units used for a simulation. It determines the units of all quantities specified in the input script and various input files read by SPARTA, as well as the units of all quantities output to the screen, log file, dump files, and other output files. Typically, this command is used at the very beginning of an input script.

IMPORTANT NOTE: 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
- 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
- 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
```

variable command

Syntax:

```
variable name style args ...
```

- name = name of variable to define
- style = *delete* or *index* or *loop* or *world* or *universe* or *uloop* or *string* or *format* or *getenv* or *file* or *equal* or *particle*

```
delete = no args
  index args = one or more strings
  loop args = N
    N = integer size of loop, loop from 1 to N inclusive
  loop args = N pad
    N = integer size of loop, loop from 1 to N inclusive
    pad = all values will be same length, e.g. 001, 002, ..., 100
  loop args = N1 N2
    N1,N2 = loop from N1 to N2 inclusive
  loop args = N1 N2 pad
    N1,N2 = loop from N1 to N2 inclusive
    pad = all values will be same length, e.g. 050, 051, ..., 100
  world args = one string for each partition of processors
  universe args = one or more strings
  uloop args = N
    N = integer size of loop
  uloop args = N pad
    N = integer size of loop
    pad = all values will be same length, e.g. 001, 002, ..., 100
  string arg = one string
  format args = vname fstr
    vname = name of equal-style variable to evaluate
    fstr = C-style format string
  getenv arg = one string
  file arg = filename
  equal or particle args = one formula containing numbers, stats keywords, math operations, pa
    numbers = 0.0, 100, -5.4, 2.8e-4, etc
    constants = PI
    stats keywords = step, 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), s
    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]
    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 f file values.txt
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).

In the discussion that follows, the "name" of the variable is the arbitrary string that is the 1st argument in the variable command. This name can only contain alphanumeric characters and underscores. The "string" is one or more of the subsequent arguments. The "string" can be simple text as in the 1st example above, it can contain other variables as in the 2nd example, or it can be a formula as in the 3rd example. The "value" is the numeric quantity resulting from evaluation of the string. Note that the same string can generate different values when it is evaluated at different times during a simulation.

IMPORTANT NOTE: When the input script line is encountered that defines a variable of style *equal* or *particle* 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.

IMPORTANT NOTE: When a variable command is encountered in the input script and the variable name has already been specified, the command is ignored. This means variables can NOT be re-defined in an input script (with 2 exceptions, read further). This is to allow an input script to be processed multiple times without resetting the variables; see the jump or include commands. It also means that using the command-line switch -var will override a corresponding index variable setting in the input script.

There are two exceptions to this rule. First, variables of style *string*, *getenv*, *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
```

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 2.5 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 2.5 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 LAMMPS 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 atom-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 *equal* and *particle* styles, a single string is specified which represents a formula that will be evaluated afresh each time the variable is used. If you want spaces in the string, enclose it in double quotes so the parser will treat it as a single argument. For *equal* style variables the formula computes a scalar quantity, which becomes the value of the variable whenever it is evaluated. For *particle* style variables the formula computes one quantity for each particle whenever it is evaluated.

Note that *equal* and *particle* variables can produce different values at different stages of the input script or at different times during a run. For example, if an *equal* variable is used in a fix print command, different values could be printed each timestep it was invoked. If you want a variable to be evaluated immediately, so that the result is stored by the variable instead of the string, see the section below on "Immediate Evaluation of Variables".

The next command cannot be used with *equal* or *particle* style variables, since there is only one string.

The formula for an *equal* or *particle* variable can contain a variety of quantities. The syntax for each kind of quantity is simple, but multiple quantities can be nested and combined in various ways to build up formulas of arbitrary complexity. For example, this is a valid (though strange) variable formula:

```
variable x equal "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.

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,z)
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]
Other variables	v_name

Most of the formula elements produce a scalar value. A few produce a per-particle vector of values. These are the particle vectors, compute references that represent a per-particle vector, fix references that represent a per-particles vector, and variables that are particle-style variables. Math functions that operate on scalar values produce a scalar value; math function that operate on per-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 vector. A formula for a particle-style variable can use formula elements that produce either a scalar value or 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 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 NOTE: 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

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 vectors. In the latter case, 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 sigma^2. For equal-style variables, every processor uses the same random number seed so that they each generate the same sequence of random numbers. For particle-style variables, a unique seed is created for each processor. This effectively generates a different random number for each particle being looped over in the particle-style variable.

IMPORTANT NOTE: Internally, there is just one random number generator for all equal-style variables and one for all particle-style variables. If you define multiple variables (of each style) which use the random() or normal() math functions, then the internal random number generators will only be initialized once.

The ceil(), floor(), and round() functions are those in the C math library. Ceil() is the smallest integer not less than its argument. Floor() if the largest integer not greater than its argument. Round() is the nearest integer to its argument.

The ramp(x,y) function uses the current timestep to generate a value linearly intepolated between the specified x,y values over the course of a run, according to this formula:

```
value = x + (y-x) * (timestep-startstep) / (stopstep-startstep)
```

The run begins on startstep and ends on stopstep. Startstep and stopstep can span multiple runs, using the *start* and *stop* keywords of the run command. See the run command for details of how to do this.

IMPORTANT NOTE: 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 vectors.

The sum() function calculates the sum of all the vector elements. The min() and max() functions find the minimum and maximum element respectively. The ave() function is the same as sum() except that it divides the result by the length of the vector.

The trap() function is the same as sum() except the first and last elements are multiplied by a weighting factor of 1/2 when performing the sum. This effectively implements an integratiion via the trapezoidal rule on the global vector of data. I.e. consider a set of points, equally spaced by 1 in their x coordinate: (1,V1), (2,V2), ..., (N,VN), where the Vi are the values in the global vector of length N. The integral from 1 to N of these points is trap().

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 or atomfile-style variable. Each time the next() function is invoked (i.e. each time the equal-style or atom-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.

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 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. 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 quantities, never both.

c_ID	global scalar, or per-particle vector
c_ID[I]	Ith element of global vector, or Ith column from per-particle 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 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. 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 quantities, never both.

f_ID	global scalar, or per-particle vector
f_ID[I]	Ith element of global vector, or Ith column from per-particle 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.

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-atom vector of numeric values; all other variables store a string. The formula for an equal-style variable can use any style of variable except 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 vector, never both.

v_name	scalar, or per-atom 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* 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 compute, thinking it is current. There are many other commands which could alter the state of the system between runs, causing a variable to evaluate incorrectly.

The solution to this issue is the same as for case (2) above, namely perform a 0-timestep run before the variable is evaluated to insure the system is up-to-date. For example, this sequence of commands would print a 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, jump, include, fix print, print

Default: none

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 NOTE: 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, create_grid

Default: none

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 is typically 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, read_restart

Default: none