SPARTA (20/07/2021)

Release 26 Feb 2021

20/07/2021

Contents:

SPARTA stands for **S**tochastic **PA**rallel **R**arefied-gas **T**ime-accurate **A**nalyzer.

Version info

The SPARTA "version" is the date when it was released, such as 3 Mar 2014. SPARTA is updated continuously. Whenever we fix a bug or add a feature, we release it immediately, and post a notice on this page of the WWW site. Each dated copy of SPARTA contains all the features and bug-fixes up to and including that version date. The version date is printed to the screen and logfile every time you run SPARTA. It is also in the file src/version.h and in the SPARTA directory name created when you unpack a tarball, and at the top of the first page of the manual (this page).

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

Note:

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

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

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

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

Once you are familiar with SPARTA, you may want to bookmark *this page* since it gives quick access to documentation for all SPARTA commands.

CHAPTER 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 *performance* for information on SPARTA performance and scalability, or the Benchmarks section of the SPARTA WWW Site.

SPARTA is a freely-available open-source code, distributed under the terms of the GNU Public License, or sometimes by request under the terms of the GNU Lesser General Public License (LGPL), which means you can use or modify

the code however you wish. The only restrictions imposed by the GPL or LGPL are on how you distribute the code further. See *open-source* below for a brief discussion of the open-source philosophy.

SPARTA is designed to be easy to modify or extend with new capabilities, such as new collision or chemistry models, boundary conditions, or diagnostics. See Section 10 for more details.

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

SPARTA was developed with internal funding at Sandia National Laboratories, a US Department of Energy lab. See Section 1.5 below for more information on SPARTA funding and individuals who have contributed to SPARTA.

1.2 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 10 of the manual, which describes how it can be added to SPARTA.

1.2.1 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

1.2.2 Models

- 3d or 2d or *2d-axisymmetric* domains
- variety of global boundary conditions
- create particles within flow volume
- emit particles from simulation box faces due to flow properties

- emit particles from simulation box faces due to profile defined in file
- emit particles from surface elements due to normal and flow properties
- *ambipolar* approximation for ionized plasmas

1.2.3 Geometry

- *Cartesian, hierarchical 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

1.2.4 Gas-phase collisions and chemistry

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

1.2.5 Surface collisions and chemistry

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

1.2.6 Performance

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

1.2.7 Diagnostics

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

1.2.8 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

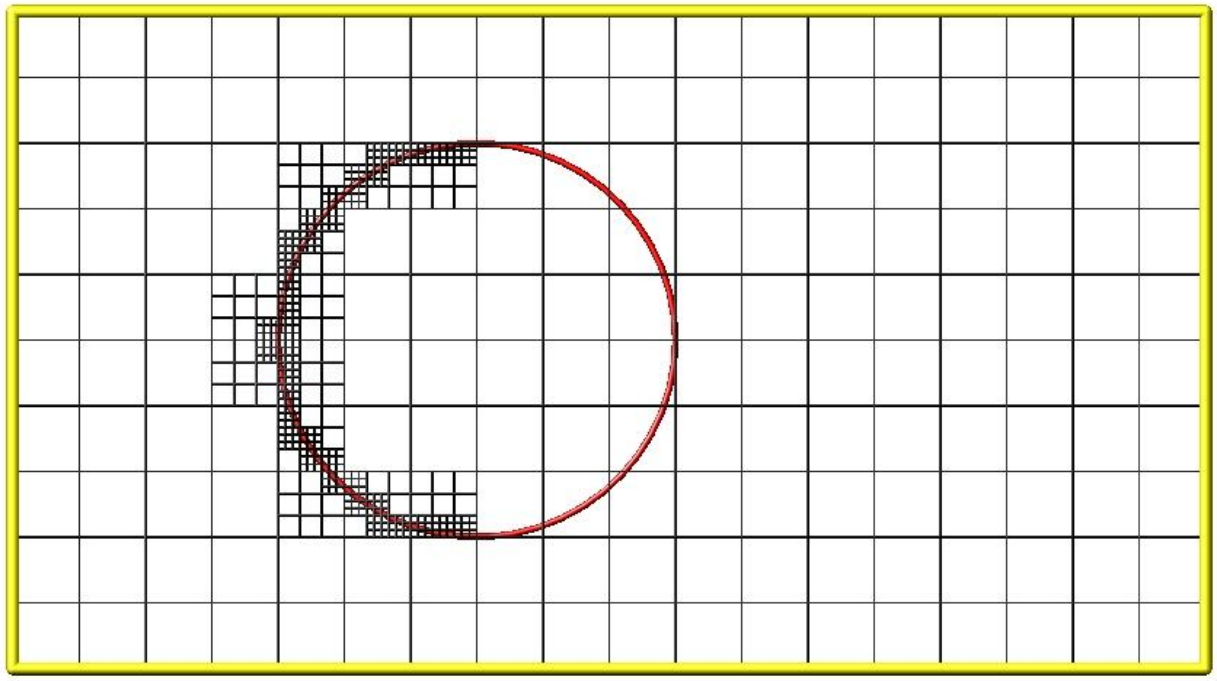
1.2.9 Pre- and post-processing

- Various pre- and post-processing serial tools are packaged with SPARTA; see *Section 9* of the manual.
- Our group has also written and released a separate toolkit called Pizza.py which provides tools for doing setup, analysis, plotting, and visualization for SPARTA simulations. Pizza.py is written in Python and is available for download from the Pizza.py WWW site.

1.3 Grids and surfaces in SPARTA

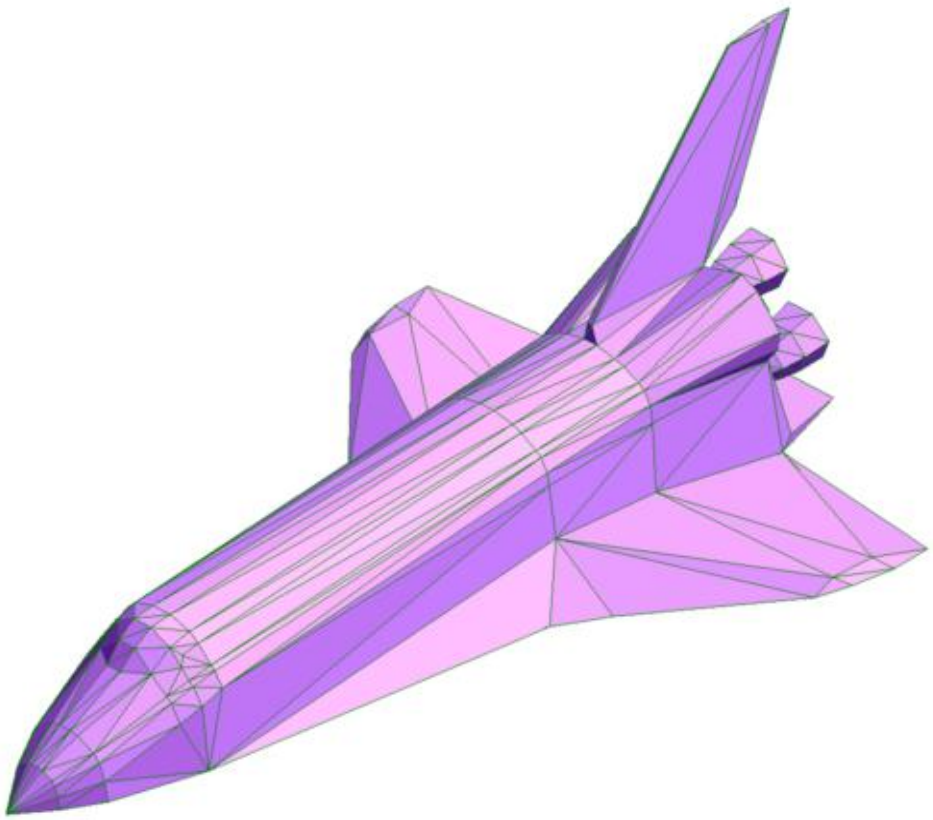
SPARTA overlays a grid over the simulation domain which is used to track particles and to co-locate particles in the same grid cell for performing collision and chemistry operations. SPARTA uses a Cartesian hierarchical grid. Cartesian means that the faces of a grid cell are aligned with the Cartesian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

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



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

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



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

1.4 Open source distribution

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

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

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

2. If you distribute a modified version of SPARTA, it must remain open-source, meaning you distribute it under the terms of the GPL. You should clearly annotate such a code as a derivative version of SPARTA.
3. If you release any code that includes SPARTA source code, then it must also be open-sourced, meaning you distribute it under the terms of the GPL.
4. If you give SPARTA files to someone else, the GPL LICENSE file and source file headers (including the copyright and GPL notices) should remain part of the code.

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

- Point prospective users to the SPARTA WWW Site. Mention it in talks or link to it from your WWW site.
- If you find an error or omission in this manual or on the SPARTA WWW Site, or have a suggestion for something to clarify or include, send an email to the developers.
- If you find a bug, *error-common* describes how to report it.
- If you publish a paper using SPARTA results, send the citation (and any cool pictures or movies) to add to the Publications, Pictures, and Movies pages of the SPARTA WWW Site, with links and attributions back to you.
- The tools sub-directory of the SPARTA distribution has various stand-alone codes for pre- and post-processing of SPARTA data. More details are given in *Additional tools*. If you write a new tool that others will find useful, it can be added to the SPARTA distribution.
- SPARTA is designed to be easy to extend with new code for features like boundary conditions, collision or chemistry models, diagnostic computations, etc. *Modifying & extending SPARTA* of the manual gives details. If you add a feature of general interest, it can be added to the SPARTA distribution.
- The Benchmark page of the SPARTA WWW Site lists SPARTA performance on various platforms. The files needed to run the benchmarks are part of the SPARTA distribution. If your machine is sufficiently different from those listed, your timing data can be added to the page.
- Cash. Small denominations, unmarked bills preferred. Paper sack OK. Leave on desk. VISA also accepted. Chocolate chip cookies encouraged.

1.5 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

CHAPTER 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:

2.2.1 Read this first

Building SPARTA can be non-trivial. You may need to edit a makefile, there are compiler options to consider, additional libraries can be used (MPI, JPEG).

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

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

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

2.2.2 Steps to build a SPARTA executable using make

Step 0

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

```
make g++
or
gmake mac
```

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

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

Note that by default none of the SPARTA optional packages are installed. To build SPARTA with optional packages, see *this section* below.

Step 1

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

Step 2

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

Step 3

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

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

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

Step 4

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

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

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

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

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

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

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

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

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

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

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

Step 5

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

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

Failing this, with these 3 variables you can specify where the mpi.h file is found (via MPI_INC), and the MPI library file is found (via MPI_PATH), and the name of the library file (via MPI_LIB). See Makefile.serial for an example of how this can be done.

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

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

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

Step 6

The 3 FFT variables allow you to specify an FFT library which SPARTA uses (for performing 1d FFTs) when built with its FFT package, which contains commands that invoke FFTs.

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

Otherwise, select which kinds of FFTs to use as part of the FFT_INC setting by a switch of the form -DFFT_XXX. Recommended values for XXX are: MKL or FFTW3. FFTW2 and NONE are supported as legacy options. Selecting -DFFT_FFTW will use the FFTW3 library and -DFFT_NONE will use the KISS library described above. described above.

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

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

select the correct include file. In this case, for FFT_LIB you must also manually specify the correct library, namely -lsfftw or -ldfftw.

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

Step 7

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

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

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

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

Step 8

Note that by default none of the SPARTA optional packages are installed. To build SPARTA with optional packages, see *this section* below, before proceeding to Step 9.

Step 9

That's it. Once you have a correct Makefile.foo, and you have pre-built any other needed libraries (e.g. MPI), all you need to do from the src directory is type one of the following:

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

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

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

2.2.3 Steps to build a SPARTA executable using CMake

Step 0

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

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

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

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

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

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

Step 1

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

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

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

```
make help
```

Now you can try to build the SPARTA binaries with:

```
make
```

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

Step 2

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

```
cd build

# Clear the CMake files
rm -rf CMake*

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

Step 3

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

```
cd build
rm -rf CMake*
```

(continues on next page)

(continued from previous page)

```
cmake -C /path/to/sparta/cmake/presets/<NAME>.cmake -D<OPTION_NAME>=<VALUE> /path/to/
↪sparta/cmake
make
```

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

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

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

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

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

Step 4

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

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

```
-DSPARTA_GZIP
-DSPARTA_JPEG
-DSPARTA_PNG
-DSPARTA_FFMPEG
-DSPARTA_MAP
-DSPARTA_UNORDERED_MAP
-DSPARTA_SMALL
-DSPARTA_BIG
-DSPARTA_BIGBIG
-DSPARTA_LONGLONG_TO_LONG :ul
```

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

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

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

If you use -DSPARTA_MAP, SPARTA will use the STL map class for hash tables. This is less efficient than the unordered map class which is not yet supported by all C++ compilers. If you use -DSPARTA_UNORDERED_MAP, SPARTA will use the unordered_map class for hash tables and will assume it is part of the STL (e.g. this works for

Clang++). The default is to use the unordered map class from the "tri1" extension to the STL which is supported by most compilers. So only use either of these options if the build complains that unordered maps are not recognized.

Use at most one of the -DSPARTA_SMALL, -DSPARTA_BIG, -DSPARTA_BIGBIG settings. The default is -DSPARTA_BIG. These refer to use of 4-byte (small) vs 8-byte (big) integers within SPARTA, as described in src/spatype.h. The only reason to use the BIGBIG setting is if you have a regular grid with more than ~2 billion grid cells or a hierarchical grid with enough levels that grid cell IDs cannot fit in a 32-bit integer. In either case, SPARTA will generate an error message for "Cell ID has too many bits". See Section 6.8 of the manual for details on how cell IDs are formatted. The only reason to use the SMALL setting is if your machine does not support 64-bit integers.

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

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

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

Step 5

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

Now just do:

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

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

Step 6

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

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

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

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

```
# Show which modules are loaded
module list
```

(continues on next page)

(continued from previous page)

```
# Show which modules are available
module avail

module load <MPI> :pre
```

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

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

Verify that cmake found the correct MPI installation:

```
cd build
rm -rf CMake*

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

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

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

Step 7

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

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

```
# Show which modules are loaded
module list

# Show which modules are available
module avail

module load <FFT> :pre
```

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

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

Verify that cmake found the correct MPI installation:

```
cd build
rm -rf CMake*

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

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

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

Step 8

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

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

or:

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

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

Step 9

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

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

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

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

Step 10

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

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

or:

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

```
make -j N
```

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

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

2.2.4 Errors that can occur when making SPARTA

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

Here are two non-obvious errors that can occur:

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

   ```
   make makelist
   make -f Makefile.list g++
   gmake -f Makefile.list mac
   ```

   The first "make" command will create a current Makefile.list with all the file names in your src dir. The 2nd "make" command (make or gmake) will use it to build SPARTA.

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

2.2.5 Additional build tips using make

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

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

2.2.6 Additional build tips using CMake

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

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

2.2.7 Building for a Mac

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

2.2.8 Building for Windows

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

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

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

2.3 Making SPARTA with optional packages

This section has the following sub-sections:

2.3.1 Package basics

The source code for SPARTA is structured as a set of core files which are always included, plus optional packages. Packages are groups of files that enable a specific set of features. For example, the FFT package which includes a command-compute-fft-grid and a 2d and 3d FFT library.

For make:

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

For CMake:

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

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

2.3.2 Including/excluding packages with make

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

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

or

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

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

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

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

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

Packages are included or excluded by typing `make yes-name` or `make no-name`, where `name` is the name of the package in lower-case, e.g. name = fft for the FFT package. You can also type `make yes-all`, or `make no-all` to include/exclude all packages. Type `make package` to see all of the package-related make options.

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

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

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

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

Typing `make package-installed` or `make pi` will show which packages are currently installed in the src directory.

2.3.3 Including/excluding packages with CMake

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

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

or

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

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

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

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

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

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

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

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

2.4 Building SPARTA as a library

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

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

2.4.1 Static library:

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

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

```
make foo mode=lib
```

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

2.4.2 Shared library:

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

```
make foo mode=shlib
```

or:

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

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

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

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

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

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

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

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

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

2.4.3 Additional requirement for using a shared library:

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

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

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

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

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

2.4.4 Calling the SPARTA library:

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

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

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

See *How-to discussions* of the manual for ideas on how to couple SPARTA to other codes via its library interface. See *Python interface to SPARTA* of the manual for a description of the Python wrapper provided with SPARTA that operates through the SPARTA library interface.

The files src/library.cpp and library.h define the C-style API for using SPARTA as a library. See *Library interface to SPARTA* of the manual for a description of the interface and how to extend it for your needs.

2.5 Running SPARTA

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

```
spa_g++
```

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

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

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

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

Commands describes how input scripts are structured and what commands they contain.

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

Here is how you might run one of the benchmarks on a Linux box, using mpirun to launch a parallel job:

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

or:

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

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

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

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

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

SPARTA can run a problem on any number of processors, including a single processor. The random numbers used by each processor will be different so you should only expect statistical consistency if the same problem is run on different numbers of processors.

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

2.6 Command-line options

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

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

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

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

Here are the details on the options:

```
-echo style
```

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

```
-in file
```

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

```
-help
```

Print a list of options compiled into this executable for each SPARTA style (fix, compute, collide, etc). SPARTA will print the info and immediately exit if this switch is used.

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

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

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

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

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

```
device Nd
```

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

```
gpus Ng Ns
```

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

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

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

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

```
threads Nt
```

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

```
-log file
```

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

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

Invoke SPARTA in multi-partition mode. When SPARTA is run on P processors and this switch is not used, SPARTA runs in one partition, i.e. all P processors run a single simulation. If this switch is used, the P processors are split into separate partitions and each partition runs its own simulation. The arguments to the switch specify the number of processors in each partition. Arguments of the form MxN mean M partitions, each with N processors. Arguments of the form N mean a single partition with N processors. The sum of processors in all partitions must equal P. Thus the command "-partition 8x2 4 5" has 10 partitions and runs on a total of 25 processors. Note that with MPI installed on a machine (e.g. your desktop), you can run on more (virtual) processors than you have physical processors.

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

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

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

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

```
-plog file
```

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

```
-pscreen file
```

Specify the base name for the partition screen file, so partition N writes screen information to file.N. If file is none, then no partition screen files are created. This overrides the filename specified in the `-screen` command-line option. This option is useful when working with large numbers of partitions, allowing the partition screen files to be suppressed

(`-pscreen none`) or placed in a sub-directory (`-pscreen replica_files/screen`). If this option is not used the screen file for partition N is screen.N or whatever is specified by the -screen command-line option.

```
-screen file
```

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

```
-suffix style args
```

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

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

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

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

The command-suffix can also be used within an input script to set a suffix, or to turn off or back on any suffix setting made via the command line.

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

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

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

2.7 SPARTA screen output

As SPARTA reads an input script, it prints information to both the screen and a log file about significant actions it takes to setup a simulation. When the simulation is ready to begin, SPARTA performs various initializations and prints the amount of memory (in MBytes per processor) that the simulation requires. It also prints details of the initial state of the system. During the run itself, statistical information is printed periodically, every few timesteps. When the run

concludes, SPARTA prints the final state and a total run time for the simulation. It then appends statistics about the CPU time and size of information stored for the simulation. An example set of statistics is shown here:

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

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

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

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

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

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

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

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

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

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

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

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

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

CHAPTER 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 *Errors* 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, 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 until a newline 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 command-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 command-variable for more details of how strings are assigned to variables and evaluated, and how they can be used in input script commands.

4. The line is broken into "words" separated by whitespace (tabs, spaces). Note that words can thus contain letters, digits, underscores, or punctuation characters.
5. The first word is the command name. All successive words in the line are arguments.
6. If you want text with spaces to be treated as a single argument, it can be enclosed in either double or single quotes. A long single argument enclosed in quotes can even span multiple lines if the `&` character is used, as described above. E.g.

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

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

See the dump modify format or command-print, or command-if for examples. A "#" or "$" character that is between quotes will not be treated as a comment indicator in (2) or substituted for as a variable in (3).

Important: If the argument is itself a command that requires a quoted argument (e.g. using a command-print as part of an command-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 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 *Example problems*, and animated on the SPARTA WWW Site.

A SPARTA input script typically has 4 parts:

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

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

1. Initialization

   Set parameters that need to be defined before the simulation domain, particles, grid cells, and surfaces are defined.

   Relevant commands include command-dimension command-units, and command-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 or read_isurf for surfaces
   - species for particle species properties
   - create_particles for particles

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

   The system can also be load-balanced after the grid and/or particles are defined in the setup stage using the command-balance-grid. The grid can also be adapted before or between simulations using the command-adapt-grid.

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 command-run.

3.4 Commands listed by category

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

Initialization: dimension, package, seed, suffix, units

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

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

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

Actions: adapt_grid, balance_grid, run, scale_particles

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

3.5 Individual commands

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

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

3.5.1 Fix styles

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

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

3.5.2 Compute styles

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

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

3.5.3 Collide styles

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

vss (k)

3.5.4 Surface collide styles

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

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

3.5.5 Surface reaction styles

See the command-surf-react for details of each style.

global	prob

CHAPTER 4

Packages

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

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

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

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

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

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

Table 1: SPARTA packages

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

4.1 FFT package

4.1.1 Contents

Apply Fast Fourier Transforms (FFTs) to simulation data. The FFT library is specified in the Makefile.machine using the FFT_INC, FFT_PATH, and FFT_LIB variables. Supported external FFT libraries that can be specified include FFTW2, FFTW3, and MKL.If no FFT library is specified in the Makefile, SPARTA will use the internal KISS FFT library that is included with SPARTA. See the see discussion in Section 2.2.2: *Step 6*.

4.1.2 Install or un-install with make:

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

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

4.1.3 Install or un-install with CMake:

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

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

4.1.4 Supporting info:

- compute fft/grid
- examples/fft

4.2 KOKKOS package

4.2.1 Contents:

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

You must have a C++14 compatible compiler to use this package.

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

4.2.2 Install or un-install:

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

For multicore CPUs using OpenMP:

Using Makefiles:

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

Using CMake:

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

For Intel KNLs using OpenMP:

Using Makefiles:

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

For NVIDIA GPUs using CUDA:

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

Using CMake:

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

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

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

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

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

Using make:

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

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

Using CMake:

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

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

4.2.3 Supporting info:

- src/KOKKOS: filenames -> commands
- src/KOKKOS/README
- lib/kokkos/README
- Section 5: *Accelerating SPARTA performance*
- accelerate-kokkos
- Section 2.6 `-k` on *Command-line options*
- Section 2.6 `-sf kk`
- Section 2.6 `-pf kokkos`
- package kokkos
- Benchmarks page of web site

CHAPTER 5

Accelerating SPARTA performance

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

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

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

5.1 Measuring performance

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

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

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

Running on one (or a few processors) should give a good estimate of the serial performance and what portions of the timestep are taking the most time. Running the same problem on a few different processor counts should give an

estimate of parallel scalability. I.e. if the simulation runs 16x faster on 16 processors, its 100% parallel efficient; if it runs 8x faster on 16 processors, it's 50% efficient.

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

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

5.2 Packages with optimized styles

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The KOKKOS accelerator package doc page explains:

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

5.3 KOKKOS package

Kokkos is a templated C++ library that provides abstractions to allow a single implementation of an application kernel (e.g. a collision style) to run efficiently on different kinds of hardware, such as GPUs, Intel Xeon Phis, or many-core CPUs. Kokkos maps the C++ kernel onto different backend languages such as CUDA, OpenMP, or Pthreads. The Kokkos library also provides data abstractions to adjust (at compile time) the memory layout of data structures like 2d and 3d arrays to optimize performance on different hardware. For more information on Kokkos, see Github. Kokkos is part of Trilinos. The Kokkos library was written primarily by Carter Edwards, Christian Trott, and Dan Sunderland (all Sandia).

The SPARTA KOKKOS package contains versions of collide, fix, and compute styles that use data structures and macros provided by the Kokkos library, which is included with SPARTA in /lib/kokkos. The KOKKOS package was developed primarily by Stan Moore (Sandia) with contributions of various styles by others, including Dan Ibanez (Sandia), Tim Fuller (Sandia), and Sam Mish (Sandia). For more information on developing using Kokkos abstractions see the Kokkos programmers' guide at /lib/kokkos/doc/Kokkos_PG.pdf.

The KOKKOS package currently provides support for 3 modes of execution (per MPI task). These are Serial (MPI-only for CPUs and Intel Phi), OpenMP (threading for many-core CPUs and Intel Phi), and CUDA (for NVIDIA GPUs). You choose the mode at build time to produce an executable compatible with specific hardware.

Note: Kokkos support within SPARTA must be built with a C++14 compatible compiler. For a list of compilers that have been tested with the Kokkos library, see the Kokkos README.

5.3.1 Building SPARTA with the KOKKOS package with Makefiles:

To build with the KOKKOS package, start with the provided Kokkos Makefiles in /src/MAKE/. You may need to modify the KOKKOS_ARCH variable in the Makefile to match your specific hardware. For example:

- for Sandy Bridge CPUs, set KOKKOS_ARCH=SNB
- for Broadwell CPUs, set KOKKOS_ARCH=BWD
- for K80 GPUs, set KOKKOS_ARCH=KEPLER37
- for P100 GPUs and Power8 CPUs, set KOKKOS_ARCH=PASCAL60,POWER8

5.3.2 Building SPARTA with the KOKKOS package with CMake:

To build with the KOKKOS package, start with the provided preset files in /cmake/presets/. You may need to set -D Kokkos_ARCH_{TYPE}=ON to match your specific hardware. For example:

- for Sandy Bridge CPUs, set -D Kokkos_ARCH_SNB=ON
- for Broadwell CPUs, set -D Kokkos_ARCH_BWD=ON
- for K80 GPUs, set -D Kokkos_ARCH_KEPLER37=ON
- for P100 GPUs and Power8 CPUs, set -D Kokkos_ARCH_PASCAL60=ON, -D Kokkos_ARCH_POWER8=ON

See the *Advanced Kokkos Options:* section below for a listing of all Kokkos architecture options.

5.3.3 Compile for CPU-only (MPI only, no threading):

Use a C++14 compatible compiler and set Kokkos architecture variable as described above. Then do the following:

Using Makefiles:

```
cd sparta/src
make yes-kokkos
make kokkos_mpi_only
```

using CMake:

```
cd build
cmake -C /path/to/sparta/cmake/presets/kokkos_mpi_only.cmake
make
```

5.3.4 Compile for CPU-only (MPI plus OpenMP threading):

Note: To build with Kokkos support for OpenMP threading, your compiler must support the OpenMP interface. You should have one or more multi-core CPUs so that multiple threads can be launched by each MPI task running on a CPU.

Use a C++14 compatible compiler and set KOKKOS architecture variable as described above. Then do the following:

using Makefiles:

```
cd sparta/src
make yes-kokkos
make kokkos_omp
```

using CMake:

```
cd build
cmake -C /path/to/sparta/cmake/presets/kokkos_omp.cmake
make
```

5.3.5 Compile for Intel KNL Xeon Phi (Intel Compiler, OpenMPI):

Use a C++14 compatible compiler and do the following:

using Makefiles: .. code-block:: make

    cd sparta/src make yes-kokkos make kokkos_phi

using CMake: .. code-block:: make

    cd build cmake -C /path/to/sparta/cmake/presets/kokkos_phi.cmake make

5.3.6 Compile for CPUs and GPUs (with OpenMPI or MPICH):

Note: To build with Kokkos support for NVIDIA GPUs, NVIDIA CUDA software version 7.5 or later must be installed on your system.

Use a C++14 compatible compiler and set Kokkos architecture variable in for both GPU and CPU as described above. Then do the following:

using Makefiles: .. code-block:: make

cd sparta/src make yes-kokkos make kokkos_cuda

using CMake: .. code-block:: make

cd build cmake -C /path/to/sparta/cmake/presets/kokkos_cuda.cmake make

5.3.7 Running SPARTA with the KOKKOS package:

All Kokkos operations occur within the context of an individual MPI task running on a single node of the machine. The total number of MPI tasks used by SPARTA (one or multiple per compute node) is set in the usual manner via the mpirun or mpiexec commands, and is independent of Kokkos. The mpirun or mpiexec command sets the total number of MPI tasks used by SPARTA (one or multiple per compute node) and the number of MPI tasks used per node. E.g. the mpirun command in OpenMPI does this via its -np and -npernode switches. Ditto for MPICH via -np and -ppn.

Running on a multi-core CPU:

Here is a quick overview of how to use the KOKKOS package for CPU acceleration, assuming one or more 16-core nodes.

```
mpirun -np 16 spa_kokkos_mpi_only -k on -sf kk -in in.collide         # 1 node, 16 MPI␣
↪tasks/node, no multi-threading
mpirun -np 2 -ppn 1 spa_kokkos_omp -k on t 16 -sf kk -in in.collide  # 2 nodes, 1 MPI␣
↪task/node, 16 threads/task
mpirun -np 2 spa_kokkos_omp -k on t 8 -sf kk -in in.collide          # 1 node,  2 MPI␣
↪tasks/node, 8 threads/task
mpirun -np 32 -ppn 4 spa_kokkos_omp -k on t 4 -sf kk -in in.collide  # 8 nodes, 4 MPI␣
↪tasks/node, 4 threads/task
```

To run using the KOKKOS package, use the "-k on", "-sf kk" and "-pk kokkos" *command-line switches* in your mpirun command. You must use the "-k on" *command-line switch* to enable the KOKKOS package. It takes additional arguments for hardware settings appropriate to your system. Those arguments are *documented here*. For OpenMP use:

```
-k on t Nt
```

The "t Nt" option specifies how many OpenMP threads per MPI task to use with a node. The default is Nt = 1, which is MPI-only mode. Note that the product of MPI tasks * OpenMP threads/task should not exceed the physical number of cores (on a node), otherwise performance will suffer. If hyperthreading is enabled, then the product of MPI tasks * OpenMP threads/task should not exceed the physical number of cores * hardware threads. The "-k on" switch also issues a "package kokkos" command (with no additional arguments) which sets various KOKKOS options to default values, as discussed on the package command doc page.

The "-sf kk" *command-line switch* will automatically append the "/kk" suffix to styles that support it. In this manner no modification to the input script is needed. Alternatively, one can run with the KOKKOS package by editing the input script as described below.

Note: When using a single OpenMP thread, the Kokkos Serial backend (i.e. Makefile.kokkos_mpi_only) will give better performance than the OpenMP backend (i.e. Makefile.kokkos_omp) because some of the overhead to make the code thread-safe is removed.

Note: The default for the package kokkos command is to use "threaded" communication. However, when running on CPUs, it will typically be faster to use "classic" non-threaded communication. Use the "-pk kokkos" *command-line*

switch to change the default package kokkos options. See its doc page for details and default settings. Experimenting with its options can provide a speed-up for specific calculations. For example:

```
mpirun -np 16 spa_kokkos_mpi_only -k on -sf kk -pk kokkos comm classic -in in.collide␣
↪      # non-threaded comm
```

For OpenMP, the KOKKOS package uses data duplication (i.e. thread-private arrays) by default to avoid thread-level write conflicts in some compute styles. Data duplication is typically fastest for small numbers of threads (i.e. 8 or less) but does increase memory footprint and is not scalable to large numbers of threads. An alternative to data duplication is to use thread-level atomics, which don't require duplication. When using the Kokkos Serial backend or the OpenMP backend with a single thread, no duplication or atomics are used. For CUDA, the KOKKOS package always uses atomics in these computes when necessary. The use of atomics instead of duplication can be forced by compiling with the "-DSPARTA_KOKKOS_USE_ATOMICS" compile switch.

5.3.8 Core and Thread Affinity:

When using multi-threading, it is important for performance to bind both MPI tasks to physical cores, and threads to physical cores, so they do not migrate during a simulation.

If you are not certain MPI tasks are being bound (check the defaults for your MPI installation), binding can be forced with these flags:

```
OpenMPI 1.8: mpirun -np 2 -bind-to socket -map-by socket ./spa_openmpi ...
Mvapich2 2.0: mpiexec -np 2 -bind-to socket -map-by socket ./spa_mvapich ...
```

For binding threads with KOKKOS OpenMP, use thread affinity environment variables to force binding. With OpenMP 3.1 (gcc 4.7 or later, intel 12 or later) setting the environment variable OMP_PROC_BIND=true should be sufficient. In general, for best performance with OpenMP 4.0 or better set OMP_PROC_BIND=spread and OMP_PLACES=threads. For binding threads with the KOKKOS pthreads option, compile SPARTA the KOKKOS HWLOC=yes option as described below.

5.3.9 Running on Knight's Landing (KNL) Intel Xeon Phi:

Here is a quick overview of how to use the KOKKOS package for the Intel Knight's Landing (KNL) Xeon Phi:

KNL Intel Phi chips have 68 physical cores. Typically 1 to 4 cores are reserved for the OS, and only 64 or 66 cores are used. Each core has 4 hyperthreads, so there are effectively N = 256 (4*64) or N = 264 (4*66) cores to run on. The product of MPI tasks * OpenMP threads/task should not exceed this limit, otherwise performance will suffer. Note that with the KOKKOS package you do not need to specify how many KNLs there are per node; each KNL is simply treated as running some number of MPI tasks.

Examples of mpirun commands that follow these rules are shown below.

```
Intel KNL node with 64 cores (256 threads/node via 4x hardware threading):
mpirun -np 64 spa_kokkos_phi -k on t 4 -sf kk -in in.collide      # 1 node, 64 MPI␣
↪tasks/node, 4 threads/task
mpirun -np 66 spa_kokkos_phi -k on t 4 -sf kk -in in.collide      # 1 node, 66 MPI␣
↪tasks/node, 4 threads/task
mpirun -np 32 spa_kokkos_phi -k on t 8 -sf kk -in in.collide      # 1 node, 32 MPI␣
↪tasks/node, 8 threads/task
mpirun -np 512 -ppn 64 spa_kokkos_phi -k on t 4 -sf kk -in in.collide  # 8 nodes, 64␣
↪MPI tasks/node, 4 threads/task
```

The -np setting of the mpirun command sets the number of MPI tasks/node. The "-k on t Nt" command-line switch sets the number of threads/task as Nt. The product of these two values should be N, i.e. 256 or 264.

Note: The default for the package kokkos command is to use "threaded" communication. However, when running on KNL, it will typically be faster to use "classic" non-threaded communication. Use the "-pk kokkos" *command-line switch* to change the default package kokkos options. See its doc page for details and default settings. Experimenting with its options can provide a speed-up for specific calculations. For example:

```
mpirun -np 64 spa_kokkos_phi -k on t 4 -sf kk -pk kokkos comm classic -in in.collide␣
↪    # non-threaded comm
```

Note: MPI tasks and threads should be bound to cores as described above for CPUs.

Note: To build with Kokkos support for Intel Xeon Phi coprocessors such as Knight's Corner (KNC), your system must be configured to use them in "native" mode, not "offload" mode.

Running on GPUs:

Use the "-k" *command-line switch* to specify the number of GPUs per node, and the number of threads per MPI task. Typically the -np setting of the mpirun command should set the number of MPI tasks/node to be equal to the # of physical GPUs on the node. You can assign multiple MPI tasks to the same GPU with the KOKKOS package, but this is usually only faster if significant portions of the input script have not been ported to use Kokkos. Using CUDA MPS is recommended in this scenario. As above for multi-core CPUs (and no GPU), if N is the number of physical cores/node, then the number of MPI tasks/node should not exceed N.

```
-k on g Ng
```

Here are examples of how to use the KOKKOS package for GPUs, assuming one or more nodes, each with two GPUs.

```
mpirun -np 2 spa_kokkos_cuda -k on g 2 -sf kk -in in.collide           # 1 node,   2␣
↪MPI tasks/node, 2 GPUs/node
mpirun -np 32 -ppn 2 spa_kokkos_cuda -k on g 2 -sf kk -in in.collide  # 16 nodes, 2␣
↪MPI tasks/node, 2 GPUs/node (32 GPUs total)
```

Note: The default for the package kokkos command is to use "parallel" reduction of statistics along with threaded communication. However, using "atomic" reduction is typically faster for GPUs. Use the "-pk kokkos" *command-line switch* to change the default package kokkos options. See its doc page for details and default settings. Experimenting with its options can provide a speed-up for specific calculations. For example:

```
mpirun -np 2 spa_kokkos_cuda -k on g 2 -sf kk -pk kokkos reduction atomic -in in.
↪collide      # set reduction = atomic
```

Note: Using OpenMP threading and CUDA together is currently not possible with the SPARTA KOKKOS package.

Note: For good performance of the KOKKOS package on GPUs, you must have Kepler generation GPUs (or later). The Kokkos library exploits texture cache options not supported by Telsa generation GPUs (or older).

Note: When using a GPU, you will achieve the best performance if your input script does not use fix or compute styles which are not yet Kokkos-enabled. This allows data to stay on the GPU for multiple timesteps, without being copied back to the host CPU. Invoking a non-Kokkos fix or compute, or performing I/O for stat or dump output will cause data to be copied back to the CPU incurring a performance penalty.

Run with the KOKKOS package by editing an input script:

Alternatively the effect of the "-sf" or "-pk" switches can be duplicated by adding the package kokkos or suffix kk commands to your input script.

The discussion above for building SPARTA with the KOKKOS package, the mpirun/mpiexec command, and setting appropriate thread are the same.

You must still use the "-k on" *command-line switch* to enable the KOKKOS package, and specify its additional arguments for hardware options appropriate to your system, as documented above.

You can use the suffix kk command, or you can explicitly add a "kk" suffix to individual styles in your input script, e.g.

```
collide vss/kk air ar.vss
```

You only need to use the package kokkos command if you wish to change any of its option defaults, as set by the "-k on" *command-line switch*.

Speed-ups to expect:

The performance of KOKKOS running in different modes is a function of your hardware, which KOKKOS-enable styles are used, and the problem size.

Generally speaking, when running on CPUs only, with a single thread per MPI task, the performance difference of a KOKKOS style and (un-accelerated) styles (MPI-only mode)is typically small (less than 20%).

See the Benchmark page of the SPARTA web site for performance of the KOKKOS package on different hardware.

5.3.10 Advanced Kokkos options:

There are other allowed options when building with the KOKKOS package. A few options are listed here; for a full list of all options, please refer to the Kokkos documentation. As above, these options can be set as variables on the command line, in a Makefile, or in a CMake presets file. For default CMake values, see `cmake -LH | grep -i kokkos`.

The CMake option Kokkos_ENABLE_{OPTION} or the makefile setting KOKKOS_DEVICE={OPTION} sets the parallelization method used for Kokkos code (within SPARTA). For example, the CMake option Kokkos_ENABLE_SERIAL=ON or the makefile setting KOKKOS_DEVICES=SERIAL means that no threading will be used. The CMake option Kokkos_ENABLE_OPENMP=ON or the makefile setting KOKKOS_DEVICES=OPENMP means that OpenMP threading will be used. The CMake option Kokkos_ENABLE_CUDA=ON or the makefile setting KOKKOS_DEVICES=CUDA means an NVIDIA GPU running CUDA will be used.

As described above, the CMake option Kokkos_ARCH_{TYPE}=ON or the makefile setting KOKKOS_ARCH={TYPE} enables compiler switches needed when compiling for a specific hardware:

As above, they can be set either as variables on the make command line or in Makefile.machine. This is the full list of options, including those discussed above. Each takes a value shown below. The default value is listed, which is set in the /lib/kokkos/Makefile.kokkos file.

Arch-ID	HOST or GPU	Description
AMDAVX	HOST	AMD 64-bit x86 CPU (AVX 1)
EPYC	HOST	AMD EPYC Zen class CPU (AVX 2)
ARMV80	HOST	ARMv8.0 Compatible CPU
ARMV81	HOST	ARMv8.1 Compatible CPU
$ARMV8_{THUNDERX}$	HOST	ARMv8 Cavium ThunderX CPU
$ARMV8_{THUNDERX2}$	HOST	ARMv8 Cavium ThunderX2 CPU
WSM	HOST	Intel Westmere CPU (SSE 4.2)
SNB	HOST	Intel Sandy/Ivy Bridge CPU (AVX 1)
HSW	HOST	Intel Haswell CPU (AVX 2)
BDW	HOST	Intel Broadwell Xeon E-class CPU (AVX 2 + transactional mem)
SKX	HOST	Intel Sky Lake Xeon E-class HPC CPU (AVX512 + transactional mem)
KNC	HOST	Intel Knights Corner Xeon Phi
KNL	HOST	Intel Knights Landing Xeon Phi
BGQ	HOST	IBM Blue Gene/Q CPU
POWER7	HOST	IBM POWER7 CPU
POWER8	HOST	IBM POWER8 CPU
POWER9	HOST	IBM POWER9 CPU
KEPLER30	GPU	NVIDIA Kepler generation CC 3.0 GPU
KEPLER32	GPU	NVIDIA Kepler generation CC 3.2 GPU
KEPLER35	GPU	NVIDIA Kepler generation CC 3.5 GPU
KEPLER37	GPU	NVIDIA Kepler generation CC 3.7 GPU
MAXWELL50	GPU	NVIDIA Maxwell generation CC 5.0 GPU
MAXWELL52	GPU	NVIDIA Maxwell generation CC 5.2 GPU
MAXWELL53	GPU	NVIDIA Maxwell generation CC 5.3 GPU
PASCAL60	GPU	NVIDIA Pascal generation CC 6.0 GPU
PASCAL61	GPU	NVIDIA Pascal generation CC 6.1 GPU
VOLTA70	GPU	NVIDIA Volta generation CC 7.0 GPU
VOLTA72	GPU	NVIDIA Volta generation CC 7.2 GPU
TURING75	GPU	NVIDIA Turing generation CC 7.5 GPU
AMPERE80	GPU	NVIDIA Ampere generation CC 8.0 GPU
VEGA900	GPU	AMD GPU MI25 GFX900
VEGA906	GPU	AMD GPU MI50/MI60 GFX906
INTEL_GEN	GPU	Intel GPUs Gen9+

The CMake option Kokkos_ENABLE_CUDA_{OPTION} or the makefile setting KOKKOS_CUDA_OPTIONS=*OPTION* are additional options for CUDA. For example, the CMake option Kokkos_ENABLE_CUDA_UVM=ON or the makefile setting KOKKOS_CUDA_OPTIONS="enable_lambda,force_uvm" enables the use of CUDA "Unified Virtual Memory" (UVM) in Kokkos. UVM allows to one to use the host CPU memory to supplement the memory used on the GPU (with some performance penalty) and thus enables running larger problems that would otherwise not fit into the RAM on the GPU. Please note, that the SPARTA KOKKOS package must always be compiled with the CMake option Kokkos_ENABLE_CUDA_LAMBDA=ON or the makefile setting KOKKOS_CUDA_OPTIONS=enable_lambda when using GPUs. The CMake configuration will thus always enable it.

The CMake option Kokkos_ENABLE_DEBUG=ON or the makefile setting KOKKOS_DEBUG=yes is useful when developing a Kokkos-enabled style within SPARTA. This option enables printing of run-time debugging information that can be useful and also enables runtime bounds checking on Kokkos data structures, but may slow down performance.

5.3.11 Restrictions:

Currently, there are no precision options with the KOKKOS package. All compilation and computation is performed in double precision.

CHAPTER 6

How-to discussions

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

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

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 command-dimension to specify a 2d simulation.
- Make the simulation box periodic in z via the command-boundary. This is the default.
- Using the command-create-box, 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 command-read-surf, use 2d objects defined by line segements.

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

6.2 Axisymmetric simulations

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

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

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

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

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

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

6.3 Running multiple simulations from one input script

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

If "multiple simulations" means continue a previous simulation for more timesteps, then you simply use the command-run 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 command-clear 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 command-variable, and command-next, and command-jump to loop over the same input script multiple times with different settings. For example, this script, named `in.flow`

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

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

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

# other commands ...

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

All of the above examples work whether you are running on 1 or multiple processors, but assumed you are running

SPARTA on a single partition of processors. SPARTA can be run on multiple partitions via the "-partition" command-line switch as described in *Command-line options* of the manual.

In the last 2 examples, if SPARTA were run on 3 partitions, the same scripts could be used if the "index" and "loop" variables were replaced with *universe*-style variables, as described in the command-variable. 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.

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

There are four basic kinds of SPARTA output:

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

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

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

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

6.4.1 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.

6.4.2 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.

6.4.3 Statistical output

The frequency and format of statistical output is set by the stats, command-stats-style, and command-stats-modify. The command-stats-style 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 command-compute or command-fix or command-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 command-stats-style.

6.4.4 Dump file output

Dump file output is specified by the command-dump and command-dump-modify. 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 command-compute or command-fix or command-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 command-dump.

6.4.5 Fixes that write output files

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

The command-fix-ave-time 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 command-stats-style (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 command-fix-print 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 command-fix-print is a means to output a wide variety of quantities separate from normal statistical or dump file output.

6.4.6 Computes that process output quantities

The command-compute-reduce 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.

6.4.7 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.

6.4.8 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 command-fix-ave-grid and command-fix-ave-surf.

The command-fix-ave-grid enables time-averaging of per grid vectors. The user specifies one or more quantities as input. These can be per grid vectors or arrays from command-compute or command-fix. 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 command-fix-ave-surf enables time-averaging of per surf vectors. The user specifies one or more quantities as input. These can be per surf vectors or arrays from command-compute or command-fix. 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.

6.4.9 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.

6.4.10 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

6.5 Visualizing SPARTA snapshots

The command-dump-image 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 command-dump 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.

6.6 Library interface to SPARTA

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

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

Library.cpp contains these 4 functions:

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

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

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

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

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

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

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

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

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

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

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

6.7 Coupling SPARTA to other codes

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

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

(1) Define a new command-fix that calls the other code. In this scenario, SPARTA is the driver code. During its timestepping, the fix is invoked, and can make library calls to the other code, which has been linked to SPARTA as a library. See *Modifying & extending SPARTA* of the documentation for info on how to add a new fix to SPARTA.

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

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

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

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

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

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

Section 2.3 of the manual describes how to build SPARTA as a library. Once this is done, you can interface with SPARTA either via C++, C, Fortran, or Python (or any other language that supports a vanilla C-like interface). For example, from C++ you could create one (or more) "instances" of SPARTA, pass it an input script to process, or execute individual commands, all by invoking the correct class methods in SPARTA. From C or Fortran you can make function calls to do the same things. See Section 11 of the manual for a description of the Python wrapper provided with SPARTA that operates through the SPARTA library interface.

The files src/library.cpp and library.h contain the C-style interface to SPARTA. See Section 6.6 of the manual for a description of the interface and how to extend it for your needs.

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

6.8 Details of grid geometry in SPARTA

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

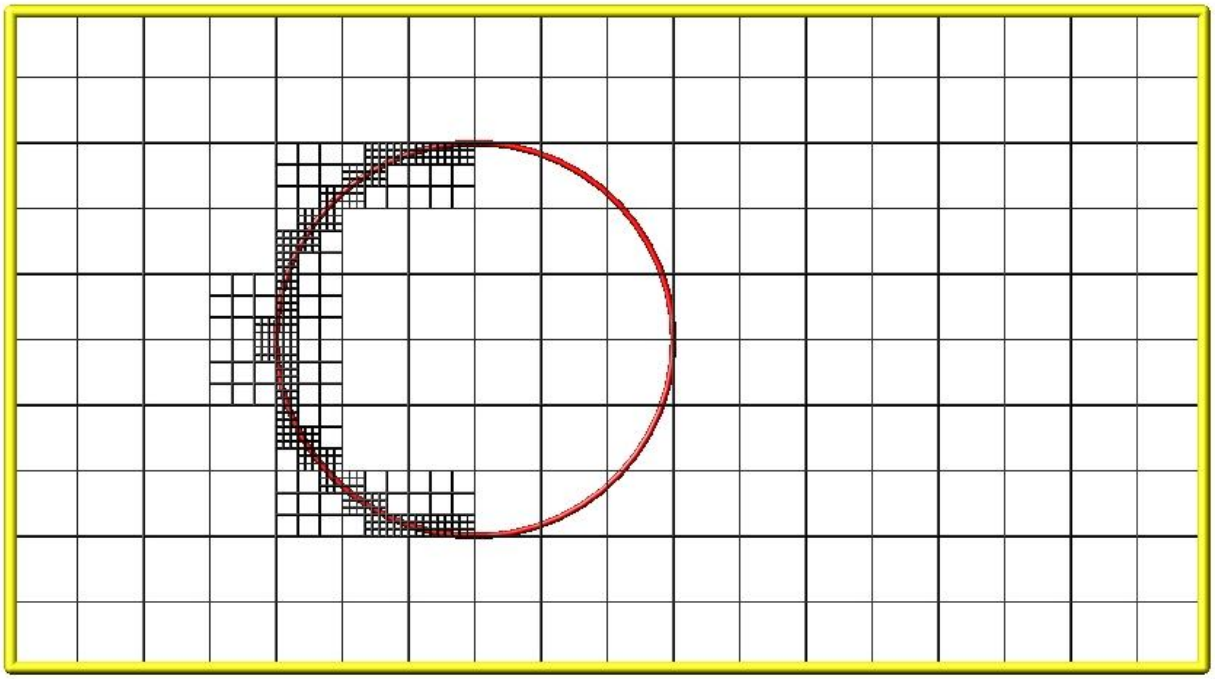
SPARTA uses a Cartesian hierarchical grid. Cartesian means that the faces of a grid cell, at any level of the hierarchy, are aligned with the Cartesian xyz axes. I.e. each grid cell is an axis-aligned pallelpiped or rectangular box.

The hierarchy of grid cells is defined for N levels, from 1 to N. The entire simulation box is a single parent grid cell, conceptually at level 0. It is subdivided into a regular grid of Nx by Ny by Nz cells at level 1. "Regular" means all the Nx*Ny*Nz sub-divided cells within any parent cell are the same size. Each of those cells can be a child cell (no further sub-division) or it 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. The Nx,Ny,Nz values for each level of the grid can be different, but they are the same for every grid cell at the same level. The per-level Nx,Ny,Nz values are defined by the command-create-grid, command-read-grid, or command-fix-adapt.

As described below, each child cell is assigned an ID which encodes the cell's logical position within in the hierarchical grid, as a 32-bit or 64-bit unsigned integer ID. The precision is set by the -DSPARTA_BIG or -DSPARTA_SMALL or -DSPARTA_BIGBIG compiler switch, as described in Section 2.2.2. The number of grid levels that can be used depends on this precision and the resolution of the grid at each level. For example, in a 3d simulation, a level that is refined with a 2x2x2 sub-grid requires 4 bits of the ID. Thus a maximum of 8 levels can be used for 32-bit IDs and 16 levels for 64-bit IDs.

This manner of defining a hierarchy grid 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 used at every level.

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.

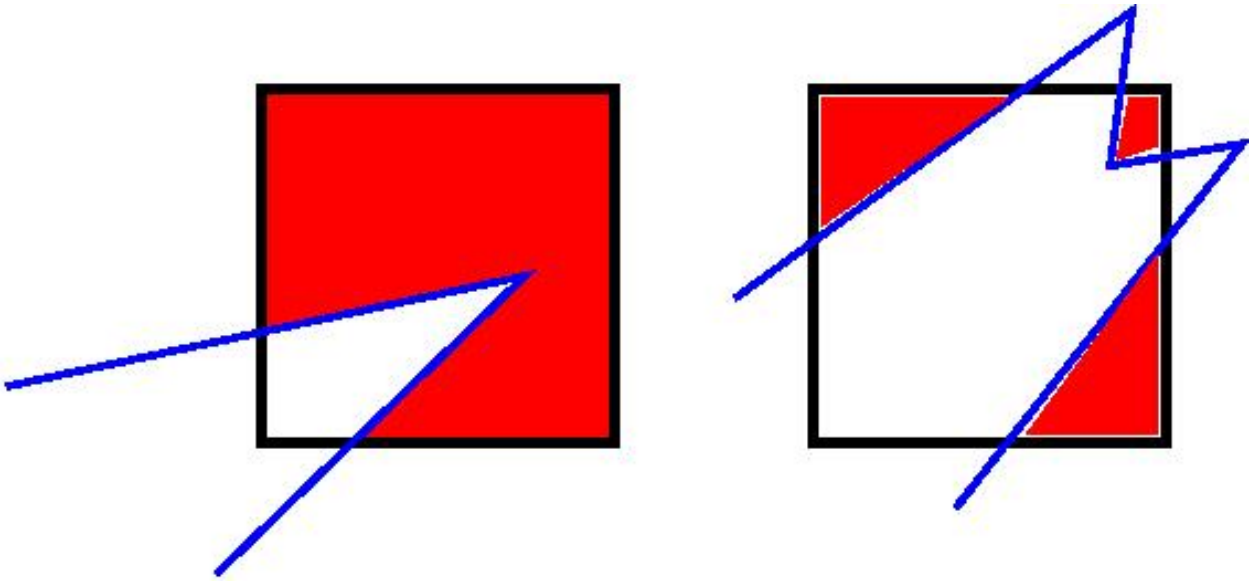


In the rest of the SPARTA manual, the following terminology is used to refer to the cells of the hierarchical grid. The flow region is the portion of the simulation domain that is "outside" any surface objects and is typically filled with particles.

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

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



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

Child cell IDs can be output in integer or string form by the command-dump, using its *id* and *idstr* attributes. The integer form can also be output by the compute property/grid.

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

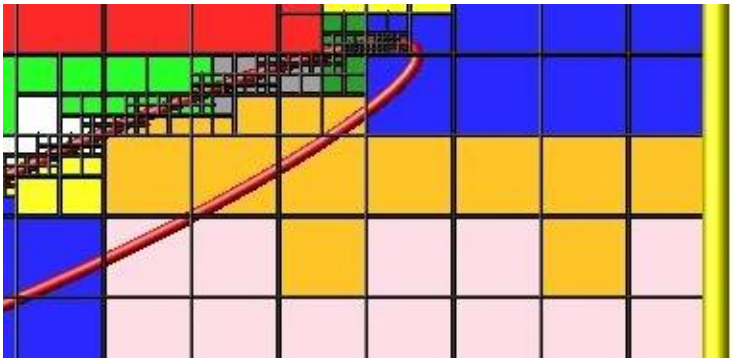
Note that a child cell has the same ID whether it is unsplit, cut, or split. Currently, sub cells of a split cell also have the same ID, though that may change in the future.

The command-create-grid, command-balance-grid, and command-fix-balance 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 which assignment options in these 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 command-create-grid and command-read-grid may produce dispersed assignments, depending on the options used, which can be converted to a clumped assignment by the command-balance-grid.

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 command-read-surf 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.

6.9 Details of surfaces in SPARTA

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

See the command-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 command-read-surf 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 command-surf-modify and command-surf-collide.

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

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

6.10 Restarting a simulation

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

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

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

```
restart             50 tmp.restart
```

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

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

```
read_restart    tmp.restart.50

seed            12345
collide         vss air ar.vss

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

timestep         7.00E-9
run              80
```

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

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

6.11 Using the ambipolar approximation

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

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

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

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

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

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

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

Use the command-fix-ambipolar to specify which species is the ambipolar electron and what (multiple) species are ambipolar ions. This is required for all the other options listed here to work. The fix defines two custom per-particles attributes, an integer vector called "ionambi" which stores a 1 for a particle if it is an ambipolar ion, and a 0 otherwise. And a floating-point array called "velambi" which stores a 3-vector with the velocity of the associated electron for each ambipolar ion or zeroes otherwise. Note that no particles should ever exist in the simulation with a species matching ambipolar electrons. Such particles are only generated (and destroyed) internally, as described above.

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

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

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

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

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

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

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

See the command-surf-react doc page for syntax details. A sample surface reaction data file is provided in data/air.surf. You assign the surface reaction model to surface or the simulation box boundaries via the command-surf-modify and command-bound-modify.

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

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

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

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

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

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

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

6.12 Using multiple vibrational energy levels

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

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

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

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

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

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

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

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

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

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

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

6.13 Surface elements: explicit, implicit, distributed

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

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

The data and attributes of explicit surface elements can be stored in one of two ways. The default is for each processor to store a copy of all the elements. Memory-wise, this is fine for most models. The other option is distributed, where each processor only stores copies of surface elements assigned to grid cells it owns or has a ghost copy of. For models with huge numbers of surface elements, distributing them will use much less memory per processor. Note that a surface element requires about 150 bytes of storage, so storing a million requires about 150 MBytes.

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

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

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

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

These command do not yet support distributed surfaces:

- move_surf
- fix move/surf
- remove_surf

6.14 Implicit surface ablation

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

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

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

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

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

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

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

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

6.15 Transparent surface elements

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

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

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

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

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

- read_surf transparent
- surf_collide transparent
- compute surf

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

The command-surf-collide must be used with its *transparent* model and assigned to all transparent surface elements via the command-surf-modify.

The command-compute-surf can be used to tally the count, mass flux, and energy flux of particles that pass through transparent surface elements. These quantities can then be time averaged via the fix ave/surf command or output via the dump surf command in the usual ways, as described in Section 6.4: *Output from SPARTA (stats, dumps, computes, fixes, variables)*.

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

CHAPTER 7

Example problems

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

Sample log file outputs on different machines and different numbers of processors are included in the directories to compare your answers to. E.g. a log file like log.free.date.foo.P means it ran on P processors of machine "foo", using the dated SPARTA version.

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

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

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

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

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

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

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

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

CHAPTER 8

Performance & scalability

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

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

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

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

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

- 1-processor runs:

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

- 32-processor runs:

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

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

CHAPTER 9

Additional tools

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

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

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

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

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

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

- *dump2cfg tool* - convert a particle dump file to CFG format
- *dump2xyz tool* - convert a particle dump file to XYZ format
- *grid_refine* - refine a grid around a surface
- *implicit_grid* - create a random porous region with implicit surfaces
- *jagged* - create jagged 2d/3d surfaces with explicit surfaces
- *log2txt* - extract columns of info from a log file
- *logplot* - plot columns of info from a log file via GnuPlot
- *paraview* - converters of SPARTA data to ParaView format

- *stl2surf* - convert an STL text file into a SPARTA surface file
- *surf_create* - create a surface file with simple objects
- *surf_transform* - transform surface via tranlate/scale/rotate operations

9.1 dump2cfg tool

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

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

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

9.2 dump2xyz tool

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

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

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

9.3 grid_refine tool

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

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

9.4 implicit_grid tool

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

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

The examples/implicit directory uses these files as input.

9.5 jagged tools

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

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

The examples/jagged directory uses these files as input.

9.6 log2txt tool

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

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

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

9.7 logplot tool

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

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

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

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

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

9.8 paraview tools

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

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

The scripts were developed by Tom Otahal (Sandia).

9.9 stl2surf tool

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

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

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

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

9.10 surf_create tool

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

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

9.11 surf_transform tool

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

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

CHAPTER 10

Modifying & extending SPARTA

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

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

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

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

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

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

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

collide foo mix-ID params.foo 3.0

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

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

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

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

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

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

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

- Think about whether what you want to do would be better as a pre- or post-processing step. Many computations are more easily and more quickly done that way.
- Don't do anything within the timestepping of a run that isn't parallel. E.g. don't accumulate a large volume of data on a single processor and analyze it. This runs the risk of seriously degrading the parallel efficiency.

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

10.1 Compute styles

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

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

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

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

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

10.2 Fix styles

Fix style commands perform operations during the timestepping loop of a simulation. They can define methods which are invoked at different points within the timestep. They can be used to insert particles, perform load-balancing, or perform time-averaging of various quantities. They can also define and maintain new per-particle vectors and arrays that define quantities that move with particles when they migrate from processor to processor or when the grid is rebalanced or adapated. They can also produce output of various kinds, similar to command-compute.

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

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

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

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

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

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

10.3 Region styles

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

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

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

10.4 Collision styles

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

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

init	initialization before a run
modify_params	process style-specific options of the command-collide-modify
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

10.5 Surface collision styles

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

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

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

10.6 Chemistry styles

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

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

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

10.7 Dump styles

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

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

init_style	style-specific initialization before a run
modify_param	process style-specific options of the command-dump-modify
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

10.8 Input script commands

New commands can be added to SPARTA that will be recognized in input scripts. For example, the command-create-particles command-read-surf, and command-run 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.

CHAPTER 11

Python interface to SPARTA

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

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

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

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

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

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

Before using SPARTA from a Python script, you need to do two things. You need to build SPARTA as a dynamic shared library, so it can be loaded by Python. And you need to tell Python how to find the library and the Python wrapper file python/sparta.py. Both these steps are discussed below. If you wish to run SPARTA in parallel from Python, you also need to extend your Python with MPI. This is also discussed below.

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

11.1 Building SPARTA as a shared library

Instructions on how to build SPARTA as a shared library are given in *build-library*. A shared library is one that is dynamically loadable, which is what Python requires. On Linux this is a library file that ends in ".so", not ".a".

For make, from the src directory, type

```
make mode=shlib foo
```

For CMake, from the build directory, tyoe

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

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

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

11.2 Installing the Python wrapper into Python

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

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

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

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

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

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

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

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

You can invoke install.py from the python directory as

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

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

Note that libdir must be a location that is in your default LD_LIBRARY_PATH, like /usr/local/lib or /usr/lib. And pydir must be a location that Python looks in by default for imported modules, like its site-packages dir. If you want to copy these files to non-standard locations, such as within your own user space, you will need to set your `PYTHONPATH` and `LD_LIBRARY_PATH` environment variables accordingly, as above.

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

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

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

```
% make install-python
```

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

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

11.3 Extending Python with MPI to run in parallel

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

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

These include

- pyMPI
- maroonmpi
- mpi4py
- myMPI
- Pypar

All of these except pyMPI work by wrapping the MPI library and exposing (some portion of) its interface to your Python script. This means Python cannot be used interactively in parallel, since they do not address the issue of interactive input to multiple instances of Python running on different processors. The one exception is pyMPI, which

alters the Python interpreter to address this issue, and (I believe) creates a new alternate executable (in place of "python" itself) as a result.

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

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

```
import numpy
```

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

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

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

To install Pypar (version pypar-2.1.4_94 as of Aug 2012), unpack it and from its "source" directory, type

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

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

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

```
import pypar
```

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

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

where test.py contains the lines

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

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

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

11.4 Testing the Python-SPARTA interface

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

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

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

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

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

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

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

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

11.4.1 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.

11.4.2 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.

11.4.3 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.

11.5 Using SPARTA from Python

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

```
from sparta import sparta
```

These are the methods defined by the sparta module. If you look at the file src/library.cpp you will see that they correspond one-to-one with calls you can make to the SPARTA library from a C++ or C or Fortran program.

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


spa.close()               # destroy a SPARTA object


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


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


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


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

Important: Currently, the creation of a SPARTA object from within sparta.py does not take an MPI communicator as an argument. There should be a way to do this, so that the SPARTA instance runs on a subset of processors if desired, but I don't know how to do it from Pypar. So for now, it runs with MPI_COMM_WORLD, which is all the processors. If someone figures out how to do this with one or more of the Python wrappers for MPI, like Pypar, please let us know and we will amend these doc pages.

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

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

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

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

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

For `extract_compute()`, the global, per particle, per grid cell, or per surface element results calulated by the compute can be accessed. What is returned depends on whether the compute calculates a scalar or vector or array. For a scalar, a single double value is returned. If the compute or fix calculates a vector or array, a pointer to the internal SPARTA data is returned, which you can use via normal Python subscripting. See *Output from SPARTA (stats, dumps, computes, fixes, variables)* of the manual for a discussion of global, per particle, per grid, and per surf data, and of scalar, vector, and array data types. See the doc pages for individual computes for a description of what they calculate and store.

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

For `equal-style` variables a single double value is returned and the group argument is ignored. For `particle-style` variables, a vector of doubles is returned, one value per particle, which you can use via normal Python subscripting.

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

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

11.6 Example Python scripts that use SPARTA

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

See the python/README file for more details.

CHAPTER 12

Errors

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

12.1 Common problems

If two SPARTA runs do not produce the same answer on different machines or different numbers of processors, this is typically not a bug. 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 command-echo 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 command-stats 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.

12.2 Reporting bugs

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

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

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

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

12.3 Error & warning messages

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

ERROR: Illegal create_particles command (create_particles.cpp:68)

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

12.3.1 Errors

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

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

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

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

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

Arccos of invalid value in variable formula Argument of arccos() must be between -1 and 1.

Arcsin of invalid value in variable formula Argument of arcsin() must be between -1 and 1.

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

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

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

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

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

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

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

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

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

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

Bound_modify surf_collide ID is unknown Self-explanatory.

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

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

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

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

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

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

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

Cannot balance grid before grid is defined Self-explanatory.

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

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

Cannot create particles before grid is defined Self-explanatory.

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

Cannot create/grow a vector/array of pointers for %s SPARTA code is making an illegal call to the templated memory allocaters, to create a vector or array of pointers.

Cannot create_box after simulation box is defined A simulation box can only be defined once.

Cannot open VSS parameter file %s Self-explantory.

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

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

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

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

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

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

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

Cannot open input script %s Self-explanatory.

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

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

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

Cannot open print file %s Self-explanatory.

Cannot open reaction file %s Self-explanatory.

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

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

Cannot open species file %s Self-explanatory.

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

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

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

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

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

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

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

Cannot read_surf before grid is defined Self-explantory.

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

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

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

Cannot set global surfmax when surfaces already exist This setting must be made before any surfac elements are read via the read_surf command.

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

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

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

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

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

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

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

Cannot use fix inflow on periodic boundary Self-explanatory.

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

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

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

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

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

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

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

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

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

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

Cannot weight cells before grid is defined Self-explanatory.

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

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

Cell ID has too many bits Cell IDs must fit in 32 bits (SPARTA small integer) or 64 bits (SPARTA big integer), as specified by the -DSPARTA_SMALL, -DSPARTA_BIG, or -DSPARTA_BIGBIG options in the low-level Makefile used to build SPARTA. See Section 2.2 of the manual for details. And see Section 6.8 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 in/out other-mark error %d 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 A grid cell was incorrectly marked as inside, outside, or overlapping with surface elements. Please report the issue to the SPARTA developers.

Grid-style variables are not yet implemented Self-explanatory.

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

Inconsistent surface to grid mapping in read_restart When surface elements were mapped to grid cells after reading a restart file, an inconsitent count of elements in a grid cell was found, as compared to the original simulation, which should not happen. Please report the issue to the SPARTA developers.

Incorrect format of parent cell in grid file Number of words in a parent cell line was not the expected number.

Incorrect line format in VSS parameter file Number of parameters in a line read from file is not valid.

Incorrect line format in species file Line read did not have expected number of fields.

Incorrect line format in surf file Self-explanatory.

Incorrect point format in surf file Self-explanatory.

Incorrect triangle format in surf file Self-explanatory.

Index between variable brackets must be positive Self-explanatory.

Input line quote not followed by whitespace An end quote must be followed by whitespace.

Invalid Boolean syntax in if command Self-explanatory.

Invalid Nx,Ny,Nz values in grid file A Nx or Ny or Nz value for a parent cell is <= 0.

Invalid SPARTA restart file The file does not appear to be a SPARTA restart file since it does not have the expected magic string at the beginning.

Invalid attribute in dump grid command Self-explanatory.

Invalid attribute in dump modify command Self-explantory.

Invalid attribute in dump particle command Self-explanatory.

Invalid attribute in dump surf command Self-explanatory.

Invalid balance_grid style for non-uniform grid Some balance styles can only be used when the grid is uniform. See the command doc page for details.

Invalid call to ComputeGrid::post_process_grid() This indicates a coding error. Please report the issue to the SPARTA developers.

Invalid call to ComputeSonineGrid::post_process_grid() This indicates a coding error. Please report the issue to the SPARTA developers.

Invalid cell ID in grid file A cell ID could not be converted into numeric format.

Invalid character in species ID The only allowed characters are alphanumeric, an underscore, a plus sign, or a minus sign.

Invalid collide style The choice of collision style is unknown.

Invalid color in dump_modify command The specified color name was not in the list of recognized colors. See the dump_modify doc page.

Invalid color map min/max values The min/max values are not consistent with either each other or with values in the color map.

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

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

Invalid compute property/grid field for 2d simulation Fields that reference z-dimension properties cannot be used in a 2d simulation.

Invalid compute style Self-explanatory.

Invalid dump frequency Dump frequency must be 1 or greater.

Invalid dump grid field for 2d simulation Self-explanatory.

Invalid dump image filename The file produced by dump image cannot be binary and must be for a single processor.

Invalid dump image persp value Persp value must be >= 0.0.

Invalid dump image theta value Theta must be between 0.0 and 180.0 inclusive.

Invalid dump image zoom value Zoom value must be > 0.0.

Invalid dump movie filename The file produced by dump movie cannot be binary or compressed and must be a single file for a single processor.

Invalid dump style The choice of dump style is unknown.

Invalid dump surf field for 2d simulation Self-explanatory.

Invalid dump_modify threshhold operator Operator keyword used for threshold specification in not recognized.

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

Invalid fix ave/time off column Self-explantory.

Invalid fix style The choice of fix style is unknown.

Invalid flag in grid section of restart file Unrecognized entry in restart file.

Invalid flag in header section of restart file Unrecognized entry in restart file.

Invalid flag in layout section of restart file Unrecognized entry in restart file.

Invalid flag in particle section of restart file Unrecognized entry in restart file.

Invalid flag in peratom section of restart file The format of this section of the file is not correct.

Invalid flag in surf section of restart file Unrecognized entry in restart file.

Invalid image up vector Up vector cannot be (0,0,0).

Invalid immediate variable Syntax of immediate value is incorrect.

Invalid keyword in compute property/grid command Self-explantory.

Invalid keyword in stats_style command One or more specified keywords are not recognized.

Invalid math function in variable formula Self-explanatory.

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

Invalid point index in line Self-explanatory.

Invalid point index in triangle Self-explanatory.

Invalid react style The choice of reaction style is unknown.

Invalid reaction coefficients in file Self-explanatory.

Invalid reaction formula in file Self-explanatory.

Invalid reaction style in file Self-explanatory.

Invalid reaction type in file Self-explanatory.

Invalid read_surf command Self-explanatory.

Invalid read_surf geometry transformation for 2d simulation Cannot perform a transformation that changes z cooridinates of points for a 2d simulation.

Invalid region style The choice of region style is unknown.

Invalid replace values in compute reduce Self-explanatory.

Invalid reuse of surface ID in read_surf command Surface IDs must be unique.

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

Invalid run command start/stop value Self-explanatory.

Invalid run command upto value Self-explanatory.

Invalid special function in variable formula Self-explanatory.

Invalid species ID in species file Species IDs are limited to 15 characters.

Invalid stats keyword in variable formula The keyword is not recognized.

Invalid surf_collide style Self-explanatory.

Invalid syntax in variable formula Self-explanatory.

Invalid use of library file() function This function is called thru the library interface. This error should not occur. Contact the developers if it does.

Invalid variable evaluation in variable formula A variable used in a formula could not be evaluated.

Invalid variable in next command Self-explanatory.

Invalid variable name Variable name used in an input script line is invalid.

Invalid variable name in variable formula Variable name is not recognized.

Invalid variable style in special function next Only file-style or atomfile-style variables can be used with next().

Invalid variable style with next command Variable styles *equal* and *world* cannot be used in a next command.

Ionization and recombination reactions are not yet implemented This error conditions will be removed after those reaction styles are fully implemented.

Irregular comm recv buffer exceeds 2 GB MPI does not support a communication buffer that exceeds a 4-byte integer in size.

Label wasn't found in input script Self-explanatory.

Log of zero/negative value in variable formula Self-explanatory.

MPI_SPARTA_BIGINT and bigint in spatype.h are not compatible The size of the MPI datatype does not match the size of a bigint.

Migrate cells send buffer exceeds 2 GB MPI does not support a communication buffer that exceeds a 4-byte integer in size.

Mismatched brackets in variable Self-explanatory.

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

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

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

Mixture %s fractions exceed 1.0 The sum of fractions must not be > 1.0.

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

Mixture group ID must be alphanumeric or underscore characters Self-explanatory.

Mixture species is not defined One or more of the species ID is unknown.

Modulo 0 in variable formula Self-explanatory.

More than one positive area with a negative area SPARTA cannot determine which positive area the negative area is inside of, if a cell is so large that it includes both positive and negative areas.

More than one positive volume with a negative volume SPARTA cannot determine which positive volume the negative volume is inside of, if a cell is so large that it includes both positive and negative volumes.

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

Next command must list all universe and uloop variables This is to insure they stay in sync.

No dump grid attributes specified Self-explanatory.

No dump particle attributes specified Self-explanatory.

No dump surf attributes specified Self-explanatory.

No positive areas in cell This is an error when calculating how a 2d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

No positive volumes in cell This is an error when calculating how a 3d grid is cut or split by surface elements. It should not normally occur. Please report the issue to the SPARTA developers.

Non digit character between brackets in variable Self-explantory.

Number of groups in compute boundary mixture has changed This mixture property cannot be changed after this compute command is issued.

Number of groups in compute grid mixture has changed This mixture property cannot be changed after this compute command is issued.

Number of groups in compute sonine/grid mixture has changed This mixture property cannot be changed after this compute command is issued.

Number of groups in compute surf mixture has changed This mixture property cannot be changed after this compute command is issued.

Number of groups in compute tvib/grid mixture has changed This mixture property cannot be changed after this compute command is issued.

Number of species in compute tvib/grid mixture has changed This mixture property cannot be changed after this compute command is issued.

Numeric index is out of bounds A command with an argument that specifies an integer or range of integers is using a value that is less than 1 or greater than the maximum allowed limit.

Nz value in read_grid file must be 1 for a 2d simulation Self-explanatory.

Only ylo boundary can be axi-symmetric Self-explanatory. See the boundary doc page for more details.

Owned cells with unknown neighbors = %d One or more grid cells have unknown neighbors which will prevent particles from moving correctly. Please report the issue to the SPARTA developers.

Parent cell child missing Hierarchical grid traversal failed. Please report the issue to the SPARTA developers.

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.

12.3.2 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.

CHAPTER 13

Future and history

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

13.1 Coming attractions

The 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.

13.2 Past versions

Sandia's predecessor to SPARTA is a DSMC code called ICARUS. It was developed in the early 1990s by Tim Bartel and Steve Plimpton. It was later modified and extended by Michael Gallis.

ICARUS is a 2d code, written in Fortran, which models the flow geometry around bodies with a collection of adjoining body-fitted grid blocks. The geometry of the grid cells within in a single block is represented with analytic equations, which allows for fast particle tracking.

Some details about ICARUS, including simulation snapshots and papers, are discussed on this page

Performance-wise ICARUS scaled quite well on several generations of parallel machines, and is still used by Sandia researchers today. ICARUS was export-controlled software, and so was not distributed widely outside of Sandia.

SPARTA development began in late 2011. In contrast to ICARUS, it is a 3d code, written in C++, and uses a hierarchical Cartesian grid to track particles. Surfaces are embedded in the grid, which cuts and splits their flow volumes.

The Authors link on the SPARTA web page gives a timeline of features added to the code since its initial open-source release.