

OINK Users Manual

OINK and MapReduce–MPI Library

<http://www.sandia.gov/~sjplimp/mapreduce.html> – Sandia National Laboratories

Copyright (2009) Sandia Corporation. This software and manual is distributed under the BSD License.

Table of Contents

OINK Documentation.....	1
Version info:.....	1
Goals of OINK.....	1
Contents of OINK Manual.....	2
1. Building OINK.....	3
1.1 Making OINK.....	3
1.2 Building OINK as a library.....	4
1.3 Running OINK.....	5
2.6 Command-line options.....	5
2. OINK Commands.....	7
2.1 Input script operation.....	7
2.2 Parsing rules.....	7
2.3 Input script commands.....	8
3. Adding Callback Functions to OINK.....	10
Map() functions.....	11
Reduce() functions.....	13
Compare() functions.....	14
Hash() functions.....	14
Scan() functions.....	14
4. Adding Commands to OINK.....	16
4.1 Source files for the new class.....	16
4.2 Methods in the new class.....	17
4.3 Calls to the OINK object manager.....	17
4.4 Calling back to map() and reduce() functions.....	22
5. Errors.....	24
5.1 Error & warning messages.....	24
Errors:.....	24
Warnings:.....	27
cc_find command.....	28
cc_stats command.....	29
clear command.....	30
named commands.....	31
degree command.....	34
degree_stats command.....	35
degree_weight command.....	36
echo command.....	37
edge_upper command.....	38
if command.....	39
include command.....	42
input command.....	43
jump command.....	45
label command.....	47
log command.....	48
luby_find command.....	49
mr command.....	50
library commands.....	51
neigh_tri command.....	54
neighbor command.....	55

Table of Contents

next command.....	56
output command.....	58
pagerank command.....	60
print command.....	61
rmat command.....	62
rmat2 command.....	62
set command.....	64
shell command.....	66
sssp command.....	67
sssp2 command.....	67
tri_find command.....	68
variable command.....	69
Math Operators.....	72
Math Functions.....	72
Variable References.....	73
vertex_extract command.....	75
wordfreq command.....	76

OINK Documentation

Version info:

The OINK "version" is the date when it was released, such as 1 Feb 2011. OINK and MR-MPI library are 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 OINK contains all the features and bug-fixes up to and including that version date. The version date is printed to the screen and log file every time you run OINK. It is also in the file `oink/version.h` and in the MR-MPI directory name created when you unpack a tarball.

- If you browse the HTML doc pages for OINK on the MR-MPI WWW site, they always describe the most current version of OINK.
- If you browse the HTML doc pages for OINK included in your tarball, they describe the version you have.
- The [PDF file](#) for OINK on the WWW site or in the tarball is updated about once per month. This is because we don't want it to be part of every patch.

OINK is a simple scripting wrapper around the "MapReduce-MPI [library](#)", and also provides a easy-to-use development framework for writing new MapReduce algorithms and codes. Like the MR-MPI library, OINK will run on any platform (serial or parallel) that supports [MPI](#). Note the MR-MPI library has its own [manual and doc pages](#).

The name OINK is meant to evoke the aroma of the [Apache Pig](#) platform which wraps the [Hadoop](#) MapReduce capabilities with its high-level Pig Latin language. Since OINK has only a small fraction of Pig's capability, it is more the sound of a pig, than the pig itself.

Source code for OINK and the MR-MPI library were developed at Sandia National Laboratories, a US Department of Energy facility. They are freely available for download from the [MR-MPI web site](#) and are distributed under the terms of the modified [Berkeley Software Distribution \(BSD\) License](#). This basically means they can be used by anyone for any purpose. See the LICENSE file provided with the distribution for more details.

The authors of OINK and the MR-MPI library are [Steve Plimpton](#) at and [Karen Devine](#) who can be contacted via email: `sjplimp,kddevin` at `sandia.gov`.

Goals of OINK

- (1) To allow MapReduce algorithms which call the MR-MPI library to be written with a minimum of extraneous code, to work with input/output in various forms, and to be chained together and driven via a simple, yet versatile scripting language.
- (2) To create an archive of `map()` and `reduce()` functions for re-use by different algorithms.
- (3) To provide a scripted interface to the low-level MR-MPI library calls that can speed development/debugging of new algorithms before coding them up in C++ or another language.

We think the first two goals are largely met. See the section on [Adding Commands to OINK](#) and the [named command](#), [input](#), and [output](#) doc pages for details of the first goal. See the section on [Adding Functions to OINK](#) for details of the second goal.

The third goal, however, is only partially met. See the [MR-MPI library commands](#) doc page for its current status. The sticking point here is that in a real programming language you can pass a pointer to an arbitrary data structure to your `map()` or `reduce()` functions, but it is hard to do that from a scripting language using text input without re-inventing something like [Python](#).

Contents of OINK Manual

OINK aims to be a simple scripting interface and development environment and the lightweight documentation reflects that.

Once you are familiar with OINK, you may want to bookmark [this page](#) at `Section_script.html#comm` since it gives quick access to documentation for all OINK commands.

[PDF file](#) of the entire manual, generated by [htmldoc](#)

- [Building OINK](#)
 - 1.1 [Making OINK](#)
 - 1.2 [Building OINK as a library](#)
 - 1.3 [Running OINK](#)
 - 1.4 [Command-line options](#)
- [OINK Scripts](#)
 - 2.1 [Input script operation](#)
 - 2.2 [Parsing rules](#)
 - 2.3 [Input script commands](#)
- [Adding Functions to OINK](#)
 - 3.1 [Map\(\) functions](#)
 - 3.2 [Reduce\(\) functions](#)
 - 3.3 [Compare\(\) functions](#)
 - 3.4 [Hash\(\) functions](#)
 - 3.5 [Scan\(\) functions](#)
- [Adding Commands to OINK](#)
 - 4.1 [Source files for the new class](#)
 - 4.2 [Methods in the new class](#)
 - 4.3 [Calls to the OINK object manager](#)
 - 4.4 [Calling back to map\(\) and reduce\(\) functions](#)
- [Errors](#)
 - 5.1 [Error &warning messages](#)

1. Building OINK

This section describes how to build and run OINK, which is a simple C++ program which wraps the MapReduce-MPI (MR-MPI) library.

- 1.1 [Making OINK](#)
 - 1.2 [Building OINK as a library](#)
 - 1.3 [Running OINK](#)
 - 1.4 [Command-line options](#)
-

1.1 Making OINK

All of the OINK source files are in the "oink" directory of the MR-MPI distribution tarball. The "src" directory contains the source files for the MR-MPI library itself.

These are the 4 steps to building OINK:

- (1) Insure MPI is installed on your system.
- (2) Build the MR-MPI library.
- (3) Use or create a oink/MAKE/Makefile.machine file appropriate for your machine.
- (4) Type "make machine"

Here are more details on each step:

(1) MPI installation

MPI is the message passing interface library, which is likely already installed on your Linux box or Mac, and on most parallel machines. If not, it is freely available. The two most commonly used generic versions are [OpenMPI](#) and [MPICH](#). Download and install one of these if you need to. The default installation location on Linux is under /usr/local.

Or if you do not plan to run the MR-MPI library or OINK in parallel, you can use the provided dummy MPI library in the mpistubs dir. From mpistubs, type "make" and you should get a libmpi.a file. If not, you may need to edit the mpistubs/Makefile.

(2) Build the MR-MPI library

See [this section](#) of the MapReduce-MPI library doc pages for instructions on how to do this. When you have done this a file named src/libmrmpi_machine.a should exist.

(3) Create a Makefile.machine appropriate for your machine.

See the oink/MAKE dir for examples of these. You may be able to use one of these, or edit one that is close to create one for your machine.

The only settings you need to worry about are those in the top section. Set the C++ compiler name and settings appropriate for your box. The only extra libraries used by OINK are MPI and MR-MPI. The settings for the latter are already present in the Makefile.machine files. You may need to change the MPI settings depending on how you did your installation.

If you use the MPI compiler wrappers (mpiCC) for building an MPI-based program like OINK, then you likely need no additional `-I` or `-L` or `LIB` settings.

If you use your system compilers directly, e.g. `g++`, then you will typically need these MPI-related settings:

- An `-I` setting for where to find the file `mpi.h`.
- A `-L` setting for where the MPI library is, `libmpich.a`
- A `LIB` setting for the MPI library, e.g. `-lmpich`

If you are using the provided dummy MPI library (no parallelism), then see `MAKE/Makefile.serial` for how to compile/link with it.

Note that you should insure you build both OINK and the MR-MPI library with the same MPI. If not, confusion will ensue.

(4) Type "make machine"

Do this from the oink directory where its source files are. If you just type "make" you will see what machine options are available (first line of `oink/MAKE/Makefile.machine` files). Some other options are also listed, e.g. for cleaning up.

If you type "make machine", an executable file `oink_machine` should be created, e.g. `oink_linux` or `oink_mac`. If that happens, you're done.

If some error is generated, then you'll need to edit your `oink/MAKE/Makefile.machine`. Find a local make or machine expert to help if you have problems.

If you build OINK on a new kind of machine, for which there isn't a similar `Makefile` for in the `oink/MAKE` directory, send it to the developers and we'll add it to the OINK distribution.

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

1.2 Building OINK as a library

OINK can be built as a library, which can then be called from another application or a scripting language. This is done by typing:

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

where `foo` is the machine name. The first "make" command will create a current `Makefile.lib` with all the file names in your `src` dir. The 2nd "make" command will use it to build OINK as a library. This requires that `Makefile.foo` have a library target (`lib`) and system-specific settings for `ARCHIVE` and `ARFLAGS`. See `Makefile.linux` for an example. The build will create the file `liboink_foo.a` which another application can link to.

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

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

1.3 Running OINK

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

You can test OINK on any of the sample inputs provided in the examples directory. OINK input scripts are named `in.*`.

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

```
cd src
make -f Makefile.linux          # builds src/libmrmpi.a
cd ../oink
make linux                      # builds oink/oink_linux
cd ../examples
mpirun -np 4 ../oink/oink_linux ../doc/*.txt <in.wordcount
```

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

OINK can run a MapReduce calculation on any number of processors, including a single processor.

2.6 Command-line options

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

- `-echo` or `-e`
- `-partition` or `-p`
- `-in` or `-i`
- `-log` or `-l`
- `-screen` or `-s`
- `-var` or `-v`

For example, `oink_ibm` might be launched as follows:

```
mpirun -np 16 oink_ibm -var file tmp.out -log my.log -screen none <in.graph
```

Here are the details on the options:

`-echo style`

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

`-partition 8x2 4 5 ...`

Invoke OINK in multi-partition mode. When OINK is run on *P* processors and this switch is not used, OINK runs in one partition, i.e. all *P* processors run a single calculation. If this switch is used, the *P* processors are split into

separate partitions and each partition runs its own calculation. The arguments to the switch specify the number of processors in each partition. Arguments of the form $M \times N$ 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.

The input script specifies what simulation is run on which partition; see the [variable](#) and [next](#) commands.

`-in file`

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

`-log file`

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

`-screen file`

Specify a file for OINK to write its screen information to. In one-partition mode, if the switch is not used, OINK writes to the screen. If this switch is used, OINK 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.

`-var name value1 value2 ...`

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

2. OINK Commands

This section describes OINK input scripts and what commands are used to define an OINK calculation.

2.1 [Input script operation](#)

2.2 [Parsing rules](#)

2.3 [Input script commands](#)

2.1 Input script operation

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

Note that OINK does not read your entire input script and then perform a calculation 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:

```
set verbosity 1
mr foo
```

does something different than this sequence:

```
mr foo
set verbosity 1
```

In the first case, the MR object created will have its verbosity set to 1. In the latter case it will have the default verbosity of 0, since the set command was not used until after the MR object was created.

Many input script errors are detected by OINK and an ERROR or WARNING message is printed. [This section](#) gives more information on what errors mean. The documentation for each command gives additional information.

2.2 Parsing rules

Each non-blank line in the input script is treated as a command. OINK commands are case sensitive. Pre-defined 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 OINK:

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

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

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

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

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

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

```
print "Value = $t"  
print 'Value = $t'
```

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

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

2.3 Input script commands

There are 4 kinds of OINK commands:

(1) Set command to alter parameters:

- [set](#)

(2) MR–MPI library commands:

- [mr foo](#)
- [foo map ...](#), [foo reduce ...](#), etc

(3) Named commands:

- [input](#)
- [output](#)
- [myfoo params ... -i ... -o ...](#)

(4) Miscellaneous commands that are part of the scripting language:

- [clear](#)
- [echo](#)
- [if](#)
- [include](#)
- [jump](#)
- [label](#)

- [log](#)
 - [next](#)
 - [print](#)
 - [shell](#)
 - [variable](#)
-

Here is a list of all OINK input script commands alphabetically:

clear	echo	if	include	input	jump
label	log	mr	library commands	named commands	next
output	print	set	shell	variable	

These are the named commands currently included in OINK. We will add to this list from time to time. If you write a useful new command, send it to us and we can include it in the distribution.

cc_find	cc_stats
degree	degree_stats
degree_weight	edge_upper
luby_find	neigh_tri
neighbor	rmat
rmat2	sssp
sssp2	tri_find
vertex_extract	wordfreq

Here is a link to the MR-MPI library commands that can be invoked directly from an OINK input script:

MR-MPI library commands

3. Adding Callback Functions to OINK

In the oink directory, the files `map_*.cpp`, `reduce_*.cpp`, `compare_*.cpp`, `hash_*.cpp`, and `scan_*.cpp` each contain one or more functions which can be used as callback methods, passed to MR-MPI library calls, such as the `map()` and `reduce()` operations. This can be done either in [named commands](#) that you write, as described in [this section](#) of the documentation, or in [MR-MPI library commands](#) made directly from an OINK input script.

The collection of these files and callback functions is effectively a library of tools that can be used by new [named commands](#) or your input script to speed the development of new MapReduce algorithms and workflows. Over time, we intend to add new callback function to OINK, and also invite users to send their own functions to the developers for inclusion in OINK.

The `map()`, `reduce()`, and `scan()` callback functions include a `"void *ptr"` as a final argument, which the caller can pass to the callback. This is typically done to enable the callback function to access additional parameters stored by the caller. When doing this with functions listed in the `map_*.cpp`, `reduce_*.cpp`, and `scan_*.cpp` files in OINK, you will want to make the data these pointers point to "portable", so that and "named command" can use it. Thus you should not typically encode class-specific or command-specific data in the structure pointed to. Instead, your caller should create the minimal data structure that the callback function needs to operate, and store the structure in a `map_*.h` file that corresponds to the specific `map_*.cpp` file that contains the function (or `reduce_*.h` or `scan_*.h`). See the file `oink/map_rmat_generate.h` file as an example. It contains the definition of an `RMAT_params` structure, which is used by both the [rmat command](#) and the `map()` methods it uses, listed in `map_rmat_generate.cpp`. Both the `rmat.h` and `map_rmat_generate.cpp` files include the `map_rmat_generate.h` header file to accomplish this. Other commands or callback functions could use the same data structure by including that header file.

The following sections list the various callback function currently included in OINK, and a brief explanation of what each of them does.

Note that `map()` functions come in 4 flavors, depending on what MR-MPI library [map\(\) method](#) is being used. Similarly, `scan()` functions come in 2 flavors, as documented on the [scan\(\) method](#) page. `Map_*.cpp` and `scan_*.cpp` files within OINK can contain any of the 4 or 2 flavors of `map()` and `scan()` methods.

- 3.1 [Map\(\) functions](#)
- 3.2 [Reduce\(\) functions](#)
- 3.3 [Compare\(\) functions](#)
- 3.4 [Hash\(\) functions](#)
- 3.5 [Scan\(\) functions](#)

The documentation below this double line is auto-generated when the OINK manual is created. This is done by extracting C-style documentation text from the `map_*.cpp`, `reduce_*.cpp`, `compare_*.cpp`, `hash_*.cpp`, and `scan_*.cpp` files in the oink directory. Thus you should not edit content below this double line.

In the `*.cpp` files in the oink directory, the lines between a line with a `"/*"` and a line with a `"*/"` are extracted. In the tables below, the first such line of extracted text is assumed to be the function name and appears in the left column. The remaining lines appear in the right columns.

Map() functions

add_label	add a default integer label to each key, key could be vertex or edge input: key = anything, value = NULL output: key = unchanged, value = 1
add_weight	add a default floating point weight to each key, key could be vertex or edge input: key = anything, value = NULL output: key = unchanged, value = 1.0
edge_to_vertex	emit 1 vertex for each edge, just first one input: key = $V_i V_j$, value = NULL output: key = V_i , value = NULL
edge_to_vertex_pair	emit 1 vertex for each edge, just first one input: key = $V_i V_j$, value = NULL output: key = V_i , value = NULL
edge_to_vertices	emit 2 vertices for each edge input: key = $V_i V_j$, value = NULL output: key = V_i , value = NULL

	key = V_j , value = NULL
edge_upper	emit each edge with V_i < V_j , drop self-edges with $V_i = V_j$ input: key = $V_i V_j$, value = NULL output: key = $V_i V_j$, value = NULL, with $V_i < V_j$
invert	invert key and value input: key, value output: key = value, value = key
read_edge	read edges from file, formatted with 2 vertices per line output: key = $V_i V_j$, value = NULL
read_edge_label	read edges and labels from file file format = 2 vertices and integer label per line output: key = $V_i V_j$, value = label
read_edge_weight	read edges and weights from file file format = 2 vertices and floating point weight per line output: key = $V_i V_j$, value = weight
read_vertex_label	

	read vertices and labels from file file format = vertex and integer label per line output: key = V_i , value = label
read_vertex_weight	read vertices and weights from file file format = vertex and floating point weight per line output: key = V_i , value = weight
read_words	read words from file, separated by whitespace output: key = word, value = NULL
rmat_generate	generate graph edges via recursive R-MAT algorithm input: # to generated & R-MAT params extracted from RMAT_struct in ptr output: key = $V_i V_j$, value = NULL

Reduce() functions

count	count number of values associated
-------	-----------------------------------

	with key input: K MV with key and one or more values output: key = unchanged, value = count
cull	eliminate duplicate values input: K MV with key and one or more values (assumed to be duplicates) output: key = unchanged, value = first value

Compare() functions

Hash() functions

Scan() functions

print_edge	print out an edge to a file input: key = Vi Vj, value = NULL
print_string_int	print out key as string and value as int,

	to a file input: key = string, value = int
print_vertex	print out an vertex to a file input: key = Vi, value = NULL

4. Adding Commands to OINK

The purpose of this section is to give details of how to write new [named commands](#) that can be added to OINK and which will be invocable by your input scripts and will interact appropriately with other OINK commands.

OINK is designed to make this easy to do with a minimum of special coding on your part. Several such named commands are included with the OINK distribution; more will be added over time. See [this section](#) of the manual for a list of the current named commands in OINK. We also invite OINK users to send email to the developers with new commands they have written and wish to share, so we can add them to the distribution, attributed to you.

4.1 [Source files for the new class](#)

4.2 [Methods in the new class](#)

4.3 [Calls to the OINK object manager](#)

4.4 [Calling back to map\(\) and reduce\(\) functions](#)

4.1 Source files for the new class

In OINK a named command is a child class that derives from the Command parent class (see `src/command.cpp` and `src/command.h`), meaning that it contains several methods that can be called by the OINK framework. Adding a new named command to OINK is as simple as writing the code for it in two new files (e.g. `foo.cpp` and `foo.h`), dropping them into the `src` directory, and re-building OINK.

It is easiest to understand the description that follows if you look at an example named command in OINK. In what follows we will use the [degree](#) command, contained in `src/degree.cpp` and `src/degree.h` for illustration purposes.

The `*.h` file for a new named command should have lines like these at the top (from the `src/degree.h` file):

```
#ifndef COMMAND_CLASS
CommandStyle(degree, Degree)
#else
```

`CommandStyle(arg1, arg2)` is a macro that gets converted by the OINK build procedure into source code. `Arg1` is the "name" of the named command, which is how you reference it in your input script, e.g. as

```
degree -i graphdir -o out/outfile NULL
```

`Arg2` is the class that implements that command.

The list of all such named commands will appear in the `style_command.h` file after OINK is (re)built, via a `make command`.

The remainder of the `*.h` file (between the `#else` and final pair of `#endif`) is the definition of your new class. Note that you will need to include the `mapreduce.h` file (from the MR-MPI library `src` dir) and the `MAPREDUCE_NS` namespace if your class definition includes any `map()`, `reduce()`, etc callback functions since they have the "KeyValue" class name in their prototype.

4.2 Methods in the new class

When a line in the input script starts with a [named command](#), the associated class is instantiated, its `params()` and `run()` methods are called, and the instance of the class is destroyed.

The constructor of your new class should set two variables, `ninputs` and `noutputs`, which are the number of input and output descriptors it requires you to list in the input script. For the [degree](#) command, `ninputs = noutputs = 1`, as illustrated in the example above.

Your new class is required to define only two methods: `params()` and `run()`.

`Params()` is passed the list of arguments following the command name, excluding the `"-i"` and `"-o"` arguments, which are processed separately by the parent `Command` class. Your `params()` method should parse and check the arguments and generate an error message if the number of arguments is incorrect or any of their values is invalid. Note that the [degree](#) command takes 0 arguments.

`Run()` is called to invoke the meat of your command and it can perform any series of `MapReduce` or other operations you wish, using as many [MapReduce objects](#) (from the `MR-MPI` library) and [MR-MPI objects](#) (managed by OINK) as you wish. The calls that `run()` can make to the OINK object manager (`obj`) are discussed in the next section.

The destructor of your new class should free any memory it has allocated, including any local `MapReduce` objects that it allocated. Note that this is different from [MR-MPI objects](#) and the underlying `MapReduce` objects they wrap, and which are often associated with the input and output descriptors of your command in the input script; those objects are created/destroyed by OINK itself, as discussed in the next section.

4.3 Calls to the OINK object manager

These are the calls that the `run()` method of your new class can make to the OINK object manager. Each is discussed below. Note that you make the calls via the `"obj"` pointer which is visible to your class, e.g. `obj->cleanup()`. This means you should add the line

```
#include "object.h"
```

at the top of your `*.cpp` file.

```
MapReduce *Object::create_mr();
MapReduce *Object::create_mr(int verbosity, int timer,
                              int memsize, int outofcore);
MapReduce *Object::copy_mr(MapReduce *mr);
int Object::permanent(MapReduce *mr);

MapReduce *Object::input(int index);
MapReduce *Object::input(int index,
                          void (*map1)(int, char *, KeyValue *, void *),
                          void *ptr);
MapReduce *Object::input(int index,
                          void (*map2)(int, char *, int, KeyValue *, void *),
                          void *ptr);
MapReduce *Object::input(int index,
                          void (*map1)(int, char *, KeyValue *, void *),
                          void (*map2)(int, char *, int, KeyValue *, void *),
                          void *ptr);
```

```

void Object::output(int index, MapReduce *mr);
void Object::output(int index, MapReduce *mr,
                    void (*scankv)(char *, int, char *, int, void *),
                    void *ptr, int disallow);
void Object::output(int index, MapReduce *mr,
                    void (*scankmv)(char *, int, char *, int, int *, void *),
                    void *ptr, int disallow);
void Object::output(int index, MapReduce *mr,
                    void (*map)(uint64_t, char *, int, char *, int,
                                KeyValue *, void *),
                    void *ptr, int disallow);
void Object::output(int index, MapReduce *mr,
                    void (*reduce)(char *, int, char *, int, int *,
                                    KeyValue *, void *),
                    void *ptr, int disallow);
void Object::output(int index, MapReduce *mr,
                    void (*scankv)(char *, int, char *, int, void *),
                    void (*scankmv)(char *, int, char *, int, int *, void *),
                    void (*map)(uint64_t, char *, int, char *, int,
                                KeyValue *, void *),
                    void (*reduce)(char *, int, char *, int, int *,
                                    KeyValue *, void *),
                    void *ptr, int disallow);

void Object::cleanup();

```

Here is a brief summary of the calls your `run()` method will typically make:

- 1 call to `input()` for each of its `Ninput` input descriptors
- calls to `create_mr()` or `copy_mr()` for additional `MapReduce` objects it uses
- 1 call to `output()` for each of its `Noutput` output descriptors
- final call to `cleanup()` at the end

The details are discussed below.

Your `run()` method should call one of the 4 variants of the `input()` methods one time for each of its inputs. Which variant it calls depends on what forms of input you wish to support, which is related to the "-i" arguments specified with your [named command](#) in the input script and additional options set by the [input](#) command.

Each call takes an "index" argument which is the index of the input descriptor being referenced, from 1 to `Ninputs`. Each call returns a pointer to a `MapReduce` object which will contain the desired input data as key/value (KV) pairs. As the [named command](#) doc page explains, each input descriptor for your command can be specified in the input script as one or more files or directories or as an existing MR-MPI object. For reading files, there are 2 kinds of `map()` methods that can be used to convert the file contents into KV pairs, one where a filename is passed to your callback function, and the other where a chunk of bytes is passed to your callback function. See the [map\(\) method](#) doc page for details.

If you invoke this method:

```
MapReduce *Object::input(int index);
```

then the input descriptor must be specified in your input script as an existing MR-MPI object. No reading of files is allowed.

If you invoke this method:

```
MapReduce *Object::input(int index,
```

```
void (*map1)(int, char *, KeyValue *, void *),
void *ptr);
```

then the input can be specified as either an MR–MPI object or as files which will be processed via the `map1()` callback function which receives a filename as an argument, so that it can open the file, read it, and generate KV pairs.

If you invoke this method:

```
MapReduce *Object::input(int index,
    void (*map2)(int, char *, int, KeyValue *, void *),
    void *ptr);
```

then the input can be specified as either an MR–MPI object or as files which will be processed via the `map2()` callback function which receives a chunk of bytes read from a file as an argument, so that it can convert the byte string into KV pairs. To use this `map2()` method, you would also need to specify an [input](#) command in your input script that setup various options needed to call the [MR–MPI library map\(\) method](#) that uses `map2()` as a callback function.

If you invoke the 4th variant:

```
MapReduce *Object::input(int index,
    void (*map1)(int, char *, KeyValue *, void *),
    void (*map2)(int, char *, int, KeyValue *, void *),
    void *ptr);
```

then both kinds of `map()` callback functions can be specified, `map1()` and `map2()`, and OINK will select which to use depending on what options have been setup via the [input](#) command for this input descriptor.

Note that you have to provide the `map1()` and/or `map2()` callback functions to the `input()` calls, with the correct prototype. As discussed below and on [this doc page](#), they can be static methods in your class, or they can be `map()` methods in separate files in the OINK src directory, which are named `map_*.cpp`.

Also note that if you want to provide maximum flexibility for using your [named command](#), then you should provide one of both flavors of callback `map()` functions for allowing input from files along with input from an existing MR–MPI object. If you do not provide either callback or just one of the two, then input scripts will be limited in what forms of input descriptor they can define.

Your `run()` method should call one of the 6 variants of the `output()` methods one time for each of its outputs. Which variant it calls depends on what forms of output you wish to support, which is related to the `"-o"` arguments specified with your [named command](#) in the input script and additional options set by the [output](#) command.

Each call takes an "index" argument which is the index of the output descriptor being referenced, from 1 to Noutputs. Each call also takes a MapReduce object pointer "mr", which contains the data you wish to output. As the [named command](#) doc page explains, each output descriptor for your command is specified in the input script with 2 parts, either of which can be NULL. The first part is a filename for writing output to files. The second part is the ID of an MR–MPI object which will contain the output. For writing files, there are 4 kinds of callback methods that can be used to write the contents of "mr" to a file. Each of these 4 methods is called with a "FILE *" as its final "void *" argument. This is the file pointer to a file created and opened (and later closed) by OINK which the callback method can write its data to. If you pass your own non-NULL pointer to the callback method via the "void *ptr" argument to the `output()` calls, then it will be appended to the FILE *, so that it can be dereferenced as a 2nd pointer passed to the callback function.

If you invoke this method:

```
void Object::output(int index, MapReduce *mr);
```

then the output descriptor must be specified in your input script as only defining a MR-MPI object for output. No writing to files is allowed. This call will assign the ID specified in your input script to the MR-MPI object that wraps "mr". Also note, that this will remove the ID from any other MR-MPI object that has the same ID. They then become unnamed or temporary MR-MPI objects which will be deleted at the end of your run() method. See further discussion of temporary versus permanent MR-MPI objects in the next section.

If you invoke one of these 2 methods:

```
void Object::output(int index, MapReduce *mr,
                    void (*scankv)(char *, int, char *, int, void *),
                    void *ptr, int disallow);
void Object::output(int index, MapReduce *mr,
                    void (*scankmv)(char *, int, char *, int, int *, void *),
                    void *ptr, int disallow);
```

then the output can be specified as either an MR-MPI object or as files which will be written to via the scankv() or scankmv() callback functions respectively. In the first case, the scankv() function will receive key/value (KV) pairs, one at a time from the "mr" MapReduce object. In the second case, the scankmv() function will receive key/multivalue (KMV) pairs, one at a time from the "mr" MapReduce object. The MapReduce object will be unaltered by this operation. See the [scan\(\) method](#) doc page in the MR-MPI library for details. The "disallow" flag is explained below.

If you invoke one of these 2 methods:

```
void Object::output(int index, MapReduce *mr,
                    void (*map)(uint64_t, char *, int, char *, int,
                                KeyValue *, void *),
                    void *ptr, int disallow);
void Object::output(int index, MapReduce *mr,
                    void (*reduce)(char *, int, char *, int, int *,
                                    KeyValue *, void *),
                    void *ptr, int disallow);
```

then the output can be specified as either an MR-MPI object or as files which will be written to via the map() or reduce() callback functions respectively. In the first case, the map() function will receive key/value (KV) pairs, one at a time from the "mr" MapReduce object. In the second case, the reduce() function will receive key/multivalue (KMV) pairs, one at a time from the "mr" MapReduce object. For the first case, the MapReduce object will typically be unaltered by this operation, since the [MR-MPI library map\(\) method](#) is called with addflag=1, so that the existing KV pairs are preserved. But your map() callback function should not emit any new KV pairs. For the second case, the MapReduce object will be altered by this operation, since the [MR-MPI library reduce\(\) method](#) deletes the KMV pairs and replaces them with new KV pairs which your reduce() callback function may or may not emit. The "disallow" flag is explained below.

If you invoke the 6th variant:

```
void Object::output(int index, MapReduce *mr,
                    void (*scankv)(char *, int, char *, int, void *),
                    void (*scankmv)(char *, int, char *, int, int *, void *),
                    void (*map)(uint64_t, char *, int, char *, int,
                                KeyValue *, void *),
                    void (*reduce)(char *, int, char *, int, int *,
                                    KeyValue *, void *),
                    void *ptr, int disallow);
```

then any of the 4 kinds of callback functions can be specified, namely `scankv()`, `scankmv()`, `map()`, or `reduce()`. Those that you do not wish to provide or that are not compatible with the current state of the MapReduce object "mr" (which will contain either kV or KMV pairs, but not both), can be specified as NULL.

Note that you have to provide each of these 4 callback functions to the `output()` calls, with the correct prototype. As discussed below and on [this doc page](#), they can be static methods in your class, or they can be methods in separate files in the OINK src directory, which are named `scan_*.cpp`, `map_*.cpp`, and `reduce_*.cpp` respectively.

Also note that if you want to provide maximum flexibility for using your [named command](#), then you should provide at least one flavor of a callback function for allowing output to files along with output to an MR–MPI object. If you do not do this, then input scripts will be limited in what forms of output descriptor they can define.

All but the first of the `output()` variants can be called with an optional `disallow` flag which is set to 0 by default. If these methods are called with `disallow=1`, then no output to an MR–MPI object is allowed. This is useful if you expect the `run()` method of your [named command](#) to subsequently change the data stored in the MapReduce object, and thus make the data written to an output file differ from what is stored in the MapReduce object.

Your `run()` method may need to use additional MapReduce objects as workspace, in addition to its inputs. Some of these may end up holding the data you wish to output.

One key point to understand is that the OINK object manager keeps track of two kinds of MR–MPI objects, each of which is a thin wrapper on MapReduce objects which hold your key/value (KV) or key/multivalue (KMV) data. Each MR–MPI object can be "permanent" meaning it has an ID which can be referenced by input script commands. Or it can be "temporary", meaning it has no ID and was created to hold data input from a file or by the function calls discussed below. Permanent MR–MPI objects persist until they are explicitly deleted by your input script. Temporary MR–MPI objects are deleted at the end of your `run()` method; they can be thought of as workspace created and used by your `run()` method.

These two calls create a new temporary MR–MPI object and return a pointer to the MapReduce object contained within it:

```
MapReduce *Object::create_mr();
MapReduce *Object::create_mr(int verbosity, int timer,
                             int memsize, int outofcore);
```

The first variant will use the default settings for the MapReduce object; see the [set](#) command and the [settings doc page](#) of the MR–MPI library for details. The second variant allows you to override a few of the settings with specified values.

This call makes a copy of the "mr" MapReduce object, wraps it in a new temporary MR–MPI object, and returns a pointer to the new MapReduce object:

```
MapReduce *Object::copy_mr(MapReduce *mr);
```

There are two reasons to create new and copied MapReduce objects via these calls, rather than directly invoking MR–MPI library calls within your `run()` method, e.g. do one of these:

```
MapReduce *mr = obj->create_mr();
MapReduce *mr2 = obj->copy(mr);
```

instead of one of these:

```
MapReduce *mr = new MapReduce();
MapReduce *mr2 = mr->copy();
```


The first reason is that OINK will manage the memory associated with the new MR–MPI objects and free them for you at the end of your `run()` method; see the `cleanup()` method discussion below. The second is that you can assign an ID to these temporary MR–MPI objects via the `output()` calls discussed above, which you cannot do if you create the MapReduce object directly yourself. I.e. you cannot pass to an `output()` method a pointer to a MapReduce object that you allocated yourself if that operation will assign an ID (specified in your input script) to the MR–MPI object.

You are of course free to create additional MapReduce objects yourself via direct calls to the MR–MPI library. In this case you should insure you free the objects yourself before the `run()` method ends, so as not to leak memory.

One additional point is that it is fine to do this within your `run()` method, where `mr` is a pointer returned by `obj->create_mr()`:

```
mr = obj->copy(mr);
```

whereas you should not do this:

```
mr = mr->copy();
```

The former simply overwrites the local `mr` pointer, but OINK will manage and free the memory for the underlying MapReduce objects associated with both the original and new `mr` pointers. The latter will leak memory since the underlying MapReduce object associated with the original `mr` pointer is lost.

This call is useful for checking whether a MR–MPI object has been assigned a name or not, when it was used for input or output:

```
int Object::permanent(MapReduce *mr);
```

It is called using a MapReduce object pointer and returns a 1 if the associated MR–MPI object that wraps it has a name, and a 0 if it does not. There are two uses for this call.

First, it can be used after an `input()` call to determine whether the input was done from a file or an existing MR–MPI object. In the former case `permanent()` will return 0, since the new MR–MPI object holding the data is unnamed. In the latter case it will return 1, since the MR–MPI object holding the data was named in your input script as one of the `"-i"` arguments to the [named command](#). If the `run()` method will subsequently alter the MapReduce object and it is permanent, you can make a copy of it, so as to not alter the original.

Second, it can be used after an `output()` call to determine whether the MapReduce object was assigned a name. This will be the case if a MR–MPI ID was specified in your input script as one of the `"-o"` arguments to the [named command](#). If this is the case, you typically do not want to alter the data in the MapReduce object after outputting it. If you wish to further process the data, you can make a copy.

Finally, this method should be called at the end of your `run()` method to free all the temporary MR–MPI objects stored by OINK, and perform other internal cleanup:

```
void Object::cleanup();
```

4.4 Calling back to `map()` and `reduce()` functions

Your `run()` method will typically invoke various methods from the MR–MPI library which involve callback functions, e.g. for performing `map()` or `reduce()` operations.

The [MR–MPI library manual](#) discusses the general rules for passing a pointer to a callback function to a MR–MPI library method. Since you will be doing this from within the class that encodes your [named command](#) you have two choices.

First, you can pass a pointer to a static function declared within your class. This function cannot directly access any class variables, but you can pass it the "this" pointer for the class (as the void * argument to the map() or reduce() function) which the callback function can use to access class variables indirectly, through that pointer. If you do this, then the map() and reduce() methods defined in your class can only be used by that [named command](#).

An alternative is to put your callback functions in their own files, named map_*.cpp for map() functions, reduce_*.cpp for reduce() functions, compare_*.cpp for compare() functions, hash_*.cpp for hash() functions, and scan_*.cpp functions. By doing this the callback functions can be used by any [named command](#) or as arguments to the "MR–MPI library commands" used in an input script to invoke the MR–MPI library methods directly. See the oink/rmat.cpp file, which implements the [rmat](#) command, for an example of a [named command](#) which accesses several of its callback functions in this manner.

Each map_*.cpp file (and reduce_*.cpp, compare_*.cpp, etc) can contain one or more map() (reduce(), compare(), etc) callback functions. These are not class methods, but stand-alone functions. See examples in the oink directory. The header files that contain the prototypes for these functions are named style_map.h, style_reduce.h, etc and are auto-generated when OINK is built. Your [named command](#) class, e.g. rmat.cpp, simply needs to include these style header files in order to use any of the callback functions in OINK. Likewise, any callback function included in one of these files can be accessed by name in your input script when using one of the [MR–MPI library commands](#). Documentation for the collection of map(), reduce(), etc functions is also auto-extracted and included in [this section](#) of the OINK documentation. Instructions on how to pass generic pointers to the callback functions is also discussed in [this section](#).

It is also possible in the run() method of your [named command](#) to select a callback function based on an input script parameter to your command. For example, the input script could list the name of a particular compare() function you wish to use to sort the data in a MapReduce object. By calling the appropriate lookup() method in the MRMPI class (oink/mrpmi.cpp), the parameter string can be converted into a matching function pointer. For example, consider these lines of code:

```
MapReduce *mr = obj->create_mr();
...
CompareFnPtr compare = compare_lookup(userparam);
mr->sort_keys(compare);
```

In this example "userparam" is a string, listed in the input script as a command parameter, which contains a function name, e.g. mySpecialCompare. Assuming that function is included in OINK in a compare_*.cpp file, the compare_lookup() method will be able to match the string to the function and return a pointer to the function which can then be used as an argument to the [sort_keys\(\)](#) MR–MPI library method.

The definition of CompareFnPtr and all other callback function pointers is in the "typedefs.h" file, which can be included at the top of your [named command](#) *.cpp file.

5. Errors

OINK tries to flag errors and print informative error messages so you can fix the problem.

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.oink file or using the [echo command](#) to see it on the screen. For a given command, OINK expects certain arguments in a specified order. If you mess this up, OINK will often flag the error.

Generally, OINK 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. If OINK crashes or hangs without spitting out an error message first then it could be a bug

If you think you have found a bug in OINK, please send an email to the [developers](#) with info about the problem. Anything you can do to isolate the problem and reproduce it on a small data set will be helpful.

5.1 Error &warning messages

These are two alphabetic lists of the [ERROR](#) and [WARNING](#) messages OINK prints out and the reason why. If the explanation here is not sufficient, the documentation for the offending command may help. Grepping the source files for the text of the error message and staring at the source code and comments is also not a bad idea! Note that sometimes the same message can be printed from multiple places in the code.

Errors:

All universe/uloop variables must have same # of values

Self-explanatory.

All variables in next command must be same style

Self-explanatory.

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

Arccos of invalid value in variable formula

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

Arcsin of invalid value in variable formula

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

Cannot open input script %s

Self-explanatory.

Cannot open log.oink

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

Cannot open logfile %s

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

Cannot open logfile

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

Cannot open screen file

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

Cannot open universe log file

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

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

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

Command input is equal-style variable
Only variables that store strings can be used.

Command input variable is unknown
Self-explanatory.

Command outputs must be specified in pairs
Self-explanatory.

Could not create dir for file %s\n
Self-explanatory.

Could not open file in print
This comes from the output "print" routine of the neigh_tri command.

Could not open output file %s for output object()
Self-explanatory.

Divide by 0 in variable formula
Self-explanatory.

Expected floating point parameter in variable definition
The quantity being read is an integer on non-numeric value.

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

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

Failed to reallocate %d bytes for array %s
Your OINK simulation has run out of memory. You need to run a smaller simulation or on more processors.

ID in mr command is already in use
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 OINK to see the offending line.

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

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

Invalid Boolean syntax in if command
Self-explanatory.

Invalid named command switch
Only -i and -o are allowed.

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

Invalid keyword in variable formula
Self-explanatory.

Invalid math function in variable formula
Self-explanatory.

Invalid seed for Marsaglia random # generator

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

Invalid syntax in variable formula
Self-explanatory.

Invalid variable evaluation in variable formula
A variable used in a formula could not be evaluated.

Invalid variable in next command
Self-explanatory.

Invalid variable name in variable formula
Variable name is not recognized.

Invalid variable name
Variable name used in an input script line is invalid.

Invalid variable style with next command
Variable styles *equal* and *world* cannot be used in a next command.

Label wasn't found in input script
Self-explanatory.

Log of zero/negative value in variable formula
Self-explanatory.

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

MR object map command variable is unknown
Variable used as collection of strings is not recognized.

MR object add comand MR object does not exist
Second MR object is not recognized.

MR object command input is equal-style variable
Only variables that store strings can be used.

MR object created by copy already exists
This command creates a new MR object, which cannot already be defined.

MR object map command MR object does not exist
Second MR object is not recognized.

Mismatch in command inputs
Named command defines different number of inputs.

Mismatch in command outputs
Named command defines different number of outputs.

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.

Object input() invoked with invalid index
Index argument must be from 1 to Ninputs.

Object input() map function does not match input mode
Input command is used to select map mode (mmode) which much match map() function.

Object input() with no map function
Input script specified input from file, but no map() method was provided by named command.

Object output() as MR object not allowed
Input script speficed output as MR object, but named command invoked disallow flag.

Object output() called for unknown MR object
A MapReduce object unknown to the object manager was passed to the output() method by a named command.

Object output() invoked with invalid index
Index argument must be from 1 to Noutputs.

Object permanent() called for unknown MR object
A MapReduce object unknown to the object manager was passed to the permanent() method by a named

command.

Ouput MR ID must be alphanumeric or underscore characters
Self-explanatory.

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 LAMMPS is running on.

RMAT a,b,c,d must sum to 1
Self-explanatory.

RMAT fraction must be < 1
Self-explanatory.

Sqrt of negative value in variable formula
Self-explanatory.

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

Too many edges for one vertex in reduce first_degree
Number of edges of one vertex exceeds max integer in tri_find command. Will never be able to emit N^2 angles.

Tour + vertex reduce exceeds one block
No matching end double quote was found following a leading double quote.

Universe/uloop variable count < # of partitions
A universe or uloop style variable must specify a number of values \geq to the number of processor partitions.

Unknown command: %s
The command is not known to OINK. Check the input script.

Variable name must be alphanumeric or underscore characters
Self-explanatory.

World variable count doesn't match # of partitions
A world-style variable must specify a number of values equal to the number of processor partitions.

Warnings:

Placeholder

No warnings are yet defined in OINK.

cc_find command

Syntax:

```
cc_find nthresh -i in1 -o out1.file out1.mr
```

- nthresh = threshold size at which components are split across processors
- in1 = graph edges: Key = $V_i V_j$, Value = NULL
- out1 = components: Key = V_i , Value = C_i

Examples:

```
cc_find 1000 -i mre -o cc.list cc
```

Description:

This is a named command which finds all the connected components (CCs) in an undirected graph. A connected component is a set of vertices where each is connected to one or more other vertices in the set via an edge. The CCs are found via the MapReduce algorithm of [\(Cohen\)](#) discussed in his paper with extensions described in the paper of [\(Plimpton\)](#) which attempt to load-balance the calculation across processors when one or a few very large components exist in the graph.

See the [named command](#) doc page for various ways in which the `-i` inputs and `-o` outputs for a named command can be specified.

In1 stores a set of edges, assumed to have no duplicates or self-edges. This means that either (V_i, V_j) or (V_j, V_i) appears, but not both. Also (V_i, V_i) does not appear. The input is unchanged by this command.

Out1 will store the assignment of each vertex in the graph to a component ID. The component ID is the smallest vertex ID of any vertex in the component.

Related commands:

[cc_stats](#)

(Cohen) Cohen, "Graph Twiddling in a MapReduce World", Computing in Science and Engineering, 11, 29–41 (2009).

(Plimpton) Plimpton and Devine, "MapReduce in MPI for Large-Scale Graph Algorithms", to appear in Parallel Computing (2011).

cc_stats command

Syntax:

```
cc_stats -i in1
```

in1 = component assignment for each vertex: Key = V_i , Value = C_i

Examples:

```
cc_stats -i ccdir
```

Description:

This is a named command which summarizes and prints out the statistics on connected components (CCs) that each vertex in a graph belongs to. For each component size n , one line is printed to the screen with the number of components of size n . This is done in sorted order.

See the [named command](#) doc page for various ways in which the `-i` inputs and `-o` outputs for a named command can be specified.

In1 stores a set of vertices V_i and the component ID C_i which each is assigned to. Typically this will have been computed previously by the [cc_find](#) command. The input is unchanged by this command.

This command produces no output, other than what is written to the screen.

Related commands:

[cc_find](#)

clear command

Syntax:

```
clear
```

Examples:

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

Description:

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

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

Related commands: none

named commands

Syntax:

```
cmomand-name params ... -i input1 input2 ,... -o output1.file output1.ID output2.file output2.ID ...
```

- `command-name` = name of command
- `params` = zero or more params required by command
- `-i` = start of input definitions required by command
- `inputN` = list of 0 or more input objects
- `-o` = start of output definitions to command
- `outputN.file` = list of 0 or more output files
- `outputN.ID` = list of 0 or more output MR-MPI objects

Examples:

```
wordfreq 5 -i v_files -o NULL NULL
rmat 10 8 0.25 0.25 0.25 0.25 0.0 12345 -o tmp.rmat NULL
degree -i graph/edges -o degree/degree degree
```

Description:

Invoke a named command with the list of parameters it requires, as well as the list of input and output objects it expects. In OINK a named command is a child class that derives from the `Command` parent class, meaning that it contains several methods that can be called from the OINK framework. See [this section](#) of the manual for details on how to write new named commands and add them to OINK. The list of named commands currently included with OINK are listed on [this page](#). They are also listed in the source code in the file `src/style_command.h` which is auto-generated each time that OINK is built.

Each named command has a "name", defined in the `*.h` file for the class, which is the command name used in the input script to invoke the command, e.g. `wordfreq` or `rmat` or `degree` in the examples above.

Any arguments that follow the command name, upto a `-i` or `-o` argument are passed as *params* to the command before it is invoked, so that it can process and store them as needed. The number and nature of these parameters are defined by the command itself and it should generate errors if they are not specified correctly. The code that processes parameters can be written to allow for optional parameters and keywords within the list of *params*.

The `-i` and `-o` arguments can be listed in either order. The arguments that follow each of them, either between them, or upto the end of the command, are passed to an "input" and "output" processing routine within the command class. Each command requires a specific number of input and output "definitions", as explained below. Input definitions are single arguments. Output definitions are pairs of arguments. If zero input (or output) definitions are required by the command, then the `-i` (or `-o`) argument need not be specified. If 2 output definitions are required, then 4 arguments must follow the `-o`.

Typically each required input definition is a form of data input that the command requires. It can come from reading one or more files or from an MR-MPI object that already exists. Similarly, each required output definition is a form of data output that the command produces. It can be stored either in one or more files or in an MR-MPI object that the command creates. In OINK, an MR-MPI object is a thin wrapper on a MapReduce object created via the [MR-MPI library](#). See [this doc page](#) for more discussion of MR-MPI objects and input script operations that can be performed on them directly.

Each input definition *inputN* is one of three things.

First, it can be the ID of an existing MR–MPI object, which wraps a MapReduce object which contains key/value pairs. If *inputN* matches such an ID, then the second or third options are not considered. In this case, it is assumed that the data stored within the MapReduce object is already in the form that would be produced by the `map()` method that would read the input from one or more file or directory names.

Second, the *inputN* can be the path name of a file or directory. Third, it can be a variable defined elsewhere in the OINK input script that contains one or more strings. In the third case *inputN* should be specified as `v_name`, where name is the name of the variable. All the different styles of variables (except equal–style) store strings; see the [variable](#) command for details. Also note that there is a [command–line option](#) `–var` or `–v` which can be specified when OINK is executed to store a list of strings in an index–style variable. The strings are treated as a list of file or directory names. Thus in both the first or second case the effect is that a list of one or more file/directory names is passed to the command. The command creates a temporary, unnamed MR–MPI object and invokes a `map()` method within it, as specified in the code of the command class, using the list of file/directory names as input.

There are several options available which influence how the list of strings specified in the input script are converted into actual file/directory paths passed to the `map()` method. This include wildcard characters "%" and "*". See the [input](#) command for details.

Each required output definition is a pair of arguments: *outputN.file* and *outputN.ID*. Either or both can be specified as NULL if no output in that form is desired.

The *outputN.file* argument is the path name of a file. A `map()` or `reduce()` or `scan()` method, as specified in the code of the command class, will be invoked which will write data to that file when the command is finished. A processor–ID (0 to `Nprocs–1`) will be appended to the filename, so that when running on multiple processors, multiple files will be created.

If the specified path name does not entirely exist, additional directories in the path name will be created as needed. Also, there are several options available which influence how the file name specified in the input script is converted into the file name actually opened by OINK and written to by the `map()`, `reduce()`, or `scan()` method. This include wildcard characters "%" and "*". See the [output](#) command for details.

The *outputN.ID* argument is the ID of an MR–MPI object which wraps a MapReduce object. The code in the command class will have created or altered the MR–MPI object and its associated MapReduce object and populated it with data. As a final step, the specified ID is assigned to that MR–MPI object. If the ID is already in use, then the name is removed from the other MR–MPI object. This means that if an *outputN.ID* is the same as an *inputN* to the command then the output will effectively overwrite that input.

When the command completes, named MR–MPI objects persist so that they can be used in subsequent input script commands. All unnamed MR–MPI objects are deleted.

When any named command is executed, its elapsed execution time is stored internally by OINK. This value can be accessed by the keyword "time" in an [equal–style variable](#) and printed out in the following manner:

```
variable t equal time
rmat 10 8 0.25 0.25 0.25 0.25 0.0 12345 -o tmp.rmat NULL
print "Time for RMat generation = $t"
```

Related commands:

[MR-MPI library commands](#), [mr](#), [MR-MPI library documentation](#), [how to write named commands](#), [input](#), [output](#)

degree command

Syntax:

```
degree dupflag -i in1 -o out1.file out1.mr
```

- dupflag = 1/2 for counting edge once/twice
- in1 = graph edges: Key = Vi Vj, Value = NULL
- out1 = degree of each vertex: Key = Vi, Value = degree

Examples:

```
degree 1 -i mrv -o degree.list NULL
```

Description:

This is a named command which calculates the degree of each vertex in a graph.

If the dupflag is 2, then each edge increments the degree of both of its vertices. If the dupflag is 1, then each edge only increments the degree of the first of its vertices. The former is usually more appropriate for undirected graphs; the latter for directed graphs, in which case the out-degree of each vertex is being calculated.

See the [named command](#) doc page for various ways in which the `-i` inputs and `-o` outputs for a named command can be specified.

In1 stores a set of edges. No assumption is made about duplicates or self edges, i.e. (Vi,Vj) may appear multiple times, both (Vi,Vj) or (Vj,Vi) may appear, as may (Vi,Vi). The input is unchanged by this command.

Out1 will store the degree count of each vertex.

Related commands:

[neighbor](#), [degree_stats](#)

degree_stats command

Syntax:

```
degree_stats dupflag -i in1
```

dupflag = 1/2 for counting edge once/twice in1 = graph edges: Key = $V_i V_j$, Value = NULL

Examples:

```
degree_stats 2 -i edges
```

Description:

This is a named command which summarizes and prints out the degree statistics of an undirected graph. For each degree d , one line is printed to the screen with the number of vertices of degree d . This is done in sorted order.

If the dupflag is 2, then each edge increments the degree of both of its vertices. If the dupflag is 1, then each edge only increments the degree of the first of its vertices. The former is usually more appropriate for undirected graphs; the latter for directed graphs, in which case the out-degree of each vertex is being calculated. The latter can also be useful if the graph represents a sparse matrix and you want the statistics on non-zeroes in each row of the matrix.

See the [named command](#) doc page for various ways in which the `-i` inputs and `-o` outputs for a named command can be specified.

In1 stores a set of edges. No assumption is made about duplicates or self edges, i.e. (V_i, V_j) may appear multiple times, both (V_i, V_j) or (V_j, V_i) may appear, as may (V_i, V_i) . The input is unchanged by this command.

This command produces no output, other than what is written to the screen.

Related commands:

[degree](#)

degree_weight command

Syntax:

```
degree_weight dupflag -i in1 -o out1.file out1.mr
```

- dupflag = 1/2 for counting edge once/twice
- in1 = graph edges: Key = Vi Vj, Value = NULL
- out1 = degree of each vertex: Key = Vi, Value = degree

Examples:

```
degree 1 -i mrv -o degree.list NULL
```

Description:

This is a named command which calculates the degree of each vertex in a graph.

If the dupflag is 2, then each edge increments the degree of both of its vertices. If the dupflag is 1, then each edge only increments the degree of the first of its vertices. The former is usually more appropriate for undirected graphs; the latter for directed graphs, in which case the out-degree of each vertex is being calculated.

See the [named command](#) doc page for various ways in which the `-i` inputs and `-o` outputs for a named command can be specified.

In1 stores a set of edges. No assumption is made about duplicates or self edges, i.e. (Vi,Vj) may appear multiple times, both (Vi,Vj) or (Vj,Vi) may appear, as may (Vi,Vi). The input is unchanged by this command.

Out1 will store the degree count of each vertex.

Related commands:

[neighbor](#), [degree_stats](#)

echo command

Syntax:

```
echo style
```

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

Examples:

```
echo both
echo log
```

Description:

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

The [command-line switch](#) `-echo` can be used in place of this command.

The default is `echo log`, i.e. commands are echoed to the log file.

Related commands: none

edge_upper command

Syntax:

```
edge_upper -i in1 -o out1.file out1.mr
```

- in1 = graph edges: Key = $V_i V_j$, Value = NULL
- out1 = graph edges: Key = $V_i V_j$, Value = NULL

Examples:

```
edge_upper -i tmp.matrix -o mre
```

Description:

This is a named command which eliminates duplicate and self-edges from a graph. A duplicate edge is when both (V_i, V_j) or (V_j, V_i) appear in the edge list. A self-edge is when (V_i, V_i) appears.

See the [named command](#) doc page for various ways in which the `-i` inputs and `-o` outputs for a named command can be specified.

In1 stores a set of edges, which may have duplicates or self edges. The input is unchanged by this command.

Out1 will store the edges of a new graph which has the duplicates and self edges removed. In the new graph, $V_i < V_j$ for every edge, so it also represents the non-zeroes of an upper-triangular matrix.

Related commands:

[rmat](#)

if command

Syntax:

```
if boolean then t1 t2 ... elif boolean f1 f2 ... elif boolean f1 f2 ... else e1 e2 ...
```

- boolean = a Boolean expression evaluated as TRUE or FALSE (see below)
- then = required word
- t1,t2,...,tN = one or more OINK commands to execute if condition is met, each enclosed in quotes
- elif = optional word, can appear multiple times
- f1,f2,...,fN = one or more OINK commands to execute if elif condition is met, each enclosed in quotes (optional arguments)
- else = optional argument
- e1,e2,...,eN = one or more OINK commands to execute if no condition is met, each enclosed in quotes (optional arguments)

Examples:

```
if "${steps} > 1000" then exit
if "$x <= $y" then "print X is smaller = $x" else "print Y is smaller = $y"
if "(${flag} == 0) || ($n < 1000)" then & "graph reduce myfunc" &elif ${flag} == 1 & "graph reduce
if "${niter} > ${niter_previous}" then "jump file1" else "jump file2"
```

Description:

This command provides an in-then-else capability within an input script. A Boolean expression is evaluated and the result is TRUE or FALSE. Note that as in the examples above, the expression can contain variables, as defined by the [variable](#) command, which will be evaluated as part of the expression. Thus a user-defined formula that reflects the current state of the simulation can be used to issue one or more new commands.

If the result of the Boolean expression is TRUE, then one or more commands (t1, t2, ..., tN) are executed. If it is FALSE, then Boolean expressions associated with successive elif keywords are evaluated until one is found to be true, in which case its commands (f1, f2, ..., fN) are executed. If no Boolean expression is TRUE, then the commands associated with the else keyword, namely (e1, e2, ..., eN), are executed. The elif and else keywords and their associated commands are optional. If they aren't specified and the initial Boolean expression is FALSE, then no commands are executed.

The syntax for Boolean expressions is described below.

Each command (t1, f1, e1, etc) can be any valid OINK input script command. If the command is more than one word, it must be enclosed in quotes, so it will be treated as a single argument, as in the examples above.

IMPORTANT NOTE: If a command itself requires a quoted argument (e.g. a [print](#) command), then double and single quotes can be used and nested in the usual manner, as in the examples above and below. See [this section](#) of the manual for more details on using quotes in arguments. Only one level of nesting is allowed, but that should be sufficient for most use cases.

Note that by using the line continuation character the if command can be spread across many lines, though it is still a single command:

```
if "$a < $b" then & "print 'Minimum value = $a'" & "graph ..." &else &
```

```
'print "Minimum value = $b"' & "graph ..."
```

Note that if one of the commands to execute is an invalid OINK command, such as "exit" in the first example above, then executing the command will cause OINK to halt.

Note that by jumping to a label in the same input script, the if command can be used to break out of a loop. See the [variable delete](#) command for info on how to delete the associated loop variable, so that it can be re-used later in the input script.

Here is an example of a double loop which uses the if and [jump](#) commands to break out of the inner loop when a condition is met, then continues iterating thru the outer loop.

```
label      loopa
variable   a loop 5
  label    loopb
  variable b loop 5
  print    "A,B = $a,$b"
  ...
  if       '$b > 2' then "print 'Jumping to another script'" "jump in.script break"
  next     b
  jump     in.script loopb
label      break
variable   b delete

next       a
jump       in.script loopa
```

The Boolean expressions for the if and elif keywords have a C-like syntax. Note that each expression is a single argument within the if command. Thus if you want to include spaces in the expression for clarity, you must enclose the entire expression in quotes.

An expression is built out of numbers:

```
0.2, 100, 1.0e20, -15.4, etc
```

and Boolean operators:

```
A == B, A != B, A < B, A <= B, A > B, A >= B, A & B, A || B, !A
```

Each A and B is a number or a variable reference like \$a or \${abc}, or another Boolean expression.

If a variable is used it must produce a number when evaluated and substituted for in the expression, else an error will be generated.

Expressions are evaluated left to right and have the usual C-style precedence: the unary logical NOT operator "!" has the highest precedence, the 4 relational operators "<", "<=", ">", and ">=" are next; the two remaining relational operators "==" and "!=" are next; then the logical AND operator "&" and finally the logical OR operator "||" has the lowest precedence. Parenthesis can be used to group one or more portions of an expression and/or enforce a different order of evaluation than what would occur with the default precedence.

The 6 relational operators return either a 1.0 or 0.0 depending on whether the relationship between x and y is TRUE or FALSE. The logical AND operator will return 1.0 if both its arguments are non-zero, else it returns 0.0. The logical OR operator will return 1.0 if either of its arguments is non-zero, else it returns 0.0. The logical NOT operator returns 1.0 if its argument is 0.0, else it returns 0.0.

The overall Boolean expression produces a TRUE result if the result is non-zero. If the result is zero, the expression result is FALSE.

Related commands:

[variable](#), [print](#)

include command

Syntax:

```
include file
```

- file = filename of new input script to switch to

Examples:

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

Description:

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

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

Related commands:

[variable](#), [jump](#)

input command

Syntax:

input N keyword value ...

- N = which input to set options for
- one or more keyword/value pairs may be appended

keyword = *prepend* or *substitute* or *multi* or *mmode* or *recurse* or *self* or *readfile* or *nmap* or *se*
prepend value = string to prepend to file/directory path names
substitute value = 0 or 1 = how to substitute for "%" in path name
multi value = Nmulti = multiplicity of path names to generate
mmode value = 0 or 1 or 2 = which style of map() method to invoke
recurse value = 0 or 1 = passed to map() method
self value = 0 or 1 = passed to map() method
readfile value = 0 or 1 = passed to map() method
nmap value = number of map tasks = passed to map() method
sepchar value = single character = passed to map() method
sepstr value = string = passed to map() method
delta value = Ndelta = passed to map() method

Examples:

```
input 1 multi 4
input 2 self 1 substitute 4 prepend /scratch%/hadoop-datastore/local_files
```

Description:

This command is used to control the reading of input data that a [named command](#) performs as part of its input. It does this by setting options on specific inputs to [named commands](#). The options set by this command are in effect for ONLY the next named command. After a named command is invoked, it restores all input options to their default values. Note that all of the options which can be set by this command have default values, so you don't need to set those you don't want to change.

As described on the [named command](#) doc page, a named command may specify one or more input descriptors. If the descriptor is one or more file or directory names, then each of them is converted into a list of strings which is passed to a map() method of a created MR-MPI object, along with various other arguments needed by the map() method. The purpose of the input command is to give you control over how that conversion takes place and what the values of those additional arguments are.

The *N* value corresponds to a particular input descriptor, as defined by the [named command](#). It should be an integer from 1 to *Ninput*, where *Ninput* is the number of input descriptors the command requires. The input command can be used multiple times with the same *N* to specify different parameters, e.g. one at a time.

The remaining arguments are pairs of *keywords* and *values*. One or more can be specified.

The *prepend*, *substitute*, and *multi* keywords alter the file and directory path names specified with an input descriptor in a named command.

IMPORTANT NOTE: The *prepend*, *substitute*, and *multi* keywords are applied to each file or directory name in the list of such names that the [named command](#) uses in its input descriptor.

IMPORTANT NOTE: The *prepend* and *substitute* keywords can also be set globally so that their values will be applied to all input descriptors of all [named commands](#). See the [set](#) command for details. If an input command is not used to override the global value, then the global value is used by the [named command](#).

The *prepend* keyword specifies a path name to prepend to each input file or directory name specified with the [named command](#). The prepend string is presumed to be a directory name and should be specified without the trailing "/" character, since that is added when the prepending is done.

Input file or directory names specified with the [named command](#) can contain either or both of two wildcard characters: "%" or "*". Only the first occurrence of each wildcard character is replaced.

If the *substitute* keyword is set to 0, then a "%" is replaced by the processor ID, 0 to Nprocs-1. If it is set to $N > 0$, then "%" is replaced by $(\text{proc-ID} \% N) + 1$. I.e. for 8 processors and $N = 4$, then the 8 processors replace the "%" with (1,2,3,4,1,2,3,4). This can be useful for multi-core nodes where each core has its own local disk. E.g. you wish each core to read data from one disk.

If a "*" appears, then the file or directory name is duplicated N times where N is the value set by the *multi* keyword. In each of the N duplicates, the "*" is replaced by the number 1 to N. Again, this can be useful for multi-core nodes where each core has its own local disk. E.g. you want a single core to read data from each of several local disks on the node, presumably because you have launched an MPI job so that it runs on a single core per node.

The *mmode* keyword stands for "map mode" and determines what form of the [MR-MPI library map\(\) method](#) is invoked by the [named command](#). It is up to the coding of the "named command" to determine which of these forms of data input it supports. There are 3 variants of the map() method which involve file input:

```
mmode = 0 = map(int nstr, char **strings, int self, int recurse, int readfile, void (*mymap)(), void  
mmode = 1 = map(int nmap, int nstr, char **strings, int recurse, int readfile, char sepchar, int del  
mmode = 2 = map(int nmap, int nstr, char **strings, int recurse, int  
readfile, char *sepstr, int delta, void (*mymap)(), void *ptr)
```

The "nstr" and "strings" arguments to these methods are created by OINK, using the settings described above. The remaining arguments are set by the keywords of the input command, as needed. Note that some keywords have no meaning for certain map() method variants, in which case they are simply ignored.

The meaning of the *self*, *recurse*, *readfile*, *nmap*, *sepchar*, *sepstr*, and *delta* keywords is the same as explained on the doc page for the [map\(\) method](#) of the MR-MPI library. The value for the *sepchar* keyword will be treated as a single character. The value for the *sepstr* keyword will be treated as a string.

Related commands:

[output](#), [named commands](#), [how to write named commands](#), [set](#)

Defaults:

The option defaults are *prepend* = NULL, *substitute* = 0, *multi* = 1, *mmode* = 0, *recurse* = 0, *self* = 0, *readfile* = 0, *nmap* = 0, *sepchar* = newline character, *sepstr* = newline, *delta* = 80.

jump command

Syntax:

```
jump file label
```

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

Examples:

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

Description:

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

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

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

```
lmp_g++ <in.script
```

since the SELF option invokes the C-library `rewind()` call, which may not be supported for stdin on some systems. This can be worked around by using the [-in command-line argument](#) or the [-var command-line argument](#) to pass the script name as a variable to the input script. In the latter case, the "fname" [variable](#) could be used in place of SELF. E.g.

```
lmp_g++ -in in.script
```

```
lmp_g++ -var fname n.script <in.script
```

The 2nd argument to the jump command is optional. If specified, it is treated as a label and the new file is scanned (without executing commands) until the label is found, and commands are executed from that point forward. This can be used to loop over a portion of the input script, as in this example. These commands perform 10 runs, each of 10000 steps, and create 10 dump files named file.1, file.2, etc. The [next](#) command is used to exit the loop after 10 iterations. When the "a" variable has been incremented for the tenth time, it will cause the next jump command to be skipped.

```
variable a loop 10
label loop
dump 1 all atom 100 file.$a
run 10000
undump 1
next a
jump in.lj loop
```


If the *jump file* argument is a variable, the jump command can be used to cause different processor partitions to run different input scripts. In this example, LAMMPS is run on 40 processors, with 4 partitions of 10 procs each. An in.file containing the example variable and jump command will cause each partition to run a different simulation.

```
mpirun -np 40 lmp_ibm -partition 4x10 -in in.file
```

```
variable f world script.1 script.2 script.3 script.4
jump $f
```

Here is an example of a double loop which uses the [if](#) and jump commands to break out of the inner loop when a condition is met, then continues iterating thru the outer loop.

```
label      loopa
variable   a loop 5
  label    loopb
  variable b loop 5
  print    "A,B = $a,$b"
  run      10000
  if       $b > 2 then "jump in.script break"
  next     b
  jump     in.script loopb
label      break
variable   b delete

next       a
jump      in.script loopa
```

IMPORTANT NOTE: If you jump to a file and it does not contain the specified label, OINK will come to the end of the file and exit.

Related commands:

[variable](#), [include](#), [label](#), [next](#)

label command

Syntax:

```
label ID
```

- ID = string used as label name

Examples:

```
label xyz  
label loop
```

Description:

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

Related commands: none

log command

Syntax:

```
log file
```

- file = name of new logfile

Examples:

```
log log.graph
```

Description:

This command closes the current OINK log file, opens a new file with the specified name, and begins logging information to it. If the specified file name is *none*, then no new log file is opened.

If multiple processor partitions are being used, the file name should be a variable, so that different processors do not attempt to write to the same log file.

The file "log.oink" is the default log file for a OINK run. The name of the initial log file can also be set by the command-line switch `-log`. See [this section](#) for details.

The default OINK log file is named log.oink

Related commands: none

luby_find command

Syntax:

```
luby_find seed -i in1 -o out1.file out1.mr
```

- seed = random number seed (positive integer)
- in1 = graph edges: Key = $V_i V_j$, Value = NULL
- out1 = minimal independent set: Key = V_i , Value = NULL

Examples:

```
luby_find 982938 -i mre -o mis.list mis
```

Description:

This is a named command which computes a minimal independent set (MIS) of vertices for an undirected graph. A MIS contains vertices which do not share an edge and to which no additional vertex can be added. For a given graph there are typically many possible MIS's; the MIS that is computed is a function of the specified random number seed. The MIS is found by [Luby's algorithm](#), which is an iterative method. The MapReduce version of Luby's algorithm implemented by this command is discussed in the paper of [\(Plimpton\)](#).

See the [named command](#) doc page for various ways in which the `-i` inputs and `-o` outputs for a named command can be specified.

In1 stores a set of edges, assumed to have no duplicates or self-edges. This means that either (V_i, V_j) or (V_j, V_i) appears, but not both. Also (V_i, V_i) does not appear. The input is unchanged by this command.

Out1 will store the list of vertices in the MIS.

Related commands: none

(Luby) Luby, "A Simple Parallel Algorithm for the Maximal Independent Set Problem", SIAM J Computing, 15, 1036–1055 (1986).

(Plimpton) Plimpton and Devine, "MapReduce in MPI for Large-Scale Graph Algorithms", to appear in Parallel Computing (2011).

mr command

Syntax:

```
mr MR-ID verbosity timer memsize outofcore
```

- MR-ID = ID of new MR-MPI object to create
- verbosity = verbosity setting (optional)
- timer = timer setting (optional)
- memsize = memsize setting (optional)
- outofcore = outofcore setting (optional)

Examples:

```
mr edge
mr edge 2
mr edge 1 1 16 0
```

Description:

Create a new MR-MPI object, which can be referenced by name elsewhere in your input script. In OINK, a MR-MPI object is simply a thin wrapper on a MapReduce object created via the [MR-MPI library](#). The MR-MPI object has an ID which can be used elsewhere in the input script. For example it can be used as input to a [named command](#) or in a [MR-MPI library command](#).

The ID of an MR-MPI object can only contain alphanumeric characters and underscores.

When the underlying MapReduce object is created, it will have default settings as described [here](#). Several of these settings can be overridden by the 4 options listed above. If none of them are specified, then the default settings are used. To reset one of the settings, you must specify all the settings that precede it. E.g. if just two optional arguments are used, they are the verbosity and timer settings.

Related commands:

[MR-MPI library commands](#), [named commands](#)

library commands

Syntax:

MR-ID keyword args ...

- MR-ID = ID of previously created MR object
- keyword = MR-MPI library function to invoke
- args = arguments to library function

Examples:

```
mr edge
edge map/task 100 mymap
edge map/task 100 mymap 1
edge collate NULL
edge reduce myreduce
edge kv_stats 1
edge set timer 1
```

Description:

Invoke a MR-MPI library function directly on a previously created MR-MPI objects. In OINK, an MR-MPI object is a thin wrapper on a MapReduce object created via the [MR-MPI library](#). They can be created by the [mr](#) command or can be output by a [named command](#). Such an MR-MPI object has an ID which is the command name used in the input script to trigger the library calls, e.g. "edge" in the examples above.

The keyword is the library function to invoke on the underlying MapReduce object wrapped by the MR-MPI object. These have a one-to-one correspondence with the methods available in the [MR-MPI library](#). Here is the list of keywords and their arguments. The arguments used in the OINK input script correspond to the arguments used by each library method. Arguments in parentheses are optional. More details are discussed below.

Keywords	Arguments
delete	none
copy	MR2-ID
add	MR2-ID
aggregate	NULL or hash-function
broadcast	root
clone	none
close	none
collapse	type key
collate	NULL of hash-function
compress	reduce-function
convert	none
gather	nprocs
map/task	nmap map-function (addflag)
map/char	nmap strings recurse readfile sepchar delta map-function (addflag)

map/string	nmap strings recurse readfile sepstr delta map-function (addflag)
map/mr	MR2-ID map-function (addflag)
open	none
print	(file) (fflag) proc nstride kflag vflag
reduce	reduce-function
scan/kv	scan-function
scan/kmv	scan-function
scrunch	nprocs type key
sort_keys	flag or compare-function
sort_values	flag or compare-function
sort_multivalues	flag or compare-function
kv_stats	level
kmv_stats	level
cummulative_stats	level reset
set	name value

The MR2-ID used as an argument to the "copy", "add", and "map/mr" keywords should be the ID of another previously defined MR-MPI object.

IMPORTANT NOTE: The syntax for the copy keyword in an OINK script is as follows: MR-ID copy MR2-ID. This creates a new MR-MPI object MR2-ID, which is a copy of the existing MR-MPI object MR-ID. The MR2-ID object cannot already exist. This corresponds to the following C++ calling syntax for the [copy\(\) method of the MR-MPI library](#), but note that the OINK syntax is somewhat reversed:

```
MapReduce *mr2 = mr->copy();
```

The map-function, reduce-function, hash-function, compare-function, scan-function arguments to various keywords are the names of functions that will be called back to by the MR-MPI library. Within OINK, these must be names of functions defined in map_*.cpp, reduce_*.cpp, hash_*.cpp, compare_*.cpp, or scan_*.cpp files with the appropriate function prototype. When you build OINK, these files are scanned, the function prototypes extracted, and the style_map.h, style_reduce.h, style_hash.h, style_compare.h, style_scan.h files are created which enables a function name you list in your input script to be recognized by OINK. Note that as new map(), reduce(), etc functions are added to the OINK src directory, they automatically become available to your script to use in MR-MPI library commands. Thus you can use to OINK to accumulate a collection of useful map(), reduce(), etc functions. These functions can also be used with [named commands](#) as discussed [here](#).

Note that map() functions come in 4 different flavors, with different prototypes, as [detailed here](#). Which you should use depends on which map variant you invoke, i.e. map/task, map/char, map/string, or map/mr. Likewise, scan() functions come in 2 different flavors, as [detailed here](#), one for use with scan/kv and the other with scan/kmv.

The "strings" argument to the *map/char* and *map/string* keywords can take one of two forms. It can be a single filename or directory. If the latter, then the [map\(\) method in the MR-MPI library](#) reads the files in the directory. Or it can be a variable defined elsewhere in the OINK input script that contains one or more strings which are passed to the map() method as a collection of strings. In this case the "strings" argument should be specified as v_name, where name is the name of the variable. All the different styles of variables (except equal-style) store strings; see the [variable](#) command for details. Also note that there is a [command-line option](#) -var or -v which can be specified when OINK is executed to store a list of filenames in an index-style variable.

The `sepchar` and `sepstr` arguments to the `map/char` and `map/string` keywords should be a single character or a string of characters.

The `addflag` argument to the various `map` keywords is optional. It should be 1 if you wish to add key/value pairs to those already contained in a MapReduce object.

The `type` argument to the `collapse` and `scrunch` keywords should be one of the following: "int", "uint46", "double", or "str". The key that follows will be converted into that data type to use as the key argument to the MR-MPI library function.

The `print` keyword takes either 4 or 6 arguments. If 6 are used, the first two are a file name and file flag, the same as is available with the [print\(\) method in the MR-MPI library](#).

The `flag` argument to the various `sort` keywords is an integer (e.g. 1 or -1) that can be used in place of a compare-function. This is the same integer that the [sort methods in the MR-MPI library](#) takes as a valid argument.

The `set` keyword takes a "name" and "value" argument. These can be any of the options that are valid to set for a MapReduce object in the MR-MPI library, as [discussed here](#). E.g. the command "edge verbosity 1" will set the verbosity level to 1 in the MapReduce object wrapped by the MR-MPI object named "edge".

IMPORTANT NOTE: There is currently no way in OINK to pass a data pointer to the various MR-MPI library functions that accept it, e.g. to `map()` or `reduce()`. When using the library from a programming language, such as C++ or C, this is powerful option for passing extra information to the user callback `map()` or `reduce()` function. We are still thinking about the best way to do this, at least in some limited fashion, from an OINK input script.

When any MR-MPI library command is executed, its elapsed execution time is stored internally by OINK. This value can be accessed by the keyword "time" in an [equal-style variable](#) and printed out in the following manner:

```
variable t equal time
edge map/task 100 mymap
print "Time for map/task = $t"
```

Related commands:

[named commands](#), [mr](#), [MR-MPI library documentation](#), [map\(\)](#), [reduce\(\)](#), etc functions

neigh_tri command

Syntax:

```
neigh_tri dirname -i input1 input2 -o out1.file out1.mr
```

- `dirname` = directory name to create set of output files in, one per vertex
- `input1` = graph neighbors: Key = V_i , Value = $V_j V_k \dots$
- `input2` = triangles: Key = $V_i V_j V_k$, Value = NULL
- `out1` = neighbors + triangle edges of each vertex: Key = V_i , MultiValue = $V_j V_k \dots (V_j V_k) (V_m V_n) \dots$

Examples:

```
neigh_tri myneigh -i mrn mrtri -o NULL mrnplus
```

Description:

This is a named command which calculates a list of edges associated with each vertex in an undirected graph, which include all edges the vertex is in (its first neighbors) and also edges between pairs of its first neighbors (triangle edges). This set of data is written to a file per vertex as a list of edges.

See the [named command](#) doc page for various ways in which the `-i` inputs and `-o` outputs for a named command can be specified.

Input1 stores a set of neighbors of each vertex. See the [neighbor](#) command, which can compute this data. Input2 stores a set of triangles. See the [tri_find](#) command, which can compute this data. The two inputs are unchanged by this command.

These 2 data sets are merged to identify the edges that exist between pairs of neighbors of each vertex. This information is written to a file per vertex. The name of each file is `dirname/Vi` where *dirname* is the specified argument (a directory name), and V_i is the vertex ID. Each file will contain a list of edges, one per line, written as $V_m V_n$. For some of the V_m will equal V_i , which means they are edges containing V_i , i.e. they are the first neighbors of V_i . Other edges will have $V_m \neq V_i$. These are edges between pairs of first neighbors.

Out1 will store the neighbor and triangle edge information as key/multivalue (KMV) pairs, not as key/value (KV) pairs (the usual form of output).

Out1.file must be specified as NULL with the `"-o"` argument so that the output is only allowed as an MR-MPI object, not as a file. This is because the file would contain data for all the vertices together. The equivalent info is already output as one file per vertex, as described above.

NOTE: alter the `neigh_tri.cpp` code so that it uses the input `dirname` with `expandpath()` to apply the global prepend and substitute settings ??

Related commands:

[neighbor](#), [tri_find](#)

neighbor command

Syntax:

```
neighbor -i in1 -o out1.file out1.mr
```

- in1 = graph edges: Key = $V_i V_j$, Value = NULL
- out1 = neighbors of each vertex: Key = V_i , Value = $V_j V_k \dots$

Examples:

```
degree -i mrw -o degree.list NULL
```

Description:

This is a named command which calculates the list of neighbors of each vertex in an undirected graph. A list of all the vertices each vertex shares an edge with is calculated. These are the first neighbors of each vertex.

See the [named command](#) doc page for various ways in which the `-i` inputs and `-o` outputs for a named command can be specified.

In1 stores a set of edges, assumed to have no duplicates or self-edges. This means that either (V_i, V_j) or (V_j, V_i) appears, but not both. Also (V_i, V_i) does not appear. The input is unchanged by this command.

Out1 will store the list of neighbor vertices for each vertex. The list is a single value in a key/value pair which is a vector of vertex IDs, one after the other.

Related commands:

[degree](#), [neigh_tri](#)

next command

Syntax:

```
next variables
```

- variables = one or more variable names

Examples:

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

Description:

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

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

If multiple variables are used as arguments to the *next* command, then all must be of the same variable style: *index*, *loop*, *universe*, or *uloop*. An exception is that *universe*- and *uloop*-style variables can be mixed in the same *next* command.

All the variables specified with the next command are incremented by one value from their respective list or values. *String*- or *equal*- or *world*-style variables cannot be used with the the next command, since they only store a single value.

When any of the variables in the next command has no more values, a flag is set that causes the input script to skip the next [jump](#) command encountered. This enables a loop containing a next command to exit. As explained in the [variable](#) command, the variable that has exhausted its values is also deleted. This allows it to be used and re-defined later in the input script.

When the next command is used with *index*- or *loop*-style variables, the next value is assigned to the variable for all processors. When the next command is used with *universe*- or *uloop*-style variables, the next value is assigned to whichever processor partition executes the command first. All processors in the partition are assigned the same value. Running OINK on multiple partitions of processors via the "-partition" command-line switch is described in [this section](#) of the manual. *Universe*- and *uloop*-style variables are incremented using the files "tmp.oink.variable" and "tmp.oink.variable.lock" which you will see in your directory during such a OINK run.

Here is an example of running a series of simulations using the next command with an *index*-style variable. If this input script is named in.graph, 8 simulations would be run using data files from directories run1 thru run8.

```
variable d index run1 run2 run3 run4 run5 run6 run7 run8
shell cd $d
graph -i data.graph
shell cd ..
clear
next d
```

```
jump in.graph
```

If the variable "d" were of style *universe*, and the same in.graph input script were run on 3 partitions of processors, then the first 3 simulations would begin, one on each set of processors. Whichever partition finished first, it would assign variable "d" the 4th value and run another simulation, and so forth until all 8 simulations were finished.

Jump and next commands can also be nested to enable multi-level loops. For example, this script will run 15 simulations in a double loop.

```
variable i loop 3
  variable j loop 5
  clear
  ...
  print Running simulation $i.$j
  graph -i data.polymer.$i$j
  next j
  jump in.script
next i
jump in.script
```

Here is an example of a double loop which uses the [if](#) and [jump](#) commands to break out of the inner loop when a condition is met, then continues iterating thru the outer loop.

```
label      loopa
variable   a loop 5
  label    loopb
  variable b loop 5
  print    "A,B = $a,$b"
  run      10000
  if       $b > 2 then "jump in.script break"
  next     b
  jump     in.script loopb
label      break
variable   b delete

next       a
jump      in.script loopa
```

Related commands:

[jump](#), [include](#), [shell](#), [variable](#),

output command

Syntax:

```
output N keyword value ...
```

- N = which output to set options for
- one or more keyword/value pairs may be appended keyword = *prepend* or *substitute* prepend value = string to prepend to file path names *substitute* value = 0 or 1 = how to substitute for "%" in path name

Examples:

```
output 1 substitute 4
output 2 substitute 4 prepend /scratch%/hadoop-datastore/local_files
```

Description:

This command is used to control the writing of data that a [named command](#) performs as part of its output. It does this by setting options on specific outputs to [named commands](#). The options set by this command are in effect for ONLY the next named command. After a named command is invoked, it restores all output options to their default values. Note that all of the options which can be set by this command have default values, so you don't need to set those you don't want to change.

As described on the [named command](#) doc page, a named command may specify one or more output descriptors. Each descriptor is a pair of arguments, the first of which is an output filename (if it is not specified as NULL). OINK converts the specified argument into an actual filename which is opened by each processor. The purpose of the output command is to give you control over how that conversion takes place.

The *N* value corresponds to a particular output descriptor, as defined by the [named command](#). It should be an integer from 1 to Noutput, where Noutput is the number of output descriptors the command requires. The output command can be used multiple times with the same *N* to specify different parameters, e.g. one at a time.

The remaining arguments are pairs of *keywords* and *values*. One or more can be specified.

The *prepend* and *substitute* keywords alter the file and directory path names specified with the filename of an output descriptor in a named command.

IMPORTANT NOTE: The *prepend* and *substitute* keywords can also be set globally so that their values will be applied to all output descriptors of all [named commands](#). See the [set](#) command for details. If an output command is not used to override the global value, then the global value is used by the [named command](#).

The *prepend* keyword specifies a path name to prepend to the output file specified with the [named command](#). The prepend string is presumed to be a directory name and should be specified without the trailing "/" character, since that is added when the prepending is done.

Output file or directory names specified with the [named command](#) can contain either or both of two wildcard characters: "%" or "*". Only the first occurrence of each wildcard character is replaced.

If the *substitute* keyword is set to 0, then a "%" is replaced by the processor ID, 0 to Nprocs-1. If it is set to $N > 0$, then "%" is replaced by $(\text{proc-ID} \% N) + 1$. I.e. for 8 processors and $N = 4$, then the 8 processors replace the "%"

with (1,2,3,4,1,2,3,4). This can be useful for multi-core nodes where each core has its own local disk. E.g. you wish each core to write data to one disk.

IMPORTANT NOTE: The processor ID is also added as a suffix to the specified output file by each processor, so that one output file per processor is generated. This is in addition to any replacement of a "%" wildcard character.

If a "*" appears, then it is replaced with a 1. Unlike for [input files](#), this is not a particularly useful wildcard for output files.

Related commands:

[input](#), [named commands](#), [how to write named commands](#), [set](#)

Defaults:

The option defaults are prepend = NULL, substitute = 0, multi = 1, mmode = 0, recurse = 0, self = 0, readfile = 0, nmap = 0, sepchar = newline character, sepstr = newline, delta = 80.

pagerank command

Syntax:

```
pagerank tolerance Nmax alpha -i in1 -o out1.file out1.mr
```

- tolerance = stopping tolerance on PageRank iterations
- Nmax = max # of matrix–vector iterations to allow
- alpha = ???
- in1 = graph edges: Key = $V_i V_j$, Value = weight
- out1 = distance to each vertex: Key = V_i , Value = distance

Examples:

```
pagerank 0.01 50 0.3 -i mre -o prank.txt NULL
```

Description:

This is a named command which computes the PageRank numeric score for vertices in a directed graph. The PageRank of a vertex is a measure of how "important" it is in the graph, based on what vertices point to it, and the PageRank of those vertices. If the graph represents WWW pages linked to each other, then this is part of how Google ranks the relative importance of pages it shows you as the result of a search. See [this Google site](#) for more information.

The PageRank calculation is performed via an iterative matrix–vector multiply operation, where the graph can be thought of as a sparse matrix. The MapReduce version of this PageRank implementation is described in the paper of ([Plimpton](#)).

Describe what alpha and tolerance and maxiter do.

See the [named command](#) doc page for various ways in which the `-i` inputs and `-o` outputs for a named command can be specified.

In1 stores a set of edges with weights, assumed to have no duplicates, meaning that (V_i, V_j) only appears once. Each edge is directed in the sense that V_i points to V_j . The weight is effectively the non–zero value of the (V_i, V_j) element of the matrix. Typically it should be $1/D$ where D is the out–degree of V_i , but this is not required. The input is unchanged by this command.

Out1 will store the list of vertices and the numeric rank of each vertex.

Related commands: none

(Google) Some citation to a Google paper or WWW site, explaining PageRank, (2005).

(Plimpton) Plimpton and Devine, "MapReduce in MPI for Large–Scale Graph Algorithms", to appear in Parallel Computing (2011).

print command

Syntax:

```
print str
```

- str = text string to print, which may contain variables

Examples:

```
print "Done with first stage" print "Elapsed time = $t secs on $p procs"
```

Description:

Print a text string to the screen and logfile. One line of output is generated. If the string has white space in it (spaces, tabs, etc), then you must enclose it in quotes so that it is treated as a single argument. If variables are included in the string, they will be evaluated and their current values printed.

See the [variable](#) command for a description of various kinds of variables, any of which can be used with the print command. Note that there are keywords for the number of processors and elapsed time for a command or MR-MPI library call which can be accessed with variables, e.g.

```
variable t equal time
variable p equal nprocs
print "Elapsed time = $t secs on $p procs"
```

Related commands:

[variable](#)

rmat command

rmat2 command

Syntax:

```
rmat N Nz a b c d fraction seed -o out1.file out1.mr
rmat2 N Nz a b c d fraction seed -o out1.file out1.mr
```

- N = order of matrix, 2^N = number of rows in matrix
- Nz = average # of non-zeroes per row, $Nz * 2^N$ = total # of non-zeroes
- a, b, c, d = R-MAT parameters which sum to 1.0
- fraction = R-MAT twiddle factor
- seed = random number seed (positive integer)
- out1 = graph edges: Key = $V_i V_j$, Value = NULL

Examples:

```
rmat 20 8 0.45 0.25 0.25 0.05 0.0 284958 -o NULL mre
rmat2 20 8 0.45 0.25 0.25 0.05 0.0 284958 -o tmp.rmat NULL
```

Description:

These are named commands which generate a sparse random matrix via the procedure defined for R-MAT matrices, as discussed in the paper by ([Chakrabarti](#)). Such matrices are often used to represent graphs where the vertices are numbered 1 to N_{rows} , and the non-zero matrix entries represent edges. The number of rows and non-zero entries are determined by the specified N and Nz arguments.

Depending on the choice of the R-MAT parameters the degree distribution of the resulting graph can be roughly uniform or highly skewed, which is useful in modeling different kind of graphs, e.g. Internet connectivity. The a, b, c, d parameters must sum to 1.0 and represent weighting for the 4 different quadrants of the matrix. As non-zero entries are generated, they are assigned to each quadrant in a recursive manner using the a, b, c, d weightings and a random number generator. A fraction value of 0.0 means the a, b, c, d weightings are used as-is. A fraction value > 0.0 but < 1.0 means the weightings are randomly twiddled at each iteration of the recursion.

The MapReduce algorithms used for performing the R-MAT generation are described in the paper by ([Plimpton](#)). The `rmat` command implements the first of the two algorithms discussed in the paper. The `rmat2` command implements the second of the two algorithms.

See the [named command](#) doc page for various ways in which the `-i` inputs and `-o` outputs for a named command can be specified.

These commands take no inputs.

Out1 will store the list of edges of the R-MAT graph, or equivalently, the I, J indices of non-zeroes in the matrix. There will be exactly $Nz * 2^N$ entries in out1. This may include some duplicate or self-edges. A duplicate edge is when both (V_i, V_j) or (V_j, V_i) appear. A self-edge is when (V_i, V_i) appears. If desired, these can be removed by further processing; see the [edge_upper](#) command.

Related commands:

(Chakrabarti) Chakrabarti, Zhan, Faloutsos, "R-MAT: A recursive model for graph mining", in SIAM Data Mining (2004).

(Plimpton) Plimpton and Devine, "MapReduce in MPI for Large-Scale Graph Algorithms", to appear in Parallel Computing (2011).

set command

Syntax:

set keyword value ...

- one or more keyword/value pairs may be appended
- keyword = *verbosity* or *timer* or *memsize* or *outofcore* or *scratch* or *prepend* or *substitute*

verbosity value = setting for created MapReduce objects
timer value = setting for created MapReduce objects
memsize value = setting for created MapReduce objects
outofcore value = setting for created MapReduce objects
minpage value = setting for created MapReduce objects
maxpage value = setting for created MapReduce objects
freepage value = setting for created MapReduce objects
zeropage value = setting for created MapReduce objects
scratch value = setting for created MapReduce objects
prepend value = string to prepend to file/directory path names
substitute value = 0 or 1 = how to substitute for "%" in path name

Examples:

set verbosity 2 set verbosity 1 timer 1 memsize 16 set scratch /tmp/mr set prepend /scratch%/data substitute 1

Description:

This command sets global settings which are used in the creation of MR-MPI objects and the underlying MapReduce objects they wrap. Note that many of these setting names have the same meaning they do in the MR-MPI library themselves, as discussed on [this doc page](#).

The settings for the *verbosity*, *timer*, *memsize*, *outofcore*, *minpage*, *maxpage*, *freepage*, and *zeropage* keywords are used by the [mr](#) command creates a MapReduce object to set its attributes. Note that the [mr](#) command itself can override several of these global settings.

[Named commands](#) can also create MapReduce objects, either when inputting and outputting data, or when the `run()` method in the named command class invokes certain methods, like `create_mr()` or `copy_mr()`. Each time a new MapReduce object is created, these same global settings are applied to it. See [this doc page](#) for more discussion of the input/output options and these methods.

The *scratch* keyword is a directory pathname which all MapReduce objects will use for writing temporary files when they operate in out-of-core mode. Every MapReduce object created by OINK will have its scratch directory set to this value, via the `fpath()` call described on [this doc page](#)

The *prepend* and *substitute* keywords affect how file and directory names are interpreted by OINK. File and directory names are used as input and output options to [named commands](#) via the "-i" and "-o" arguments in an input script. Before these path names are passed to the MR-MPI library, e.g. as part of a `map()` method, they can have a directory name prepended to them, and "%" characters in the path name substituted for with a processor ID. This is to enable flexible options for input/output of different files by different processors.

If the *prepend* keyword is set, its value should be a directory name (without the trailing "/"). This will be prepended to every input and output pathname used by OINK, including the scratch directory noted above. This

global setting can be overridden for a single input or output of the next-executed [named command](#) by setting the same *prepend* keyword in the [input](#) or [output](#) command.

Input file or directory names can contain the wildcard character "%". Only the first occurrence of the wildcard character is replaced.

If the *substitute* keyword is set to 0, then a "%" is replaced by the processor ID, 0 to Nprocs-1. If it is set to $N > 0$, then "%" is replaced by $(\text{proc-ID} \% N) + 1$. I.e. for 8 processors and $N = 4$, then the 8 processors replace the "%" with (1,2,3,4,1,2,3,4). This can be useful for multi-core nodes where each core has its own local disk. E.g. you wish each core to read data from one disk.

As with the *prepend* keyword, this substitution rule will be applied to every input and output pathname used by OINK, including the scratch directory noted above. This global setting can be overridden for a single input or output of the next-executed [named command](#) by setting the same *substitute* keyword in the [input](#) or [output](#) command.

Related commands:

[input](#), [output](#), [named commands](#), [MR-MPI library commands](#), [Section_commands](#)

Defaults:

The setting defaults are the same as for the MR-MPI library itself, namely verbosity = 0, timer = 0, memsize = 64, outofcore = 0, minpage = 0, maxpage = 0, freepage = 1, zeropage = 0, scratch = ".". There are additional default values: prepend = NULL, and substitute = 0.

shell command

Syntax:

shell style args

- style = *cd* or *mkdir* or *mv* or *rm* or *rmdir*

```
cd arg = dir
    dir = directory to change to
mkdir args = dir1 dir2 ...
    dir1,dir2 = one or more directories to create
mv args = old new
    old = old filename
    new = new filename
rm args = file1 file2 ...
    file1,file2 = one or more filenames to delete
rmdir args = dir1 dir2 ...
    dir1,dir2 = one or more directories to delete
```

Examples:

```
shell cd sub1
shell cd ..
shell mkdir tmp1 tmp2 tmp3
shell rmdir tmp1
shell mv log.lammps hold/log.1
shell rm TMP/file1 TMP/file2
```

Description:

Execute a shell command. Only a few simple file-based shell commands are supported, in Unix-style syntax. With the exception of *cd*, all commands are executed by only a single processor, so that files/directories are not being manipulated by multiple processors.

The *cd* style executes the Unix "cd" command to change the working directory. All subsequent OINK commands that read/write files will use the new directory. All processors execute this command.

The *mkdir* style executes the Unix "mkdir" command to create one or more directories.

The *mv* style executes the Unix "mv" command to rename a file and/or move it to a new directory.

The *rm* style executes the Unix "rm" command to remove one or more files.

The *rmdir* style executes the Unix "rmdir" command to remove one or more directories. A directory must be empty to be successfully removed.

IMPORTANT NOTE: OINK does not detect errors or print warnings when any of these Unix commands execute. E.g. if the specified directory does not exist, executing the *cd* command will silently not do anything.

Related commands: none

sssp command

sssp2 command

Syntax:

```
sssp N seed -i in1 -o out1.file out1.mr
```

```
sssp2 N seed -i in1 -o out1.file out1.mr
```

- N = # of random starting vertices to choose
- seed = random number seed (positive integer)
- in1 = graph edges: Key = $V_i V_j$, Value = weight
- out1 = distance to each vertex: Key = V_i , Value = distance

Examples:

```
sssp 10 12345 -i mre -o sssp.dist mrdist  
sssp2 10 12345 -i mre -o sssp.dist mrdist
```

Description:

These are named commands which compute the shortest path to each vertex from a source vertex in an undirected graph. This is called a single-source shortest-path (SSSP) calculation. The source vertex is selected randomly. Each edge in the graph has a weight. The shortest-path distance to any other vertex is the minimum summed weight of a list of consecutive edges that connect the two vertices.

This calculation involves a breadth-first search on the graph. The MapReduce algorithms used for performing the SSSP calculation are described in the paper of ([Plimpton](#)). The sssp command implements the first of the two algorithms discussed in the paper. The sssp2 command implements the second of the two algorithms.

See the [named command](#) doc page for various ways in which the `-i` inputs and `-o` outputs for a named command can be specified.

In1 stores a set of edges with floating point weights, assumed to have no duplicates or self-edges. This means that either (V_i, V_j) or (V_j, V_i) appears, but not both. Also (V_i, V_i) does not appear. The input is unchanged by this command.

Out1 will store the list of vertices and the distance to each vertex. If the specified $N > 1$, then this will be the SSSP result for only the last source vertex randomly selected.

Related commands: none

(Plimpton) Plimpton and Devine, "MapReduce in MPI for Large-Scale Graph Algorithms", to appear in Parallel Computing (2011).

tri_find command

Syntax:

```
tri_find -i in1 -o out1.file out1.mr
```

- in1 = graph edges: Key = $V_i V_j$, Value = NULL
- out1 = triangles: Key = $V_i V_j V_k$, Value = NULL

Examples:

```
tri_find -i mre -o tri.list mtri
```

Description:

This is a named command which enumerates all the triangles in an undirected graph. A triangle is a set of 3 vertices I,J,K for which the edges IJ, JK, IK all exist. The triangles are found via the MapReduce algorithm of [\(Cohen\)](#) discussed in his paper and in the paper of [\(Plimpton\)](#). Note that even small graphs can have large numbers of triangles if there are very high-degree vertices.

See the [named command](#) doc page for various ways in which the `-i` inputs and `-o` outputs for a named command can be specified.

In1 stores a set of edges, assumed to have no duplicates or self-edges. This means that either (V_i, V_j) or (V_j, V_i) appears, but not both. Also (V_i, V_i) does not appear. The input is unchanged by this command.

Out1 will store the list of triangles.

Related commands: none

(Cohen) Cohen, "Graph Twiddling in a MapReduce World", Computing in Science and Engineering, 11, 29–41 (2009).

(Plimpton) Plimpton and Devine, "MapReduce in MPI for Large-Scale Graph Algorithms", to appear in Parallel Computing (2011).

variable command

Syntax:

variable name style args ...

- name = name of variable to define
- style = *delete* or *index* or *loop* or *world* or *universe* or *uloop* or *string* or *equal* or *atom*

```

delete = no args
index args = one or more strings
loop args = N
    N = integer size of loop, loop from 1 to N inclusive
loop args = N pad
    N = integer size of loop, loop from 1 to N inclusive
    pad = all values will be same length, e.g. 001, 002, ..., 100
loop args = N1 N2
    N1,N2 = loop from N1 to N2 inclusive
loop args = N1 N2 pad
    N1,N2 = loop from N1 to N2 inclusive
    pad = all values will be same length, e.g. 050, 051, ..., 100
world args = one string for each partition of processors
universe args = one or more strings
uloop args = N
    N = integer size of loop
uloop args = N pad
    N = integer size of loop
    pad = all values will be same length, e.g. 001, 002, ..., 100
string arg = one string
equal args = one formula containing numbers, keywords, math
operations, variable references
    numbers = 0.0, 100, -5.4, 2.8e-4, etc
    constants = PI
    keywords = nprocs, time
    math operators = (), -x, x+y, x-y, x*y, x/y, x^y,
                    x==y, x!=y, xy, x>=y, xx||y, !x
    math functions = sqrt(x), exp(x), ln(x), log(x),
                    sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), atan2(y,x),
                    random(x,y,z), normal(x,y,z), ceil(x), floor(x), round(x)
                    ramp(x,y), stagger(x,y), logfreq(x,y,z), vdisplace(x,y), swiggle(x,y,z),
variable references = v_name

```

Examples:

```

variable x index run1 run2 run3 run4 run5 run6 run7 run8
variable LoopVar loop $n
variable p equal nprocs
variable b1 equal 0.5*v_flag
variable b1 equal "10 + 0.5*v_flag"
variable foo myfile
variable x universe 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
variable x uloop 15 pad
variable x delete

```

Description:

This command assigns one or more strings to a variable name for evaluation later in the input script or during a simulation.

Variables can be used in several ways in OINK. A variable can be referenced elsewhere in an input script to become part of a new input command. For variable styles that store multiple strings, the [next](#) command can be used to increment which string is assigned to the variable. Variables of style *equal* store a formula which when evaluated produces a single numeric value which can be output via the [print](#) command. Variables that store a collection of strings can be used as input to a named command, e.g. to process a collection of filenames. See the [named command](#) doc page for details.

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

IMPORTANT NOTE: When the input script line that defines a variable of style *equal* is encountered, the formula is NOT immediately evaluated and the result stored. See the discussion below about "Immediate Evaluation of Variables" if you want to do this.

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

There are two exceptions to this rule. First, variables of style *string* and *equal* ARE redefined each time the command is encountered. This allows these style of variables to be redefined multiple times in an input script. In a loop, this means the formula associated with an *equal*-style variable can change if it contains a substitution for another variable, e.g. `$x`.

Second, as described below, if a variable is iterated on to the end of its list of strings via the [next](#) command, it is removed from the list of active variables, and is thus available to be re-defined in a subsequent variable command. The *delete* style does the same thing.

[This section](#) of the manual explains how occurrences of a variable name in an input script line are replaced by the variable's string. The variable name can be referenced as `$x` if the name "x" is a single character, or as `${LoopVar}` if the name "LoopVar" is one or more characters.

As described below, for variable styles *index*, *loop*, *universe*, and *uloop*, which string is assigned to a variable can be incremented via the [next](#) command. When there are no more strings to assign, the variable is exhausted and a flag is set that causes the next [jump](#) command encountered in the input script to be skipped. This enables the construction of simple loops in the input script that are iterated over and then exited from.

As explained above, an exhausted variable can be re-used in an input script. The *delete* style also removes the variable, the same as if it were exhausted, allowing it to be redefined later in the input script or when the input script is looped over. This can be useful when breaking out of a loop via the [if](#) and [jump](#) commands before the variable would become exhausted. For example,

```
label      loop
variable   a loop 5
print      "A = $a"
if         $a > 2 then "jump in.script break"
next       a
jump       in.script loop
label      break
```

For the *index* style, one or more strings are specified. Initially, the 1st string is assigned to the variable. Each time a [next](#) command is used with the variable name, the next string is assigned. All processors assign the same string to the variable.

Index style variables with a single string value can also be set by using the command-line switch `-var`; see [this section](#) for details.

The *loop* style is identical to the *index* style except that the strings are the integers from 1 to N inclusive, if only one argument N is specified. This allows generation of a long list of runs (e.g. 1000) without having to list N strings in the input script. Initially, the string "1" is assigned to the variable. Each time a [next](#) command is used with the variable name, the next string ("2", "3", etc) is assigned. All processors assign the same string to the variable. The *loop* style can also be specified with two arguments N1 and N2. In this case the loop runs from N1 to N2 inclusive, and the string N1 is initially assigned to the variable.

For the *world* style, one or more strings are specified. There must be one string for each processor partition or "world". See [this section](#) of the manual for information on running OINK with multiple partitions via the `"-partition"` command-line switch. This variable command assigns one string to each world. All processors in the world are assigned the same string. The next command cannot be used with *equal* style variables, since there is only one value per world. This style of variable is useful when you wish to perform different calculations on different partitions.

For the *universe* style, one or more strings are specified. There must be at least as many strings as there are processor partitions or "worlds". See [this page](#) for information on running OINK with multiple partitions via the `"-partition"` command-line switch. This variable command initially assigns one string to each world. When a [next](#) command is encountered using this variable, the first processor partition to encounter it, is assigned the next available string. This continues until all the variable strings are consumed. Thus, this command can be used to run 50 simulations on 8 processor partitions. The simulations will be run one after the other on whatever partition becomes available, until they are all finished. *Universe* style variables are incremented using the files `"tmp.oink.variable"` and `"tmp.oink.variable.lock"` which you will see in your directory during such a OINK run.

The *uloop* style is identical to the *universe* style except that the strings are the integers from 1 to N. This allows generation of long list of runs (e.g. 1000) without having to list N strings in the input script.

All *universe*- and *uloop*-style variables defined in an input script must have the same number of values.

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

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

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

The formula for an *equal* variable can contain a variety of quantities. The syntax for each kind of quantity is simple, but multiple quantities can be nested and combined in various ways to build up formulas of arbitrary complexity.

Specifically, an formula can contain numbers, keywords, math operators, math functions, and references to other variables.

Number	0.2, 100, 1.0e20, -15.4, etc
Constant	PI
Keywords	nprocs, time
Math operators	(), -x, x+y, x-y, x*y, x/y, x^y, x==y, x!=y, xy, x>=y, xx y, !x
Math functions	sqrt(x), exp(x), ln(x), log(x), sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), atan2(y,x), random(x,y,z), normal(x,y,z), ceil(x), floor(x), round(x), ramp(x,y), stagger(x,y), logfreq(x,y,z), vdisplace(x,y), swiggle(x,y,z), cwiggle(x,y,z)
Other variables	v_name

The keywords allowed in a formula are *nprocs* and *time*. Nprocs is the number of processors being used. Time is the elapsed time of the most recently executed [named command](#) or [MR-MPI library command](#).

Math Operators

Math operators are written in the usual way, where the "x" and "y" in the examples can themselves be arbitrarily complex formulas, as in the examples above.

Operators are evaluated left to right and have the usual C-style precedence: unary minus and unary logical NOT operator "!" have the highest precedence, exponentiation "^" is next; multiplication and division are next; addition and subtraction are next; the 4 relational operators "=", "<", ">=", and ">" are next; the two remaining relational operators "==" and "!=" are next; then the logical AND operator "&" and finally the logical OR operator "||" has the lowest precedence. Parenthesis can be used to group one or more portions of a formula and/or enforce a different order of evaluation than what would occur with the default precedence.

The 6 relational operators return either a 1.0 or 0.0 depending on whether the relationship between x and y is TRUE or FALSE. For example the expression x

Math Functions

Math functions are specified as keywords followed by one or more parenthesized arguments "x", "y", "z", each of which can themselves be arbitrarily complex formulas.

Most of the math functions perform obvious operations. The ln() is the natural log; log() is the base 10 log.

The random(x,y,z) function takes 3 arguments: x = lo, y = hi, and z = seed. It generates a uniform random number between lo and hi. The normal(x,y,z) function also takes 3 arguments: x = mu, y = sigma, and z = seed. It generates a Gaussian variate centered on mu with variance sigma^2. In both cases the seed is used the first time the internal random number generator is invoked, to initialize it. For equal-style variables, every processor uses the same seed so that they each generate the same sequence of random numbers.

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

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

Variable References

Variable references access quantities calculated by other variables, which will cause those variables to be evaluated. The name in the reference should be replaced by the name of a variable defined elsewhere in the input script.

IMPORTANT NOTE: If you define variables in circular manner like this:

```
variable a equal v_b
variable b equal v_a
print $a
```

then OINK may run for a while when the print statement is invoked!

Immediate Evaluation of Variables:

There is a difference between referencing a variable with a leading \$ sign (e.g. \$x or \${abc}) versus with a leading "v_" (e.g. v_x or v_abc). The former can be used in any command, including a variable command, to force the immediate evaluation of the referenced variable and the substitution of its value into the command. The latter is a required kind of argument to some commands (e.g. the [fix ave/spatial](#) or [dump custom](#) or [thermo_style](#) commands) if you wish it to evaluate a variable periodically during a run. It can also be used in a variable formula if you wish to reference a second variable. The second variable will be evaluated whenever the first variable is evaluated.

As an example, suppose you use this command in your input script to define the variable "v" as

```
variable v equal vol
```

before a run where the simulation box size changes. You might think this will assign the initial volume to the variable "v". That is not the case. Rather it assigns a formula which evaluates the volume (using the [thermo_style](#) keyword "vol") to the variable "v". If you use the variable "v" in some other command like [fix ave/time](#) then the current volume of the box will be evaluated continuously during the run.

If you want to store the initial volume of the system, you can do it this way:

```
variable v equal vol
variable v0 equal $v
```

The second command will force "v" to be evaluated (yielding the initial volume) and assign that value to the variable "v0". Thus the command

```
thermo_style custom step v_v v_v0
```

would print out both the current and initial volume periodically during the run.

Note that it is a mistake to enclose a variable formula in double quotes if it contains variables preceeded by \$ signs. For example,

```
variable vratio equal "${vfinal}/${v0}"
```

This is because the quotes prevent variable substitution (see [this section](#) on parsing input script commands), and thus an error will occur when the formula for "vratio" is evaluated later.

Related commands:

[next](#), [jump](#), [include](#), [print](#)

vertex_extract command

Syntax:

```
vertex_extract -i in1 -o out1.file out1.mr
```

- in1 = graph edges: Key = $V_i V_j$, Value = NULL
- out1 = all vertices in graph: Key = V_i , Value = NULL

Examples:

```
vertex_degree -i mre -o vlist.txt mrv
```

Description:

This is a named command which extracts a list of all the vertices in a graph from the edge list.

See the [named command](#) doc page for various ways in which the `-i` inputs and `-o` outputs for a named command can be specified.

In1 stores a set of edges. No assumption is made about duplicates or self edges, i.e. (V_i, V_j) may appear multiple times, both (V_i, V_j) or (V_j, V_i) may appear, as may (V_i, V_i) . The input is unchanged by this command.

Out1 will store the vertices in the graph.

Related commands: none

wordfreq command

Syntax:

```
wordfreq Ntop -i input1 -o out1.file out1.mr
```

- Ntop = print Ntop of the most frequently occurring words to screen
- input1 = words: Key = word, Value = NULL
- out1 = frequency count of each word: Key = word, Value = count

Examples:

```
wordfreq 10 -i v_files -o full.list NULL wordfreq 10 -i v_files -o NULL NULL
```

Description:

This is a named command which calculates the frequency of word occurrence in an input data set, which is typically a set of files.

See the [named command](#) doc page for various ways in which the `-i` inputs and `-o` outputs for a named command can be specified.

Input1 stores a set of words. The input is changed by the command as it is processed further.

If the input is one or more files then the files are read and each "word" is defined as separated by whitespace. Note that you can pass a list of files as the input argument after the `-i` argument by using a variable, which in turn can be initialized with a command-line argument to OINK. E.g. this line would work with the first example above:

```
oink_linux -var files *.cpp <in.script
```

See [this section](#) of the manual and the [variable](#) doc page for more details.

A count of the number of times each word occurs is calculated. The highest frequency *Ntop* occurrences are printed to the screen with their count. If *Ntop* is 0, nothing is printed.

Out1 will store the frequency count of all unique words.

Related commands: none