CHAPTER

FOUR

RUN LAMMPS

These pages explain how to run LAMMPS once you have *installed an executable* or *downloaded the source code* and *built an executable*. The *Commands* doc page describes how input scripts are structured and the commands they can contain.

4.1 Basics of running LAMMPS

LAMMPS is run from the command line, reading commands from a file via the -in command line flag, or from standard input. Using the -in in.file variant is recommended (see note below). The name of the LAMMPS executable is either lmp or lmp_<machine> with *<machine>* being the machine string used when compiling LAMMPS. This is required when compiling LAMMPS with the traditional build system (e.g. with make mpi), but optional when using CMake to configure and build LAMMPS:

```
lmp_serial -in in.file
lmp_serial < in.file
lmp -in in.file
lmp < in.file
/path/to/lammps/src/lmp_serial -i in.file
mpirun -np 4 lmp_mpi -in in.file
mpiexec -np 4 lmp -in in.file
mpirun -np 8 /path/to/lammps/src/lmp_mpi -in in.file
mpiexec -n 6 /usr/local/bin/lmp -in in.file
```

You normally run the LAMMPS command in the directory where your input script is located. That is also where output files are produced by default, unless you provide specific other paths in your input script or on the command line. As in some of the examples above, the LAMMPS executable itself can be placed elsewhere.

Note

The redirection operator "<" will not always work when running in parallel with mpirun or mpiexec; for those systems the -in form is required.

As LAMMPS runs it prints info to the screen and a logfile named log.lammps. More info about output is given on the *screen and logfile output* page.

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

LAMMPS can run the same problem on any number of processors, including a single processor. In theory you should get identical answers on any number of processors and on any machine. In practice, numerical round-off due to using floating-point math can cause slight differences and an eventual divergence of molecular dynamics trajectories. See the *Errors common* page for discussion of this.

LAMMPS can run as large a problem as will fit in the physical memory of one or more processors. If you run out of memory, you must run on more processors or define a smaller problem. The amount of memory needed and how well it can be distributed across processors may vary based on the models and settings and commands used.

If you run LAMMPS in parallel via mpirun, you should be aware of the *processors* command, which controls how MPI tasks are mapped to the simulation box, as well as mpirun options that control how MPI tasks are assigned to physical cores of the node(s) of the machine you are running on. These settings can improve performance, though the defaults are often adequate.

For example, it is often important to bind MPI tasks (processes) to physical cores (processor affinity), so that the operating system does not migrate them during a simulation. If this is not the default behavior on your machine, the mpirun option --bind-to core (OpenMPI) or -bind-to core (MPICH) can be used.

If the LAMMPS command(s) you are using support multi-threading, you can set the number of threads per MPI task via the environment variable OMP_NUM_THREADS, before you launch LAMMPS:

```
export OMP_NUM_THREADS=2    # bash
setenv OMP_NUM_THREADS 2    # csh or tcsh
```

This can also be done via the *package* command or via the *-pk command-line switch* which invokes the package command. See the *package* command or *Speed* doc pages for more details about which accelerator packages and which commands support multi-threading.

You can experiment with running LAMMPS using any of the input scripts provided in the examples or bench directory. Input scripts are named in.* and sample outputs are named log.*.P where P is the number of processors it was run on.

Some of the examples or benchmarks require LAMMPS to be built with optional packages.

4.2 Command-line options

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

- *-e or -echo*
- *-h or -help*
- *-i or -in*
- *-k or -kokkos*
- *-l or -log*
- *-mdi*
- *-m or -mpicolor*
- *-c or -cite*
- *-nc or -nocite*
- *-nb or -nonbuf*
- *-pk or -package*

- *-p or -partition*
- *-pl or -plog*
- *-ps or -pscreen*
- *-ro or -reorder*
- *-r2data or -restart2data*
- *-r2dump or -restart2dump*
- *-r2info or -restart2info*
- *-sc or -screen*
- *-sr or skiprun*
- *-sf or -suffix*
- *-v or -var*

For example, the lmp_mpi executable might be launched as follows:

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

-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.

-help

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

-in file

Specify a file to use as an input script. This is an optional but recommended switch when running LAMMPS in one-partition mode. If it is not specified, LAMMPS reads its script from standard input, typically from a script via I/O redirection; e.g. lmp_linux < in.run. With many MPI implementations I/O redirection also works in parallel, but using the -in flag will always work.

Note that this is a required switch when running LAMMPS in multi-partition mode, since multiple processors cannot all read from stdin concurrently. The file name may be "none" for starting multi-partition calculations without reading an initial input file from the library interface.

-kokkos on/off keyword/value ...

Explicitly enable or disable KOKKOS support, as provided by the KOKKOS package. Even if LAMMPS is built with this package, as described in the *the KOKKOS package page*, this switch must be set to enable running with KOKKOS-enabled styles the package provides. If the switch is not set (the default), LAMMPS will operate as if the KOKKOS

package were not installed; i.e. you can run standard LAMMPS or with the GPU or OPENMP packages, for testing or benchmarking purposes.

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

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

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

```
device Nd
```

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

```
gpus Ng Ns
```

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

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

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

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

```
threads Nt
```

This option assigns Nt number of threads to each MPI task for performing work when Kokkos is executing in OpenMP or pthreads mode. The default is Nt = 1, which essentially runs in MPI-only mode. If there are Np MPI tasks per physical node, you generally want Np*Nt = the number of physical cores per node, to use your available hardware optimally. This also sets the number of threads used by the host when LAMMPS is compiled with CUDA=yes.

Deprecated since version 22Dec2022.

Support for the "numa" or "n" option was removed as its functionality was ignored in Kokkos for some time already.

-log file

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

-mdi 'multiple flags'

This flag is only recognized and used when LAMMPS has support for the MolSSI Driver Interface (MDI) included as part of the *MDI* package. This flag is specific to the MDI library and controls how LAMMPS interacts with MDI. There are usually multiple flags that have to follow it and those have to be placed in quotation marks. For more information about how to launch LAMMPS in MDI client/server mode please refer to the *MDI Howto*.

-mpicolor color

If used, this must be the first command-line argument after the LAMMPS executable name. It is only used when LAMMPS is launched by an mpirun command which also launches another executable(s) at the same time. (The other executable could be LAMMPS as well.) The color is an integer value which should be different for each executable (another application may set this value in a different way). LAMMPS and the other executable(s) perform an MPI_Comm_split() with their own colors to shrink the MPI_COMM_WORLD communication to be the subset of processors they are actually running on.

-cite style or **file name**

Select how and where to output a reminder about citing contributions to the LAMMPS code that were used during the run. Available keywords for styles are "both", "none", "screen", or "log". Any other keyword will be considered a file name to write the detailed citation info to instead of logfile or screen. Default is the "log" style where there is a short summary in the screen output and detailed citations in BibTeX format in the logfile. The option "both" selects the detailed output for both, "none", the short output for both, and "screen" will write the detailed info to the screen and the short version to the log file. If a dedicated citation info file is requested, the screen and log file output will be in the short format (same as with "none").

See the *citation page* for more details on how to correctly reference and cite LAMMPS.

-nocite

Disable generating a citation reminder (see above) at all.

-nonbuf

Added in version 15Sep2022.

Turn off buffering for screen and logfile output. For performance reasons, output to the screen and logfile is usually buffered, i.e. output is only written to a file if its buffer - typically 4096 bytes - has been filled. When LAMMPS crashes for some reason, however, that can mean that there is important output missing. With this flag the buffering can be turned off (only for screen and logfile output) and any output will be committed immediately. Note that when running in parallel with MPI, the screen output may still be buffered by the MPI library and this cannot be changed by LAMMPS. This flag should only be used for debugging and not for production simulations as the performance impact can be significant, especially for large parallel runs.

-package style args ....

Invoke the *package* command with style and args. The syntax is the same as if the command appeared at the top of the input script. For example -package gpu 2 or -pk gpu 2 is the same as *package gpu 2* in the input script. The possible styles and args are documented on the *package* doc page. This switch can be used multiple times, e.g. to set options for the INTEL and OPENMP packages which can be used together.

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

-partition 8x2 4 5 ...

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

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

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

-plog file

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

-pscreen file

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

-reorder

This option has 2 forms:

```
-reorder nth N
-reorder custom filename
```

Reorder the processors in the MPI communicator used to instantiate LAMMPS, in one of several ways. The original MPI communicator ranks all P processors from 0 to P-1. The mapping of these ranks to physical processors is done by MPI before LAMMPS begins. It may be useful in some cases to alter the rank order. E.g. to ensure that cores within each node are ranked in a desired order. Or when using the *run_style verlet/split* command with 2 partitions to ensure that a specific Kspace processor (in the second partition) is matched up with a specific set of processors in the first partition. See the *General tips* page for more details.

If the keyword *nth* is used with a setting *N*, then it means every Nth processor will be moved to the end of the ranking. This is useful when using the *run_style verlet/split* command with 2 partitions via the -partition command-line switch. The first set of processors will be in the first partition, the second set in the second partition. The -reorder command-line switch can alter this so that the first N procs in the first partition and one proc in the second partition will be ordered consecutively, e.g. as the cores on one physical node. This can boost performance. For example, if you use -reorder nth 4 and -partition 9 3 and you are running on 12 processors, the processors will be reordered from

```
0 1 2 3 4 5 6 7 8 9 10 11
```

to

```
0 1 2 4 5 6 8 9 10 3 7 11
```

so that the processors in each partition will be

```
0 1 2 4 5 6 8 9 10
3 7 11
```

See the "processors" command for how to ensure processors from each partition could then be grouped optimally for quad-core nodes.

If the keyword is *custom*, then a file that specifies a permutation of the processor ranks is also specified. The format of the reorder file is as follows. Any number of initial blank or comment lines (starting with a "#" character) can be present. These should be followed by P lines of the form:

```
I J
```

where P is the number of processors LAMMPS was launched with. Note that if running in multi-partition mode (see the -partition switch above) P is the total number of processors in all partitions. The I and J values describe a permutation of the P processors. Every I and J should be values from 0 to P-1 inclusive. In the set of P I values, every proc ID should appear exactly once. Ditto for the set of P J values. A single I,J pairing means that the physical processor with rank I in the original MPI communicator will have rank J in the reordered communicator.

Note that rank ordering can also be specified by many MPI implementations, either by environment variables that specify how to order physical processors, or by config files that specify what physical processors to assign to each MPI rank. The -reorder switch simply gives you a portable way to do this without relying on MPI itself. See the *processors file* command for how to output info on the final assignment of physical processors to the LAMMPS simulation domain.

-restart2data restartfile datafile keyword value ...

Convert the restart file into a data file and immediately exit. This is the same operation as if the following 2-line input script were run:

```
read_restart restartfile
write_data datafile keyword value ...
```

The specified restartfile and/or datafile name may contain the wild-card character "*". The restartfile name may also contain the wild-card character "%". The meaning of these characters is explained on the *read_restart* and *write_data*

doc pages. The use of "%" means that a parallel restart file can be read. Note that a filename such as file.* may need to be enclosed in quotes or the "*" character prefixed with a backslash ("") to avoid shell expansion of the "*" character.

The syntax following restartfile, namely

```
datafile keyword value ...
```

is identical to the arguments of the *write_data* command. See its documentation page for details. This includes its optional keyword/value settings.

-restart2dump restartfile group-ID dumpstyle dumpfile arg1 arg2 ...

Convert the restart file into a dump file and immediately exit. This is the same operation as if the following 2-line input script were run:

```
read_restart restartfile
write_dump group-ID dumpstyle dumpfile arg1 arg2 ...
```

Note that the specified restartfile and dumpfile names may contain wild-card characters ("*" or "%") as explained on the *read_restart* and *write_dump* doc pages. The use of "%" means that a parallel restart file and/or parallel dump file can be read and/or written. Note that a filename such as file.* may need to be enclosed in quotes or the "*" character prefixed with a backslash ("") to avoid shell expansion of the "*" character.

The syntax following restartfile, namely

```
group-ID dumpstyle dumpfile arg1 arg2 ...
```

is identical to the arguments of the *write_dump* command. See its documentation page for details. This includes what per-atom fields are written to the dump file and optional dump_modify settings, including ones that affect how parallel dump files are written, e.g. the *nfile* and *fileper* keywords. See the *dump_modify* page for details.

-restart2info restartfile keyword ...

Added in version 29Aug2024.

Write out some info about the restart file and and immediately exit. This is the same operation as if the following 2-line input script were run:

```
read_restart restartfile
info system group computes fixes
```

The specified restartfile name may contain the wild-card character "*". The restartfile name may also contain the wild-card character "%". The meaning of these characters is explained on the *read_restart* documentation. The use of "%" means that a parallel restart file can be read. Note that a filename such as file.* may need to be enclosed in quotes or the "*" character prefixed with a backslash ("") to avoid shell expansion of the "*" character.

Optional keywords may follow the restartfile argument. These must be valid keywords for the *info command*. The most useful ones - *system*, *group*, *computes*, and *fixes* - are already applied. Appending keywords like *coeffs* or *communication* may provide additional useful information stored in the restart file.

-screen file

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

-skiprun

Insert the command *timer timeout 0 every 1* at the beginning of an input file or after a *clear* command. This has the effect that the entire LAMMPS input script is processed without executing actual *run* or *minimize* and similar commands (their main loops are skipped). This can be helpful and convenient to test input scripts of long running calculations for correctness to avoid having them crash after a long time due to a typo or syntax error in the middle or at the end.

-suffix style args

Use variants of various styles if they exist. The specified style can be *gpu*, *intel*, *kk*, *omp*, *opt*, or *hybrid*. These refer to optional packages that LAMMPS can be built with, as described in *Accelerate performance*. The "gpu" style corresponds to the GPU package, the "intel" style to the INTEL package, the "kk" style to the KOKKOS package, the "opt" style to the OPT package, and the "omp" style to the OPENMP package. The hybrid style is the only style that accepts arguments. It allows for two packages to be specified. The first package specified is the default and will be used if it is available. If no style is available for the first package, the style for the second package will be used if available. For example, -suffix hybrid intel omp will use styles from the INTEL package if they are installed and available, but styles for the OPENMP package otherwise.

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

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

For the GPU package, using this command-line switch also invokes the default GPU settings, as if the command "package gpu 1" were used at the top of your input script. These settings can be changed by using the -package gpu command-line switch or the *package gpu* command in your script.

For the INTEL package, using this command-line switch also invokes the default INTEL settings, as if the command "package intel 1" were used at the top of your input script. These settings can be changed by using the -package intel command-line switch or the *package intel* command in your script. If the OPENMP package is also installed, the hybrid style with "intel omp" arguments can be used to make the omp suffix a second choice, if a requested style is not available in the INTEL package. It will also invoke the default OPENMP settings, as if the command "package omp 0" were used at the top of your input script. These settings can be changed by using the -package omp command-line switch or the *package omp* command in your script.

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

For the OMP package, using this command-line switch also invokes the default OMP settings, as if the command "package omp 0" were used at the top of your input script. These settings can be changed by using the -package omp command-line switch or the *package omp* command in your script.

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

-var name value1 value2 ...

Specify a variable that will be defined for substitution purposes when the input script is read. This switch can be used multiple times to define multiple variables. "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 the *Parsing rules* page for more info on using variables in input scripts.

Note

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

4.3 Screen and logfile output

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

```
Loop time of 0.942801 on 4 procs for 300 steps with 2004 atoms

Performance: 54.985 ns/day, 0.436 hours/ns, 318.201 timesteps/s, 637.674 katom-step/s
195.2% CPU use with 2 MPI tasks x 2 OpenMP threads

MPI task timing breakdown:
Section |  min time  |  avg time  |  max time  |%varavg| %total
---------------------------------------------------------------
Pair    | 0.61419    | 0.62872    | 0.64325    |   1.8 | 66.69
Bond    | 0.0028608  | 0.0028899  | 0.002919   |   0.1 |  0.31
Kspace  | 0.12652    | 0.14048    | 0.15444    |   3.7 | 14.90
Neigh   | 0.10242    | 0.10242    | 0.10242    |   0.0 | 10.86
Comm    | 0.026753   | 0.027593   | 0.028434   |   0.5 |  2.93
Output  | 0.00018341 | 0.00030942 | 0.00043542 |   0.0 |  0.03
Modify  | 0.039117   | 0.039348   | 0.039579   |   0.1 |  4.17
Other   |            | 0.001041   |            |       |  0.11

Nlocal:          1002 ave        1006 max         998 min
Histogram: 1 0 0 0 0 0 0 0 0 1
Nghost:         8670.5 ave        8691 max        8650 min
Histogram: 1 0 0 0 0 0 0 0 0 1
Neighs:         354010 ave      357257 max      350763 min
Histogram: 1 0 0 0 0 0 0 0 0 1
```

```
Total # of neighbors = 708020
Ave neighs/atom = 353.30339
Ave special neighs/atom = 2.3403194
Neighbor list builds = 26
Dangerous builds = 0
```

The first section provides a global loop timing summary. The *loop time* is the total wall-clock time for the MD steps of the simulation run, excluding the time for initialization and setup (i.e. the parts that may be skipped with *run N pre no*). The *Performance* line is provided for convenience to help predict how long it will take to run a desired physical simulation and to have numbers useful for performance comparison between different simulation settings or system sizes. The *CPU use* line provides the CPU utilization per MPI task; it should be close to 100% times the number of OpenMP threads (or 1 if not using OpenMP). Lower numbers correspond to delays due to file I/O or insufficient thread utilization from parts of the code that have not been multi-threaded.

The *MPI task* section gives the breakdown of the CPU run time (in seconds) into major categories:

- *Pair* = non-bonded force computations
- *Bond* = bonded interactions: bonds, angles, dihedrals, impropers
- *Kspace* = long-range interactions: Ewald, PPPM, MSM
- *Neigh* = neighbor list construction
- *Comm* = inter-processor communication of atoms and their properties
- *Output* = output of thermodynamic info and dump files
- *Modify* = fixes and computes invoked by fixes
- *Other* = all the remaining time

For each category, there is a breakdown of the least, average and most amount of wall time any processor spent on this category of computation. The "%varavg" is the percentage by which the max or min varies from the average. This is an indication of load imbalance. A percentage close to 0 is perfect load balance. A large percentage is imbalance. The final "%total" column is the percentage of the total loop time is spent in this category.

When using the *timer full* setting, an additional column is added that also prints the CPU utilization in percent. In addition, when using *timer full* and the *package omp* command are active, a similar timing summary of time spent in threaded regions to monitor thread utilization and load balance is provided. A new *Thread timings* section is also added, which lists the time spent in reducing the per-thread data elements to the storage for non-threaded computation. These thread timings are measured for the first MPI rank only and thus, because the breakdown for MPI tasks can change from MPI rank to MPI rank, this breakdown can be very different for individual ranks. Here is an example output for this section:

```
Thread timings breakdown (MPI rank 0):
Total threaded time 0.6846 / 90.6%
Section |  min time  |  avg time  |  max time  |%varavg| %total
---------------------------------------------------------------
Pair    | 0.5127     | 0.5147     | 0.5167     |   0.3 | 75.18
Bond    | 0.0043139  | 0.0046779  | 0.0050418  |   0.5 |  0.68
Kspace  | 0.070572   | 0.074541   | 0.07851    |   1.5 | 10.89
Neigh   | 0.084778   | 0.086969   | 0.089161   |   0.7 | 12.70
Reduce  | 0.0036485  | 0.003737   | 0.0038254  |   0.1 |  0.55
```

The third section above lists the number of owned atoms (Nlocal), ghost atoms (Nghost), and pairwise neighbors stored per processor. The max and min values give the spread of these values across processors with a 10-bin histogram showing the distribution. The total number of histogram counts is equal to the number of processors.

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

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

```
Minimization stats:
  Stopping criterion = linesearch alpha is zero
  Energy initial, next-to-last, final =
        -6372.3765206     -8328.46998942     -8328.46998942
  Force two-norm initial, final = 1059.36 5.36874
  Force max component initial, final = 58.6026 1.46872
  Final line search alpha, max atom move = 2.7842e-10 4.0892e-10
  Iterations, force evaluations = 701 1516
```

The first line prints the criterion that determined minimization was converged. The next line lists the initial and final energy, as well as the energy on the next-to-last iteration. The next 2 lines give a measure of the gradient of the energy (force on all atoms). The 2-norm is the "length" of this 3N-component force vector; the largest component (x, y, or z) of force (infinity-norm) is also given. Then information is provided about the line search and statistics on how many iterations and force-evaluations the minimizer required. Multiple force evaluations are typically done at each iteration to perform a 1d line minimization in the search direction. See the *minimize* page for more details.

If a *kspace_style* long-range Coulombics solver that performs FFTs was used during the run (PPPM, Ewald), then additional information is printed, e.g.

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

The first line is the time spent doing 3d FFTs (several per timestep) and the fraction it represents of the total KSpace time (listed above). Each 3d FFT requires computation (3 sets of 1d FFTs) and communication (transposes). The total flops performed is 5Nlog_2(N), where N is the number of points in the 3d grid. The FFTs are timed with and without the communication and a Gflop rate is computed. The 3d rate is with communication; the 1d rate is without (just the 1d FFTs). Thus you can estimate what fraction of your FFT time was spent in communication, roughly 75% in the example above.

4.4 Running LAMMPS on Windows

To run a serial (non-MPI) executable, follow these steps:

- Get a command prompt by going to Start->Run... , then typing "cmd".
- Move to the directory where you have your input script, (e.g. by typing: cd "Documents").
- At the command prompt, type "lmp -in in.file", where in.file is the name of your LAMMPS input script.

Note that the serial executable includes support for multi-threading parallelization from the styles in the OPENMP packages. To run with 4 threads, you can type this:

```
lmp -in in.lj -pk omp 4 -sf omp
```

For the MPI executable, which allows you to run LAMMPS under Windows in parallel, follow these steps.

Download and install a compatible MPI library binary package:

- for 32-bit Windows: mpich2-1.4.1p1-win-ia32.msi
- for 64-bit Windows: mpich2-1.4.1p1-win-x86-64.msi

The LAMMPS Windows installer packages will automatically adjust your path for the default location of this MPI package. After the installation of the MPICH2 software, it needs to be integrated into the system. For this you need to start a Command Prompt in *Administrator Mode* (right click on the icon and select it). Change into the MPICH2 installation directory, then into the subdirectory **bin** and execute **smpd.exe -install**. Exit the command window.

- Get a new, regular command prompt by going to Start->Run... , then typing "cmd".
- Move to the directory where you have your input file (e.g. by typing: cd "Documents").

Then you can run the executable in serial like in the example above or in parallel using MPI with one of the following commands:

```
mpiexec -localonly 4 lmp -in in.file
mpiexec -np 4 lmp -in in.file
```

where in.file is the name of your LAMMPS input script. For the latter case, you may be prompted to enter the password that you set during installation of the MPI library software.

In this mode, output may not immediately show up on the screen, so if your input script takes a long time to execute, you may need to be patient before the output shows up.

The parallel executable can also run on a single processor by typing something like this:

```
lmp -in in.lj
```

Note that the parallel executable also includes OpenMP multi-threading, which can be combined with MPI using something like:

```
mpiexec -localonly 2 lmp -in in.lj -pk omp 2 -sf omp
```