GROMACS - Task #2616

Model for MD state

08/21/2018 04:21 AM - Roland Schulz

Status:	New
Priority:	Normal
Assignee:	
Category:	mdrun
Target version:	
Difficulty:	uncategorized

Description

Currently we don't have a coherent model for our MD state. Most of the important information (such as t_state, gmx_domdec_t, t_inputrec, t_commrec, MdrunOptions, t_mdatoms, ...) is just widely passed around to whoever needs it. Given how widely they are passed, it would be about as good if they simply were global variables. At least then most function would have a reasonable number of arguments.

Recent efforts at modernization (such as https://gerrit.gromacs.org/c/7854, https://gerrit.gromacs.org/c/8088) try to avoid the problem by storing pointers to them in some Handler/Task object but that doesn't actually solve the problem. It is also ad-hoc and the states which store the information have no relationship to the data stored. As far as I know there is no suggestion for a long term solution yet.

One thing we need to do is collect requirements for our MD state data-structure / object model. A start of this is the following:

One requirement is, that we need to be able to write certain data (e.g. forces, counter, log) from multiple tasks in a thread safe way to support scheduling. For this I suggest we split out all data which is regularly modified from the other, so that data which is never(/rarely) modified can be made a read-only (/can be accessed by most through a read only view). For the data which needs per step writing in parallel from different tasks and are write-only (such as force) we can create buffers which automatically handle the buffer reduction.

Do we need the capability to create more than one state of any of the major MD variables? In other words for the API effort do we want to support creating multiple states in the same process? Or do we want to have things like -multi/-replex continue to be limited to MPI and have always one state per MD process? Where do we already already need copies of state objects (such as t_state in minimize)? And are there exceptions either way? Meaning do we want most state to have a single/multiple instance(s) but some state should be different.

Another requirement is that we need to be able to unit test functions. If we make sure that the global state is fast to default construct and have utility function for testing to initialize parts of the state required for the function under test.

I think a solution might be the following:
Basic idea: I think there should be one object which fully described the total state of the MD simulation. This should contain everything needed to perform an MD step excluding only data which changes frequently (e.g. forces, note 4). Obviously it shouldn't be one class. But instead one class with (smart) pointers to all the relevant other data. Method signatures can be simplified to only require that object. Encapsulation is preserved (note 3). That MD state object should be shallow copyable. We have methods (such as minimize) which require to make local modifications to the state. Without being able to make a full shallow copy of the MD state object, such a method would have to have one object which contains all the non-modified state (where the modified parts are invalid) plus another which contains the modified state. To avoid this mess, such a method should be able to make a shallow copy of the whole MD state and a deep copy for the parts it wants to modify. This can also help with initialization.

Details:
Have one MDState object which contains a shared_ptr<const X> to each of the state objects. Every method gets a const& to that MDState object (or only the subobject needed but not multiple sub-objects to avoid the many arguments). All our state objects should be made fast/shallow copyable. That means:
- any data which is expensive to copy should use also use a shared_ptr<const X> and just copy the pointer
- data which should be deep-copied by default should either not be a pointer or have a copy constructor which copies it (or use value_ptr)
- and function which needs to locally modify MDState (e.g. minimize) can make a writeable copy and modify whatever it wants (2)
- for function which needs to modify the copy given by the caller (such as initialization/input reading) has two options:
a) the caller can pass a non-const pointer to the part it wants to give write access to (obvious the caller has to have write access itself, ideally again not multiple subobjects and thus multiple arguments)
b) the callee creates a non-const copy (deep-copy for parts it wants to change, same note 2 applies) and returns that copy. The caller than commits those changes by copying them back into its version (again the caller has to have write access). This might be too expansive in some cases. But it has the advantage that it makes things exception safe (if a function returns early things don't get

written back to the original version) and lets the caller do sanity checks before committing the changes (including verifying that only expected things were deep copied). It also avoids the issue that granting write access to multiple subobjects doesn't require many arguments.

1) (got removed by edit but don't want to renumber)
2) After creating a shallow copy it is still not modifiable through it parent object (everything is pointers to const data) but because one created the deep copy one has a non-const reference to the part which was deep copied.
3) Things can be private and only accessible to a class. They can also be restricted to a module if the type the smart-pointer points to is forward declared and the implementation is local the module. For read access I don't it is helpful to know what part of MDState is accessed from the function signature. And it can be documented. Write access has to be granted specially and thus it is visible in the code what parts can be written too.
4) "data which changes frequently" only refers to data which has observable simulation result an needs to be correctly propagate to the next step. Mutable data (caches, log, tmp-buffers) would be fine.

Related issues:

Related to GROMACS - Task #2644: Replace compute_globals	**New**
Related to GROMACS - Task #1793: cleanup of integration loop	**New**

Associated revisions

Revision 04645ace - 09/11/2018 09:47 AM - Pascal Merz

Added trivial const qualifiers

Const qualifiers were added to function arguments. No changes were made inside function bodies, except for adding const qualifiers to convenience references within functions. These changes help the development of a MD state object, as they reflect more clearly where write access to data is *really* needed.

Refs #2616

Change-Id: I779b4dfed2fbd0ad129e226c9b535ed97a8bb01f

History

#1 - 08/21/2018 05:41 AM - Roland Schulz

- Description updated

#2 - 08/30/2018 04:04 AM - Michael Shirts

> Where do we already already need copies of state objects (such as t_state in minimize)? And are there exceptions either way? Meaning do we want most state to have a single/multiple instance(s) but some state should be different.

Monte Carlo algorithms need to have a saved copy of the state that can be returned to. For some algorithms (not the ones we will implement first, but there are some of these), you might actually try several different MC moves, and only accept one of them in the end.

#3 - 08/30/2018 08:13 AM - Pascal Merz

I agree that this is an important question, and one that is central to the new integrator framework. Thank you for bringing this up! I'll be out of the office for the rest of the week, but I'll be putting more thoughts in that early next week.

#4 - 08/30/2018 04:01 PM - Mark Abraham

Agree this is of high importance. I have many thoughts here, and will contribute tomorrow.

#5 - 09/01/2018 09:10 AM - Mark Abraham

Sorry, out of time today for a detailed post, but I agree with Roland's general leaning to something like a referenced-counted vector. Pascal and I spent a bunch of time talking through these issues in May when he visited, and we mooted that there are several use cases for being able to get a read-only or read-write view of a vector in the state and have it persist, e.g.

- SD integrator needs to keep the unconstrained velocities,
- trajectory output would like to be able to handling writing from a different thread than the main work path, but without blocking the main work path with a deep copy
- any kind of task parallel engine needs to have tasks that can have a handle to access the logical current state (or parts of it), **before** we are certain what memory address it is found at

Fundamentally that means we need both the notion of preparing to take a view of what is / will be the current state, and taking a view of the current state to which this kernel has been bound.

Agree also that we need a much more sophisticated approach to force-virial-energy reduction.

For example, a position update kernel needs a read-only view on initial positions and a write-only view on updated positions, but those do not have to be the same memory, and the latter does not need to be initialized. The component that interprets the requirements for views of the current state for the kernel can look at the reference count of the position vector and create the two views from that same vector (when the reference count is one; using cache to best effect), or create the two views from different vectors (when someone else will read the initial positions, the result has to go into a difference view; that costs more cache).

Also, the state data structure needs to be able to be extended by the modules that are in play, rather than being a monolithic thing that is aware of all possible modules (as now). Those modules need to be capable of understanding how those aspects of state get handled in grompp and mdrun setup, and how they get checkpointed (whether to disk or for MC snapshots). This will need some organization, e.g that modules implement an IStateContributor, etc. That way things like thermostats that work differently with leapfrog vs velocity-verlet integrators can be different classes, which should help avoid things like the incomprehensible mess of combined full and half step data structures that we have now.

#6 - 09/10/2018 04:52 PM - Gerrit Code Review Bot

Gerrit received a related DRAFT patchset '1' for Issue #2616.
Uploader: Pascal Merz (pascal.merz@colorado.edu)
Change-Id: gromacs~master~I779b4dfed2fbd0ad129e226c9b535ed97a8bb01f
Gerrit URL: https://gerrit.gromacs.org/8329

#7 - 09/13/2018 01:53 AM - Gerrit Code Review Bot

Gerrit received a related patchset '3' for Issue #2616.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: gromacs~master~I44a7193cc5492e3d6b37b896e16c6b438d2e28cc
Gerrit URL: https://gerrit.gromacs.org/7990

#8 - 09/14/2018 05:12 PM - Prashanth Kanduri

This idea also goes well with the aims in #2594. The key objective of this idea is to separate the molecular system specific data from the GROMACS specific objects (centred around communicators, domain decomposition, etc).

The molecular system information overlaps strongly with the state of the system. It would be nice if these two things could be unified. Not only for the programming benefit, but also perhaps hot starting a simulation based on the 'System' object might be helpful for perturbation analysis, where spaces of trajectories might be explored using branching of independent ensembles from a set point.

#9 - 09/14/2018 05:16 PM - Mark Abraham

- Related to Task #2644: Replace compute_globals added

#10 - 09/17/2018 09:56 AM - Mark Abraham

- Related to Task #1793: cleanup of integration loop added