GROMACS - Feature #1137

Proposal for integrator framework (do_md) in future GROMACS

02/05/2013 08:59 PM - Michael Shirts

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

Description

I'm going to put these all together at first, and then start breaking them up when we move from proposals and discussion to actual tasks.

I. Remove the iterative structure required for MTTK integration with constraints. MTTK integration will be left in (still a very nice formalism), but only for systems without constraints. This way, we get the benefits for some systems (like LJ) without the ugly and often-unstable-in-single precision iteration. This will help solve some of the issues with the difficulty to work with the md framework. I'll discuss how to get longer timesteps with MTTK below . . . One disadvantage: Will not be able to do do full, 'correct' pressure control *dynamics* with constraint systems, but pressure control is a bit artificial already; one will at least be able to constant pressure sampling using a MC barostat (see below). This is a todo in the next month or two (MRS). It will make it possible to make the integrator much simpler.

II. Convert the md integrator to a full Trotter decomposition framework.

This will have a number of advantages going forward. It will make it much easier to fit md and md-vv into the same framwork. Note that this does not *change* either integrator; it just changes how it is written out and makes in more clear what steps go together in the code and where to put modifications. This is partly already in the code but not included. This formalism easily extends to the pressure control and temperature control algorithms; again, in most cases, this is the formalism that's implemented already, it's just a matter of rewriting it so it's easier to see where to add additional code when new methods are added.

III. Make it easier to do multtstep integration using the Trotter decomposition framework.

It's already possible to do types of multistep integration using the Trotter decomposition framework; for example, doing pressure control or temperature control only every N steps. However, it would be useful to make it possible to only evaluate certain components of the potential energy / force every N steps as well, where N can varied independently for each component.

This will make a number of things possible, such as
1) harmonic bonds with short timesteps, with long timesteps for the nonbonded.
2) more rigorous short/long cutoff integration
3) Make it less computationally intensive to do PME/LJ, as this only needs to be called relatively rarely (10-20 steps, likely). We know that PME/LJ is required for getting lipid densities to be cutoff independent.

IV. Adding a Monte Carlo barostat

This consists in random volume changes, accept and reject based on Boltzmann weight exp(-(E+PV)). This has an advantage that there is no need for fancy integrators for NPT, which is giant pain, and the virial doesn't even need to be calculated - Virial only needs to be calculated when you want to see the pressure (every nstcalcenergy) Generalizable to anisotropic very easily. MTTK is especially a giant pain, and requires iteration to work with constraints, and current Parrinello-Rahman is incorrect because it uses mismatched virial and kinetic energies.

One disadvantage: this will require a second force/energy calculations per application (though only every nstpcouple steps). Computationally, this is not a big problem, but it will require making it easier to run do_force multiple times in a loop. Again, only the energy must be calculated additional times, not the forces.

V. Redo the main md loop to allow for either MC or MD.

Now that MD can be seen as rejectionless MD, so it is easiest to write the 'parent' loop as MC., with MD being a type of MC with 100% acceptance. Clearly point IV (MC barostat) fits under this as well. Expanded ensemble and replica exchange also fit into this.

Advantage: lots of potential advantages for calculating thermodynamic quantities and equilibrium ensembles. Lots of interest in improving MC in gromacs. Could be very useful with implicit solvent, especially, where large conformation changes could be made. Aggregation in implicit solvent can particularly be made faster. Also very useful for simple fluids.

Disadvantage: Makes it a little bit more complicated, but not much. Payoff should be worth it. As long as the general formalism is set up, extending MC by other developers or volunteers is very easy.

Thoughts? I can probably handle a lot of the gruntwork here, though there are some places help would be useful; for example, in adding the ability to call only parts of the force calculation each time.

Related issues:

Related to GROMACS - Task #1793: cleanup of integration loop	**New**	
Related to GROMACS - Feature #1867: make coupling implementations reversible	**New**	
Related to GROMACS - Bug #1339: Center of mass drift with Nose-Hoover, MTTK a...	**New**	**09/19/2013**

Associated revisions

Revision 30113ccc - 10/18/2013 12:11 AM - Mark Abraham

Move mdrun trajectory writing into wrapper function

Refs #1292, #1193, #1137

Change-Id: I3f19e0995ff7fab465184d5bab8c2683260af853

Revision 8df8c14d - 10/28/2013 06:36 PM - Mark Abraham

Create fileio module

This patch contains only code motion. There are no functional code changes. Moves lots of I/O code into src/gromacs/fileio. This means lots of changes to avoid having everything as a compile-time dependency of everything else because everything is pulled in via typedefs.h, etc. Note that src/gromacs/legacyheaders/filenm.h and src/gromacs/legacyheaders/types/filenm.h have been consolidated into src/gromacs/fileio/filenm.h.

I/O code in files named stat*[ch] now lives in various new files in fileio.

Files outside of the module now #include its header files in a proper way, e.g. #include #include "../fileio/filenm.h" or "gromacs/fileio/filenm.h" according to whether they are an installed header, or not. Files within the module are blessed and do not need that qualifier.

This module installs most of its headers (because they're almost all inter-dependent; gmxfio_int.h is not installed because it is only useful internally, vmdio.h is not installed because it relies on a header from src/external)

Files in new module

- conform to preferred include-guard format.
- have up-to-date copyright headers thanks to Teemu's automatic script
- that are installed headers refer to other GROMACS include files via relative paths

Moves mdrun trajectory writing into wrapper function.

Removes small pieces of I/O code that was hiding behind "#if 0".

Some pieces of I/O code specific to the gmxpreprocess module have remained there.

Moved a cppcheck suppression to follow matio.cpp to its new home.

Minor fix to xdrf.h logic, since it is now the subject of a CMake test.

Refs #1292, #1193, #1137

Change-Id: I820036298d574966d596ab9e258ed8676e359184

Revision 488464e7 - 12/15/2014 09:40 PM - Mark Abraham

Removing iteration + constraints framework

Getting rid of iteration + constraints required by the use of MTTK + constraints, in order to simplify the main loop.

Eliminated related variables and arguments that are now unused.

Left some otherwise useless brace pairs in do_md(), so that uncrustify-friendly formatting was preserved, so we can more easily review this for correctness. Left TODOs to remove those braces later.

Implemented mdrun check so that an old .tpr with MTTK + any form of constraints cannot be run.

Refs #1137

Change-Id: I22816de7db4420a66341fa8bf35d967a71ad6568

Revision 6ead809b - 11/16/2015 03:22 PM - Mark Abraham

Remove "support" for twin-range with VV integrators

Group-scheme twin-range non-bonded interactions never worked with VV+constraints, and removing it was a to-do item. There are no plans to make it work with VV, and there are plans to remove the twin-range scheme entirely, as well as rework leap-frog more closerly into the Trotter scheme.

Refs #1137, #1793

Change-Id: Ibd70b5397568bfcd328cd6dd1c5c99384d7aaca8

Revision bd1807e6 - 01/07/2016 09:16 AM - Mark Abraham

Remove SD2 integrator

This integrator has known problems, and is in all ways inferior to sd. It has no tests, and was deprecated in GROMACS 5.0. There are no plans to replace it.

Old checkpoint files written by sd2 integrators will no longer be readable, but this is not a problem since continuing an MD simulation across minor versions is not supported, and sd2 has no replacement implementation anyway.

Refs #1137

Change-Id: I71b688a67a80e1f55134e81dab7a11a66942cff4

Revision efa13a69 - 08/27/2019 06:51 PM - Mark Abraham

Add integration tests for exact restarts

These tests demonstrates the extent to which mdrun checkpoint restarts reproduce the same run that would have taken place without the restart.

I've been working on these, and the bugs they exposed, for a few years, but the code has been fixed for a few years now.

The tests don't run with OpenCL because they have caused driver out of memory issues.

Refs #1137, #1793, #1882, #1883

Change-Id: I8bc441d945f13158bbe10f097e772ea87cc6a559

History

#1 - 02/05/2013 09:04 PM - Michael Shirts

- Description updated

Yeah, formatting looks awful . . . trying to figure out how to make it look better. Comments can be updated, but I can't figure out how to do the same with the original post. [Update] OK, I figured this out now. For later users, go to update, and then at the top of the updating section, click on the 'more' next to 'Change properties' , and you will be able to edit the original post.

#2 - 02/05/2013 09:04 PM - Michael Shirts

- Assignee set to Michael Shirts

#3 - 02/05/2013 09:09 PM - Michael Shirts

- Description updated

#4 - 02/05/2013 09:11 PM - David van der Spoel

As a rule of thumb it would be great to separate the math (integrator/constraints) from the physics (forces). Maybe that is not feasible though.

Apropos PME/LJ, at least for relatively simple membrane systems one can relatively simply implement an analytical dispersion correction that is Z-dependent. It is useful for surface tension calculations as well.

#5 - 02/05/2013 09:13 PM - Michael Shirts

- Description updated

#6 - 02/05/2013 09:14 PM - Michael Shirts

- Description updated

#7 - 02/05/2013 09:16 PM - Michael Shirts

- Description updated

#8 - 02/05/2013 10:43 PM - Michael Shirts

David van der Spoel wrote:

> As a rule of thumb it would be great to separate the math (integrator/constraints) from the physics (forces). Maybe that is not feasible though.

Could you be more specific? I'm not sure what you mean here.

#9 - 02/05/2013 10:44 PM - Michael Shirts

> Apropos PME/LJ, at least for relatively simple membrane systems one can relatively simply implement an analytical dispersion correction that is Z-dependent. It is useful for surface tension calculations as well.

Right, but what if the membrane starts to buckle? Then the Z-dependence fails. I think it's best to have something that is more general, and involves far fewer assumptions!

#10 - 02/06/2013 11:05 AM - David van der Spoel

What I meant is that the integrator should be ignorant of what it is integrating. It gets x,v,a and does it's thing. It should therefore not need any information about atoms. Again, this may be merely academic, but if this is feasible one can also test the integrator without actually doing an MD simulation.

In C++ this could be implemented - in principle - by having an integrator class with a virtual method "calc_gradient" which is then overload by the descendant class, be it mdrun or a test program.

#11 - 02/06/2013 12:06 PM - Berk Hess

Well, you do need masses, which are properties of atoms. But that's about it.
It is indeed useful to be able to test the integrator without doing a simulation, but I don't how having atoms or not comes into play here.

#12 - 02/06/2013 02:57 PM - Michael Shirts

Berk Hess wrote:

> Well, you do need masses, which are properties of atoms. But that's about it.
> It is indeed useful to be able to test the integrator without doing a simulation, but I don't how having atoms or not comes into play here.

If the inputs are accelerations, then you don't need masses, since f/m =a. For a number of reasons, that might not be the best way to actually pass the data in, but in theory, it could be done.

#13 - 02/06/2013 03:10 PM - Michael Shirts

David van der Spoel wrote:

> What I meant is that the integrator should be ignorant of what it is integrating. It gets x,v,a and does it's thing. It should therefore not need any information about atoms. Again, this may be merely academic, but if this is feasible one can also test the integrator without actually doing an MD simulation.
>
> In C++ this could be implemented - in principle - by having an integrator class with a virtual method "calc_gradient" which is then overload by the descendant class, be it mdrun or a test program.

OK, I see. The one issue that I see with this is that all integrators have certain ingredients -- position incrementor, velocity incrementor, box volume incrementor, box velocity incrementor, bath variable integrator -- and there are certain other operations that need to be done, such as computing kinetic energies and pressures, collecting statistics, performing parallel operations, writing checkpoints, etc. The biggest difference between integrators is the order that these are done.

So it's not clear one can completely modularize each integrator in the way you propose. It's very possible to modularize each of the components, but the integrator itself will be calling lots of different things that require atomic information inside it -- so testing independent of atoms may not be possible.

#14 - 02/06/2013 03:15 PM - Erik Lindahl

I'm not enough of a C++ expert to say the virtual method approach is wrong, but overall the force calculation, load balancing and all communication in the main MD loop is very complicated, and we will likely soon move to a data-driven scheme where different tasks can be carried out asynchronously whenever possible, and in particular separate operations on local-only vs. remote data. Obviously, this should include the integration too.

While it might be clean from an integrator testing point-of-view to abstract everything else away in a function, the integration is still a relatively straightforward operation compared to lots of other parallel algorithms, so I still **think** it is going to be easier to plug integration into the data-driven task framework rather than plug everything else into integration, but maybe that remains to be seen.

#15 - 02/06/2013 03:44 PM - Michael Shirts

> While it might be clean from an integrator testing point-of-view to abstract everything else away in a function, the integration is still a relatively straightforward operation compared to lots of other parallel algorithms, so I still **think** it is going to be easier to plug integration into the data-driven task framework rather than plug everything else into integration, but maybe that remains to be seen.

An advantages of Trotter factorization is that even NPT and NVT integrators become relatively straightforward operations, in that you write out the decomposition, and just make sure that all the quantities are computed in the proper order in the code (or, if required for efficiency, you swap steps that commute, and one can compile a list of those commuting steps).

I'm not quite visualizing what "data-driven task framework" means in the context of integrators, but I'm happy to address any issues there once I figure that out :)

#16 - 02/06/2013 03:50 PM - Peter Kasson

My comment to Michael on all of this is that it sound good in theory (and would indeed make a lot of cool things easier)--the best way to evaluate this would really be to take a simplified test case, implement it, and show code architecture & performance. Of course, that's a lot of work, but one could do a more feature-sparse test.

It's not on the surface an integrator issue (although it involves that code heavily), but an MC barostat really makes sense as the right way to do things (and much simpler too).

And support for MC moves in general would make a lot of nice extensions fairly straightforward. In particular, as we "library-ize" gromacs, this would make it easy for people to write code/scripts for a lot of more complicated operations without really messing with the Gromacs core code. A Good Thing (TM) IMO.

#17 - 02/06/2013 04:19 PM - David van der Spoel

I hadn't heard the suggestion Erik just made about data-driven code. That sounds a lot like the Charm framework that NAMD uses and that would be huge change in the code, wouldn't it?

In my proposal the integrator becomes the do_md routine, calling functions as needed in order to move forward in time.

Obviously everything needs to be very tightly coupled for large scale simulations.

#18 - 02/06/2013 04:29 PM - Erik Lindahl

It is still just plans, and whatever we do will be much more specific than Charm++, but we simply don't see much alternatives in a future where you are running simulations over hundreds of nodes each with multiple accelerators and maybe 128 cores each. Amdahl, imperfect load balancing and synchronization will kill us otherwise.

> In my proposal the integrator becomes the do_md routine, calling functions as needed in order to move forward in time.

Yes, that sounds nice on the high level, but it relies on the concept of a black box routine that magically does the right thing with the gradient returned

as a single vector. do_md() isn't that - even without a data-driven approach such a new integrator module will now have to be aware of and handle MPI, thread, and accelerator setup, parallelization, direct-space vs. PME decomposition and load balancing (home atoms will change), and probably do some communication too.

#19 - 02/06/2013 07:41 PM - David van der Spoel

Maybe this is just semantics but the integration algorithm is what determines which operations have to be be done in which order. Sure, we have to move data around and there are different possibilities on how to parallellize the sub-tasks. I'm not saying that it will be less complicated in the end (though one can hope). But the integrator does not have to decide (or even know) how the force computations are divided over cores.

#20 - 02/18/2013 08:54 PM - Teemu Murtola

Responding to some of Mark's thoughts from gmx-dev:

- In C++, being able to construct an MDLoop object that contains (lots of) objects that already have their own “constant” data will mean we only need to pass to methods of those objects any remaining control values for the current operation
  - passing of state information managed by letting the MDLoop own that data and have the object implementing the strategy ask for what it needs?

If it is feasible, I think a better design would be such that the MDLoop pushes any data that is required into the object. So that, e.g., the force evaluation object does not have any dependency on the MDLoop object itself. This of course requires that the MDLoop knows what data the object needs, but it would anyways be better to have control over this (in particular when it comes to modifying the data). If this is too complex, it is also possible to just pass the entire "state" into the object and let it deal with it. A design where there are no cyclic dependencies is much easier to unit test.

- Those objects will have a lot of inter-relationships, so probably need a common interface for (say) thermostat algorithms so that (say) the MDLoop update method lnows it can just call (say) the thermostat object’s method and the result will be correct, whether there’s a barostat involved, or not
  - easily done with an (abstract?) base class and overriding virtual functions
    - however, that kind of **dynamic-binding** run-time polymorphism is overkill - likely any simulation knows before it gets into the main loop that it’s only ever going to call (say) AndersenThermostat’s methods
    - the overhead from such function calls is probably not a big deal - this loop is always going to be heavily dominated by CalculateForces()
    - inheritance can maximise code re-use

I don't understand what you mean by this. What is the base class, and what would the subclasses correspond to? In particular, I don't understand how this differs from the second alternative below (any reasonable implementation that I can think of is some flavor of the second approach).

And as a nitpick, in modern object-oriented programming, re-using code is not a good justification for inheritance. ;)

- can be done by having function pointers that get set up correctly in the MDLoop constructor (i.e. “static” run-time polymorphism, as dictated by the .tpr)
  - this might lead to code duplication?
  - might lead to the current kind of conditional-heavy code, because it is now the coder’s job to choose the right code path, but hopefully only in construction

There is no need to use "function pointers" for this, but instead interfaces and classes implementing those interfaces. Those classes can still share some common implementation to avoid unnecessary code duplication. Sure, the conditionals need to exist somewhere, but in the initialization there is much more freedom to organize the code, since performance is not a concern.

- could be done with compile-time polymorphism (i.e. templates)
  - lots of duplicated object code because of the explosion of templated possibilities

There are simply too many combinations to enumerate them all for templates, and the benefit of avoiding indirect function calls is very small compared to the explosion in the binary size.

- Perhaps a good way to start to get a handle on what kinds of objects and relationships we needs is to make an ideal flowchart for a plausible subset of mdrun functionality, and see what data has to be known where. Perhaps Michael can sketch something for us that illustrates what the algorithmic requirements of a “full Trotter decomposition framework” would be. (But probably not in time for this week!)

I'm strongly in favor of this approach. I don't think there is a need for any elaborate proof-of-concepts that drill into every detail at this point, but instead it is probably better to try to understand the big picture. And this kind of flow chart (and in particular the data dependencies) is essential no matter which approach we choose in the end to implement it.

#21 - 02/18/2013 09:35 PM - Michael Shirts

The first step is to strip out the iterative algorithm required for constraints + MTTK. I am debugging a draft of this right now, and can hopefully post it by this weekend.

I'll start working on a draft of the Trotter integration framework, at least the algorithmic part. And yes, it will not be done by Wednesday! I can

probably get a draft done by the next meeting.

#22 - 02/21/2013 12:45 PM - Mark Abraham

Sounds great, Michael. We really need something upon which we can start visualizing the way data has to be accessed/transferred and the way parallelism has to be incorporated into the design.

At the teleconference 20 Feb we also discussed:

That the use of virtual functions for managing the different behaviours of related kinds of objects is acceptable - if later there's a performance cost to them, then we should do optimization once we know that.

The idea of trying to construct a container of work task objects at MDLoop construction time might help limit the explosion of conditionality. That's still going to mean some conditional code in constructing the order of the objects in the container. These objects might do their work by knowing what virtual function to call. Inside the work triggered from the ComputeForces object there's likely going to be some similar stuff going on for the task-based parallelism we have in mind.

#23 - 02/21/2013 11:08 PM - David van der Spoel

It would be good to have a flowchart in a format that we can edit and have under version control. Any suggestions? Or is this not necessary? Such a flowchart could be part of the developer documentation. Or can it be made by software like doxygen?

#24 - 02/22/2013 12:04 AM - Michael Shirts

David van der Spoel wrote:

> It would be good to have a flowchart in a format that we can edit and have under version control. Any suggestions? Or is this not necessary? Such a flowchart could be part of the developer documentation. Or can it be made by software like doxygen?

I agree it would be nice. I also do not know the right format to put one together. I'll focus on getting the latex documentation of the iterator algorithms done (estimated 2-3 weeks).

#25 - 02/22/2013 06:09 AM - Teemu Murtola

David van der Spoel wrote:

> It would be good to have a flowchart in a format that we can edit and have under version control. Any suggestions? Or is this not necessary? Such a flowchart could be part of the developer documentation. Or can it be made by software like doxygen?

This is a good idea. It can't be generated directly by Doxygen (this is a higher-level representation than what can be automatically generated). Depending a bit on what information we want to have in the graph, and how we want to represent it, I can think of the following formats (of the top of my head, did not do any searching):

- msc: http://www.mcternan.me.uk/mscgen/
- dot: http://www.graphviz.org
- graphml: http://graphml.graphdrawing.org
  All are plain text (the last is XML), so can be readily version-controlled. msc and dot may have some limitations in the amount of text that can be easily put into nodes/edges and on the layout (both are designed for automatic layout), while graphml (depending on the editor) can be used to represent anything.

The first two can be directly embedded into Doxygen documentation (using \dot, \dotfile, \msc and \mscfile tags). Have not used msc in practise, and have only used dot as an intermediate format for automatically generated graphs, so don't know whether there are good editors for them. But the syntax is not that hard to edit by hand. Have used yEd for creating graphml graphs; as graphml is quite "flexible", we probably need to agree on an editor to use to have it understand and write the same extensions.

#26 - 02/22/2013 08:16 AM - David van der Spoel

I played a bit with a tiny script called cpp2dia http://cpp2dia.sourceforge.net/ which generates dia or dot output. Unfortunately the output is not very useful. However for the purpose of designing a high level flow chart we do not need every detail. yEd maybe a decent tool indeed, although it may take some learning.

Including \dot or \msc into the code seems like a lot of work judging from the documentation at http://www.stack.nl/~dimitri/doxygen/manual/commands.html#cmddot.

#27 - 02/22/2013 09:11 PM - Teemu Murtola

David van der Spoel wrote:

> I played a bit with a tiny script called cpp2dia http://cpp2dia.sourceforge.net/ which generates dia or dot output. Unfortunately the output is not very useful. However for the purpose of designing a high level flow chart we do not need every detail.

As I said, I don't think that this kind of diagram can ever be extracted from the source code directly. It requires too much human input in what is

relevant and what not to produce a clear and concise representation of, e.g., the data and control flow within mdrun. For class diagrams and such, Doxygen already creates quite a few diagrams like inheritance and collaboration diagrams. Not all of them are currently enabled in our config, and they would require installing graphviz on Jenkins to get them produced, but you can take a look at them locally if you are interested.

> Including \dot or \msc into the code seems like a lot of work judging from the documentation at http://www.stack.nl/~dimitri/doxygen/manual/commands.html#cmddot.

I don't understand your point. If the dot/msc diagram already exists, it takes *literally* 30 seconds to include it into the documentation produced by Doxygen (well, a bit more if you want to add cross-references from the diagram and they are not yet there). If the dot/msc diagram does not exist, then the effort required to create it doesn't depend at all on whether it will be put into the Doxygen documentation or somewhere else. And I'm not proposing that we start maintaining, e.g., class diagrams manually; as I said above, Doxygen already produces at least some types of diagrams quite nicely fully automatically.

As an addition to my list, Dia is also one option (haven't tried it myself). We could also have the diagram as simple SVG, but a diagram created in a dedicated graph-drawing tool may be easier to both create initially and maintain in the long run.

#28 - 02/25/2013 04:25 AM - Michael Shirts

- Category set to mdrun

- % Done changed from 0 to 10

OK, I've committed in the "remove constraints + MTTK self-consistent iterations" each step. Next task is to put together the trotter factorization outline with suggestions for code organization. Will probably take 2-3 weeks to find time to finish, I'm estimating.

#29 - 05/22/2013 06:08 AM - Mark Abraham

Going to need something on paper soon, Michael, if we will have time to design and implement on the proposed time scale.

#30 - 08/07/2013 04:32 AM - Michael Shirts

I've been working off and on the document for this, and it's not coming quickly. I'll try to post something this weekend before I head off on travel.

However, I've come to the conclusion that probably the only way to make everything very clear will be to work on the code directly. Even to myself, it's hard to know exactly where everything fits without making the changes myself! In conjunction with this document, then I will work on the following steps:

1) Validate the removal of iteration, including catching up with master.
2) explicitly "Trotterize" the code so that both leapfrog and velocity Verlet are explicitly written in terms of full position and half step velocities. This should not change any output, as the same steps will be done in the same order -- they will just be broken up into routines differently. SD will probably be removed in this draft version, as I'm proposing an alternative to that I think will work better in the end (see next post).
3) see if I can implement a VERY basic Monte Carlo wrapping code in the integrator, of which MD is a subset.

I think MC barostat will come after, as that will be pretty easy to do as a stand alone. More straightforward multistep will come after as well. But the parts above are the ones that really need to be organized to see how it all goes together.

We'll see how this goes. A draft of the document will be posted this weekend, but getting it all into code will take a bit longer. Plus, I'm still working out all of the free energy issues + organizing the workshop in Sept. But if I get the parts listed above in, even roughly, then it should start making it much easier for other people to join in to get more of the work done.

#31 - 08/07/2013 04:35 AM - Michael Shirts

In terms of Langevin integrators: The framework here by Sivak, Chodera, and Crooks looks very appealing.

http://arxiv.org/pdf/1301.3800v2.pdf

It's very general, and reduces to standard integrators in a number of cases (Brownian, Bussi thermostat, velocity verlet).

I *believe* the Nose-Hoover approach can be plugged in trivially instead of the stochastic Ornstein-Uehlenbeck step.

#32 - 08/07/2013 04:48 AM - Michael Shirts

One general note:

I think a lot of the organization issue comes down to leapfrog vs. velocity Verlet. For simple cases, they are pretty much interchangeable. I would argue there are two main perceived advantages of leapfrog.

- 1/2 step velocities are less biased better than full step velocities as a function of time step. But this isn't actually a property of leapfrog. It's a property of where you collect statistics for the kinetic energies, and where in the integral you supply your temperature exchange. The right solution might be to implement AS velocity verlet, but with 1/2 step averaged kinetic energies.

- You condense the integration steps into fewer steps. However, as long as you are not collecting global statistics at at every half step, the speed cost is negligible. 2 extra order N routines are cheap. There are perhaps marginal roundoff error improvements, but given the errors inherent with 1-2 fs timesteps, these should be negligible.

- Fewer constraint steps In this case as well -- if you are not collecting full time step statistics, then you don't need to constraint the full time steps. So you can write a "leapfrog" version that only constraints every other step.

I'm leaving out a few details here. But the point is, I think that one can get most of the advantages that leapfrog provides while explicitly casting the integrator in a symmetric Trotter form that is organized more like velocity Verlet.

So I'd argue that to make it most general, we write the code in terms of a Trotter formalism, with explicit half velocity and full velocity steps, and place the data collection / constraints in a way that can get the "leapfrogish" advantages.

Please bring up anything I'm missing here before I jump in and start coding :)

#33 - 08/09/2013 02:18 AM - Michael Shirts

Being a bit more explicit on the last comment:

As I go through the code I'm making some notes. This is my basic understanding so far. I want to make clear the plans before I post them.

- Leapfrog and velocity Verlet are alternative ways to write a symmetric Trotter decompostion.

coupl(1/2) + vel(1/2) + pos(full) + vel(1/2) + coupl(1/2)

Where coupl performs all the coupling steps (pressure,temperature, etc.)

- There are two main reasons leapfrog as traditionally handled may have advantages.

  1. It explicitly calculated the kinetic energy from the half-steps, which is a lower bias estimate of the kinetic energy
  2. It allows constraint and coupling half step terms to be "collapsed," which can save communication.

These are the prime reasons. For NVT, when one collapses the terms, then one would like to do this:

vel(1/2) + coupl(1/2) + coupl(1/2) + vel(1/2) + pos(full)

However, for NVT methods, you are actually doing this:

coupl'(full) + vel(full) + pos(full)

in which some algebra is used to do the split step reversibly, so that the coupling step inside is equivalent to the coupling-prime step outside.

With pressure control, the algebra to move the coupling term outside is not quite exact, and there can be slight errors.

If this is correct, then the plan will be:

To write the code in terms of velocity Verlet, with the leapfrog essentially being velocity Verlet with 1) a 'doubled' coupling operator to save constraint and communication costs and 2) a half-step averaged KE.

When using the doubled coupling operators, velocities would be output as half steps; otherwise, they would be full step velocities.

#34 - 08/09/2013 02:57 AM - Mark Abraham

That all sounds great Michael, inasmuch as I still do not completely understand the whole of it!

My mental grand scheme is to move to a family of objects implementing integration strategy elements via Runner owning objects whose virtual methods resolve at run time to code that executes what the .tpr said should be happening (rather than horrible n-part conditionals and thousand-line functions of conditional nests that have to pass a dozen parameters if they dare call a function). That is somewhat orthogonal to this development, but it is much easier to do one before the other. So go right ahead. Anything that clarifies, comments, consolidates or corrects is fine by me!

#35 - 08/09/2013 03:13 AM - Peter Kasson

Agree here--I haven't gone through the integrator math but am following with interest. Comments from Berk etc. on how all this works with constraints would be most helpful I think.

#36 - 08/09/2013 06:08 AM - Michael Shirts

Peter Kasson wrote:

> Agree here--I haven't gone through the integrator math but am following with interest. Comments from Berk etc. on how all this works with constraints would be most helpful I think.

constraints are pretty much transparent to NVT scaling methods, regardless of how you do the bookkeeping. Since all temperature control methods scale velocities isotropically, whether you scale before or after constraints doesn't matter. Randomization methods (langevin or andersen) are a bit different, since you need to remove the component of the randomization that is parallel to the bonds). But this constraint step is cheaper, since the Lagrange multipliers for each bond can be used, i.e. noniterative. Sometimes you need to do some bookkeeping so you have the right components in the right place, and it will take a bit of work to make it clean, but not a basic problem.

Barostats are all horrible with constraints. There is no way to do barostats exactly right with constraints without iteration, which is nightmarish. You can get sort of close with some tricky math. Current PR is a case in point -- close to being right, but not quite.

A MC barostat (next task after rationalizing the basic integrator step) can address problem. MC barostats are not superefficient (cant' take large steps) but they don't care about constraints if set up correctly.

#37 - 08/09/2013 07:42 AM - Erik Lindahl

Not having to do a second communication step on the velocities is indeed one of the main reasons why we have liked leapfrog, and communication will likely be a worse bottleneck in the future, so that is an important feature to keep.

If we can get both that and the half-step kinetic energies (at least optionally) I think it would be great. The latter aren't discussed as much in the field (I still think Gromos has it wrong, for instance), but it can lead to strange issues where some properties are quite sensitive to your timestep, although the timestep being small enough for accurate integration!

#38 - 08/09/2013 01:56 PM - Michael Shirts

> and the half-step kinetic energies (at least optionally) I think it would be great. The latter aren't discussed as much in the field (I still think Gromos has it wrong, for instance), but it can lead to strange issues where some properties are quite sensitive to your timestep, although the timestep being small enough for accurate integration!

Erik, can you restate this? I think a word might have gone missing. Do you mean using the half-step kinetic energies results in some properties that are sensitive to timestep, or that not using them results in some properties that are sensitive to timestep? And you mean in NVT, correct? In NVE, the kinetic energy is the only property that will change - the trajectory will be exactly the same for leapfrog and vv (mod rounding error)

#39 - 08/09/2013 02:06 PM - Erik Lindahl

IIRC, our current approach of calculating velocity-derived properties at the velocity steps (n+0.5)*dt and *then* averaging them around each whole step provides more accurate estimates than first interpolating velocities corresponding to the position steps (n*t) and then using these to calculate e.g. temperature. I even think there might be something about this in a (very) old commit message from Berk...

#40 - 08/09/2013 04:35 PM - Michael Shirts

Erik Lindahl wrote:

> IIRC, our current approach of calculating velocity-derived properties at the velocity steps (n+0.5)*dt and *then* averaging them around each whole step provides more accurate estimates than first interpolating velocities corresponding to the position steps (n*t) and then using these to calculate e.g. temperature. I even think there might be something about this in a (very) old commit message from Berk...

For kinetic energies, then averaging the half step properties is definitely lower bias (though it's noiser). A few citations on this that I'll post eventually. However, were there other velocity dependent properties that this has been explored with?

#41 - 08/09/2013 04:38 PM - Michael Shirts

Also note that there are thermostats that require NO communication. For example, Langevin thermostats and Andersen thermostat require zero calculation of total kinetic energies -- temperature control is entirely local, not global. Bussi applied at a per-atom or per DOF level is also zero communication (though it would need to be tweaked to be efficient when doing per atom communication).

#42 - 08/11/2013 11:02 PM - Mark Abraham

On the Trotter decompositions, everybody I've read makes the arbitrary choice of splitting the operator that combines A and B in the following way:

exp(dt [A+B]) = exp(0.5 dt A) exp(dt B) exp(0.5 dt A) + O(dt^3).

Is the choice of one half just about minimizing the size of the O(dt^3) term? Put more concretely, that the integration phase of either x or v is more accurate if each is done with the half-step update of the other? So that using (say) 1/3 and 2/3 would be less effective?

#43 - 08/11/2013 11:36 PM - Michael Shirts

Mark Abraham wrote:

> On the Trotter decompositions, everybody I've read makes the arbitrary choice of splitting the operator that combines A and B in the following way:
>
> exp(dt [A+B]) = exp(0.5 dt A) exp(dt B) exp(0.5 dt A) + O(dt^3).
>
> Is the choice of one half just about minimizing the size of the O(dt^3) term? Put more concretely, that the integration phase of either x or v is more accurate if each is done with the half-step update of the other? So that using (say) 1/3 and 2/3 would be less effective?

My understanding is that the 1/2 + 1/2 is necessary to make it time symmetric, which is good for a number of reasons (physics are time-symmetric,

minimizes error, etc).

Doing the position operator as the inner operation vs. velocity operator is arbitrary, but would require two force evaluations in the inner loop, which isn't so good.

#44 - 08/12/2013 09:15 AM - Mark Abraham

Michael Shirts wrote:

> Mark Abraham wrote:
>
> > On the Trotter decompositions, everybody I've read makes the arbitrary choice of splitting the operator that combines A and B in the following way:
> >
> > exp(dt [A+B]) = exp(0.5 dt A) exp(dt B) exp(0.5 dt A) + O(dt^3).
> >
> > Is the choice of one half just about minimizing the size of the O(dt^3) term? Put more concretely, that the integration phase of either x or v is more accurate if each is done with the half-step update of the other? So that using (say) 1/3 and 2/3 would be less effective?
>
> My understanding is that the 1/2 + 1/2 is necessary to make it time symmetric, which is good for a number of reasons (physics are time-symmetric, minimizes error, etc).

Time symmetry does not seem to me to be the right explanation; the overall step should be symmetric in time (and the non-commutativity of the position and velocity updates makes that non-trivial), but one can do an x(1/3)v(1)x(2/3) in both time directions as readily as any other x splitting.

Having now read further, the issue of the choice of 1/2 was dealt with very early by Newmark (1959) https://engineering.purdue.edu/~ce573/Documents/Newmark_A%20Method%20of%20Computation%20for%20Structural%20Dynamics.pdf. Interpolation of (say) accelerations to update a velocity is a fundamental operation, and that is most accurately done when done evenly. It seems that doing anything else introduces some artificial damping. So using 1/2 is about minimizing error, but it is not the error in the splitting of the update operator that is being minimized.

Also, nobody needs to make the attempt to read the actual Trotter paper. It's on Banach algebras and not for the faint of heart. Fortunately, I had a friend visiting from Australia who **is** a Banach algebraist... The Trotter result only confirms that in the limit of an infinitessimal time step, the split propagation operator reduces to the unsplit operator. That's the least of our concerns!

> Doing the position operator as the inner operation vs. velocity operator is arbitrary, but would require two force evaluations in the inner loop, which isn't so good.

Sure

#45 - 08/19/2013 06:48 AM - Michael Shirts

I've posted a .tex file 'integrators5.tex' in the git manual repository (master branch) which outlines the integrator, thermostat, barostat and MC plans for 5.0. It is still VERY preliminary, and not entirely complete, but I wanted to get something up there.

Current todos

- The first step is to remove all iterative steps. A draft has already been completed in the master branch.
- The next step is to make integrators explicitly Trotter factorization, making leapfrog code branch ``doubling'' of Trotter steps, starting with the temperature control, which is generally straightforward. This process is underway.
- The third step is to combine and rationalize the pressure control integrators.
- The next step is to Monte Carlo-ize the integrator routine.

Ideally the second step will be completed by the GROMACS workshop, but since I've been running it, that may not happen.

#46 - 09/16/2013 02:19 AM - Michael Shirts

Just had a good discussion with Mark A. at the conference. We discussed particularly:

- the organization necessary to reorganize the loop so that we can better support multistep integrators and MC/MD. The basic format is then:

do from n=0 to nstep

```
do_force(x(t))

save the energy for next loop.

(accept/reject) (MD is a 100% acceptance, others are and based on new energy and previous energy )
            If reject, return to backup state

save trajectory and energy information
```

```
copy the state into a state_old

propose a new configuration (in MD, is N steps of MD integrator)
```

This ordering will support wide variety of functionality MD and MC functionality, including hybrid MC and an MC barostat.

2) hide a lot of the signaling and trajectory processing inside functions so md.c is clearer, and reorder it so that the integrator can be isolated to the same self-contained parts of the code.

#47 - 09/18/2013 12:53 AM - Mark Abraham

Agreed. Various clean-up patches have hit gerrit. I plan some wrapping of signalling stuff, too.

#48 - 05/13/2014 09:34 AM - Mark Abraham

- Subject changed from Proposal for integrator framework (do_md) in 5.0 to Proposal for integrator framework (do_md) in 5.1

- Target version changed from 5.0 to 5.x

#49 - 08/03/2015 10:13 PM - Mark Abraham

- Subject changed from Proposal for integrator framework (do_md) in 5.1 to Proposal for integrator framework (do_md) in future GROMACS

- Target version changed from 5.x to future

#50 - 08/03/2015 11:08 PM - Mark Abraham

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

#51 - 10/07/2015 09:33 PM - Gerrit Code Review Bot

Gerrit received a related patchset '1' for Issue #1137.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: Ibd70b5397568bfcd328cd6dd1c5c99384d7aaca8
Gerrit URL: https://gerrit.gromacs.org/5187

#52 - 11/27/2015 01:56 AM - Michael Shirts

Michael Shirts wrote:

First steps: I propose replacing the do_md and do_md_vv with multiple calls to do_velocity and do_coordinates. Eventually (soon), things like sd will also be replaced with do_velocity + do_coordinates + randomization step, nose-hoover a do_velocity + do_coordinates + scale coordinate step, etc. This is the first step in turning it entirely into a Trotter splitting framework.

Once consequence of this is that instead of going through one all-N-particles loop each time integration occurs, it would go through several. This should not be too much overhead, especially since it's all completely parallelized.

Just wanted to check to make sure people were on board with this concept before I start the pull request.

#53 - 11/27/2015 07:51 AM - Erik Lindahl

There's been a history of lots of complex bugs with the update changes, so with any new change the first task should likely be to turn the update parts that change into C++ classes with very short and well-documented methods (~10-15 lines).

#54 - 11/27/2015 03:44 PM - Michael Shirts

Erik Lindahl wrote:

> There's been a history of lots of complex bugs with the update changes, so with any new change the first task should likely be to turn the update parts that change into C++ classes with very short and well-documented methods (~10-15 lines).

Which means we need to agree a bit more before coding (and why I'm asking before charging ahead at this point). Should "state" be a C++ class with methods do_velocity and do_coordinate update? It seems somewhat natural, but the determination of what exactly the class should be is complex here.

#55 - 11/27/2015 03:53 PM - David van der Spoel

It would be great to have a virtual base class that does the integration that can be inherited by a number of parallel implementations. Separate the math from the physics.

#56 - 11/27/2015 04:53 PM - Michael Shirts

David van der Spoel wrote:

> It would be great to have a virtual base class that does the integration that can be inherited by a number of parallel implementations. Separate the math from the physics.

Can you be a bit more specific as to how this would be designed? I know all the integrator math and algorithms very well -- my issue is that my C++ is somewhat weak, so I'm not good at figuring out beforehand how exactly design the class.

I would envision having some sort of class with the current state of the molecule, then having the different components of the integrators (coordinate update, velocity update, box update, alchemical variable update) be methods of the class. I'm happy to rewrite code as methods of this class, though I would want to make sure the class itself is right before plowing forward.

Talking with Mark earlier, he seemed to think that it might be better to refactor first, and then classify after, but I'm open to suggestions.

#57 - 11/27/2015 05:14 PM - David van der Spoel

I haven't given this a whole lot of thought, but the idea would be to have the dynamics be the driver of whatever we do in mdrun.

```
class DynamicsEngine {
DynamicsEngine() {}
virtual ~DynamicsEngine() = 0;
virtual initEngine() = 0;
virtual calculateForces() = 0;
virtual updateVelocities() = 0;
virtual updateCoordinates() = 0;
virtual finalizeEngine() = 0;
}
```

The main routine would just initiate a variable of this class and run it.

Maybe others have more clear ideas and would like to break it down further.

#58 - 11/27/2015 05:43 PM - Teemu Murtola

That proposal by itself doesn't add much in terms of modularity, and seems to mix different concepts. I think that the top-level integrator object probably should only have a runSimulation() method (or possibly runSingleStep()). And that object knows nothing about how to evaluate the forces; it just has a reference to another class that does that.

PS. What you propose is called an interface, not a virtual base class. The latter is an complicated construct that we probably never want to use...

#59 - 11/27/2015 06:05 PM - Michael Shirts

I think there are two different philosophies, and I have no clue which the better one is (again, if I say something that doesn't make sense for the C++ version of object orientation, please alert men).

One is to treat the algorithm as a class, and the other is to have the data as a class, which has different algorithm methods that operate on it.

One thing I think is particularly important for modularity in the trotter formalism; all integrators are sequences of the same

Velocity verlet is: half step velocity, full step coordinates, half step velocity - output
leapfrog is: full step coordinates, half step velocity, half step velocity - output.

Nose-hoover temperature control schemes involve decompositions like:
half step velocity, half-step T-coupling, full step coordinates, half-step T-coupling, half step velocity - output

And so forth.

So perhaps there could be a base class, integrate_coordinates, with all of the individual elements in the integration factorization defined, and then each integrator is an inherited class that calls the elements in the proper order?

I was actually thinking (in the procedure orientation) is that each integrator is simply a label for an array of element function names, and the integrator routine takes that array and execute the functions listed in the array -- but I think that could still be done, but having a method that executes an arbitrary array of the derived methods. Writing a new method would be as simple as making sure all of the elements were defined, and specifying an ordering. There are likely to be some complications (for example, specifying where communication may need to be done).

Though I think some error checking could be built in by having each method check certain booleans, such as whether the pressure is defined for a current state, whether the kinetic energy is defined for the current state, etc, that would get reset upon incrementing the state, but I'll have to think a little about that.

Though that still leaves me wondering how this relates to the coordinate and velocities; do we make the base integrator function a method of the class of the state?

#60 - 11/27/2015 06:13 PM - Teemu Murtola

I think that it will be much clearer if the state is just that, and does not provide any complicated logic for updating itself. Because if you have multiple ways to update the data (e.g., multiple integrators, EM, MC, ...), it doesn't really make sense that each of them is a different subclass of the state...

The kind of arrays of function pointers you mention is exactly what an C++ interface provides you, with much nicer syntax and compile-time checking. But I think the essential piece at this point is to decide the high-level structure and desired responsibilities of the different parts, not the exact C++ constructs that it will use.

#61 - 12/01/2015 06:38 PM - Michael Shirts

Teemu Murtola wrote:

> I think that it will be much clearer if the state is just that, and does not provide any complicated logic for updating itself. Because if you have multiple ways to update the data (e.g., multiple integrators, EM, MC, ...), it doesn't really make sense that each of them is a different subclass of the state...

That's fine with me to not have the state a class with integrators for the methods. I'm just trying to figure out the organization.

> The kind of arrays of function pointers you mention is exactly what an C++ interface provides you, with much nicer syntax and compile-time checking.

Can you point to an example?

> But I think the essential piece at this point is to decide the high-level structure and desired responsibilities of the different parts, not the exact C++ constructs that it will use.

HIGHEST level in a MC/MD framework, there are a set of state variables:

- coordinates
- momentum
- box vectors
- state variables that might with some algorithms be dynamic (pressure, temperature, alchemical energy parameters)

- And we iteratively change some or all of the above, based on information about the current energy and forces. There are a whole lot of ways to update them, but they can mostly (always) can be decomposed into the same, much smaller set of elementary operations applied repeatedly in different orders. Complications involve properly communicating information between processors and making sure that output is done in the correct place.

So, after phrasing it that way, what additional algorithmic structure is needed to start making specific choices about the C++ constructs?

In the short term, I could start writing the elementary routines as standard C if that would make the conversion simpler . . . but I want to get all the good information first.

#62 - 12/01/2015 07:30 PM - David van der Spoel

Browsing over the whole discussion again, I think we are not ready for coding yet, and moreover coding in C will be barely helpful, rather create double work.

It would be great to have that flowchart that we discussed few years ago, then based on that we have to design an interface matching the flowchart. Then we can start coding.

#63 - 12/01/2015 08:11 PM - Michael Shirts

> It would be great to have that flowchart that we discussed few years ago, then based on that we have to design an interface matching the flowchart. Then we can start coding.

Just so we're on the same page, can you link to the sort of flowchart you'd like to see (from any software you can find)?

#64 - 12/01/2015 08:25 PM - David van der Spoel

We have a very simple one in the manual, fig 3.3, but that does not have sufficient detail. Can't find a better illustration right away though.

#65 - 12/08/2015 06:47 PM - Michael Shirts

- File integrator.tex added

- File integrator.pdf added

OK, I've attached a first draft of a proposal for the organization (both .pdf and .tex). It's not quite a flowchart, since the main loop is pretty simple; the interesting stuff comes by swapping in and out different modular parts of the main loop. It's definitely more algorithmically focused that exactly what data structures to use -- because I know the algorithms and am not sure exactly what data structures to use. It is not superdetailed yet (Don't have Parrinello-Rahman dynamics in yet) as I wanted to get some more feedback before moving forward further.

Please hit at this hard so I can iterate this and make it something people can be satisfied with.

#66 - 12/09/2015 03:56 PM - Michael Shirts

Any thoughts, when you get a chance.

#67 - 12/11/2015 12:02 PM - Teemu Murtola

That kind of structure would probably already improve things a lot. A few high-level comments, the first partially about the math, the others more about software organization:

- It would be nice that also the other parts (not just do_md()) could use the same structure. So also energy minimization, TPI etc. It's probably not too difficult, but should be kept in mind during the design phase.
- It would be nice that the code would be very clear about which elementary steps modify which parts of the state and the collectives. In many cases, all of the state, or all of the collectives, are actually constant during some of the steps, and it would be best if this was visible from the code structure without analyzing all the code, and that the compiler could enforce this through proper use of const etc.
- Related to the above, but on a more fine-grained level: there will be parts of the state and collectives that are specific to certain elementary steps. For example, the piston coordinates/momenta for N-H would ideally not be visible to other steps, again making it clear from the code that they are not modified elsewhere. Also, from a modularity point of view, it would be nice to be able to add new types of thermostats etc. without changing the data structure used for the state and/or collectives. In other words, the state and collectives should be somehow extensible.

#68 - 12/11/2015 08:12 PM - Michael Shirts

Thanks for getting back with comments! With regards to some of my comments here - I'm not a C++ expert, so you might need to point me to some actual examples of some code somewhere if there is some specific C++ way to do some of these things.

> That kind of structure would probably already improve things a lot. A few high-level comments, the first partially about the math, the others more about software organization:
>
> - It would be nice that also the other parts (not just do_md()) could use the same structure. So also energy minimization, TPI etc. It's probably not too difficult, but should be kept in mind during the design phase.

it should be relatively easy to put most existing functionality like energy minimization in, at least itself as an entire element -- breaking it up into separate elements might take a bit longer, but could be done at leisure. TPI is interesting, since it's essentially a bunch of rejected MC moves + bookkeeping. I'll have to think about the way to break it into elements.

> - It would be nice that the code would be very clear about which elementary steps modify which parts of the state and the collectives. In many cases, all of the state, or all of the collectives, are actually constant during some of the steps, and it would be best if this was visible from the code structure without analyzing all the code, and that the compiler could enforce this through proper use of const etc.

That's a great idea. How would you imagine this actually carried out in code? Clearly, one can state in comments at the beginning "This element changes only X,Y,Z and leaves other things constant). In terms of the interface -- is it best to make function interfaces explicit, and rather than passing in the whole state, have functions called like:

```
integrate_coordinates(state->x, state->v, otherstuff, with the function definition
```

With function definition

```
integrate_coordinates(x, const v, const otherstuff) {
```

Or are there other formalisms to do this?

> - Related to the above, but on a more fine-grained level: there will be parts of the state and collectives that are specific to certain elementary steps. For example, the piston coordinates/momenta for N-H would ideally not be visible to other steps, again making it clear from the code that they are not modified elsewhere.

So again, would you imagine something like:

```
integrate_piston(box_vector, const box_velocity, otherstuff)
```

Called as

```
integrate_piston(state->box_vector, const state->box_velocity, otherstuff)
```

Or are you referring to some other way of calling it?

To some extent, it would be nice to just have all of the elements take the state as an argument, and then the details could be worked out inside of the integrator, but I'll defer to people with more programming expertise on this.

> Also, from a modularity point of view, it would be nice to be able to add new types of thermostats etc. without changing the data structure used for the state and/or collectives. In other words, the state and collectives should be somehow extensible.

Is it required that the data structure not change? It seems to me to be pretty extensible to just add a new variable in a data structure; that doesn't stop any of the other functionality from working. What pseudocode would you imagine?

#69 - 12/13/2015 11:50 AM - Teemu Murtola

Michael Shirts wrote:

> - It would be nice that the code would be very clear about which elementary steps modify which parts of the state and the collectives. In many cases, all of the state, or all of the collectives, are actually constant during some of the steps, and it would be best if this was visible from the code structure without analyzing all the code, and that the compiler could enforce this through proper use of const etc.

That's a great idea. How would you imagine this actually carried out in code? Clearly, one can state in comments at the beginning "This element changes only X,Y,Z and leaves other things constant). In terms of the interface -- is it best to make function interfaces explicit, and rather than passing in the whole state, have functions called like:

integrate_coordinates(state->x, state->v, otherstuff)

This will not work, if you want to have the level of generality that you have in your proposal. At the highest level, all the elementary steps need to have the same function signature. It could be possible to put an enum or some other dynamic information that allows selecting the signature from a few different alternatives, but that probably just leads into more mess than it solves.

To some extent, it would be nice to just have all of the elements take the state as an argument, and then the details could be worked out inside of the integrator, but I'll defer to people with more programming expertise on this.

I think this is by far the simplest approach, at least on the highest level. But at some level it should be made const, and or broken into smaller pieces so that all the data is not exposed to all the code.

> Also, from a modularity point of view, it would be nice to be able to add new types of thermostats etc. without changing the data structure used for the state and/or collectives. In other words, the state and collectives should be somehow extensible.

Is it required that the data structure not change? It seems to me to be pretty extensible to just add a new variable in a data structure; that doesn't stop any of the other functionality from working.

There are three potential issues with this:

- This does not provide any guarantees about encapsulation. To know for certain that a piece of state belongs to a particular part of code, one then needs to read through all code that may be accessing the state.
- This easily leads to the code structure that we currently have that is difficult for newcomers to extend: to add stuff to the state, you also need to know half a dozen other places that are accessing the state and add code in all of those places. It would be much nicer that to add new functionality, the compiler would force you to implement all the necessary functionality (by forcing you to implement a bunch of interface methods), and you could implement all of that in a single place.
- With this approach, it is not possible to use Gromacs as an extensible library, if any new functionality will need changing the base data structures.

#70 - 12/13/2015 01:38 PM - Erik Lindahl

I'll need to think more about this over X-mas, but a few quick comments:

- While I have no particular love for Nosé-Hoover chains, whenever we make decisions that "X will be impossible to support", that is usually an indication we'll also find several other things that are impossible to implement in the future - and some of those might be important. Thus, even if we decide not to do it, we should design a framework where it is possible to.

- Unfortunately the assumption that it won't cost anything extra to split the update in two calls won't work. As we're pushing more things on accelerators we'll soon be in the situation where integration is the performance-critical step on the CPU. At least for the default integrator pathway we can't afford to split it in multiple steps. However, it might be possible to do something fancy in C++ where the compiler achieves this for you...

- Have a look at the discussions we've had about task parallelization. To make this work we shouldn't separate calculation and communication as different types of tasks, but rather have them as similar operations with dependencies on each other (and let a scheduler decide what gets done in what order)

Finally, once we have a finished design I would strongly recommend introducing many small gradual changes rather than trying to do everything in one go!

#71 - 12/13/2015 05:19 PM - Michael Shirts

> integrate_coordinates(state->x, state->v, otherstuff)

> This will not work, if you want to have the level of generality that you have in your proposal.

OK, good, that rules out that. I have no love for that way of writing the functions, I'm just trying to figure out the code that is consistent.

> At the highest level, all the elementary steps need to have the same function signature.

OK, good, moving towards a specific way to code it.

> It could be possible to put an enum or some other dynamic information that allows selecting the signature from a few different alternatives, but that probably just leads into more mess than it solves.

Seems reasonable.

> > To some extent, it would be nice to just have all of the elements take the state as an argument, and then the details could be worked out inside of the integrator, but I'll defer to people with more programming expertise on this.
>
> I think this is by far the simplest approach, at least on the highest level. But at some level it should be made const, and or broken into smaller pieces so that all the data is not exposed to all the code.

So this is the crux, because parts of the state will be changed for each function, but different parts of the state will be changed for different functions. In fact, each elementary function will likely update a different part of a state variable (some might update a two or three for various reasons). And the generally classification of a something that one would put into a state variable is that it will change at some point in the simulation. Any thoughts on this? I don't see a way to do it without exposing all the data to all of the elementary functions. It seems (from my naive view) that there is not a way to both 1) have the same function signature and 2) not expose all data to all parts of the code.

I think one thing that should be certain is that all of the changes to the state should be within the elementary integrators, and nowhere else.

> > Is it required that the data structure not change? It seems to me to be pretty extensible to just add a new variable in a data structure; that doesn't stop any of the other functionality from working.
>
> There are three potential issues with this:
>
> - This does not provide any guarantees about encapsulation. To know for certain that a piece of state belongs to a particular part of code, one then needs to read through all code that may be accessing the state.
> - This easily leads to the code structure that we currently have that is difficult for newcomers to extend: to add stuff to the state, you also need to know half a dozen other places that are accessing the state and add code in all of those places.
> - With this approach, it is not possible to use Gromacs as an extensible library, if any new functionality will need changing the base data structures.

OK, good, this is the information that I needed to understand. Though my limited mastery of C++ means I'm not sure what the solution will be. . .

> It would be much nicer that to add new functionality, the compiler would force you to implement all the necessary functionality (by forcing you to implement a bunch of interface methods), and you could implement all of that in a single place.

I agree that this would be important to do. Talking of encapsulation, it seems like the correct object-oriented way to do this would be to make all of the elementary functions methods of the state class, so that's the only place that changes to the state go.

I could imagine some meta-code (run at compile time? runs independently) that reports on what exactly the sequence of elementary integrator steps would modify in sequence for human readability.

#72 - 12/13/2015 05:38 PM - Michael Shirts

Erik Lindahl wrote:

> I'll need to think more about this over X-mas,

Sounds good. The couple of weeks after X-mas would be when I would have the most time to push the main chunk out.
but a few quick comments:

- While I have no particular love for Nosé-Hoover chains, whenever we make decisions that "X will be impossible to support", that is usually an indication we'll also find several other things that are impossible to implement in the future - and some of those might be important. Thus, even if we decide not to do it, we should design a framework where it is possible to.

The main reason for removing Nose-Hoover chains are that they are mathematically problematic. One counts on the system being chaotic over all

degrees of freedom (i.e. no orbits anywhere). Nose-Hoover chains definitely fail for harmonic oscillators, and it's not clear which systems they also fail for. There are energies that are provably not visited for any system with a Nose-Hoover chains thermostat. At most temperatures, for all liquids, they appear to have to low a contribution to the thermo to matter -- but it could be that it does make a difference with, say, crystals. Much better to take the Nose-Hoover bath variable and explicitly randomize it by using Langevin dynamics (Nose-Hoover-Langevin of Leimkuler). Then you get both the gentle thermostat + provably correct randomization.

However, Nose-Hoover chains WOULD be supported in this formalism -- it would just require a lot of extra elementary functions and state variables to be defined. In fact, the velocity verlet nose hoover chains uses something like this elementary function process already.

- Unfortunately the assumption that it won't cost anything extra to split the update in two calls won't work. As we're pushing more things on accelerators we'll soon be in the situation where integration is the performance-critical step on the CPU. At least for the default integrator pathway we can't afford to split it in multiple steps. However, it might be possible to do something fancy in C++ where the compiler achieves this for you...

It IS very easy to have doubled steps, so that you don't have to take 2 consecutive velocity steps, but can take one step of length 2dt. It would be great to test the assertion of the integration being the time limiting factor explicitly, because once you join x and v integration, then you destroy a lot of the advantages, since you can't stick things in between position/velocity updates, without explicitly creating a lot of new functions. And certainly all of the elementary integrations steps are fully parallel -- they operate only on the local number of particles. It seems odd to me that communication will not be the limiting factor rather than integration, unless you are referring to 1 GPU + 1 CPU systems?

It's certainly is possible that it one could create merged integrators for particular code paths in parallel with the fully modular system, but those should be minimized (and there can be tests to checking to make sure that they do exactly what they are supposed to by direct comparison with the unmerged integrators.

- Have a look at the discussions we've had about task parallelization. To make this work we shouldn't separate calculation and communication as different types of tasks, but rather have them as similar operations with dependencies on each other (and let a scheduler decide what gets done in what order)

OK, I will do that! Usually, it's always safe to assume Energy/Force/Virial calculation should be done immediately after the coordinate update, and kinetic energy after the velocity update, which might guide the discussion.

> Finally, once we have a finished design I would strongly recommend introducing many small gradual changes rather than trying to do everything in one go!

If we are creating a new type of class for the state, then the most gradual change would probably be writing the elementary functions in C (compatible with the current code) and then C++ ifying them once the framework is in. This can be done now. It's extra work, but it's all gradual work.

#73 - 12/13/2015 09:41 PM - Teemu Murtola

Michael Shirts wrote:

> > integrate_coordinates(state->x, state->v, otherstuff)
>
> This will not work, if you want to have the level of generality that you have in your proposal.

OK, good, that rules out that. I have no love for that way of writing the functions, I'm just trying to figure out the code that is consistent.

> At the highest level, all the elementary steps need to have the same function signature.

OK, good, moving towards a specific way to code it.

Actually, I probably need to take back my original statement. It is true if one wants to write the code really like in your pseudocode, with a loop over generic elementary steps. But I'm not really sure whether that adds any value; the alternative is just writing out the list of elementary steps in code as a list of function calls, with separate code for each integrator. The latter is simpler, both to write and to understand, and it provides much more flexibility in terms of what the elementary steps can take as input and output.

Talking of Erik's comment on task parallelism, it should also be relatively straightforward to do a transformation that instead of the elementary steps doing the computation directly, that they would return the tasks that will do the computation, set up with the relevant dependencies. This will also be easier if the different steps can take different inputs, since in this case they probably need to get some kind of references to their prerequisite tasks, and the types of these tasks depend on the step.

> So this is the crux, because parts of the state will be changed for each function, but different parts of the state will be changed for different functions. In fact, each elementary function will likely update a different part of a state variable (some might update a two or three for various reasons). And the generally classification of a something that one would put into a state variable is that it will change at some point in the simulation. Any thoughts on this? I don't see a way to do it without exposing all the data to all of the elementary functions. It seems (from my naive view) that there is not a way to both 1) have the same function signature and 2) not expose all data to all parts of the code.

It's still possible to limit the exposure to the few highest levels. And some of the data is intrinsically shared; it does not make much sense to try to hide, e.g., the particle positions from any of the code, but it can add value to know that they will never be modified below a certain level (i.e., that they get passed down as const). For data that is really private to a certain elementary step, it should also be possible to make it an opaque part of the state such that although it is technically there, it is not possible to access it from anywhere else.

> There are three potential issues with this:
>
> - This does not provide any guarantees about encapsulation. To know for certain that a piece of state belongs to a particular part of code, one then needs to read through all code that may be accessing the state.
> - This easily leads to the code structure that we currently have that is difficult for newcomers to extend: to add stuff to the state, you also need to know half a dozen other places that are accessing the state and add code in all of those places.
> - With this approach, it is not possible to use Gromacs as an extensible library, if any new functionality will need changing the base data structures.

OK, good, this is the information that I needed to understand. Though my limited mastery of C++ means I'm not sure what the solution will be. . .

> It would be much nicer that to add new functionality, the compiler would force you to implement all the necessary functionality (by forcing you to implement a bunch of interface methods), and you could implement all of that in a single place.

I agree that this would be important to do. Talking of encapsulation, it seems like the correct object-oriented way to do this would be to make all of the elementary functions methods of the state class, so that's the only place that changes to the state go.

I still think this is not the way to go. If only methods of the state class are allowed to modify it, ~everything in the md loop will need to be a member of the state. And again, this means that it is not possible to add new functionality without modifying existing classes.

I don't have a full, ready-thought solution to this, but I have several alternative ideas. We probably need to try out a few different alternatives with some concrete cases to see which one of them strikes the best balance between complexity, readability, and encapsulation. But this does not need to be the first thing that happens.

#74 - 12/14/2015 08:22 AM - Michael Shirts

> Actually, I probably need to take back my original statement. It is true if one wants to write the code really like in your pseudocode, with a loop over generic elementary steps. But I'm not really sure whether that adds any value; the alternative is just writing out the list of elementary steps in code as a list of function calls, with separate code for each integrator. The latter is simpler, both to write and to understand, and it provides much more flexibility in terms of what the elementary steps can take as input and output.

I'm fine with this, as long as each integrator is, to the extent possible, elementary integrator steps. It forces people to understand what is being done. Then, each elementary step would not take the state, just the things of the state that it needs, and would take a const the other things? Or what are you imagining?

> Talking of Erik's comment on task parallelism, it should also be relatively straightforward to do a transformation that instead of the elementary steps doing the computation directly, that they would return the tasks that will do the computation, set up with the relevant dependencies. This will also be easier if the different steps can take different inputs, since in this case they probably need to get some kind of references to their prerequisite tasks, and the types of these tasks depend on the step.

OK, I'll have to think about this -- can you point me to good existing examples in the code that describes how to return tasks and dependencies? If this is a todo in the code, then I might have to try it the old ways first . . .

> It's still possible to limit the exposure to the few highest levels. And some of the data is intrinsically shared; it does not make much sense to try to hide, e.g., the particle positions from any of the code, but it can add value to know that they will never be modified below a certain level (i.e., that they get passed down as const). For data that is really private to a certain elementary step, it should also be possible to make it an opaque part of the state such that although it is technically there, it is not possible to access it from anywhere else.

OK, those all make sense, though it's not clear exactly what would be used where to me yet (since this discussion is pretty abstract).

> There are three potential issues with this:
>
> - This does not provide any guarantees about encapsulation. To know for certain that a piece of state belongs to a particular part of code, one then needs to read through all code that may be accessing the state.
> - This easily leads to the code structure that we currently have that is difficult for newcomers to extend: to add stuff to the state, you also need to know half a dozen other places that are accessing the state and add code in all of those places.
> - With this approach, it is not possible to use Gromacs as an extensible library, if any new functionality will need changing the base data structures.

> I still think this is not the way to go. If only methods of the state class are allowed to modify it, ~everything in the md loop will need to be a member of the state.

But only the integrators would need to modify it, not everything in the MD loop? Or I'm missing something.

> And again, this means that it is not possible to add new functionality without modifying existing classes.

I guess I don't see how to do encapsulation without requiring modifying existing classes to add functionality.

> I don't have a full, ready-thought solution to this, but I have several alternative ideas. We probably need to try out a few different alternatives with some concrete cases to see which one of them strikes the best balance between complexity, readability, and encapsulation. But this does not need to be the first thing that happens.

Well, Erik is insisting on designing it before we try it (which is not a bad idea at all) but does preclude trying a bunch of different alternatives :) So I'm caught with not quite knowing what steps to take other than keep this discussion up until we can get everyone's requirements articulated and a good path forward starts coalescing.

#75 - 12/17/2015 12:47 PM - Teemu Murtola

Michael Shirts wrote:

> OK, I'll have to think about this -- can you point me to good existing examples in the code that describes how to return tasks and dependencies? If this is a todo in the code, then I might have to try it the old ways first . . .

People have not even decided on the tasking framework to use, so there can't really be any examples...

> > I still think this is not the way to go. If only methods of the state class are allowed to modify it, ~everything in the md loop will need to be a member of the state.
>
> But only the integrators would need to modify it, not everything in the MD loop? Or I'm missing something.

If you think of it that way, then the state should be a private member of the integrator, and no code except the integrator should ever see the full state. But then you need some simpler state data structure that you can actually pass into methods that need to operate on the data outside the integrator. This simpler data structure can of course also be on the level of coordinate arrays etc. that all get passed down individually, but that can also become tedious when more and more things accumulate into the state with features added.

> > And again, this means that it is not possible to add new functionality without modifying existing classes.
>
> I guess I don't see how to do encapsulation without requiring modifying existing classes to add functionality.

This is a basic design principle in object-oriented programming (called the open/closed principle). https://gerrit.gromacs.org/5372 will hopefully evolve into something that demonstrates at least the basic principles of how to achieve that (although the state is not involved there; that is mostly about t_inputrec). But you may need to wait some time.

> > I don't have a full, ready-thought solution to this, but I have several alternative ideas. We probably need to try out a few different alternatives with some concrete cases to see which one of them strikes the best balance between complexity, readability, and encapsulation. But this does not need to be the first thing that happens.
>
> Well, Erik is insisting on designing it before we try it (which is not a bad idea at all) but does preclude trying a bunch of different alternatives :) So I'm caught with not quite knowing what steps to take other than keep this discussion up until we can get everyone's requirements articulated and a good path forward starts coalescing.

It is not realistic to design something like this down to the last detail, without actually implementing it. You will always miss some detail that you will only come across when you write the code. And with people not very experienced with C++, arriving at the final class design without writing any code is even more impossible. I think we should at most agree on the general principles, and then refactor the code to be close enough to this structure that it is possible to identify what exactly needs to be done to support all the existing paths. And introduce C++ gradually, when we find useful abstractions that can be put in place without a full rewrite of large parts of the code. I could take https://gerrit.gromacs.org/5449 as a concrete example of the latter, although even that already is a bit larger than I would like for a single change.

As a concrete path forward, I would suggest that we refactor the existing C code in small steps towards a structure where we can write the top-level MD loop using this formalism (I think Mark is already extracting rerun functionality into a separate function). Then we can start introducing C++ for some parts (hopefully, the group kernels are really gone by that time, so that we can actually have C++ code throughout). But, e.g., this encapsulation can be put in later, as a separate set of refactorings.

#76 - 12/17/2015 06:00 PM - Michael Shirts

> People have not even decided on the tasking framework to use, so there can't really be any examples...

OK, the integrator improvements should not necessarily have this as a roadblock.

> This is a basic design principle in object-oriented programming (called the open/closed principle). https://gerrit.gromacs.org/5372 will hopefully evolve into something that demonstrates at least the basic principles of how to achieve that (although the state is not involved there; that is mostly about t_inputrec). But you may need to wait some time.

Ah, I see - the parent object is never modified, but classes that inherit from it can be. Hmm. We don't want to have a chain of inherited integrator classes, such that each time new functionality is added, you just inherit from the previous enlarged integrator, but one certainly can have partial shielding by making some parts of the state non-modifiable, and having additions be made only on an inherited class. That might help a bit.

> It is not realistic to design something like this down to the last detail, without actually implementing it. You will always miss some detail that you will only come across when you write the code. And with people not very experienced with C++, arriving at the final class design without writing any code is even more impossible. I think we should at most agree on the general principles, and then refactor the code to be close enough to this structure that it is possible to identify what exactly needs to be done to support all the existing paths. And introduce C++ gradually, when we find useful abstractions that can be put in place without a full rewrite of large parts of the code. I could take https://gerrit.gromacs.org/5449 as a concrete example of the latter, although even that already is a bit larger than I would like for a single change.

This seems reasonable.

> As a concrete path forward, I would suggest that we refactor the existing C code in small steps towards a structure where we can write the top-level MD loop using this formalism (I think Mark is already extracting rerun functionality into a separate function). Then we can start introducing C++ for some parts (hopefully, the group kernels are really gone by that time, so that we can actually have C++ code throughout). But, e.g., this encapsulation can be put in later, as a separate set of refactorings.

Then the likely thing to do would be to start rewriting parts of the integrator as small integrator elements so people can see how they would fit together. Then once they are in place, we can think about generalizing it. This is something I can start now in a branch.

#77 - 12/18/2015 12:26 PM - Teemu Murtola

Michael Shirts wrote:

> > This is a basic design principle in object-oriented programming (called the open/closed principle). https://gerrit.gromacs.org/5372 will hopefully evolve into something that demonstrates at least the basic principles of how to achieve that (although the state is not involved there; that is mostly about t_inputrec). But you may need to wait some time.
>
> Ah, I see - the parent object is never modified, but classes that inherit from it can be. Hmm. We don't want to have a chain of inherited integrator classes, such that each time new functionality is added, you just inherit from the previous enlarged integrator, but one certainly can have partial shielding by making some parts of the state non-modifiable, and having additions be made only on an inherited class. That might help a bit.

I still think that inheritance in this form is not going to solve this. Another basic design principle is that inheritance should describe an "is a" relationship, and there isn't really any sensible definition that would lead to "integrator is a state" or "elementary step is a state" being true. And without very messy virtual and multiple inheritance, you cannot create combinations that would have the properties of multiple derived classes. And it is not possible at all to create such combinations at runtime instead of compile time. Another, perhaps slightly more modern principle is that the main reason for inheritance should not be reusing code from the base class; instead, it should be driven by code outside the class being able to only reference the base class, and thus that code outside the class being reusable with different kinds of derived classes.

At this point, however, I think the essential question should be the data ownership and visibility (i.e., which parts of the code can see/modify what data), and not exactly how that is achieved in C++.

#78 - 12/20/2015 04:03 AM - Mark Abraham

I sketched some code based upon the discussion, found at http://pastebin.com/UmMzE6du. Comments most welcome, hopefully we can comment there, but I haven't tried yet. More to say when I have more time!

#79 - 12/20/2015 09:15 AM - Mark Abraham

Same content is also at https://gist.github.com/mabraham/5db12701291e3f5e8754, though I don't think that allows per-line commentary either. But at least you can comment on the lot, or git clone it and rework something.

A key part of the plan needs to consider change management. We can't disable the former md or md-vv integrators until they have a first-rate replacement available and tested. We might not finish even the early stages of the plan before GROMACS 2016 will ship, and the old functionality must remain in a state of release quality. Thus, my sketch includes a GenericIntegrator that is approximately the contents of do_md right now.

I suggest we make an exception that we should remove the do_md parts of md-vv-avek (inserting a fatal error) on the grounds that probably nobody

is using that integrator (it comes with some serious limitations), and it is a notable simplification of do_md(). I expect we can replace it in a simple way in the new formalism, ideally before GROMACS 2016.

Also, if we agree to deprecate e.g. old-school N-H in favour of new equivalents, then we should do that at some point before we release GROMACS 2016.

#80 - 12/22/2015 01:21 PM - Mark Abraham

I spent some time today writing some tests that restarts are exact, starting with an argon box. The results were displeasing, and are reproducible back to release-4-6 HEAD. I'll share some stuff tomorrow-ish when I've looked at it with fresh eyes.

Leap-frog restarts are exact on everything I tried. md-vv restarts are

- exact for NVE, and N-H NVT
- subtly wrong for N-H with MTTK and Berendsen barostat with no thermostat
- badly wrong on the restart+1 step KE for everything else that runs (i.e. Berendsen or v-rescale thermostat with either no or Berendsen barostat)

The md-vv combinations that work are the easiest to re-implement in the new scheme, the ones subtly wrong are of niche value, and the ones badly wrong are worthy of turning into a separate bug report. (And all this is before any thoughts of testing exact restarts with constraints or free-energy calculations.)

So, I think there is a serious case for purging do_md, and routines that calculate globals, energies, pressure, temperature, etc. of anything to do with any of the velocity-Verlet integrators (including the ekinf vs ekinh stuff). We can keep the orphaned low-level VV-specific update routines temporarily, since we anticipate being able to use some of that. I know I'm a bit trigger-happy in wanting to get rid of doubtful code, but here

- it is clear that the original implementation is still deficient in essential qualities years later,
- that implementation greatly complicates the implementation of core functionality that is working correctly,
- preserving the old implementation of VV will not give us much of value moving forward with a new implementation,
- if we remove the implementation of VV, the simplified simulation routines that result will be easier to form into building blocks we can use to re-implement VV integrators that clearly work correctly.

#81 - 12/22/2015 06:53 PM - Michael Shirts

> I spent some time today writing some tests that restarts are exact, starting with an argon box. The results were displeasing, and are reproducible back to release-4-6 HEAD. I'll share some stuff tomorrow-ish when I've looked at it with fresh eyes.

Writing tests is a great idea.

> Leap-frog restarts are exact on everything I tried. md-vv restarts are
>
> - exact for NVE, and N-H NVT
> - subtly wrong for N-H with MTTK and Berendsen barostat with no thermostat
> - badly wrong on the restart+1 step KE for everything else that runs (i.e. Berendsen or v-rescale thermostat with either no or Berendsen barostat)

I'm not surprised that there were KE problems -- that was 1) the hardest thing to get right while implementing VV without changing the leapfrog code path and 2) the sort of thing that are easy to break if people ignore a branch, which I think had a tendency to happen with md-vv. If we want to scale development up with number of developers, then having a broader testing suite will help immensely so that these things are caught earlier and less-used functionality doesn't hang around untested.

> The md-vv combinations that work are the easiest to re-implement in the new scheme, the ones subtly wrong are of niche value, and the ones badly wrong are worthy of turning into a separate bug report. (And all this is before any thoughts of testing exact restarts with constraints or free-energy calculations.)

I have some (not generalizable to automated testing currently) restart tests, and I can try looking at the current state to see if it's a straightforward fix. It will be easier to compare a new organization with a correct implementation rather than a problematic one.

> So, I think there is a serious case for purging do_md, and routines that calculate globals, energies, pressure, temperature, etc. of anything to do with any of the velocity-Verlet integrators (including the ekinf vs ekinh stuff). We can keep the orphaned low-level VV-specific update routines temporarily, since we anticipate being able to use some of that. I know I'm a bit trigger-happy in wanting to get rid of doubtful code, but here
>
> - it is clear that the original implementation is still deficient in essential qualities years later,
> - that implementation greatly complicates the implementation of core functionality that is working correctly,
> - preserving the old implementation of VV will not give us much of value moving forward with a new implementation,
> - if we remove the implementation of VV, the simplified simulation routines that result will be easier to form into building blocks we can use to re-implement VV integrators that clearly work correctly.

I think that there is value in having some working implementation to compare to. md-vv-avek can be purged, though, and I think that will make the existing code simpler. I can do this today if needed (as well as checking over the restart bugs). Also, one of the harder things to do with the new

implementation is getting the evaluation of the kinetic energy right and have it be efficient. Keeping the existing logic in for a bit could be useful for checking that.

#82 - 12/23/2015 04:24 AM - Gerrit Code Review Bot

Gerrit received a related patchset '1' for Issue #1137.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: I8bc441d945f13158bbe10f097e772ea87cc6a559
Gerrit URL: https://gerrit.gromacs.org/5472

#83 - 12/23/2015 04:55 AM - Mark Abraham

> Yeah, so long as somebody reviews them. Working on the testing infrastructure is something most people seem to avoid doing :-) Uploaded first round of these to gerrit.

You're completely right (cross fingers again for NIH grant to get funded to do this). I think that one issue is that it's a bit hard to add to the testing infrastructure -- it's involved regenerating the tests, there's one overall error tolerance and test that try to find any change don't get flagged, small changes can go under the radar. Perhaps it would make sense to put some documentation up on the website to make it clear exactly what the testing infrastructure is?

> Agreed, we can and do break things while cleaning things up. However, as already said, I reproduced these issues with release-4-6, and AFAICR do_md didn't get all much action since 4.5 days...

I'll take a look and see . .

> Nor does it seem a great idea that VV checkpoints write on-step V where LF checkpoints write half-step V (AFAICS).

Well, the natural place to checkpoint VV is on step v, while FL is halfstep V. Things get much more complicated if you try to checkpoint in a non-natural place -- you have to restart halfway, which requires restart ordering that is not part of the natural algorithm.

> And nobody anywhere knows whether the contents of ekinf vs ekinh have meaningful content, so the checkpoints just contain everything...

Agreed, checkpoints really should contain only one.

> > So, I think there is a serious case for purging do_md, and routines that calculate globals, energies, pressure, temperature, etc. of anything to do with any of the velocity-Verlet integrators (including the ekinf vs ekinh stuff). We can keep the orphaned low-level VV-specific update routines temporarily, since we anticipate being able to use some of that. I know I'm a bit trigger-happy in wanting to get rid of doubtful code, but here

Or alternately, branch, purge, and implement the new organization there in that code. . . .

> Really, we should be thinking about a battery of tests that will prove that we are implementing VV, e.g. for a single particle with a position restraint, or something like that.

Well, that prove we are implementing anything correctly. This is essentially a unit test, which is a good thing.

> Then scale those up to more complex cases. And also proving that, given appropriate starting points, equivalent LF and VV will run the same trajectory. Those are much more valuable tests than "it does the same as the previous thing, but we don't know if that is broken."

Yes. Though it seems like will require a more general testing structure (run two tests and compare them against each other).

#84 - 12/23/2015 07:11 AM - Mark Abraham

Looks like Michael edited my last post inline to reply to it. No big deal, but someone reading might otherwise be confused.

#85 - 12/23/2015 07:12 AM - Michael Shirts

Mark Abraham wrote:

> Looks like Michael edited my last post inline to reply to it. No big deal, but someone reading might otherwise be confused.

Whoops! Thought I hit quote, not edit. Apologies. Thanks for noting that.

#86 - 12/23/2015 08:22 AM - Mark Abraham

Mark Abraham wrote:

> Yeah, so long as somebody reviews them. Working on the testing infrastructure is something most people seem to avoid doing :-) Uploaded first round of these to gerrit.

You're completely right (cross fingers again for NIH grant to get funded to do this). I think that one issue is that it's a bit hard to add to the testing infrastructure -- it's involved regenerating the tests, there's one overall error tolerance and test that try to find any change don't get flagged, small changes can go under the radar. Perhaps it would make sense to put some documentation up on the website to make it clear exactly what the testing infrastructure is?

Such stuff is all in the repo and auto-built to the website: http://jenkins.gromacs.org/job/Documentation_Nightly_master/javadoc/. The developer guide there currently doesn't have much mention of regression or integration tests - Teemu wrote some nice docs for the unit testing, and I am using some of the same technologies for the integration tests. Now that I have added the ability in my current Gerrit patches for them to get values from .edr files, then we have the minimum required machinery to re-implement the regression tests. All that a regression test in our current style requires is the ability to compare some of the terms of the final energies. I will add some docs on how to use those, now that there's enough stuff to suggest that others can write tests in such a way. There's various suggestions at #1587, but so far the style of the new testing framework suits my convenience because I'm the one who's actually done some work :-)

> Agreed, we can and do break things while cleaning things up. However, as already said, I reproduced these issues with release-4-6, and AFAICR do_md didn't get all much action since 4.5 days...

I'll take a look and see . .

Will make a bug report shortly.

> Nor does it seem a great idea that VV checkpoints write on-step V where LF checkpoints write half-step V (AFAICS).

Well, the natural place to checkpoint VV is on step v, while FL is halfstep V. Things get much more complicated if you try to checkpoint in a non-natural place -- you have to restart halfway, which requires restart ordering that is not part of the natural algorithm.

I don't see anything intrinsically un-natural about checkpointing at the point we have half-step velocities, i.e. move the checkpointing to before the first VV half step. We're going to write all the coupling-algorithms parameters to the checkpoint, so that seems like it should work. Now normal start, restart and rerun all naturally enter the MD loop at the same place. What's the implementation advantage to doing VV checkpoints with full-step velocities?

> And nobody anywhere knows whether the contents of ekinf vs ekinh have meaningful content, so the checkpoints just contain everything...

Agreed, checkpoints really should contain only one.

> So, I think there is a serious case for purging do_md, and routines that calculate globals, energies, pressure, temperature, etc. of anything to do with any of the velocity-Verlet integrators (including the ekinf vs ekinh stuff). We can keep the orphaned low-level VV-specific update routines temporarily, since we anticipate being able to use some of that. I know I'm a bit trigger-happy in wanting to get rid of doubtful code, but here

Or alternately, branch, purge, and implement the new organization there in that code. . . .

We could indeed make an integrator-development branch, purge vv to get the simplifications I hope for, and re-implement there. But to compare with a working vv implementation still means checking out master branch and running code there, and one could just as easily check out the commit before the branch event. The main advantage is that if progress turns out to be unsatisfactory, well, it isn't master branch and we can walk away from it. Recent Jenkins and releng improvements makes implementing automated testing on a couple of branches straightforward. We might have to do such for developing task parallelism, so the experiment isn't a waste of any time there. The main caveat is that we have to merge from master into this development branch every week or two, else someone (me) gets a hell of a job merging at the end...

> Really, we should be thinking about a battery of tests that will prove that we are implementing VV, e.g. for a single particle with a position restraint, or something like that.

Well, that prove we are implementing anything correctly. This is essentially a unit test, which is a good thing.

Definitely. Sivak 2014 looked like it had some useful leads.

> Then scale those up to more complex cases. And also proving that, given appropriate starting points, equivalent LF and VV will run the same trajectory. Those are much more valuable tests than "it does the same as the previous thing, but we don't know if that is broken."

> Yes. Though it seems like will require a more general testing structure (run two tests and compare them against each other).

Exactly - but my new rerun and restart tests do just that. This side-steps both the storage of large chunks of binary data, and arbitrary variation from re-ordering arithmetic. There's some extra run-time cost, but we have already accepted the principle that there are going to be compiler-hardware-OS combinations that we won't test every commit, and tests that take noticable time can go into the same Jenkins category. Ideally, we would write the .edr files to memory (or a mock) so it's faster, and I have some WIP for that, but we don't need it to get started.

#87 - 12/24/2015 04:42 AM - Gerrit Code Review Bot

Gerrit received a related patchset '1' for Issue #1137.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: I76b968ab60417778c0461bb89005eb0cf28c8a75
Gerrit URL: https://gerrit.gromacs.org/5487

#88 - 12/24/2015 10:54 AM - Gerrit Code Review Bot

Gerrit received a related patchset '1' for Issue #1137.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: Idf4b19c0ad0b385a260f7d59a72215b59c94d4f1
Gerrit URL: https://gerrit.gromacs.org/5488

#89 - 12/24/2015 12:47 PM - Gerrit Code Review Bot

Gerrit received a related patchset '1' for Issue #1137.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: I71b688a67a80e1f55134e81dab7a11a66942cff4
Gerrit URL: https://gerrit.gromacs.org/5489

#90 - 12/26/2015 04:27 PM - Erik Lindahl

While it will be great to have more general solutions long-term, short term I agree with Mark that we will either need somebody to prioritize looking into and fixing the bugs affecting restarts within the next week or so, or we should remove the bad code paths.

#91 - 01/03/2016 03:44 AM - Michael Shirts

Erik Lindahl wrote:

> While it will be great to have more general solutions long-term, short term I agree with Mark that we will either need somebody to prioritize looking into and fixing the bugs affecting restarts within the next week or so, or we should remove the bad code paths.

The vast majority of the restart issues have been addressed (see issue 1883).

#92 - 01/03/2016 03:57 AM - Michael Shirts

A question about integrator organization:

Currently, there are a significant number of conditionals in the update code; for example, checking whether it is a virtual site or shell particle, checking whether it is frozen, and so forth. As the possibility of more modular functions is introduced that might iterate over the particles multiple times, I'm interested in the possibility of trying various things to simplify this, for example:

1) creating arrays that contain only indices of the particles integrated (would only need to be reinitialized when domain decomposition is redone).
2) creating constant arrays (size of the particles) at the start of each iteration that contain constants like the velocity scaling factors for temperature control or precomputed booleans for complex logicals.

Note this might not work for freezing, since freezing is currently done per-dimension as well as per-particle.

Any other thoughts on simplifying the loops themselves without sacrificing much speed or introducing worse complications?

#93 - 01/03/2016 09:19 AM - David van der Spoel

Just a quick thought: the external accelerations should be moved out of the integrator for reasons of clarity of code. Even though it will mean an additional loop somewhere else if one use it, it will make the integrators somewhat cleaner.

#94 - 01/03/2016 03:48 PM - Teemu Murtola

I have one question about checkpointing: with the new structure, do we need to support checkpoint elsewhere than at the top level where the integrator loop repeats? In particular with multiple time stepping, the repeating unit can be quite a few steps; I'm not sure whether the current code supports checkpointing at any step, but it would simplify things somewhat if we would not need to support restarting the integration somewhere in the middle of the loop.

A similar question related to starting the simulation: the velocities and coordinates might not be in the same state after the repeating unit for different integrators, but ideally the same input would mean the same thing with different integrators. A prime example is the half- vs full-step velocities, where

using one or the other as input may require starting the integration halfway the integrator loop for velocity verlet.

#95 - 01/03/2016 04:50 PM - Mark Abraham

Michael Shirts wrote:

> A question about integrator organization:
>
> Currently, there are a significant number of conditionals in the update code; for example, checking whether it is a virtual site or shell particle, checking whether it is frozen, and so forth.

In the case of virtual sites, the update is meaningless because we'll recompute from the positions of the real particles, so I imagine a reasonable solution is to have the vsite code zero the forces after projection, and now the update code does not need to know anything.

Is there a way to treat shells without needing to have conditionals outside the shell code, David?

> As the possibility of more modular functions is introduced that might iterate over the particles multiple times, I'm interested in the possibility of trying various things to simplify this, for example:

If so, and depending how often this might run, then we should also consider SIMD-izing the update. Currently the performance is dominated by constraints (and particularly their communication), however.

> 1) creating arrays that contain only indices of the particles integrated (would only need to be reinitialized when domain decomposition is redone).

That would be quite feasible, but the DD code already sorts the particles into those local and non-local to the present domain, so we could consider having further categories here.

> 2) creating constant arrays (size of the particles) at the start of each iteration that contain constants like the velocity scaling factors for temperature control or precomputed booleans for complex logicals.

That's doable. If there's more than a couple of constants for each particle, then cache pressure could become a thing. If so, then e.g. using pre-computed masks on AVX512 is an option (ie. if there's three possible velocity scaling factors, we do a SIMD broadcast of each, and then loop through the masks, doing the logical equivalent of two ternary ?: operations in two SIMD operations).

> Note this might not work for freezing, since freezing is currently done per-dimension as well as per-particle.

Yeah, we should special-case that out, like David also suggests for accelerations. Currently it's not slow enough to be a problem, but we are not all that far from it being one.

> Any other thoughts on simplifying the loops themselves without sacrificing much speed or introducing worse complications?

They're currently a non-problem, so with a bunch of the complexity readily able to be lifted out, and with various further kinds of fancy tricks up our sleeves, I think we're fine to write code naturally, avoiding branches in inner loops, and optimize later.

#96 - 01/03/2016 05:10 PM - David van der Spoel

As regards both shells and vsites, I think they can be ignored. So one could instead check whether a particle is eptAtom. In order to deal with frozen particles, maybe we should make their mass infinite, such that they don't move in the constraint algorithm either? Or give them a new particle type?

#97 - 01/03/2016 05:28 PM - Mark Abraham

David van der Spoel wrote:

> As regards both shells and vsites, I think they can be ignored.

Kind of. We have to avoid doing arithmetic that leads to floating-point exceptions, even when we know we're going to ignore the result, because that lets us run our tests using a mode that makes any exception fatal.

> So one could instead check whether a particle is eptAtom.

I think the objective should be to avoid having any checks in loops - if a shell particle or virtual site would have zero force by construction, and then we won't get any issues with floating-point exceptions.

> In order to deal with frozen particles, maybe we should make their mass infinite, such that they don't move in the constraint algorithm either? Or give them a new particle type?

Setting the invmass to zero is workable, but needing to use a different mass for different dimensions is awkward. We'll keep a generic integrator loop that can do anything (also checks the implementation of specialized versions), and simulations with frozen atoms can use that. I can't think of a case where you'd want to use some fancy integrator that might be somewhat slow **and** need frozen atoms.

#98 - 01/03/2016 05:57 PM - Mark Abraham

Teemu Murtola wrote:

> I have one question about checkpointing: with the new structure, do we need to support checkpoint elsewhere than at the top level where the integrator loop repeats? In particular with multiple time stepping, the repeating unit can be quite a few steps; I'm not sure whether the current code supports checkpointing at any step, but it would simplify things somewhat if we would not need to support restarting the integration somewhere in the middle of the loop.

The implementation has to coordinate all domains agreeing to checkpoint, so in practice there is an underlying granularity to the checkpoint period. I'd strongly suggest we checkpoint only from the outermost level - there's no need to be fancy with that. I'd also prefer that we checkpoint from the start of the loop, but I'm open to discussion.

> A similar question related to starting the simulation: the velocities and coordinates might not be in the same state after the repeating unit for different integrators, but ideally the same input would mean the same thing with different integrators. A prime example is the half- vs full-step velocities, where using one or the other as input may require starting the integration halfway the integrator loop for velocity verlet.

First, I see no intrinsic reason why the implementation of leap-frog and velocity Verlet should use conditionals spread over the same body of code. We should be able to express both of them in ~100 lines of code, so concerns about maintaining duplicate slabs of code should be low.

I wasn't around for the discussion at the time, but probably we required md-vv to write velocity output at the half-step for consistency with leapfrog, because the file formats didn't permit any distinction to be drawn. It would make sense for md-vv to write all output on the full step, and to checkpoint there too. **Edit: looks like that is already what it does.** If so, we'd want to require that md-vv writes TNG files, so we can make sure the right metadata gets into the user's hands. Leapfrog output should stay as it is. The checkpoint file format can grow some fields to make sure we don't permit anything to go wrong.

Given the above, I would suggest we plan to checkpoint at the top of the loops for the various integrators in whatever way is natural for that integrator.

If not, then I suggest we consider having one-time code that runs before the loop to update the state so that we always enter the loop with the invariants maintained. This will help keep the integrator loop as simple as possible, which has to be a pre-requisite for considering multi-part integrators. :-)

#99 - 01/03/2016 06:17 PM - Teemu Murtola

Sure, there is no need to have that kind of conditionals. And there is no need for the checkpoint files to be interchangeable between different integrators. But what I was referring to was that given a particular structure with velocities, starting a simulation from it should interpret it as the same physical state. Otherwise, things can be very confusing for users. Similarly, trying to simplify the integrator should not lead to explosion of complexity in analysis: the user should not be forced to use different analysis tools depending on what integrator they chose to use, and remember this kind of details.

#100 - 01/03/2016 07:28 PM - Mark Abraham

Teemu Murtola wrote:

> Sure, there is no need to have that kind of conditionals. And there is no need for the checkpoint files to be interchangeable between different integrators. But what I was referring to was that given a particular structure with velocities, starting a simulation from it should interpret it as the same physical state. Otherwise, things can be very confusing for users.

I think it is already surprising that in order to run a velocity-Verlet simulation I have to construct my input with velocities to be interpreted as one half step before the positions :-). Most people won't care either way, of course, so long as something correct happens. Regardless of how we right the code, it is in principle possible for a user to replicate the trajectory from a leapfrog simulation with velocity Verlet (and vice versa). The interpretation of the state doesn't matter except in the context of an integrator, and given that there is an integrator, then interpreting the state in the way natural for that integrator seems reasonable to me.

Otherwise, some integrators will have to implement extra code (e.g.) for a pre-loop half step, just to preserve the undocumented(?) invariant that grompp+mdrun interprets the .gro input with v trailing x by a half timestep. We could ask gmx-users if anybody cares about this point.

> Similarly, trying to simplify the integrator should not lead to explosion of complexity in analysis: the user should not be forced to use different analysis tools depending on what integrator they chose to use, and remember this kind of details.

Agree on the principle, but find it hard to think of an analysis that would suffer from needing a second implementation. At the moment, probably none of the tools acknowledge that the leapfrog velocities trail by a half step. Looking more closely at the current code, md-vv looks like it actually writes on-step velocities, so already the user has to keep track of what the integrator was, in order to correctly interpret the results...

#101 - 01/03/2016 07:50 PM - Teemu Murtola

There are analyses that use both the coordinates and velocities, and some of those need to perform the half-step forward/backward integration if fed

with half-step velocities to be rigorously correct. Depending on the application, the magnitude of the error varies, and some of the most sensitive parts may anyways depend on the internals of the integrator (e.g., if you need to compute the "force" that would correspond to the constraints used), but tools like this will definitely care. And people have been confused on the mailing list that why the velocities in a constrained molecule do not appear to preserve the constraint lengths. I agree that in most cases this does not matter at all, but people who need to troubleshoot stuff (whether in our code or in something they have done) would probably appreciate consistent behavior.

#102 - 02/03/2017 04:35 PM - Gerrit Code Review Bot

Gerrit received a related patchset '6' for Issue #1137.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: gromacs~master~I8bc441d945f13158bbe10f097e772ea87cc6a559
Gerrit URL: https://gerrit.gromacs.org/5472

#103 - 02/15/2017 06:41 AM - Michael Shirts

- File integrator.pdf added

#104 - 02/15/2017 06:41 AM - Michael Shirts

- File deleted (integrator.pdf)

#105 - 02/15/2017 06:41 AM - Michael Shirts

- File deleted (integrator.tex)

#106 - 02/15/2017 06:42 AM - Michael Shirts

New integrator proposal to discuss during the call 2/15/2017 posted.

#107 - 02/15/2017 05:11 PM - Michael Shirts

- File integrator.pdf added

#108 - 02/15/2017 05:12 PM - Michael Shirts

- File deleted (integrator.pdf)

#109 - 02/17/2017 07:26 PM - Mark Abraham

I recall Michael was planning to upload an update, but I don't see it yet. Fortunately I can still get the old PDF, so can offer some feedback. (Can we do some simple versioning of these, please?) Happy to discuss over telco, or whatever.

First, I do appreciate this is an early stage proposal. As such, there's a lot of good things. Most of my comments relate to how I think we'll want our classes to eventually look, and this should influence our design thinking, even at high level. And some are optimizations we might plan to make later.

Classes should have single responsibilities. Setting up an object with complex configurable behaviour is definitely a responsibility worth separating from the behaviour of the fully constructed object. It makes that object easier to test, and harder to misuse than in two-stage initialization scenarios. For example, an Integrator makes sense as a container of (handles to?) elements, that knows how to execute them to do_one_step. But e.g an IntegratorFactory should have the responsibility for parsing e.g. a description string to decide what elements to construct (perhaps before getting ElementFactory to do so, and giving them to whoever stores them), before putting them in an ordered container with which one can construct the Integrator pretty much ready to use. This kind of approach gives us lots of advantages, including better testability, lower code coupling, more type safety, and loading fewer cache lines in inner loops.

Similarly, I'm not convinced there's a good role for Element::update(). Each element knows what requirements it has of the state, e.g. PositionUpdate element knows it will need an up-to-date view of the positions of the home atoms for this domain. During setup, it registers that interest with the domain decomposition manager, providing some kind of access or callback, so that only when the set of home atoms changes does the view of the positions in the PositionUpdate element (eg. a start and end pointer) get updated via that callback (or visitor or whatever). For example:

```
// In PostionUpdateBuilder():
  stateManager->registerNeed(State::Needs::ToReadHomeAtomPositions, &newPositionUpdater.readPositionView_)
  stateManager->registerNeed(State::Needs::ToWriteHomeAtomPositions, &newPositionUpdater.writePositionView_)
// Obviously this requires that stateManager_ can rely on the lifetime of newPositionUpdater,
// and vice-versa, but this seems like a natural constraint to accept.
```

Otherwise, the domain's StateManager has two ugly options: call a polymorphic updater on everything which will know what to go and ask for from StateManager, or contain Element-specific logic that means that useless function calls aren't made.

The Simulation class currently has responsibility for holding some data structures and doing initialization. I can perhaps see a role for it along the lines of "manage access to the state" so maybe StateManager :-) But there may be things I haven't thought of yet!

We won't do any dynamic allocation once we enter the outer loop.

Integrator had a depth field - I don't think it should ever need to know.

Integrator had a repeat field - perhaps we should distinguish between

- an Integrator as a nearly arbitrary sequence of Elements and
- a single Integrator as an Element of a higher Integrator
- repeats of Integrators as an Element of a higher Integrator,

so that now it is natural to express a multi-step integrator as a sequence of Elements, where one Element is a RepeatingSequence of an Integrator, which has its own sequence of elements. I haven't thought of a strong advantage yet, but if there will be Integrators that don't repeat within a multi-step integrator scheme, then building in the notion of repetition into the Integrator is overhead on the hot path.

I envisage the key-value based Options infrastructure (to which mdp/tpr handling is migrating) will place logically const data in an object for each module. Those modules might later have methods that serve the above Factory roles for Integrators and Elements. So keep an eye on those developments in gerrit.

I'll be happy to discuss / help remove from t_state those archaic things mentioned.

Dynamical state definition looks pretty good. Conjugate gradient EM has the cg_p vector that should probably go into the mix.

We should carefully differentiate kinds of communication. The important ones are halo-style, reduce, and all-to-all. Referring to "global" or nothing is misleading, because it is the number and pattern of messages that determines the cost, not the number of participating ranks. Most MPI messages in MD are tiny in size but frequent in occurence, compared to what hardware and software infrastructure is typically designed for.

It'll be straightforward to have do_force able to do any relevant combination of force, virial or energy, but currently various parts of the infrastructure is a bit hard-coded on the idea that we always use the force for MD.

Saving output should probably be done by handing off a buffer according to the user's schedule to a thread e.g. doing TNG compression. We'd pre-allocate enough room for that. Our first implementation would probably just do a buffer copy on the hot path, which is neither great nor horrible (unless very frequent). However, an instructive example is where domain decomposition happens every 50 steps, but we write positions every 10 steps. There's no strong reason that the StateManager has to keep using the same position vector each Integrator stage (so long as it knows who its clients are, as above). Conceivably, the StateManager and IOManager would share a pre-allocated pool of buffers each large enough for the positions. The Runner tells the IOManager to read from the current position vector (and work gets scheduled some time), and the Runner tells the StateManager that the current position vector is read only (to use in the next step), and it has to give another position vector from the pool to whoever next needs to write (but I'm currently not sure how to coordinate that). Or we just have the I/O thread do the buffer copy, and enforce that it has to have signalled completion before anybody can write to the position vector (again not yet sure how to coordinate).

Task parallelism will initially be localized within Elements (most likely ForceCalculator).

#110 - 02/19/2017 03:42 AM - Michael Shirts

Will be responding over several comments

> I recall Michael was planning to upload an update, but I don't see it yet. Fortunately I can still get the old PDF, so can offer some feedback. (Can we do some simple versioning of these, please?) Happy to discuss over telco, or whatever.

The pdf there is an updated one. I will start versioning with the next one I put up (probably tomorrow) so it is clear. I will give you direct access to the git repository as well, so you can file issues against it or edit it yourself.

> First, I do appreciate this is an early stage proposal. As such, there's a lot of good things. Most of my comments relate to how I think we'll want our classes to eventually look, and this should influence our design thinking, even at high level. And some are optimizations we might plan to make later.

Great! More information coming in subsequent comments.

#111 - 02/19/2017 04:17 AM - Michael Shirts

> Integrator had a depth field - I don't think it should ever need to know.

Looking at this more, I think we can avoid the integrators having to know the depth, if the integrators are created at run time out of the sequence string.

> Integrator had a repeat field - perhaps we should distinguish between
>
> - an Integrator as a nearly arbitrary sequence of Elements and
> - a single Integrator as an Element of a higher Integrator
> - repeats of Integrators as an Element of a higher Integrator,

All of these are correct functionalities.

> so that now it is natural to express a multi-step integrator as a sequence of Elements, where one Element is a RepeatingSequence of an

> Integrator, which has its own sequence of elements. I haven't thought of a strong advantage yet, but if there will be Integrators that don't repeat within a multi-step integrator scheme, then building in the notion of repetition into the Integrator is overhead on the hot path.

Agreed. We can simply have an Element that is a RepeatingSequence.

> Classes should have single responsibilities. Setting up an object with complex configurable behaviour is definitely a responsibility worth separating from the behaviour of the fully constructed object. It makes that object easier to test, and harder to misuse than in two-stage initialization scenarios. For example, an Integrator makes sense as a container of (handles to?) elements, that knows how to execute them to do_one_step. But e.g an IntegratorFactory should have the responsibility for parsing e.g. a description string to decide what elements to construct (perhaps before getting ElementFactory to do so, and giving them to whoever stores them), before putting them in an ordered container with which one can construct the Integrator pretty much ready to use. This kind of approach gives us lots of advantages, including better testability, lower code coupling, more type safety, and loading fewer cache lines in inner loops.

Yes, we can definitely have separate IntegratorFactory and ElementFactory concepts. Still working on the exact ordering and responsibilities.

> I envisage the key-value based Options infrastructure (to which mdp/tpr handling is migrating) will place logically const data in an object for each module. Those modules might later have methods that serve the above Factory roles for Integrators and Elements. So keep an eye on those developments in gerrit.

OK.

> We should carefully differentiate kinds of communication. The important ones are halo-style, reduce, and all-to-all. Referring to "global" or nothing is misleading, because it is the number and pattern of messages that determines the cost, not the number of participating ranks. Most MPI messages in MD are tiny in size but frequent in occurence, compared to what hardware and software infrastructure is typically designed for.

I'll try to make this clearer in the document.

> It'll be straightforward to have do_force able to do any relevant combination of force, virial or energy, but currently various parts of the infrastructure is a bit hard-coded on the idea that we always use the force for MD.

I think this can be addressed in the future. We don't really need to worry about this until we optimize MC and other code paths.

> Task parallelism will initially be localized within Elements (most likely ForceCalculator).

OK, we'll only worry about separating tasks for I/O right now.

#112 - 02/19/2017 04:29 AM - Michael Shirts

> Similarly, I'm not convinced there's a good role for Element::update(). Each element knows what requirements it has of the state, e.g. PositionUpdate element knows it will need an up-to-date view of the positions of the home atoms for this domain. During setup, it registers that interest with the domain decomposition manager, providing some kind of access or callback, so that only when the set of home atoms changes does the view of the positions in the PositionUpdate element (eg. a start and end pointer) get updated via that callback (or visitor or whatever). For example:

OK, I think we can do something like that. Depending on ordering of tasks, we might implement something 'simple' like the update approach, but we'll think about how to set up the registers and callbacks. Can, does this make sense to you? Can you try writing up a version in the dummy code in integrator.tex?

> Otherwise, the domain's StateManager has two ugly options: call a polymorphic updater on everything which will know what to go and ask for from StateManager, or contain Element-specific logic that means that useless function calls aren't made.

That sounds like a reasonable thing to avoid.

> The Simulation class currently has responsibility for holding some data structures and doing initialization. I can perhaps see a role for it along the lines of "manage access to the state" so maybe StateManager :-) But there may be things I haven't thought of yet!

OK, we'll keep thinking about this. Eric, Can, any thoughts?

#113 - 02/19/2017 05:04 AM - Michael Shirts

> Saving output should probably be done by handing off a buffer according to the user's schedule to a thread e.g. doing TNG compression. We'd pre-allocate enough room for that. Our first implementation would probably just do a buffer copy on the hot path, which is neither great nor

> horrible (unless very frequent).

OK. Eric has some thoughts on this in terms of memory management. He may comment here.

> However, an instructive example is where domain decomposition happens every 50 steps, but we write positions every 10 steps. There's no strong reason that the StateManager has to keep using the same position vector each Integrator stage (so long as it knows who its clients are, as above). Conceivably, the StateManager and IOManager would share a pre-allocated pool of buffers each large enough for the positions. The Runner tells the IOManager to read from the current position vector (and work gets scheduled some time), and the Runner tells the StateManager that the current position vector is read only (to use in the next step), and it has to give another position vector from the pool to whoever next needs to write (but I'm currently not sure how to coordinate that).

Eric, any additional thoughts here.

> Or we just have the I/O thread do the buffer copy, and enforce that it has to have signalled completion before anybody can write to the position vector (again not yet sure how to coordinate).

This is how I was thinking about it, but I'm up for different approaches.

#114 - 02/20/2017 07:29 PM - Peter Kasson

Maybe we want to either break the StateManager discussion into another thread or reserve detailed consideration for later?
(Unless this is strictly limited to integrator concerns, then this starts to touch a number of design considerations.)

#115 - 02/21/2017 12:48 AM - Eric Irrgang

Michael Shirts wrote:

> > The Simulation class currently has responsibility for holding some data structures and doing initialization. I can perhaps see a role for it along the lines of "manage access to the state" so maybe StateManager :-) But there may be things I haven't thought of yet!
>
> OK, we'll keep thinking about this. Eric, Can, any thoughts?

Managing access to state data needs to be discussed for code beyond the integrator, and elements may have state that is not related to trajectory data, so we might be talking about two different objects for StateDataManager and IntegratorData. I think a question to consider is whether library-wide state/trajectory data can be managed the same way as performance-critical integrator access or whether the integrator needs its own solution.

Another point I would like to avoid closing the book on is whether the Elements really ought to be classes or whether they can be functions with state and data managed by other structures. Michael will think I'm beating a dead horse since fused elements can be written at a later time, but automatic building of fused_elements that allow for compile-time optimizations could be done with variadic templates and free functions, but I don't know whether there is a way to allow the compiler a chance to do loop fusion (whether the element methods could be inlined in a sequence of element calls) by iterating through a container of previously constructed objects. Optimization may seem premature, but such a design decision would substantially change the nature of possible read/write access control in addition to substantially different code structure. I talked with Can about it a bit over the weekend and he may have additional comments.

> > Saving output should probably be done by handing off a buffer according to the user's schedule to a thread e.g. doing TNG compression. We'd pre-allocate enough room for that. Our first implementation would probably just do a buffer copy on the hot path, which is neither great nor horrible (unless very frequent).
>
> OK. Eric has some thoughts on this in terms of memory management. He may comment here.

Yes, I would think the elements should neither be allocating heap memory nor performing substantial copy operations, particularly if the same copied data might be useful by several consumers. Mark's suggestion allows the shortest path forward, but there may be sufficiently light-weight options for more general access to state data throughout the library API that probably warrant a separate discussion thread. One implementation detail would be whether to manage read/write access through required API calls or to have a simpler interface by hiding the implementation in the construction/destruction of short-lived data handle/view objects. I can also think of some more advanced features of a "snapshot" mechanism that the StateManager could provide that I will elaborate on elsewhere.

> > However, an instructive example is where domain decomposition happens every 50 steps, but we write positions every 10 steps. There's no strong reason that the StateManager has to keep using the same position vector each Integrator stage (so long as it knows who its clients are, as above). Conceivably, the StateManager and IOManager would share a pre-allocated pool of buffers each large enough for the positions. The Runner tells the IOManager to read from the current position vector (and work gets scheduled some time), and the Runner tells the StateManager that the current position vector is read only (to use in the next step), and it has to give another position vector from the pool to whoever next needs to write (but I'm currently not sure how to coordinate that).
>
> Eric, any additional thoughts here.

I've given this a fair bit of thought because it is the sort of functionality Peter and I want for API access to state data. I have some notes I can share, but in the simplest case, I imagine a set of ManadedData<T> structures that are "write-once" but can clone themselves as new write handles are needed, notifying any subscribers that a shared pointer to the updated data is available or being produced. The memory could/should be allocated from a pool, presumably managed by something like a StateDataManager. I don't know if anyone can predict the performance impact (good or bad) related to cache pressure or memory pipelining (at least I can't), but a big downside is that there would be much less opportunity for fruitful loop fusion if every logical write happens in a different memory location (see above).

> > Or we just have the I/O thread do the buffer copy, and enforce that it has to have signalled completion before anybody can write to the position vector (again not yet sure how to coordinate).
>
> This is how I was thinking about it, but I'm up for different approaches.

That last is certainly the simplest approach and appropriate for proof-of-concept code, but implies either hard-coded sequential operation or a decision on some sort of synchronization scheme. There are several incremental evolutions being discussed here for both performance and access control, and it seems appropriate for the code to evolve in small steps while Can and Michael iterate on their implementation. I think some questions that might guide the longer-term planning, though, are

1. whether to couple state data access by the integrator to library-level state data API,
2. what access control is needed or desirable in either case,
3. whether/what asynchronous access should be designed in, what design decisions allow the best combination of extensibility, interface simplicity, and opportunity for optimization,
4. and what performance profiling tells us about what really matters.

Peter Kasson wrote:

> Maybe we want to either break the StateManager discussion into another thread or reserve detailed consideration for later?
> (Unless this is strictly limited to integrator concerns, then this starts to touch a number of design considerations.)

If there is interest in coupling the design of Integrator <-> state data access to more general state data API, then I would be interested in continuing the discussion sooner than later, but I acknowledge that the requirements may not be similar enough or the integrator implementation might not warrant increased complexity, at least in the short term.

#116 - 02/26/2018 04:03 PM - Mark Abraham

- Related to Feature #1867: make coupling implementations reversible added

#117 - 03/06/2018 11:09 PM - Mark Abraham

- Related to Bug #1339: Center of mass drift with Nose-Hoover, MTTK and md-vv added

Files

integrator.pdf	410 KB	02/15/2017	Michael Shirts