GROMACS - Task #1140

Class design for passing options and data

02/06/2013 09:13 PM - Erik Lindahl

Status:	New
Priority:	Normal
Assignee:	
Category:	core library
Target version:	
Difficulty:	uncategorized

Description

At the Gromacs meeting today we started talking about ways to clean up forcerec and related structure, and possibly creating substructures.

This is a more general problem. Right now Gromacs is design so that everything knows about everything, which causes lots of stupid cross-dependencies. However, I don't think the solution should obviously be a set of hierarchical open structures where the highest-level code knows about everything.

1) It would be good if settings too are truly opaque, i.e. that the internal settings of the nonbonded interactions really are internal to that module. The table data is probably a good example of such information. The challenge here is how we should "set" the settings of a lower-level object while keeping it opaque, and not having lots of glue code: If we add a new option to the mdp file we don't want to modify half a dozen intermediate routines to make sure it propagates all the way to the module that should use it.

2) On the other hand, it would be nice if we could have a closer correspondence between future modules and a future XML-based mdp format. It would be even better if the user could then add a new record, and that we could then query this user setting based on the text string rather than creating entries in each structure first. The problem is that this can almost reverse the problem, i.e. that the mdp file is the center that all modules query, so it is not trivial to combine with (1).

3) It should be possible to initialize and use submodules independently of each other. This might mean that some settings have to be duplicated in multiple modules.

This was just a couple of random thoughts - please continue!

Related issues:

Related to GROMACS - Task #1170: mdlib reorganization	**New**

History

#1 - 02/07/2013 06:21 AM - Teemu Murtola

It would be great if people would finally have time to look in detail in how this is done in the trajectory analysis framework, because I did it on purpose quite generally. In particular for options that are local to one module, this solution should work for mdp options as well. All the glue code required already exists in the src/gromacs/options/ module (most of the code there is not really for the glue, but instead for converting strings to other types of values, for error-checking, and for specifying characteristics of the options).

The basic approach is that each module has a initOptions(Options *options) method, and calls options->addOption(...) for any option that it needs. At the same time, it specifies local variables that will receive the values of those options. For any code higher up, the contents of the Options is opaque (although they can query something about the properties of individual options), and they don't need to know what options the module added. It is also possible to construct a hierarchy of the Options objects for subsectioning (not really used in the current code, but could be useful for mdp). After all modules have added their options, then a parser is called to push values to these options (in the analysis framework, this is a command-line parser). Again, the parser doesn't need to know what options are specified; it will get an error (exception) from the options engine if it tries to assign a value to an unknown value, and the options themselves are responsible of converting the input strings to their internal value type.

Shared options need some more thought (there are some in the analysis framework, e.g., for plotting, but don't know whether the design for that is the best possible). But we definitely shouldn't duplicate the definitions of any single option in multiple places, since maintenance becomes then messy. One option would be that shared options are dealt with on a higher level, suitably separated out such that it is possible to initialize them without initializing the whole higher-level module.

Any feedback on why this is not suitable or how it could be improved is very welcome, and I can adjust the implementation in src/gromacs/options/ if necessary. I cannot have predicted all possible uses beforehand, so the implementation focuses on features that are currently used. And please ask if there is anything about the current code that you don't understand or if you need help finding the relevant code.

#2 - 03/08/2013 01:11 PM - Mark Abraham

While looking for something else, I found Google's Protocol Buffers https://code.google.com/p/protobuf/. I think it's just about perfect for a replacement for .mdp:

- simple text format for arbitrarily structured input (see below)
- the protc binary compiles the format description into C++ classes with getters and setters that preserve field names (https://developers.google.com/protocol-buffers/docs/proto#generating)
- those classes can then be used to serialize back to text **or** to binary (https://developers.google.com/protocol-buffers/docs/cpptutorial)
- binary format is fully portable (https://groups.google.com/forum/?fromgroups=#!starred/protobuf/SwIl-bCB3gA)
- versioning of formats (back- and forward-compatible; https://developers.google.com/protocol-buffers/docs/overview)
- the binary format looks pretty compact (https://developers.google.com/protocol-buffers/docs/encoding)
- text format much more compact and human-editable than XML
- formats we develop will be extensible by third parties
- possibly makes it easy to parse our input files with future Python tools
- perhaps possible to include documentation (e.g. mdp_opt.html stuff) with the format description (https://developers.google.com/protocol-buffers/docs/techniques#self-description)
- permissive source code license
- heavy existing deployment by Google (48,162 different message types defined in the Google code tree across 12,183 .proto files.)
- no need for GROMACS to write/maintain code to parse .mdp files and serialize that component of .tpr files

So I'd envisage a next-gen .tpr format that had three components:

- protocol buffer for .mdp-type info,
- some data structure for the topology info (perhaps preserve the existing .itp input file formats, but express them internally with protocol buffers for consistency?)
- XDR-based binary restart data (x, v, etc.)

We'd probably have to offer a conversion utility from .mdp (so use our current .mdp reading machinery to fill the above classes. which we can then write out in the new text format with the generated classes; deprecate that legacy code, of course).

Example of text format (from their webpage):

```
# Textual representation of a protocol buffer.
# This is *not* the binary format used on the wire.
person {
  name: "John Doe"
  email: "jdoe@example.com"
}
```

The "messages" can be nested programmatically and arbitrarily, so (e.g.) the T-coupling groups would be a variable-length collection of "structs" of info for each T-coupling group.

Example of format description:

```
package tutorial;

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
```

We'd still need our own code for checking for required fields and consistency of options, of course.

How we propagate information from the protocol buffers into the modules that use them is still an open question.

We'd package the protocol buffer source code. Their COPYING.txt is permissive:

```
Copyright 2008, Google Inc.
```

```
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

    * Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above
copyright notice, this list of conditions and the following disclaimer
in the documentation and/or other materials provided with the
distribution.
    * Neither the name of Google Inc. nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Code generated by the Protocol Buffer compiler is owned by the owner
of the input file used when generating it.  This code is not
standalone and requires a support library to be linked with it.  This
support library is itself covered by the above license.
```

Source code is 14MB unpacked, but there's probably gtest and java dirs we don't need to include.

With CMake, we could build and run protoc at configure time (if we think we need to see the generated classes in order to write code that uses them) or at build time (probably best).

#3 - 03/08/2013 04:45 PM - David van der Spoel

Mark, do you mean that user should type something like:

1. Textual representation of a protocol buffer.
2. This is **not** the binary format used on the wire.
   person {
   name: "John Doe"
   email: "jdoe@example.com"
   }

That seems quite a bit less readable and writable than mdp options now. We definitely do not want those files to be binary, so that argument falls, and even though xml may be slower that will hardly be measurable. I think XML with an XML folding editor is much superior in terms of usability. On freecode there are quite some examples of free editors: http://freecode.com/search?q=xml+editor&submit=Search but if we decide to stay with something more along the lines of what we have now than Teemu's implementation of options can probably help increasing the maintainability of such files. Don't know how well Teemu's would play with an XML file?

#4 - 03/09/2013 06:25 AM - Teemu Murtola

For analysis, this issue can be broken into several related topics (could have missed a few):

1. How are the settings stored in memory/in code? Can be further split down into two subtopics:
   1. What is the final storage format that is used during computation/processing? Is it simple variables in the modules that need the values, or something else?
   2. Do we need a separate, intermediate storage format that is used for different forms of serialisation?
2. How are the settings propagated from serialisation to the modules that actually need them, and also how are they propagated back?
3. How are the settings serialised from and to a text-based format (".mdp")?
4. How are the settings serialised from and to a binary format (".tpr")? Do we need to keep the ability to read old tpr files? How much are we prepared to change the format? Or do we want to get rid of it completely?
5. How are the settings manipulated generally for printing (e.g., "md.log", "gmxdump") and for comparison (e.g., in "gmxcheck")?

Protocol buffers seem to provide the an option for the infrastructure at least for 1b, 4 and seem to also allow doing 5 in a generic way. I'm quite neutral/slightly positive for using them (in particular versioning support that doesn't need to be implemented by hand is nice). There are still some issues that would need to be solved/would require additional analysis:

- If we don't want to require the user to compile protoc separately, and if we don't want to include the generated sources in git, cross-compilation will be difficult. This is because CMake would need to first compile protoc that runs on the build host, then run it to generate the source, and then compile the protocol buffer library and Gromacs such that it runs on the target architecture. I'm sure there are ways to make this happen relatively easily for the user, but it may create quite a complex build system.

- To work around the above, we can of course include the generated sources in git, but we already have something like 75% of the source code under version control generated, and that is not going to reduce when new kernels are added...
- Protocol buffers don't seem to be designed for user input as much as efficient serialisation. Users may not appreciate an error message like "your mdp file is invalid" for any syntax errors they may make.
- Have not looked at whether it is possible to make the protocol buffers in any way "modular" such that only some parts of the code would know of some parts of the buffer. But this may be difficult to achieve, essentially making the in-memory protocol buffer the same as the current inputrec. However, it will be difficult for any solution that uses a separate binary format to avoid such modularity completely, unless the binary format simply uses the mdp strings as keywords.

My options implementation is relatively easy to interface to any input format, but it does not provide abilities to process arbitrarily structured information. In particular, repeating elements (other than simple vectors of strings/numbers) are not currently really supported. There may be some ways to improve this, at some cost in complexity...

#5 - 03/09/2013 06:46 AM - Teemu Murtola

- File options.png added

To clarify how the options currently work on a high level, I made a diagram. For demonstration, it only includes two fictional modules, A and B, and omits a lot of details on the interaction between "parser" and "options". It may be a bit unorthodox in some of the symbols it uses, but maybe it is more clear than my textual description.

#6 - 03/14/2013 01:41 PM - Mark Abraham

David van der Spoel wrote:

> Mark, do you mean that user should type something like:
>
> 1. Textual representation of a protocol buffer.
> 2. This is **not** the binary format used on the wire.
>   person {
>   name: "John Doe"
>   email: "jdoe@example.com"
>   }
>
> That seems quite a bit less readable and writable than mdp options now.

I was thinking that

```
RunControlParameters
{
  integrator: md
  dt: 0.002
  nsteps: 20
}

EnergyMinimizationOptions
{
  emtol: 0.001
}

TemperatureCoupling
{
  type: v-rescale
  Groups:
  {
    name: Protein
    ref-t: 298
    tau-t: 1
  }
  {
    name: SOL
    ref-t: 298
    tau-t: 1
  }
}
```

is barely a change from our existing .mdp format. A script to convert users files from legacy .mdp to something like that would be easy to write. It'd be a pain if users had to wrap quotation marks around strings, though.

> We definitely do not want those files to be binary, so that argument falls,

Agreed, but we probably want to express it in binary when we want to write a .tpr file. Currently we have a pile of grompp code that reads in the .mdp options, parses them and writes out a new t_inputrec data structure in the .tpr file. Then reads it again in mdrun/tools. If we use protoc, most of that legacy C code leaves the GROMACS maintenance burden. We still have to have our own input sanity checks, but now they're all about the logic of the simulation process, not crap about whether the number of temperature coupling group names matches the number of temperature coupling reference temperatures. Crap about parsing floats goes away. We write a format description and end up with code that delivers some pre-populated objects that replace what should be the current role of t_inputrec. Now (say) Teemu's option management machinery can just cherry-pick for values.

> and even though xml may be slower that will hardly be measurable. I think XML with an XML folding editor is much superior in terms of usability. On freecode there are quite some examples of free editors: http://freecode.com/search?q=xml+editor&submit=Search

I have a lot of experience of editing a particular large and highly structured XML document. I did develop effective emacs workflows, but it was expensive. Doubtless there are good XML-aware WYSIWYG editors by now. But that feels like a big barrier to adoption for GROMACS. People are going to want to edit an .mdp file on a remote supercomputer, and X may not even be available. Even with good syntax highlighting, there's a lot of noise on each line.

> but if we decide to stay with something more along the lines of what we have now than Teemu's implementation of options can probably help increasing the maintainability of such files. Don't know how well Teemu's would play with an XML file?

I haven't yet looked at Teemu's code, but I was assuming from the above that it wasn't doing .mdp file handling or parsing. protoc would be replacing the layer of grompp and mdrun that handles setting up and serializing t_inputrec. Actually propagating configuration values into modules that use them will be our job, via Teemu's machinery.

#7 - 03/14/2013 02:06 PM - Mark Abraham

Teemu Murtola wrote:

> For analysis, this issue can be broken into several related topics (could have missed a few):
>
> 1. How are the settings stored in memory/in code? Can be further split down into two subtopics:
>   1. What is the final storage format that is used during computation/processing? Is it simple variables in the modules that need the values, or something else?
>   2. Do we need a separate, intermediate storage format that is used for different forms of serialisation?
> 2. How are the settings propagated from serialisation to the modules that actually need them, and also how are they propagated back?
> 3. How are the settings serialised from and to a text-based format (".mdp")?
> 4. How are the settings serialised from and to a binary format (".tpr")? Do we need to keep the ability to read old tpr files? How much are we prepared to change the format? Or do we want to get rid of it completely?
> 5. How are the settings manipulated generally for printing (e.g., "md.log", "gmxdump") and for comparison (e.g., in "gmxcheck")?
>
> Protocol buffers seem to provide the an option for the infrastructure at least for 1b, 4 and seem to also allow doing 5 in a generic way. I'm quite neutral/slightly positive for using them (in particular versioning support that doesn't need to be implemented by hand is nice). There are still some issues that would need to be solved/would require additional analysis:
>
> - If we don't want to require the user to compile protoc separately, and if we don't want to include the generated sources in git, cross-compilation will be difficult. This is because CMake would need to first compile protoc that runs on the build host, then run it to generate the source, and then compile the protocol buffer library and Gromacs such that it runs on the target architecture. I'm sure there are ways to make this happen relatively easily for the user, but it may create quite a complex build system.

For these reasons, I'd be strongly in favour of bundling the protoc source in git. There's a straightforward dependency of the protoc binary and the format description file for generating the resulting code. The CMake solution in the cross-compiling case is to have two build trees and import the host binary into the cross-compile build (http://www.vtk.org/Wiki/CMake_Cross_Compiling#Using_executables_in_the_build_created_during_the_build).

However, for tarballs we distribute, there's no need to get involved. The format description is fixed for that code version, so we just bundle the code protoc produces. No need to bundle protoc in the tarball.

We do version some code that we generate (e.g. kernels), but that serves the purpose of helping us review that it is correct. Versioning the code generated by protoc would not be useful. Only someone needing to handle a new option ever needs to see the generated code, and they can do that in their build tree.

> - To work around the above, we can of course include the generated sources in git, but we already have something like 75% of the source code under version control generated, and that is not going to reduce when new kernels are added...

Yeah, I don't think the cross-compiling burden is worth adding more generated versioned code.

> - Protocol buffers don't seem to be designed for user input as much as efficient serialisation. Users may not appreciate an error message like "your mdp file is invalid" for any syntax errors they may make.

Yes, we'd have to try things out to see how these kinds of errors are handled. I'm not sure how good a job grompp does, because I don't make those errors much!

> - Have not looked at whether it is possible to make the protocol buffers in any way "modular" such that only some parts of the code would

> > know of some parts of the buffer. But this may be difficult to achieve, essentially making the in-memory protocol buffer the same as the current inputrec. However, it will be difficult for any solution that uses a separate binary format to avoid such modularity completely, unless the binary format simply uses the mdp strings as keywords.

That'd be nice if it was feasible. I think it is reasonable to have some structure in our .mdp file format so that users don't see a million unrelated strings (currently grompp emits comments to provide some structure, which is fine). I also think it is reasonable to not require that our structure of our code components mirrors the structure of our input files (or vice-versa). That seems to risk the implementation driving the interface too much. As such, I'm very happy if we just use protoc to replace the t_inputrec I/O and .mdp parsing. t_inputrec is used everywhere in mdrun, and that's bad design we should get rid of. I think the way to do it is to accept that I/O, parsing, sanitizing and propagating of input are all different tasks. If there's an external tool that will do several of them well for us, that's fantastic. I'm quite happy if Teemu's mdp_parser and options modules are just calling the interface protoc (or anything else reasonable) can build.

> My options implementation is relatively easy to interface to any input format, but it does not provide abilities to process arbitrarily structured information. In particular, repeating elements (other than simple vectors of strings/numbers) are not currently really supported. There may be some ways to improve this, at some cost in complexity...

There are several instances of arbitrary repetition in .mdp settings (anything to do with groups, for starters). The T-coupling module needs to be prepared to receive an arbitrary number of groups to handle, and whatever feeds it needs to be able to do that.

#8 - 03/15/2013 03:47 PM - Teemu Murtola

Only a comment on one part for now.

Mark Abraham wrote:

> David van der Spoel wrote:
>
> > but if we decide to stay with something more along the lines of what we have now than Teemu's implementation of options can probably help increasing the maintainability of such files. Don't know how well Teemu's would play with an XML file?
>
> I haven't yet looked at Teemu's code, but I was assuming from the above that it wasn't doing .mdp file handling or parsing. protoc would be replacing the layer of grompp and mdrun that handles setting up and serializing t_inputrec. Actually propagating configuration values into modules that use them will be our job, via Teemu's machinery.

The options machinery doesn't do mdp parsing, but using the facilities it provides it is possible to write and .mdp parser that is only few tens (or perhaps a hundred) lines long (only counting non-trivial lines). Take a look at what the command-line parser (src/gromacs/commandline/cmdlineparser.cpp) needs to do, it isn't that much...

> Teemu Murtola wrote:
>
> > My options implementation is relatively easy to interface to any input format, but it does not provide abilities to process arbitrarily structured information. In particular, repeating elements (other than simple vectors of strings/numbers) are not currently really supported. There may be some ways to improve this, at some cost in complexity...
>
> There are several instances of arbitrary repetition in .mdp settings (anything to do with groups, for starters). The T-coupling module needs to be prepared to receive an arbitrary number of groups to handle, and whatever feeds it needs to be able to do that.

Maybe I need to rephrase this: anything in the current .mdp format should be easily parseable with the options module. Arbitrary number of string or numeric values for a single option are supported. The only thing that isn't is your proposed vector-of-structures. This isn't impossible to implement, but requires quite a bit of work (even if the code may not get that much more complex; can't say without actually trying...).

#9 - 05/13/2014 10:27 AM - Mark Abraham

- Target version changed from 5.0 to 5.x

#10 - 07/15/2014 05:17 PM - Teemu Murtola

- Category set to core library

#11 - 07/11/2016 08:19 PM - Mark Abraham

- Target version deleted (5.x)

Files

options.png	8.65 KB	03/09/2013	Teemu Murtola