GROMACS - Task #2916

Task # 2833 (New): Update topology datastructures

Decide future of symtab

04/03/2019 01:35 PM - Mark Abraham

Status:	New	
Priority:	Normal	
Assignee:	Paul Bauer	
Category:	core library	
Target version:	2021-infrastructure-stable	
Difficulty:	uncategorized	

Description

We have a bunch of strings, particularly in the topology data structures (e.g. atom name, name of atom type, name of residue, name of residue type, element, FEP B-state versions of some of those).

For reasons that probably include minimizing size, we have a legacy t_symtab structure, into which strings can be inserted. Data structures like t_atom then contain pointers that refer to entries in the symtab. We need to modernize this so that we can improve the data structures that use these strings.

Option 1)
Migrate to using a std::string everywhere. This makes use, allocation and serialization trivial. Small-string optimizations are sometimes our friend here. However the .tpr file might e.g. double in size, because now alongside per-atom data like x[3],v[3],q we also have a string by value for at least some of the above types of data. That doubling is probably no big deal.

Option 2)
A staged replacement to use a new StringTable type that encapsulates some storage, provides an interface to insert into, can be serialized predictably, can return handles into it suitable for serialization of dependent data structures, and is at least somewhat efficient to look up once we're done inserting things into it. That could be implemented with e.g. a std::vector<string> that perhaps gets sorted when insertion is done, and if we can find a suitable C++14 version of https://en.cppreference.com/w/cpp/header/string_view then data structures like t_atom can hold such views and be serialized in terms of an index into the underlying container (which is roughly what t_symtab does now).

Option 2 would usefully generalize to a (future) data structure that is a std::unordered_map<std::string, StringTable> so that we can have separate tables for atom names, atom types, etc. that we refer to like stringTables["atomname"].insert("CA"). That will hopefully map naturally to a future key-value format to replace the .tpr where for each particle we index into a table of particle names, a table of possible residue types (which itself might index into a table of residue type names?). Having deserialized all the tables first, now when we read the particular per-particle fields that refer into them, we can have the correct string_view.

History

#1 - 04/03/2019 01:38 PM - Mark Abraham

- Assignee set to Paul Bauer

Paul, can you look into a gmx::compat::string_view for us to consider adding? We need

- implements the C++17 std::string_view interface as far as possible in C++14
- suitable license for us to include it
- open source repo we can import a known commit from
- portable enough

Ideally it also has

- test suite we can adopt
- no dependencies other than the stdlib

Not finding one would not be a deal breaker for the above, but if we find one, then we'll probably find other uses for it.

#2 - 04/03/2019 02:17 PM - Paul Bauer

Already found one, from the same guy + same license you found for std::optional. Working on porting it right now.
Repo is here: https://github.com/martinmoene/string-view-lite

#3 - 04/03/2019 03:13 PM - Mark Abraham

Paul Bauer wrote:

> Already found one, from the same guy + same license you found for std::optional. Working on porting it right now.
> Repo is here: https://github.com/martinmoene/string-view-lite

Great

#4 - 04/04/2019 03:16 PM - Paul Bauer

While working on the replacement I ran into a rather large problem with using something like a vector as the underlying table storage.
There, both iterators and references are invalidated after inserting a new element if this changes the capacity of the container, making the vector pretty much useless during the initial assignment.
It might still be possible to use it for storing a finalized table (as the size will be constant then), but during construction something else has to be used.

#5 - 04/04/2019 04:03 PM - Paul Bauer

I have a new patch set ready to be uploaded that changes the implementation back to the std::map of <std::string, int>, with a return type used that is a wrapper around gmx::compat::string_view.

The wrapper contains both the view on the string in the table, as well as the unique integer value associated with it that can be used to serialize the datastructure.
One question is if we want to continue with this proposal, or try something else that does the same.
I explicitly went with the ordered map, as other container types don't have the advantage of no reference invalidation (std::vector would otherwise be the obvious choice) or don't allow associating the string with an integer for serialization (otherwise I would go back to std::set).

Another question is if we want to use the unordered_map instead, as we could generate the new container after serialization again with the explicit numbering and thus circumvent the penalty for looking up entries in the ordered version.

Open for suggestions here on how to continue this work.

#6 - 04/04/2019 11:17 PM - Roland Schulz

With the ordered vector any mapping of the SymTab (both idx->StringView (get_symtab_handle) and StringView->idx (lookup_symtab)) would be only valid after the table is finalized (should be checked by assert). Any strings used in code which might be used prior to finalizing the table would need to use normal std::string.

If you use map<string, int> (either ordered or unordered) then idx->StringView (get_symtab_handle) would be slow (O(N)) because you would be forced to do a linear search. SymTab would need to contain both a unordered_map<string, int> and an unordered_map<int, string_view> to support fast lookup in both directions. Or do we not need to have fast lookup in both directions? If we want to use it for MPI we need one direction for send and one direction for receive right?

I don't see any reason to use map over unordered_map (independently of whether we have one map or two map with one per direction). unordered_map also guarantees that references stay valid. See https://en.cppreference.com/w/cpp/container#Iterator_invalidation for a nice overview what stays valid after insert.

#7 - 04/05/2019 02:44 PM - Paul Bauer

We need to lookup at least during the initial send, so lookup should not be too slow in the end.
I can update the implementation to use two maps, but I'm not sure what would be best for constructing the elements in it (as we don't need to have a duplication of the strings).
Changed locally to use the unordered_map and wrote test to see that it will be serialize the same way.

#8 - 04/06/2019 02:47 PM - Paul Bauer

I was thinking more about this and realized that the table is never used for index based lookup before it is supposed to be finalized.
This means that we should be able to live with the unordered_map for the entries during building of the topology, and then can move it an ordered vector for serialization and de-serialization.
Any thoughts about that?

#9 - 04/09/2019 10:29 AM - Roland Schulz

Based on your use pattern (didn't verify) this sounds like a good plan. Not sure whether it is faster to use std::map or unorderd_map for this. std::map is slower to insert but is already sorted so you don't need to pay for sorting when copying it into vector. Either way it should be pretty fast and the difference is probably minor.

BTW: If you do need two maps for some reasons: I would make one to be std::string and the other const char* (based on the discussion that we need C-API). I would add it first to the one taking the std::string and the insert the return value of .c_str() into the other. That way you wouldn't store the string twice.

#10 - 04/15/2019 05:56 PM - Berk Hess

Serializing into one single char buffer and an index buffer would be useful for MPI broadcasting mtop. Currently this can take a lot of time when there are many strings (as there can be for e.g. atom types and names).

#11 - 05/07/2019 06:02 AM - Mark Abraham

Berk Hess wrote:

> Serializing into one single char buffer and an index buffer would be useful for MPI broadcasting mtop. Currently this can take a lot of time when there are many strings (as there can be for e.g. atom types and names).

That sounds like an optimization for serialization. It would likely be useful, but doesn't seem like a priority, and I don't think it helps us make a good first move.

#12 - 05/10/2019 04:29 PM - Mark Abraham

- Description updated

#13 - 09/06/2019 02:35 PM - Paul Bauer

- Target version changed from 2020-beta1 to 2021-infrastructure-stable

not likely for 2020