GROMACS - Feature #3439

Feature # 3379 (New): C++ API for simulation input and output

Optimize successive simulation segments

03/11/2020 01:42 PM - Eric Irrgang

<table>
<thead>
<tr>
<th>Status:</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td></td>
</tr>
<tr>
<td>Category:</td>
<td>core library</td>
</tr>
<tr>
<td>Target version:</td>
<td></td>
</tr>
<tr>
<td>Difficulty:</td>
<td>uncategorized</td>
</tr>
</tbody>
</table>

**Description**

Filesystem I/O and various initialization code impose a lot of overhead in transitioning from one simulation job to the next. Traditionally, dynamic simulation parameters or bundles of sequential simulation segments (replica exchange) have required deep integration into the simulator code. There is a lot of middle ground that could be addressed to improve the maintainability of logic that is traditionally invasive and improve the performance of logic implemented outside of the GROMACS core.

In general, this issue is filed to coordinate discussion of the reusability of resources within a GROMACS process and the evolution of lighter-weight transitions between simulation segments.

**Two major paths of optimization**

1. **Reduce I/O requirements for resource ownership.** Resources that are currently expensive to acquire, or that are potentially reusable, need to have a representation that is decoupled from expensive operations. As interfaces are converted to be agnostic to filesystem (or other expensive) sources, we can provide less expensive mechanisms for resource acquisition or transfer, as appropriate.

2. **Increase resource reusability.** For various resources that are historically assumed to be used only once in a GROMACS program, we can work towards being able to (a) use a resource an arbitrary number of times in a process, and (b) prototype a new single-use resource on a previous instance to reduce initialization costs. We need to compartmentalize resource acquisition and clean-up, identify invariants, encapsulate the (re)establishment of valid state, and formalize dependency relationships.

**Near term optimization targets**

I think the biggest overhead in simulation (re)launch is probably (a) the initial file I/O and (presumably to a much lesser degree) (b) the coordinate data scatter. Comments? What else?

This seems like a two-step optimization, then, requiring (a) conversion to initialization from an abstraction that may be memory-backed instead of file-backed, and (b) allowing the external representation of TPR/input data to (internally) rely on the data locality management framework.

**Proposed road map**

1. Decouple gmx::Mdrunner initialization from filesystem I/O (ref #3374)
2. Allow gmx::Mdrunner output to be manipulated/converted into input (trivial example: extend nsteps) for a new runner (complete reinitialization, but no extra filesystem I/O) through Builder interaction and/or Prototype pattern.
3. Begin identifying and handling components that do not always need to be reinitialized in terms of various inputs/parameters. Avoid unnecessary reinitialization in special Builder cases.
4. Expand subscription/signaling for initialization dependencies so that components can be converted to take charge of their own (re)initialization logic.

**Concepts / Questions**

General philosophical question: to avoid miscommunication, do we prefer to think in terms of modifying and reusing a gmx::Mdrunner or separating out the reusable bits for lightweight creation of new Mdrunners? My inclination is to use unique and tightly scoped implementation objects at each level, and to think in terms of optimizing their acquisition, but this is an important conceptual detail that needs to be clear in discussion. It seems reasonable to me that the MdModules container might be longer lived, but previous discussions concluded that long-lived state should live in outer code scopes, with something like the SimulationContext (or a conceptual parent XContext) being the only thing expected to persist for the entire program life time.
Essentially, there are roughly three ways we could think of simulation segments in terms of evolving the current code base. (1) The work represented by a gmx::Mdrunner instance, (2) The work represented by a Simulator instance, or (3) the scope of an API "run" call or call-back interval. The technical impact of the decision is minimal, but the architectural impact is significant.