GROMACS - Task #986

Task # 838 (New): Improve generic error reporting routines

Handling C++ out-of-memory errors

08/03/2012 09:54 AM - Teemu Murtola

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

Description

The C code uses macros snew() and sfree() for handling out-of-memory errors. snew() is implemented as a macro, and on out-of-memory it prints the source location and the size of the attempted allocation and terminates the program. There is also some logging implemented in these macros. This issue is about how much effort and complexity we want to put to try to duplicate this behavior in C++ code, which uses exceptions for error reporting and a different approach to memory management.

Some alternatives for the exceptions to use:

1. Use plain std::bad_alloc. This is thrown by default by the C++ standard library.
2. Use GMX_THROW(std::bad_alloc()).
3. Implement our own gmx::OutOfMemoryError that derives from std::bad_alloc and GMX_THROW it.

Some alternatives for how to actually produce the exceptions:

1. Let the standard library throw std::bad_alloc, and only use something else in situations where our code explicitly throws on out-of-memory. Note that this means that we hardly ever throw anything else than std::bad_alloc, except in code that is converted from C and still has to use malloc or something similar for some purpose.
2. Use set_new_handler() and throw our custom exception type. Note that this potentially affects the whole binary, also for other projects that link to Gromacs. Also note that the handler does not have access to the attempted size of the allocation.
3. Use custom syntax for memory allocation, for example gmxnew ClassName(parameters) instead of new ClassName(parameters), and possibly also for gmxdelete. Implement these as macros that pass source location etc. to the actual allocation function, and implement a wrapper for the standard allocator (possibly as operator new/operator delete with additional parameters) that throws a different exception type. Note that this does not affect allocation done within the standard library (e.g., std::vector), nor delete operations in smart pointers (this is possibly fixable with some additional work).
4. Overload global operator new and operator delete to throw exceptions that include the allocation size. This may have multiple caveats; I'm not very familiar with overloading these.

Note that none of these approaches give direct source locations for allocations occurring in the standard library (e.g., std::vector), which are the most likely to fail (because they most likely are the largest allocations). It may be possible to include some stack trace information to the exception (at least on some platforms) to provide a partial solution.

If we implement anything more complex than std::bad_alloc and/or plain throws/GMX_THROWs, we need to be extra careful to avoid infinite recursion in a case that parts of the out-of-memory exception are allocated on the heap, which may in turn trigger an out-of-memory condition.

History

#1 - 08/03/2012 09:54 AM - Teemu Murtola

- Tracker changed from Bug to Task

#2 - 02/15/2014 07:55 PM - Teemu Murtola

- Project changed from Source code reorganization to GROMACS

- Category set to core library

- Target version changed from 5.0 to future

#3 - 12/24/2014 08:42 PM - Mark Abraham

First, long overdue thanks for the analysis.

I think it is misguided for us to continue attempt to adorn all OOM errors with file name and line number. For Gromacs, such cases are basically either

- user error (e.g. gmx rms on huge trajectory, which we should catch with a more descriptive error than we historically have), or
- programmer error (and thus generally amenable to generating a stack trace in a debugger).
  We've had one known per-step memory leak from released version of mdrun (Forgotten sfree() in the expanded-ensemble implementation) and I don't remember any fixed before release.

Additionally, some OOM conditions on some systems might thrash to death (or get killed by a job-queue manager) before we learn about it, and (as Teemu notes) the largest allocations will probably be in std::vector anyway, where the relevant information is a few calls back up the stack trace. So attempts to decorate are not always going to work usefully, even if it can be done.

In some cases, we'll probably want to catch an OOM exception and decorate it with some information perhaps before a re-throw. In most cases, I think the right behaviour will be to eventually catch the OOM exception from a try block very close to where the command-line runner called the C-main-style function for this program (and then probably terminate). Various people on the Internet suggest this, so it must be right. :-D

Thus, I don't think there's value in getting involved with any decorated new, delete, throw, exception types, handlers, etc. for OOM. (We already have GMX_THROW that does nice decoration for exceptions we choose to throw, thanks to Boost and Teemu.)

Doing this would mean that future mdrun modules are expected to throw std::bad_alloc across their boundaries. I'm comfortable with that. My impression is that mainline mdrun code does very few dynamic allocations within an MD step, and those are normally small extensions to existing buffers.

If/when we can design a real library API, we would need to reconsider the throw-across-boundaries behaviour at the library boundaries (at least). There may be some analysis tools that can recover from OOM on one frame and proceed to the next - they would need to implement logic for doing that in a reasonable way, and probably still need a top-level catch.

Where we are calling a library that might report OOM (e.g. TNG, libxml2) then we probably need to detect the error code and GMX_THROW whichever exception makes sense.