improve the configurability of regression tests

Currently there is no mechanism to pass some run configuration/hardware support setting to the regression tests. A variety of different ways have been proposed/implemented so far (e.g. "max-mpi-ranks" file, checking the name of the test directory) but these are more of the workaround/quick and dirty solutions for a problem that could use a more elegant, extensible, and future-proof solution.

An easy to implement and quite extensible solution would be to replace the "max-mpi-ranks" file with general rc/config file containing key-value pairs. The file could be optional and it could contain runtime, hardware, or software/configuration requirements of the tests e.g. whether GPU or MIC can be used, max DD, max ranks, max threads, etc.

Related issues:
- Related to Gromacs - Task #1455: fix regression tests to correctly rerun and r...
- Related to Gromacs - Bug #1730: gmx compare does not compare all fields of a ...

Associated revisions
- Revision d2d27247 - 03/02/2015 09:53 AM - Mark Abraham
  Add infrastructure for running MPI integration tests
  Some algorithms need a minimum number of ranks to run meaningfully. With this patch, ctest acquires the necessary logic that "make tests" can do the right thing if the MPIEXEC_* variables are set correctly for the execution environment.
  Added machinery so that integration tests can be written to use the new logic, but no tests actually do so yet.
  Updated install-guide section for the new (and some old) functionality.
  Minor fix to the ordering of mdrun integration test files so that they match the comments.
  Refs #1587
  Change-Id: I01b419c10db42e98be708d1d4dfeba34a6d22d
Run replica-exchange tests with MPI enabled

Replica exchange only works with real MPI and at least two ranks. This can now be handled by ctest, so we can run such tests automatically. The other integration tests run in MPI mode, but do not place any requirements upon the number of ranks, so they continue to run in the usual way.

Fixed the replica-exchange integration tests so that they actually do run. Changed tests to use mdrun -multi because it is easier to write the tests that way. (Using mdrun -multidir is problematic, because grompp needs to write the .tpr file in the subdirectories. When using mdrun -multi, the only issue is that the name of the .tpr file for grompp and mdrun is different, because the latter will append the MPI rank.)

Fixed member variables to conform to naming convention.

Refs #1587

Change-Id: I11dc06f3aac81a80d679b036aef24762e9e6c819

---

History

#1 - 08/26/2014 09:25 PM - Mark Abraham

I think the only thing that is elegant, future-proof and extensible is a standalone test binary. Take a look at the history of regressiontests suite on BG (where gmx and mdrun are built for different platforms) and Windows (where POSIX is a fantasy), and think about things hardware might do in future!

I've spent the last two days discovering that if we'd had a Verlet-scheme energy-group test case all along, then it's never been able to pass on BlueGene/Q, because the intermediate precision of everything is double and so the accumulation of the potential in the j loop rounds differently from the reference code for each energy group. Even hacking in round-to-single at the j-loop accumulation point is not enough to fix the issue. So we need another hack to relax the tolerance of just the energy part (not forces!) of the new test for Verlet-scheme energy-group and the old nbnxn-free-energy, which requires we run gmx check with different tolerances in those specific cases, or risk having a bug in forces exposed by changes in that code path be less likely to be detected. Fun fun fun. The only elegant way to add such flexibility is right next to the test case definition - in the same test source file, in the same repo as the code change.

This is part of the reason for my work on the integration test machinery - coding this stuff in C++ is not pretty either, but at least it's a language devs already have to be comfortable hacking! Accordingly, I'll continue to have limited interest in anything that looks like adding features to the regressiontests machinery.

The retry-mdrun-until-it works feature is not great, but it achieves the purpose of getting the code to run and produce a testable result. Permitting an empty .rc to fail back to old behaviour is not actually an improvement; it also can allow a test that might run in either CPU or GPU mode to silently run in just one mode unless someone does the necessary change to the regressiontests repo. So we would need an .rc to have real content so there's a chance someone will think about it. Curating the .rc contents once per test case is all very well, but someone has to write the new machinery to use it, perhaps ripping out the old rerunning machinery, then later someone else will want the values to change because we've relaxed the DD limitations somehow and there could be a bug exposed by regression test case xqczq that we're currently not testing on old_nmax+1 ranks, ...
Two points to Mark's comment (sorry a bit off-topic):

The point of integration tests is to find problems which only occur in the combination of feature A and B. If you you test parallelization separate with a few tests then for any feature not utilized by those few (e.g. ener-grps or vsite), we won't notice if its parallelization doesn't work.

I think we should try to improve reproducibility and not accept non-reproducible parallelization. In those cases where the fastest algorithm isn't reproducible we should have a reproducible fallback. Currently most algorithms are reproducible. The flop option of dlb is a good example of a option to make it testable under reproducible conditions. And the fact that the GPU doesn't have a reproducible aggregation makes it impossible to find a problem if the problem only occurs at large scale. I think this is an unsustainable situation.

Roland Schulz wrote:

Correct, and we should test energies and thus energy groups component-wise, not total like I said above. But I do not think that "testing parallelization" is the right way to think.

Testing that a component works in parallel, by reproducing the result of other implementations, is good. The vsite implementation has nothing directly to do with parallelism - that the various projections work is unit-testable, that DD correctly recognizes that vsite-related data needs communication is testable, that the thread-parallel implementation works the same as the serial implementation is testable. Wanting reproducible parallel hardware-agnostic end-to-end regression test machinery is the poor man's version. With those tests of vsite functionality in place, and machinery to observe that (say) protein in water with vsites produces the right ensemble across various parallelization schemes, what's missing?

I think we should try to improve reproducibility and not accept non-reproducible parallelization.

That's a topic in its own right, but to do that properly we'd have to have a fixed precision implementation. There's a lot to be said for that (convenience in developing & thus confidence in the correct result). But for an opening gambit, to do that on BlueGene/Q (and probably K), where there is only double-precision SIMD, we'd have to do at least the j-loop accumulation for q, C6 and C12 in fixed-point on the non-SIMD part of the CPU, and still we'd have issues with (lack of) intermediate rounding to single precision while actually computing stuff on the FPU before that.

Fixed-point update phase is somewhat more achievable (caveat: I don't understand the detailed arithmetic needs of LINCS and SHAKE).

In those cases where the fastest algorithm isn't reproducible we should have a reproducible fallback.

Sure, but that's only a partial solution - very useful, but mostly of use to devs. We also need tests that are valid on the code that we expect people to use in production simulations, and those are going to be using parallelism, load balance, redundant computation, etc.

Currently most algorithms are reproducible. The flop option of dlb is a good example of a option to make it testable under reproducible conditions.

DLB is reproducible, and thus a great thing to have while developing. But the flop counts are unmaintained, and not a great proxy for CPU cycles any more. Aside, should we stop using -notunepme in regressiontests, in favour of this mode?

And the fact that the GPU doesn't have a reproducible aggregation makes it impossible to find a problem if the problem only occurs at large scale. I think this is an unsustainable situation.
I think we should be thinking in terms of fixed point here, also. Accumulation and reduction starts at the end of the j loop, and we don't really have a mixed-precision mode yet.

#4 - 10/08/2014 11:49 PM - Szilárd Páll

If the test binary approach provides high-level regression testing functionality that can replace what gmxtest does now, I can fully agree with your suggested approach - perhaps even with the unwillingness of maintaining gmxtest if:

- We at least tried to avoid hiding our heads in the sand. It is dangerous to ignore the major shortcomings of the current testing - some of which wouldn't be hard to fix - and there have been several bugs that confirm this. It's easy, but not productive to just point at some of the flaws of the regression testing and use that as an argument for keeping it unmaintained and abandoning right now.

- The framework was ready to replace gmxtest including most test cases (or functionalities) that gmxtests supports ATM ported to the test binary.

- Extending on the above points, regression testing has become an essential part of our workflow, developers rely on it. Hence, creating the false sense of security by leaving regression testing issues not addressed and instead focusing on a different solution is dangerous and overall not a reasonable approach, I think.

#5 - 12/26/2014 12:48 PM - Mark Abraham

Szilárd Páll wrote:

If the test binary approach provides high-level regression testing functionality that can replace what gmxtest does now, I can fully agree with your suggested approach - perhaps even with the unwillingness of maintaining gmxtest if:

- We at least tried to avoid hiding our heads in the sand. It is dangerous to ignore the major shortcomings of the current testing - some of which wouldn't be hard to fix - and there have been several bugs that confirm this. It's easy, but not productive to just point at some of the flaws of the regression testing and use that as an argument for keeping it unmaintained and abandoning right now.

Agreed - it should be kept in place while being replaced, but I think implementing new functionality is a good thing to do in a new harness (whatever form it takes). In 2014, we extended gmxtest.pl for doing CPU-only reruns and per-test-case limiting of parallelism. We did the refactorings in a way that should permit similar behaviour extensions. I'm not opposed to people using that, but I'll be much more interested in moves towards using better infrastructure - and will choose to spend my time creating new infrastructure to address the kinds of concerns you guys raise, rather than extending the old.

- The framework was ready to replace gmxtest including most test cases (or functionalities) that gmxtests supports ATM ported to the test binary.

Regarding the existing regression tests modules:

- simple should pretty much go and be unit tests on bonded functions - I don't think they do anything much else (but we should check)
- kernel will die with the group scheme (these take about half the run time at the moment)
- pdb2gmx is already "run pdb2gmx... grompp... mdrun and check the EM output energy." This is a few hours of work to do in the integration test machinery right now, but would be better done with code that intercepts writing the .edr file and does the checking with reference data in memory. Note that these tests spent years silently doing nothing until someone noticed that the implementation that grepped md.log for the final energy was silently not working...
- complex and free energy test coverage needs to be largely preserved (and in some places converted to Verlet and in many places extended).

Some specific considerations:

1. Already several test cases are documented in a README to be "this other test case run under these different .mdp conditions," which is a bit of a waste from redundant checking for some stuff. It's also hard to get perspective on the test coverage with similar-but-different things at the same level as totally different things. This would be much better with a loop in a single place over matching sets of inputs + expected outputs + tolerances (in old or new harness).

2. It is perfectly reasonable to test that some set of .mdp inputs produces a given set of grompp warnings. But such a test ought to be focused only on that. If the text of the warning is checked, it should be checked against the same format string that produces it, so that there is not manual work for us regenerating reference data for a separate repo requiring coordinated review and merging when someone just wants to fix something trivial. I think the format string should be defined where it is used, though - having to add a format string for a warning to some database of strings somewhere else will lead to people being lazy about adding warnings. Better would be build-time (and tarball-time) parsing of the handful

11/25/2015
of source files so that grompp_warning(timeStepTooShort, "Shortest oscillation period is %f, but time step is %f", period, delta_t) generates a build-tree header file of format strings in some container indexed by timeStepTooShort, so that we can have grompp integration tests parameterized on the .mdp settings and matching expected format strings. Same for mdrun warnings (currently untested). If so, then when someone changes the code so that a warning is no longer issued, then they can address that in the same commit, and reviewers can prompt them to add a test to the set with just adding a couple of lines of .mdp fragment. (One could also argue that grompp_warning when run by the test binary should have a mock implementation that just records the event, so that no fragile parsing of anything is ever needed. That's probably a lot less work. There should also be a separate test that output from grompp_warnings() normally does in fact end up on stderr.)

3. I'm not sure what value exists comparing that the .tpr file generated by the new code matches that of the old code. David probably added it long ago just because it was possible. It's pretty hard to inadvertently change the .tpr output. When we change the .tpr format deliberately there's some value in being able to see via .tpr-diff exactly what we changed, but a Jenkins implementation that accepts an .mdp fragment, builds/gets the old code, builds the new code and runs the new gmx check on the .tpr files is a good way to give developers and reviewers that feedback. I don't see the value in requiring a manual update, review and simultaneous merge of some number of archived .tpr files for every commit that is some kind of breaking change to old .tpr behaviour, in particular because the amount of work tends to increase as we improve test coverage and having to do that work stops us doing other things. (But we should keep tests that show that gmx check correctly observes equivalence and difference between all of its input file types.)

4. That leaves the checks on .edr and .trr output. These are valuable and I have a number of proposals here, but I will speak more of these in a later post.

Note also that the integration test framework is no longer completely new - it's been testing things including mdrun trajectory writing, IMD and Carsten's swap code in Gromacs 5 and hasn't fallen over (but doesn't assert very much about the code it runs...). I'm aware of problems getting ctest to run MPI binaries in parallel, which needs work, e.g the replica-exchange cases are currently neither run automatically (nor useful if they are run manually).

- Extending on the above points, regression testing has become an essential part of our workflow, developers rely on it. Hence, creating the false sense of security by leaving regression testing issues not addressed and instead focusing on a different solution is dangerous and overall not a reasonable approach, I think.

I am prepared to help fix existing problems, but even coordinating fixes needs timely help from other people if we want two people to sign off on all fixes... I could email qualified individuals or spam gmx-developers every time such things need to happen, but now I sound like your mother nagging. I'd like to think the qualified individuals browse git approximately daily and see such things happen.

I think there is greater current and future value if we focus some time on getting the above kinds of things done (plus perhaps being able to run grompp and mdrun without writing a .tpr to disk). In particular, I think we can make the end-to-end testing much more useful through making it run in seconds rather than minutes (through fewer binaries to load, less disk access overall, less parsing of files, less checking of useless things), and not needing so many stored binary blobs or a separate repo. Then, adding more code paths under automatic tests is a much more palatable proposition, for users, devs and Jenkins.
Roland Schulz wrote:

I think we should try to improve reproducibility and not accept non-reproducible parallelization. In those cases where the fastest algorithm isn't reproducible we should have a reproducible fallback. Currently most algorithms are reproducible. The flop option of db is a good example of an option to make it testable under reproducible conditions. And the fact that the GPU doesn't have a reproducible aggregation makes it impossible to find a problem if the problem only occurs at large scale. I think this is an unsustainable situation.

And Mark then wrote:

... Wanting reproducible parallel hardware-agnostic end-to-end regression test machinery is the poor man's version...

I've since learned that there is a fairly cheap way of implementing a reproducible floating-point reduction, which is the key thing we'd like to have to make it possible to demonstrate the equivalence of different implementations or under different degrees of parallelism. I got the idea from http://htor.inf.ethz.ch/publications/img/arteaga-fuhrer-hoefler-reproducible-apps-ipdps14.pdf; you measure the largest force component, and use that to set a power of two called M that is greater than the largest partial sum that will be correctly accumulated. Of course, this is the same thing you need to do for implementing fixed-point reduction, so there might be some guidance available from Amber's choice. You then add and subtract M from each component before accumulating it in the reduction. This does a pre-rounding that makes the accumulation binary reproducible. Since we'll have about a few hundred force contributions for each atom (counting long-ranged as one contribution), then our M from their equation 1 will probably be about 1e7 kJ/mol/nm. That's approximately 2e23, so using such M for forces means the total force on an atom will have around zero digits of precision after the decimal point. However, their choice of M caters to theoretical worst-case maximal partial sums, which we are unlikely to see in practice, so M one or two orders of magnitude smaller is probably useful. There is the option to measure the remainder and do a second pass of accumulation if we think we need to (see their paper), but that would mean implementing some force data structure whose implementation we can vary in a way that doesn't slow down the normal code. We can play with this in a plain-C kernel to see what our actual numbers look like.

The good thing I think we can do is add a special testing build mode where the accumulation at the end of all force component calculations is modified to use a pre-computed M, flagging if any overflow actually happens from choosing M too small. Because the accumulation is now binary reproducible, I think we can run integration tests using any implementation of our algorithms on any platform, and we do not need to choose any tolerance for accumulation or integration losses. Because the only difference in the code is the few flops with M before reduction, then we can be confident that the normal build has no issues arising from parallelism+algorithm+implementation choices that have passed in reproducible mode, and we can store a single set of reference data that many combinations can compare against. Testing in normal mode should also occur (and be available to normal users), but for those we need some old-school floating-point tolerances. The point is to try to stop us missing bugs (e.g. #1647, #1661, and doubtless others) because the problem of choosing old-school tolerances that are tight enough to detect subtle real bugs but loose enough for no false hits is perhaps too hard to solve and maintain. We'd perhaps also have caught #1603 if we'd been able to feel confident to test with PME tuning on (but perhaps no existing test case ran long enough to see the effect).

Such testing does not assert anything directly about the quality of the normal reduction wrt floating-point precision or operand ordering; those issues should be addressed differently e.g. through conservation and ensemble-correctness tests. Once we've got such a modified implementation, it could also be good for probing the limits of the normal accumulation. An alternative could be to pick a scaling factor to convert force components to an integer (whose byte width matches real) and accumulate in that type. The use of M above restricts the practical range of the resulting force, so we waste some of the bits in the significand that we can get back if we use an integer. Then we convert back to real before the update. This could even be cheap enough to use in production code if we find there's something real to gain. (Hur hur, "real") Note that our inter-rank reduction is in double precision, but the j-loop accumulates in single, so I don't think we have something we should really call mixed-precision mode at present.
Mark said

Replacing [current FEP regression test approach], sure...

Szilard said

Getting off-topic here, but let me answer briefly this time because this is getting old. As far as I can tell, that's your personal opinion that the developer community never agreed on.

Right, but what I think counts a lot, because I've made more than half the commits to that repo in the last two years, and if it takes me announcing that I'm not going to add more functionality to it in order to force some change, then that will cost me no sleep. I had been the sole maintainer for several years when we adopted it for CI testing. I said at that time that I thought it was unsuitable, but as IIRC Roland said at the time there was no reasonable alternative in the short term. That was true, but about three years have now passed.

For instance, both Roland and I were on the opinion that currently there is nothing on the horizon that can take up the role that regression-testing has (that is integration tests).

Not true. We already have other machinery for running integration tests for grompp and mdrun. It needs more work, but it can already do as much as the regression tests can do (run gmx grompp, mdrun, and gmx check on the output files, dumping issues to XML/terminal).

The fact that it took me 6 hours to discover that Windows does not have the same behaviour for flushing stderr and stdout, so the test harness can't parse it properly, so that we can't reliably observe that the grompp warnings haven't changed, is itself a bug ([https://gerrit.gromacs.org/4332](https://gerrit.gromacs.org/4332)). This is not an isolated incident, and I'm unwilling to commit more time than absolutely necessary to work on machinery that I do not think serves our needs well enough any more. See above posts for my proposals for new forms to cover the range of testing behaviours that I think are good to preserve.
Gerrit URL: https://gerrit.gromacs.org/4344

#9 - 01/04/2015 03:05 PM - Gerrit Code Review Bot
Gerrit received a related patchset '1' for Issue #1587.
Uploader: Mark Abraham (mark.i.abraham@gmail.com)
Change-Id: I11dc06f3aac81a80d679e036aef24762e9eeec819
Gerrit URL: https://gerrit.gromacs.org/4345

#10 - 04/09/2015 11:23 AM - Mark Abraham
- Related to Task #1455: fix regression tests to correctly rerun and report the mdrun usage added

#11 - 05/12/2015 02:46 PM - Mark Abraham
- Related to Bug #1730: gmx compare does not compare all fields of a .tpr added

#12 - 06/15/2015 02:14 PM - Teemu Murtola
Let me just paste an e-mail from gmx-developers (from Oct 4, 2014) that never got any reply: maybe it gets considered in this context at some point.
There is actually some discussion in this Redmine issue on things that were lacking in that e-mail thread, but it does not hurt to reiterate things. And that said, I'm still a bit sceptical about the possibility to creating large-scale integration tests that are binary-reproducible across different compilers, platforms, and different optimization levels, even if the force reduction could be improved in this respect.

The discussion here focuses mostly on technical implementation details of how to organize the testing, but I think an even more important discussion would be to investigate what we should be testing. A major reason why the existing regression tests do not test things like -tunepme is that they try test for exact reproducibility of trajectories in an inherently chaotic algorithm, and that simply cannot support testing longer runs that would be required for testing the actual mdrun functionality.

This is where unit tests could help: if there were separate unit tests that would test the individual forces and energies, the regression tests could concentrate on testable global properties of longer runs that are still sensitive to correct implementation, such as energy conservation, instead of trying to test every fragile property of the trajectory.

Like Roland said, I think there is nothing inherently bad with an external script to run the tests (although, if we rewrite the script from scratch, I'd propose Python). The main benefit of a C++ testing framework comes from the ability to access internal data structures, which in turn allows more lightweight tests and assertions on the internal state. But in order to have such access, substantial refactoring of the code under test may be required, so this is not a magic bullet. For integration-style tests for whole executables, an external driver written in a scripting language is probably a much easier solution.

Finally, if the blocker is the availability of resources for writing such scripts (and not deciding what they should test), I can volunteer to help with the script infrastructure, if people think that the most valuable thing I could be doing. I can't promise to deliver such scripts in O(days) time, but if the alternative is that it never happens, waiting a few weeks or months should not be a big deal.
And about the configurability (which is the main topic here), I think we should restructure the test cases: instead of creating a set of tests and a script that tries to run all the tests with the same parallelization options. Instead, each test should specify (a set of) parallelization options that it needs to use, selected based on what we think that test should be testing and what different parallelization options make sense in this context. The user/caller of the test script can then tell what resources are available, and the test either runs or it doesn't run (and if it doesn't, it clearly reports why it didn't), using at most the resources the user provided.

Running tests on a user-provided number of ranks etc. is probably only useful for performance testing, and possibly for ensemble validation and such. For this, it does not matter what is the mechanism to run the tests; both the regression tests and the new stuff that uses CTest have the same problem to solve. The issue is particularly bad with the regression tests, leading to the current complicated machinery.

Concern and discussion over the state of pdb2gmx and its tests took place here https://gerrit.gromacs.org/#/c/4772/