I made some observations that the same .tpr ran 5-10% slower under REMD than without. I added a new timer for the REMD routines and found that the cost of doing exchanges only accounted for a small fraction of the increase. See http://lists.gromacs.org/pipermail/gmx-users/2011-January/058059.html for details. My nstlist=5.

REMD uses the "multi-simulation" capability of GROMACS. There is a signalling mechanism that allows the master node of each simulation to communicate to each other. This mechanism controls whether
1) the heuristic neighbourlists are updated (but I think this feature is supposed to be disabled until further notice),
2) checkpointing will occur soon,
3) the simulation will stop soon, and
4) the timing counters will reset soon.

Checkpointing occurs after any of the simulation master nodes observes that the run time has exceeded the time that should elapse after the last scheduled checkpoint, when a signal is set such that after the signal is received at the next neighbour-search step in each simulation the checkpoint file is written. However they don't have to communicate at every neighbour-search step, which is what is happening at the moment. On smaller simulation systems, or with faster processors, or with slower networks, the cost of the intra-simulation communication to cater for 2), 3) and 4) can be too large compared with the cost of doing the simulation itself. As REMD usage, GROMACS scaling and computer sizes improve, this will become an increasingly significant issue.

The code that handles simulation stop conditions also acts on neighbour-search steps, but this seems to me a bit backwards. Non-emergency stop conditions have to be handled in a way that can be synchronized across multi-simulations so that checkpointing is synchronous. However in the absence of 1), there's no reason why the frequency of checking for 2), 3) and 4) needs to be no larger than nstlist, as currently implemented.

There should be a way of configuring such checks to be dependent on nstglobalcomm, rather than nstlist. Better still might be to introduce nstsignalcomm, to cater for multi-simulation scenario where you have lots of simulations each doing large-scale parallel simulations with fairly frequent neighbour-searching (relative to execution time of a parallel MD step). Now the best scenario can be something like nstsignalcomm = 200, nstglobalcomm = 50 and nstlist=10 (nstsignalcomm should be a multiple of nstglobalcomm, I expect). Inter-simulation global communication happens when steps % nstsignalcomm  0, intra-simulation global communication happens when steps % nstglobalcomm  0 and neighbourlists happen as usual. When 1) is used, then probably nstsignalcomm must equal nstlist and that's life.

I'm happy to implement this nstsignalcomm feature, but I thought I should solicit comments on whether my analysis is accurate and complete, and whether there are any pitfalls of which people are aware.
simulations are now decoupled.

It is unclear what the expected behaviour of a multi-simulation should be if the user supplies any of the possible non-uniform distributions of init_step and nsteps, sourced from any of .mdp, .cpt or command line. Instead, we report on the non-uniformity and proceed. It's always possible that the user knows what they are doing. In particular, now that multi-simulations are no longer explicitly coupled, any heterogeneity in the execution environment will lead to checkpoints and -maxh acting at different time steps, unless a user-selected algorithm requires that the simulations stay coordinated (e.g. REMD or ensemble restraints).

In the implementation of signalling, we have stopped checking gs for NULL as a proxy for whether we should be doing signalling at that communication phase. Replaced with a helper object in which explicit flags are set. Added unit tests of that functionality.

Improved documentation of check_nstglobalcomm. mdrun now reports the number of steps between intra-simulation communication to the log file.

Noted minor TODOs for future cleanup.

Added some trivial test cases for termination by maxh in normal-MD, multi-sim and REMD cases. Refactored multi-sim tests to make this possible without duplication. This is complicated by the way filenames get changed by mdrun -multi by the former par_fn, so cleaned up the way that is handled so it can work and be re-used better. Introduced mdrun integration-test object library to make that build system work a little better. Made some minor improvements to Doxygen setup for integration tests.

Fixes #860, #692, #1857, #1942.

Change-Id: I5f7b98f331db801b058ae2b196d79716b5912b09

**History**

**#1 - 02/01/2011 01:53 AM - Roland Schulz**

This is a copy of Bug 691 created by Mark which I deleted accidentally. I'm sorry! I hope I copied everything correctly (I luckily still had the bug open in a Tab).

**#2 - 02/01/2011 01:57 AM - Roland Schulz**

I agree that this should be improved. I have also observed that his can cause significant overhead.

But may be nstsignalcomm is not the best option. Instead I propose the following alternative:

- If one of the above events happen, MPI_Send is used to send the event to all masters
- The masters use MPI_IRecv+MPI_Test or MPI_IProbe to check whether a signal has been send. This should be very low overhead (non-blocking) so that it could be done every step (not only every nstlist)
- After the masters have send/received an event they use a MPI_Gather to find the largest step number on any core (because this is blocking all wait for all masters to have received the event)
- This largest step number (potentially rounded up to e.g. nstlist) is used as the step number at which the event is carried out (e.g. the checkpoint is written)
- All masters which haven't reached this largest number continue now (after the MPI_Gather has finished) their simulation until they reached this point
- Now that all reached the number, the event is carried out

The only possible bottleneck/disadvantage is that one rank has to send a message to all masters (1-N) in the fist step. This is inherently not scalable but should be fine for reasonable number of masters. This could be improved for very large number of masters (e.g. >128) by using a tree. But it would delay the distribution of the event (and thus also more time spent in calculating the missing steps after the MPI_Gather), because the event could only be forwarded to the next tree node after it has been received by MPI_IProbe/MPI_Test.

**#3 - 02/01/2011 03:53 AM - Mark Abraham**

Roland Schulz wrote:

I agree that this should be improved. I have also observed that his can cause significant overhead.

Thanks for the feedback.

But may be nstsignalcomm is not the best option. Instead I propose the following alternative:
Ideally, non-blocking collective communications could be used, but these are not in any MPI standard (yet - see [http://www.unixer.de/publications/img/hoefler-standard-nbcoil.pdf](http://www.unixer.de/publications/img/hoefler-standard-nbcoil.pdf))

- If one of the above events happen, MPI_Send is used to send the event to all masters
- The masters use MPI_IRecv+MPI_Test or MPI_IProbe to check whether a signal has been send. This should be very low overhead (non-blocking) so that it could be done every step (not only every nstlist)

IIRC, these events are only detected on simulation master nodes, and if so, that simplifies these points. Each master sets up an array of persistent sends to each other master, and persistent receives from each other master (e.g. see [https://computing.llnl.gov/tutorials/mpi_performance/#Persistent](https://computing.llnl.gov/tutorials/mpi_performance/#Persistent)). A master calls MPI_Startall when they have to send a signal, and MPI_Testany at each MD step to see if a signal has been received. MPI_Testany has local completion semantics and should be fast enough to use here. Otherwise, doing it every nstlist/nstsignalcomm steps is also an option. Multiple different signals (even detected on different simulations) get handled in the same way as now. At the right time, the persistent send/receives get refreshed. One downside is that you need a two arrays of requests on each master whose size grows with the number of simulations, but that's not a large cost.

- After the masters have send/received an event they use a MPI_Gather to find the largest step number on any core (because this is blocking all wait for all masters to have received the event)

It's actually an MPI_Allgather - all masters need the same data from every other master.

- This largest step number (potentially rounded up to e.g. nstlist) is used as the step number at which the event is carried out (e.g. the checkpoint is written)
- All masters which haven't reached this largest number continue now (after the MPI_Gather has finished) their simulation until they reached this point
- Now that all reached the number, the event is carried out

  The only possible bottleneck/disadvantage is that one rank has to send a message to all masters (1-N) in the first step. This is inherently not scalable but should be fine for reasonable number of masters. This could be improved for very large number of masters (e.g. >128) by using a tree. But it would delay the distribution of the event (and thus also more time spent in calculating the missing steps after the MPI_Gather), because the event could only be forwarded to the next tree node after it has been received by MPI_IProbe/MPI_Test.

Indeed - and this is the motivation for non-blocking collective communication, so that that kind of optimization can be done at the vendor level, not the application level.

Comparing the approaches:

- The current implementation requires a pseudo-synchronization every nstlist steps because of the collective communication in the signalling mechanism. Latency between event and action is at most nstlist-1 steps. Accumulated latency when no signal occurs can be significant, and roughly inversely proportional to nstlist.
- My suggestion requires a pseudo-synchronization every nstsignalcomm, likewise. Latency between event and action is at most nstsignalcomm-1 steps. Accumulated latency when no signal occurs is roughly inversely proportional to nstsignalcomm.
- Roland's approach synchronizes only after an event is observed in at least one simulation and the signaled received by the rest at the end of their an MD step. In theory a signal could take longer than 1 MD step to be received everywhere. Latency between event and action is at most the number of steps by which the leading simulation is ahead of the lagging simulation plus any latency from sending and receiving the signal, possibly some rounding up to nstlist steps, and is worst when the lagging simulation is the one to notice the event, and so signals and blocks first. Accumulated latency is zero, however.

Roland's approach offers the potential for the fastest action on signals with smallest losses when there are no signals. However, if simulations could really get out of sync, then it could see a significant delay between signal and action. The only scenario I could see where that might happen is when a multi-simulation has some significant heterogeneity (e.g. NPT REMD where the particle density varies and so under PME the computation load would vary across the simulations, unless actively managed - and frankly that deserves a warning from mdrun in the first place!) Perhaps some kind of warning-generating check across the multi-simulations for some kind of interaction density is in order.

Roland's approach did occur to me at one stage, but I was concerned that the N-way non-collective communication would be too bad to use and didn't think it all the way through. The persistent communication requests relieve some of the local cost (and so help with reducing global latency). So I'm leaning towards implementing Roland's approach, assuming my assertion that only simulation master nodes participate in the signalling mechanism holds.

#4 - 02/01/2011 04:59 AM - Roland Schulz
I agree with everything important.

Just a few small comments about communication methods:
1) I don't think it will help a lot when the nonblocking collectives are in the standard. As long as their is no hardware support the implementation will have to have a separate communication thread if it wants to make asynchronous progress. But having either no asynchronous progress (not useful for us) or having a separate communication thread is also possible on the application level. But having a separate communication thread is not ideal. It would be better to just probe more often than every step if traversing the tree is to slow.
2) Using a tree with probing every step is not that slow. If one step is 10ms and you have 128 masters than after 70ms you have completed the tree. And you wouldn't need to use a binary tree and could thus cut this time done even further. For reacting to a signal 70ms(=ln(128)steps) is totally OK and still faster than the nstlist approach is currently. Of course you need to then also finish the remaining steps thus it would be another 70ms and thus slightly more steps than nstlist but still OK.
3) I don't think that persistent communication is helpful in this case because we don't send many events. I don't think you need persistent
communication for making repeated probing faster.

Conclusion: If the 1-N sending pattern ever becomes a problem a tree-based asynchronous application-level collective broadcast is always possible.

Regarding simulations not in sync:
On some system (e.g. Cray) the individual simulations often run at different speeds. This is usually because the network is more or less congested on different nodes.
For non-REMD multi-simulations this means that the asynchronous approach might spend significant time in reaching the same maximum number of steps. But for those simulation I don't see why it would be necessary to synchronize to the maximum number of steps. For those it is fine e.g. to write checkpoints at different number of steps.
For REMD this means, with the suggested approach implemented, one looses a lot of time in the synchronizing for the temperature exchange. Thus it might be interesting to look into algorithm variations of REMD which don't need synchronous exchange.

#5 - 02/01/2011 07:29 AM - Mark Abraham

Roland Schulz wrote:

I agree with everything important.

Just a few small comments about communication methods:
1) I don't think it will help a lot when the nonblocking collectives are in the standard. As long as their is no hardware support the implementation will have to have a separate communication thread if it wants to make asynchronous progress. But having either no asynchronous progress (not useful for us) or having a separate communication thread is also possible on the application level. But having a separate communication thread is not ideal. It would be better to just probe more often than every step if traversing the tree is to slow.

Having non-blocking collectives would permit the current implementation to avoid accumulating lost time, by not synchronizing until the algorithm requires it (e.g. to do replica exchange, or react to a signal); this is because when there is no event, no process blocks. Any plausible way of implementing it (library thread, application thread, application probing) has to be an improvement. Your approach amounts to an application-level implementation of non-blocking collectives at (perhaps) a higher frequency than the current approach and perhaps as using a communication tree. An implementation in a standard would permit us to make only a trivial code change, in the hope that MPI library and hardware support will come along sooner or later (as it might have already for BlueGene).

2) Using a tree with probing every step is not that slow. If one step is 10ms and you have 128 masters than after 70ms you have completed the tree. And you wouldn't need to use a binary tree and could thus cut this time done even further. For reacting to a signal 70ms(=ln(128)steps) is totally OK and still faster than the nstlist approach is currently. Of course you need to then also finish the remaining steps thus it would be another 70ms and thus slightly more steps than nstlist but still OK.

Sure. It would be a total loss on BlueGene's mesh network, however.

3) I don't think that persistent communication is helpful in this case because we don't send many events. I don't think you need persistent communication for making repeated probing faster.

OK. Easy to implement the simple way and see how it goes.

Conclusion: If the 1-N sending pattern ever becomes a problem a tree-based asynchronous application-level collective broadcast is always possible.

True.

Regarding simulations not in sync:
On some system (e.g. Cray) the individual simulations often run at different speeds. This is usually because the network is more or less congested on different nodes.

That can happen on other architectures, too.

For non-REMD multi-simulations this means that the asynchronous approach might spend significant time in reaching the same maximum number of steps. But for those simulation I don't see why it would be necessary to synchronize to the maximum number of steps. For those it is fine e.g. to write checkpoints at different number of steps.

Currently the implementation requires checkpoints to restart at the same step. It is not clear to me why we need this. If not, then synchronization for checkpoints can go away. Perhaps Berk knows something here.

For REMD this means, with the suggested approach implemented, one looses a lot of time in the synchronizing for the temperature exchange. Thus it might be interesting to look into algorithm variations of REMD which don't need synchronous exchange.

I have plans to fix part of this as well. Only neighbouring replicas need to communicate. Currently we have

- a global synchronization of master nodes when they MPI_Allreduce data to test for exchanges,
then an intra-simulation MPI_Bcast to prepare them for possible replica exchange,
then collection of data on the master node,
then exchange of simulation data,
then re-partitioning DD.

Improving the latter three stages would be a nightmare, unless we were to switch to communicating the ensemble data, rather than the simulation data (which AMBER does). Improving the first two stages is easier - adjacent master nodes use MPI_Scatter to a communicator that is the union of their (PP) simulation nodes, and all nodes work to keep the RNG in sync. Now there's only pairwise synchronization, but collecting REMD statistics will have to occur at the end of the run, and demuxing a trajectory will require parsing all the .log files.

There's a bunch of other improvements possible for REMD as well - including preservation of the RNG state across restarts. But that's another thread for another day.

#6 - 02/02/2011 12:34 AM - Mark Abraham

Roland Schulz wrote:

- The masters use MPI_IRecv+MPI_Test or MPI_IProbe to check whether a signal has been send. This should be very low overhead (non-blocking) so that it could be done every step (not only every rlist)
- After the masters have send/received an event they use a MPI_Gather to find the largest step number on any core (because this is blocking all wait for all masters to have received the event)

Actually, there's a problem here. All masters have posted non-blocking receives for all masters. After a simulation master has used non-blocking sends for its signal(s), it has to enter a loop polling for incoming messages and completions of its sends. It must leave the loop when all the sends are complete, and enter a global communication to compare the simulation step. Another master can also be in or entering such a loop. It is chance whether both masters happen to receive the message from each other before the first one observes that all its sends have completed. Unfortunately, all masters need to know not only that they've had all their messages received, but also that they have received all messages for them, so that the corresponding non-blocking sends can complete. However, they can't deduce that there's no message to receive from the absence of a message, so a race condition can ensue. Using blocking sends probably makes the problem more rare, but does not eliminate it.

So, all-to-all gather semantics are required at each signaling stage so that all simulations know the desires of all others. (N-1)*(N-1) single messages are a much larger issue then the N-1 we were envisaging. If these messages are non-blocking, then leading simulations can pile up multiple receives for each lagging simulation - that's a solvable problem, but it consumes memory. Non-blocking MPI_Allgather would simplify the house-keeping, but probably not the cost. If these single messages are blocking, then we may as well be using MPI_Allgather.

I'm leaning back towards my approach.

#7 - 02/03/2011 02:07 AM - Roland Schulz

You are right - I overlooked that.

But I think it is enough if the master of masters (I think it is called sim-master) is allowed to signal. That avoids the problem. If we ever need to allow others to signal they could signal the master which in turn signals all (this can be made easily non-blocking by the master checking for an additional signal from a master before forwarding it to that master.

#8 - 03/09/2011 03:40 PM - Rossen Apostolov

- Affected version - extra info set to current git master

#9 - 03/30/2014 05:25 PM - Mark Abraham

- Affected version set to 4.6

So, there are some aspects of signalling for coordinating operations that I had not appreciated properly.

Signalling within a simulation currently works fine, even if some of the implementation aspects could be improved.

Signalling between simulations that are intrinsically decoupled (e.g. you're using mdrun -multi to run a set of .tpr files that are logically related) should almost never happen. The only thing they actually need to communicate is whether one of the simulations stopping should trigger others to stop. Checkpointing, neighbour-list reconstruction, and counter reset are all issues that are not affected by the existence of other simulations. For example, if a crash happens and only some of the simulations have completed writing the checkpoint at around ~30 minutes, then the set of simulations can still continue - if you'll want them all to have done about the same number of steps, then you'll just have to extend that subset further and no amount of signalling complexity was going to save you from all instances of that kind of situation.

Signalling between simulations that are intrinsically coupled (e.g. replica exchange, or ensemble restraints) should generally only happen when that coupling already has synchronization points (so, at exchange attempts for RE, and probably nstcalcenergy for ensemble restraints). Doing (say) checkpointing in this way automatically ensures that a set of checkpoints ondisk for RE is mutually consistent, because the existing synchronization provides that consistency guarantee.

The latter is easy to implement - with RE we should communicate the "should we checkpoint" signal between simulations immediately before the within-simulation signalling attempt that coincides with the between-simulation synchronization point for the exchange attempt. Ensemble restraints are probably fine similarly, but the between-simulation synchronization point is probably already nstcalcenergy, and probably nobody knows if the
ensemble restraint code still works (I'll bet a week of lunches it doesn't work with more than one rank per simulation).

From here on, we consider only the case of decoupled multi-simulations. When should they stop each other? There is a potential implementation problem because you have to use non-blocking semantics so that you don't unnecessarily couple the simulations. Also, you can't detect that a simulation has stopped from the absence of a message. Sending affirmative heartbeats is all very well, but if the simulations are decoupled, why is a missing heartbeat a problem? And you can't assume that simulation progress is even in lock step (e.g. performance inhomogeneity from network traffic, pressure coupling, or even diffusion).

Ideally, a SIGTERM or SIGHUP should be known to all simulations, but if the MPI implementation doesn't send that to all ranks already, it's not essential that mdrun does that dirty work to propagate between simulations. (N.B. multi-simulation has never been implemented to work with thread-MPI) So we can just let each simulation cope with that on its own.

If all the .mdp files asked for nsteps, then I think we should just do that. So when a simulation reaches nsteps, it can just write final output and block in MPI_Finalize() or something. I think it should be an error for the set of .mdp files to include some positive and some -1 values for nsteps. Maybe mdrun should give a warning if it notices that the range of nsteps is more than 10% of its value, or something. The current implementation assumes lock-step progress (e.g. see code for multisim_nsteps in md.c), which I think is a bug worth fixing (and will fix).

If a simulation detected mdrun -maxh, then the others will notice that condition at least as fast as they'd notice a signal between simulations, so again they don't need to react to each other as well.

Same goes for counter reset, and checkpointing, as far as I can see. Decoupled multi simulations should never be artificially coupled.

That means there is not actually a "market" for inter-simulation signalling at all. Decoupled multi simulations should not be coupled. Replica exchange should write checkpoints only at existing synchronization or exit points (e.g. even the pathological case of using mdrun -maxh, NPT and REMD always writes a self-consistent set of checkpoints, unless one replica notices -maxh while another replica is already blocked in an exchange attempt communication, which can't be solved without a major re-write).

#10 - 03/31/2014 12:50 AM - Roland Schulz
If signals are not send to all processes forwarding them to all simulations is very nice. Otherwise we don't write a checkpoint before the job is killed (without maxh or if job runs shorter than anticipated). But I don't know whether this is common enough to worry about. If we want to support it, it shouldn't be hard. You don't need a hard-beat. You just need to poll from time to time to check whether any messages have arrived. Your approach seems OK to me either way. If we want to support signal forwarding that should be possible to be added as an exception to how inter-simulation communication is handled.

#11 - 05/14/2014 10:41 AM - Rossen Apostolov
- Related to Bug #860: Checkpoint not created upon reaching time given in maxh added

#12 - 06/17/2014 06:01 PM - Erik Lindahl
- Priority changed from Normal to Low

#13 - 12/14/2014 11:20 PM - Gerrit Code Review Bot
Gerrit received a related patchset '1' for Issue #692.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: i5f7b98f331db801b058aae2b196d79716b5912b09
Gerrit URL: https://gerrit.gromacs.org/4312

#14 - 12/14/2014 11:20 PM - Gerrit Code Review Bot
Gerrit received a related patchset '1' for Issue #692.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: i5f7b98f331db801b058aae2b196d79716b5912b09
Gerrit URL: https://gerrit.gromacs.org/4312

#15 - 12/15/2014 09:48 PM - Mark Abraham
- Category set to mdrun
- Priority changed from Low to Normal
- Target version set to 5.1

#16 - 01/08/2015 04:40 PM - Mark Abraham
- Status changed from New to Fix uploaded

#17 - 04/01/2015 07:51 PM - Mark Abraham
Roland Schulz wrote:

If signals are not send to all processes forwarding them to all simulations is very nice.
OpenMPI, Intel and IBM all say they do the kind of signal propagation to all processes that one would expect.

Otherwise we don't write a checkpoint before the job is killed (without maxh or if job runs shorter than anticipated). But I don't know whether this is common enough to worry about.

We've got the regular checkpoint. I don't think we should want to write a checkpoint after a "please terminate" signal - particularly since that has to wait for an NS step, do intra-simulation global communication, and then I/O, which feels like too big an operation for the kind of time windows before MPI systems might send a second signal.

If we want to support it, it shouldn't be hard. You don't need a hard-beat. You just need to poll from time to time to check whether any messages have arrived.

Don't think we need to.

Your approach seems OK to me either way. If we want to support signal forwarding that should be possible to be added as an exception to how inter-simulation communication is handled.

#18 - 06/06/2015 06:37 PM - Erik Lindahl
- Priority changed from Normal to Low

#19 - 06/15/2015 06:11 AM - Mark Abraham
- Target version changed from 5.1 to 5.x

#20 - 05/21/2016 03:16 PM - Mark Abraham
https://gerrit.gromacs.org/#/c/4312/ has some discussion about lack of graceful shutdown upon KILL signal - the fix for #1918 may or may not be relevant

#21 - 05/26/2016 05:20 PM - Mark Abraham
- Related to Bug #1857: -multidir for runs with different number of steps only runs for the shortest number of steps added

#22 - 05/27/2016 12:35 AM - Mark Abraham
- Related to Bug #1942: maxh option and checkpoint writting do not work with REMD simulations added

#23 - 06/21/2016 04:33 PM - Mark Abraham
- Target version changed from 5.x to 2016

https://gerrit.gromacs.org/#/c/5899/11 resolves this issue, along the lines of my comment 9

#24 - 06/27/2016 07:32 PM - Mark Abraham
- Status changed from Fix uploaded to Resolved

#25 - 06/27/2016 07:32 PM - Mark Abraham
- Status changed from Resolved to Closed