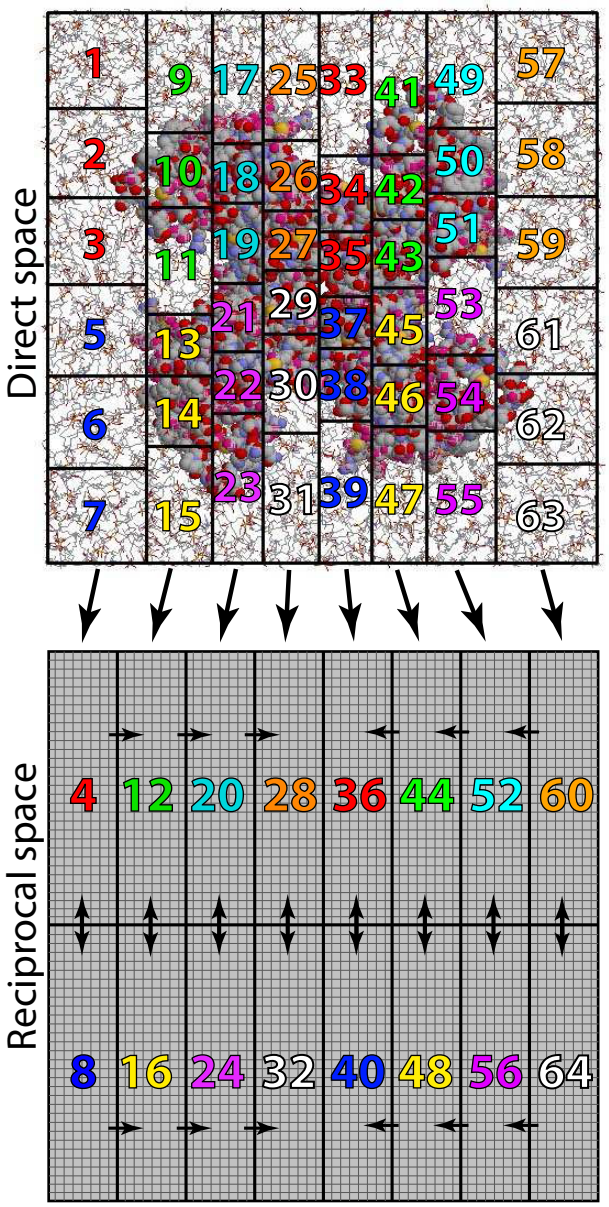
Molecular Simulation Methods with Gromacs



Hands-on tutorial

Making Gromacs go faster

Background

This is a tutorial about various methods to speed up your Gromacs simulations, and introduces some new features of Gromacs 4.6. We'll try to do this using the standard Lysozyme system. This tutorial assumes you have access to a moderately recent computer with a large enough number of cores (say 4 or more) and possibly a GPU to make optimizing parallel runs worthwhile. Although not explicitly covered, the optimizations here are very similar to those you would do with MPI.

The procedures described here are very machine and system dependent, so things that work for my example machine might not work for you, and vice versa.

Setting up Gromacs

You will need a working Gromacs 4.6 installation on your system. Since Gromacs 4.6, the x86_64 kernels (which are the ones you most likely will be using, unless you are running on supercomputers not based on Intel or AMD CPUs) support Intel and AMD's new SIMD (single-instruction, multiple data) instruction set: AVX (short for 'Advanced Vector Extensions). This instruction set allows for explicitly parallel operations on 4 to 8 floating point numbers at once, greatly increasing theoretical CPU performance.

Unfortunately, recent Intel CPUs (since Sandy Bridge) support AVX with full 256-bit instructions, while AMD CPUs support 128-bit instructions at full performance in their Bulldozer instruction set (the 256-bit instructions use registers that are shared with another core, thereby possibly halving performance). In addition, AMD supports a fused-multiply-add instruction that is not supported on Intel, yet is important for performance on AMD CPUs.

Support for these extensions – and previous extensions such as SSE, making the list of x86_64 accelerated kernels: **`SSE2`**, **`SSE4.1`**, **`AVX_128`**, **`AVX_256`** – is now crucial for performance in many parts of the Gromacs 4.6 source code. Yet if a binary has support for AVX-128 (the AMD variant) it will crash with **`Illegal instruction (core dumped)`**. This means that each x86 CPU family will now have to have its own version of Gromacs compiled, and CPU support for accelerations will be auto-detected at compile time.

In addition to this, there is now support for (nVidia) GPUs, in the form of CUDA kernels. These must be compiled in but won't be run unless there is a GPU present (the CUDA libraries must be present, though).

So in short: *compile Gromacs on the system you'll be running on*.

The PDB Structure

The example system we'll be using is the standard **1LYD.pdb**. obtainable from http://www.pdb.org. This is the standard Lysozyme structure used in other tutorials; it has 2885 protein atoms, and 246 co-crystallized water atoms.

Creating a basic Gromacs system

We'll be using the AMBER99SB-ILDN force field, a refinement on AMBER99 with improved backbone ('SB') and side-chain ('ILDN') torsion potentials. See the reference given in **pdb2gmx** for details. To generate a basic topology, use

```
pdb2gmx -f 1LYD.pdb -ff amber99sb-ildn -water tip3p -o start.gro
```

which should generate the topology in **topol.top**. We won't do much with box shapes with this protein because it is roughly spherical: a truncated dodecahedron will be the most optimal box shape. We'll keep neighboring boxes a bare minimum of at least 0.5nm away:

```
editconf -f start.gro -bt dodecahedron -d 0.5 -o box.gro
```

and solvate with

```
genbox -cp box.gro -cs spc216.gro -p topol.top -o solvated.gro
```

This should leave you with a box with 6194 additional water molecules, giving a total of 21471 atoms.

Now we can minimize the energy with some standard minimization parameters

```
------em.mdp------
integrator      = steep
nsteps          = 200
nstlist         = 10
cutoff-scheme   = verlet
vdw-type        = cut-off
rvdw            = 1.0
coulombtype     = pme
rcoulomb        = 1.0
------------------
```

allowing us to run **grompp** with

```
grompp -f em.mdp -c solvated.gro -o em.tpr
```

and we could now run **mdrun**, but first we'll neutralize the system with 50mmol NaCl .

```
genion -conc 0.05 -neutral -p topol.top -s em.tpr -o neutral.gro
```

selecting the **SOL** group for replacement with a few ions. after which we must re-generate the **tpr** with

```
grompp -f em.mdp -c neutral.gro -o em.tpr
```

which allows us to run the energy minimization on a neutral system:

```
mdrun -v -deffnm em
```

Where on machines with large numbers of cores you might need to limit the number of threads, say with **-nt 4**. Next we will equilibrate the water with

```
------equil.mdp------
define          = -DPOSRES
integrator      = md
nsteps          = 2500
dt              = 0.002
nstlist         = 10
rlist           = 1.0
coulombtype     = pme
rcoulomb        = 1.0
cutoff-scheme   = verlet
vdw-type        = cut-off
rvdw            = 1.0
tcoupl          = v-rescale
tc-grps         = protein non-protein
tau-t           = 0.1 0.1
ref-t           = 298 298
Pcoupl          = Berendsen
tau-p           = 1.0
compressibility = 1e-5 1e-5 1e-5 0 0 0
ref-p           = 1.0
refcoord-scaling = all
nstenergy       = 100
constraints     = all-bonds
------------------
```

by running

```
grompp -f equil.mdp -c em.gro -o equil
```

and

```
mdrun -deffnm equil
```

Equilibrating the water around the protein.

The production run

We're now ready to start running the 'production run' which we'll try to optimize. This will be done with the baseline settings:

```
------run.mdp------
integrator      = md
nsteps          = 10000
dt              = 0.002
; Output
nstxtcout       = 100000
nstenergy       = 100000
; VdW
cutoff-scheme   = Verlet
vdw-type        = cut-off
rvdw            = 1.0
; Charges
coulombtype     = pme
rcoulomb        = 1.0
fourierspacing  = 0.12
; Temperature
tcoupl          = v-rescale
tc-grps         = protein non-protein
tau-t           = 0.1 0.1
ref-t           = 298 298
constraints     = all-bonds
------------------
```

where a few things are notable:
- **dt** is 2fs, the maximum safe time with all bonds constrained.
- **cutoff-scheme = Verlet** is the new (for 4.6) potential cut-off treatment, which consists of a standard pair list with buffer for particles that fall just outside it but may be counted before a new pair list is created. This is a standard MD scheme, as outlined in many textbooks (see, for example, Frenkel and Smit, *Understanding Molecular Simulation*). This is the only scheme implemented on GPUs and is intended to replace the old **group** cutoff-scheme without cut-off buffer, which relied on the concept of charge groups to prevent large force fluctuations.
  In addition to GPU support, the Verlet scheme supports OpenMP thread parallelization for improved performance on CPUs (the previous thread implementation 'thread_MPI' or 'tMPI' still exists and performs better between CPU sockets).
  There are a few cases where the group scheme performs better than the Verlet scheme.

Most notably, this is in water-water interactions. For most protein-based systems the Verlet scheme is faster, however.

We are now ready to run the baseline CPU simulation. First preprocess with

```
grompp -f run.mdp -c equil.gro
```

which creates a run-file named **topol.tpr**. Run this with

```
mdrun -maxh 0.04 -nb cpu
```

where the **-maxh 0.04** option tells **mdrun** to run only for 0.04 hours (~2.5 minutes). The **-nb cpu** option makes sure the simulation is run on the CPU only (i.e. not on the GPU). The performance numbers should be visible after the simulation is done: on my 32-core AMD Bulldozer system, I get 35.2 ns/day with these basic settings.

Note that it may be impossible to run the system because the number of threads is too high to decompose the system effectively. You will see an error like this:

```
There is no domain decomposition for 24 nodes that is compatible
with the given box and a minimum cell size of 1.85491 nm
```

The minimum cell size is often given by the requirements of the constraint solver; these can be relaxed somewhat by setting, for example, the **mdrun** option **-rcon 0.7**, although this might lead to crashes (the default value is quite conservative so it's safe to try - it won't affect simulation accuracy).

The file **md.log** should show you a detailed performance analysis. In my case, it is clear that most of the useful CPU cycles are spent calculating real-space forces:

```
Computing:                                  M-Number         M-Flops  % Flops
-----------------------------------------------------------------------------
Pair Search distance check                8178.212202       73603.910     1.0
NxN Ewald Elec. + VdW [F]                53505.085800     3531335.663    49.7
NxN Ewald Elec. + VdW [V&F]                546.111784       58433.961     0.8
NxN Ewald Elec. [F]                      42400.497592     2586430.353    36.4
NxN Ewald Elec. [V&F]                      432.821016       36356.965     0.5
...
3D-FFT                                   76560.955330      612487.643     8.6
```

where the last line is for the PME FFT calculations.

Tuning PME

It is often more instructive to look at the parallelization accounting, which tells a different story, due to communication overhead:

```
Computing:          Nodes   Th.     Count  Wall t (s)     G-Cycles
-----------------------------------------------------------------------------
...
Neighbor search        24    1       1001       5.742      316.990     8.8
Comm. coord.           24    1       9000       0.721       39.788     1.1
Force                  24    1      10001      32.633     1801.420    50.2
Wait + Comm. F         24    1      10001       2.369      130.780     3.6
PME mesh                8    1      10001      24.092      443.306    12.3
PME wait for PP         8                     24.703      454.560    12.7
```

where a few things become clear: force calculation is expensive, and there are 8 PME nodes that spend almost half their time waiting. In fact, this is mentioned in text form in **md.log**:

```
Average PME mesh/force load: 0.693
Part of the total run time spent waiting due to PP/PME imbalance: 5.6 %

NOTE: 5.6 % performance was lost because the PME nodes
      had less work to do than the PP nodes.
      You might want to decrease the number of PME nodes
      or decrease the cut-off and the grid spacing.
```

The first thing we can do is reduce the number of PME nodes. This is limited by the amounts of ways the remaining nodes can be used. The ideal number of PME nodes would be 4 or 5 in my case, but there is no good way of dividing the remaining 28 or 27 cores over the system:

```
There is no domain decomposition for 28 nodes that is compatible
with the given box and a minimum cell size of 1.02425 nm
```

so we'll try 0 PME nodes (dividing the PME work between all the nodes. This actually gives worse performance in my case: 32.7 ns/day.

PME can be further tuned by increasing or decreasing the real-space component while simultaneously increasing or decreasing the Fourier spacing with the **fourierspacing** or **fourier-nx**/**ny**/**nz** parameters. For single-node runs, and especially with GPUs, **mdrun** does a very good job of finding the optimum settings itself: for GPU runs, it does some of the real space vs. reciprocal space tuning automatically.

For high parallelization (over MPI) it may be worth it to spend more time tuning PME through **g_tune_pme**: this tool tries a number of different fourier spacing vs. real space cutoff settings to find the optimal combination combined with an optimum number of PME nodes.

Scaling limits

The limits of parallel performance are set by the number of cores/nodes at which the simulation stops scaling efficiently: adding more won't yield worthwhile performance benefits. Scaling is limited when the amount of communication necessary to run the simulation becomes too big compared to the amount of useful calculation. In general, on CPUs, this limit can be expressed in terms of numbers of atoms per core, and a good rule of thumb for Gromacs 4.6 is that it should be possible to get good scaling down to approximately 200-400 atoms per core.

A few performance optimizations can be done that help reach that limit: there, the energy calculation and center of mass removal start to become expensive. We can perform those less often by adding

```
nstcalcenergy    = 1000
nstcomm          = 1000
```

to the .mdp file. In many situations calculating the energies once every 1000 steps is more than enough. In my case, this yields a very minor speedup to 35.5 ns/day.

Thread pinning

In Gromacs 4.6, there are two different ways of running multi-threaded simulations: thread_MPI, which is also found in Gromacs 4.6, and which mimics the communication of MPI, and the new OpenMP parallelism which divides force calculations, reductions, and PME calculations among tightly coupled threads. Gromacs 4.6' **mdrun** is very good at selecting the right parallelization, but it can be set manually by setting **-ntmpi** or **-ntomp**. In this case on my machine this doesn't lead to better performance, though.

If you'd like to run fewer threads than there are cores, the easiest is to set the total number of threads with **-nt**. Be aware, however, that in that case **mdrun** cannot set thread affinity: it can't 'pin' threads to cores, which causes real performance loss, because if you would run two simulations on the same system, their pinning might clash and you might end up with two simulation trying to run on the same core, while other cores idle.

To safely set thread pinning, use the **mdrun** options **-pin on -pinoffset <n> -pinstride 1**, where **<n>** is the simulation number (starting from 0) that you're starting. If you have a CPU with hyperthreading, you might need to set **-pinstride 2** to avoid placing threads on the same core.

Neighbor list buffers

The Verlet scheme's cutoff can be tuned so that we can get away with fewer neighbor list construction steps. This makes the Verlet scheme's buffer with particles that are in the neighbor list but don't have interactions bigger, however, so it's a tradeoff. Setting

```
nstlist         = 20
```

increases performance slightly (37.2 ns/day in my case) but setting it to 40 decreases performance again to 31.2 ns/day

Virtual sites

Another way of increasing the performance of the simulation would be to increase the step size. This is currently limited by caused by fast vibrations of hydrogens and fast dihedral motions in aromatic rings. In Gromacs, the fastest of these can be removed safely by making the hydrogens and the centers of aromatic rings virtual sites (or **vsites**), removing the fastest degrees of freedom. This allows us to double the step size to 4ns (see the Gromacs manual and references cited therein for details). Practically speaking, the virtual sites must be introduced when the topology is made, so we must re-do the **pdb2gmx** step, and the steps thereafter:

```
pdb2gmx -vsite aromatics -f 1LYD.pdb -ff amber99sb-ildn\
     -water tip3p -o start.gro
```

and repeat the steps necessary to create the box (**editconf**), solvate the system (**genbox**), neutralize the system (**genion**), run the energy minimization and the equilibration. We can now safely set the time step to 4fs in **run.mdp**:

```
dt              = 0.004
```

and run the simulation, which now yields a respectable 67.5 ns/day.

Running on a GPU

Gromacs 4.6 is now capable of running on nVidia GPUs if compiled with CUDA support. If you have CUDA installed, you can check for the presence of GPUs with **nvidia-smi**. When compiled with GPU support, **mdrun** will also detect the presence of supported GPUs:

```
Using 2 MPI threads
Using 16 OpenMP threads per tMPI thread

2 GPUs detected:
  #0: NVIDIA Tesla K20c, compute cap.: 3.5, ECC: yes, stat:
compatible
  #1: NVIDIA Tesla K20c, compute cap.: 3.5, ECC: yes, stat:
compatible
```

GPUs are capable of calculating the real-space force calculations, and have quite high performance doing so. So high that it might require re-tuning of real-space vs. reciprocal

space trade-offs. However, running on GPUs is in many ways easier than on CPUs: for example, this real-space vs. reciprocal space tuning will be automatically done, and the threading options are limited.

On my system, with the above GPUs, I get 126.3 ns/day on one GPU and 119.4 ns/day on both GPUs. In fact, the force calculation is so fast now that it only takes 12% of the (wallclock) time: 47% of the time is now spent on PME, and constraints are starting to become expensive (14%).

Conclusion

In this tutorial we went from 32 to 126 ns/day (admittedly by running on GPUs), showing that real improvements in performance are possible. Whether you can achieve the same is of course very dependent on the system you're simulating, and the hardware you have available.

Of course it's not always about raw single-simulation performance: when doing a free energy perturbation calculation, for example, it's usually necessary to split it up into 10-20 independent simulations. That provides another opportunity for parallelism: those can be run in parallel, independently of each other, giving near-perfect scaling.