Micromanaging Threads for Faster MD Simulations

John D Eblen and Jeremy C Smith
Center for Molecular Biophysics, University of Tennessee and Oak Ridge National Laboratory

Introduction
The past decade has seen a rapid increase in the number of cores on processors, especially processors for HPC (high-performance computing) clusters. To take advantage of this new resource, HPC developers frequently rely on threading to parallelize tasks, such as computational loops, or to run tasks asynchronously, such as overlapping computation and communication code. Common threading frameworks, though, tend to lack flexibility or require unwanted overhead. We present a new threading framework called the Static Thread Scheduler (STS) that addresses this problem. We are currently integrating STS into GROMACS, a popular molecular dynamics package, to replace the existing OpenMP threading and improve performance of MD simulations.

Static Thread Scheduler (STS)
The STS (static thread scheduler) library supports simple, static (as opposed to dynamic), and flexible scheduling of threads for applications that require maximum performance. STS achieves performance by minimizing overhead (static) and allowing full control over thread schedules (flexible).

STS, like OpenMP, can easily parallelize loops. Thus, it is easy to get started with STS or, for simple loop parallelization, to replace OpenMP with STS. For task-oriented parallelization, the programmer creates a “schedule,” a set of assignments of threads to tasks. STS relies heavily on C++ lambdas to simplify the API. The developer creates and names lambdas (tasks) inside the code, places them inside a schedule, and then adds code to assign specific threads to those tasks. This assignment code runs prior to running the schedule.

The diagram to the right shows a sample schedule for computing forces in GROMACS. Each row represents tasks assigned to a single thread. Each color represents a specific task. In GROMACS, computing the nonbonded (NB) forces takes the bulk of the time. Thus, we assign one pool of threads to run NB-related tasks, one pool to run tasks for other forces asynchronously, and a single thread to handle communication. The bottom thread in each of the two pools runs one additional task (black and yellow tasks). These are the main tasks that contain the computational loops. These two threads execute the serial code around these loops, while the other threads only run a portion of the loops themselves. Note the red task, which is shared by all threads (other than the communication thread). Threads can migrate between different main tasks, important for filling otherwise idle time, such as during communication. The next panel discusses this in detail.

GROMACS and STS
GROMACS is a well-known tool for molecular dynamics on HPC machines, with an emphasis on performance and portability. GROMACS parallelizes code at three levels: system-wide using MPI, process-level using OpenMP threads, and at the level of individual instructions using SIMD. The middle level, threading, is the focus of this work. Specifically, we seek to replace OpenMP with STS. Our main focus is on the time-consuming force calculations. Development and testing is primarily done on 2nd generation Intel(R) Xeon Phi™ systems.

Micromanaging Communication and Computation
Our initial attempt at applying STS to GROMACS succeeded in a 25% improvement in force calculations at high (128) thread counts. This initial implementation ran NB asynchronously with the other force calculations and was able to exploit the poor scaling of PME (particle-mesh ewald), which has several communication-bound areas. We are now attempting to build precise thread schedules that overlap communication and computation. The above diagram is similar to the STS block diagram, except that it also shows the time of each task execution (white indicates idle time). It is a snapshot of a single simulation step, using 32 threads with a total runtime of 2.35 milliseconds. The upper threads compute NB (local NB in gray, nonlocal NB in peach, and NB force reduction in burgundy). The lower threads frequently encounter communication gaps, in which only thread 0 (bottom thread) is needed. Previously, the other threads were idle during these gaps. We now schedule them to “help out” with the NB calculations. Thread schedules are adjusted after each time step based on prior results. Note that schedules are static and fixed before each step, so there is no load balancing during the calculations. Both idle time and “overruns” (threads running after communication has ended) can erode performance, so these tuning algorithms executed between steps are key. Today, we are able to reduce the number of threads to eight and still achieve schedules with good balance and little idle time. One current challenge is to make this scheme work with the existing GROMACS dynamic load balancer, which attempts to balance the workload at the MPI level, the step above where STS operates.

Resources
- GROMACS 2016.4 released Sep. 2017
  http://www.gromacs.org
- STS Library
  https://github.com/eblen/STS
- GROMACS with STS
  https://gerrit.gromacs.org (change 5998)

Acknowledgements
- Intel® PCC Program
- Dr. Roland Schulz
- GROMACS Core Developers
- NERSC NESAP
- NERSC Cori KNL