Further Optimization of an Ultracold Neutron Spin Dynamics Simulation Code

Chris Swindell, Adam Holley

Tennessee Technological University

Sources: C/C++ Standard (https://en.cppreference.com/w/)
LLNL (Lawrence Livermore National Lab, https://computing.llnl.gov/tutorials/)

Introduction

Polarized neutrons, or neutrons that are in a single spin state, are vital to many low-energy fundamental physics experiments, such as the UCNτ experiment, which is designed to measure the average lifetime τ_n of a free neutron which undergoes β-decay within a magnetic field. The combination of τ_n with the average lifetime of neutrons that leave the trap by other means ($\tau_{non\text{-}\beta}$) forms the total trap lifetime (τ_{trap}). $\tau_{non\text{-}\beta}$ constitutes a set of systematic effects for the experiment. One of the factors that contributes to $\tau_{non\text{-}\beta}$ is the average neutron depolarization lifetime (τ_{depol}), which is what the simulation that we worked on optimizing is designed to calculate.

$$\frac{1}{\tau_{trap}} = \frac{1}{\tau_n} + \frac{1}{\tau_{non-\beta}} + \frac{1}{\tau_{depol}} + \dots \qquad \textbf{Eqn (1)}$$

Figure 1: Relationship between the average lifetime of a free neutron (τ_n), the average lifetime of all UCNs in the trap (τ_{trap}), and the average lifetime of neutrons that leave the trap through non-β-decay processes ($\tau_{non\text{-}\beta}$). The average neutron depolarization lifetime (τ_{depol}) is a part of $\tau_{non\text{-}\beta}$ and is the primary relationship with which our simulation is concerned.

The Problem

While simulations will never replace real world experiments, they are helpful in understanding the real world equivalents. However, one drawback of using them is that they can be significantly slower compared to running the experiment in real life (i.e. the "simulation clock" runs slower than the wall clock). Figure 2 displays the total run time of 100 neutrons during one instance of our initially Python-based spin dynamics code on our development computer.

```
2019-03-27 08:06:34.731639 Simulating...
Simulations complete.2019-03-28 20:37:40.932914
```

Figure 2: Time stamp output from 100 neutrons in our simulation using the original code. The total run time comes out to: 36 hr, 31 min, 6.2 sec.

A much larger (and more realistic) batch size such as one million neutrons would take over 40 years to complete at that rate. Because of this, optimization of the simulation code was required for better time efficiency.

The Solution

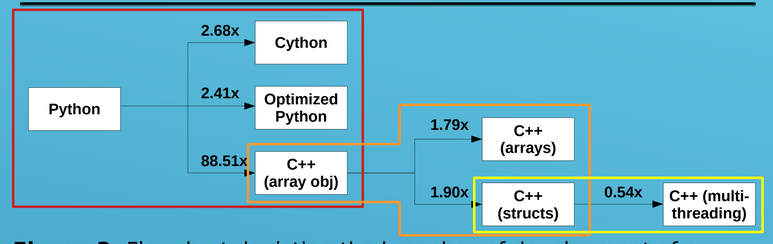

Figure 3: Flowchart depicting the branches of development of our current UCN spin dynamics code. The weight of each link is the speedup factor gained from the previous iteration, which is derived using Eqn (2) in Figure 4. Outlined are the first three variants from the original Python script, the first C++ code and its further optimized derivatives, and the best performing C++ sequential code and its multithreading derivative.

After testing some optimization methods in Python and Cython during previous work, we determined that the best results would come from completely converting the current code from Python to C++. Some optimization was also done preemptively in order to reduce the number of calculations being performed using a method we refer to as "cache variables", which was discovered to give ~2x speedup in Python. For 7 weeks, an extensive process of converting and debugging was conducted, resulting in code that was ~88x faster than the original Python version. Figure 3 shows a flowchart of all the different versions of the spin dynamics code that resulted throughout the entirety of the optimization process as reference. The relative speedup factors were calculated using Eqn (2) from Figure 4.

This first C++ version following the conversion process made heavy use of array objects of the form "array<data type>" from the C++11 <array> library to represent physical vectors. Through some small-scale tests, we discovered that data structures built into the C++ language (e.g. basic arrays and structs) would likely be faster than array objects or any other data structure built on top of such C++ builtin functionalities. Highlighted in Figure 3 is the first C++ simulation code and its subsequent iterations of optimizations described above.

Once we had determined an optimal representation of physical vectors, our next focus was on how our data was being recorded, which had all been done via writing to text files after a simulation had finished execution. Realistically, data needs to be recorded either

The Solution (cont.)

alongside or intermittently during a simulation's run.

To accomplish this, parallel programming was implemented via multithreading, with data recording occurring on a separate thread simultaneously along with a thread running a simulation. Our implementation worked successfully, but resulted in degraded performance, as highlighted in the flowchart in Figure 3. The likely reason for the degraded performance was the use of atomic boolean variables (a "busy-wait" approach to prevent race conditions between both threads) was inefficient. The use of something more efficient like mutexes might prove to be more efficient.

$$\frac{avg\,time_{initial}\left(\frac{s}{\text{sim s}}\right)}{avg\,time_{final}\left(\frac{s}{\text{sim s}}\right)} = Speedup\ factor \qquad \textbf{Eqn (2)}$$

Figure 4: Formula for calculating the speedup factors in the flowchart in Figure 3. In the flowchart's case, the "original" version is the version that came immediately before the "current" version.

Results and Further Work

Conversion of the code to C++ from Python on its own resulted in the largest gains. If a speedup factor of ~2.2 is assumed from the cached variable method as we saw in Python, the conversion to C++ on its own netted us at least ~40x increase in speed when using array objects to represent physical vectors. Converting array objects to something more native to the C++ language like basic arrays or structs netted an additional ~2x speed increase, with structs having slightly better improvement over basic arrays in our simulation. Our implementation of atomic boolean variables with multithreading unfortunately resulted in ~2x degradation in speedup. Another obstacle for later consideration is ensuring resources are still properly allocated for multithreading in combination with brute force parallel processing, which is how these simulations are typically run in production using separate individual instances of a neutron. Our final sequential and parallel code versions had speedup factors of **168.6** and **91.5**, respectively.

This work was supported by the National Science Foundation, grant PHY-1553861