CS 352 HW 6: Parallelism

Due Friday, April 24, 10pm

This assignment demonstrates use of concurrency to accelerate calculation of a simple financial model. This is neither a business class nor a statistics class, so I make no claims about the applicability of this particular model to real life, but the concurrency tools you apply should be applicable to more complex or realistic simulations.

Saving for Retirement

The problem we are going to look at in this assignment involves *compound interest*, in the context of a retirement savings calculator. Here are the main parameters:

- ***annual contribution***: our model assumes a fixed contribution that will be added to the retirement fund every year. This is provided as a floating-point number of dollars.
- ***average interest rate***: the retirement fund will earn interest on both its existing value and the annual contribution each year. This interest varies a bit with market fluctuations, but the average is expressed as a fractional floating-point percentage.
- ***number of years***: how many years until retirement (when the fund will start being drawn down)
- ***standard deviation***: we are going to model variation in annual interest rates with a normal distribution (a bell-curve). The standard deviation is a measure of variability of the interest rate around the average; it is also expressed as a fractional floating-point percentage. Approximately ⅔ of the time the random interest rate will be within one standard deviation on either side of the average.
- ***number of iterations***: this is a *Monte Carlo simulation*. What that means is that for each iteration, we are going to run a randomized simulation of contributions and interest for the specified number of years. We can then collect the results of the full set of randomized iterations and do statistical analyses on them. The more iterations of the simulation we run, the more precisely the aggregated final will model the space of possible outcomes of the model.

In pseudo-code, the basic algorithm looks like this:

```
for each iteration {
  total_savings = 0.0;
  for each year {
    total_savings += annual_contribution;
    total_savings += total_savings * (random interest rate);
  }
  record total_savings as iteration result
}
report 25th percentile, median, and 75th percentile iteration results
```

There is a sequential implementation of this model in each language in the starter code; these implementations handle time measurement, reading the parameters from the command line arguments, setting up the pseudo-random number generators and random distributions, running the simulations, and reporting the results. Your task will be to modify the starter code for each language to create a parallel version.

Tips for Parallelizing

- `#omp parallel for` will split the iterations of a for-loop evenly across threads, but other concurrency mechanisms may require you to explicitly divide the Monte Carlo iterations between the threads.
- Pseudo-random number generators (PRNGs) are (usually) not thread-safe. Make sure that each thread has its own PRNG, and that each thread's PRNG has a distinct random seed (shared PRNGs may break, while PRNGs given the same seed will produce identical series of iterations, weakening the model).
- The vector of results, being shared data, needs mutually-exclusive access. To reduce the need for locking between threads, it may be more efficient to give each thread its own vector, and append into the shared vector once all the threads have finished (an alternate approach would be to pre-initialize the vector and give each thread a range of indices, though this split may be difficult to express in the Rust borrowing system).
- Turning on compiler optimizations is important for the timing results; this is done with the `-O2` flag to the `g++` C++ compiler or the `--release` flag to the `cargo` Rust build system.

Completing the Assignment

As an option to reduce your workload in these stressful times, you may choose to complete only one of the two questions. If you do so, this homework will only count for half-weight in your final grade calculation, with the remaining weight spread proportionately over the rest of the course. If you choose to complete both questions the homework will still count for its usual weight.

Question 1

[12] Modify `retirement_cpp/retirement_par.cpp` to split the Monte Carlo iterations evenly among *N* threads using the OpenMP framework. The starter code also includes `retirement_seq.cpp`, a sequential variant for comparison; the only initial difference between the two files is code accounting for the number of threads.

Given the following model parameters (an example only, modify as you like):

- 4 parallel threads
- $15,000 annual contribution
- 6% average interest rate
- 35 years
- 4% standard deviation in interest rate (⅔ of years will be between 2% and 10%)
- 1,000,000 Monte Carlo iterations (adjust until sequential run takes 10-20 seconds)

`retirement_par` can be compiled and run with the following commands from the directory where the code is[1]:

```
g++ -std=c++14 -O2 -fopenmp -o retirement_par retirement_par.cpp
export OMP_NUM_THREADS=4
./retirement_par 15000 0.06 35 0.04 1000000
```

`retirement_seq` can be compiled and run using the similar series of commands:

```
g++ -std=c++14 -O2 -o retirement_seq retirement_seq.cpp
./retirement_seq 15000 0.06 35 0.04 1000000
```

Question 2

[12] Modify `retirement_rs/src/bin/retirement_par.rs` to split the Monte Carlo iterations evenly among threads using the standard library `thread::spawn` functionality. The starter code also includes `retirement_seq.rs`, a sequential variant for comparison; the only initial difference between the two files is code accounting for the number of threads.

Using the same model parameters as question 1, `retirement_par` can be compiled and run with the following commands from the directory where the code is (note that number of threads is a command-line parameter in Rust):

```
cargo build --release
target/release/retirement_par 15000 0.06 35 0.04 1000000 4
```

The same `cargo build` command will build both the sequential and parallel versions, so it doesn't need to be re-run to compile the sequential version, though note that the sequential version doesn't have a number of threads parameter:

```
target/release/retirement_seq 15000 0.06 35 0.04 1000000
```

Note that you are not expected to understand, reproduce, or modify the `cpu.rs` library shipped with the Rust starter code. It uses Rust's C compatibility features to wrap the same clock used for the C++ timing for consistency of results. For most cases in Rust you would use the standard library `std::time` module.

Style

[3 per question] Your code should not be significantly more complex than necessary, and should be structured in a way that facilitates understanding. Make use of whitespace and comments as appropriate to make your code clearly readable. Function and variable names should clearly represent their purpose or be sufficiently limited in scope to be clear from context.

Extra (no grade – for personal interest)

Compute the *speedup* for your parallel implementation for various numbers of threads. The formula for speedup is *sequential time/parallel time* (computed based on wall time in both

[1] This is the command for a bash shell (Linux/Mac); if you're on Windows, the appropriate command is `set OMP_NUM_THREADS=4` (in cmd.exe) or `$env:OMP_NUM_THREADS = 4` (in PowerShell); also `./` ⇒ `.\`

cases). Your results will be more robust if you take the median time of a small odd number of runs (say 5). Graph the speedup versus the number of threads and consider the following questions:

- What is the *overhead* of parallelism? That is, how much does the speedup fall short of *linear speedup* (equal to the number of threads)?
- At what number of threads are there *diminishing returns* to parallelism? That is, where does speedup hold effectively constant (or decrease) when adding more threads. Do you have a theory about why?

Submission Instructions

Your code each of Questions 1 & 2 should be put into a separate zip file; include all the starter code that was provided in `cs352_hw6_{cpp,rs}_starter.zip`.

The zip files should be submitted on Moodle by the assignment due date.