CS 4444 Introduction to Parallel Computing – Spring 2016
Sequential Program Optimization

Homework #1

Oscar Sandoval

Problem Description and Statement of End Progress:

The goal of this assignment was to optimize the code for an algorithm written in a poor fashion. This assignment will cover the issues of large input sets, function cost calculations, caching, compilation flags, and needless variable calculations. I was able to complete the assignment with working C code by the due date. My variant of the original.c file managed to reduce the execution time of the algorithm by approximately 1.7x on the Rivanna HPC.

Approach:

In order to optimize the original.c code, I employed several techniques from Class Lecture Set 3: Serial Optimization in the following order:

(1) **Avoid Needless rij Calculations:** By changing the cutoff test to (vec2<=(cut*cut)) instead of the original (rij <= cut), where rij used the costly sqrt() function approximately n^2 times, the case test was changed to a cheap multiplication function instead. Only if the case test was proven to be true could sqrt() be used in calculating the rest of the equation.

(2) **Utilize Exponent Rules:** the original algorithm used e^(rij*qi)*e^(rij*qj) as part of the current_e value, which uses two costly exp() functions in its calculation. By combining the powers because both values share the same base (e), the equation was transformed into a single e^(rij*(qi+qj)) calculation

(3) Loop Invariant iConst: I created a variable named iConst, which was equal to the i-1 value used in the inner looping functions. Since this value did not change during the j loop execution, hoisting it outside that loop meant avoid needless calculations of the unchanging value.

(4) Change x^2 to x*x: Since multiplication is essentially a free operation compared to pow(), changing the interior vec2 calculation from x^2 to x*x would save a preposterous amount of time in runtime execution. The same algorithm changed was also performed on the y and z variables of vec2.

(5) Loop Invariant 1/a: As variable a was an unchanging constant declared at the beginning of the code, calculating 1/a before it was used in the equation saved unnecessary recalculations of the same value.

(6) Initial Variable Instantiation: As I was creating additional variables for optimizing original.c, I found that instantiating all variables at the beginning of the code would save the program time utilized in having to create space in memory each time said variable would be used. Instead the variable would always have space reserved in memory for it while only having its value changed during runtime execution.

(7) 2D Array Access into 1D Arrays: As I hypothesized that having the computer traverse a 2D array would be a costly operation, I changed the 2D array holding the X,Y, and Z coordinates to three arrays holding the individual X,Y, and Z coordinates. As the X,Y, and Z coordinates would always be used in tandem with the current i and j values, separating the values into individual arrays would not create incorrect e values when calculated.

(8) Using the -O3 compile flag: The last optimization used as I wished to hand-optimize as much of the code as I possible. The -O3 flag compiles the code using expensive optimizations that may increase program speed and program size

Results:

All tests were conducted with an input.txt file generated with 10,000 coordinate points, the 10 random seed, and 0.5 cutoff.

Handmade Optimizations

Optimization Performed	Execution Phase Runtime (Seconds)
None (gcc original.c -lm -o original)	10.3480
Avoid Needless rij Calculations	9.1111
Utilize Exponent Rules	8.2015
Loop Invariant iConst	8.1583
Change x^2 to x*x	3.0679
Loop Invariant 1/a	2.8852
Initial Variable Instantiation	2.8706
2D Array Access into 1D Arrays	2.6567
Using the -O3 compile flag	1.5153

These runtimes are based on results from my machine. Each descending row's runtime includes the optimizations performed in the previous rows.

OX Flag Comparisons

Optimization Flag Utilized	Execution Phase Runtime (Seconds)
None (gcc original.c -lm -o original)	10.3480
O1 (gcc original.c -lm -O1 -o original)	3.0723
O2 (gcc original.c -lm -O2 -o original)	3.1061
O3 (gcc original.c -lm -O3 -o original)	3.0812
Hand-made Optimizations (without -O3)	2.6567

These runtimes are based on results from my machine. Each row's runtime is based exclusively on a single optimization action.

Rivanna Tests Comparisons

File	Execution Phase Runtime (Seconds)
original.c (with -O3)	1.5700
Hand-made Optimizations (with -O3)	0.8800

These tests were conducting on Rivanna's training queue with the following shell script:

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -t 01:00:00
#SBATCH -o oks2vd_output
#SBATCH -p training
#SBATCH -A parallelcomputing

./original input.txt 0.5
./oks2vdhw1 input.txt 0.5
```

Analysis:

Handmade Optimizations:

As noted by the results section, the biggest changes to runtime occurred when x^2 became x*x and when the O3 flag was introduced. The pow() to multiple function change was expected as it was the base operation to be executed with all values of the input set, and changing that operation to a less expensive one proved to be immensely powerful. The O3 flag provided additional optimization with how my C file was compiled, leading to an amazing combo for the final runtime.

OX Flag Comparisons:

Though the O1 flag lived up to its name in wanting to compile the C file in the shortest time possible, the O3 flag proved to be a worthwhile flag to utilize as well with its additional expensive optimizations. Despite how powerful the optimization flags were alone, my hand-made optimization proved to be the most powerful of all the changes to original.c, proving that lazily-written code cannot be simply be repaired with compiler flags.

Rivanna Tests Comparisons:

Combining my hand-made optimizations with the O3 flag led to a reduction in runtime by almost ½ of the original file with only an O3 flag. The trend in runtime improvements due to the usage of coding optimizations and the O3 flag held true despite the machine that the files were ran on.

Problems Encountered:

To ensure I was calculating the correct values while optimizing the code, I always checked the Num Pairs and Total E values against original.c 's non-optimized values. Though no differences occurred while running optimizations on my machine, comparing the original.c's output values with my optimizations on the Rivanna HPC lead to slightly different values as listed below:

```
Value of system clock at start = 0
Coordinates will be read from file: input.txt
Natom = 10000
cut =     0.5000
Value of system clock after coord read = 0
Value of system clock after coord read and E calc = 1570000
                        Final Results
                        -------------
                  Num Pairs = 13829158
                    Total E = 58261662.7646147534
     Time to read coord file =         0.0000 Seconds
         Time to calculate E =         1.5700 Seconds
        Total Execution Time =         1.5700 Seconds
Value of system clock at start = 0
Coordinates will be read from file: input.txt
Natom = 10000
cut =     0.5000
Value of system clock after coord read = 0
Value of system clock after coord read and E calc = 880000
                        Final Results
                        -------------
                  Num Pairs = 13829155
                    Total E = 58261654.5481551513
     Time to read coord file =         0.0000 Seconds
         Time to calculate E =         0.8800 Seconds
        Total Execution Time =         0.8800 Seconds
```

The Num Pairs of my optimized code cut off three additional pairs compared to the original, leading to a different Total E value. My hypothesis is that changing the cut case from (rij <= cut) to vec2<=(cut*cut)) lead to different comparison values since the sqrt() of rij creates irrational numbers that are different that the finite calculations of vec2.

Conclusions:

The most important lessons drawn from this homework is that the combination of hand-made optimizations and the use of compiler flags leads to necessary runtime improvements for programs wishing to be processed in a parallel fashion. The biggest side-effect to be wary of is having your optimizations change the results of your outputs. Dealing with floating point operations can be tricky given their difficulty in compounding calculations, but awareness of this fact can minimize the change in program outputs to a degree where it's still correct.

Pledge:

On my honor as a student I have neither given nor received aid on this assignment.
Oscar Sandoval

Code Utilized:

- **original.c:** the base file for comparison and the foundation for optimization to be built on
- **generate_input.c:** a program utilized in creating the input.txt file, creates N atom of random coordinates with an additional parameter for random seed number to create the same instance of random coordinates each time
- **input.txt:** in my test cases, N = 10,000 coordinates and charges with a random seed of 10
- o**ks2vdScript:** the shell script utilized in performing the test cases on the Rivanna HPC (code listed under the Results section)
- **oks2vdhw1.c:** my modifications to the original.c file as listed below: