CS 4444 Parallel Computing Spring 2017

Homework 1: Sequential Optimization

Eamon Collins

The code in both opt.c and original.c and their respective binaries work to calculate a consistent value of E. Through purely sequential optimizations I was able to work the execution time down to 6.92 seconds from 18.93 on an input size of 20,000.

Problem Description:

The task in this assignment was to take an input file specifying the coordinates and charges of a certain number of atoms and, given a function to work with, compute a value E as fast as possible. The naive solution was given to us already completed, so the work came in applying optimizations and documenting the speedup that each provided in producing a consistent value for E.

Approach:

The environment on which I ran these was the eqa-cs6414 partition of Rivanna, using that instead of my own computer so that things like concurrent processes or not being plugged in would not affect my results. Also I was hoping that you had generated your input on a Rivanna head node and therefore might get the same results for the same random seed as me, although despite this I was never able to reproduce any of the exact values of E listed in the assignment, even with the unoptimized, original code. I did however get consistent values for E, that is, using the same input file I got the same value of E computed from the original code all the way through to my most highly optimized code.

What I did was simply start by benchmarking the original code without any optimizations or flags, then ran it with progressively higher levels of optimization flags, and then, keeping it at -O3, started adding optimizations to the actual code one by one and recording the execution time produced by each.

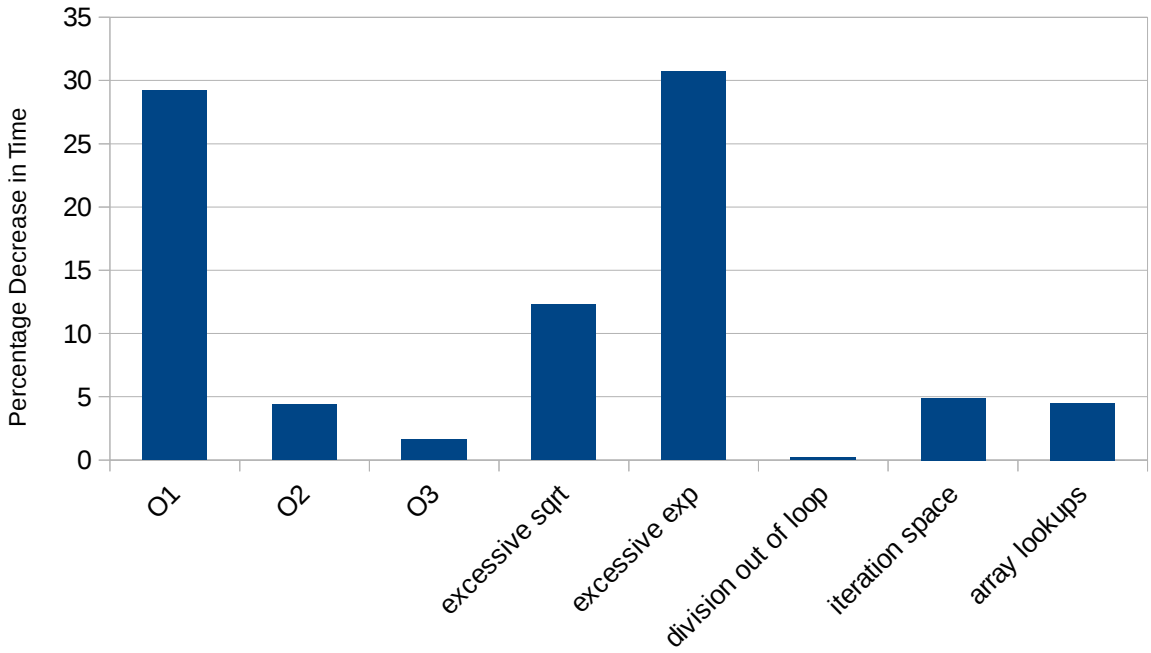
Results:

I was able to cut the time down from a starting point of 18.93 seconds of total execution time to 6.92 seconds, a 63% decrease in time, or a speedup of 2.74. These results were achieved with an input file of size 20000, a seed of 1, and a cutoff value of 0.5.

The execution times for all the levels of optimization are seen below in the table

Optimization level	Time to Calculate E (sec)	Total Execution Time (sec)
None	18.92	18.93
O1	13.37	13.4
O2	12.78	12.81
O3	12.57	12.6
+ remove excessive sqrt	11.01	11.05
+ remove excessive exp	7.62	7.65
+ remove division in loop body	7.6	7.63
+ cleaned up loop iteration space	7.21	7.25
+ factored out array lookups	6.91	6.92

In the graph below each optimization is seen as the percentage decrease in total execution time from the last level level of optimizations. (As each optimization I added was added on top of the previous ones, this is the best way to delineate each one's relative value in decreasing the overall time.)



A brief description of each of the by hand optimizations:
Excessive sqrt: the square root of vec2 was being taken before testing whether it made a cutoff to enter an if loop, I squared the cutoff value outside the loop and compared the original vec2, then took the square root of vec2 as needed once inside the conditional, so that the expensive operation would only happen when necessary.

Excessive exp: there were two calls to exp being made side by side and multiplied with each other, according to simple mathematical rules this can be simplified into one exp call with the arguments being the sum of the arguments of the original exp calls.
Division out of loop: The value of 1/a was being used in a computation in the inner loop body, but the value of a was effectively a constant, set outside both loops. I precalculated the value of 1/a and used that instead in the loop body to replace n^2 divisions with one.
Iteration space: The loops were going from 1 to natom each and each time the iterators were used as indices, one was subtracted from them. In addition, there was an if statement in the inner loop body that restricted anything from happening if j>=i. I simply set the loops to go from 0 <= i < natom and 0 <= j < i and removed the if statement altogether. I then removed the -1 from each of the array accesses using the iterators. This took away a great deal of for loop overhead and conditional testing as the inner loop no longer had to iterate and test until j reached natom every time j began to exceed i. In addition, the small but unnecessary subtractions at each array lookup were removed entirely.
Array lookups: In the inner loop body, during some calculations some array lookups depended solely on i, the outer loop iterator. These values I stored in scalar variables in the outer loop body so that they could be reused on every inner loop iteration after that, avoiding having to look them up unnecessarily each time.

Analysis:

The optimization flags, when taken as a whole, produced the bulk of the speedup. I was not surprised by this as they were crafted by professionals and have been open for review for years, and serve as the backbone for the optimization of much production code. Although the assignment mentions levels beyond O3, the only one that promised a speedup was Ofast, for which the only difference from O3 in c seems to be the addition of -ffast-math, which can cause incorrect results from math functions, which discounted it from my consideration.

The next most valuable optimizations were cutting down on the instances of exp and sqrt calls. This makes sense as these are both expensive operations, and they were cut down by potentially a large amount. In the case of exp, the number of calls was cut down by exactly half.

One common optimization that I didn't have to apply was cache optimizations. Since the arrays were all being accessed along the rows with a stride of 1, the cache hit percentage would be large and no code needed to be altered.

One irregularity I noticed was that occasionally I would get roughly half the execution time I got with the exact same code and optimization flag at other times. (The original code without any flags would get about 8.5 seconds, and my final optimized version about 3.5.) When this happened however, the result for E would remain constant. I suspect this is the result of Rivanna's resource allocation, although nothing I can think of other than SLURM scheduling multiple processes per core at a time would produce this. That also doesn't seem likely as when I run the code locally on my own machine I get results very similar to the times posted above, which are the times I was getting most of the time on Rivanna. Since none of the nodes on Rivanna

in our partition differ greatly in processor speed and the amount of RAM is more than large enough for my purposes, and the cache sizes are the same, I can't explain this difference.

Conclusion:

Although the optimization flags are useful, they can often be substantially helped by thinking about the best way to write the code so as not to waste processing power and memory bandwidth. In particular, factor down complex math operations and keep in mind which operations are expensive relative to others so as to reduce waste, and move as much computation out of loop bodies as possible.

Pledge:

On my honor as a student, I have neither given nor received unauthorized aid on this assignment.
-Eamon Collins

Relevant Files:

benchmark.ods – the spreadsheet I kept track of my benchmarks
input.txt – the input file I used for benchmarking, 20000 with seed 1
input2.txt – another input, size 10000 seed 10
opt.c – My version of the original problem, with all my optimizations included
sub.slurm – my SLURM submission script
Makefile – my makefile that will compile the opt executable
report.pdf – you already found this one.

OPT.C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
double **alloc_2D_double(int nrows, int ncolumns);
void double_2D_array_free(double **array);

int main(int argc, char *argv[])
{
   long natom, i, j;
   long cut_count;

   /* Timer variables */
   clock_t time0, time1, time2;

   double cut;     /* Cut off for Rij in distance units */
   double **coords;
   double *q;
   double total_e, current_e, vec2, rij;
   double a;
   FILE *fptr;
   char *cptr;

   a = 3.2;

   time0 = clock(); /*Start Time*/
   printf("Value of system clock at start = %ld\n",time0);

   /* Step 1 - obtain the filename of the coord file and the value of
      cut from the command line.
            Argument 1 should be the filename of the coord file (char).
            Argument 2 should be the cut off (float). */
   /* Quit therefore if iarg does not equal 3 = executable name,
      filename, cut off */
   if (argc != 3)
   {
      printf("ERROR: only %d command line options detected", argc-1);
      printf (" - need 2 options, filename and cutoff.\n");
      exit(1);
   }
   printf("Coordinates will be read from file: %s\n",argv[1]);

   /* Step 2 - Open the coordinate file and read the first line to
      obtain the number of atoms */
```

```
if ((fptr=fopen(argv[1],"r"))==NULL)
{
    printf("ERROR: Could not open file called %s\n",argv[1]);
    exit(1);
}
else
{
    fscanf(fptr, "%ld", &natom);
}

printf("Natom = %ld\n", natom);

cut = strtod(argv[2],&cptr);
printf("cut = %10.4f\n", cut);

/* Step 3 - Allocate the arrays to store the coordinate and charge
   data */
coords=alloc_2D_double(3,natom);
if ( coords==NULL )
{
    printf("Allocation error coords");
    exit(1);
}
q=(double *)malloc(natom*sizeof(double));
if ( q == NULL )
{
    printf("Allocation error q");
    exit(1);
}

/* Step 4 - read the coordinates and charges. */
for (i = 0; i<natom; ++i)
{
    fscanf(fptr, "%lf %lf %lf %lf",&coords[0][i],
          &coords[1][i],&coords[2][i],&q[i]);
}

time1 = clock(); /*time after file read*/
printf("Value of system clock after coord read = %ld\n",time1);


/* Step 5 - calculate the number of pairs and E. - this is the
   majority of the work. */
total_e = 0.0;
cut_count = 0;
```

```
   //square cut so that taking the sqrt of rij is not necessary later
   cut = cut * cut;
   //take the inverse of a once to remove all the needless divisions inside the loop
body
   double inverse_a = 1.0 / a;
   double qi, coords0, coords1, coords2;
   //set loop bounds to 0 and <natom and took away the extraneous -1 everytime they
were used
   //also took away to if(i < j) branch and simply set the loop bound at j<i
   for (i=0; i<natom; ++i)
   {
        //taking coords lookups dependent on i out of the inner loop so it only has to be
done once per i loop
        coords0 = coords[0][i];
        coords1 = coords[1][i];
        coords2 = coords[2][i];
        qi = q[i];
      for (j=0; j<i; ++j)
      { //pow calls left in instead of x*x because the difference seems to be minimal
              //and it would introduce another array lookup each
            vec2 = pow((coords0-coords[0][j]),2.0)
                 + pow((coords1-coords[1][j]),2.0)
                 + pow((coords2-coords[2][j]),2.0);
            /* X^2 + Y^2 + Z^2 */
            /* Check if this is below the cut off */
            if ( vec2 <= cut )
            {
               //place rij calculation here so it only occurs sometimes
               rij = sqrt(vec2);
               /* Increment the counter of pairs below cutoff */
               ++cut_count;
               //refactored expression, reducing number of exp calls
               current_e = (exp(rij*(qi+q[j])))/rij;
               total_e = total_e + current_e - inverse_a;
            }
      } /* for j=0 j<i */
   } /* for i=0 i<natom */

   time2 = clock(); /* time after reading of file and calculation */
   printf("Value of system clock after coord read and E calc = %ld\n",
         time2);

   /* Step 6 - write out the results */
   printf("                    Final Results\n");
   printf("                    -------------\n");
```

```
    printf("                  Num Pairs = %ld\n",cut_count);
    printf("                    Total E = %14.10f\n",total_e);
    printf("     Time to read coord file = %14.4f Seconds\n",
          ((double )(time1-time0))/(double )CLOCKS_PER_SEC);
    printf("         Time to calculate E = %14.4f Seconds\n",
          ((double )(time2-time1))/(double )CLOCKS_PER_SEC);
    printf("        Total Execution Time = %14.4f Seconds\n",
          ((double )(time2-time0))/(double )CLOCKS_PER_SEC);

    /* Step 7 - Deallocate the arrays - we should strictly check the
       return values here but for the purposes of this tutorial we can
       ignore this. */
    free(q);
    double_2D_array_free(coords);

    fclose(fptr);

    exit(0);
}

double **alloc_2D_double(int nrows, int ncolumns)
{
    /* Allocates a 2d_double_array consisting of a series of pointers
       pointing to each row that are then allocated to be ncolumns
       long each. */

    /* Try's to keep contents contiguous - thus reallocation is
       difficult! */

    /* Returns the pointer **array. Returns NULL on error */
    int i;

    double **array = (double **)malloc(nrows*sizeof(double *));
    if (array==NULL)
        return NULL;
    array[0] = (double *)malloc(nrows*ncolumns*sizeof(double));
    if (array[0]==NULL)
        return NULL;

    for (i = 1; i < nrows; ++i)
        array[i] = array[0] + i * ncolumns;

    return array;

}
```

```
void double_2D_array_free(double **array)
{
   /* Frees the memory previously allocated by alloc_2D_double */
   free(array[0]);
   free(array);
}
```

SUB.SLURM

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=00:15:00
#SBATCH --part=eqa-cs6414
#SBATCH --account=parallelcomputing
#SBATCH --output=output

./original input.txt 0.5
```