CS 4444 – Report for Homework 1

Jerry Sun (ys7va)

1 Problem Description

The goal of this assignment is to optimize a computationally complicated algorithm for a serial processor using several different optimization strategies that were mentioned in class, and to compare their performance along with different optimization flags(-O) for gcc. The main equation is

$$E = \sum_{j<i, r_{ij}<=cut} \frac{e^{r_{ij}*q_i} * e^{r_{ij}*q_j}}{r_{ij}} - 1/a$$

2 Approach

Several different strategies were applied during testing including (The effect of each modification will be further explained in Section Analysis):

- Reduce expensive operations:

  Square root and exponentiation are usually very expensive and time consuming, so we want to minimize those kinds of operations as much as possible.

- Avoid repetitive operations in loop header or memory lookup:

  Typically we want to minimize the unnecessarily repetitive process within the for loop, like function calls and other minor operations.

- Avoid branch prediction causing by if and loop:

  Loop and if statement cause a lot of branch mispredictions, and thus causing stall and swipe based on different assembly languages, so we want to minimize the possible for loop and if statement.

- Avoid memory(array) lookup:

The program usually read memory in a block size. Especially in multidimensional array, we want to minimize the array lookup or at least put the lookup process into linear.

3 Results

Optimization Strategies					Result(s)	
Flags	Expansive Operation	Repetitive call	Branch Prediction	Memory Lookup	Calculation	To
N/A					25.9121	25.9
-O1					3.3636	3.3
-O2					3.3180	3.3
-O3					3.2800	3.2
N/A	✓				6.0292	6.0
N/A		✓			25.4549	25.4
N/A			✓		25.2274	25.2
N/A				✓	25.3950	25.4
-O3	✓	✓	✓	✓	5.2601	5.2
-O3	✓		✓		5.2451	5.2
-O3	✓				2.3043	2.3
-O3		✓			3.2490	3.2
-O3			✓		3.0640	3.0
-O3				✓	3.2504	3.2
-O3	✓	✓	✓	✓	1.9922	2.0
-O3	✓		✓		1.9888	2.0

[1] Given the space of the report, only those representative results are shown in the table.
[2] The metadata of the test can be found in the Section 4 Analysis.
[3] The test result was the average of 10 runs for each version

4 Analysis

In this section, first we will introduce the metadata of this project and then we will further explore each single optimization strategies applied whether it worked or not.

4.1 Metadata

The result above was tested under 2.4GHZ Intel Core i5(single processor, two cores). The size of the corresponding caches are L2 Cache (per Core):256 KB, L3 Cache:3 MB, Memory:8 GB. The version of the compiler is llvm 7.3.0/gcc 4.2.1. Further, all the input is generated by a size of 20000 within a random seed 1, and take a cut off at 0.5.

4.2 Expansive Operations

According to the result graph, we can find it is the most effective optimization applied except the -O flags. In particular, after some experiment we find out there are three changed spots that are useful.

```
// Remove all the pow function use simple multiplication
vec2 = (coords[0][i]-coords[0][j]) * (coords[0][i]-coords[0][j])
         + (coords[1][i]-coords[1][j]) * (coords[1][i]-coords[1][j])
         + (coords[2][i]-coords[2][j]) * (coords[2][i]-coords[2][j]);
```

```
rij = vec2; //remove the original sqrt function
            // Check if this is below the cut off
if ( rij <= cut_new ) { //take cut_new = cut^2 before the whole computation ↩
    process
rij = sqrt(rij);    //apply sqrt only necessary
    continue with rij..
    }
```

```
current_e = (exp(rij*q[i-1] + rij*q[j-1]))/rij; //combine two exp function into ↩
    one
```

There are two major reasons that they can offer so much speed-up in the whole process. First, we greatly reduce the amount of complex computation within the program, and secondly both of those two changes are also function calls, which result into unnecessary assembly lines as well as branch misprediction within the for loop. As we can see in the real code there are several other minor update within the operation side but none of the others has been more effective than the three above.
However, there are some problem within this method as well. In general, within float operations as it is not so sensitive within large numbers. By using the second modification, we find that the precision has changed, as both sides of the comparison has became smaller, leading to a minor loss in final result, but the detraction can be controlled within 1e-6 which we assume can be neglected.

4.3 Unnecessary operations in for loop and array

Usually we want to make the header of the for loop as simple as possible, so that there will be no more unnecessary additional function and operations being called during the execution. There is only one such place here.

```
for (i=0; i<natom; ++i) {       // change from 1 -> natom to 0 -> natom - 1
for (j=0; j < i; ++j) {         //similar as above
/* vec2 = pow((coords[0][i-1]-coords[0][j-1]),2.0)
               + pow((coords[1][i-1]-coords[1][j-1]),2.0)
               + pow((coords[2][i-1]-coords[2][j-1]),2.0);*/
```

```
//We changed it into by reducing one in the start and end point within the for ↩
    loop
vec2 = pow((coords[0][i]-coords[0][j]),2.0)
               + pow((coords[1][i]-coords[1][j]),2.0)
               + pow((coords[2][i]-coords[2][j]),2.0)

...

//Compute with q[i] instead of q[i-1]
```

However, this turns out to be not so effective. We believe the key reason is that within the amount of calculation within a single for loop. The operation inside the array might only cost one single assembly line instead of making some function call or complex calculation which can usually give much better performance boost.

4.4 Number of for loops and if statement

if statements are usually very expensive because of the branch misproduction and its consequences, so we want to minimize such operation as much as possible. It turns out to be rather effective during the experiment.

```
for (i=1; i<=natom; ++i) {
for (j = 1; j < i; ++j) {           //for (j=1; j<=natom; ++j) {
    Execute }}}                     //if ( j < i ) {
```

By doing this we can actually reduce half of the inner for loop and half of the unnecessary if statement. However, this method also doesn't turn out to work so well as usual. This is mainly because, while we are able to reduce so much branch misprediction, the inner loops cutoff can't give much because those are originally ineffective iteration, as there is no real calculation taking place in those reduced for loops.

4.5 memory lookup

Each layers of memory read data in blocks, so we always want to turn the array lookup into a linear way so that we can maximize cache hit and reduce low-effectiveness memory read. Also we want to minimize some repetitive memory lookup.

```
double i1 = coords[0][i];
double i2 = coords[1][i];
double i3 = coords[2][i];
/* for loops, etc. */
vec2 = pow(i1-coords[0][j-1]),2.0)
               + pow(i2-coords[1][j-1]),2.0)
               + pow(i3-coords[2][j-1]),2.0)
```

By doing this, in each iteration for i, we only need to access three value of i only once. However, it also doesn't work out well. Of course, we can change the arrangement of the arrays, making it like natom * 3, so that the read of coords[i][0,1,2] would now be in linear, instead of jumping around. However, this doesn't work out and even resulted into negative effect, taking longer time in execution.

5 Conclusion

The Analysis above gave us some idea about some instincts in strategies for computationally heavy problems. The main focus for such optimization shall always lay in major operations, for instance, the exponential operation and pow functions in this particular problem. The effectiveness of the optimization is closely connected with the effort to improve the performance of the computation itself and reduce unnecessary computations. Other from that, assume the algorithm isn't very poorly designed, those other optimizations might not be able to achieve the effect of the above two.

6 Curious

During the whole assignment, there is one thing in particular makes me curious, that is the failure of the optimization of memory lookup. Based on my understanding of how cache and memory work, if I changed the structure of the whole array from 3*n to n*3, the memory read should be pretty effective(almost no memory will be swiped out without usage). However, it turns out to have zero or even negative effect, which I can't explain. I tried to look up online for related explanation but also didn't get anything useful.

7 Pledge

On my honor as a student, I have neither received nor given help on this assignment.

Signature: _____

8 Appendix

```
/*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Bad coding example 1
! !
! Shamefully written by Ross Walker (SDSC, 2006)
!
! This code reads a series of coordinates and charges from the file
! specified as argument $1 on the command line.
!
! This file should have the format:
!  I9
! 4F10.4   (repeated I9 times representing x,y,z,q)
!
! It then calculates the following fictional function:
!
!              exp(rij*qi)*exp(rij*qj)   1
!    E = Sum( ----------------------- - - )  (rij <= cut)
!         j<i           r(ij)            a
!
! where cut is a cut off value specified on the command line ($2),
! r(ij) is a function of the coordinates read in for each atom and
! a is a constant.
!
! The code prints out the number of atoms, the cut off, total number of
! atom pairs which were less than or equal to the distance cutoff, the
! value of E, the time take to generate the coordinates and the time
! taken to perform the calculation of E.
!
! All calculations are done in double precision.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/

#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);
/* coords=alloc_2D_double(natom, 3); // testing for reallocate array arrangement*/
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]);
    /* fscanf(fptr, "%lf %lf %lf %lf",&coords[i][0],
           &coords[i][1],&coords[i][2],&q[i]); //effort to restructure array*/
}

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;                      /* Total Energy (Output) */
cut_count = 0;                      /* count of included pair */
double a_inverse = 1.0/a;           /* inverse of  a to reduce operation inside for ↩
    loop*/
```

```
    double cut_new = cut * cut;         /*the square of threshold cut to reduce sqrt ↩
        operations*/
    for (i=0; i<natom; ++i)             /*outer for loop iterate through atom */
    {
        for (j=0; j < i; ++j)  /*inner for loop iterate atom before i to avoid double ↩
            counting */
        {
                rij = (coords[0][i]-coords[0][j]) *  (coords[0][i]-coords[0][j])
                       + (coords[1][i]-coords[1][j]) * (coords[1][i]-coords[1][j] )
                       + (coords[2][i]-coords[2][j]) * (coords[2][i]-coords[2][j]);
                /* X^2 + Y^2 + Z^2 using multiplication instead of pow function*/

                if ( rij <= cut_new ) /* Check if this is below the cut off */
                {
                    rij = sqrt(rij);  /* calculate the real distance between two point */
                    ++cut_count;      /* Increment the counter of pairs below cutoff */
                    current_e = (exp(rij*q[i] + rij*q[j]))/rij;
                    /* turn exp(a) * exp(b) into exp(a+b) to reduce exp call*/
                    total_e = total_e + current_e - a_inverse; /*add up to the final ↩
                        result*/
                  } /*if rij <= cutnew */
        } /* for j=1 j<=natom */
    } /* for i=1 j < i */

    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);
}
```