|epcc|

Performance Programming

Aristotelis Kotsomitopoulos
Exam No. B084151

April 1, 2016

Contents

1 Introduction

In this report we are going to demonstrate the whole optimization process for a specific provided code in C language. The aim is to perform a single processor performance optimisation for the *morar* computer platform. The challenge is not only to optimize the code but also to maintain or even improve the code quality. We will detect and exam the performance issues and we will suggest changes to the original code to optimize them. There would be a representation of multiple optimization techniques regarding the execution time (speed), the memory usage and the code readability and quality. We will compare different compilers with different flags in order to achieve an admirable performance optimization. To achieve this we also used multiple performance and profiling tools like PGPROF and CrayPAT's. In addition, we created scripts that where able to automatically run our optimization tests and check that the results where correct after every execution. At first, there would be a demonstration of the main optimization techniques such as compiler, IR level, instruction scheduling, data structures and other optimizations. Subsequently, we will combine those techniques and suggest our final optimizations changes, there would be a detailed demonstration of how we succeed to **reduce the whole programs execution speed from 19 minutes to less than 40 seconds** and at the end we will visually illustrate the overall results.

2 Compiler optimization

In this section we will discuss some of the main compiler optimization techniques like compiler flags and hints. This is one of the most important optimizations in order to achieve the best results regarding memory usage and execution speed. The compiler is likely more skilled than the programmer in deciding which of the available optimization techniques should be used. In addition compilers break codes much less often than humans. The right choice of the compiler flags in combination with the appropriate code modification can produce huge optimizations.

Before we start, the aim of this challenge is to perform a single processor performance optimization in the specific platform Morar. Undoubtedly, we have to check the exact hardware in order to focus our optimization process on it. We simply run the command `cat /proc/cpuinfo` in our machine and we can observe that the CPU model is **AMD Opteron™ Processor 6276**, this model belongs to AMD *Opteron™ 6200 Series* processors so we will specify our optimization flags to that series.

At first we will demonstrate how different compilers with different flags can lead to a huge impact in the performance. We will focus our optimization process using **GCC** and **PGCC** compilers while we observed that **CLANG** compiler wasn't that efficient in this particular code. The given program uses the PGCC compiler with a completely needless flag that specify the type of the target processor `-tp=x` and with the debugging flag `-g` enabled. If we execute the original program, we can observe that needs approximately **18m39.537s** to complete.

2.1 Compiler hints

Compiler hints is a mechanism that can deliver more information to the compiler, in this project we will not use any hint like preprocessor pragmas but we will use macros that will be explained in the section 3.3 Procedure calls optimization.

2.2 Compiler flags

Every compiler has hundreds of flags and options, some of them are made for performance optimization and we have to find out which flags are most efficient with our code. Furthermore, there are flags that can can improve the compile process in a specific kind of hardware. This is not something simple that we can decide by just running tests in the original code, we have to check almost after every code modification the reaction of the code performance regarding to all flags. For both compilers we mainly used flags specialized for the AMD *Opteron™ 6200 Series* processors, for the **GCC** we used:

Flag	Description
`-march=bdver1`	Generate instructions specific to *Opteron™ 6200*
`-O1 -O2 -O3 -Ofast`	Optimization levels (inlining, vectorization)
`-mprefer-avx128`	Advanced Vector Extensions tuning
`-funroll-all-loops`	Enable loop unrolling
`-fprefetch-loop-arrays`	Generate prefetch instructions for loops
`-fprofile-generate`	Enable profile guided optimization
`-ftree-vectorize`	Enable Vectorization
`-ffast-math`	Enable faster less precice math operations

while for the **PGCC** compiler we used:

Flag	Description
`-tp bulldozer`	Generate instructions specific to *Opteron™ 6200*
`-O1 -O2 -O3 -O4 -fast`	Optimization levels (inline, vectorizzation)
`-Mmovnt -fastsse`	Force (disable) generation of nontemporal moves
`-Msafeptr=all`	Override data dependence between C pointers
`-Mfprelaxed`	Relaxed precision in floating point operations
`-Munroll`	Invoke the loop unroller
`-Mvect=prefetch`	Prefetch instructions
`-Mipa=fast,inline`	Interprocedural Optimization
`-Msmartalloc=huge`	Huge pages
`-Mvect`	Enable vectorization

Lets start by just removing the debugging flag `-g` using PGCC compiler, we can immediately observe an **23%** decrease in the whole execution time, in particular from **18m39.537s** to **14m21.090s**

The next flag we will use is `-tp bulldozer` which is one of the most important flags while it specifies the compilation process for the *Opteron™ 6200 series*. Here we can see an extraordinary decrease of **35%** from **14m21.090s** to **9m20.751s**

Lets demonstrate and discuss briefly some of the results in the original code:

PGCC Compiler		
	Compilation Flags	**Execution Time**
1	`pgcc -g -tp=x` (original)	18 minutes 39 seconds
2	`pgcc -tp=x`	14 minutes 21 seconds
3	`pgcc -tp=bulldozer`	9 minutes 20 seconds
4	`pgcc -O2 -tp=bulldozer`	9 minutes 45 seconds
5	`pgcc -O3 -tp=bulldozer`	8 minutes 53 seconds
6	`pgcc -fast -tp=bulldozer`	10 minutes 14 seconds
7	`pgcc -fastsse -tp=bulldozer`	10 minutes 1 seconds
8	`pgcc -Mmovnt -fastsse -tp=bulldozer`	**8 minutes 35 seconds**
9	`pgcc -O3 -Munroll -tp=bulldozer`	10 minutes 1 seconds

First of all, we can observe as we expect that `pgcc -O3 -tp=bulldozer` is really fast with **52%** reduction in the programs execution time.`-fastsse` is a bit faster

than `-fast` because the first is specific for Opteron processors. The **8th** combination of flags has the best results **8m35.965s** with more than **55%** improvement in the whole execution speed. However, these flags are mostly efficient for the original code.

Now if we follow exactly the same process for the GCC compiler we will observe that the best optimization flag is by far `-Ofast` with **9m37.476s** execution time. This flag turns on almost every possible optimization but can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules/specifications for math functions. In our program we tested the results and there is absolutely no problem to use this optimization flag.

Consequently, we will start the optimization process using the most efficient flags for each compiler, for PGCC `-Mmovnt -fastsse -tp=bulldozer` and for GCC `-Ofast -march=bdver1`

3 IR level optimization

First we will analyze and use IR(Intermediate Representation) level optimizations, these optimization are actually the transformations the compiler will make at the source level in order to optimize the program. The majority of them are done by almost every compiler so we will not spent a lot of time on them while there will be not an actual benefit. However, we will focus on some features like *redundancy elimination* and *inlining* that will have a great impact in the execution speed.

3.1 Basic optimization

3.1.1 Constant folding

Lets start our optimization process with something really simple and easy to implement. Constant folding can help the compiler's optimization process by setting predefined hard-coded values rather than variables. In that way the compiler will not try to "guess" all the possible values and combinations but will optimize the code for the certain values. In our program there will be no real benefit because these values don't interact too much with the whole code except **Ndim** that will have a great impact to the performance. The values that we can use constant folding are:

```
double dt=0.02;
int Nstep=100;
int Nsave=5;
```

and of course *Ndim*, for simplicity and code readability we can change these values by using **Macros** (more details in 3.3). Macros in C language have the ability to inform the compiler to replace every appearance of the definitions with the actual constant number, so its exactly the same with hand-writing the numbers. In that way we will maintain the code quality and modifiability:

```
#define dt 0.02
#define Nstep 100
#define Nsave 5
#define Ndim 3
```

3.1.2 Algebraic Simplifications

The compile will simplify every algebraic expression so there is no need to optimize any simple algebraic expression. Even the old compilers are able to do this kind of optimizations. For example we could optimize the division by 2:

```
#define  Npair ((Nbody*(Nbody-1))/2)
```

with right shift:

```
#define  Npair ((Nbody*(Nbody-1))>>1)
```

but there is no point of doing this not only because the compiler will do this for us but in this specific example the Npair calculation will be done only once before replacing every Npair definition with the number (because of the macro definition). However we can observe an algebraic simplification that the compiler might not recognize. The application of this technique must be used only after unrolling the inner Ndim loop (4.1). In the *add pairwise forces* large loop there is an needless variable `collided` that works like a boolean, we can eliminate this variable as well as the branch that follows it and replace it with `collisions++` immediately.

3.2 Redundancy elimination

The goal in this technique is to eliminate redundant computations. For example **Common sub-expression elimination** (CSE) tries to eliminate expressions that are used more than once in the code and replace them with a single one. **Loop invariant code motion** is almost the same and tries to replace commonly used expressions and calculations inside loops and nested loops that can be calculated only once outside the loops. This will reduce considerably the number of instructions, the larger the loops and the iterations the biggest the benefit. In particular, we can modify the *addition of the pairwise forces*:

```
for(i=0;i<Nbody;i++){
  for(j=i+1;j<Nbody;j++){
    Size = radius[i] + radius[j];
    collided=0;
    for(l=0;l<Ndim;l++){
/*  flip force if close in */
      if( delta_r[k] >= Size ){
        f[l][i] = f[l][i] -
        force(G*mass[i]*mass[j],delta_pos[l][k],delta_r[k]);
        f[l][j] = f[l][j] +
        force(G*mass[i]*mass[j],delta_pos[l][k],delta_r[k]);
      }else{
        f[l][i] = f[l][i] +
        force(G*mass[i]*mass[j],delta_pos[l][k],delta_r[k]);
        f[l][j] = f[l][j] -
```

```
        force(G*mass[i]*mass[j],delta_pos[l][k],delta_r[k]);
        collided=1;
      }}
    if( collided == 1 )
          collisions++;
    k = k + 1; }}
```

by storing the expression `G*mass[i]*mass[j]` in the temporally variable `temp` inside the second loop, in that way we will reduce the number of this expression calculation by factor of 6 ($Ndim*2$ where $Ndim = 3$), we can also apply the same technique to move `G*mass[i]` in the variable `G_M` to the first loop so after the transformation the code would look like:

```
for(i=0;i<Nbody;i++){
  G_M = G*Mass[i];          // <-- outer outer variable
  for(j=i+1;j<Nbody;j++){
    Size = radius[i] + radius[j];
    collided=0;
    temp = G_M*mass[j]      // <-- outer variable
    for(l=0;l<Ndim;l++){
      if( delta_r[k] >= Size ){
        f[l][i] = f[l][i] -
        force(temp,delta_pos[l][k],delta_r[k]);
        f[l][j] = f[l][j] +
        force(temp,delta_pos[l][k],delta_r[k]);
      }else{
        f[l][i] = f[l][i] +
        force(temp,delta_pos[l][k],delta_r[k]);
        f[l][j] = f[l][j] -
        force(temp,delta_pos[l][k],delta_r[k]);
        collided=1;
      }}
    if( collided == 1 ) collisions++;
    k = k + 1; }}
```

In the same way we can transform the *central force calculation* to:

```
for(i=0;i<Nbody;i++){
    temp = G_M_central*mass[i];  // G_M_central = G*M_central
    for(l=0;l<Ndim;l++){
        f[l][i] = f[l][i] -force(temp,pos[l][i],r[i]); }}
```

we may use more registers but the advantage of this can be shown by improving the execution speed by **4%**, from **8m35.965s** to **8m12.542s** (using PGCC compiler)

Another optimization that we can use is like redundancy elimination but instead of storing an expression in a variable we can store constant expressions that the whole program re-uses in a macro. In that way the calculation would be done only once without even using extra registers, for instance by adding the following Macro:

```
#define G_M_central G*M_central
```

we will eliminate this calculation completely in the program.

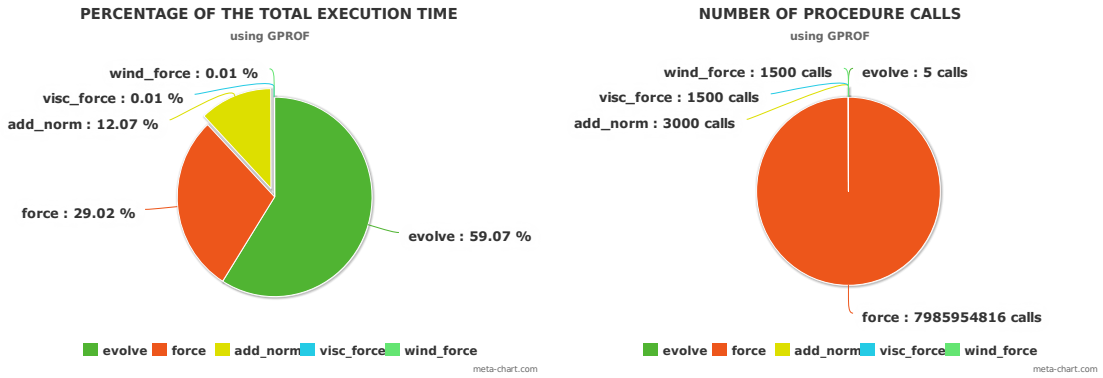
3.3 Procedure call optimizations

In this section we will discuss how we can reduce or even eliminate the expensive procedures. A procedure call can be really expensive (tens of clock cycles). If the procedure's body is small then the calling overhead can be really high. In general there are several ways to optimize the function calls. If the function is small and not commonly used it can be inserted hard-coded in the code but this will affect code *clarity* and *modifiability*. A good solution would be to *inline* the procedures. Inline replaces a call to a function with the body of the function, however, **inline is just a request to the compiler that can be ignored**. Of course, the majority of the compilers have the ability to force inlining by adding the appropriate flags. An even better solution is to use *macros*. **Macros are expanded by the preprocessor before the compilation process start**, so it's just like text substitution where every definition will be replaced with the appropriate function or value. The only disadvantage is that macros are not type checked in comparison with inline functions.

In the provided code we can observe four small procedures (`visc_force, wind_force, add_norm, force`) in the *util.c* file and one really large function (`evolve`) that does the expensive job. Now let's try to analyze the number of calls for every function in order to start the optimization process. We will use the profiling tool **GPROF** *(for GNU GCC compiler)* and **PGPROF** *(for PGCC compiler)*. In order to use those tools we have to insert the compiler flag `-pg` that generates extra code to write profiling information and execute the program again. A file named `gmon.out` would be generated that can produce the following results:

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
59.07	188.90	188.90	5	37.78	64.07	evolve
29.02	281.71	92.80	7985954816	0.00	0.00	force
12.07	320.31	38.60	3000	0.01	0.01	add_norm
0.01	320.33	0.02	1500	0.00	0.00	visc_force
0.00	320.33	0.00	1500	0.00	0.00	wind_force
0.00	320.33	0.00	10	0.00	0.00	second

The most striking feature is the extremely large number of calls for the function `force`, this can be demonstrated by the Figure 1b. It is also clear from this pie chart that the functions `visc_force` and `wind_force` need only 3000 procedure calls while they are not time consuming at all as can be seen in Figure 1a. On the other hand, the function `add_norm` used **12%** of the total execution time. The function `evolve` is really large and uses almost every possible resource and function so we have to examine it further for optimization. Overall, we have to focus our optimization process to the functions `force`, `add_norm` and of course `evolve` and some small modifications to the other two.



(a) (b)

Figure 1: Visual results using GPROF profiling tool

Lets start with the function `force`:

```
double force(double W, double delta, double r){
  return W*delta/(pow(r,3.0));
}
```

it is a really simple calculation with a huge number of **7985954816** calls, so one solution would be to inline it:

```
inline double force(double W, double delta, double r){
  return W*delta/(pow(r,3.0));
}
```

but as we discussed earlier a better solution is to **macro** the function, we delete the function from the *util.c* and we add the following line in the *coord.h* file:

```
#define force(w,d,r)  ((w)*(d))/((r)*(r)*(r))
```

Now we execute again our program, with GCC compiler there no actual benefit (we guess that the $-Ofast$ flag have already inline that function) while with *PGCC* compiler and we can see an extraordinary decrease of execution time by **20%**:

```
real    6m53.084s
user    6m52.530s
sys     0m0.305s
```

Someone may ask the following question: ***"why to use macros instead of just use the compiler flags for inlining?"***. First of all, in large programs with a large number of different functions of course it is easier and sometimes even more efficient to just use the optimization flags for inlining. For example in our program with the inlining flag `-Mipa=inline` we can observe almost the same results (`6m32`) and even better because it is also inlining all the other functions. Now lets see the reasons we prefer **NOT** use this flag in this particular program:

- The function `force` will be divided into two smaller macro functions in order to use redundancy elimination techniques that will be explain in a while

- I will combine and eliminate the functions `visc_force` and `wind_force` not only for the small performance benefit but also for code clarity
- The function `add_norm` will be optimized in a way that there will be no point for the existence of this function (7.1)

As we can see there is no point to use this flag in this specific program. However, it would be really efficient to use this flag when we do not want to change or eliminate the provided functions.

Lets divide the function `force`:

```
#define force(w,d,r) ((w)*(d))/((r)*(r)*(r))
```

to the following two macro functions:

```
#define forcePartB(w,d) ((w)*(d))
#define forcePartA(w,r) ((w))/((r)*(r)*(r))
```

in that way we will be able to store the result of the `forcePartA` in a temporally variable in order to eliminate the recalculation of this expensive expression inside $Ndim$ loop. Sequentially we will use the output of the `forcePartA` function as input for the `forcePartB`, in other words we just use the division first and then the multiplication while this produce the same results.

Here we will combine the functions:

```
/* set the viscosity term in the force calculation */
void visc_force(int N,double *f, double *visc, double *vel){
  for(i=0;i<N;i++)
    f[i] = -visc[i] * vel[i]; }
/* add the wind term in the force calculation */
void wind_force(int N,double *f, double *visc, double vel){
  for(i=0;i<N;i++)
    f[i] = f[i] -visc[i] * vel; }
```

the new function will just combine them into one function, later we will use and explain **loop fusion** and **loop unrolling** techniques in order to eliminate the whole function and optimize it even more (section 4.3). As a result with the new function there will be no actual benefit in the execution time but it will help us for the next optimizations:

```
void visc_wind_force(int N,double *f, double *visc, double *vel,double vel1){
  for(i=0;i<N;i++)  // <-- visc_force
    f[i] = -visc[i] * vel[i];
  for(i=0;i<N;i++)  // <-- wind_force
    f[i] = f[i] -visc[i] * vel1; }
```

4 Loop optimizations

In this section we will demonstrate some techniques that we use to optimize this specific program. Sometimes the compiler needs some help from the programmer in order to make more efficient optimizations.

4.1 Loop unrolling

Loop unrolling is an instruction scheduling optimization technique that increases the size of the loop block, in that way the compiler will have more scope for better optimizations. In addition, branch frequency or penalty will be minimized and there can be a great benefit with the independent instructions that can be reused easier. However, the code may become less readable while may also cause an increase in instruction cache misses and especially when the block is quite large.

Now lets demonstrate how we can apply this technique in our code, we can see for example the *the pairwise separation of particles* and apply a double loop unrolling:

```
k = 0;
for(i=0;i<Nbody;i++){
  for(j=i+1;j<Nbody;j++){
    for(l=0;l<Ndim;l++){
      delta_pos[l][k] = pos[l][i] - pos[l][j];
    }
    k = k + 1;}}
```

This is a triangular array so it would be quite tricky. For the third loop it is simple to eliminate completely the loop because we know the number of $Ndim = 3$. Now the second loop has the problem that every time the number of iterations change from odd to even so if we have a step+=2 in our loop we might lose one iteration. One solution to this is to iterate until $Nbody - 1$ and after these loops are complete we can simply check if the starting value of $j = i + 1$ is odd or even, and we can determine whether should calculate the last iteration separately. If the number is odd then we add the last iteration:

```
k = 0;
for(i=0;i<Nbody;i++){
  for(j=i+1;j<Nbody-1;j+=2){     // <-- step j=j+2
    delta_pos[0][k] = pos[0][i] - pos[0][j];
    delta_pos[0][k+1] = pos[0][i] - pos[0][j+1];
    delta_pos[1][k] = pos[1][i] - pos[1][j];
    delta_pos[1][k+1] = pos[1][i] - pos[1][j+1];
    delta_pos[2][k] = pos[2][i] - pos[2][j];
    delta_pos[2][k+1] = pos[2][i] - pos[2][j+1];
    k = k + 2;  }
  if ((i+1)%2!=0){  // <-- add the last iteration if odd
       delta_pos[0][k] = pos[0][i] - pos[0][Nbody-1];
       delta_pos[1][k] = pos[1][i] - pos[1][Nbody-1];
```

```
delta_pos[2][k] = pos[2][i] - pos[2][Nbody-1];
k+=1; }}
```

while the application of this technique is improving the execution speed of the program, if we want to change the number of $Ndim$ with a larger number it would be extremely hard to change by hand all the unrolled loops, so this modification will work only for this specific number $Ndim$. Using PGCC compiler there is a **4%** reduction in the whole execution time from **6m53.084s** to **6m35.988s** while using the GCC compiler there is an extraordinary **20%** decrease from **8m11.937s** to **6m58.219s**.

We can loop unroll the third loop in the *add pairwise forces* large loop too, the same technique doesn't work for the second loop because the loop block size would be increased too much and this will affect not only the register allocation process but also the code readability. Here we can demonstrate how the two compilers continue to interact completely different to these changes, PGCC compiler reduced the execution time to **6m25.988s** and GCC compiler to **6m19.988s**. We can see how the GCC compiler now is faster than PGCC this is probably because PGCC already did this kind of optimization while the GCC did not, so after the code changes only the GCC has an important impact to the execution time.

4.2 Loop Distribution

Loop Distribution has the ability to reduce capacity misses. The importance of this technique in our code can be demonstrated by applying this technique in the above code (*the pairwise separation of particles*) we used loop unrolling in the previous section. We will distribute the inner loop to three smaller loops one for each Ndim, it is important to reinitialize **k** after every loop, the first of the three loops would look like this:

```
for(j=i+1;j<Nbody-1;j+=2){      // Loop unroll & distribution combination
    delta_pos[0][k1] = pos[0][i] - pos[0][j];
    delta_pos[0][k1+1] = pos[0][i] - pos[0][j+1];
k1+=2; }
```

where **k1** and **k2** will always be initialized with the value of real **k**. Exactly the same is for the other two loops and of course at the end we check again if the number of our iterations where odd or even and add the last iteration:

```
if ((i+1) % 2 != 0){
     delta_pos[0][k1] = pos[0][i] - pos[0][Nbody-1];
     delta_pos[1][k2] = pos[1][i] - pos[1][Nbody-1];
     delta_pos[2][k] = pos[2][i] - pos[2][Nbody-1];
     k+=1;}
```

the results after this technique has a great impact in both compilers, PGCC reduced the whole execution speed by **30%** more and fell from 6m25.988s to **4m49.855s** while GCC compiler fell from 6m19.988s to **4m20.112s** with more than **32%** reduction.

4.3 Loop Fusion

Loop fusion is the opposite of loop distribution, it merges two different loops when they have the same iteration space. This technique sometimes improves the performance and the temporary locality and especially when each loop has not increased data locality. It is really important to note here that the number of the iterations in loops like:

```
for(i=0;i<Nbody;i++)
  for(j=i+1;j<Nbody-1;j+=2)
```

is exactly **Npair** so it is possible to merge loops with Npair iterations with these kind of loops. Now Lets represent the merging of Visc and Wind force calculation. We simultaneously replace the function we created before (*visc_wind_force*) with:

```
for(i=0; i<Nbody; i++){
    f[0][i] = -visc[i] * vel[0][i] -visc[i] * wind[0];
    f[1][i] = -visc[i] * vel[1][i] -visc[i] * wind[1];
    f[2][i] = -visc[i] * vel[2][i] -visc[i] * wind[2];}
```

Figure 2: visc_force and wind_force force optimization

in this way the code is much more readable (we eliminate the function completely) and is only one strait forward loop. The performance advantage is not that much but it will help us to merge more calculations and expression with this loop later (section 7.3). In the same way we can merge the two last loops for updating velocities and positions (for code readability especially):

```
for(i=0;i<Nbody;i++){
  for(j=0;j<Ndim;j++){  // update velocities & positions
    pos[j][i] = pos[j][i] + dt * vel[j][i];
    vel[j][i] = vel[j][i] + dt * (f[j][i]/mass[i]); }}
```

Figure 3: update velocities and positions

4.4 Loop Interchange

Loop interchange has the ability to take advantage of the CPU cache while traversing array elements. Each time an array element is accessed an entire block of data is loaded to cache so on the next array element access the value would be assigned directly from the cache and not from the memory. Loop interchange can help to avoid cache misses while changes the block that would be first loaded in the cache. We applied loop interchange in multiple loops but we conclude that is not important to succeed the best optimization results (that would be illustrated in the next sections). However, a good interchange in our previous code would be to swap the third $Ndim$ loop with the outer i loop. The only code transformation that we need to have correct results is just to set the $k = 0$ at the end of the outer $Ndim$ loop.

4.5 Loop Tiling

Loop tilling was one of the most difficult modifications for this particular program because the two main loops traverse the array in a triangular way. So it was really challenging to apply this technique, we like challenges so we made the transformation. Loop tilling in our code requires to know the correct value of *k* with respect to every *i* and *j* so we used the following equation:

```
k = (n*(n-1)/2) - (n-i)*((n-i)-1)/2 + j - i - 1
```

we can clearly see that $(n*(n-1)/2) = Npair$, so now we know exactly the value of *k* everywhere before we demonstrate the tilling lets do one more optimization to reduce the recalculation of the above expression. We create an array $clever\_k$ and only once before the whole process starts we initialize it (in the *control.c* file):

```
for(i=0; i<Nbody; i++)
  clever_k[i] =  Npair - (Nbody-i)*((Nbody-i)-1)/2 - i - 1;
```

if you look closely you can see that the **+j** is missing from the expression this is on purpose to add the appropriate **j** in our main tilling body loops, we may use a bit more memory but is totally worth it. Variable B is the block size, so our final loop should look like:

```
for (ii=0;ii<Nbody;ii+=B){
  for (jj=ii;jj<Nbody;jj+=B){
  /* check if we are in the diagonal */
    if (ii==jj ){
    /* Here we will traverse each diagonal tile triangular */
      for (i=ii;i<ii+B;i++){
        G_M = G*mass[i];
        Size = radius[i];
        for (j=i+1;j<jj+B;j++){
          k = clever_k[i] + j;
          ....
         }}}
    }else {
    /* if we are not in the diagonal continue with the upper triangular array*/
      for (i=ii;i<ii+B;i++){
        G_M = G*mass[i];
        Size = radius[i];
        for (j=jj;j<jj+B;j++){
          k = clever_k[i] + j;
          .....
        }}}
}}
```

the best results where observed with block size **B=512** but only a small reduction of the execution speed where observed. In particular with the GCC compiler we just observed more stable results over multiple executions while with the PGCC compiler we observed only **4%** reduction of the speed. Finally, loop tilling is not an efficient optimization for our code not only because of the results but also for the side effects in code readability/quality and modification.

5 Data structures optimization

In this section we will analyze and demonstrate how we can improve the code's existing data structures. The right choice and techniques regarding data structures can have a huge impact to the performance.

5.1 Memory allocation

The first problem we can observe is that the memory allocation for `delta_r` and `delta_pos` is wrong and much more than the needed one. We should replace the `Nbody*Nbody` with `Npair` in both allocations. In addition we can observe that the use of **calloc** slow down a bit the program because it zero-initialize the buffer while **malloc** leaves the the memory uninitialized. Both of the functions allocate the array in the heap and we have to manually free them.

In addition, the main problem that slow down the code with this dynamic allocation is the way the 2D arrays are allocated. The whole 2D arrays are allocated in the first pointer and the other pointers just point the correct position for the first one. This results to waste the use of 2D arrays while is like we have one large 1D array with some pointers in it. Sometimes is better to have this implementation but in our case this is a really bad way of allocation. Lets demonstrate visually the already existing dynamic allocation process for `*delta_pos[Ndim]`:

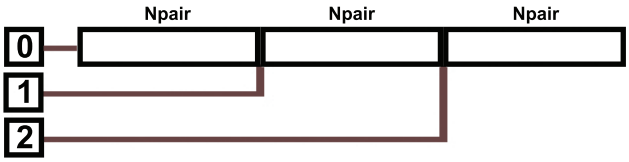


Figure 4: Bad Dynamic Allocation

Now we are going to replace this dynamic allocation with the "original" allocation technique for the 2D arrays, that for every 1D pointer we allocate another array and can visually illustrated by the next figure:

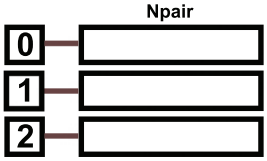


Figure 5: Fixed Dynamic Allocation

so we have to transform the allocation in the *control.c* to the following:

```
r = malloc(Nbody*sizeof(double));          // r will be deleted
delta_r = malloc(Npair*sizeof(double));    // delta_r will be deleted
mass = malloc(Nbody*sizeof(double));
radius = malloc(Nbody*sizeof(double));
visc = malloc(Nbody*sizeof(double));
for(i=0;i<Ndim;i++){
  f[i] = malloc(Nbody*sizeof(double));
  pos[i] = malloc(Nbody*sizeof(double));
  vel[i] = malloc(Nbody*sizeof(double)); // deta_pos will be deleted
  delta_pos[i] = malloc(Npair*sizeof(double));}
```

After these modifications we can immediately observe an extraordinary performance improvement, using GCC compiler there is a **23%** reduction, in particular from 4m20.112s to **3m25.480s** while using PGCC compiler the whole execution time fell **43%!** from 4m49.855s to **2m57.117s**.

An ever better modification is to **static allocate** the arrays, the only disadvantage to this technique is for the program scalability. So with the static allocation the arrays are stored on the stack and we don't need to manually manage the memory. Stack allocation is much faster since all it really does is move the stack pointer. In the section with our final modifications the best implementation is with static allocation, we will also completely eliminate the functions `delta_pos`, `delta_r` and `r` as there is no need to store in memory these values.

5.2 Array padding

Array padding is a technique that adds a small amount of unused space between arrays or between dimensions of arrays in order to reduce conflict misses. In particular we will see in the next sections the extraordinary performance improvement(**40%**) applying this simple modification. For example, an array like `f[Ndim][4096]` can be simply transform to `f[Ndim][4096+64]`

6 Other optimizations

6.1 Initialize arrays with zero

We can replace every zero initialization of arrays with the appropriate use of *memset* function, *memset* has the ability to initialize the whole array at once. For example, we can replace the normal initialization:

```
for(k=0;k<Nbody;k++)
  r[k] = 0.0;
```

with:

```
memset(r,0,Nbody * sizeof(r[0]));
```

in our specific program we will not use this technique because there is no real need for this initialization, details will be demonstrated in **add_norm** optimization (section 7.1).

6.2 Check input data

Another way to improve our performance is to check the input values, for example we can observe that **radius** has the same value in every array position and **mass** has only two or three different values. So one solution would be to check in the *control.c* file while reading the values if there is any array with unique values to simply replace the whole data structure with only one variable or we can use *realloc* to decrease the size of those arrays. This has quite good impact on the performance but it decreases a lot the code readability and clarity.

7 Final version efficient optimizations

In this section we will demonstrate the most efficient optimizations for this code with regard to all the techniques we represented until now. At first we will explain how we optimized the *add norm* function, after that we will illustrate the final structures that we choose and finally we will analyze our main optimization process. This main process is about the whole code in *MD.c* file (inside the *evolve* function) will be divided in two parts, both parts will combine the previous techniques and some new ones in order to achieve the best feasible results. We will also observe that GCC is much more efficient for the following optimizations.

7.1 Distance from central mass (add norm)

Here we will optimize the **calculate distance from central mass** section. First of all, we can observe that this process has three stages:

- Zero initialization for the *r* array
- add the *pos[i][k]* squared to the every f[k] element where *i=0,1,2*
- finally replace every f[k] element with its square root *sqrt(k)*

we can observe that there are so many needless loops (three loops and one of them is inside an *Ndim* loop) and calculations for something that simple. Easily we can combine all those stages in one single loop:

```
for(i=0; i<Nbody; i++)
  sqrt( (pos[0][i] * pos[0][i]) + (pos[1][i] * pos[1][i]) +
        (pos[2][i] * pos[2][i]) );
```

the above code can produce exactly the same results in a more efficient way. Exactly the same technique can be applied for the **calculate norm of seperation vector** where

the only difference is instead of *Nbody* iterations are *Npair* and instead of the *pos* array is the *delta_pos*.

The performance benefit is awesome, after the application of this technique to our previous code we can observe more than **46%** speed improvement using the GCC compiler, especially from 3m25.480s seconds to **1m54.696s**. Now for the PGCC compiler we can see only a **8%** improvement in the whole execution speed (2m47.112s).

7.2 Final structures optimizations

Now in order to achieve the maximum results with the next optimizations (*I&II*) we will **change from dynamic to static allocation** for all the arrays. Furthermore in every 2D array we will **add a padding of *+64*** at the second dimension like we saw in the section 5.2. Finally, we will **define the value of Ndim** like Npair and Nbody. These optimization are really important in order achieve the following results.

7.3 Final optimizations I

The main problem that we can observe here is that there is absolutely no need to store the values of *distance from central mass* (**r**) and of the *norm of separation vector* (**delta_r**) while they are just used only one time at the next loop of each one. The elimination of these arrays will produce a great advantage to the memory and cache that the program uses while the code would be much more smaller and readable.

In this section we will combine and optimize the following code parts:

- set the viscosity term in the force calculation
- add the wind term in the force calculation
- calculate distance from central mass
- calculate central force

First of all, we can observe that the array **r**(*distance from central mass*) is only needed to be subtracted from the array **f** in the *central force calculation* so we will completely eliminate the array **r** and calculate the required values exactly before their usage. So now we can reinforce the optimization we did for the *visc* and *wind* force calculation (section 4.3, figure 2) by subtracting the apropriate **r** value. In combination to that we will use the *forcePartA* and *forcePartB* we discussed in the section 3.3. Finally all the above code parts can be replaced by this:

```
for(i=0; i<Nbody; i++){
  temp = forcePartA(G_M_central*mass[i],sqrt( (pos[0][i] * pos[0][i])  +
                                              (pos[1][i] * pos[1][i])  +
                                              (pos[2][i] * pos[2][i]) ));
  f[0][i] = -visc[i]*vel[0][i]-visc[i]*wind[0]-forcePartB(temp,pos[0][i]);
  f[1][i] = -visc[i]*vel[1][i]-visc[i]*wind[1]-forcePartB(temp,pos[1][i]);
  f[2][i] = -visc[i]*vel[2][i]-visc[i]*wind[2]-forcePartB(temp,pos[2][i]);
}
```

7.4 Final optimizations II

In this section we will optimize the following code parts:

- calculate pairwise separation of particles
- calculate norm of separation vector
- add pairwise forces

Similar with the above section there is no need to store the *norm of separation vector* (**delta_r**) while it is only used to calculate the force (**f**), we can eliminate it completely and instead of this to do the calculation only before we need it. Furthermore, we also don't need the *separation vector for each particle pair* (**delta_pos**) because it's only usage is to calculate again the force **f** and the **delta_r** array. Lets see the code:

```
f1=0; f2=0; f3=0;k=0;
for(i=0;i<Nbody;i++){
  G_M = G*mass[i];
  Size = radius[i];
  for(j=i+1;j<Nbody;j++){
    my_delta_r = sqrt( ((pos[0][i] - pos[0][j]) * (pos[0][i] - pos[0][j]))  +
              ((pos[1][i] - pos[1][j]) * (pos[1][i] - pos[1][j]))  +
              ((pos[2][i] - pos[2][j]) * (pos[2][i] - pos[2][j])) );
    k1=copysign(1,my_delta_r -Size - radius[j]); // Instead of IF!!
    temp1 = forcePartA(G_M*mass[j],my_delta_r);
    temp = k1 * forcePartB(temp1,pos[0][i] - pos[0][j]);;
    f1-=temp;
    f[0][j] = f[0][j] + temp;
    temp = k1 * forcePartB(temp1,pos[1][i] - pos[1][j]);
    f2-=temp;
    f[1][j] = f[1][j] + temp;
    temp = k1 * forcePartB(temp1, pos[2][i] - pos[2][j]);
    f3-=temp;
    f[2][j] = f[2][j] + temp;
    k++;}
  f[0][i]+=f1; f[1][i]+=f2;f [2][i]+=f3;
  f1=0;f2=0;f3=0;}
```

The first we can observe between the two loops is the use of the redundancy elimination technique that we have already seen in section 3.2. Lets start with the important optimizations, at the first operation inside the loop we calculate the **delta_r** value and store it in *my_delta_r* variable (exactly as we explained in section 7.1). In order to calculate **delta_r** we need the **delta_pos** but we can easily see that $delta\_pos[l][k] = pos[l][i] - pos[l][j]$ so we can just replace the *delta_pos* with $pos[l][i] - pos[l][j]$. After the calculation of **my_delta_r**, is following the if replacement trick.

Lets see how we eliminate the following if branch:

```
if (delta_r[k] >Size + radius[j])
```

We use the the *copysign(x,y)* function which simply returns a value with the magnitude of x and the sign of y and we store it in the variable **k1**, so in our code we inserted the values $x = 1$ and $y = my\_delta\_r - Size - radius[j]$. The value of **y** is because:

$delta\_r[k] > Size + radius[j] \iff delta\_r[k] - Size - radius[j] > 0$. So now we can determine whether the branch statement is true or false from the value of $k1$. If **k1** is positive ($k1 = 1$) means that we are inside the branch otherwise ($k1 = -1$) in the else statement. The only difference between the two branch blocks (true/false) is that the addition in the variable **f** is with opposite signs. So we have to simply multiply the $k1$ with the appropriate value of $force$ and then continue with the addition. Again with the technique we explained in section 3.3 we store the first part of the force calculation in the variable $temp1$ that requires the value of my_delta_r we already calculate. After that, we calculate the second last part for the function $force$ that needs both the **k1** and the $pos[l][i] - pos[l][j]$ (***delta_pos[l][k]***), and add this to the apropriate $f[l][j]$ value. We simply do exactly the same for every Ndim dimension (this requires to unroll the inner $Ndim$ loop we discussed at the end of the section 4.1).We can also observe that we used again the redundancy elimination technique to avoid traversing the array $f[i][j]$ for every j iteration.

Finally, we finish our optimization process by simply updating the positions and velocities after the above loop (section 4.3, figure 3).

7.5 Results

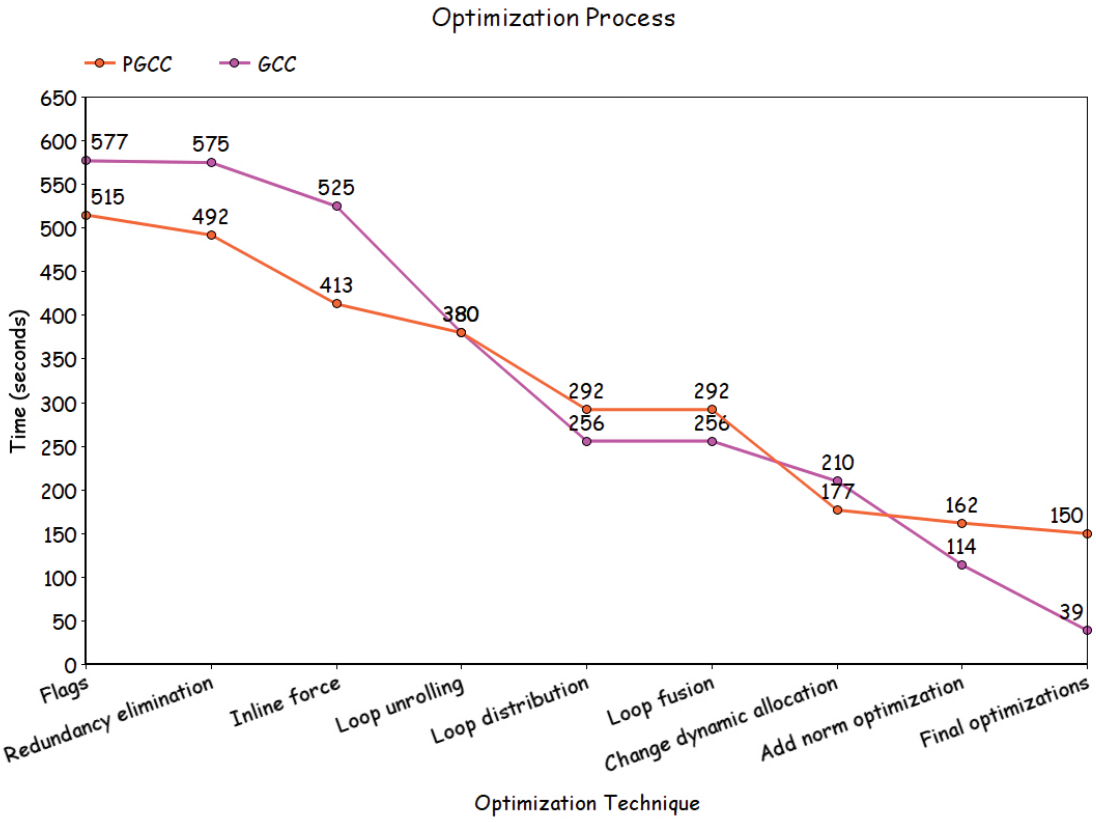
The results with above final optimizations are extraordinary using the **GCC** compiler. We can see an overall execution time of **0m39.956** while each timestep can be executed in less than **8** seconds:

```
100 timesteps took 7.912661 seconds
200 timesteps took 7.912733 seconds
300 timesteps took 7.918800 seconds
400 timesteps took 7.908377 seconds
500 timesteps took 7.910347 seconds
```

we also tested the final code to reassure that the majority of the loops where vectorized using the flag `-fdump-tree-vect-all=<filename>`. From the results we can also demonstrate the huge difference between the two compilers. We focused our final optimization process specific for the GCC compiler thats why for PGCC the results are not that efficient (more than 2 minutes for the whole execution time). We conclude that using the last optimizations in combination with the GCC compiler with the flags `-Ofast -march=bdver1` produce one of the best feasible optimizations for this program.

8 Conclusion

We have presented the full optimization process we followed to achieve our results. Every optimization was tested and compiled with both compilers using different flags, while we used multiple tools to measure the speed and memory performance. We used profiling tools like PGPROF and CrayPAT's to check the vectorization process and the Assembly code modifications. We identified the main problems of the original code and we suggested detailed solutions to overwhelm them regarding memory, speed and code readability. In addition to that we demonstrated the differences between the compilers. We observed that applying the appropriate code modifications, the compiler may have huge improvement in the optimization techniques that chooses. In addition, it is really important to focus our instructions and flags to the specific hardware and platform we are using. Finally, in the following diagram we illustrate the whole optimization process from the compiler flags to our final optimizations *(original code was **1115** seconds)*:



9 Useful Links

GCC Optimization Options and Fags
PGCC Optimization Options and Flags
Optimization of Computer Programs in C
Specific Opteron 6200 series Optimization Flags