PARALLEL PROGRAMMING
C PROGRAMMING AND PERFORMANCE ENGINEERING – LAB ASSIGNMENT

N-BODY SIMULATION USING ALL PAIRWISE FORCES

Authors: Lawrence Adu-Gyamfi & Maximilian Grunwald

Date: 05/11/2018

Table of Contents

1 INTRODUCTION

The baseline code is meant as a solution to the N-body problem, which calculates the forces exerted on bodies in 3-D space by each other. The net force exerted on each of the bodies is computed over time steps and the position of the bodies are updated concurrently.

1.1 Baseline Code Description

The initial positions and velocities of the bodies are randomized at the start of the program with the ***randomizeBodies*** function.
The main work of the computation is performed by the ***integrate*** function which computes the forces exerted on each of the bodies (***bodyBodyInteraction*** function) and subsequently calculates the resulting position and velocity over the predefined time step. The computation is repeated over the next time steps based on the number of iterations.

1.2 Algorithmic and Memory Complexity

In determining the algorithmic and memory complexity, we focus mainly on the loops which result in the most part of the final instructions. The integrate function has 3 main loops (for loops) with one nested in another. The third loop is executed separately afterwards. The loops are repeated for the number of bodies n.
Finally, when the program is executed, the main function runs these loops for the number of iterations. For large values on n, we can assume that the operations run with the nested loop is sufficiently larger than any of the additional instructions. As a result, we can safely arrive at the computational and memory complexity of the program using only the nested loop and the number iterations.
Below is the final computational (algorithmic) complexity of the program:

O(n*n*iter) operations.

For each iteration, the n*n array will have to stored, so we can assume the memory complexity also as:

O(n*n) cell values

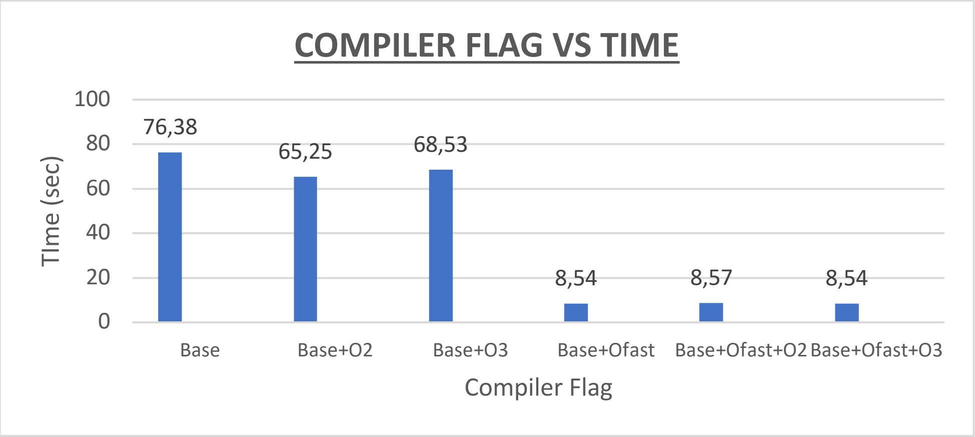
2 EXPERIMENT METHODOLOGY AND RESULTS

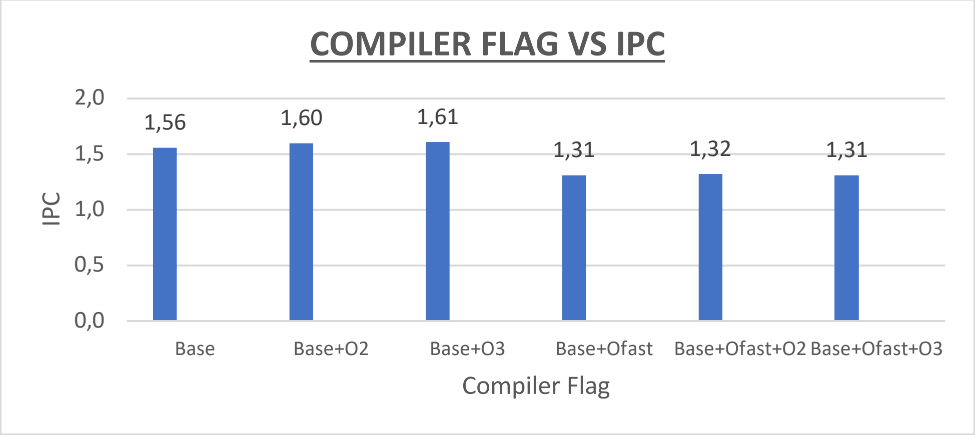
The initial part of the experiment has been performed using only the baseline code. This part of the experiment focused only on the effect of the using additional compiler flags, as well as the effect of the problem size (number of bodies). In both these cases the performance of the program has been computed.

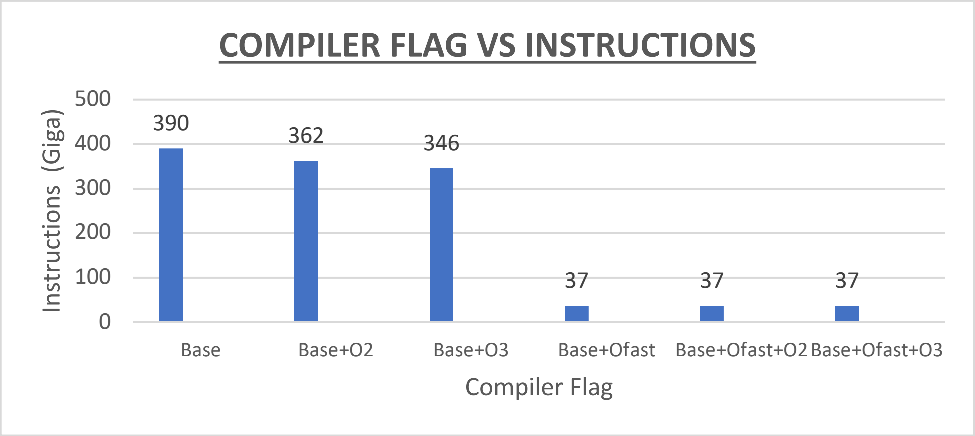
2.1 Compiler Flags

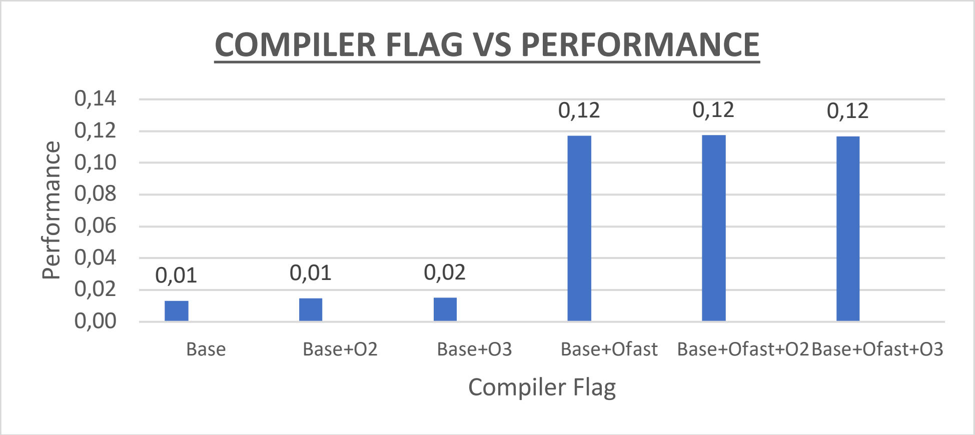
The compiler flags considered for check in the experiment are O2, O3, and Ofast. The impact on performance of using these flags are measured against compiling the code without any flags. This is expected to reduce the computation time. The problem size considered for this section of the experiment is n=10000 bodies and for 10 iterations.

The findings from this experiment are presented below.







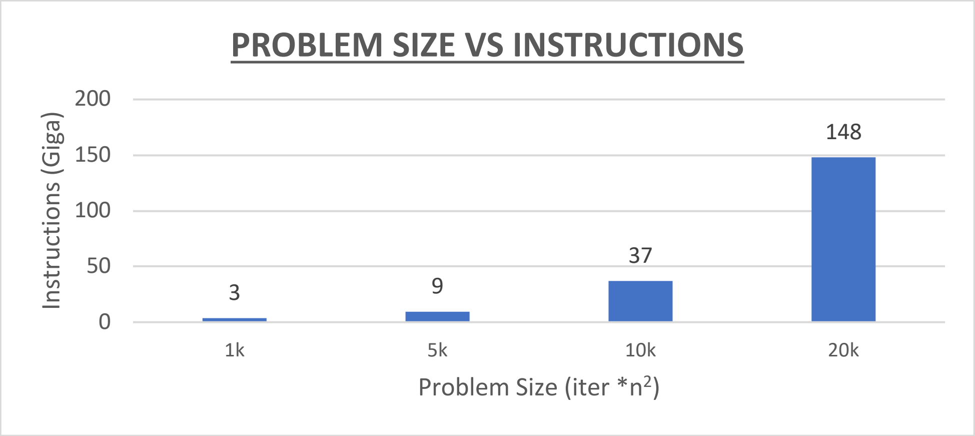


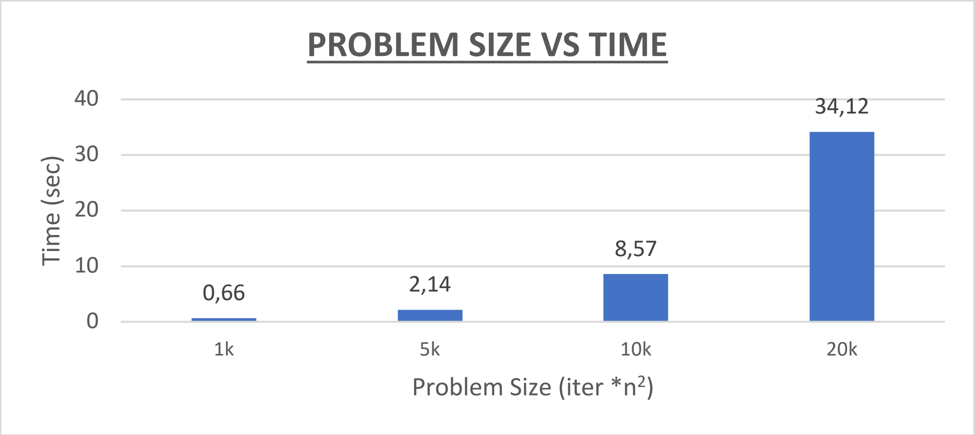
We notice a drastic change in all the results right after the introduction of the Ofast flag. This is consistent in the results for the metrics measured. Afterwards, implementing the Ofast flag with either of the O2 or O3 does not seem to cause much change. We can conclude that the Ofast flag has a higher impact on the code efficiency and performance than the other in this case. And this is explained by the fact that the Ofast flag goes for speed at the expense of accuracy. However, in the execution of this program, the Inclusion of the Ofast flag did not impact the functional quality.

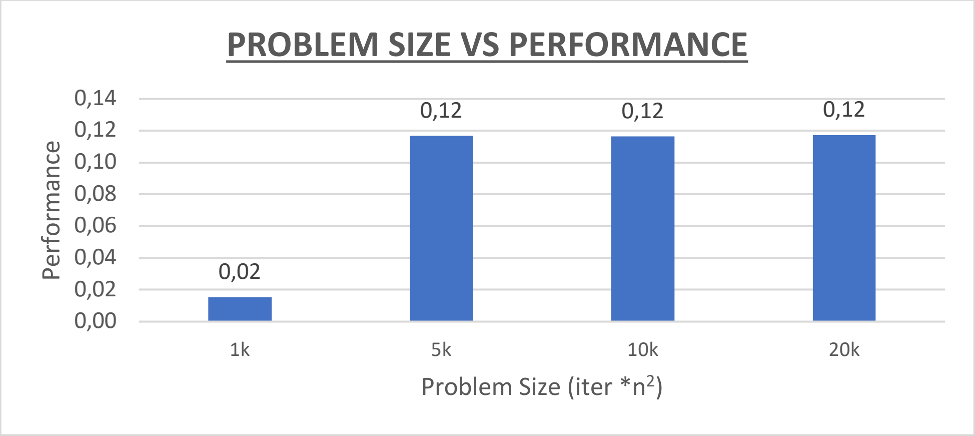
2.2 Problem Size.

The problem size (n) has been varied between 1000, 5000, 10000 and 20000 bodies, and for all these variations, the resulting metrics have been tracked and used to compute the performance. The Ofast and O3 compiler flags were used during this part of the tests.

The results of the experiment are presented below.







Looking at the impact of the problem size on the time of execution and the quantity of instructions, we can conclude that these metrics increase quadratically depending on the problem size. This makes sense based on the algorithmic complexity of our program - O(n^2).

However, aside the case of n=1000, we do not notice much difference between the performance calculated for each of the different problem sizes.

2.3 Code Modification

Another major aspect of the experiment included making several code changes to the baseline code and tracking several metrics in order to measure the effects of these changes on the performance. For this section of the experiment, a problem size of 10000 bodies run for 10 iterations, has been used as it provided the best balance between enough time to run the program and obtaining the desired performance metrics.

The main metrics tracked include; CPU-Cycles, machine instructions, the time taken to complete run of the program and the clock frequency. Finally, the performance of the program which is the eventual ultimate metric is computed.

The performance defined as the **operations/second** is calculated based on the following formula:

operations/second = (operations/Instructions) * (Instructions/Clock cycle) * (Clock cycles / Second).

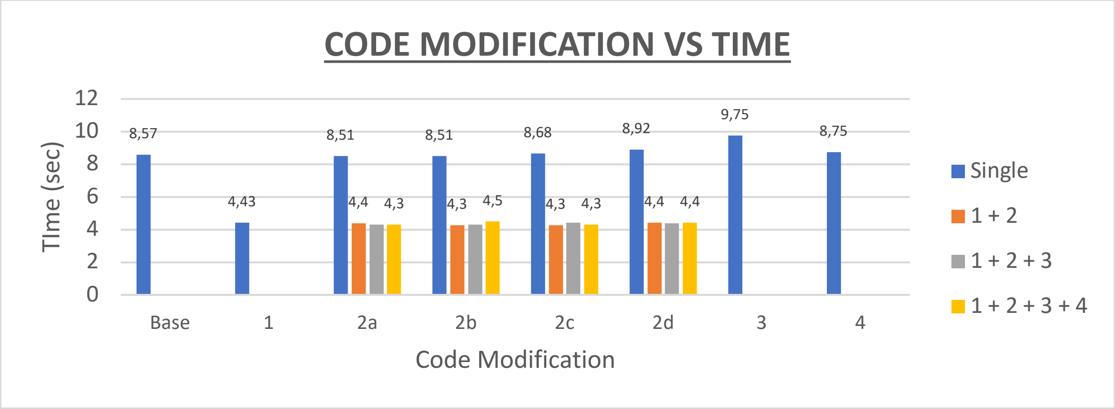
Operations is calculated from the problem size and number of iterations. For the purpose of this experiment, the **operations = 10000 * 10000 * 10.**
Instructions – Obtained from output of perf profiler
Clock cycle – obtained from output of perf profiler
Time - Obtained from output of perf profiler.

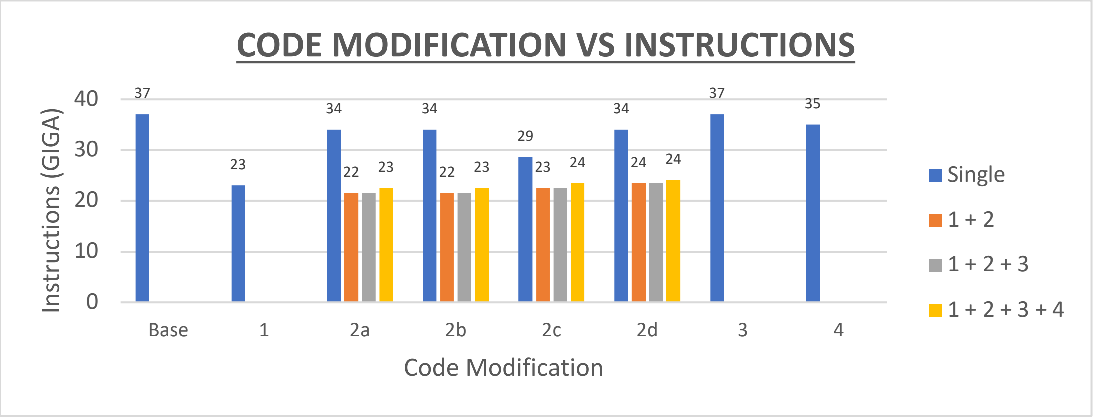
The following code modifications(as per reference [3]) were implemented:

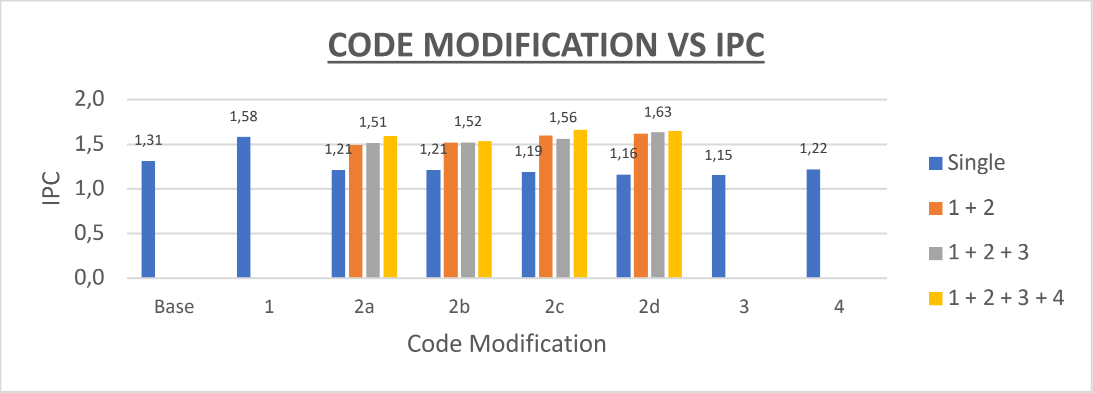
1. Removal of same body check before execution of loop in integrate function
2. Trying different implementations for the computation of the force between bodies. These were classified as a, b, c, and d for the different implementations.
3. Function inlining
4. Removal of the force vector

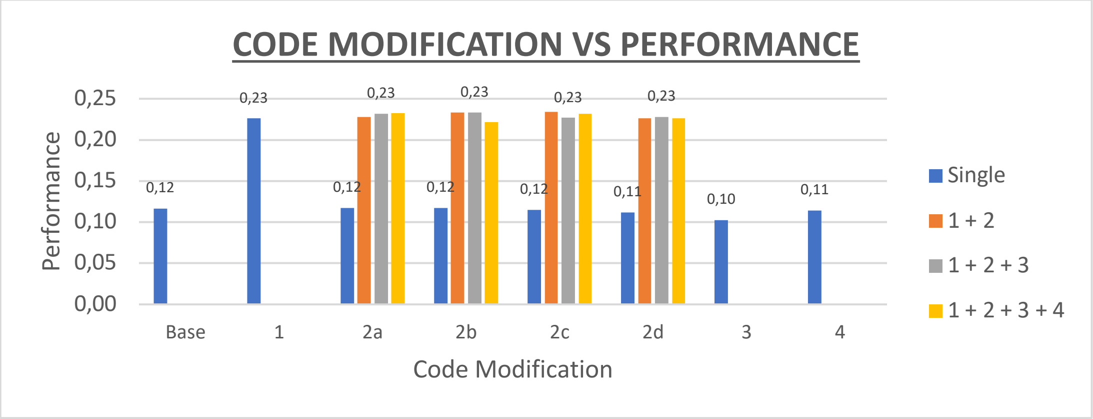
The most significant impact is noticed after the implementation of medication 1, where the check for inequality between "I" and "j" bodies before the start of the loop in the integrate function. The execution time reduces by half while the instructions reduce by almost one-third. The instructions per cycle increases from 1.31 to 1.58 from the baseline code. Overall the performance increases by more than half.
For the rest of the modifications, only a marginal increase is seen when implemented. In some cases, these modifications (aside 1) tended to reduce the performance of the base code.
The only difference in the results of the 0-body position is noticed after implementation of modification 2c when calculating the distance between 2 bodies.

Below are the results of the implementations of the code modifications.





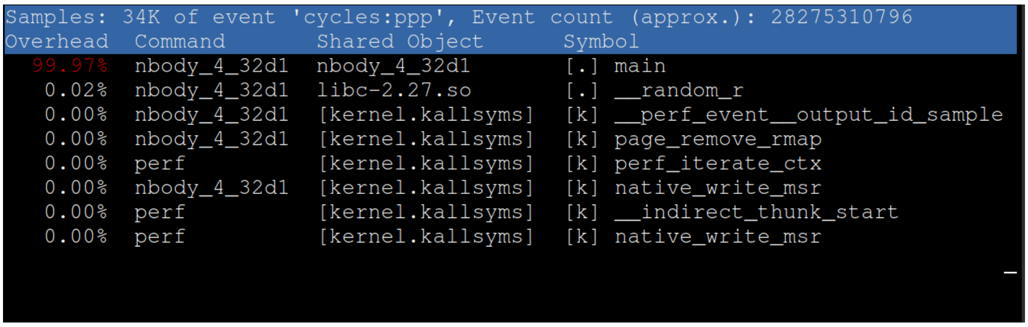




3 ASSEMBLY CODE ANALYSIS

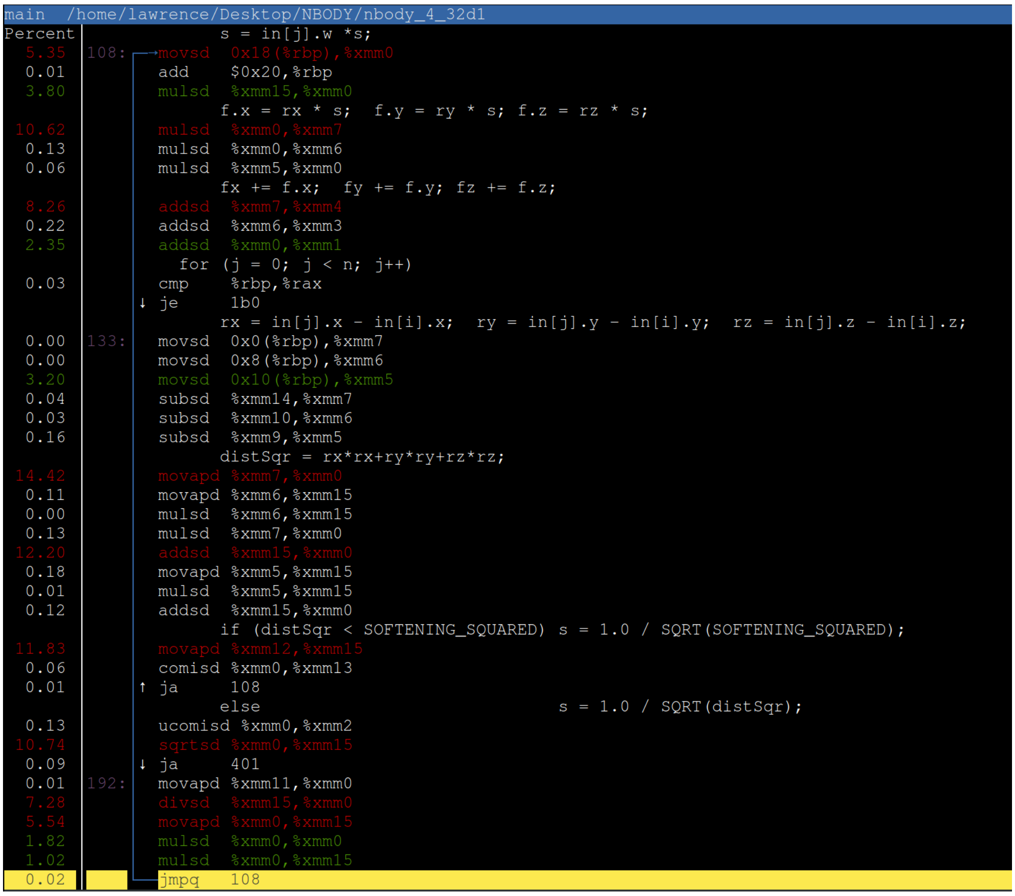
In this section we view and analyze the assembly code to identify what percentage of the resources each of the instructions are using during the execution of the code.

The final code after the implementation of the code modifications is used for this analysis.



```
Samples: 34K of event 'cycles:ppp', Event count (approx.): 28275310796
Overhead  Command          Shared Object        Symbol
  99.97%  nbody_4_32d1     nbody_4_32d1         [.] main
   0.02%  nbody_4_32d1     libc-2.27.so         [.] __random_r
   0.00%  nbody_4_32d1     [kernel.kallsyms]    [k] __perf_event__output_id_sample
   0.00%  nbody_4_32d1     [kernel.kallsyms]    [k] page_remove_rmap
   0.00%  perf             [kernel.kallsyms]    [k] perf_iterate_ctx
   0.00%  nbody_4_32d1     [kernel.kallsyms]    [k] native_write_msr
   0.00%  perf             [kernel.kallsyms]    [k] __indirect_thunk_start
   0.00%  perf             [kernel.kallsyms]    [k] native_write_msr
```

From the table above we can conclude for the samples taken by the perf profiler, the majority of the execution time is spent in the computations in the main function of our program.



```
main  /home/lawrence/Desktop/NBODY/nbody_4_32d1
Percent│                s = in[j].w *s;
  5.35 │108: ┌─→movsd  0x18(%rbp),%xmm0
  0.01 │     │  add    $0x20,%rbp
  3.80 │     │  mulsd  %xmm15,%xmm0
       │     │        f.x = rx * s;  f.y = ry * s; f.z = rz * s;
 10.62 │     │  mulsd  %xmm0,%xmm7
  0.13 │     │  mulsd  %xmm0,%xmm6
  0.06 │     │  mulsd  %xmm5,%xmm0
       │     │        fx += f.x;  fy += f.y; fz += f.z;
  8.26 │     │  addsd  %xmm7,%xmm4
  0.22 │     │  addsd  %xmm6,%xmm3
  2.35 │     │  addsd  %xmm0,%xmm1
       │     │    for (j = 0; j < n; j++)
  0.03 │     │  cmp    %rbp,%rax
       │     ↓  je     1b0
       │     │        rx = in[j].x - in[i].x;  ry = in[j].y - in[i].y;  rz = in[j].z - in[i].z;
  0.00 │133: │  movsd  0x0(%rbp),%xmm7
  0.00 │     │  movsd  0x8(%rbp),%xmm6
  3.20 │     │  movsd  0x10(%rbp),%xmm5
  0.04 │     │  subsd  %xmm14,%xmm7
  0.03 │     │  subsd  %xmm10,%xmm6
  0.16 │     │  subsd  %xmm9,%xmm5
       │     │        distSqr = rx*rx+ry*ry+rz*rz;
 14.42 │     │  movapd %xmm7,%xmm0
  0.11 │     │  movapd %xmm6,%xmm15
  0.00 │     │  mulsd  %xmm6,%xmm15
  0.13 │     │  mulsd  %xmm7,%xmm0
 12.20 │     │  addsd  %xmm15,%xmm0
  0.18 │     │  movapd %xmm5,%xmm15
  0.01 │     │  mulsd  %xmm5,%xmm15
  0.12 │     │  addsd  %xmm15,%xmm0
       │     │        if (distSqr < SOFTENING_SQUARED)  s = 1.0 / SQRT(SOFTENING_SQUARED);
 11.83 │     │  movapd %xmm12,%xmm15
  0.06 │     │  comisd %xmm0,%xmm13
  0.01 │     ↑  ja     108
       │     │        else                            s = 1.0 / SQRT(distSqr);
  0.13 │     │  ucomisd %xmm0,%xmm2
 10.74 │     │  sqrtsd %xmm0,%xmm15
  0.09 │     ↓  ja     401
  0.01 │192: │  movapd %xmm11,%xmm0
  7.28 │     │  divsd  %xmm15,%xmm0
  5.54 │     │  movapd %xmm0,%xmm15
  1.82 │     │  mulsd  %xmm0,%xmm0
  1.02 │     │  mulsd  %xmm0,%xmm15
  0.02 │     └──jmpq   108
```

One evident remark about the current version of the code as shown by the output of the profiler is that a lot of the of the computations are single scalar which means we are not able to generate SIMD instructions. This is due to the way our data is stored in memory. Also the compiler is not able to generate SIMD instructions by itself.
We can also see that a lot of the time is spent on the some of the addition(addsd), multiplication(mulsd) and division(divsd) operations (close to 50%). This can also be attributed to the fact that most of the computations are done without SIMD.
We can also see the remaining amount of the time, which is quite substantial, is spent in the memory access operations. With the knowledge that most of the operations are done without SIMD, this greatly affects the performance of our code.

4 CONCLUSIONS

As part of this exercise we have analyzed the base code for the solution of the N-body problem.

We have established for that a good number of n bodies for the computation (over 5000 in this case), the performance did not vary much based on the size of the problem or n. Also, we established that the time of executed and the quantity of instructions increased quadratically based on the size of the problem.

In terms of compiler the Ofast compiler was identified to increase the performance substantially when used. Though in this particular we did not lose accuracy when using this flag, one needs to pay attention when choosing which compiler flag to use as this could end up improving the speed at the expense of the accuracy.

In terms of code modifications, removing the check of same "I"and "j" at the beginning of the loop halved the execution time and increased the performance of the code more than any of the other implementation. Even if this was a simple check, removing it improved the efficiency tremendously.

Analysis of the assembly code helped to identify which sections of the code consumed a lot of the resources and affected the performance.

4.1 References

1. Lecture slides, Parallel Programming, *Juan Carlos Moure*
2. ParPrg-Perf-Nbody-Hints, *Juan Carlos Moure*
3. https://perf.wiki.kernel.org/index.php/Tutorial