*Cornell University*

*CS5220 – Applications of Parallel Computing*

*Leonardo Neves – lr437*

*Rafael Marinheiro - rf356*

*Victor Silva – vd85*

*Mar 19, 2014*

**Homework 3: Smoothed Particle Hydrodynamics**

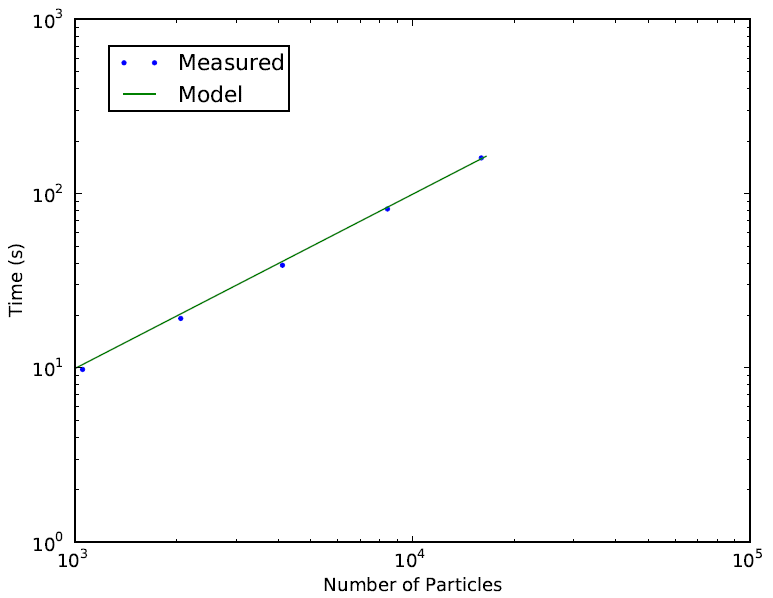
## Introduction

The Smoothed Particle Hydrodynamics (SPH) is a technique to simulate the flow of liquids within a given space. It uses the values of both near-field interactions and the local particle density in order to calculate the movement of particles over time. For this assignment, the team has first have managed to complete the spatial hashing code in order to bring the algorithm to a O(n) order. After the code was complete, the TAU profiling tool was executed to look for bottlenecks that could be improved by using the OpenMP methods to parallelize it and achieve better performance. The last part was coding the OpenMP methods and analyzing the consequent improvements brought by the parallelization.

## Spatial Hashing

The spatial hashing was coded based on the Z-order curve so it was possible to simulate the multidimensional interaction between particles. The functions compute\_density and compute\_accel where implemented to make use of the new spatial hashing and improve performance. The result was more than 80 times faster than the naïve implementation. Another important modification was changing the hash size, the HASH\_DIM variable, from 8 to 64. This had a significant role on the performance improvement.

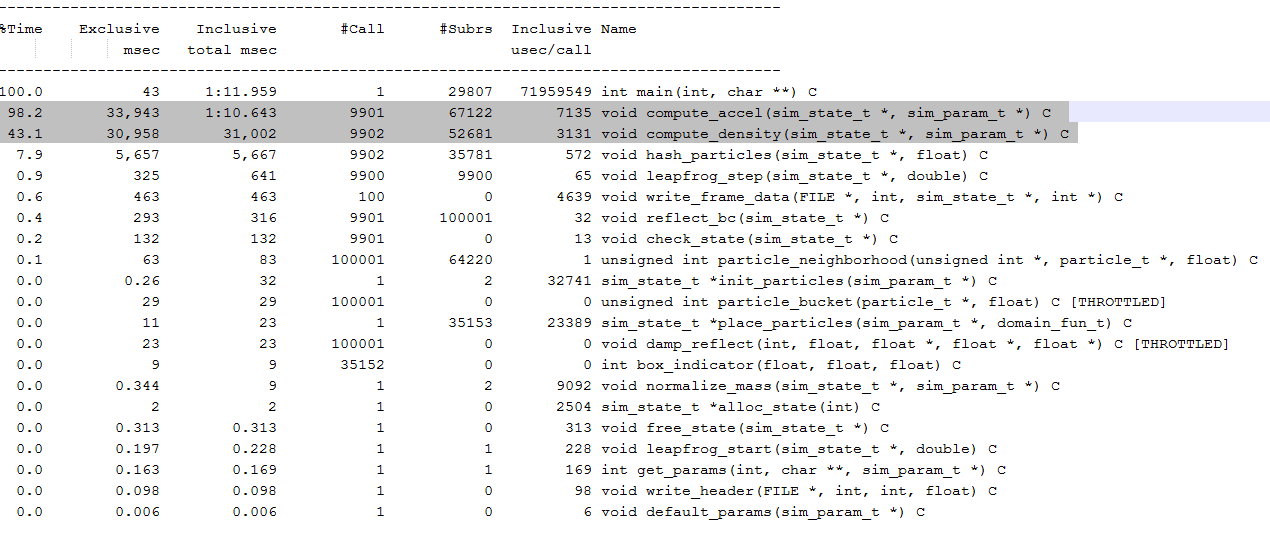
It is possible, by the image1, to see that the order was indeed reduced to something near O(n) on the serial code.



**Image 1: log-log scale of serial code**

## Profiling

Even though the results were already interesting, in order to look for bottlenecks which could be improved, the TAU profiling tool was used. As expected, the two functions that would take more time were compute\_density and compute\_accel as shown in image 2.

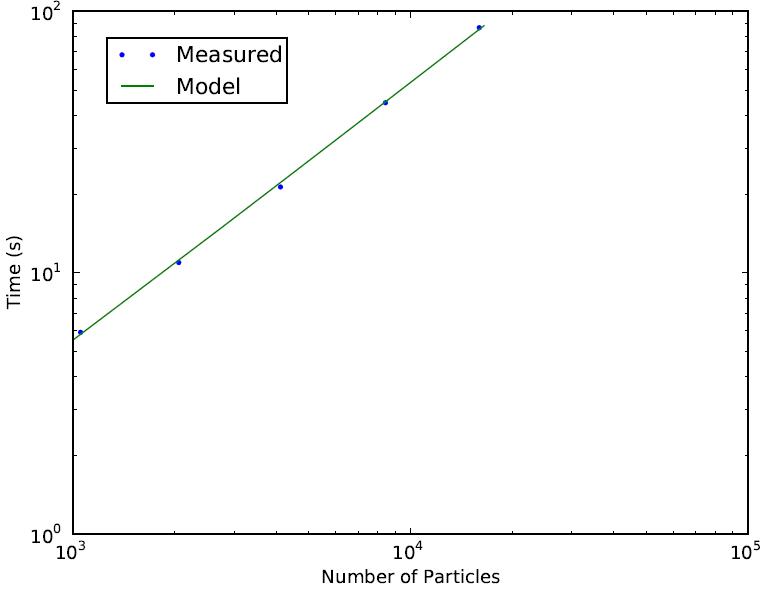


**Image 2: Tau Profile Output**

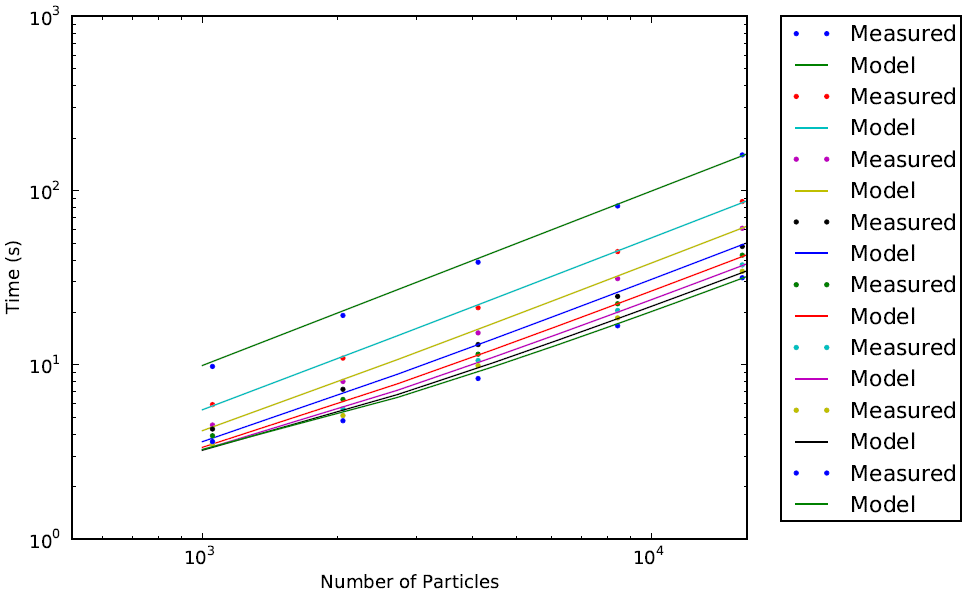
## Parallelization

Given the functions to be improved, the for loops where optimized for parallelism. The variable bucket used inside both functions was turned into a private variable so it would not request to be allocated every time. Also, differently from the serial code, instead of updating pi and pj, each thread would just update pi, not updating the other variables that are responsibilities of other processors. Even though those were simple improvements, the results were quite significant.

The images 3 and 4 show that the parallel code was also working on a O(n) order.

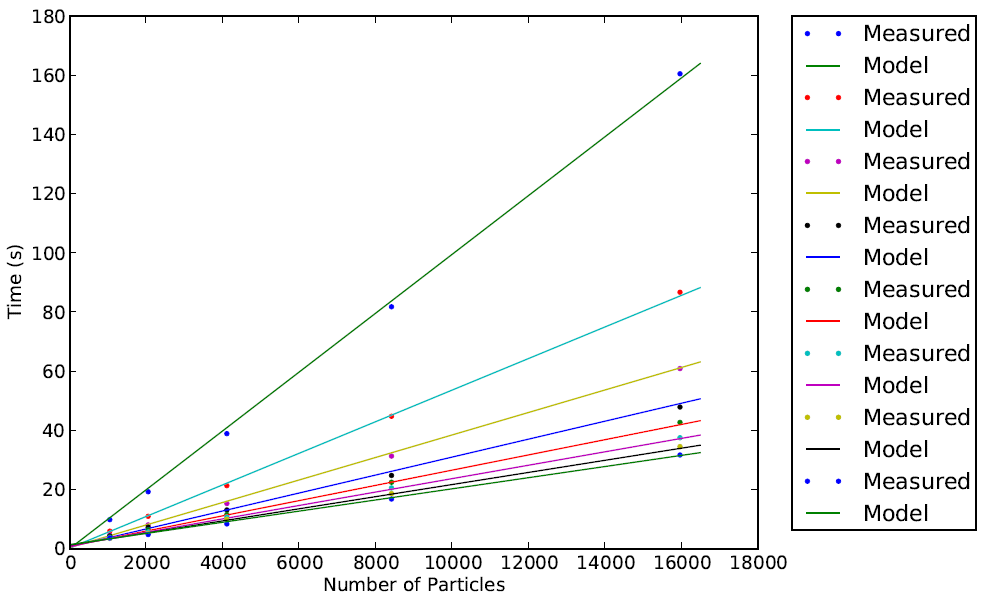


**Image 3: log-log scale for 2 processors**



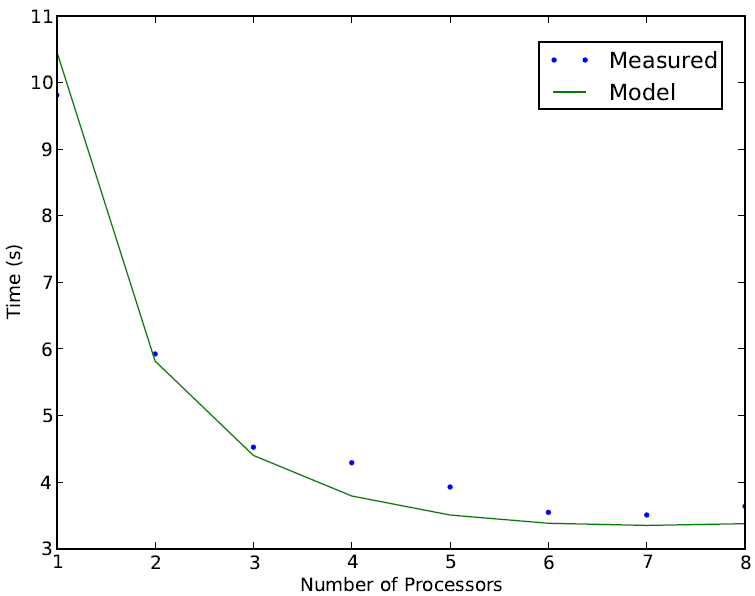
**Image 4: log-log scale for all processors**

The speed-up was also quite consistent. Image 5 shows the performance of each number of processors while increasing the number of particles.

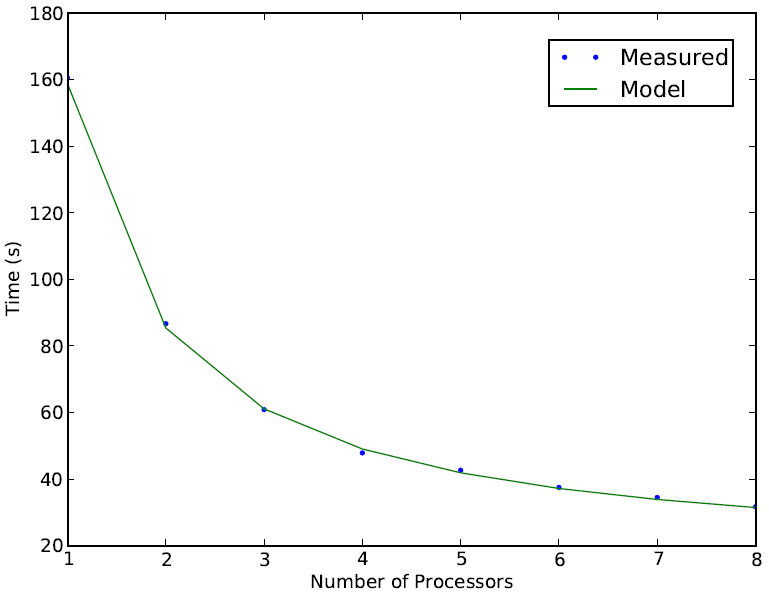


**Image 5: performance of all cores**

For the speed-up plot, it was also easy to see that the performance would become close to the p-times speedup. Obviously, considering overhead and communication time, it would be really difficult to really achieve the idealized model. Image 6 shows us the improvements on 1k particles and image 7 show the same but now for 16k particles. You can see that the more the number of particles, the closest to the model the values are.

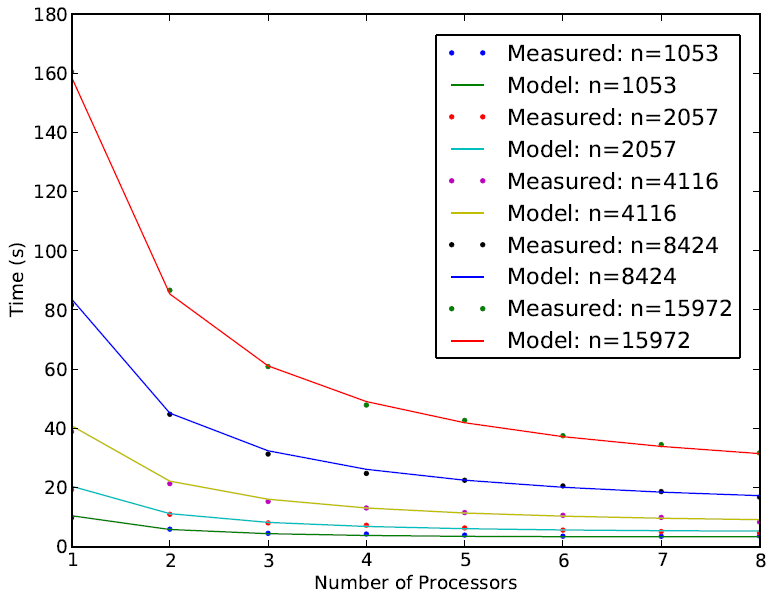


**Image 6: Speedup for 1k particles**



**Image 7: Speedup for 16k particles**

Image 8 compares the speedup for 1k, 2k, 4k, 8k and 16k. The improvement becomes really clear while analyzing it



**Image 8: Overall Speedup**

It would probably be possible to improve the parallel performance by using more sophisticated parallel control technics such as creating a more effective load balancer for the cores. On the other hand, the results were quiet satisfactory and the improvements might not be justify the time spent on creating the code.