*International Journal of Engineering &Technology*, 7 (3.12) (2018) 589-593

**International Journal of Engineering & Technology**

*Website: www.sciencepubco.com/index.php/IJET*

***Research paper***

# Realizing Multilayered Soft Tissue Model Using Multithreading in Python

**Jayasudha.K[1*], Mohan.G.Kabadi[2]**

[1*]*Research Scholar, VTU-RRC, Jnana Sangama, Belagavi, India*
[2]*Presidency University, Itgalpur, Bangalore, India*
**Corresponding Author E-mail: [1]jayanataraja@gmail.com, [2]mohankg@presidencyuniversity.in*

**Abstract**

Interactive Virtual Reality based surgical simulators are gaining popularity where an inexperienced surgeon can practice the surgery irrespective of time and cost which is an effective way in medical training. In order to guarantee the real time performance, a multi-threading technique is applied to the model in different languages. This results in increased utilization of the processor resources and high update rate is achieved with a realistic rendering of the proposed graphics model for the virtual reality simulation. This paper presents a comparative analysis of the model's performance using multithreading process. This is based on addition of matrices of the 3D model simulation with large dimensions in C and python programming languages implementations.

## 1. Introduction

In Virtual environments the main challenging task is to run the application model in real time that means how much time step is needed to simulate the application dynamically within a short period of time. As per industry standards performance increases with respect to higher clock speed. In fact even in desktop computers the computation power is increased by utilizing all computational resources to run parallel. In this paper, the comparative analysis of the performance were analyzed in different languages for the simulation of multilayer soft tissue modeling for virtual surgery application with special focus on parallel computations.

## 2. Method

Human skin is the single largest organ which is about 16% of human adult weight is not just a single layer of white membrane present beneath the skin. But is composed of multiple layers [1], with each layer clearly defined with proper functionality. As per figure 1, human skin is made up of three layers namely epidermis, dermis and sub-cutis layers. Epidermis is the topmost layer which is thin and smooth, acts as primary protective structure. Dermis is the second layer quite thick, also called as hypodermis. This contains only flesh and provides cushion effect. It is subdivided into two more sub layers called superficial dermis and deep dermis layers. Sub-cutis is the last layer which contains fat, connected to muscles and also supplies nutrients to top two layers. It is also called as hypo-dermis. Hence the core work is to simulate the prototype of multiple layers soft tissues that mimics the human skin. The prototype is developed by creating a cube data structure using three dimensional Delaunay triangulation techniques [2].The cube data structure consists of vertex-list, edge-list and triangle-list. The algorithm and data structure [3] satisfies the following criteria:

1 cube = 8 vertices + 18 edges + 12 triangles.

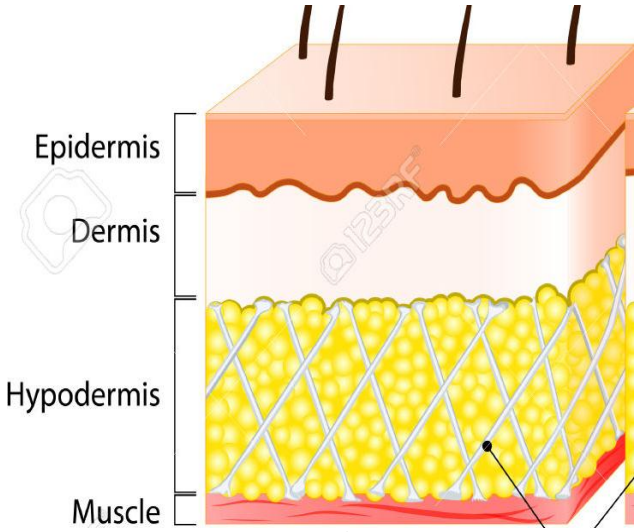


**Fig. 1:** Structure of human skin

Each layer of skin is different in terms of thickness and behavior. Hence layers are modeled giving importance to the thickness. Therefore the first layer is modeled using four adjacent cubes placing next to each other in an horizontal manner forming epidermal layer. The second layer is quite thick, it is modeled as double layered cubical data structure by placing four adjacent cubes in two horizontal layers. The third layer is thick and hard, hence it is also modeled as double layered cubical data structure by placing four adjacent cubes in two horizontal layers. Later all these layers are texture mapped to resemble visual appearance of skin as shown in figure 2.

**Copyright © 2018 Authors. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.**

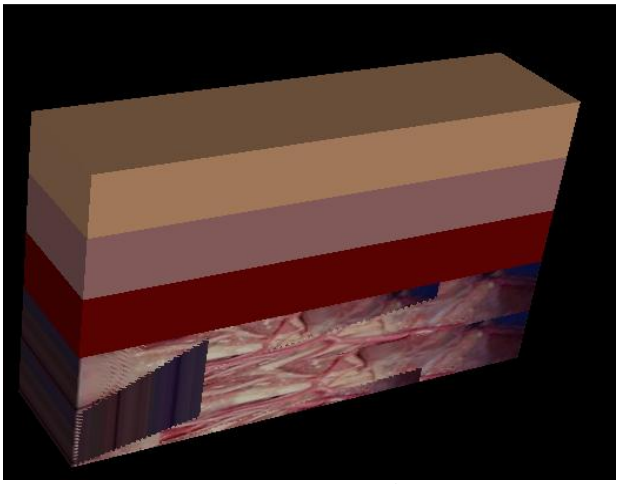

**Fig. 2:** Texture mapped model of human skin

These cubes are placed with respect to 3D points along x-y-z axis on a plane, hence a 3D graphic image is realized on a 2D computer screen. To perform parallel computation these 3D points are analyzed as 2D matrices to perform basic addition operations on various programming platforms like C programming and python programming. In the next section a comparative study is performed to analyze the performance of matrix addition using C language implementation under sequential computing paradigm and python implementation under parallel computing paradigm.

## 3. C language Implementation

C is a basic procedural language that supports dynamic memory management providing several functions for memory allocation and management, these are found in <stdlib.h> header file. Their function and description are as listed below:

a) *void *calloc (int num, int size)*: Allocates array of num elements each of which size is in bytes.

b) *void free (void *address)*: Releases block of memory as specified by address.

c) *void *malloc (int num)*: Allocates array of num bytes and leave them uninitialized.

d) *void *realloc (void *address, int newsize)*: Reallocates memory to the specified address extending up to new size.

C library provides clock() function through header file <time.h> that returns number of clock ticks elapsed since the start of the program. To get number of seconds used by CPU it is required to divide by CLOCKS_PER_SEC. This is because on a 32 bit system CLOCKS_PER_SEC equals 1000000 and will return same value approximately every 72 minutes. During dynamic memory allocation blocks of memory can be requested at run time using library function calloc() from a region of memory called heap. The dynamic memory allocation algorithm is as shown below.

**Dynamic Programming Algorithm:**

1. Begin the program by entering row and column of matrix

```
printf ("Enter the order of matrix");
printf ("Rows=");
scanf ("%d", &p);
printf ("Cols=");
scanf ("%d", &q);
```

2. Dynamically allocate memory for 2 input matrices and resultant matrix

```
mat1=(int*)calloc(p*q, sizeof(int));
mat2=(int*)calloc(p*q, sizeof(int));
res=(int*)calloc(p*q, sizeof(int));
```

3. Enter the elements of matrices

```
for (i=0;i<p;i++)
{
  for (j=0;j<q;j++)
   {
     scanf ("%d",(mat1+i*q+j));
   }
 }

 for (i=0;i<p;i++)
{
   for (j=0;j<q;j++)
    {
      scanf ("%d", (mat2+i*q+j));
  }
 }
```

4. Calculate sum of matrices and put it to resultant matrix

```
for (i=0;i<p;i++)
{
   for (j=0;j<q;j++)
{
  *(res+i*q+j) =*(mat1+i*q+j)+(*(mat2+i*q+j));
}
 }
```

5. Output the resultant matrix ",*(res+i*q+j) with CPU time.

## 4. Python language Implementation

Python is an high level object oriented programming language for general purpose interpreted and interactive computing. Python is a scripting language and has fewer syntactical constructions than other languages. It has many similarities to perl, c and java. It can be installed on various platforms like unix and linux, windows and macintosh. It supports different modes of programming namely interactive mode and in script mode.

It is popular because of the following features:

- easy to learn- read- maintain
- broad standard library
- interactive mode
- portable
- extendable
- scalable
- databases
- GUI programming.

Python supports multi threaded programming that means running many threads concurrently. A thread is a light weighted process and do not require much memory. Multiple threads with in a process share the same data space. The thread class consists of several methods to implement threading, they are as follows:

- Run() : Entry point for a thread
- Start() : starts a thread by calling run()
- Join() : waits for threads to terminate
- isAlive() : checks whether thread is still executing
- getName() : return name of a thread
- setName() : sets name of a thread

One can define function in python to provide required functionality, it begins with keyword *def* followed by function name and parenthesis (()) and starts with a colon (:) and ends with return statement with an expression which is optional. The syntax is as follows:

```
def function-name(parameters):
……
return [expression]
```

Python includes time module through header file <time.h> which provides functions for working with times. The function time.time () return the current system time. The algorithm for multi threading is as shown below.

**Multithreading Algorithm:**

1. Incorporate the threading module an API for spawning multiple threads

```
import threading
```

2. Define function with array of cubes

```
def firstcube (nos):
   print ("The firstcube of given array\n")
   for n in nos:
       time.sleep (0.2)
       print ("firstcube \n", arr1)
def secondcube (nos):
   print ("The secondcube of given array\n")
   for n in nos:
       time.sleep (0.2)
       print ("secondcube \n", arr2)
```

3. Initialize 2D array elements of cubes

```
arr1=[[0,0,0],[20,0,0],[20,10,0],[0,10,0],[0,0,20],[20,0,20],[20,10,
20],[0,10,0]]
arr2=[[0,0,0],[40,0,0],[40,10,0],[0,10,0],[0,0,20],[40,0,20],[40,10,
20],[0,10,0]]
```

4. Create threads

```
t1 = threading.Thread (target=firstcube, args= (arr1,))
t2 = threading.Thread (target=secondcube, args= (arr2,))
```

5. Starting thread1 and thread2

```
t1.start ()
t2.start ()
```

6. Wait for threads until completion

```
t1.join ()
t2.join ()
```

7. Print threads after completion of execution with CPU time

```
print (arr1 [::])
print (arr2 [::])
```

## 5. Results

To increase the performance of the model, the three layers of soft tissues with cubical data structure has been analyzed for parallel and serial computation in the form of addition of matrices with different dimensions. The simulation result tools used are as follows: dynamic programming using C compiler and multithreading using python compiler. Accordingly they are implemented in C and Python programming languages with computing environment as shown below:

**C Language computing environment:**

Intel®core(TM) i3 CPU M380@2.53Ghz
Ubuntu 16.04 64 bit OS, 2GB RAM
gcc version 4.8.4 – C compiler

**Python Language computing environment:**

Intel®core(TM) i3 CPU M380@2.53Ghz
VTK 6.1.0 Python 64-bit(Intel) on windows7
Python compiler

Matrix addition is performed by considering 3D points of all cubes along XYZ axis as a two dimensional matrix. In C programming matrix addition is performed directly with respect to number of cubes in an horizontal manner. In python matrix addition is performed treating each matrix as a single thread of execution in horizontal layered order. Table1 shows the comparisons of actual time taken for execution of addition of matrices in c and python language implementations. Dynamic programming in C allocates blocks of memory of any user defined size requested at run time using library function called *calloc ( )*. These blocks can be reused or freed subsequently. Multithreading in python performs multi tasking i.e. parallelization of threads using functions: *start ( )* and *join ( )*. That means when one thread goes to sleep, the other thread will be active and vice versa towards completion of parallel execution. Figure 3 shows the graphical representation of program execution time. It is clearly understandable from table 1 that the execution time for python is almost same as in c for up to 80 cubes and then decreases gradually. This is because c utilizes heap for dynamic memory allocation, and stack for multithreaded situation such as python. Heap is application specific where as stack is thread specific. In heap memory allocation is at run time, where as in stack it is at compile time. In heap allocating and making free a block of memory is done at any time, so as the data increases it becomes more complex to keep track of the data and hence access to memory becomes a bit slower. And therefore takes more execution time.

**Table 1:** Comparison Table for Addition of Matrices

| Sl No | No of Nodes | Time taken for execution(in sec) using C | Time taken for execution(in sec) using Python |
|---|---|---|---|
| 1 | 4 | 0.846 | 1.888 |
| 2 | 8 | 1.223 | 1.965 |
| 3 | 12 | 1.737 | 2.168 |
| 4 | 20 | 2.577 | 2.464 |
| 5 | 40 | 4.564 | 3.339 |
| 6 | 60 | 4.724 | 4.275 |
| 7 | 80 | 5.574 | 5.023 |
| 8 | 100 | 8.948 | 6.24 |
| 9 | 120 | 11.036 | 7.036 |
| 10 | 140 | 11.576 | 7.986 |
| 11 | 160 | 13.32 | 8.953 |
| 12 | 180 | 13.524 | 10.217 |
| 13 | 200 | 15.737 | 11.513 |

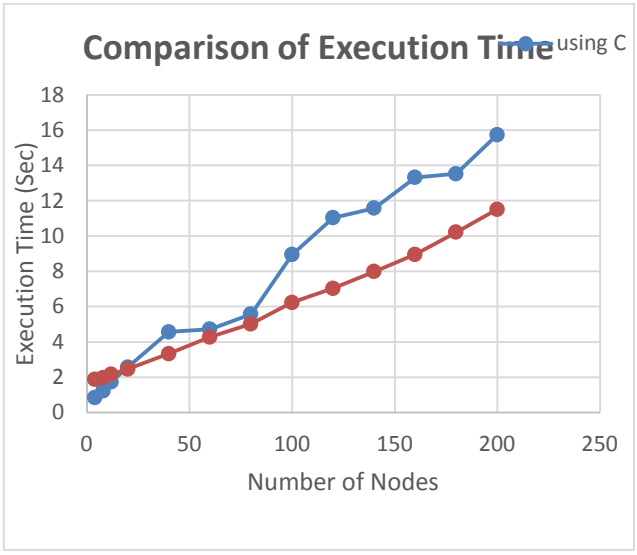


**Fig. 3:** Graphical representation of execution times

Where as in multithreaded situation, each thread will have independent stack and always reserved in LIFO (Last In First Out) order, so even though the data increases, keeping track and freeing a block is nothing compared to heap and hence access to memory is at a faster rate resulting in less execution time.

**Table 2:** Comparison Table for Binary Form Of Execution

| No of Nodes | using C | using Python |
|---|---|---|
| 2 | 0.846 | 0.825 |
| 4 | 1.084 | 1.888 |
| 8 | 1.223 | 1.965 |
| 16 | 1.737 | 2.541 |
| 32 | 3.741 | 3.198 |
| 64 | 7.177 | 4.992 |
| 128 | 13.817 | 7.471 |
| 256 | 16.993 | 14.555 |

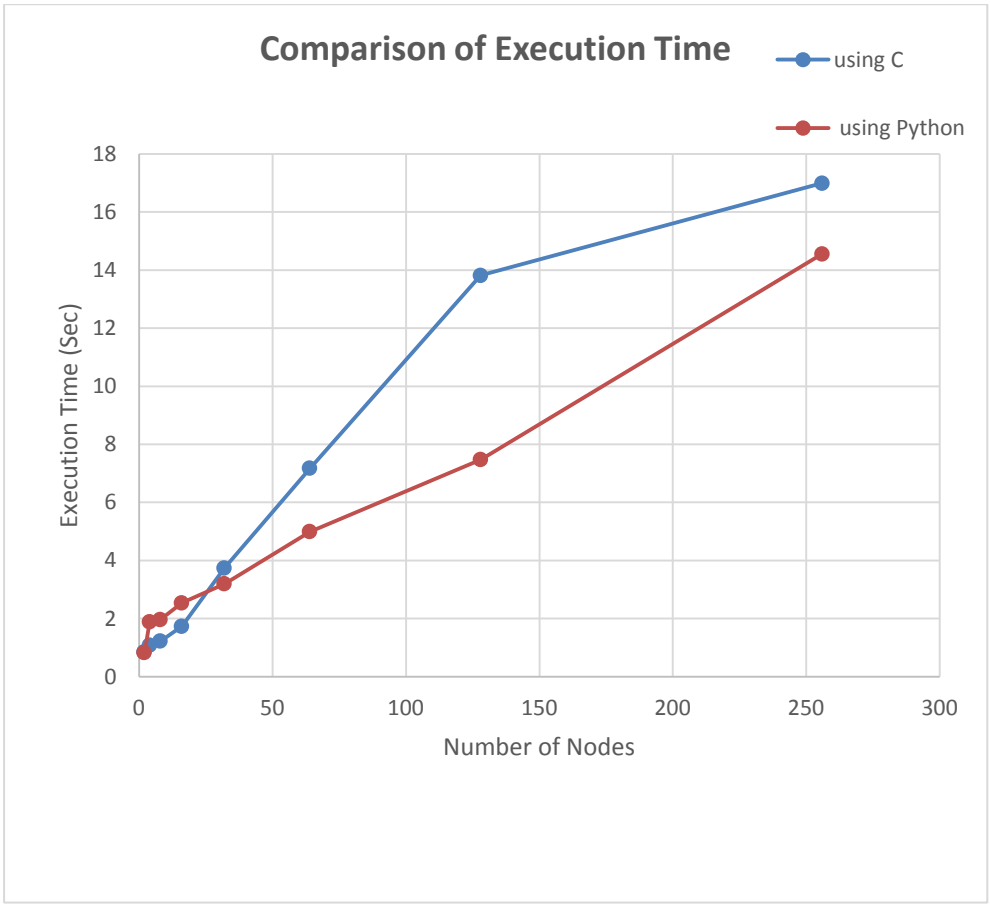


**Fig. 4:** Graphical representation of execution times in binary form

Table 2 shows the same comparison in C and python multithreading, but uses binary form of execution $2^n$ where n is the number of cubes. Figure 4 shows the same in graphical representation. It is to be observed that for small number of cubes say about 32 the execution time is almost the same. And as the number of cubes increases, the execution time gradually decreases for python multithreading. Also to be noted that in multithreading environment the common recommendation is n+ 1 thread, where n is number of CPU cores available. That means when n threads work for CPU, one thread will be waiting for disk I/O. Hence having fewer threads will not utilize the CPU resource.
Therefore more the number of threads will improve the performance by concurrently keeping the CPU busy.

This is also one of the reasons to be observed in graph that having less number of nodes say up to 80 in figure 3 and 32 in figure 4 the execution time is nearly the same as in c.
So it can be analyzed and understood from the graph that the computation time taken for python multithreading implementation which works on parallel computation tremendously decreases with increase in number of cubes in comparison with C implementations which work on sequential computing environments.

## 6. Conclusion

In real time virtual environments it is essential that the performance of the models execution time is expected to be at a

higher rate for realistic rendering of graphics model such that proper utilization of resources takes place. The above mentioned implementation performs matrix addition operations under sequential and parallel computing paradigms. It can be concluded that many matrix operations on parallel computations like multithreading are fast and effective compared to sequential computing environments. To achieve much higher performance at a very fast rate and to solve many complex computational problems in an efficient way CUDA enabled GPU can be used for parallel computing applications as a future work.

## References


[1] Nebel, Jean-Christophe. "Soft tissue modeling from 3D scanned data."*Deformable Avatars*. Springer US, 2001. 85-97.

[2] Maur, Pavel. *Delaunay triangulation in 3D*. Technical Report, Department. of Computer Science and Engineering, University of West Bohemia, Pilsen, Czech Republic, 2002.

[3] Jayasudha.K and Dr.K.G.Mohan. "Variations in Delaunay Model Representation for Soft Tissue Modeling*", International Journal of Computer Applications*, ISBN 973-93-80892-57-2 (online) August 3, 2016.

[4] https://en.wikipedia.org/wiki/C_(programming_language).

[5] https://en.wikipedia.org/wiki/python_(programming_language)

[6] https://en.wikipedia.org/wiki/C_dynamic_memory_allocation

[7] https://www.tutorialspoint.com/cprogramming/c_memory_management.htm

[8] https://www.tutorialspoint.com/python/python_basic_syntax.htm

[9] http://www.vtk.org/doc/release/6.1/html/classvtkDelaunay3D.html