**Enhance Performance**

**Computation Methods of C.Elegans**

Mohd Sadique

Illinois Institute of Technology

Email: [msadique@hawk.iit.edu](mailto:msadique@hawk.iit.edu)

**Abstract**

The Caenorhabditis elegans provides a unique opportunity to interrogate the neural basis of behavior at single neuron resolution. C.Elegans are a type of roundworm that is 1 mm in size and very difficult to track with our naked eyes C. elegans exhibits several elaborate behaviors that can be empirically quantified and analyzed, thus providing a means to assess the contribution of specific neural circuits to behavioral output. C. Elegans locomotory behavior can be recorded and analyzed with computational and mathematical tools. Here, we describe a robust single worm-tracking system, comprising of hardware and software which is initially based on python then java then its ported into distributed systems in C/C++ programming languages, *later to CUDA to read decode jpeg image for improved performance and better utilization of available cores/hardware threads and GPU power.* Our tracking system was designed to accommodate worms that explore a large area with frequent turns and reversals at high speeds.

*This work is about rewriting the software in C++ without any dependency to external libraries and to scale in a highly distributed environment to make use of all the available compute resources in terms of cores and nodes. As a part of this project, a standalone C++ Program is developed to perform worm segmentation utilizing all the cores in the system. Results from evaluation shows that the application is highly scalable and efficiently utilizes the compute resources in a distributed environment. To speed up computation* process blur algorithm and threshold calculation fused together and able to predict the centroid which helps to speedup largestComponent algorithm and distribute reading and decoding on different threads and processing on different threads. Our work is focused on running the code in different distributed systems and select the best performance methodology to implement the segmentation to do observation.

1. **INTRODUCTION**

C.Elegans was the first multicellular organism to have its whole genome sequenced which provides many advantages for unraveling the principles underlying functional neural circuits. C. elegans has a simple nervous system that consists of only 302 neurons and approximately 7000 synaptic connections. Current C. elegans neural circuit studies helps to understand more complex mammalian nervous systems.

Functional neural mapping requires monitoring of behavioral output which dependent on the environment and availability of food. The locomotion has been studied under various conditions and under different mutations to the genetic structure of the worm.

Ioan Raicu

Illinois Institute of Technology

Email: [iraicu@cs.iit.edu](mailto:iraicu@cs.iit.edu)

 To recognize the worm for tracking, a setup consisting of both hardware and software was designed. This algorithm increases the fidelity of worm tracking by effectively removing immobile dark blobs present on the agar surface, such as salt precipitates or air bubbles. It should be noted that, the setup is designed to track only one worm at a time. There are other solutions which are designed for tracking multiple worms at a time [9.2], but their use cases are different and more generalized in nature.

* 1. **Tracking Software**

Tracking software is responsible for capturing the image of the worm movement in the agar plate. The software triggers camera movement to keep track and follow the worm in the agar plate. The tracker was developed in C++ which helps to communicates with the EiBotBoard mounted on the tracker to control the location of the camera. The same Worm Segmentation algorithm is used to find and re-center the camera.

* 1. **Worm Segmentation**

Worm segmentation is an image processing application that takes the raw video of C.Elegan’s movements captured by the tracking subsystem and processes the video to track and record the movement of the worm. Individual video frames are extracted and then converted to gray scale. Then a smoothing(blur) filter is applied to reduce the noise and homogenize pixel densities on the worm body.

We started with profiling the existing code and found that

Finding largest component is slower because it checks each pixel of frame multiple time to find largest component in the frame. New algorithm is implemented which predict centroid based on high density of pixel during blur image processing and which later helps to find largest component effectively.

To track the worm location during its movement, centroid or center of mass of worm in each frame is calculated. Centroid is calculated as the average of x and y coordinated of all M pixels on the worm body

1. **Motivation**

One of the main reasons to enhance the performance and to reduce processing time to process single frames per seconds

Resolution of camera being increased from 640x480 resolution to 4K resolution. Secondly, In Future the camera records the worm more than 100 fps, so the code needs to process the frames at 100 fps or more.

Another issue with higher resolution cameras is noise also increase. This way, segmentation om the worm can occur in real time with the tracking.

1. **DESIGN AND IMPLEMENTATION**

The application is fully written in C++ from scratch without linking to any third-party libraries or modules. This was a specific goal and requirement of this work as this is intended to be used in the HPC environment where third-party library support would be sparse. Three different applications were implemented with Pthread. First application implemented with bulk images. Second application implemented with sequential image processing. Third application implemented to distribute processor to read & decode image and process image.

* 1. Enhancement : Segmenter Algorithm

Initially the application has 4 different components in terms of compute needs - A JPEG read & decoder, image blurring algorithm, threshold algorithm and largest component calculation. For higher resolutions largest Computation time takes 95% of computation time. To Reduce overall computation time Image blurring and threshold algorithm combined and during threshold calculation it calculates pixel density and provides centroid of all image which passed to finding largest Component algorithm to find largest component based on C. Elegans area.

**Algorithm**: Blur\_Threshold\_algorithm

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Input**: image(pixel array, width, height)

**Output**: BlurImage, TCx,TCy (temperory Centroid)

1. TCx=0,TCy=0 ,Tarea=0
2. for each pixel i in the image; do
3. for each neighbor i of pixel; do
4. sum = (add all neighbor pixel value)/ neighbor
5. endfor
6. if(sum >threshold value )
7. BlurImage = sum;
8. Tcx=x ;Tcy=y;Tarea++;
9. Else blurImage =0
10. end if
11. endfor
12. TCx = TCx/area ; TCy = TCy/area;

**Algorithm**: largest\_component

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Input**: image(pixel array, width, height,TCx,TCy)

**Output**: LargestCount(Count of largest connected component)

1. largestComponent\_pixel\_count = 0
2. for each pixel from TCx ,TCy ±25 pixel.
3. if i is true;
4. LargeCount= find\_connected\_components(i, image)
5. endif
6. if LargeCount > 80% of AvgArea **then**
7. Return LargeCount
8. endif
9. Endfor
10. for each pixel i in the image;
11. for each pixel i in the image; do
12. if i is true; then
13. count=0;
14. LargeCount = find\_connected\_components(i, image, &count)
15. if LargeCount > largest\_component\_pixel\_count ; **then**
16. largest\_component\_pixel\_count = LargeCount
17. endif
18. endif
19. endfor

**Function**: find\_connected\_components

**Input**: pixel(x, y cordinates of pixel) image(pixel array, width, height) \*count

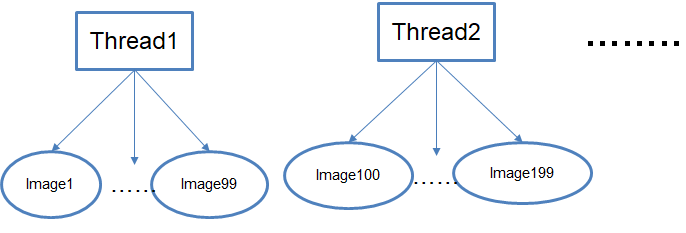
**Output**: count of pixels in this connected component

1. set i to false
2. for each neighbor i of pixel; do
3. **if** i is true; **then**
4. \*count += find\_connected\_components(i, image)
5. endif
6. **endfor**

There are couple other aspects that were considered in optimizing the performance and making best utilization of available resources

1. All high-performance variable is declared as register to allocate register memory instead of stack memory which increase overall CPU Performance.
2. Remove all static functions and variable because it is shared between all thread which reduce computation as compare to individually Pthreads.
3. Pointer variable is used instead of local variables.
4. Global variable is used to shared information between different threads.
5. Multiple mutex lock used to synchronized parallel processing of threads.
   1. **Design-1 Sequential image Implementation(P1): -**

In this implementation all images sequentially divided among different threads eg. Thread1 have 0 – 99 images and thread2 have 100 to 199. Each thread first read image then decode and convert into gray scale image of 8bit data per pixel. If its 1st image then entire image is processed and convert into blurred image and calculated centroid and area of elegans. If any thread process 2nd image then it takes small window of image based on previous image centroid and then process window of image. If in some scenario elegans is not present in window image or largest image is less than 70% of AvgArea then it checks entire image for centroid.



* 1. **Design-2 Parallel frame Implementation(P2): -**

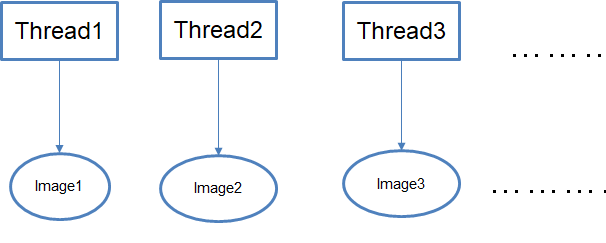
In this implementation all images distributed among different threads based on first come first service bases. eg. Thread1 have 1st images and thread3 have 2nd image. Each thread first read image then decode and convert into gray scale image of 8bit data per pixel. If any Pthread find centroid of image, then next images select window based on centroid find by previous thread. Centroid is shared among different Pthread by global variables. Sequence of all threads is maintained by Mutex lock in Pthread

pthread\_mutex\_lock(&lock);

if(GlobalPointer>=Total\_Images) break;

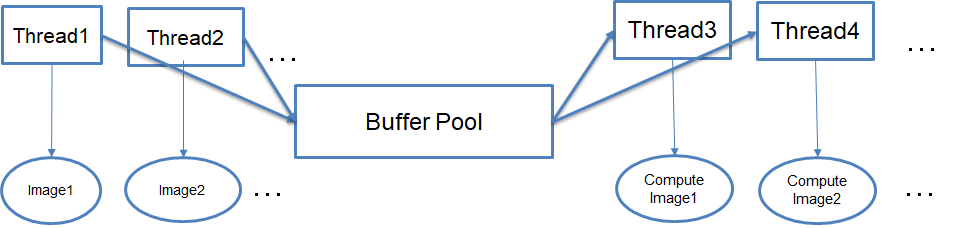
int assign= GlobalPointer ++;

pthread\_mutex\_unlock(&lock);



* 1. **Design-3 Buffer Pool Implementation (P3): -**

In this implementation all Pthread is divided among two section. In First Section, No of Pthread perform read and decode operation. And In Second Section No. of Pthread perform computation on image like implementing blurring and threshold algorithm and finding largest component. Buffer pool is created between Read thread and computation threads, So In Section 1 when allocated Pthread read and decode image into 8bit gray scale image then image saved into buffer pool. Then in 2nd Section when any image is available in buffer pool it starts processing that image. Buffer Pool is created by producer- consumer algorithm. Three different mutex lock is used to synchronized all process.



First Mutex Lock: Read filename position for the list.

pthread\_mutex\_lock(&lock);

if(GlobalPointer>=Total\_Images) break;

int assign= GlobalPointer ++;

pthread\_mutex\_unlock(&lock);

**Second Mutex Lock**: To store decoded image into buffer pool

pthread\_mutex\_lock(&lock2);

Gbuffer[gs]=buffer;

GPos[gs]=i;

pthread\_mutex\_unlock(&lock2);

**Third Mutex Lock**: To Read Decoded image form buffer pool

pthread\_mutex\_lock(&lock3);

if(gf<gs)

{

fs=gf++;

}

pthread\_mutex\_unlock(&lock3);

Read(Gbuffer[fs];

1. **EVALUATION** 
   1. **Test Bed**

All pthread tasks are tested in a Chameleon cloud bare metal system. The system is configured with 24 cores and 128GB memory. The test data consisted of 47000 images of varying resolutions – 640x480, 720x480, 1280x720, 1920x1080, 3840x2160. And New dataset which of 187000 images of 1280X720 resolutions and 250000 images of resolution of 640X480.

All tests were run a minimum of 5 times and the graphs and tables in the following results are an average of the values. Since the computation is purely happening with data from memory, we noticed that that results did not have much of a variation and hence, median or standard deviation is not shown in this report.

* 1. **Results:**
     1. Weak Scalling Test on resolution 1280x960

Execution time for 1000 image per threads is executed on two different architecture .

Data : che\_hr\_nf7

Resolutions: 1280x960

|  |  |  |  |
| --- | --- | --- | --- |
| No. Of Threads | Total Frames | Execution Time (P1) | Execution Time (P2) |
| 1 | 1000 | 19.2278 | 19.0932 |
| 2 | 2000 | 19.4201 | 19.39 |
| 4 | 4000 | 19.706 | 19.78 |
| 8 | 8000 | 21.7095 | 21.8043 |
| 12 | 12000 | 23.2313 | 24.4524 |
| 24 | 24000 | 23.6888 | 24.8651 |
| 48 | 48000 | 43.0521 | 44.5467 |
| 96 | 96000 | 81.0976 | 83.245 |
| 192 | 192000 | 218.666 | 172.45 |

Table 1: Execution time for weak scaling for implementation 1 and 2

Fig 1: Execution time for weak scaling for implementation 1 and 2

From Above weak scaling experiment on resolution 1280x960 to compute 1000 frames per threads is almost constant till 24 thread because Chameleon server contains 24 core means 24 hardware threads and 48 software threads so execution time till 24 threads is constant then it takes doublet time to process image.

Data Folder : che\_hr\_nf7 Resolutions: 1280x960

|  |  |  |  |
| --- | --- | --- | --- |
| No. Of Threads | Total Frames | Frames/ Sec(P1) | Frames/ Sec(P2) |
| 1 | 1000 | 52.01 | 52.37 |
| 2 | 2000 | 102.99 | 103.15 |
| 4 | 4000 | 202.98 | 202.22 |
| 8 | 8000 | 368.5 | 366.9 |
| 12 | 12000 | 516.54 | 490.75 |
| 24 | 24000 | 1013.14 | 965.21 |
| 48 | 48000 | 1114.93 | 1077.52 |
| 96 | 96000 | 1183.76 | 1153.22 |
| 192 | 192000 | 1163.56 | 1113.37 |

Table 2: No. of Frames computed on P1 and P2

Fig2. No of Frames computed on P1 and P2.

P1 and P2 are implemented on two different architecture but overall performance is similar. Maximum frames computes by P1 is 1183 frames per sec of resolution 1280x960 during 96 threads execution. And Maximum frames computes by P2 is 1153 frames per sec of resolution 1280x960.

* + 1. **Weak Scalling Test on resolution 640x480**

Execution time for 1000 image per threads is executed on two different architecture.

Data N2\_f8 Resolution : 640\*480

|  |  |  |  |
| --- | --- | --- | --- |
| No. Of Threads | Total Frames | Execution Time (P1) | Execution Time(P2) |
| 1 | 1000 | 5.94828 | 5.8475 |
| 2 | 2000 | 6.02886 | 6.0241 |
| 4 | 4000 | 6.13948 | 6.2145 |
| 8 | 8000 | 6.13948 | 6.4512 |
| 12 | 12000 | 7.53171 | 7.5642 |
| 24 | 24000 | 8.5178 | 8.6457 |
| 48 | 48000 | 15.3293 | 15.4235 |
| 96 | 96000 | 31.566 | 32.6578 |
| 192 | 192000 | 60.878 | 62.3657 |

Table 3: Execution time for Resolution 640\*480 on P1 and P2

Fig 3: Execution time for Resolution 640\*480 on P1 and P2

From Above weak scaling experiment on resolution 640\*480

fdto compute 1000 frames per threads is almost constant till 24 thread because Chameleon server contains 24 core means 24 hardware threads and 48 software threads so execution time till 24 threads is constant then it takes doublet time to process image.

Data\_folder N2\_f8

Resolution : 640\*480

|  |  |  |  |
| --- | --- | --- | --- |
| No. Of Threads | Total Frames | Frames/ Sec(P1) | Frames/ Sec(P2) |
| 1 | 1000 | 168.12 | 171.01 |
| 2 | 2000 | 331.74 | 332 |
| 4 | 4000 | 651.52 | 643.66 |
| 8 | 8000 | 1303.04 | 1240.08 |
| 12 | 12000 | 1593.26 | 1586.42 |
| 24 | 24000 | 2817.63 | 2775.95 |
| 48 | 48000 | 3131.26 | 3112.13 |
| 96 | 96000 | 3041.25 | 2939.57 |
| 192 | 192000 | 3153.85 | 3078.62 |

Table 4: No. of Frames computed on P1 and P2

Fig 4: No. of Frames computed on P1 and P2

Maximum frames computes by P1 is 3153 frames per sec of resolution 640\*480 during 96 threads execution. And Maximum frames computes by P2 is 3112 frames per sec of resolution 1280x960.

* + 1. **Buffer Pool Implementation for Resolution** 1280x960

Data : che\_hr\_nf7

Resolutions: 1280x960

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. Of Threads | Read Threads | Process Threads | Execution Time | Frames /Sec |
| 24 | 19 | 5 | 27.6958 | 866.56 |
| 24 | 20 | 4 | 26.106 | 919.33 |
| 24 | 21 | 3 | 24.1937 | 991.99 |
| 24 | 22 | 2 | 22.5431 | 1064.63 |
| 24 | 23 | 1 | 44.0708 | 544.58 |

Table 5: Buffer pool implementation for Resolution 1280x960 on 24 threads

Fig 5: Buffer pool implementation for Resolution 1280x960 on 24 threads

Elegans Images is computed based new implementation, the number of threads is allocated to read jpeg into grey scale image and then process image. Images shared among different threads by buffer pool technique.

In Above experiment on resolution 1280x960 with 24 threads to compute 24000 images. The best performance is recorded at 22 read threads and 2 process threads because maximum time taken to read image then convert into 24bit pixel then converted back into 8 bit grayscale image.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *No. Of Threads* | *Read Threads* | *Process Threads* | *Execution Time* | *Frames /Sec* |
| 48 | 40 | 8 | 42.2249 | 1136.77 |
| 48 | 41 | 7 | 41.1732 | 1165.81 |
| 48 | 42 | 6 | 40.8691 | 1174.48 |
| 48 | 43 | 5 | 39.8936 | 1203.2 |
| 48 | 44 | 4 | 43.6869 | 1098.73 |
| 48 | 45 | 3 | 53.1683 | 902.79 |

Table 6: Buffer pool implementation for Resolution 1280x960 on 48 threads

Fig 6: Buffer pool implementation for Resolution 1280x960 on 48 threads

Best performance is measured at 43 threads on read and 5 threads on compute. This implementation supersedes previous two implementation which process **1203 images** per sec and its greater then P1 and P2 implementation

* + 1. **Buffer Pool Implementation for Resolution** 640\*480

Data\_folder N2\_f8

Resolution : 640x480

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. Of Threads | Read Threads | Process Threads | Execution Time | Frames/Sec |
| 24 | 17 | 7 | 9.231 | 2599.94 |
| 24 | 18 | 6 | 8.7435 | 2744.90 |
| 24 | 19 | 5 | 8.4578 | 2837.62 |
| 24 | 20 | 4 | 8.6123 | 2786.71 |
| 24 | 21 | 3 | 11.1937 | 2144.06 |
| 24 | 22 | 2 | 13.4572 | 1783.43 |
| 24 | 23 | 1 | 17.1423 | 1400.05 |

Table 7: Buffer pool implementation for Resolution 640x480 on 24 threads

Fig 7: Buffer pool implementation for Resolution 640x480 on 24 threads

The best performance for resolution 640\*480 is recorded at 19 read threads and 5 process threads as compare to resolution 1280x960 because frame size is small compare to 1280x960 .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. Of Threads | Read Threads | Process Threads | Execution Time | Frames /Sec |
| 48 | 32 | 16 | 17.9352 | 2676.30 |
| 48 | 34 | 14 | 15.8342 | 3031.41 |
| 48 | 36 | 12 | 15.1543 | 3167.42 |
| 48 | 38 | 10 | 15.6452 | 3068.03 |
| 48 | 40 | 8 | 17.6743 | 2715.81 |
| 48 | 42 | 6 | 20.5363 | 2337.32 |
| 48 | 44 | 4 | 29.2454 | 1641.28 |

Table 8: Buffer pool implementation for Resolution 640x480 on 48 threads

Fig 8: Buffer pool implementation for Resolution 640x480 on 48 threads

* 1. **Multi Core Test results**

Java and C++ result compared with new dataset of resolution 640\*480 and 1280x960 and with some old dataset with of resolution 720x480, 1280x720, 1920x1080 and 3840x2160.

Setup consist of 24 cores chameleon server. 48000 frames processing on Java code and C++. Due to in old dataset it contains only 47000 frames, we take 1000 frames from start to compare overall results. Java and C++ code run with 48 pthread on 24 cores.

|  |  |  |
| --- | --- | --- |
| Resolution | Java | C++ |
| 640x480 | 89.748 | 7.927 |
| 720x480 | 145.124 | 9.360 |
| 1280x720 | 345.505 | 19.235 |
| 1280x960 | 475.345 | 24.235 |
| 1920x1080 | 876.247 | 45.125 |
| 3840x2160 | 3034.611 | 163.384 |

Table 9: Computation results for different resolution on JAVA and C++ with 48000 frames on 48 pthreads.

Fig 9: Computation results for different resolution on JAVA and C++ with 48000 frames on 48 pthreads.

As the graph clearly shows, C++ is more than 20 time faster than Java to compute centroid and area of C. elegans.

CUDA implementation was not compared in these tests because GPU jpeg processing is not open source and takes time to much time to implement GPU decoding on CUDA. Due to time constraints it cannot implemented now but will be implemented in future.

1. **PROFILING**

Java and C++ implementation were profiled to see the performance of individual component JPEG read & Decorder, Blur\_threshold algorithm and largest component algorithm. Blurring and threshold is combined together for better performance.

* 1. Loading Image

|  |  |  |
| --- | --- | --- |
| Resolution | Java | C++ |
| 640x480 | 13.462 | 7.292 |
| 720x480 | 21.769 | 8.611 |
| 1280x720 | 51.826 | 17.696 |
| 1280x960 | 71.302 | 22.296 |
| 1920x1080 | 131.437 | 41.515 |
| 3840x2160 | 455.192 | 150.314 |

Table 10: Time for loading a single image as gray scale.

Figure 10: Time for loading a single image as gray scale.

Figure 10 shows time taken for loading 48000 images in secs . Java is inherently slow in loading the image and converting to gray scale. C++ code is faster as it uses stdlib.h to load image into memory.

* 1. Image Blur and Thresholding Computation.

|  |  |  |
| --- | --- | --- |
| Resolution | Java | C++ |
| 640x480 | 44.874 | 1.823 |
| 720x480 | 72.562 | 0.842 |
| 1280x720 | 172.753 | 1.731 |
| 1280x960 | 237.673 | 2.181 |
| 1920x1080 | 438.123 | 4.061 |
| 3840x2160 | 1517.305 | 14.705 |

Table 11 :Time taken for blurring and thresholding.

Fig 11 :Time taken for blurring and thresholding.

In Figure 11 shows time taken for blurring and thresholding computation for 48000 images . Java is much slower as compare C++. In C++ implementation it combine both blurring and thresholding process its implementated like when blurring pixel calculated it checks its pixel value if its greater then threshold value then it set 1 else 0. Based on previous image centroid it copy window of image to calculate next image centroid and area.

* 1. Largest Component

|  |  |  |
| --- | --- | --- |
| Resolution | Java | C++ |
| 640x480 | 31.412 | 0.476 |
| 720x480 | 50.793 | 0.562 |
| 1280x720 | 120.927 | 1.154 |
| 1280x960 | 166.371 | 1.454 |
| 1920x1080 | 306.686 | 2.707 |
| 3840x2160 | 1062.114 | 9.803 |

Table 11: Time taken to find Largest Component.

Fig 11: Time taken to find Largest Component.

Figure11 shows the time taken to find the largest connected component in a binary image to detect the body of the C. elegans and calculate the centroid. Java drastically slower as compare to C++ code. In C++ temporary centroid is predicted based on high pixel density and first calculate at that location if area is larger then 80% of AvgArea of C.elegans then return area with centroid. Else it check entire image for largest area.

1. **Conclusion:-**

C++ performance during Bluring , threshold and Largest component is improved drastically it atleast 100 time faster then java code due to largest component prediction method during bluring and threshold calculation. Implementation of the segmentation. C++ implementation is bottleneck is Reading JPEG image file during this process image is decoded converted into 24bit pixel and then again converted into 8Bit gray scale image. Which takes 90% of overall performance.

1. **Future Work:-**

According to C++ Implementation the bottleneck is read jpeg image and convert into grayscale image. To improve image processing performance gpu\_image\_processing should be implemented to decode jpeg image into gray scale image with higher performance.

From above research bottleneck is JPEG conversion into 8bit gray scale image which should be take care in future for high performance support and high resolution images and process 1000 frames per sec of resolution 4K.

Needs to work on new algorithm which effectively trace multiple elegans at same with higher resolutions and faster frames processing.

1. **REFERENCES:-**
2. Moy K, Li W, Tran HP, Simonis V, Story E, Brandon C, et al. (2015). “Computational Methods for Tracking, Quantitative Assessment, and Visualization of C.elegans Locomotory Behavior” PLoS ONE 10 (12): e0145870. doi/10.1371/journal.pone.0145870
3. Aditya Shirodkar, Chiranjeevi Ankamreddy, Calin Segarceanu Ioan Raicu. Benchmarking Worm Segmenter
4. Sjie Jason Wang, Zhao-Wen Wang (2013). “Track-A-Worm, An OpenSource System for Quantitative Assessment of C.elegans Locomotory and Bending Behavior”. PLoS ONE 8(7): e69653. doi:10.1371/journal.pone.0069653
5. Derek Bradley, Gerhard Roth. Adaptive Thresholding Using Integral image.
6. Leifer AM, Fang-Yen C, Gershow M, Alkema MJ, Samuel AD. Optogenetic manipulation of neural activity in freely moving Caenorhabditis elegans. Nat Methods. 2011; 8(2):147–52. doi: 10.1038/nmeth.1554
7. Enhance Performance of computation methods of C.Elegans. Piyush Nath, Ioan Raicu, Daniela Raicu
8. Steven J Husson, Wagner Steuer Costa, Cornelia Schmitt, Alexander Gottschalk et al. (2012). “Keeping
9. Ramot D, Johnson BE, Berry J T L, Carnell L,
10. Adelina Voukadinova, Tejus Prasad, Ioan Raicu, Daniela Raicu (2016) “Improving the Performance of C.Elegans RoundWorm Tracking and Segmentation.
11. Goodman MB. The Parallel Worm Tracker: a platform for measuring average speed and drug-induced paralysis in nematodes. PLoS One. 2008; 3(5):e2208. doi: 10.1371/journal.pone.0002208
12. **Swift/T:** http://swift-lang.org/Swift-T/
13. **Spark:** https://spark.apache.org/