Universiteit Leiden
The Netherlands

Opleiding Informatica

Design and Implementation of 3D Reconstruction from Axial Views on the Leiden Life Sciences Cluster

Juliëtte Meeuwsen

Supervisors:
Yuanhao Guo, Dr. K. F. D. Rietveld, Prof.dr.ir. F.J. Verbeek

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

05/08/2017

Abstract

A microscope equipped with a VAST BioImager can efficiently produce a large amount of axial view images for the zebrafish larvae. The efficiency of subsequent image processing can be further optimized. This thesis presents a solution for parallelising such big sets of axial views using the computation power of the Leiden Life Sciences Cluster (LLSC) to design and implement a parallelisation for the 3D reconstruction algorithm. The algorithm is inspired from the independent operation of the projections among all the axial views and the 3D points. A implementation of the algorithm is described that effectively utilises all compute resources available in the cluster computer. An experimental evaluation will point out how software for axial view 3D reconstruction is best parallelised and tuned on LLSC.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 VAST BioImager

The Vertebrate Automated Screening Technology (VAST BioImager) (see Figure 1.1) is a microscope mounted system. The VAST BioImager contains a capillary that holds a zebrafish for imaging. The VAST loads a zebrafish into the capillary and positions it in front of the camera by adjusting the pressure in the tubes. Through the rotation of the capillary, multiple axial-views of a specimen can be acquired. All the remaining larvae will be rotated automatically to the same orientation using pattern recognition algorithms.

Figure 1.1: VAST BioImager connected to microscope and computer

source: http://www.unionbio.com/vast/

The VAST makes use of two CCD (Charge-Coupled Device) cameras. One camera is used to check the VAST-unit and in our in-house application, it is also used for 3D construction. The other camera is mounted on the microscope and can also use fluorescence for creating high resolution 3D images.

Using the images created by the VAST BioImager a zebrafish can be reconstructed. This process is visualised in Figure 1.2.

First segmentation (see Chapter 1.2) will be performed on all separate images. That is, every pixel in the image will be examined and if it belongs to the specimen, then the value of that pixel will be set to 1. If it belongs to the background, then is the value will be changed to 0. This results in a black and white picture.

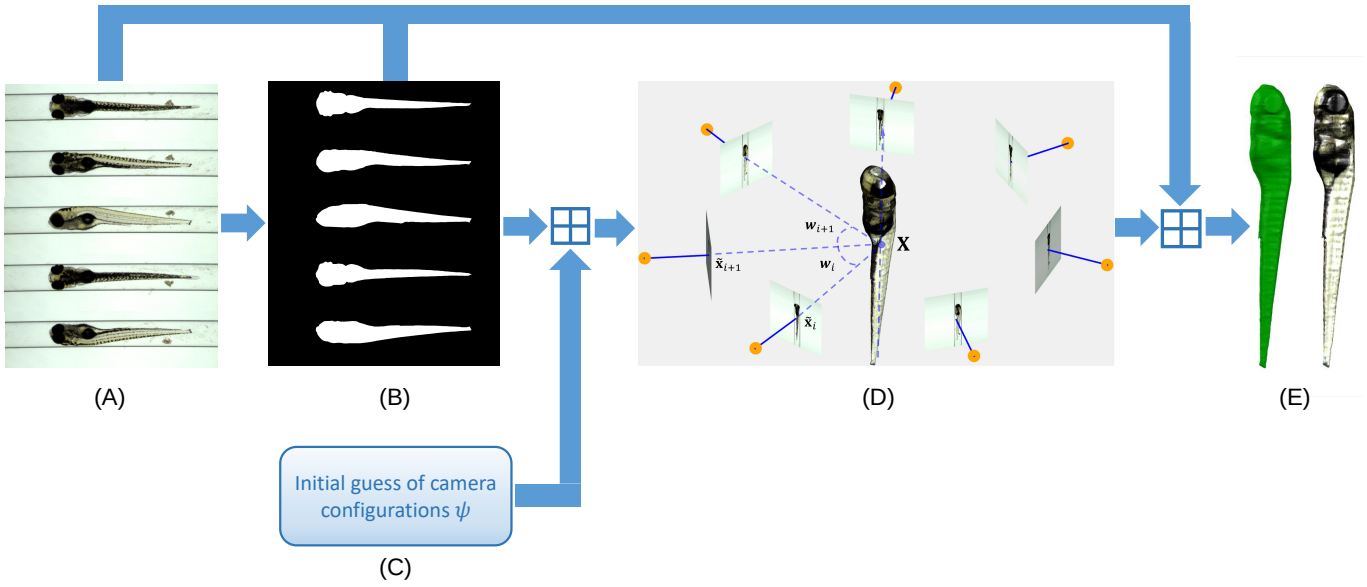


Figure 1.2: Walk-through of reconstruction process of zebrafish using VAST-images [SV17]

After that, for every camera location the associated image must be found. This is needed to construct the right sequence of images. Finally, all images can be combined into a 3D model and a texture can be mapped onto the 3D model to give the zebrafish a lifelike appearance. This method is called a shape-based/axial-view based 3D reconstruction [SV17].

When a 3D model is made, the volume of a zebrafish can be calculated. Since every pixel belonging to a object has a certain value, where every pixel which does not belong to the object has no value at all, adding all values from pixels which belong to an object will result in the volume of the object.

1.2 Segmentation

In image processing, segmentation is the process of partitioning a digital image into multiple segments. The goal is to simplify the representation of an image to make it easier to analyse. It is usually used to locate objects and boundaries in images. Actually, segmentation is comparable to labeling every pixel in an image such that all pixels with the same label share characteristics. An image segmentation can be used to create 3D reconstructions with the help of interpolation algorithms like Marching cubes (see Chapter 3.3.2).

Segmentation of zebrafish is tricky due to the transparency of the objects. Popular computational models usually fail to segment interesting edges, thus a new method introduced by Yuanhao Guo, Zhan Xiong and

Fons Verbeek [SV17] is used in this project.

The segmentation is done by fusing two segmentation candidates and refine the results making using of the contours of the zebrafish. The first segmentation makes use of the mean shift algorithm to augment the colour representation for the partial transparent regions and transform the ambiguous edges more separable. The result is a whole-overview of the shape of the zebrafish. The second segmentation is done by initialising a distance regularised level set function from this segmentation candidate, followed by feeding the result to an improved level set method to obtain a more complete shape representation which preserves explicit contours.

1.3 Reconstruction

After segmentation is applied, a 3D model of a zebrafish can be reconstructed. This is done by merging all separate images together, where each image belongs to only one axial view. However assigning images to the corresponding camera positions and merging all images into a 3D image is currently a slow process. To improve this process there has been chosen to investigate the effects of parallelising this process by executing the algorithms on a cluster computer, namely the Leiden Life Sciences Cluster (LLSC).

This thesis investigates how the MATLAB code should be translated to Python code to improve the process and to be able to run the code on the cluster computer. Several research questions arise from this problem: how is the software for axial view 3D reconstruction best parallelised and tuned on a cluster computer? What are the bottlenecks of the software?

1.4 Thesis overview

This bachelor thesis project is supervised by Dr. K. F. D. Rietveld and Prof.dr.ir. F.J. Verbeek and is executed at The Leiden Institute of Advanced Computer Science (LIACS). In Chapter 2 related work is mentioned. Implementation-wise there was some room for improvement, this all will explained in Chapter 3. After implementing these improvements, there had to been made sure that in fact the process has improved. The performance of the process is tested by executing experiments as explained in Chapter 4. Finally, in Chapter 5 some conclusions will be drawn about the entire project and suggestions for future work will be mentioned.

Chapter 2

Related Work

Analysing Electron Tomography with IMOD on the LLSC by Simon R. Klaver
This thesis [Kla15] describes how to reduce processing time on large transmission electron microscopes, which can automatically process multiple samples in sequence, by parallelising the software used for further processing of image series.

Developing an integrated environment for OPT image reconstruction by Dennis van der Zwaan
This thesis [vdZ16] describes how well the reconstruction of a sizable set of large OPT images can be distributed on the LLSC. Some optimisations on implementation were done to reduce the total execution time. Also a usable prototype GUI application has been developed to interface with a developed web service.

3D reconstruction and measurements of zebrafish larvae from High-throughput axial-view in vivo imaging by Yuanhao Guo, Wouter J. Veneman, Herman P. Spaink and Fons J. Verbeek
This paper [SV17] describes an automated system in which images of zebrafish larvae are acquired and used to reconstruct to 3D models so that 3D measurements can be applied on the whole specimen. The system integrates the axial-view imaging, segmentation, camera system optimization and a profile-based 3D reconstruction method. The 3D measurements of volume and surface area, as well as the 3D visualization become available through the proposed method on zebrafish larvae using high-throughput imaging.

Chapter 3

Methods

This Chapter describes what a cluster computer is and what is needed on the LLSC to run the Python code. Moreover, it describes the problems and solutions encountered during the process of translating the MATLAB to Python code.

3.1 Cluster Computer

Cluster computers consist of connected computers that work together, but can be viewed as one single system. Each separate machine is called a node (see Figure 3.1) and can perform the same tasks which are controlled and scheduled by a software package called the job scheduler. The components of a cluster computer could be compute nodes, user nodes, admin node and a file server, which are all connected through a fast local network.

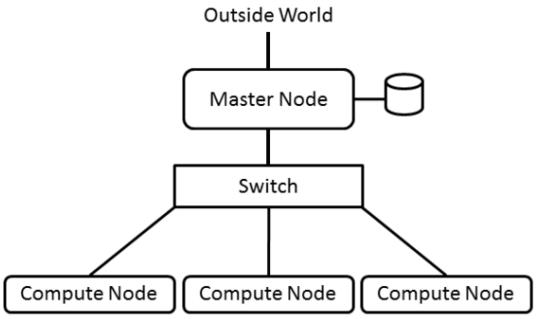


Figure 3.1: Schematic representation of a cluster computer

source: http://www.admin-magazine.com

Each node runs its own instance of an operating system. In most circumstances, all of the nodes use the same hardware and the same operating system, but this does not always has to be the case. In case of LLSC, all nodes use the same operating system.

A great advantage of clusters is the high fault tolerance. The system can continue working with a malfunctioning node. Moreover, it provides high processing capacity and data recovery in the event of a disaster is possible.

The master node will always preserve a copy of the data set and other details before diving the data over the computer nodes, in case of job failure where a compute node unexpectedly shuts down and loses its data set, the master node is able to do a resubmit of the job to another node. With a computer cluster it is easier to include parallelism and thus increase speed of executing and completion of tasks compared to using one single computer.

3.1.1 Leiden Life Sciences Cluster

LLSC is a cluster computer located at Snellius (Leiden Institute of Advanced Computer Science (LIACS)) and is used as a testbed for the optimisation of bio-informatics software. It is a cluster consisting of a admin node, three user nodes, a set of compute nodes and a file server. To schedule submitted jobs a scheduler on a user node is used. In case of LLSC TORQUE (see Chapter 3.1.2) is used as long-term job scheduler.

The compute nodes are not all alike, there are different processor types used and the amount of physical memory differs per node. The following kind of nodes are being used during the experiments:

- cpu-5150: 9 nodes, two Intel Xeon 5150 processors @ 2.66 GHz per node (4 cores total) and 16 GB memory
- cpu-5150: 2 nodes, two Intel Xeon 5150 processors @2.66 GHZ per node (4 cores total) and 8 GB memory
- cpu-e5430: 5 nodes, two Intel Xeon E5430 processors per node (8 cores total) and 16 GB memory
- cpu-x5450: 2 nodes, two Intel Xeon X5450 processors per node (8 cores total) and 16 GB memory
- cpu-x5355: 1 node, two Intel Xeon X5355 processors per node (8 cores total) and 16 GB memory

3.1.2 Task scheduling (TORQUE)

When a large multi-user cluster needs to process large numbers of tasks, task scheduling becomes a challenge. This is also the case on the LLSC. The performance of each job depends on the characteristics of the underlying cluster, this area is of ongoing research.

The Terascale Open-source Resource and QUEue Manager (TORQUE) is an open source resource manager which can be used to provide control over batch jobs and compute nodes. It manages jobs that users submit to the queues on a computer system. TORQUE is integrated on LLCS.

This job scheduler allocates computer resources requested for the job, runs the job and reports the outcome of the execution back to the user. TORQUE also handles faults and checks node health with the use of scripts. Moreover, it supports extensive logging additions and has the ability to run large jobs over large clusters (more than 2000 processors) [Inc17].

When using different approaches when executing a certain job, it is easier to determine which approach is best-suited for the job. TORQUE can be used to determine which nodes are being used and which can be used in the near future.

3.2 Implementation

A simple implementation of the entire algorithm to reconstruct 3D models from the VAST BioImager input images already existed in MATLAB. The implementation is a translation of this MATLAB code into Python code. Python 2.7.12 is chosen to be used for this implementation because Python is, unlike MATLAB, open and free to use, where MATLAB (code) can only be used by people that have paid for a license. Thus, Python is more portable than MATLAB is. Also, it is easy to use Python packages that third parties have created.

For this conversion several libraries replacing the MATLAB toolboxes where needed:

1. NumPy *Base N-dimensional array package*
2. SciPy *a fundamental library for scientific computing with Python*
3. MatplotLib *for comprehensive 2D plotting*

3.2.1 Libermate

Since a MATLAB code was provided, it seemed convenient to use that code instead of trying to create similar code in Python from scratch. Thus, it was decided to translate the MATLAB code to Python. This could be done manually, but a number of translators exist that can automatically perform the translation.

Libermate [Sch14] is such a translator and has been used to perform a rough translation from MATLAB implementation to Python implementation. That is, this software takes MATLAB files and translates these line by line making use of translation files (where preserved words are known) and the SciPy package.

After using Libermate to translate all MATLAB files, more modifications had to be made. This is because the translation was not complete and sometimes the translation did not make any sense. To solve this, the code had to be compared line by line to the MATLAB implementation to get a good view on what it should be.

Some errors were easy to resolve, e.g. by just renaming a variable (e.g. lambda is a preserved word in Python, and therefore can not be used for variable names), but some errors could also be quite severe resulting the matrix translation to be completely incorrect. In that case a good understanding of what the matrix should look like needed to be created before rewriting the code to create a correct output.

3.2.2 Construction of the code

The implementation can be divided into several parts:

1. Reading input image data and putting into a NumPy array. Then constructing 3D points into 3D space;
2. Connecting the 3D points to create a surface and visualize this by using Marching Cubes [Bou94];
3. Smooth the edges of the 3D object by using a smoothing function and write the data to a output file.

3.2.3 Cython or Ctypes

In some cases one would like to use a C implementation and integrate that with code written in another programming language. The original MATLAB implementation used C code for smoothing. That same C code could be used with Python code but only if a binding between Python and C can be created. This could be done by using Cython or Ctypes.

Cython [MS17] is an optimistic static compiler for both the Python programming language and the extended Cython programming language. It makes writing C extensions for Python as easy as Python itself. It writes Python code that calls back and forth from and to C/C++ code natively at any point. It allows to integrate natively with existing code and data from legacy, low-level or high-performance libraries and applications.

The compiler can generate efficient C code from Cython code, due to the fact that it supports calling C functions and declaring C types on variables and class attributes. The C code is generated once and then compiles with all major C/C++ compilers in Cython. Cython is a language for wrapping external C libraries, embedding Cython into existing applications, and for fast C modules that speed up execution of Python code.

Ctypes is another way of wrapping C libraries in pure Python. It is a foreign function library for Python and its provides C compatible data types, and allows calling functions in DLLs or shared libraries. [Fou17a]

3.3 Creating images

3.3.1 Detecting voxels

The 3D zebrafish models are constructed using voxels. A voxel represents a data point (value) on a regular grid in three-dimensional space (see Figure 3.2). This data point can consist of a single piece of data, such as an opacity, or multiple pieces of data, such as color in addition to opacity.

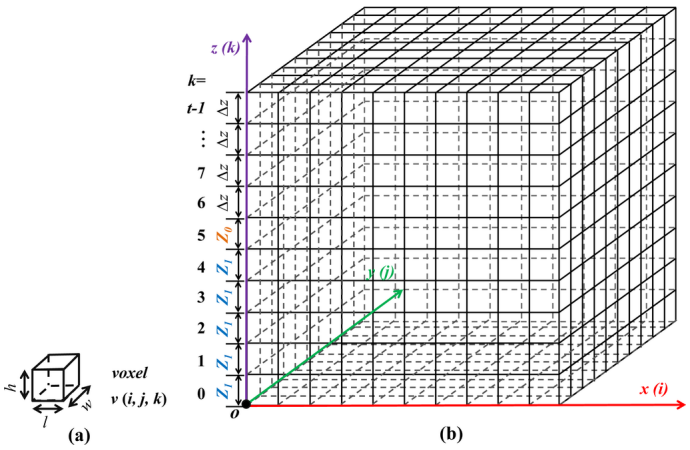


Figure 3.2: Schematic representation of a voxel in a 3D object

source: http://www.mdpi.com/2072-4292/5/2/584/htm

Voxels do not know their exact coordinates themselves. Instead, the positions of the voxels is relative to the location of other voxels. [FF90]

All these voxels in the 3D object have a value, which is either 0 or 1 (or 255). When it has value 0 then it means that the voxel belongs to the background, if the value is either 1 or 255 then the voxel belongs to the object, which is in this case a zebrafish. When for every voxel the value is determined a 3D model can be created. This is done by connecting all voxels with value 1 or 255, and changing their values all to an equal value (=1)

Also, after creating such a 3D model it is not hard to determine the volume of a zebrafish. This can be achieved by adding up all voxels belonging to the zebrafish. Since all voxels have the same value, it is only a matter of determining the amount of voxels that are part of the zebrafish and then that results in the final answer.

3.3.2 Marching Cubes Algorithm

Marching Cubes is a well-known volume rendering algorithm for computer graphics. The algorithm transforms a three-dimensional object as a set of connected polygons (usually triangles). The application is mainly concerned with medical visualisations, for example MRI scan data images. William E. Lorensen and Harvey E. Cline [LC87] have developed this algorithm. They worked on a way to efficiently visualise data CT and MRI devices and came up with this solution.

The Marching Cubes algorithm takes several voxels (points) in a 3D-space and forms an imaginary cube. Next the polygons belonging to these surfaces are determined and calculated. Combining all the individual polygons results in the desired surface.

The most important task of the algorithm is to determine the amount of triangles that are needed per cell, in what kind of shape and in which orientation they need to be. A cell is a cube and this cube is sectioned by a part of the 3D object. The intersection of the 3D object is made up out of triangles.

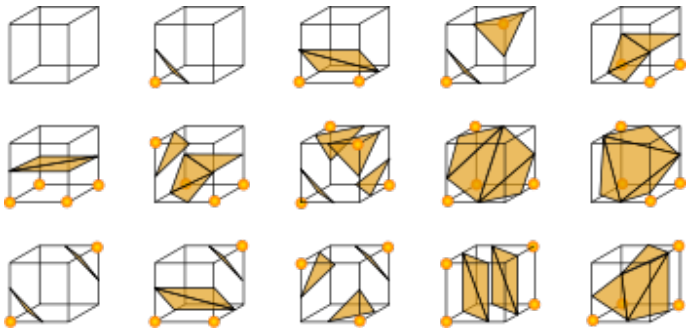


Figure 3.3: Intersection planes on a Marching Cube

source: https://en.wikipedia.org/wiki/Marching_cubes

The cutting plane will be formed by combining all these intersections (see Figure 3.3) to get the correct shape. This shape is then defined as a set of points. All these intersections with the corresponding orientations are stored before moving on to the next Marching Cube. Since for this algorithm simple polygons are used, it is easy to use for the graphics hardware and the 3D object can therefore be quickly displayed.

3.3.3 Smoothing

When a polygon has been created, it is highly likely that the surface is not equal on all points. This can be solved by smoothing out all irregularities. To do so a smoothing algorithm is used.

Before finding the current algorithm, many options have been looked in to. First there has been searched for a available Python implementation of mesh smoothing. It turns out that although there are numerous MATLAB implementations for smoothing available, only several Python implementations are published. The one that had been found was unfortunately not usable for this implementation. It was too large and still after removing all unnecessary elements, it was inefficient to apply it to the project.

An other option was to try to use the smoothing package that was used with the MATLAB implementation. This package is written in C/C++ and makes uses of MEX-files. These files are used to call C functions from MATLAB. To create these files you need to understand C or C++ and a MATLAB editor, since these files can be created within that editor. To compile the file a compiler supported by MATLAB is needed as well as the C/C++ Matrix Library API, the C MEX Library API functions and a MEX build script. However, since there is decided to exclude MATLAB from this project, this was not an option. The MEX-files had to be translated to a form that could be used from Python code for instance by using Cython or Ctypes. One problem arose, which is that there was been made use of a specific MEX matrix, which was not easy to translate into a matrix that Python could use.

CGAL

Because no suitable Python implementation of smoothing was found, attention was turned to the Computational Geometry Algorithms Library (CGAL). CGAL is a software library of computational geometry algorithms. It is mostly written in C++, but there are Scilab bindings and bindings generated with SWIG available such that it supports Python and Java.

It provides efficient and reliable geometric algorithms in the form of a C++ library. It can be used for several purposes. For example, geographic information systems, molecular biology, medical imaging, computer graphics and robotics. Within the library data structures and algorithms like triangulations, point set processing, arrangements of curves, surface and volume mesh generation and shape analysis can be found.

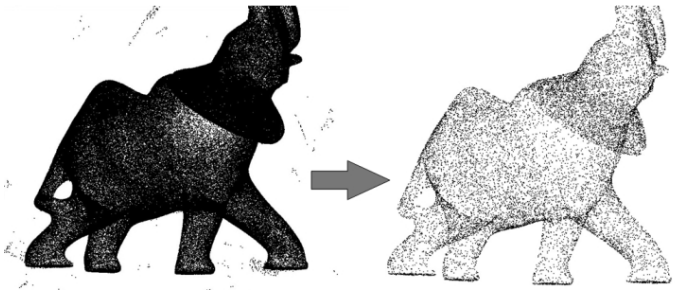


Figure 3.4: Left: 275K points sampled. Right: result of outlier removal, denoising and simplification (17K points).
source: http://doc.cgal.org/latest/Point_set_processing_3/index.html#title20

The algorithms that of interest for this project are those for smoothing, which can be found in `point processing`. This CGAL component implements methods to analyse and process 3D point sets. An example of the application of some methods is shown in Figure 3.4.

To use a CGAL algorithm in combination with Python it of course has to create some sort of binding. This is done by using SWIG [Bea17]. SWIG is a software development tool that connects programs written in C/C++ with a variety of high-level programming languages, including Python. It works by taking the declarations found in the C/C++ header files and using them to generate the wrapper code that scripting languages need to access the underlying C/C++ code. It is easy to install on Unix (Linux) computers and SWIG can be used for a variety of applications. For instance, for building more powerful C/C++ programs. When using SWIG the main function of a C program can be replaced with a scripting interpreter from which one can control the application.

Moreover, it allows C/C++ programs to be placed in scripting environment that can be used for testing and debugging as well as for controlling and gluing loosely-coupled software components together. Additionally, it can be used to turn common C/C++ libraries into components for use in popular scripting languages. Therefore, it is useful to use SWIG for making any C/C++ modules or libraries that perform smoothing on a data set applicable on Python

Based on these advantages over the advantages of writing a own smoothing code from scratch, it has been decided to first try to smooth the objects making use of some functions of CGAL.

Results and problems

The two CGAL functions that are of interest to this project are called `removing_outliers` and `jet_smooth_point_set`, both from the `point_set_processing` package [SW17]. `remove_outliers` computes average squared distance to the K nearest neighbors, and sorts the points in increasing order of average distance. This method reorders the input points in such a way that the remaining points appear first. Then an iterator goes over these points and removes any unnecessary points. `jet_smooth_point_set` smooths the input point set by projecting each point onto a smooth parametric surface patch (so-called jet surface) fitted over its K nearest neighbors.

Applying both these functions, first `remove_outliers` followed by `jet_smooth_point_set`, decreased the amount of points significantly and resulted in a smoothed surface. The only problem that arose from using CGAL was that it was not possible to create a mesh from these new points. The point cloud could not be converted to a mesh such that the smoothed zebrafish could be illustrated, thus an other solution had to be found.

PyPoisson

The final solution was PyPoisson [Kaz15]. This method is based on the poisson distribution which is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time and/or space. PyPoisson is a Python Binding of Poisson Reconstruction and is written by Miguel Molero-Armenta [Kaz15].

Previous research [Cao14] has shown that applying Poisson reconstruction to a point cloud to create the corresponding surface reconstruction is the most convenient in the field of biology. Instead of using the data points it is associated with the so-called ambient space resulting in a well-conditioned system due to the fact that it also makes use of a simple hierarchical structure.

The smoothing works as follows, first all points and normals must be extracted from a .xyz-file, then this data is used to determine the faces and vertices to apply poisson reconstruction. The result, which is an array, can be written to a .ply-file format.

This method is easily applicable to the existing code, since CGAL required a .xyz-file for their methods, there was no need to create a new .xyz-file with the data in it; it was already in correct data format. Moreover, the visualisation is not hard, since there various scientific data visualisation tools available which require .ply-files as input.

3.4 Visualisation

After creating vertices and faces using PyPoisson a new mesh had to be calculated and visualised. For this there were several options available.

The first and most basic visualisation tool that has been used is Matplotlib [DtMdt17]. It is a Python 2D plotting library which include a MATLAB-like interface. One has full control the settings (e.g. the line styles, font properties, axes properties). The mesh is created using the function `Poly3DCollection`. This creates a collection of 3D polygons using 3D coordinates which can be created using vertices and faces (see Figure 3.5).

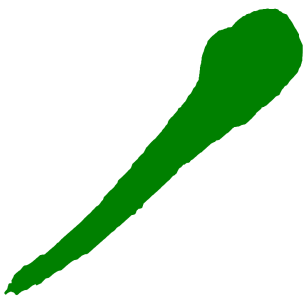


Figure 3.5: A 3D model of a zebrafish visualised with Matplotlib.

Another visualiation tool is the Visualisation Toolkit (VTK) [ML93] which is an open-source, freely available software system for 3D computer graphics, image processing and visualisation. It supports a variety of visualisation algorithms such as mesh smoothing, cutting and Delaunay triangulation.

For creating the mesh Delaunay triangulation has been used. This function maximises the minimum angle of all the angles of the triangles in the triangulation for a set of points. Thus a tetrahedral mesh is created from unorganised points. Again vertices and faces should be used, this time as the set of unorganised points.

Figure 3.6 shows the result of applying VTK to the Python code.

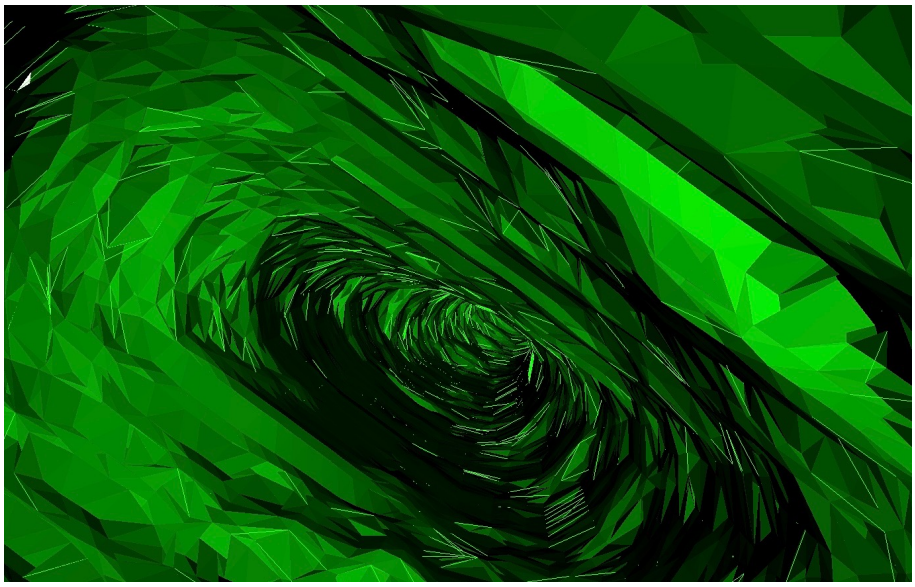


Figure 3.6: Inside of a zebrafish 3D model visualised with VTK.

3.4.1 Matplotlib versus VTK

MatplotLib gives a user more freedom in specifying the grid and other parameters, but VTK is significantly faster and its interaction is smoother as well. Whenever one tries to move the zebrafish in Matplotlib it does not react instantly. The zebrafish moves a bit clumsy, where VTK interacts very neatly and it does not show any disruptions when moving the zebrafish.

As said earlier, it depends on the user which visualisation tool is desired, but as shown above there are several options available for this code.

Chapter 4

Evaluation

A set of experiments have been performed to investigate the effect of parallelization of the implementation. Moreover, the effect of making use of a Python implementation instead of the MATLAB implementation has been carefully looked into.

One should take into account that these results are dependent on the conditions of the computer(s), which means that redoing to experiments could result in a different outcome. That is, only the execution time since the results of the calculations will always be indistinguishable.

4.1 File sorting experiment

To get a better insight in how the cluster computer divides tasks over available nodes, a test job script has been written and run on the cluster computer. The goal of the task is to sort an input file.

At the beginning, the input file is unsorted and it is alphabetically sorted after running the script. To get a broad view on how the cluster computer works several tests have been performed with different parameters. One is the size of the input file which is increased in size by iterating the same file a certain amount of time. e.g. iterating 1000 times gives a file size of roughly 1.1 GB. The number of iterations will be referred to as the scale factor.

The other parameter is the amount of nodes used to perform the task, respectively, 2, 4, 8, 16 and 32. To compare these amounts of nodes the execution time is measured on different timestamps. For instance, as soon as the job script starts running and thus iterating the files and write the output to a file. Following, the new input file must be sorted, that is also timestamped. Finally, the results must be merged together and this time is also measured.

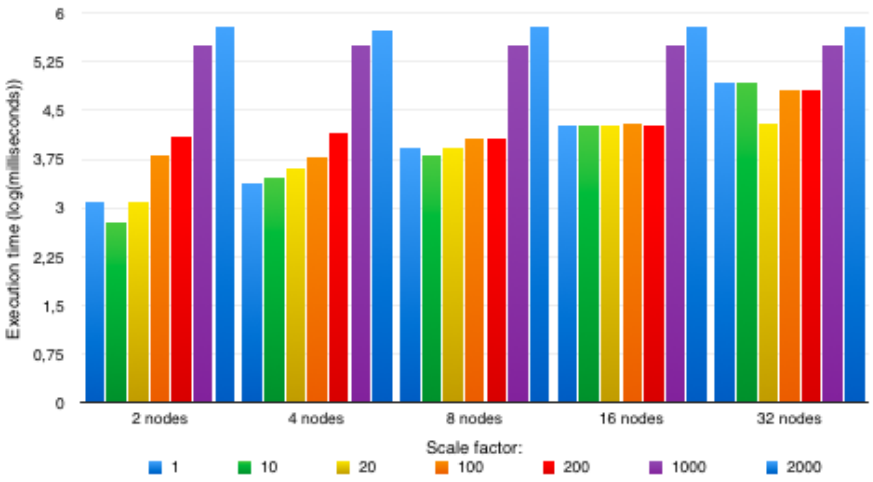


Figure 4.1: Results of first experiments on unsorted data set

In Figure 4.1 is shown how a different amount of iterations relates to a set amount of available nodes and results in a rate. A smaller chart is obviously better, since it means that a smaller amount of time is needed to do the same job.

4.2 Performance of the Python translation

Before parallelising on LLSC first there was a need to compare the MATLAB implementation with the Python implementation on a single computer. This to detect any possible improvements and to make a good comparison of both codes. To investigate this both codes have been run separately, with time measurement for execution time and determining where parallelisation is applied or could be applied. For these experiments a data set of fifty zebrafish is used.

The computer used for these experiments has the following specifications:

- Architecture: $x86\_64$
- CPU: Intel(R) Core(TM) $i7-2600$ CPU @ 3.40 GHz
- MemTotal: 16 GB

To make sure both codes work the same, that is the same data sets gives the same results, the values of the volume and surface area of both codes for every zebrafish has been compared. Previous research has set [SV17] values for volume and surface area allowing only minor loss of precision during calculation(s).

One small comparison is shown in Table 4.1. Other research results, such as the volumes and surface areas, can be found in Chapter 4.4. For these experiments a selection of seven zebrafish was made from the full dataset of fifty zebrafish.

	MATLAB	**Python**	**Python(multi-threaded)**
Zebrafish 0	20,241	69,322	39,300
Zebrafish 11	19,458	76,374	44,850
Zebrafish 13	20,359	72,474	42,966
Zebrafish 15	19,262	73,085	43,172
Zebrafish 31	19,343	74,419	44,207
Zebrafish 33	19,877	72,933	44,043
Zebrafish 35	19,788	72,695	43,459
Average per zebrafish	19,761	73,043	43,142
Total elapsed time	979,382	3712,585	2182,418

Table 4.1: Execution time in seconds for different implementations

It takes on average 17,742 minutes to run the MATLAB-code for the given data set. Python takes more time to do the exact same thing, namely 61,024 minutes on average. At first sight the MATLAB-code seems to be faster than the Python code, namely at least three times as fast as the Python code. However, this could be due to the fact that larger pieces of the MATLAB-code are parallelised than (pieces of) the Python code. Monitoring the MATLAB process with `top` has indeed confirmed that parts of the program are executed using multiple threads. `top` shows that for sampling the data of the fish (thus probably for reconstructing the 3D object and applying smoothing) all CPU percentages are above 230%. Which means that at least some multi-threading is applied in the MATLAB code. The highest CPU percentage measured is 270.8%. After applying multi-threading the Python code is faster than before (namely now only 36,374 minutes are needed to complete the task), but it is still slower than the MATLAB implementation.

4.3 Analysis and optimisation of the Python implementation

The Python code seems to be significantly slower than MATLAB code, but this could be due to the fact that the code for doing most of the computations in MATLAB are parallelised, whereas this is not the case for the Python code. The majority of the code uses NumPy, which does not make use of multi-threading [PJ17]. Monitoring the process with `top` confirms this and also indicated that the smoothing step does make use of threading, as the CPU utilisation clearly spiked to over 700% at this time. After a quick study of the source code, there was found that PyPoisson uses a C library, which in turn uses OpenMP to implement multi-threading.

In order to identify where most is spent in the Python implementation, a set of fifty zebrafish was processed with this code. Five major components where identified in the Python implementation of which the execution time was measures. The results of these experiments are shown in Figure 4.2. For these experiments a selection

of five zebrafish was made from the full dataset of fifty zebrafish.

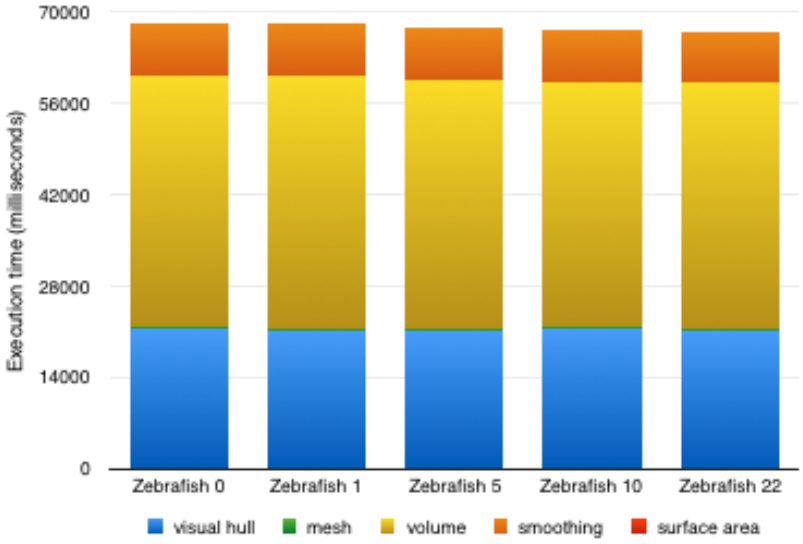


Figure 4.2: Execution time of several tasks on subset of zebrafish data set using Python code

In Figure 4.2 it is easy to see which parts of the code are really time-consuming at this moment. That is, when the Python code is single-threaded. Smoothing (●) is already multi-threaded, but it could use some improvements (see table 4.2). However, since smoothing already uses C codes the expectation is that larger performance improvements are obtained by looking at the Python code. The mesh (●) and surface area (●) are almost not visible in Figure 4.2. In Table 4.2 one can see that the execution time for these tasks is already relatively small, thus the biggest improvements could be found in optimising the codes for visual hull (●) and volume (●).

Table 4.2: A selection of the data with CPU percentage and execution time using Python code

Function	**Time (ms)**
zebrafish 0	
visual hull	21384
mesh	393
volume	38377
smoothing	7967
surface area	88
Total elapsed time	69.322
zebrafish 1	
visual hull	21236
mesh	384
volume	38524
smoothing	8133
surface area	88
Total elapsed time	74.484
zebrafish 5	
visual hull	21110
mesh	381
volume	38051
smoothing	8028
surface area	83
Total elapsed time	73.703
zebrafish 10	
visual hull	21343
mesh	381
volume	37457
smoothing	8251
surface area	85
Total elapsed time	73.576
zebrafish 22	
visual hull	29202
mesh	385
volume	37562
smoothing	7651
surface area	79
Total elapsed time	80.977
zebrafish 48	
visual hull	21226
mesh	386
volume	37778
smoothing	7645
surface area	80
Total elapsed time	73.178

4.3.1 Multi-threading

To improve the execution time the Python code has been adjusted in such a way that multi-threading is applied. A multi-threaded program distributes the computation over multiple threads, that can run in parallel on a multi-core computer.

For every zebrafish multi-threading has been introduced for calculating the volume and for visual hull, since there has been decided to improve these parts of the code in Section 4.3. Applying multi-threading is done by using the built-in `multiprocessing` package [Fou17b] in Python, which allows to leverage multiple processors.

The performance for one core is already known (see Section 4.3). The other amounts of cores that have been tested upon are four and eight. The size of the data set has been determined as well as the size of the cores, which is either four or eight. Then a division of the data set over the available cores has been made followed by using the `pool` function from `multiprocessing` package which is used to execute the multi-threaded code. Every node has a result and these results are merged together in a `RawArray` which is used further in the code.

Table 4.3 shows that the difference between four or eight cores is not significant. Four cores need a total of 36,37 minutes where eight cores need 40,08 minutes to process fifty zebrafish. However, since eight cores probably need more time to merge all separate answers together, the difference could become larger when the dataset is larger. Therefore, four cores are used for multi-threading. Moreover, there are nodes on LLSC that only are equipped with four cores, with that in mind it is in this case better to use four cores.

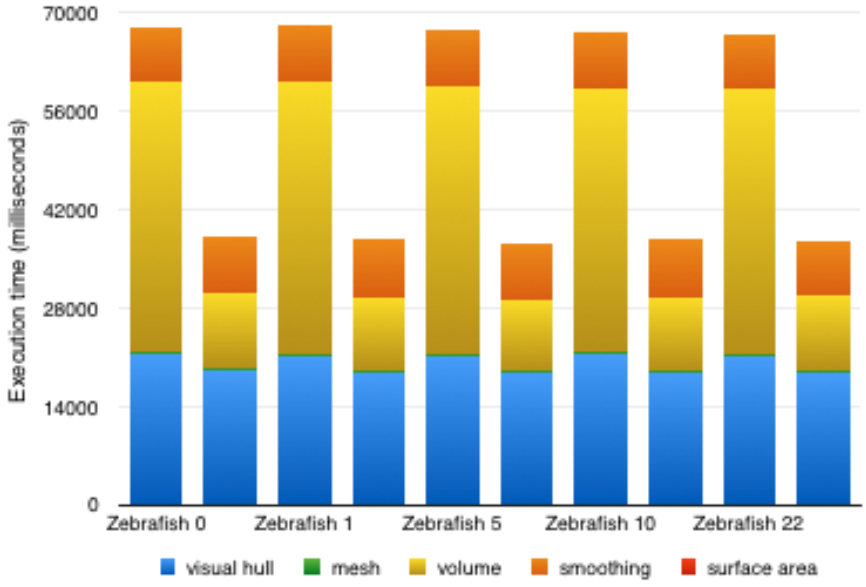


Figure 4.3: Execution time of several tasks on subset of zebrafish data set using normal and multi-threaded Python code

Figure 4.3 shows the improvements when applying multi-threading to the Python code. For these experiments a selection of five zebrafish was made from the full dataset of fifty zebrafish. Every left chart belongs to the single-threaded implementation and every right chart belongs to the multi-threaded Python code. As visible in this figure, visual hull (●) still takes a long time to calculate relatively to the other components. However, calculating the volume (●) now takes less time than before.

Table 4.3: A selection of the data with execution time using two different amount of nodes for multi-threading

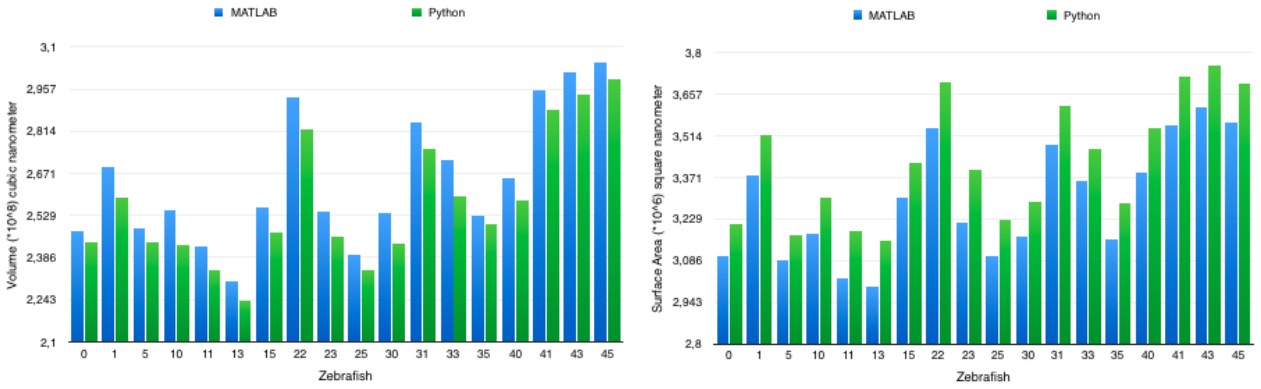
Function	**Time (ms) (4 nodes)**	**Time (ms) (8 nodes)**
zebrafish 0		
visual hull	19233	23893
mesh	390	442
volume	10449	10084
smoothing	8034	8664
surface area	90	89
Total elapsed time	39.300	44.445
zebrafish 1		
visual hull	18680	21876
mesh	387	396
volume	10465	10011
smoothing	8184	12771
surface area	87	88
Total elapsed time	43.043	51.404
zebrafish 5		
visual hull	18655	21421
mesh	395	395
volume	10276	11058
smoothing	8020	25276
surface area	91	94
Total elapsed time	43.578	64.437
zebrafish 10		
visual hull	18732	22323
mesh	390	405
volume	10386	10857
smoothing	8219	10821
surface area	92	91
Total elapsed time	43.932	50.764
zebrafish 22		
visual hull	18806	21523
mesh	400	390
volume	10490	9946
smoothing	7920	7829
surface area	84	82
Total elapsed time	43.813	45.928

4.4 Matlab versus two Python implementations

This section points out the similarities and differences between the MATLAB implementation, the single-threaded and the multi-threaded Python implementation. For these experiments the same fifty zebrafish are being processed and a selection of eighteen zebrafish was made from the full dataset.

As one instantly can see in Figures 4.4a and 4.4b is the values are not the same, otherwise the graphs would fit over each other. The volumes calculated by the MATLAB code (●) are larger than the Python values (●). Another thing that instantly arises is that the surface areas calculated by the Python code are larger than the surface areas calculated by the MATLAB code. This can be caused by the smoothing methods that are used.

Figure 4.4: Comparison of data outcomes using MATLAB and Python



(a) Resulting volumes

(b) Resulting surface areas

More detailed results can be found in Tables 4.4 (volume) and 4.5 (surface area). What also can be concluded from these tables is that applying multi-threading does not affect the values in any way, apart from the execution time of course. The values for the volume as well as for the surface area calculated by the normal and multi-threaded Python code are identical.

As stated in Section 4.2 it takes MATLAB on average 17,742 minutes to process fifty zebrafish whereas the single-threaded Python implementation needs on average 61,024 minutes to do the same job. Optimising the Pyhton code using multi-threading with four cores gives an average total excution time of 36,374 minutes. Using eight cores gives an average total execution time of 40,082 minutes. This means that the Python code on a single computer, despite using multi-threading, is slower than the MATLAB implementation. However, using multi-threading certainly improves the execution time significantly, when comparing only the two Python implementations.

Table 4.4: Comparison of results for calculating volume ($10 * 10^8$) cubic nanometer

Zebrafish	MATLAB	Python	Python (multi-threaded)
0	2.477170	2.437420	2.437420
1	2.693582	2.587302	2.587302
5	2.483786	2.437103	2.437103
10	2.548626	2.429676	2.429676
11	2.422483	2.344197	2.344197
13	2.305434	2.238364	2.238364
15	2.555273	2.468540	2.468540
22	2.928651	2.821908	2.821908
23	2.542516	2.455446	2.455446
25	2.393413	2.343854	2.343854
30	2.534966	2.433747	2.433747
31	2.845151	2.754088	2.754088
33	2.715302	2.595252	2.595252
35	2.526849	2.500100	2.500100
40	2.656178	2.580429	2.580429
41	2.951917	2.888001	2.888001
43	3.012807	2.938247	2.938247
45	3.045866	2.990190	2.990190

Table 4.5: Comparison of results for calculating surface area ($10 * 10^6$) square nanometer

Zebrafish	MATLAB	Python	Python (multi-threaded)
0	3.098979	3.213192	3.213192
1	3.377336	3.516316	3.516316
5	3.085464	3.174252	3.174252
10	3.176168	3.302766	3.302766
11	3.025841	3.186155	3.186155
13	2.995542	3.155883	3.155883
15	3.300209	3.423046	3.423046
22	3.542274	3.697346	3.697346
23	3.214486	3.398429	3.398429
25	3.100230	3.225498	3.225498
30	3.166127	3.289559	3.289559
31	3.482716	3.616092	3.616092
33	3.358839	3.467708	3.467708
35	3.157950	3.280893	3.280893
40	3.389111	3.540596	3.540596
41	3.551147	3.715781	3.715781
43	3.610835	3.757380	3.757380
45	3.558909	3.695915	3.695915

4.5 Cluster computer versus single computer

As seen in Section 4.3.1 applying multi-threading to the code shows improvements in the execution time. To test whether executing on a cluster computer gives even more improvement in the execution time, the whole data set has been put onto LLSC including the required Python programs. These are the same code as in the previous chapters, thus where multi-threading on four nodes is applied on both the calculations for the volume and the visual hull of the zebrafish. For these experiments a data set of fifty zebrafish is used and a selection of five zebrafish was made from the full dataset of fifty zebrafish.

Now, to increase the speed even more, multi-processing is introduced. That is, on the cluster computer multiple nodes are being allocated to process multiple zebrafish at the same time. Every zebrafish itself is thus multi-processed and multi-threaded. Since a zebrafish has no dependencies on other zebrafish, it is easy to process multiple zebrafish at the same time. Also, to make sure there will not be dependencies created by using multiple computers for multiple zebrafish at the same time, there has been chosen to process each zebrafish only on one computer (node).

4.5.1 Method

Before even thinking about which amount of nodes is suitable, first there had to be made sure that all outcomes are the same. This because multi-processing should not affect the values for the volume and surface area in any way. Experiments show that the amount of nodes, does not affect the values for volume and surface area.

A job script is used to divide data over the available nodes. First the size of the data set and the set of available nodes are determined. Then the data is divided over the available nodes, where each node gets assigned an

unique list of integers. Each integer represents the index of a file in the data set. Finally, the code runs where every node only processes their uniquely assigned zebrafish.

Each result of a zebrafish is being written to an output file, that is the values for the surface area en the volume of the zebrafish. Before multi-processing the order of the zebrafish was preserved, at this moment multiple computers (nodes) are using this output file at the same time. This causes to lose the strict order, but this can easily be solved by sorting the output file after all nodes have processed their data.

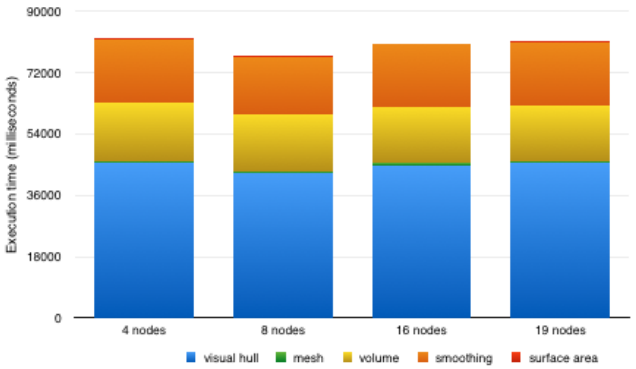
4.5.2 Results

Experiments has been done to find out to which level the amount of processors influences the execution time. This amount is limited, since there are only 19 nodes available at LLSC. Also, the baseline had to be determined, that is running the complete data set on only one node. This results in an execution time of 73,32 minutes. That means that of only one node is being used, it is better not to use LLSC and just use a single, more powerful computer. The execution time will then be around only 61,02 minutes, since it uses a more powerful processor with a higher clock speed.

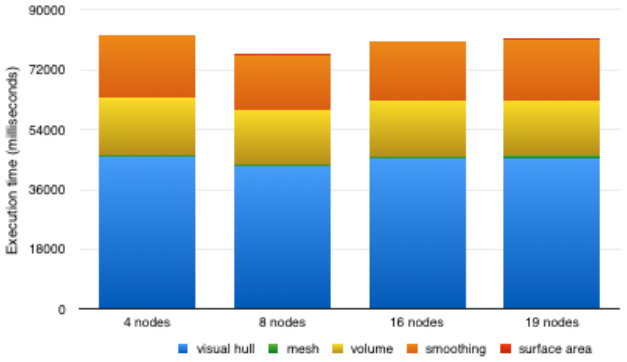
It has been chosen to investigate the execution time on 4, 8, 16 and 19 nodes. In Table 4.5 one can see that the amount of nodes almost does not influence the execution time per zebrafish; the difference is modest.

Interestingly, the average execution time per zebrafish with respect to the execution time per zebrafish on a single computer has increased. Before using multi-processing, when the code was only multi-threaded it needed on average 42,733 seconds (see Table 4.6), but now using multi-processing these averages are almost doubled in size. All steps need more time to execute, but calculating the visual hull is largest cause for this increase of the execution time. This could be due to the fact that multiple cores will do this calculation and therefore could be waiting for the other cores to finish and merge the results together. For 4 nodes the average execution time per zebrafish is 81,604 seconds, for 8 nodes the average execution time per zebrafish is 77,896 seconds, for 16 nodes the average execution time per zebrafish is 78,800 seconds and for 19 nodes the average execution time per zebrafish is 80,265 seconds.

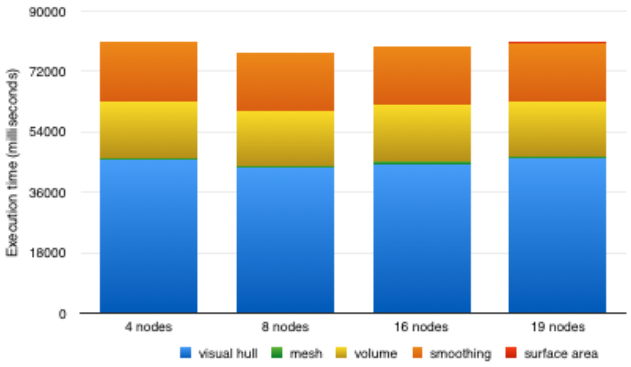
Figure 4.5: Execution time of tasks on subset of zebrafish data set using Python code and 4, 8, 16 or 19 nodes



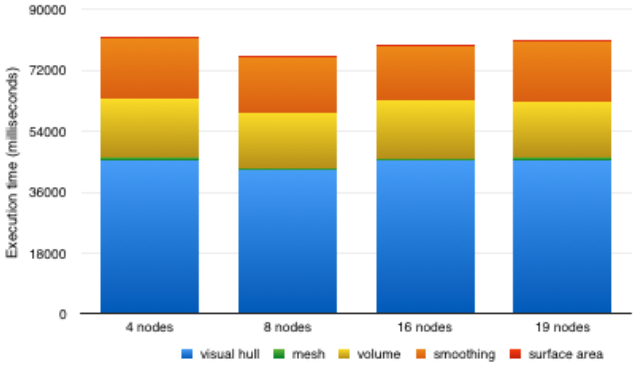
(a) Zebrafish 0



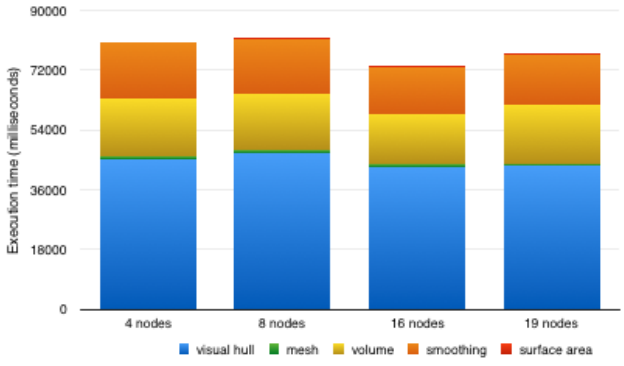
(b) Zebrafish 1



(c) Zebrafish 5



(d) Zebrafish 10



(e) Zebrafish 22

Real differences can be found when assessing the total execution time. The total execution time has improved with the introduction of multi-processing, regardless of the amount of nodes used. Before multi-processing the total execution time was 36,374 minutes (using multi-threaded Python code on a single computer), now it has improved to either 19,21 minutes (4 nodes), 10,44 minutes (8 nodes), 5,56 minutes (16 nodes) or 4,34 minutes (19 nodes). Moreover, running the multi-threaded Python code on one single node results in a total execution time of 73,32 minutes. Thus, using multiple nodes is needed to outperform the single computer. This is due to the fact that different types of processors are used on the single computer and LLSC.

The next experiment investigates the relation between using different amount of nodes or the baseline, which is one node. Figure 4.6 shows a linear relationship between the nodes and the total execution time. This means, that if more nodes are used on LLSC, the execution time could improve even more. For example, if 38 nodes are used, then according to Figure 4.6 approximately 1 minute is needed to process all zebrafish and a expected speedup of 32.

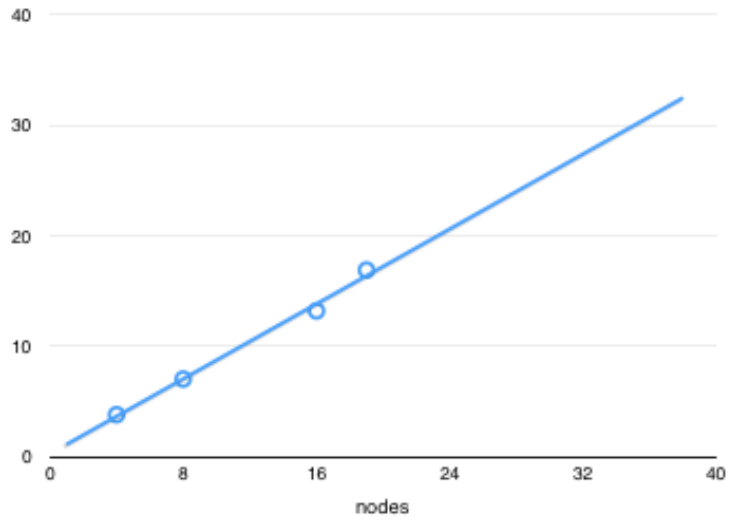


Figure 4.6: Speedup using Python code and 4, 8, 16 or 19 nodes compared to one node

Based on total execution time, it is better to use 16 or 19 nodes over 4 or 8 nodes for a data set of fifty zebrafish. Between the first two there is no significant difference, where the total execution time of the latter becomes larger. If the data set would be larger, for example two hundred zebrafish are being used, then the execution time would most likely be four times as large as it is now. This is because the new data set is four times larger than the one used in the experiments and Figure 4.6 shows a linear relationship between the amount of nodes and the speedup.

The total execution time has improved compared to the total execution time of the non multi-processed code. However, it could be improved even more if more nodes would be available on the LLSC, the more, the better.

Table 4.6: A selection of the data with execution time using multi-threading and four different amount of nodes for multi-processing

Function	**Time (ms) (4 nodes) single computer**	**Time (ms) (4 nodes) multi-processed**	**Time (ms) (8 nodes) multi-processed**	**Time (ms) (16 nodes) multi-processed**	**Time (ms) (19 nodes) multi-processed**
zebrafish 0					
visual hull	19233	45524	42584	44897	45478
mesh	390	593	557	592	591
volume	10449	17149	16464	16537	16341
smoothing	8034	18461	16993	18200	18397
surface area	90	304	277	294	292
Execution time	39300	82031	76875	80520	81099
zebrafish 1					
visual hull	18680	45695	42827	45175	45528
mesh	387	596	558	592	642
volume	10465	17172	16587	16761	16468
smoothing	8184	18802	16503	17969	18461
surface area	87	308	283	307	307
Execution time	43043	82573	76758	80804	81406
zebrafish 5					
visual hull	18655	45515	43187	44462	45962
mesh	395	589	553	585	562
volume	10276	17210	16665	17017	16392
smoothing	8020	17489	17174	17338	17635
surface area	91	272	255	264	318
Execution time	43578	81075	77834	79666	80869
zebrafish 10					
visual hull	18732	45610	42547	45510	45588
mesh	390	570	553	587	642
volume	10386	17457	16485	17109	16561
smoothing	8219	17925	16480	16179	17820
surface area	92	288	257	333	271
Execution time	43932	81850	76322	79718	80882
zebrafish 22					
visual hull	18806	45409	47255	43048	43200
mesh	400	596	594	525	556
volume	10490	17394	16876	15410	17683
smoothing	7920	16843	16712	14059	15409
surface area	84	250	256	251	221
Execution time	43813	80492	81693	73293	77069
Total execution time (m:s)	**36:37**	**19:21**	**10:44**	**5:56**	**4:34**

Chapter 5

Conclusions and discussion

5.1 Conclusions

In this thesis it has been researched how applying multi-threading and multi-processing (on a cluster computer) to Python code affects the execution time of a dataset containing 3D axial views of zebrafish. Experiments have shown that this distribution definitely yields significant speedups.

Before applying multi-threading and multi-processing to Python code, a translation from a MATLAB implementation for reconstruction zebrafish to a Python implementation had to be made. The execution time for fifty zebrafish using the MATLAB code is 17,742 minutes and for the Python code it is 61,024 minutes.

Applying multi-threading to the Python code alone improves the single-threaded code substantially. Not all code needed to be multi-threaded to improve the execution time, only the parts of the code that took significant more time to process compared to other parts. After multi-threading these bottlenecks in the code, the execution time improved significantly; from 61,024 minutes to 36,372 minutes for a data set of fifty zebrafish.

Additionally, multi-processing this code onto LLSC gives magnificent improvements compared to the single-threaded Python code. Not all optimisations result in a enormous improvement, but all together it really is noticeably improved. Now, the execution time for processing fifty zebrafish is 4,34 minutes (when using nineteen nodes) down from 73,32 minutes (when using one node).

The maximum amount of nodes that could be available at LLSC, that is when no other job is running, is nineteen. Thus when the LLSC is not completely free for use, that is, when other jobs are running at the same time and occupying the nodes. Then it could be that there are only only sixteen nodes, or even eight nodes, available, in that case the execution time, respectively 5,56 and 10,44 minutes, is smaller than when not using multi-processing at all. Worst case scenario; there are only four nodes available, then still the execution time is small (19,21 minutes).

5.2 Future work

The results described in this thesis are astonishing, however there is still room for improvement.

The translation of the MATLAB code to the Python code is mostly one-on-one. This means that there is room for improvement here. It is highly likely there are techniques and methods available in Python that are not present in MATLAB. Thus, the Python code must be analysed and adjusted to improve its quality and performance.

Moreover, there are some methods used at the moment that could be rectified. Firstly, at this moment all voxels are being used to determine the shape of the zebrafish. Thus, to improve speed, only the outer voxels could be used to reconstruct the shape of the zebrafish. Then the shape could be filled, or stay empty, and the surface area and the volume can be calculated. Another option to improve the execution time when calculating the visual hull and the volume, is to split the 3D model into slices and apply the surface area and volume calculations on these slices. The result of all slices must be merged together to get the final answer.

Another improvement, as stated in Section 4.3, is improving the execution time for smoothing. It is written in C, thus it must be investigated whether this C code can be further optimised. Writing a smoothing code in Python is not really an option, although Python has smaller source sizes, C is usually faster in doing computations.

At this moment multi-processing is implemented as follows, multiple zebrafish are being treated at the same time and every zebrafish is designated to one computer. Further research could point out whether it is more convenient to process one zebrafish on multiple computers at the time or that it is better to process each zebrafish on one computer. Processes that could benefit from this are for instance the volume and visual hull calculations, since they both use multi-threading. Thus, if multiple computers could process these parts instead of only one computer using multiple cores, it could lead to different execution times.

Lastly, this project can be put into practice by creating a interface where for example scientists would like to do some calculations on their data (e.g. axial views). Scientists could create a folder with their data and use the interface to submit the folder to the LLSC. The code for the calculations is always present at the LLSC and stays the same at all time, the only missing (and different) part is the data folder. The code reads in the data files and processes them. After the code has been run the interface receives the results from the LLSC and gives back the results to the scientist in a clear, easily understandable overview.

Acknowledgements

I wish to thank various people for their contribution to this project. Firstly, I would like to express my very great appreciation to Kris Rietveld and Fons Verbeek, my research supervisors, for their patient guidance, enthusiastic encouragement and useful critiques and recommendations throughout the project. I would like to extend my thanks to Yuanhao Guo for his MATLAB implementation and for helping me out when needed. Finally, I would like to thank my parents for their overall support and encouragement throughout my life.

Bibliography

[Bea17] David M. Beazley. Swig. "http://swig.org/exec.html", May 2017. [Accessed 12-05-2017].

[Bou94] Paul Bourke. Marching cubes. "http://paulbourke.net/geometry/polygonise/", May 1994. [Accessed 08-05-2017].

[Cao14] Lu Cao. *Biological model representation and analysis*, chapter Evaluation of Algorithms for Point Cloud Surface Reconstruction through the Analysis of Shape Parameters, pages 63–90. May 2014.

[DtMdt17] John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team. Matplotlib. "http://matplotlib.org", May 2017. [Accessed 02-05-2017].

[FF90] James D, Andries van Dam, John F. Hughes Foley and Steven K. Feiner. Spatial-partitioning representations; surface detail. *Computer Graphics: Principles and Practice*, The Systems Programming Series, 1990.

[Fou17a] Python Software Foundation. Ctypes. "http://docs.python.org/2/library/ctypes.html", May 2017. [Accessed 28-04-2017].

[Fou17b] Python Software Foundation. Multiprocessing. "http://docs.python.org/2/library/multiprocessing.html", March 2017. [Accessed 15-06-2017].

[Inc17] Adaptive Computing Enterprises Inc. Torque. "http://adaptivecomputing.com/products/open-source/torque/", January 2017. [Accessed 06-02-2017].

[Kaz15] Michael Kazhdan. Pypoisson. "https://github.com/mmolero/pypoisson", October 2015. [Accessed 12-06-2017].

[Kla15] Simon R. Klaver. Analysing electron tomography with imod on the llsc, bachelor thesis. 2015.

[LC87] William E. Lorensen and Harvey E. Cline. Developing an integrated environment for opt image reconstruction. *ComputerGraphics*, 21(4):163–169, 1987.

[ML93] Will Schroeder, Ken Martin and Bill Lorensen,. vtk. "http://vtk.org", December 1993. [Accessed 08-06-2017].

[MS17] Stefan Behnel, Robert Bradshaw, Lisandro Dalcn, Mark Florisson, Vitja Makarov and Dag Sverre Seljebotn. Cython. "http://cython.org/#about", June 2017. [Accessed 28-04-2017].

[PJ17] Travis Oliphant, Pearu Peterson and Eric Jones. Numpy. "http://scipy.github.io/old-wiki/pages/ParallelProgramming", July 2017. [Accessed 16-06-2017].

[Sch14] Eric C. Schug. Libermate. "http://github.com/awesomebytes/libermate", May 2014. [Accessed 24-04-2017].

[SV17] Yuanhao Guo, Wouter J. Veneman, Herman P. Spaink and Fons J. Verbeek. 3d reconstruction and measurements of zebrafish larvae from high-throughput axial-view in vivo imaging. 2017.

[SW17] Pierre Alliez, Clment Jamin, Quentin Mrigot, Jocelyn Meyron, Laurent Saboret, Nader Salman and Shihao Wu. Point set processing. "http://doc.cgal.org/latest/Point_set_processing_3/", May 2017. [Accessed 18-05-2017].

[vdZ16] Dennis van der Zwaan. Developing an integrated environment for opt image reconstruction, bachelor thesis. 2016.