CS 267 - FINAL PROJECT REPORT

UNIVERSITY OF CALIFORNIA, BERKELEY

DEPARTMENT OF COMPUTER SCIENCE

Parallelization of Vasculature Tracing

Authors:
Théophile Sautory *
Numi Sveinsson *

Emails:
tsautory@berkeley.edu
numi@berkeley.edu

Date: May 10th, 2022

* *equally contributed.*

1 Introduction

In cardiovascular research, an important task is to simulate the blood flow in vessels. This plays a crucial role in studying damaged blood vessels, preventing and curing cardiovascular diseases.

To do the latter, researchers must first construct a surface mesh of blood vessels, in which they then run fluid simulations and restore blood flow. The surface mesh is often built from non-invasive 3D medical images. The images are segmented to locate the blood vessel, allowing to find its centerline, and then align subsequent images along their centerline. Once the whole centerline has been drawn, the mesh can be generated. This process is called tracing (or tracking) and has recently been shifting from a manual to an automatic implementation.

In this project, we explored the possibility of performing automatic vascular training using parallel computing techniques to reduce its total runtime from hours to minutes.

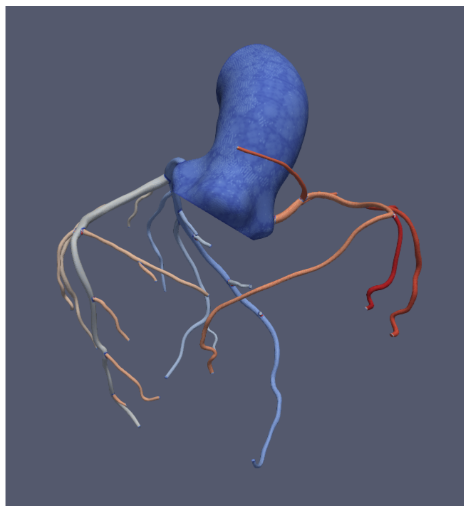

Figure 1: 3D Vasculature Model.

2 Problem Statement

The task of automatic vasculature tracing has a sequential nature. Indeed, the vessel is constructed as the algorithm progresses, and its exact length and topology are unknown beforehand. Moreover, they are specific to each patient, which prevents from introducing reliable biases in their shape. Hence, automatic vasculature tracing is traditionally implemented serially, which prevents real time model construction.

However, an important phenomena in vessel tracing is the phenomena of bifuraction. At some points along the artery, the vessel splits into two branches: 1. the continuation of the artery, 2. a coronary. The branches then evolve independently throughout the volume. This branching offers a potential avenue to introduce parallelism. Each branch can be worked on in parallel.

This sub-project lies in the scope of the SimVascular project. SimVascular is an opensource software package which provides the tools to perform patient specific blood flow simulation, building the whole pipeline starting from medical images segmentation. Developing an efficient tracing algorithm could lie at the heart of the SimVascular project and replace a tedious (when done by hand) and slow process (when done by hand or by a serial algorithm). Variants of it could be utilized prior to running blood flow simulations, which will then contribute to help patricians decision making when treating cardiovascular diseases.

One of our group members research focuses on automatic vascular tracing using a convolutional neural network for image segmentation, and has currently built a serial implementation of the algorithm. Hence our goal is to build a parallel vasculature tracing algorithm, answering the question:

Can we parallelize a tracing algorithm to enable a quasi instantaneous use of its output in a hospital?

In the next sections of the report, we will describe the computer architecture used, the serial code and initial parallelism ideas, before describing the steps we implemented, the results and their discussion.

3 Computer architecture

The implementation is ran on both an Intel i5 processor with 4 cores, and a Cori Haswell (Xeon) node with 16 physical cores on each of two sockets.

4 Serial code and initial ideas

After having defined the problem, we briefly describe the current serial implementation, and present a discussion on different potential parallel implementations. Note that we researched methodologies similar to A* algorithms for chess and parallel tree traversals and creation algorithms, as explored in [1]. We develop a method which has similarities with root parallelization.

4.1 Vasculature Tracing Strategies

Figure (2) presents three different approaches to the problem, where the color of the nodes refer to a unique processor.

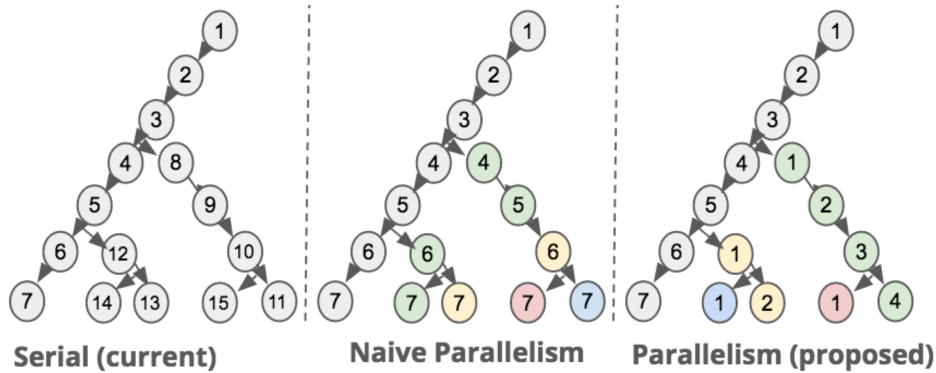

Figure 2: Potential implementations of the parallel algorithm. From left to right: Serial (the numbers refer to global timesteps), Naive Parallelism (the numbers refer to global timesteps), Proposed Parallelism (the number refer to local timesteps).

Each of the methods have specific particularities.

- *Serial:* A unique processor takes one step down one branch at a time and adds the potential branches (tasks) to a list. The single processor will sequentially perform all the tasks.
- *Naive Parallelism:* Multiple processors are used simultaneously and are synchronized with global timesteps. The processors add all new points to a shared memory queue, where new points are treated independently, i.e. no branch is assigned to a specific processor.
- *Parallelism Proposed:* Multiple processors act asynchronously, with local timesteps. The processors add new branches to a shared memory queue, from which they request tasks dynamically. Hence branches are assigned to a unique processor.

From these characteristics, it appears that the synchronization of the naive parallelism method could reduce the performance of our code, having idle processors. We explored small toy examples which revealed that the synchronization could increase the time vs. the asynchronous method by ~ 10%. This motivated us to pursue the exploration of the asynchronous version.

4.2 Serial Time Analysis

To understand the bottleneck of the serial algorithm, we explored the time distribution of the serial implementation. We show the time distribution per different functions in the serial algorithm in Figure (3).

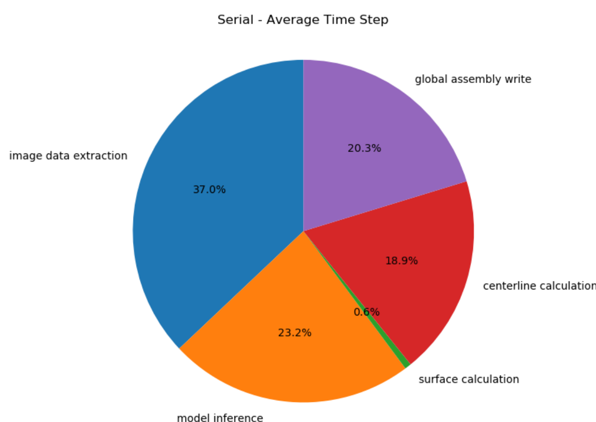

Figure 3: Average time per step in the serial implementation

Note that the term *step* refers to the whole processing of a medical image until the next one. Figure 3 revealed that the image data extraction and the global assembly write combined for ~ 57% of the time. These correspond to data movement tasks. The CNN model inference took ~ 23% of the time and the centerline calculation, which is a function form the *VMTK* package ~ 20% of the time per step. On the one hand, we assumed the model inference could be readily accelerated by using a GPU. However, in the scope of our question using the model in a hospital, we considered the introduction of a GPU harder to justify and support. On the other hand, the centerline calculation method in this algorithm will soon be replaced by a machine learning model prediction. Hence within the scope of this project and to explore the maximum of parallelization methods usefully for the bigger picture (hospital and SimVascular), we decided to tackle the parallelization of the data movement related tasks.

5 Parallel Implementation

In this section we describe our implementation of the proposed parallelism method, and the further optimizations to reduce the time spent in each data movement steps of the algorithm.

5.1 Base characteristics

Some characteristics of the implementation were shared between all parallel implementations.

The data structures used in the serial code we no longer consistent with our needs, as we required the code to be parallelized. Hence, we introduced a shared array for the global assembly volume, a shared queue of potential branches, and organized the vessel tree as a list of lists. In the serial code,

the global assembly volume was a 3D Numpy array, the potential branches and the vessel tree lists.

We managed the different processors of the computer with the *concurrent.futures* python library, which allows us to assign different tasks to different processors. As each processor is required to perform a model inference, we decided to initialize one model on each processor. We also initialized other relevant global variables.

Another challenge in parallel processing, common to all our optimizations, was to assign tasks to different processors. We viewed the problem as a task receiving a stream of data, and dispatching it to different processors. Hence, we implemented a while loop which dynamically checks if both the shared queue has waiting tasks, and if the there are any idle processors to which these can be assigned. The frequency of such checks is a parameter that we can tune in order to limit of the time spent with idle workers. In our experiments, we used a frequency of 10Hz.

We will now describe the specific changes that we introduced in each step, to accelerate the implementation and remove the serial code bottlenecks.

5.2 Shared memory with global lock

The algorithm requires to write down into a global assembly image volume. In the serial code, this is handled by a 3D Numpy array. However, the *multiprocessing* shared memory manager in Python does not handle Numpy arrays, nor multi-dimensional arrays (to the best of our knowledge, and after extensive research). We therefore created a 1D *multirocessing.Array()* object, which could be accessed by all processors. The latter is created with a global lock by default. Each processor is then able to write and read within this array safely. This 1D array represents the flattened version of the assembly image volume. We thus created a mapping from the desired 3D index position to the 1D position in the shared array using the *numpy.ravel_index* function. This first implementation allowed us to accelerate the serial code by harnessing the power of multiple cores. An example of the time distribution of this method is shown in Figure (4).

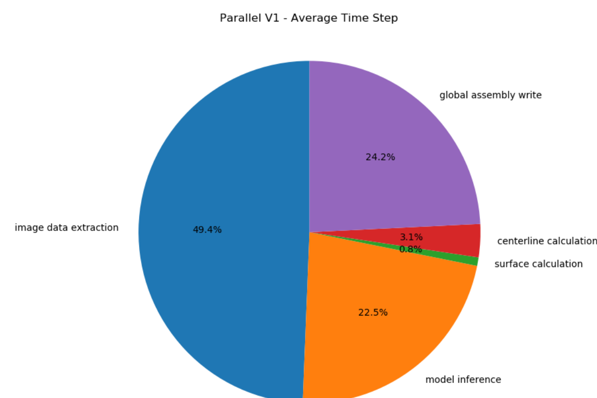

Figure 4: Average time per step in the shared memory with global lock implementation

Figure (4) revealed that with parallelization, the time taken by the algorithm for the data movement task increased to ~ 75%. This is due to both the slower access to the shared memory, and the fact that the global assembly image volume was a shared array with a global lock. Hence, the act of writing in the array remained serial. The bottleneck now appeared to be the image data extraction, however, we wanted to explore the writing parallelization even more.

5.3 Shared memory with local locks

In homework 2.2, we had not managed to completely implement an fully paprallelized OpenMP code. Indeed, our binning process had remained a serial bottleneck. We had not managed to introduce local locks to allow for multi-threading in the binning function. However, this provided us with key knowledge for our next optimization step: removing the global assembly image volume global lock and creating local lock for each processors to write in the array simultaneously.

Thus, in the following optimization step, we removed the default behaviour of the *multiprocessing* shared array by introducing the *lock=False* argument. Each processor created a *multiprocessing.Lock()* which it would *acquire* and then *release* before and after having written in the array respectively. Now, only sub-regions of the array were locked by each processor. This behaviour is even more desirable, as each processor is working on a different branch, which have little to no overlap. Thus, there should be little to no wait time for each processor to access it's desired array locations. An example of the time distribution of this method is shown in Figure (5).

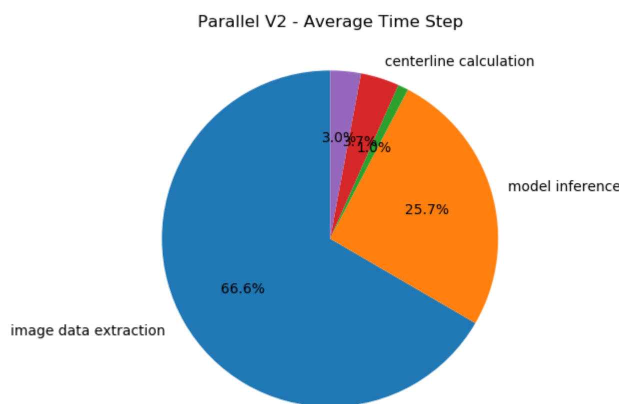

Figure 5: Average time per step in the shared memory with local locks implementation

Figure (5) highlighted that removing the lock did increase the performance of the writing time within the global array, dropping the time percentage from ~ 23% to ~ 3%. Even though the bottleneck was the data image extraction, we desired to push the data writing optimization even further. Indeed, we were not taking advantage of the spatial locality of the contiguous elements in the array. The processor was still iterating over each of the indices of its local array within a for loop.

5.4 Parallelizing within each processor's data writing

Moving the data from the global array to the local processor to perform a computation and update the value could be improved further. By exploring the positions of the 3D points of interest within the global flattened array, we realized that they were ordered according to slices within the z direction of our array. Hence, instead of accessing the array index by index serially, we were able to access whole slices of the array. This allowed us to reduce the number of times we had to fetch values within the array by a number proportional to the length of the local volume of interest in the z direction. We named this *chunking*, and essentially vectorized the computation manually, increasing the bandwidth and improving the efficiency of our algorithm. The idea of using such spatial locality stemmed from HW1, where we used SIMD instructions to accelerate matrix to matrix multiplications. An example of the time distribution of this method is shown in Figure (6).

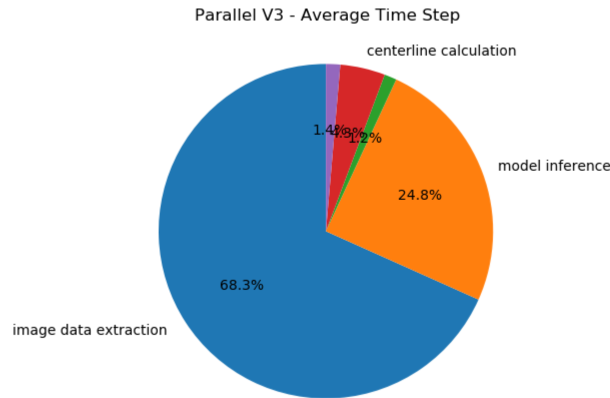

Figure 6: Average time per step parallelizing within each processor's data writing

Figure (6) showed that we managed to reduce the time of writing within the global array image volume from ~ 3% to ~ 1%. We then decided to tackle the new bottleneck: the extraction of each of the local sub-volumes of the CT image, that are then fed to the machine learning model for inference, to segment the vessel.

5.5 Parallelizing within each processor's data extraction

Motivated by the success of our paralellization of data writing for each processor, we applied the reverse process to data extraction. Instead of extracting the local volumes used for inference using *SimpleItk* functions, we created a shared flattened version of the full volume image. We then used a similar methodology to *chunking* to extract the data from the flattened image directly, removing our dependence on *SimpleItk* for that step. The new time distribution is shown in Figure (7).

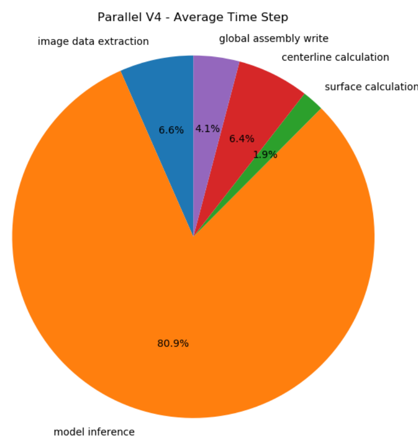

Figure 7: Average time per step parallelizing within each processor's data extraction

Figure (7) highlighted the idea that we successfully managed to remove the bottleneck in data extraction, from ~ 70% to ~ 7%. The reverse process seemed to be nearly as efficient as the writing process. The bottleneck now appeared to be the model inference. Unfortunately, we did not have the time to explore the removal of this bottleneck, which would have been performed by displacing the inference step to a GPU. We also deemed this step to be less relevant in the context of a hospital use.

5.6 Multi-Threading writing and extraction

Additionally to introducing parallelism through *multiprocessing* and vectorized calculations, we implemented *multithreading* using the *concurrent.futures* library to further accelerate the writing and extracting of data. However, we did not observe significant changes in the processing times, and did not have time to explore this avenue in depth.

5.7 Miscellaneous Parallelism Addition

Finally, we also applied *multiprocessing* to the pre-processing task performing data transformation to train the machine learning model. Doing so on a 4 cores processor CPU approximately reduced the time from 12 to 3 hours.

6 Results

The final results are shown in Figures (8-12), and discussed in Section 7. Figures (8) and (9) show the decreased cost of each step with the new implementations. Figures (10(and (11) show the strong scaling and Figure (12) shows the total time to run 108 steps for each of the working versions.

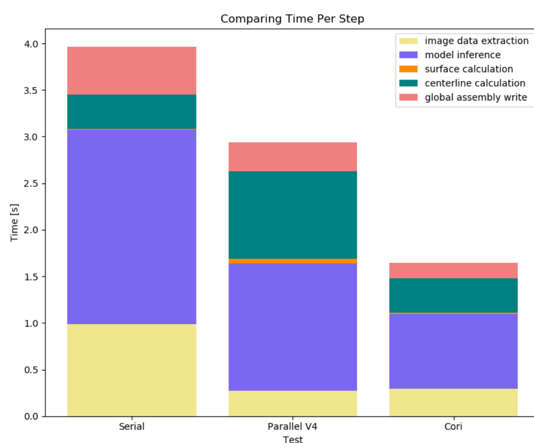

Figure 8: Time per step for different methods using a single processor for model inference.

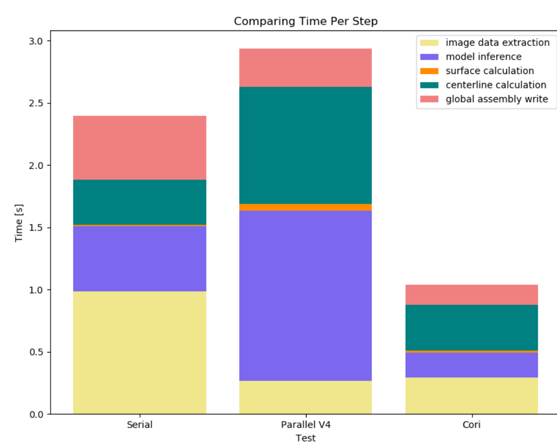

Figure 9: Time per step for different methods using multiprocessing for model inference.

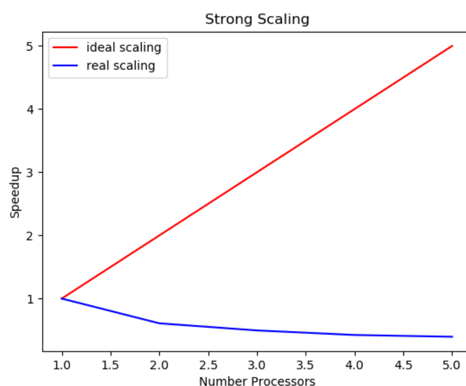

Figure 10: Total time to trace 108 steps using the different versions.

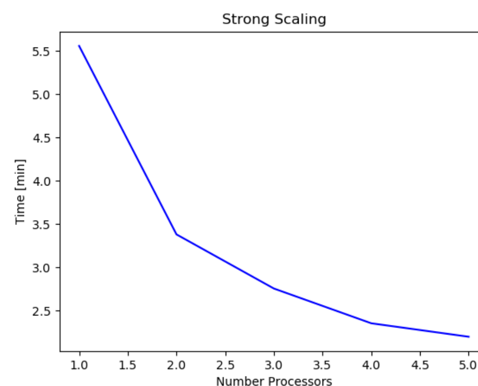

Figure 11: Strong Scaling. Tracing 320 steps with increasing number of processors.

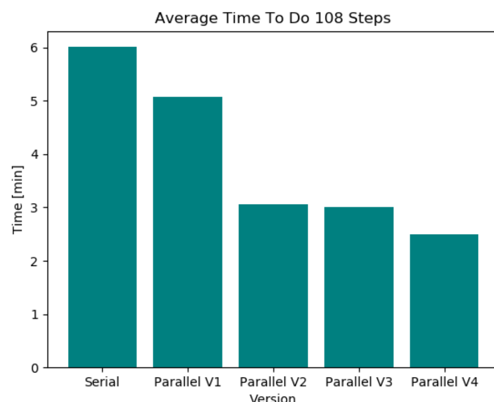

Figure 12: How long it takes the different versions to trace 108 steps.

7 Discussion

7.1 Results

The results show that parallelization worked to speed up the algorithm. Parallelization was implemented to 1) trace multiple branches at the same time 2) do model inference faster.

Figures (8) and (9) show that increased speedup was gained from 1) decreasing the data movement in data extraction and data writing to global assembly and 2) parallelizing the model inference. Like discussed previously, these two steps were the most expensive and decreasing their cost proved possible. Note that in these figures, these are average time per steps, however, in the parallel versions multiple steps are being taken at the same time. So speedup is gained from both decreased cost per step and taking multiple steps at the same time.

Figures (10) and (11) show the strong scaling of the final implementation. Even though it is far from ideal it does manage to decrease the model construction time significantly. This also shows room for improvement. We estimate that the reason behind the non-ideal strong scaling is partly because of the increased overhead of the parallel version. Longer experiments are needed to be done to see its effect dissipate. Figure (12) tells the story of the work. How the algorithm was optimized to decrease the cost by more than half on a local MacOS computer. Similar to what is available in a hospital.

This work was done mostly with the assumption that the model construction would occur on a PC computer in a hospital by a physician or healthcare worker. This implementation can be optimized further using extensive HPC abilities.

7.2 Brief comparison with reference works

In the proposal, we mentioned that the authors in [2] used a seeded region growing algorithm to perform vessel segmentation in 3D in liver slices, considering the inter-connectivty between elements. They implemented the whole algorithm on GPU achieving an 1.9x improvement in terms of speed. Though the task is different and the methods differ, we successfully showed that *multiprocessing* could also be used as a mean to parallelize vascular tracing.

We had also mentioned the work of [3], which introduces many seed points, and were concerned about the potential amount of time our processors would remain idle. However, the branches have numerous branches and the idle time for processors was smaller than ~ 3% in our experiments with 100 steps only. Moreover, as *Keras* allows to readily use multiprocessing for model inference, idle

processors which are finished with their branches will contribute to speeding up the model inference to running processors.

7.3 Future Works

The bottleneck of the method is now the model inference. A direct improvement could be obtained by performing the inference on a GPU, assuming the time to move the data from the CPU to the GPU would remain small compared with the gain.

The centerline calculation step could also be tackled. As the latter will probably changed to a model inference for more accuracy, the need of introducing a GPU could increase further. This must however be looked into in details due to the potential absence of a GPU in hospitals.

7.4 Challenges

Most of the challenges we encountered concerned the implementation of the *multiprocessing* shared memory. As we were working with Python 3.6 for dependencies reasons (with the *vtk*, *vmtk*, *SimpleITK* packages), some updates which made the latter simpler to use with the *concurrent.futures* module in Python 3.7. Transferring the implementation from our local computer to Cori was also challenging due to intertwined dependencies.

Overall, we truly enjoyed the whole class and the final project which allowed us to gain invaluable knowledge on a decisive topic for our future research careers.

References

[1] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:1:1–43, 03 2012. pages 3

[2] Nitin Satpute, Rabia Naseem, Rafael Palomar, Orestis Zachariadis, Juan Gómez-Luna, Faouzi Alaya Cheikh, and Joaquín Olivares. Fast parallel vessel segmentation. *Computer Methods and Programs in Biomedicine*, 192:105430, 2020. pages 9

[3] Jelmer Wolterink, Robbert van Hamersvelt, Max Viergever, Tim Leiner, and Ivana Išgum. Coronary artery centerline extraction in cardiac ct angiography using a cnn-based orientation classifier, 10 2018. pages 9