draft 1 manuscript No.
(will be inserted by the editor)

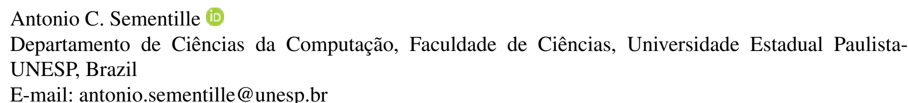
Optimizing a medical image registration algorithm based on profiling data for real-time performance

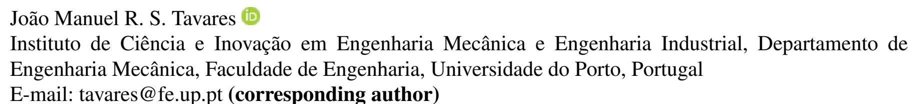
Carlos A. S. J. Gulo · Antonio C. Sementille · João Manuel R. S. Tavares

Received: date / Accepted: date

Abstract Image registration is a commonly task in medical image analysis. Therefore, a significant number of algorithms have been developed to perform rigid and non-rigid image registration. Particularly, the free-form deformation algorithm is frequently used to carry out non-rigid registration task; however, it is a computationally very intensive algorithm. In this work, we describe an approach based on profiling data to identify potential parts of this algorithm for which parallel implementations can be developed. The proposed approach assesses the efficient of the algorithm by applying performance analysis techniques commonly available in traditional computer operating systems. Hence, this article provides guidelines to support researchers working on medical image processing and analysis to achieve real-time non-rigid image registration applications using common computing systems. According to our experimental findings, significant speedups can be accomplished by parallelizing sequential snippets, i.e., code regions that are executed more than once. For the selected costly functions previously identified in the studied free-form deformation algorithm, the developed parallelization decreased the runtime by up to seven times relatively to the related single thread based implementation. The implementations were developed based on the Open

Carlos A. S. J. Gulo
CNPq National Scientific and Technological Development Council
Research Group PIXEL - UNEMAT, Brazil
Programa Doutoral em Engenharia Informática, Instituto de Ciência e Inovação em Engenharia Mecânica e Engenharia Industrial, Faculdade de Engenharia, Universidade do Porto, Portugal
E-mail: sander@unemat.br


Antonio C. Sementille
Departamento de Ciências da Computação, Faculdade de Ciências, Universidade Estadual Paulista-UNESP, Brazil
E-mail: antonio.sementille@unesp.br


João Manuel R. S. Tavares
Instituto de Ciência e Inovação em Engenharia Mecânica e Engenharia Industrial, Departamento de Engenharia Mecânica, Faculdade de Engenharia, Universidade do Porto, Portugal
E-mail: tavares@fe.up.pt **(corresponding author)**

Multi-Processing application programming interface. In conclusion, this study confirms that based on the call graph visualization and detected performance bottlenecks, one can easily find and evaluate snippets which are potential optimization targets in addition to throughput in memory accesses.

Keywords Medical image processing and analysis · Profiling tools · Performance analysis · Non-rigid image registration

1 Introduction

Medical image analysis plays a significant role in the field of medicine, and image registration is an important and widely used technique in this context. Today, patients are imaged on routine basis using different imaging systems. Patients are also monitored over time to assess disease progression or response to therapy. However, to be able to study physiological and/or structural changes over time, or to combine complementary information that different imaging systems produce, it is necessary to perform the registration of the acquired images [1]. Image registration is a computational task that determines the spatial correspondence between two images of the same object acquired at different angles, at different times, using different image modalities, or under different acquisition conditions [2, 3, 4]. In general, an image registration method can be decomposed into three parts: building a transformation model, computing a similarity measure and performing the optimization of the registration model [4, 5]. Transformation models, such as rigid or non-rigid models, delineate the transformation that can be used to represent the underlying correspondences. Rigid models describe simple linear mappings such as translations, rotations, scalings and shears. However, non-rigid transformation models can represent more complex mappings, since local deformations are also taken into account, resulting thus in longer computation times [6, 5]. Non-rigid image registration is an extensive research field, encompassing many applications and several specific algorithms. For example, the ones based on mutual information [7, 8], elastic transformation models [9], multi-resolution [10], and similarity measures [6]. However, the required computational effort is frequently high when a non-rigid image registration algorithm is used. Hence, this task is well-known as one of the most time-consuming tasks that can be found in medical image analysis [11, 12]. However, with the development of multi-core processor architecture, several solutions have been proposed that realize non-rigid image registration algorithms on multi-core CPUs [13, 14, 15]. Multi-core architecture mainly aim at improving the performance of highly demanded applications by exploiting parallelism. However, writing parallel algorithms from scratch is a very complex and demanding task. Furthermore, parallelizing legacy algorithms is even more challenging [16, 17, 18]. Fortunately, a profiling method can be effectively used to identify and evaluate portions of code responsible for consuming excessive computational resources [17, 18]. For example, a profiling tool can accurately count the activation instances of a function during runtime of an algorithm. Furthermore, it can provide timing information about the function [19]. Profiling is therefore a helpful approach in program optimization, which is based on gathering and

calculating data regarding memory space, frequency and duration of function calls, and time complexity of the algorithm under study. Many profiling tools, such as `gprof` [19], `perf` [20], `tiptop` [21] and others [22, 23, 24], have been proposed to help programmers identifying performance bottlenecks during the execution of algorithms on a CPU under a given workload [20, 17, 23].

In the presented study, we employed profiling tools to identify functions with long run-times in a popular image registration algorithm: the Free-Form Deformation (FFD) algorithm [12, 11]. Based on the collected profiling data, we carried out a performance analysis of the algorithm. In particular, we aimed to effectively decrease its processing time in order to adapt it to be feasible for real-time diagnosis. To this end, we exploited computational resources typically available in modern personal computers. We gauged "performance" by working out the operating systems efficiency during algorithms execution. This evaluation took into account the factors of throughput, latency, and availability. Thus, throughout this article, we provide guidelines and methods that can support researchers of medical image processing and analysis in identifying very time consuming functions in their algorithms using profiling tools. The experimental findings show that the gathered profiling information can point out the main bottlenecks found in an algorithm implemented in C. This study also provides insights into why profiling data is useful; in particular, for optimizing a non-rigid image registration algorithm for real-time applications.

To the best of our knowledge, this is the first time that the chosen profiling tools are used to support the parallelization of a non-rigid image registration algorithm. Our findings are thus highly pertinent for the image processing and analysis area, mainly for the medical imaging community. Frequently, medical images in real clinical scenarios are of high resolution and need to be processed and analyzed fast. Additionally, computers with multi-cores are available in medical environments with enough computational power to handle tasks of image processing and analysis efficiently. Hence, the insights presented in this work are timely and demanded for researchers developing algorithms of medical image processing and analysis.

This article is organized as follows: Section 2 presents the background concepts, mainly the profiling method used to identify snippets with excessive CPU consumption. In the same section, methods that have been proposed to speedup the computation of non-rigid image registration algorithms are reviewed. In Section 3, the material and methods used to speedup the runtime of the studied algorithm of non-rigid image registration, including the profiling tools used for tasks such as measuring the performance of the algorithm, gathering the data to be analyzed, and building the visualization of the performance analysis, are described. The main findings and observations resulting from the performed experiences using profiling data to optimize the computation of the studied algorithm are discussed in Section 4. Section 5 provides the conclusion of this study and presents future work directions.

2 Background and Related Work

In this section, the topic of medical image registration and the profiling tools used in this study are introduced. Additionally, research concerning the use of high-performance computing techniques to speedup medical image registration algorithms is reviewed.

2.1 Medical image registration

Image registration is the process of aligning images of the same object obtained at different times or from different view-points, using different or similar imaging modalities or conditions [16, 25, 8]. This process aligns geometrically two images, usually referred as the *reference* and *sensed* images. In image registration applications, the involved information can be gathered through a combination of data sources as in image fusion, change detection, and multi-channel image restoration, to name a few [14, 26]. Focusing on non-rigid registration, one accounts for changes between the images that arise not only by global rotations, translations and scaling, but also due to complex local variations. Medical image registration is commonly used to follow up information on patient anatomy along different time points, where one must take into account the deformation of the anatomy itself due to, for example, the patient's breathing or normal anatomical changes [9, 14].

A significant number of image registration methods have been developed both to obtain the combination, i.e., fusion of data acquired by different clinically useful imaging modalities through mutual co-registration, or to register one image to other images to understand how patient anatomy has changed over time [14, 15]. In general, the majority of the rigid image registration methods comprise four steps: feature detection, feature matching, transform model estimation, and image re-sampling and transformation [25, 14]. Non-rigid registration methods, on the other hand, commonly search for the optimal transformation parameters that maximise a similarity measure. All these steps are well documented in literature [25, 14, 27, 15].

In order to attained the registration of two input images, the non-rigid registration should establish a correspondence measure between a reference image, I_r, and sensed image, I_s, using a parameter transformation $T_t(\cdot)$ of image geometry in line with a similarity function $\rho(\cdot)$. When I_s has a higher dimension than I_r, projection operators P_r and P_s can be used to reduce I_s dimensionality. Then, the non-rigid image registration problem can be expressed via maximizing the similarity measure function [26]:

$$T_t^*(\cdot) = \arg_{T_t(\cdot)} \max \rho(P_r(I_r), P_s(T_t(I_s))). \quad (1)$$

An FFD model comprises a powerful tool for deforming an image volume using cubic B-splines. This technique is applied, for example, in deformation analysis in brain images, by deforming an object by adjusting an underlying mesh of control points, creating its 3D shape, and a smooth and C^2 continuous transformation [12]. To define a spline based FFD, the domain of the image volume can be denoted as

$\Omega = \{(x, y, z)|0 \leq x < X, 0 \leq y < Y, 0 \leq z < Z\}$. On the other hand, let the parameters of the transformation and the amount of deformation, Φ, be expressed as a $n_x \times n_y \times n_z$ mesh of $\phi_{i,j,k}$ control points with a uniform spacing, δ. Thus, ϕ can be formed as a low resolution mesh for modeling global non-rigid deformations, and as a high resolution mesh for modeling local deformations of the control points mesh [11, 12] with high accuracy. Thus, one can write FFD as a 3D tensor product of 1D cubic B-splines expressed as:

$$T_{local^{(x,y,z)}} = \sum_{l=0}^{3}\sum_{m=0}^{3}\sum_{n=0}^{3} B_l(u)B_m(v)B_n(w)\phi_{i+l,j+m,k+n}, \tag{2}$$

where $i = \lfloor x/n_x \rfloor - 1$, $j = \lfloor y/n_y \rfloor - 1$, $k = \lfloor z/n_z \rfloor - 1$, $u = x/n_x - \lfloor x/n_x \rfloor$, $v = y/n_y - \lfloor y/n_y \rfloor$, and $w = z/n_z - \lfloor z/n_z \rfloor$, and B_l represents the l-th basis function of the B-spline [11, 12]:

$$\begin{aligned} B_0(u) &= (1-u)^3/6, \\ B_1(u) &= (3u^3 - 6u^2 + 4)/6, \\ B_2(u) &= (-3u^3 + 3u^2 + 3u + 1/6, \\ B_3(u) &= u^3/6. \end{aligned} \tag{3}$$

Considering $B_l(u) = 0$ for $l < 0$ and $l > 3$, the derivative terms are nonzero only in the neighborhood of a given point. Therefore, the optimization of the objective function can be efficiently achieved using gradient descent [11, 12]. However, FFD algorithm is computationally intensive; in particular, when dealing with images of large dimensions, which occurs frequently in several possible applications [14]. As an example, the parallel computation of the human brain deformation is a recent field of exploration which can be efficiently studied through processing large amounts of high resolution images concurrently [3]. Moreover, the conjugate gradient descent algorithm can optimize all control points and interpolate the complete image under study at each iteration [11]. However, the computation of the similarity measure and of the geometric transformation are computational bottlenecks of the non-rigid registration method, demanding, therefore, increased efforts for effective parallelization techniques for such computations.

2.2 Profiling methods

Here, the use of profiling methods for measuring the computation time of each function in an algorithm is described. Profiling is a well-known tool that evaluates the performance of an algorithm by gathering its runtime data. Its principal objective is to assist programmers in identifying performance bottlenecks of the algorithm. Thus, the technique is commonly used to get an insight into an algorithm's performance, and to assess the use of instruction sets in order to identify and evaluate portions of code that cause excessive processor utilization. It can further check several metrics such as memory allocation, memory usage and leaks, cache performance, execution time, or even energy consumption [22]. Different profiling

approaches exist such as instrumented, event based, statistical, and simulation based [20, 19, 18].

Usually, a performance analysis based on profiling consist of the following four steps: instrumentation or modification of the algorithm under study to generate performance data, measurement of significant aspects of execution that are essential for generating the needed data, analysis and visualization of the gathered data [24] (Fig. 1).

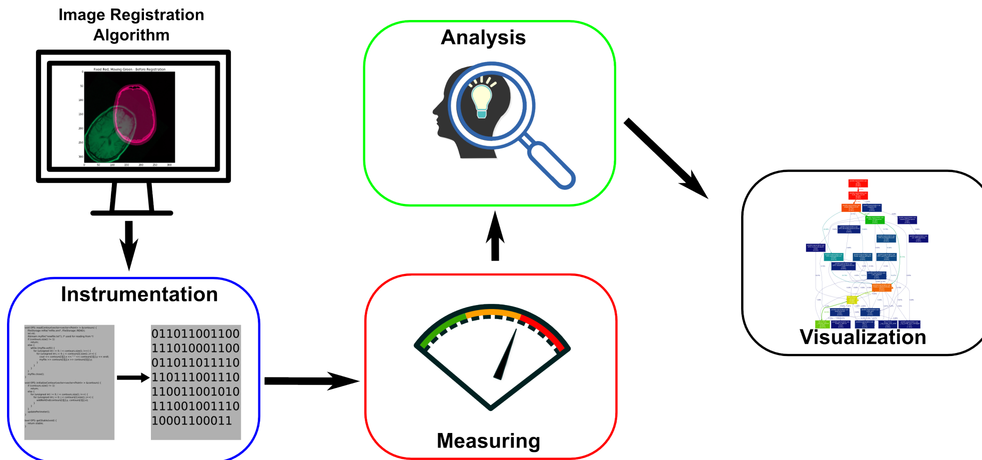


Fig. 1 Diagram of the profiling method.

2.2.1 Instrumentation

A compiler and the source code of the algorithm under study are needed for its instrumentation. The instrumentation process, at compile time, adds a detailed listing of the running statistics to the object file, and links the executable to standard libraries that have profiling information enabled. Next, the instrumentation incorporates measurement code into the implementation, resulting in an accurate assessment of running times [17, 28, 24]. Instrumentation processes are developed to determine possible options for modifying the behavior of the algorithm. Monitoring runtime behavior of algorithms involves aggregating information based on the number of executions of each basic block, and instrumenting binaries to trace various type of events such as `free` and `malloc` and similar function utilities.

2.2.2 Measuring

Gathering profile data is the second step in the profiling method. Typically, the following information is collected during the execution of the algorithm under study: approximate time spent in each function; the number of times a function is invoked; a list of the caller functions invoking a given function; a list of the

descendant functions that a given function invokes; and an estimate of the cumulative time spent in the descendant functions invoked by a given function [19]. Relevant data about functions can be collected by post-processing this information from one or more executions. This data is then stored in files for subsequent processing. Following this, a dynamic call graph for the execution is built [18, 23]. In general, gathering profiling data does not interfere with the execution of the algorithm [23, 24].

2.2.3 Data analysis

The third step of the profiling method concerns the analysis of the data gathered in the previous step. Here, related binary is produced and, subsequently, the output data is available for extraction. By convention, output files are named after the respective profiler, e.g., `perf.data` for the `perf` profiler, `gmon.out` for `gprof`. Each output file contains the execution profile. Profilers analyse the data and extract performance statistics as well as record the arc in the call graph that corresponds to the activation of each function [20, 19, 24].

Profiler determines the most costly functions and collects the arcs of the dynamic call graph traversed during the execution of the algorithm for such functions. This enables the visualization of the call graph and presentation of the measures collected from the execution. Such information can be graphically represented as returning address for a function call referred as *caller* being used to identify the source of the arc and the destination referred as *callee* [23].

2.2.4 Visualization

In the final step, collected profiling data is presented by incorporating the call graph of the algorithm under study. Visualization can employ *call stack walking* which is a technique that identifies calling relationships between functions in an implementation. Every call relationship that occurs is also represented in the graph with the CPU usage time for the respective call. Both tools, `gprof` and `perf`, provide dynamic call graph information for all instrumented code snippets. A call graph is binary, and sometimes is treated as a multi-graph, instead of as relations-relation over functions, or procedures as defined in an algorithm implementation [20, 19]. An edge (f, g) shows that function f invokes function g, and the nodes represent the individual functions in the executable.

2.3 Related work

Parallel computing has been applied to highly complex problems such as computations that involve large workloads and data, and intensive critical numerical analysis. Sequential algorithm implementations are frequently re-coded in order to decompose the algorithms or the data into smaller portions. These portions are commonly referred as tasks, and are distributed to be executed in many- or multi-cores simultaneously [29, 30]. Throughout this procedure, the tasks of

communication and coordination are performed based on memory usage by different computer processing units [30].

The growing popularity and use of multi-core processor architectures in medical imaging applications is well documented [14, 27, 15]. The primary objective of multi-core CPUs architectures has been to increase the performance of applications by exploiting parallelism. However, writing parallel implementations from scratch is a very complex and demanding task. Parallelizing legacy implementations written by someone else is even more challenging [14].

From several works [31, 32, 26], one can realize that in the field of medical image processing and analysis, it is not common for the developers to use tools that detect computationally costly functions in their algorithms. However, several studies have been conducted to address performance issues in image registration algorithms using high-performance computing [33, 13, 15]. For example, Shackleford et al. [14] presented a comprehensive survey of non-rigid registration algorithms that are suitable for use in modern multi-core architectures. Image registration tasks though computationally costly can yield to high parallelism. Therefore, multi-core architectures with high parallel processing power provide excellent opportunities for speeding up these tasks.

Computationally intensive Mutual Information (MI) based algorithms have been successfully employed in parallel architectures such as clusters [34], Graphic Processing Unit (GPU) [33, 27], multi-core Cell Broadband Engine Architecture (CBEA) [35], and Field-Programmable Gate Array (FPGA) [7], reducing their runtime and making them suitable for routine clinical use. For example, MI based algorithms have been used to correct the misalignment - an image registration application - of tissue in computed tomography (CT), positron emission tomography (PET) and magnetic resonance (MR) images, achieving accuracy comparable to one achieved by clinical experts.

Rohlfing and Maurer [3] and Christensen [34] exploited the use of shared-memory multiprocessor computer architectures as well as data and task partition parallel programming models. Rehman et al. [2] developed a parallel approach of non-rigid registration by addressing it as an Optimal Mass Transport problem. Lapeer et al. [36] presented a point based registration method, integrating a Radial Basis Function (RBF) as a smoothing function and sought to mimic the interacting deformation of biological tissues. Mafi and Sirouspour [37] exploited a GPU based computational platform for real-time analysis of soft object deformation.

Ellingwood et al. [16] developed a new computation- and memory-efficient Diffeomorphic Multi-Level B-Spline Transform Composite method on GPU for the non-rigid mass-preserving registration of CT volumetric images. The Sum of Squared Tissue Volume Difference (SSTVD) was adopted as the similarity criterion to preserve the computed tissue volume. A cubic B-Spline based FFD transformation model was used to capture the non-rigid deformation of objects like human lungs. The experiments used lung CT images, indicating an increase of speed of 112 times relative to the single-threaded CPU version, and of 11 times compared to the 12-threaded version when considering the average time per iteration using the GPU based implementation.

3 Material and Methods

As described in Section 2.1, non-rigid image registration involves transforming different sets of data into one coordinate system. Besides the optimization of an objective function, its evaluation as well as the transformation of the floating image using splines and an interpolation function can be taken into account to accelerate the FFD algorithm. In this study, the acceleration possibilities for the optimization step were identified by realizing parallelization options using profiling tools [17, 18, 24].

3.1 Environment settings

The used test infrastructure included a desktop computer, with 16 GB of RAM (DDR3-1600 MHz), an Intel(R) Core(TM) i7-4790 3.60 GHz processor, the Linux Debian 8 operating system, the GNU gcc/g++ compiler 4.9.2, the Open Multi-Processing (OpenMP) 3.1 application programming interface, Visual Studio Code 1.57.1, `gprof` 2.25, `perf` 3.16.7-ckt20, `gprof2dot` [1], and `dot` [2] 2.38. The used processor has four physical cores, and two logical threads can be simultaneously run in each core using the support of a feature commonly known as *hyper-threading technology*. Thus, one can choose to effectively run from a single thread to a maximum of 8 threads depending on partial or complete utilization of the available cores.

For the experiments, this study used Multiple Sclerosis (MS) images that were collected from the MS Longitudinal Challenge Data Set repository [38], and are freely distributed for research purposes. We randomly selected thirteen images from the original dataset to validate the non-rigid image registration results. All included images were obtained using the same imaging scan and under the same acquisition conditions, i.e., using a 3.0 Tesla MR imaging scanner (Philips Medical System, Best, The Netherlands), according to the following image acquisition parameters: T_1-weighted ($T_1 - w$) magnetization prepared rapid gradient echo (MPRAGE) with TR=10.3 ms, TE=6 ms, flip angle=8°, and 0.82x0.82x1.17 mm^3 voxel size; a double spin echo (DSE), which produces PD-w and $T_2 - w$ images with TR=4177 ms, TE$_1$=12.31 ms, TE$_2$=80 ms, and 0.82×0.82×2.2 mm^3 voxel size; and a $T_2 - w$ fluid-attenuated inversion recovery (FLAIR) with TI=835 ms, TE=68 ms, and 0.82x0.82x2.2 mm^3 voxel size [38].

[1] `gprof2dot` is an open source script written in Python used to convert the output from a range of profiles into a dot graph. This script can be freely downloaded at `https://github.com/jrfonseca/gprof2dot`.

[2] `dot` is a Graphviz feature for producing hierarchical drawings of directed graphs. Graphviz is an open source visualization software for representing structural information such as diagrams of abstract graphs. More information is available at `http://graphviz.org`.

3.2 Registration evaluation

Dice Similarity Coefficient (DSC) is a simple and useful statistical validation metric commonly used to evaluate the performance of both registration reproducibility and spatial overlap accuracy against to registration ground truths [11]. The DSC value rates the overlap of two masks between 0 (zero) and 1 (one), where 1 (one) indicates a perfect overlap and 0 (zero) none. Therefore, DSC assesses the spatial overlap between the registration result (M_m) and the corresponding registration ground truth (M_p) as [11]:

$$DSC = \frac{2\|M_m \bigcap M_p\|}{\|M_p\| + \|M_p\|}, \quad (4)$$

where $\|M_m\|$ and $\|M_p\|$ are the number of pixels, or voxels in 3D, in M_m and M_p, respectively. M_m is the area, or volume, of the registration obtained by the automated algorithm whereas M_p is the area, or volume, of the registration ground truth and $M_p \bigcap M_p$ is the overlapping area, or volume, of the two images under comparison.

3.3 Performance evaluation

In order to estimate the speedup of the studied FFD algorithm, Amdahl's law of speedup was used [39]:

$$Speedup_{enhanced} = \frac{1}{(1-f) + \frac{f}{S}}, \quad (5)$$

where $Speedup_{enhanced}$ is the overall speedup of the algorithm, f the execution time of a function eligible for optimization, and S the expected speedup of this function. The key idea of this formula is to determine time-consuming functions in an implementation that can be adapted for optimization. Such functions (complete or partial) are frequently referred as bottlenecks. To gain significant overall speedup, the value of f should be high [29, 40, 30]. Once the bottlenecks are identified, possible optimizations are postulated to help the performance improvement. These optimizations should then be individually verified to ensure that they result in real measurable improvements.

The performance of the studied FFD algorithm was, therefore, improved concerning the bottlenecks that were perviously identified by using the profiling tools `gprof` and `perf`. We selected these tools as they combined profiling methods based on distinct approaches, i.e., those based on instrumentation, statistics, and event based. Each tool comprises two essential components. One is a runtime routine; profiling tool calls this routine at the beginning of every function that needs to be compiled with profiling parameters. The other component is the post-processing version of the algorithm under analysis that aggregates and presents the data. We compiled the FFD single thread based algorithm implementation with the following parameters: (`-fno-omit-frame-pointer`) in order to enable the frame pointer analysis; (`-g`) for generating symbol information, which in turn enabled source code analysis; and the parameter `-pg`, which is used for inserting the monitor function `mcount` before each function call.

The profiling tool maintains a careful calculation of the effective computation times in different execution scenarios. The monitor function *mcount* records the function address and identifies the source of the cycles based on the addresses generated inside the profiled function. When a child function is a member of a cycle, the time shown is the appropriate fraction of the time for the complete cycle. Self-recursive routines have their calls broken down into calls from the outside and self-recursive calls; thus, only the outside calls affect the propagation time.

`gprof` is relatively easy to employ and is portable though limited in scope. It produces a detailed call graph identifying functions that call other functions and the number of times each function is invoked. Furthermore, `gprof` lists the percentage of time spent in a function and, hence, computes the execution time of that function. `perf` uses statistical sampling to collect profile data thereby generating an interruption at regular time intervals. `perf` can identify all processes running on the CPU enabling it to capture all relevant information such as the program counter, and the CPU core. Next, it writes all of this data to an output file called `perf.data`. `gprof` and `perf` runtime routines gather accurate call counts that combined with a post-processing version of the algorithm leads to an evaluation table that lists the number of calls to each function, as well as the percentage amount of time spent in such function, and the average time per call.

4 Results and Discussion

In this section, results from experiments performed aiming at getting useful profiling information, accumulating samples, and producing statistically meaningful results of the FFD algorithm under study using images of the MS Longitudinal Challenge dataset, are reported and discussed.

4.1 Algorithm evaluation

The implementation was profiled using 13 images, and then the accuracy was evaluated by comparing the registration results with those obtained using a *classical* FFD implementation [3]. The obtained comparative results are presented in Table 1.

4.2 Computation time evaluation

The required runtime was investigated in order to evaluate the impact of the profile based implementation in terms of computer performance. Each experiment was executed fifty times for each image. Mean and standard deviation values of the time required to process the profiled based algorithm were calculated. All input images were treated in the same manner, and the considered processing time includes the

[3] An executable version of the FFD algorithm used for comparison purpose, by performing quantitative analysis based on the DSC value, can be downloaded from Daniel Rueckert's webpage: `http://www.doc.ic.ac.uk/~dr`

Table 1 Comparison of *classical* and profiled based FFD algorithms: results for 13 images based on the DSC value.

Image #	Dimension	DSC
1	256x256x35	0.97262
2	256x256x120	0.95407
3	256x256x70	0.96194
4	256x256x70	0.96071
5	256x256x70	0.97167
6	256x256x120	0.93503
7	256x256x70	0.94767
8	256x256x70	0.95950
9	256x256x70	0.96650
10	256x256x120	0.97314
11	256x256x70	0.96029
12	256x256x70	0.95365
13	256x256x120	0.96493

Table 2 Means and standard deviations of the runtime (in seconds) required by the profiled based FFD algorithm implementation.

Image #	Dimension	Runtime
1	256x256x35	73.08411 ± 0.05945
2	256x256x120	79.00041 ± 0.07101
3	256x256x70	74.08051 ± 0.01961
4	256x256x70	73.48618 ± 0.01920
5	256x256x70	74.20270 ± 0.01393
6	256x256x120	79.00217 ± 0.07294
7	256x256x70	74.68039 ± 0.01279
8	256x256x70	74.99707 ± 0.01484
9	256x256x70	73.94009 ± 0.01752
10	256x256x120	79.07080 ± 0.07590
11	256x256x70	74.08260 ± 0.01220
12	256x256x70	74.83009 ± 0.01522
13	256x256x120	79.00239 ± 0.08677

391 time spent to load the data into the main system memory until the end of the
392 registration process, i.e., when the resultant image has been produced. The obtained
393 results are presented in Table 2.

394 4.3 Performance analysis

395 Concerning the performance analysis, it should be noted that the profiling tools
396 collect data while monitoring performance counters, hardware interruptions, and
397 operating system calls. Profiling tools periodically interrupt the kernel of the
398 operating system to record a new sample; these samples are stored in a ring buffer,
399 generating, therefore, overhead. `perf` mitigates sampling overhead by enforcing
400 local sampling buffer. `perf` creates one instance of the event on each used CPU;
401 then, the events are effectively measured when that CPU executes each thread. All
402 the samples are aggregated into a single output file once all profiles have been run.
403 In the experiments, the sampling mode in `perf` was used to trace the events of the

FFD algorithm in real-time. For the experiments conducted with 2, 4, and 8 threads, perf generated output files with sizes of dozens of megabytes. The obtained results are presented in Table 3. This considerable big data size is because perf depends on the adopted frequency. For the sampling rate according to the events are recorded, a rate of 4000 samples per second was used, which resulted in high overhead and large output files. However, *gprof* generates output files that are in the order of kilobytes, i.e., 768 KB; mainly, because the output file contains a histogram of program counter samples and the arc table.

Table 3 File sizes, indicated in megabytes (MB), generated by perf according to the dimension of the input images and the developed OpenMP based implementation using different number of threads.

Image #	Dimension	Number of threads			
		1 Thread	2 Threads	4 Threads	8 Threads
1	256x256x35	9.65	10.23	12.24	25.41
2	256x256x120	12.40	13.16	18.91	28.74
3	256x256x70	11.08	11.83	13.02	27.53
4	256x256x70	17.74	18.48	22.31	25.42
5	256x256x70	36.32	32.37	61.61	25.45
6	256x256x120	8.61	8.92	9.84	25.63
7	256x256x70	9.16	9.45	25.53	33.02
8	256x256x70	7.85	8.23	11.33	30.56
9	256x256x70	12.04	12.51	15.92	30.22
10	256x256x120	24.07	25.04	31.32	50.58
11	256x256x70	18.44	19.14	25.81	52.37
12	256x256x70	12.91	13.60	14.64	31.66
13	256x256x120	20.51	21.20	33.58	46.05

In order to extract performance statistics and record the arcs in the call graph, the collected data was analyzed. The call graph represents information intuitively by employing a visual map from a collection of hierarchical data in order to quickly facilitate its understanding [41, 42, 43]. The call graph represents time consuming functions and the number of times the functions were invoked. By analyzing the call graph sample of the image registration algorithm, the graph shown in Fig. 2 was generated, which includes the time propagated for each function from its descendants, and the number of times each function was called.

The built call graph displays the descendants as well as the caller of each function, including the time propagated to each routine from its descendants. The important entries of the call graph profile are the ones depicted with grey numbered circles in Fig. 2, which refers to the primary function of the studied non-rigid image registration algorithm. In this figure, *element 2* represents the name of the caller function; the percentage of the runtime accounted by the algorithm's function and its descendants is indicated by *element 3*; *element 4* accounts for a time that depends on whether it is the primary function for that section, the function's caller or descendant functions. In the first case, time is the actual function execution time during the running of the algorithm. In the second case, it indicates the amount of the self-time being propagated to that caller, based on the percentage of calls to the primary function made by that caller. Finally, for descendant functions, it represents the

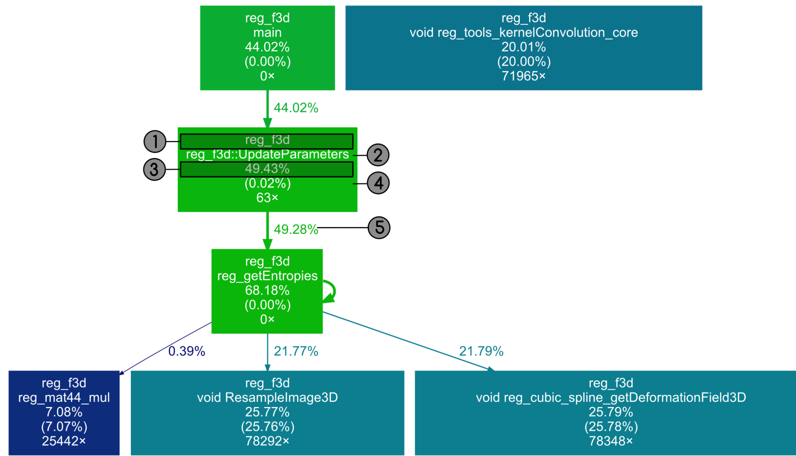


Fig. 2 Call graph generated by `perf` representing the most often called functions in the studied image registration algorithm.

amount of that descendant function's self-time being propagated to the primary function based on the percentage of calls made to the descendant by the primary function. The last entry in the green box next to *element 4* represents the number of times that function was called (63x in this case), and finally *element 5* is related to the accumulated percentage of time running a function, taking into account the propagation for each descendant function.

The built call graph is helpful in evaluating the algorithm's performance and identifying its bottlenecks. Taking full advantage of profiling tools requires to focus on the analysis of the relevant parts of the algorithm execution, making the experiments easier to understand. The used profiling tools identified that the function `reg_getEntropies` is responsible for 68% of the total running time (56.90 seconds), meaning that joint histogram filling is the main time consuming task within this function. The other costly functions identified by the profiling tools were:

- `reg_cubic_spline_getDeformationField3D` which generates the deformation field: a lattice of equally spaced control points is defined over the reference image using cubic B-splines;
- `ResampleImage3D` which computes the value $I_s(T(x))$ for every pixel x, or voxel in 3D, inside the reference image. In this case, the computational complexity is linearly dependent on the number of pixels, or voxels, in the reference image;
- `UpdateParameters` which assesses the quality of a registration using a cost function such as mutual information. In order to achieve the perfect registration between two images, the transformation parameters are iteratively optimized.

To obtain a measure of the runtime, the program that implements the algorithm was run a total of fifty times. The average of the time elapsed reported by each profiling tool was calculated. In all cases, the execution times for different runs of

the implementation were remarkably consistent. The most time consuming functions iterate a hundred times which makes them desirable parallelization targets. The algorithm under study requires large computations, iterating until convergence, aiming to ensure the best possible registration of the input images. Hence, the studied algorithm can greatly benefit from a high degree of parallelism.

For the performance analysis of the developed parallel implementation, a benchmark problem was defined, which is suitable to evaluate the performance in the setting for sequential execution as well as parallel execution. Next, the effect of using different number of physical cores on the performance of the multi-threaded algorithm was studied. For a fixed number of cores, the same number of threads per core for execution, i.e., one thread for each core, was used. The costly function `reg_getEntropies` was implemented using OpenMP.

All the experiments that were previously carried out were repeated. The results were then compared using varying degrees of parallelism, in particular using 1, 2, 4 and 8 threads. As shown in Fig. 3, the developed parallel OpenMP based implementation achieved a significant reduction in the runtime of the studied non-rigid image registration algorithm relatively to its single-thread implementation. This validates the claim that profiling tools could help programmers to quickly identify critical bottlenecks.

Fig. 4 depicts the performance gain of the parallel OpenMP based implementation of the non-rigid image registration algorithm. One can see that it scales almost exponentially, and that the parallel implementation achieved runtime approximately seven times faster than those of the single thread based implementation, taking into account the ratio of the sequential runtime to the parallel runtime. The parallel based implementation uses one thread for each core for the execution, which is created and managed by the multi-threading library.

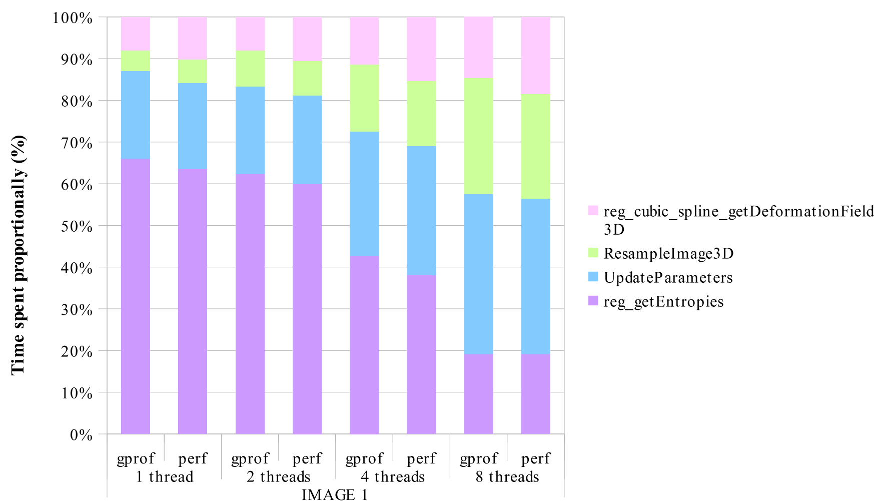


Fig. 3 Proportionality of the time consuming functions detected by the `perf` and `gprof` profiling tools using the developed OpenMP based implementation of the FFD algorithm.

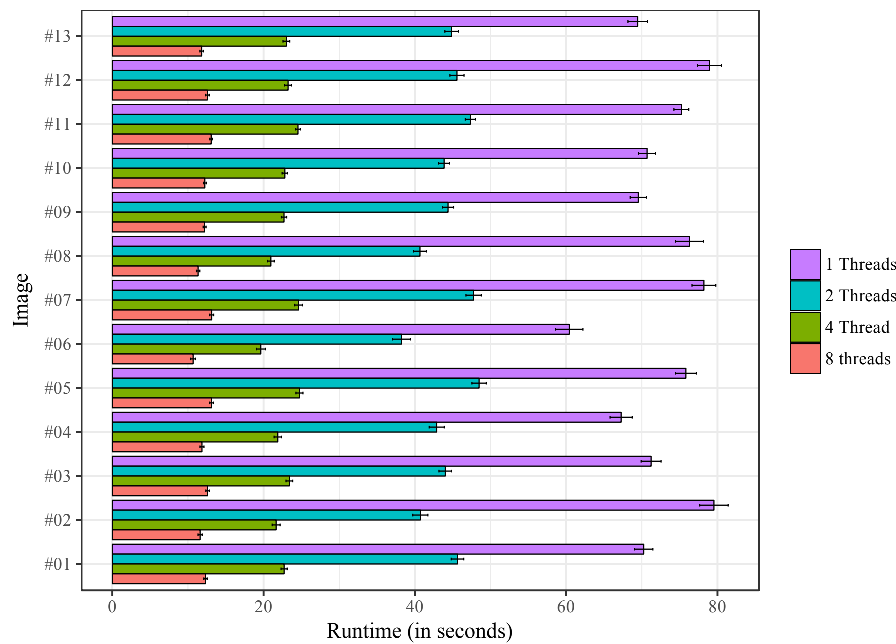


Fig. 4 Means and standard deviations of runtime spent for running the developed OpenMP based implementation of the FFD algorithm under study.

5 Conclusion and Future Work

The need for parallelization is on the rise, in part due to the facts that many computing devices now have multi-core processors available, and the applications are becoming more complex and have to deal with larger amounts of data. However, writing parallel code is a challenging task for many programmers, since it involves a strong learning curve for coding applications in parallel design, and requires a strong understanding of advanced concepts concerning memory hierarchy and optimal data paths in computer systems. While much effort has been devoted to address the issue of parallel programming, the current work was mainly focused on gathering parallelization support from profiling tools. As our findings suggest, profiling tools can be highly effective to detect and evaluate performance bottleneck snippets in a non-rigid image registration algorithm based on FFD, providing a low-impact method for gathering useful information.

The developed parallel OpenMP based implementation was compared against the corresponding single thread based implementation in several experiments. The parallelization of the costly functions of the FFD algorithm reduced the runtime by up to 7 times compared to the single thread version.

In conclusion, the proposed parallelization based on profiling tools substantially improved the runtime performance of the studied non-rigid image registration

algorithm. This will facilitate medical practitioners and researchers that commonly rely on image registration to label anatomical data, identify diseases, compare patient images and perform follow-up diagnosis. This is, therefore, a step forward to make accelerated non-rigid image registration solutions more accessible to a broader audience.

In future work, we will develop the approach further to address the all time consuming functions already detected in this study, with an objective to make these functions more efficient so that better speedups can be possible with more modern data parallel algorithms. We plan to optimize the performance of these algorithms by using heterogeneous parallel computing platforms that use GPUs. Additional challenges need to be addressed, for instance, the issue in shared memory systems of protecting simultaneous data access in order to avoid data inconsistency and errors, load balancing, and the efficient management of reading/writing data to massive data units. These challenging elements are all critical for achieving efficiency and the maximum performance possible in the underlying architecture. Finally, another interest future work would be the comparison of the suggested approach against related methods available in the literature.

6 Acknowledgments

The first author gratefully acknowledges the following institutions for the support received: Universidade do Estado de Mato Grosso (UNEMAT), in Brazil, and National Council for Scientific and Technological Development (Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq), process grant 234306/2014-9 under reference #2010/15691-0.

References

1. Parraguez SPP (2015) Fast and robust methods for non-rigid registration of medical images. PhD thesis, Imperial College of Science
2. Rehman T, Haber E, Pryor G, Melonakos J, Tannenbaum A (2009) 3D nonrigid registration via optimal mass transport on the GPU. Medical Image Analysis 13(6):931–940, DOI 10.1016/j.media.2008.10.008
3. Rohlfing T, Maurer CR (2003) Nonrigid image registration in shared-memory multiprocessor environments with application to brains, breasts, and bees. IEEE Transactions on Information Technology in Biomedicine 7(1):16–25, DOI 10.1109/TITB.2003.808506
4. Oliveira FP, Tavares JMR (2014) Medical image registration: a review. Computer Methods in Biomechanics and Biomedical Engineering 17(2):73–93, DOI 10.1080/10255842.2012.670855
5. Snape P, Pszczolkowski S, Zafeiriou S, Tzimiropoulos G, Ledig C, Rueckert D (2016) A robust similarity measure for volumetric image registration with outliers. Image and Vision Computing 52(C):97–113, DOI 10.1016/j.imavis.2016.05.006

6. El-Gamal FEZA, Elmogy M, Atwan A (2016) Current trends in medical image registration and fusion. Egyptian Informatics Journal 17(1):99 – 124, DOI 10.1016/j.eij.2015.09.002
7. Dandekar O, Shekhar R (2007) FPGA-accelerated deformable image registration for improved target-delineation during CT-guided interventions. IEEE Transactions on Biomedical Circuits and Systems 1(2):116–127, DOI 10.1109/TBCAS.2007.909023
8. Warfield SK, Jolesz FA, Kikinis R (1998) A high performance computing approach to the registration of medical imaging data. Parallel Computing 24:1345–1368, DOI 10.1016/S0167-8191(98)00061-1
9. McInerney T, Terzopoulos D (1996) Deformable models in medical image analysis: a survey. Medical Image Analysis 1(2):91 – 108, DOI 10.1016/S1361-8415(96)80007-7
10. Salomon M, Heitz F, Perrin GR, Armspach JP (2005) A massively parallel approach to deformable matching of 3D medical images via stochastic differential equations. Parallel Computing 31(1):45–71, DOI 10.1016/j.parco.2004.12.003
11. Modat M, Ridgway GR, Taylor ZA, Lehmann M, Barnes J, Hawkes DJ, Fox NC, Ourselin S (2010) Fast free-form deformation using graphics processing units. Computer Methods and Programs in Biomedicine 98(3):278 – 284, DOI 10.1016/j.cmpb.2009.09.002
12. Rueckert D, Sonoda LI, Hayes C, Hill DLG, Leach MO, Hawkes DJ (1999) Nonrigid registration using free-form deformations: application to breast MR images. IEEE Transactions on Medical Imaging 18(8):712–721, DOI 10.1109/42.796284
13. Palomar R, Gómez-Luna J, Cheikh FA, Olivares-Bueno J, Elle OJ (2017) High-performance computation of bézier surfaces on parallel and heterogeneous platforms. International Journal of Parallel Programming DOI 10.1007/s10766-017-0506-1
14. Shackleford J, Kandasamy N, Sharp G (2013) High Performance Deformable Image Registration Algorithms for Manycore Processors. Morgan Kaufmann Publishers Inc., DOI 10.1016/B978-0-12-407741-6.00007-4
15. Shams R, Sadeghi P, Kennedy RA, Hartley RI (2010) A survey of medical image registration on multicore and the GPU. IEEE Signal Processing Magazine 27(2):50–60, DOI 10.1109/MSP.2009.935387
16. Ellingwood ND, Yin Y, Smith M, Lin CL (2016) Efficient methods for implementation of multi-level nonrigid mass-preserving image registration on GPUs and multi-threaded CPUs. Computer Methods and Programs in Biomedicine 127:290 – 300, DOI 10.1016/j.cmpb.2015.12.018
17. Li Z, Atre R, Huda Z, Jannesari A, Wolf F (2016) Unveiling parallelization opportunities in sequential programs. Journal of Systems and Software 117:282 – 295, DOI 10.1016/j.jss.2016.03.045
18. Rul S, Vandierendonck H, Bosschere KD (2010) A profile-based tool for finding pipeline parallelism in sequential programs. Parallel Computing 36(9):531–551, DOI 10.1016/j.parco.2010.05.006

19. Graham SL, Kessler PB, McKusick MK (2004) gprof: A call graph execution profiler. ACM SIGPLAN Notes 39(4):49–57, DOI 10.1145/989393.989401
20. Dimakopoulou M, Eranian S, Koziris N, Bambos N (2016) Reliable and efficient performance monitoring in Linux. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Press, pp 1–13
21. Rohou E (2012) Tiptop: Hardware performance counters for the masses. In: 41st International Conference on Parallel Processing Workshops, pp 404–413, DOI 10.1109/ICPPW.2012.58
22. Ball T, Larus JR (1994) Optimally profiling and tracing programs. ACM Transactions on Programming Languages and Systems 16(4):1319–1360, DOI 10.1145/183432.183527
23. Schulz M, de Supinski BR (2007) Practical Differential Profiling, Springer, pp 97–106. DOI 10.1007/978-3-540-74466-5_12
24. Spivey JM (2004) Fast, accurate call graph profiling. Software: Practice and Experience 34(3):249–264, DOI 10.1002/spe.562
25. Li A, Kumar A, Ha Y, Corporaal H (2015) Correlation ratio based volume image registration on GPUs. Microprocessors and Microsystems 39(8):998 – 1011, DOI 10.1016/j.micpro.2015.04.002
26. Shi L, Liu W, Zhang H, Xie Y, Wang D, Shi L, Liu W, Zhang H, Xie Y, Wang D (2012) A survey of GPU-based medical image computing techniques. Quantitative Imaging in Medicine and Surgery 2(3)
27. Shams R, Sadeghi P, Kennedy R, Hartley R (2010) Parallel computation of mutual information on the GPU with application to real-time registration of 3D medical images. Computer Methods and Programs in Biomedicine 99(2):133 – 146, DOI 10.1016/j.cmpb.2009.11.004
28. Mittal S, Vetter JS (2015) A survey of CPU-GPU heterogeneous computing techniques. ACM Computing Surveys 47(4):69:1–69:35, DOI 10.1145/2788396
29. Gebali F (2011) Algorithms and Parallel Computing. John Wiley & Sons, DOI 10.1002/9780470932025
30. Vadja A (2011) Programming Many-Core Chips. Springer, DOI 10.1007/978-1-4419-9739-5
31. Eklund A, Dufort P, Forsberg D, LaConte SM (2013) Medical image processing on the GPU - past, present and future. Medical Image Analysis 17(8):1073–1094, DOI 10.1016/j.media.2013.05.008
32. Gong L, Kulikowski CA (2012) High-performance medical imaging informatics. Methods of Information in Medicine 51(3):258–259
33. Meng L (2014) Acceleration method of 3D medical images registration based on compute unified device architecture. Bio-Medical Materials and Engineering 24(1):1109–1116, DOI 10.3233/BME-130910
34. Christensen GE (1998) MIMD vs. SIMD parallel processing: A case study in 3D medical image registration. Parallel Computing 24:1369–1383, DOI 10.1016/S0167-8191(98)00062-3
35. Rohrer J, Gong L (2009) Accelerating 3D nonrigid registration using the cell broadband engine processor. IBM Journal of Research and Development 53(5), DOI 10.1147/JRD.2009.5429078

36. Lapeer RJ, Shah SK, Rowland RS (2010) An optimised radial basis function algorithm for fast non-rigid registration of medical images. Computers in Biology and Medicine 40(1):1–7, DOI 10.1016/j.compbiomed.2009.10.002
37. Mafi R, Sirouspour S (2014) GPU-based acceleration of computations in nonlinear finite element deformation analysis. International Journal for Numerical Methods in Biomedical Engineering 30(3):365–381, DOI 10.1002/cnm.2607
38. Carass A, Roy S, Jog A, Cuzzocreo JL, Magrath E, Gherman A, Button J, et al. (2017) Longitudinal multiple sclerosis lesion segmentation: Resource and challenge. NeuroImage 148:77–102, DOI 10.1016/j.neuroimage.2016.12.064
39. Hill MD, Marty MR (2008) Amdahl's law in the multicore era. Computer 41(7):33–38, DOI 10.1109/MC.2008.209
40. Kirk D, Hwu WM (2010) Programming Massively Parallel Processors: A Hands-on Approach. Elsevier
41. Bezemer CP, Pouwelse J, Gregg B (2015) Understanding software performance regressions using differential flame graphs. In: 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp 535–539, DOI 10.1109/SANER.2015.7081872
42. Gregg B (2016) The flame graph: This visualization of software execution is a new necessity for performance profiling and debugging. ACM Queue Magazine 14(2):91–110, DOI https://doi.org/10.1145/2927299.2927301
43. Kruskal JB, Landwehr JM (1983) Icicle plots: Better displays for hierarchical clustering. The American Statistician 37(2):162–168, DOI 10.2307/2685881