Parallel Monte Carlo Search for Hough Transform

Raul H. C. Lopes[1], Virginia N. L. Franqueira[2], Ivan D. Reid[1] and Peter R. Hobson[1]

[1]College of Engineering, Design and Physical Sciences, Brunel University London, Uxbridge, UB8 3PH, UK

[2]College of Engineering and Technology, University of Derby, Derby, DE22 1GB, UK

E-mail: [1]raul.lopes@brunel.ac.uk, ivan.reid@brunel.ac.uk, peter.hobson@brunel.ac.uk, [2]v.franqueira@derby.ac.uk

Abstract.

We investigate the problem of line detection in digital image processing and in special how state of the art algorithms behave in the presence of noise and whether CPU efficiency can be improved by the combination of a Monte Carlo Tree Search, hierarchical space decomposition, and parallel computing.

The starting point of the investigation is the method introduced in 1962 by Paul Hough for detecting lines in binary images. Extended in the 1970s to the detection of space forms, what came to be known as Hough Transform (HT) has been proposed, for example, in the context of track fitting in the LHC ATLAS and CMS projects. The Hough Transform transfers the problem of line detection, for example, into one of optimization of the peak in a vote counting process for cells which contain the possible points of candidate lines. The detection algorithm can be computationally expensive both in the demands made upon the processor and on memory. Additionally, it can have a reduced effectiveness in detection in the presence of noise.

Our first contribution consists in an evaluation of the use of a variation of the Radon Transform as a form of improving theeffectiveness of line detection in the presence of noise. Then, parallel algorithms for variations of the Hough Transform and the Radon Transform for line detection are introduced. An algorithm for Parallel Monte Carlo Search applied to line detection is also introduced. Their algorithmic complexities are discussed. Finally, implementations on multi-GPU and multicore architectures are discussed.

1. Introduction

We consider in this paper the problem of detecting straight lines in computer image processing, a recurring problem in areas as diverse as lane detection in vehicle vision-systems [1] or the use of the Hough Transform in muon tracks detection in the Large Hadron Collider [2, 3, 4].

We assume that a set of pixel coordinates is given as input. We do not deal with how the pixels are obtained. We also assume that, due to errors in the processes of data collection and transformation to digital images, pixels from the original lines may be missing, and noise and small deviations may be present.

A trivial brute-force algorithm can easily be devised that would test co-linearity of points with all lines defined by two pairs of input pixels. Such algorithm could be trivially parallelized, but the work demanded for N^2 pixels would be limited by an inferior bound of $\Omega(N^3)$, with each pixel being tested for co-linearity with all other ones.

As a more work efficient alternative, we consider the Hough Transform (**HT**), and its most used variation proposed by Duda and Hart. The foundations of algorithms and their complexity are discussed. Also discussed is their effectiveness in the presence of noise. The Radon Transform, that actually predates them, is introduced and discussed as an alternative to use in the presence of images with noise. For variations of those transforms possible sequential algorithms are discussed with their respective work complexity. It is shown that variations of both **HT** and **RT** can be demanding on both CPU and memory. As a result, it may be necessary to consider the possibility of reducing execution time by using parallel and probabilistic algorithms.

The main contributions of this work consist in introducing parallel algorithms for both the Hough transform and a discretized version of the Radon transform. Important issues in relation to the parallel algorithms introduced are: work complexity, scalability in the presence of parallel processing and variations of computer architectures, correctness of computation in the presence of concurrency in access to data. A performance evaluation of the parallel algorithms introduced is discussed.

2. The Hough Transform

The Hough transform (**HT**) was introduced as a patent for an analogue device that was, according to Rosenfeld [5], the first to describe the idea of mapping the pixels of a figure in a plane $X - Y$ to a parameter plane of intercept-slope, as depicted in figure 1. Rosenfeld [5] is credited by Hart [6] as the first to introduce an algorithm to compute the **HT**. Rosenfeld used the fundamental idea of the **HT** that collinear points in the initial input, when mapped to an intercept-slope, parameter space become concurrent lines.

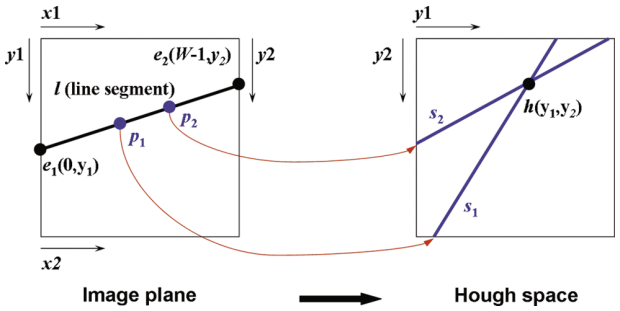


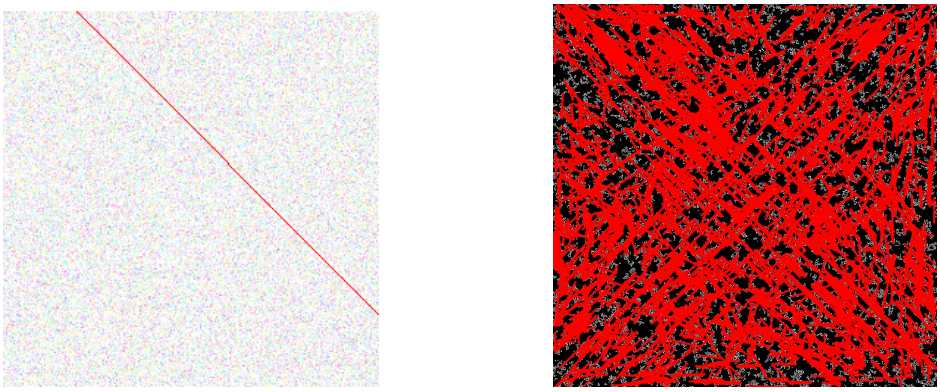
Figure 1: Hough transform map of collinear points to intersection of lines

One problem in this parameterization is that if two points in the input define a line that is almost parallel to the x-axis, their projections will meet in the infinite, and that leads to an unbound search. Rosenfeld proposed to address this by scanning the input twice at right angles. A second problem is that the input of the algorithm consists only of pixels, and any pair of them determines a line. Rosenfeld suggested representing the transform as an array of counters. Peak detection or a threshold can be used to select the most promising candidate lines.

Motivated by the idea of using a parameterization of the line that is invariant to translation and rotation of geometric figures, Hart [6] started using the normal parameterization of the line, algebraically defined by equation 1 below:

$$y = -\cos\theta/\sin\theta + \rho/\sin\theta \tag{1}$$

Duda and Hart [7] introduced the map from (x, y)-space to the (ρ, θ)-space. The Duda-Hart-Hough method, now consists of computing, for a point (x_i, y_i) and a possible angle θ,



(a) Image with Gaussian noise

(b) Image processed by **HT** in **OpenCV**

Figure 2: **DHHT** processing of image with noise

the respective ρ, that satisfies equation 1. Again, the computational work is unbounded if all possible values of θ are considered. Borrowing from Rosenfeld, the transform is proposed to be mapped into an array of counters indexed by discrete values for both θ and ρ. The work of the algorithm now consists in constructing a histogram and counting, for each pair (ρ, θ), the number of points that solve equation 1, which corresponds to the number of intersections of sinusoidals in (ρ, θ)-space, and number of collinear points in (x, y)-space.

The elements of an algorithm based one the *Duda-Hart-Hough Transform* (**DHHT**) described in [7] are thus:

- $\rho \in \mathbb{R}$ and $\theta \in [0, \pi)$;
- $\hat{\rho}$, discretized ρ, in the range of the size of the image;
- $\hat{\theta}$, the discretized number of intervals of θ;
- an array of counters, indexed by $(\hat{\rho}, \hat{\theta})$, each position counting the number of points mapped by equation 1 to the respective interval.

In this section and the rest of the paper, it is assumed than an image is a grid of pixels and N denotes its number of rows (or columns), N_p denotes the number of non-blank points, points that are part of valid lines or noise present in the image, and M represents the number of discrete values of θ.

The algorithm for the **DHHT** as described above has then a complexity that depends on the resolution of θ and is bound by $\Omega(MN_p)$, given that all possibly valid points must be considered at all possible values of θ. It is important to notice that N_p is limited from above by the dimensions of the image, which is quadratic in N. In the extreme, the algorithm can exhibit a behaviour that approaches the cube of N, in the presence of images with a lot of noise and with a number rows approximating the number of columns. This is exemplified by the figure 2a, where the grid has 1024 rows and columns, and the presence of Gaussian noise would demand around the cube of the number of rows in floating points operations. In a sequential implementation, this is possibly too much if many images are to be processed in a short time.

The idea of *Hough transform algorithm* has come to be mostly associated with some variation of the transform as described by Duda and Hart [7], which will be called in the future Duda-Hart-Hough algorithm. It is implemented, for example, in the open source package **OpenCV** [8]. Its limitations become evident both in processing time and effectiveness when applied to images with noise. Figure 2a shows one line additive Gaussian at amplitude 37, as generated by the library **CImg**. In figure 2b, the result of applying the **OpenCV** implementation of a

Hough transform algorithm to figure 2a is shown: close to three thousand lines are detected by **OpenCV**, which, of course, were not present in the input figure.

3. The Radon Transform

Hart's idea of using the normal plane for a mapping from the (x, y)-space was actually first used in the Radon Transform (**RT**). Brady [9], and, independently, Götz and Druckmüller [10], introduced the Discrete Radon Transform (**DRT**) that, at a cost of extra time complexity, has an inverse, fundamental in image reconstruction, and is well-conditioned in the presence of noise.

The **RT** takes the projection of the images at a set of angles, which maps the image into the Radon space. The discretization of the Radon Transform computes the projection as a summation of intensities along a set of angles. It takes into account the quantization of the lines normal parametrization, and, most importantly, that lines are approximated by pixels, a pixel being just a point in a grid. The integration of the original **RT** is then approximated by a summation of intensities of points lying in one unit wide strips of pixels. The range of angles values to be used will then be in $\Theta(M)$, but M here is the side of the smallest square grid containing the image.

The accumulation of intensities adds, for each angle θ in the range M, the intensity at pixel (x, y) to an accumulator $R(d, \theta)$ where

$$d = \lfloor x \cos\theta + y \sin\theta + 1/2 \rfloor \tag{2}$$

This clearly leads to an algorithm that is cubic in the number of grid rows: the computation being a nest of three loops ranging on angle values , rows and columns. The work complexity is a tight $\Theta(N^3)$, given that M is equal to N.

It leads inevitably to a work-intensive computation, but Brady [9] shows that the **DRT** defines a transform that has a well defined inverse and is robust in the presence of noise. The plot in figure 3 shows the sum of intensities computed for figure 2a. The projection with highest intensity clearly identifies the only line in figure 2a with the only peak in the plot.

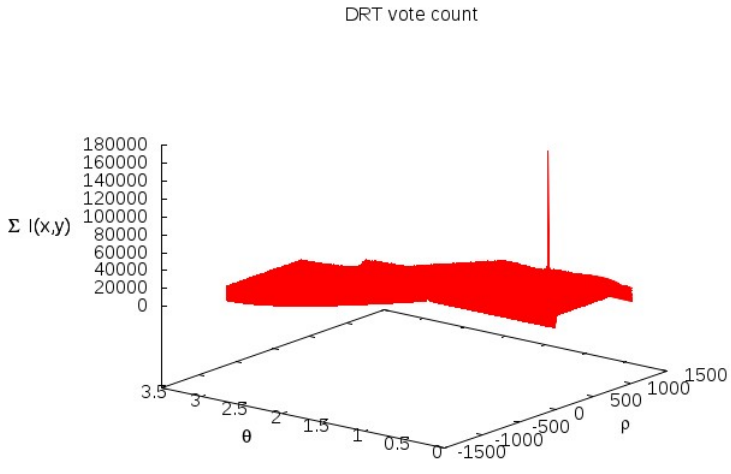


Figure 3: DRT plot showing strip of highest intensity

The **DRT** solves the limitations of the Hough transform algorithms in the presence of noise, but that at the expense of CPU resources.

4. Parallel DHHT algorithm

Parallel realizations of the **DHHT** will assume its limitations in dealing with noise and the impossibility of having an inverse transform, and try to gain in execution time, preferably in a scalable way. It is important to observe that parallel techinques applicable to the **DHHT** should be almost certainly be suitable to a **DRT** algorithm. A parallel **DHHT** algorithm (or a **DRT** one) will have to deal essentially with three steps:

- In parallel, map the application of equation 1 to all pairs of possibles values of $\hat{\theta}$ and all given points. This is a step that can be trivially parallelized to make use of as many processors as the architecture makes available and run in $O(MN/P)$, where P denotes the number of processors.
- Perform in parallel the increments of the array of counters. This step is non-trivial and has the potential to reduce the algorithm to a serialization given that for the same θ_i and two different points, one sole ρ_k will be produced and the same array counter will be incremented simultaneously with a possible loss of data.
- Select the pairs of (ρ, θ) that represent lines. This step can be trivially made parallel if a threshold approach is used as in the case of the **OpenCV** library [8]: in parallel select all pairs (ρ, θ) whose array counters lie above some given threshold. The work complexity will be limited by size of the image divided by the number of processors, giving $O(\frac{N^2}{P})$.

An obvious way to parallelize the second step above consists in completely banning interference: parallelize only over the range of θ values and all (ρ, θ) counters to be incremented will be different. This and the other two parallel steps define a first and simple parallel **DHHT**. That, however, doesn't scale with the hardware. The **OpenCV** library, for example, uses a range of 180 values of θ. It would mean doing 180 parallel steps: each would perform a loop over input points to perform in parallel 180 counter additions. A modern GPU can do at least 2000 additions in parallel, and the latest Intel AVX512 instructions perform 16 operations per core, with a Xeon Phi showing 72 cores capable of running 288 parallel threads.

A second algorithm dealing with non-deterministic interference might do:

- In parallel for a set of pairs (θ_i, p_j), where θ_i is an angle and p_j is some input point, and compute the respective ρ using equation 1. This leaves open the possibility of scheduling in parallel all possible pairs of angles and points, or a subset of them enough to keep all processors busy.
- In parallel, for each computed ρ mark a *compare* area indexed by (θ_i, ρ) with (θ_i, p_j).
- There will be concurrent writes in the compare area, but only one process will win, that process will increment the real (ρ, θ) counter. Other pairs are postponed for a next round.

That algorithm, here identified as **parht**, does the equivalent of a simpler *compare-and-swap* without using any locks. The upper bound of work is that of a sequential increment.

The third parallel variation of the **DHHT**, would realize the counter increment step in parallel by structuring it as follows:

- In parallel, mark 1 in an auxiliary counter indexed by (θ, ρ, p), where $p = (x, y)$ is the point used in the application of equation 1.
- In parallel, apply a segmented reduction indexed by (ρ, θ) to the auxiliary counters of the previous step.
- In parallel, add the resulting counter, one per value of (ρ, θ), to the (ρ, θ) accumulator.

The first and third sub-steps can be computed with parallel work of $O(N)$ each. The third step is computable in $O(N \lg N)$ work, giving a time bound in $O(\frac{N \lg N}{P})$, where P is the number of parallel processing units available.

It is important to notice that, again, in the first sub-step, the number of combinations of θ angles and points scheduled in parallel is left open. That would depend on how much memory is available for the auxiliary counter on the parallel device. On a $4GB$ RAM GPU, and with N equal to 4096 (pixels per line), all possible pairs of 1024 points and 256 values of θ could be processed in parallel.

5. Parallel Radon Transform

The DRT has been implemented on a mesh of N^2 processors, where N is the number of lines in the mesh and the grid [11]. Such a mesh can, theoretically be simulated in a GPU, at the expense of N GPU steps for each mesh parallel step.

A more realistic approach, identified here as **pardrt**, would consist in running the external loop in one parallel step over the range of angles being used, which we have previously describe as the first and cleanest parallel algorithm for the **DHHT**.

It can be presented as two nested loops, consisting of:

- In parallel for each possible angle do N^2 sequential steps.
- Each of N^2 sequential steps would update the intensity accumulator $R(d, \theta)$, base on equation 2, as described in section 3.

The algorithm shows work complexity of $O(N^3)$, and it could run in $O(N^2)$ time if enough processors are available, making it *naïvely* scalable. In the presence of multiple GPUs or many-core CPUs, this still presents a limitation in scalability: images will possibly be limited to 4 thousand lines, while the number of available CPUs may double that.

6. Monte Carlo Search Hough Transform

Any Hough transform algorithm will be limited by the fact that in principle the number of pixels to be considered may limit any algorithm to work in $\Omega(N^2)$, even when one assumes that grid of points is already in memory. Randomized versions of the **HT** go back to Kyriati et al in [12], which showed experimentally that it is possible to accelerate shape detection and obtain results identical to the standard **HT**, by using a fraction of the given points. Their algorithm is a classical case of a Monte Carlo algorithm, where some randomized procedure defines a sample of the input points to use and a stopping criterion can lead the algorithm to finish the computation before it reaches any good result. The process of course can be repeatedly run to produce a Las Vegas algorithm [13] that eventually correctly detects the same shapes as a complete Hough transform algorithm.

The features of the parallel **MCS** algorithm, here identified as **parmcsdrt**, can be summarized as composed of the following elements:

- Selection: a sample of pixels is taken from the given input.
- Expansion: In parallel, for each sampled pixel and each value of θ, (ρ, θ) is computed.
- Simulation: In parallel, counters are incremented.
- Backpropagation: a pair (ρ, θ) with a peak counter is prioritized for future tests with new samples.
- Parallel tree growth: as many branches of the tree can be grown in parallel as there are cores or SIMD parallelism available.

7. Implementation evaluation

Several parallel variations of the **HT** and the **DRT** were implemented. This section describes the evaluation of the implementation of the parallel algorithms identified in the previous sections as **parht**, **pardrt** and **parmcsdrt**.

The implementations were written in C++, using libraries of basic parallel operations for multi-core platforms and GPUs, as identified below:

- Implemented in C++, using the library *Threading Building Block* (**TBB**) for multi-core execution.
- Implemented using the library **Thrust**, for GPU execution.

7.1. Line detection in the presence of noise

Tests were perform to study the efficiency of the algorithms in line detection and the scalability and speed in the presence of parallel architectures. Table 1 shows a set of 6 different images, all produced as *jpeg* files by the library **CImg**. The column *pixels* indicates the size of the image while the column *points* shows the number of non-blank points. All images contain exactly one line and with Gaussian noise at amplitude 5 or 37, as indicated in the respective column. Tests were performed using the **OpenCV** implementation of the **HT**, and the **pardrt** implementation. All tests performed with **pardrt** successfully identified the line from the noise as indicated by a unique peak counter as shown, for example in figure 3, of the images indicated as **N37.1024**. The column **HT** uses *'-'* to indicate cases where both the **OpenCV** and the **parht** implementations failed to clearly indicate a small set of candidate lines, as in the case of figure 2b, which shows the close to three thousands detected by **OpenCV** (or **parht**) for the 1 line figure *N37.1024*.

Table 1: Images used and lines detected

Image	Pixels	Noise	**HT**	**pardrt**
N5.4096	2^{24}	5	yes	yes
N5.2048	2^{22}	5	yes	yes
N5.1024	2^{20}	5	yes	yes
N37.4096	2^{24}	37	-	yes
N37.2048	2^{22}	37	-	yes
N37.1024	2^{20}	37	-	yes

7.2. Performance scalability

Performance scalability tests were performed for parallel implementations of the **HT** (**parht**) and **DRT** (**pardrt**) algorithms. In each case, three different parallel C++ codes were developed, but different technologies were used to target the diverse architectures: parallel **OpenMP** offered by the **GCC** compiler; parallel multi-core using the Intel **TBB** library, also available in **GCC**; and GPU architecture, using the library **Thrust**.

The table 2 shows execution times for images shown in table 1. Times were obtained for an **OpenMP** implementation, compiled with **GCC** (-fopenmp $-O3$), running on GNU Linux Centos 7, Intel Xeon E5-2680 v3 (2.50GHz). The *Dual K20* shows time obtained on dual GPU K20 system, running Ubuntu 15.10 and Nvidia toolkit 7.5, on an older single socket Intel E-5430. Noticeable points about the numbers are: the best speed-up is obtained at at 5.04 on 6 cores, (efficiency 84%); drops in efficiency are shown from 6 to 12 cores, due to cache/hit miss, and even more at 24 cores; drop from 12 to 24 cores: cache hit/miss and non locality of data across physical processors; huge drop in performance for 2^{24} is again due to hit/miss ratio in processor's caches; the performance of the multi-gpu code is significantly affected by the PCI bus of the older machine hosting

An implementation of the algorithm described as **parmcsdrt** was tested on the same multi-core architecture as above. The implementation used samples of 1024 points, in a sequence

Table 2: Performance of **pardrt OpenMP** implementation

Image	1 core	6 cores	12 cores	24 cores	Dual K20
N37.4096	$1144s$	$272.3s$	$154.6s$	$83.92s$	$33.6s$
N37.2048	$87.31s$	$24.07s$	$14.92s$	$8.181s$	$3.0s$
N37.1024	$7.762s$	$1.548s$	$0.843s$	$0.46s$	$0.18s$

of rounds until a peak was detected. Tests of the algorithm for the images in table 2 show a reduction of a factor of 7 to 8 in the time demanded to run the tests with the same lines detected. For example, the image identified as *N37.4096* demanded 18.15 seconds to process on 6 cores.

8. Conclusion

The paper discusses variations of the **HT** and the **DRT**. Experimental evidence of an important weakness of the **HT** is shown. The theoretical computational bounds of both transform are shown, which make clear their limitations in the presence of big images and noise. Parallel algorithms are introduced and the possibility of using parallel Monte Carlo search to accelerate the **HT** is discussed, and an algorithm presented.

An experimental evaluation of a parallel algorithm for the **DRT** is presented which shows a good speedup for up to 12 cores. More extensive tests must be performed to investigate the relatively weak performance on mulli-GPU architectures and the parallel Monte Carlo search.

9. Acknowledgments

Lopes, Reid and Hobson are members of the GridPP collaboration and wish to acknowledge funding from the Science and Technology Facilities Council, UK.

References

[1] Mineta K, Unoura K and Ikeda T 2000 *Honda R&D Technical Review* **12** 101–108
[2] Amram N 2008 *Hough Transform Track Reconstruction in the Cathode Strip Chambers in ATLAS* Ph.D. thesis Tel Aviv University CERN-Thesis-2008-062
[3] Filho L M D A and Seixas J 2016 *XII Advanced Computing and Analysis Techniques in Physics Research* URL `https://indico.cern.ch/event/34666/contributions/813547/attachments/683827/939317/skeleton.pdf`
[4] Amstutz C *et al.* 2016 *20th IEEE Real Time Conference, Padova, Italy* URL `http://bura.brunel.ac.uk/bitstream/2438/13222/1/Fulltext.pdf`
[5] Rosenfeld A 1969 *Picture Processing by Computer* (Academic)
[6] Hart P E 2009 *IEEE Signal Processing Magazine* **26** 18–22
[7] Duda R and Hart P 1971 Use of the hough transformation to detect lines and curves in pictures Tech. Rep. 36 AI Center, SRI International 333 Ravenswood Ave, Menlo Park, CA 94025 SRI Project 8259 Comm. ACM, Vol 15, No. 1
[8] Bradski G 2000 *Dr. Dobb's Journal of Software Tools* **25** 120–126
[9] Brady M L 1998 *SIAM Journal of Computing* **27** 107–119
[10] W A Götz and H J DruckMüller 1996 *Pattern Recognition* **29** 711–718
[11] Cypher R E and ans L Snyder J L C S 1990 *SIAM Journal of Computing* **19** 805–820
[12] Yla-jaaski A and Kiryati N 1994 *IEEE Transactions on Pattern Analysis and Machine Intelligence* **16** 911–915
[13] Motwani R and Raghavan P 1995 *Randomized Algorithms* (Cambridge University Pess)
[14] Bergen J R and Shvaytser H 1991 *Journal of Algorithms* **12** 639–656
[15] Browne C, Powley E, Whitehouse D, Lucas S, Cowling P I, Rohlfshagen P, Tavner S, Perez D, Samothrakis S and Colton S 2012 *IEEE Transactions on Computational Intelligence and AI Games* **4** 1–49