Image Processing Without Sacrificing the Functional Paradigm

Antoine Bossard *
Kanagawa University, Yokohama, Kanagawa 221-8686, Japan

Abstract

Functional programming can be called modern programming in that it enables robust development and favours the programmer over hardware and performance considerations. Some characteristic features of the functional paradigm, such as map, have indeed been introduced into languages of other programming paradigms, like C++ and JavaScript. In spite of this deserved success, some challenges remain, such as input-output (I/O) operations, which often involve compromises with respect to the functional model. For instance, heavy memory I/O applications may keep prospective users afar. In this paper, we give a constructive proof of the practicability of the functional paradigm for such a scenario, by concretely considering image processing with the functional programming language Racket (a Lisp dialect). Both theoretical and experimental quantitative evaluations are conducted to show the performance of the implemented algorithms. Furthermore, in an attempt at establishing the capabilities and versatility of functional programming, this work also covers parallel processing, on both a single core and multiple cores.

Key Words: graphics; dithering; parallel; programming; software; Racket.

1 Introduction

The importance of the functional paradigm and functional programming need not be proven any more [6]. Functional constructs have even penetrated other paradigms, such as the object-oriented one with, for example, JavaScript's Array.prototype.map method [13] and C++'s std::apply function [7]. This is also the case of anonymous functions [14]. Moreover, it has been shown that functional programming is positive for Internet of Things (IoT) applications [5]. Considering concrete implementations of the functional paradigm, Lisp dialects can be found in robotics [11] and microcontroller applications [8]. It is thus no wonder that the functional paradigm is increasingly popular [2].

Despite all the advantages of the functional paradigm, users are likely to rapidly face some challenges, not to say limitations, inherent to this programming model. They mostly concern input-output (I/O) operations as those generally harm referential transparency [10].

To address some of these issues, several solutions have been proposed. For instance, Haskell, another functional programming language, relies on monads [9]. In addition, the SOF programming paradigm has been described to address the state tracking issue of pure functional programs, and especially those featuring lazy evaluation [1]. The Racket programming language is said multi-paradigm in that it allows imperative programming, albeit clearly stating that it is a best practice to limit as much as possible such non-functional constructs [3]. High-performance image processing with a functional programming style approach was proposed by Se´rot *et al.* [12], but it requires a specific, complex hardware architecture.

Our objective in this paper is to give a constructive proof that shows the functional programming paradigm remains practicable even when conducting image processing, that is, memory operations on rather large data. In other words, that such heavy I/O operations are not an excuse to sweep the functional paradigm aside [4].

To this end, we have selected the Racket language, based on Scheme and thus a Lisp dialect, as it tolerates imperative programming when needed, as previously recalled [3], which drastically improves usability over pure functional languages like Haskell.

The rest of this paper is organised as follows. First, preliminaries for this work are presented in Section 2. Then, a first image processing algorithm, dynamic palette calculation, is discussed in Section 3. Next, this algorithm is reused to describe in Section 4 a dithering algorithm. Dithering is further discussed from the parallel processing point of view in Section 5. Finally, concluding remarks are made in Section 6.

2 Preliminaries

About the notations used hereinafter, for the sake of brevity but without introducing any ambiguity, set operations such as $\setminus$ (exclusion) and $|\ldots|$ (cardinality) are sometimes applied to lists (i.e. sequences).

Regarding image file loading and pixel access, Racket provides convenient functions. First, read-bitmap takes a path to an image file (the GIF, JPEG, PNG, BMP and a few other formats are supported) and instantiates and returns the corresponding bitmap% object. Second, the pixel values, that is colours, that make the bitmap can be retrieved with the get-argb-pixels method of the bitmap%

*Graduate School of Science. Email: abossard@kanagawa-u.ac.jp.

class; note that memory needs to be allocated beforehand for their storage. Conversely, the pixels of a bitmap% object can be set with the set-argb-pixels method. Finally, a bitmap% object can be saved to an image file with the save-file method, which is how we produced the images included hereafter.

A bitmap can be conveniently displayed inside a window thanks to Racket's GUI components: it suffices to attach a canvas% object to a frame% object and to call the draw-bitmap method of the dc% class inside the paint callback of the canvas.

Finally, we have noticed that relying on the class color% provided by Racket to represent a colour is better avoided for two reasons: first, it significantly slows algorithm implementations down compared to, for instance, a simple triple (list), and second, it forbids using non-integer decimal values for RGB channel values. Non-integer decimal values are desirable for error diffusion as considered later.

3 Dynamic Palette Calculation

Two palette calculation methods plus one performance optimization solution are described in this section.

3.1 Main Approach

Be it for compression purposes or to adhere to a standard like the Graphics Interchange Format (GIF), applying a colour palette to an image has always been an essential issue of computer graphics. We thus start by considering this well-known scenario to realise our constructive proof of the relevance and practicability of the functional paradigm for image processing.

Although simple and fast, relying on a static colour palette, such as IBM's 16-colour CGA palette, produces below par results. So, given that modern computer hardware allows it, it is instead wiser to dynamically calculate an optimised palette from the image that is to be rendered.

We proceed as follows: first, we enumerate the different colours used in the image, and for each of them, we record their frequency (i.e. the number of times the colour is used in the image). This can be implemented simply: consider the list l of all the pixels (i.e. colours) making the image; get the first colour of l, say c, count in l the number n_c of pixels of same colour c and repeat this process from the list of colours that differ from c, list which becomes the new l. This can be easily realised with the partition function called in a way that it returns the list of colours equal to c, thus inducing n_c, and the list of colours different from c, thus inducing the new list l.

Now that all the image colours and their frequency have been obtained, the next task is to retain as many colours as can hold the palette, say k. A na¨ıve approach to this issue is to sort the obtained list of colours in descending order of frequencies and to copy the first k colours into the palette. (All the colours of the image are retained if there are less than or the same number as the palette size k.) This way, the palette consists of the most frequent colours of the image. Although simple, this first dynamic palette calculation method produces unsatisfactory results. A picture is worth a thousand words: refer to Figure 1a.

So, instead of selecting the colours to be retained inside the palette depending on their respective frequencies, it is indeed better to group colours according to their similarity: for two similar colours, the one with the higher frequency is retained, the other discarded. Precisely, we start by sorting the image colours in ascending order of their frequency so that the most infrequent colours will be grouped, that is eliminated, first; say this is the colour list l. Then, we iterate l, starting with its first colour, say c, each time finding within $l \setminus \{c\}$ the colour that is the nearest to c, say c', and we retain from c and c' only the one with the higher frequency, as explained. This is repeated with the new, smaller sorted list of colours $l \setminus \{c\tilde{}\}$, with $c\tilde{} \in \{c, c'\}$ the discarded colour. This iteration is terminated as soon as the number of the remaining colours, that is $|l|$ the size of l, is smaller than or equal to the palette size. The superiority of this second dynamic palette calculation method is clear: refer to Figure 1b.

Finally, a word on palette application to an image: in one single pass, for each pixel (colour) of the image, iterate the palette to find the nearest colour, which is stored as the new pixel value. The nearest colour is found by simply summing the difference between each of the three RGB channels.

3.2 Optimization

If instead of relying, as previously, on a simple list to count distinct colours we rely on a hash table, performance can be raised since Racket provides a hash table mechanism with constant time access operations. Furthermore, hash tables can be immutable, which allows us to avoid any trade-off with the functional paradigm for that matter.

Concretely, we iterate the image pixels only once (i.e. in a single pass), each time incrementing the hash value corresponding to the pixel colour; colours serve as hash keys.

Source code is given in Listing 1 to illustrate the elegance of this optimised approach which induces significantly higher performances (refer to Section 3.3).

Listing 1: Counting colours faster, in one pass, with a hash table.

```
1 (hash->list ; returns colours and frequencies conventionally as a list
2   (foldl (lambda (c hash-table) ; 'c' is the current colour
3            (let ([current-value (hash-ref hash-table c 0)])
4              (hash-set hash-table c (add1 current-value))))
5     (hash) image-colours)) ; '(hash)' returns a new, empty hash table
```

3.3 Quantitative Evaluation

We begin by considering the worst-case time complexity of the non-optimised approach to colour enumeration and frequency calculation (i.e. based on the partition function). In the worst case, which corresponds to an image with no two pixels of the same colour, for each of the n pixels of the image, the remaining pixels are split into pixels of the same colour (none in the worst case as just explained) and pixels of a different colour. This process is repeated for each pixel, each time starting over from

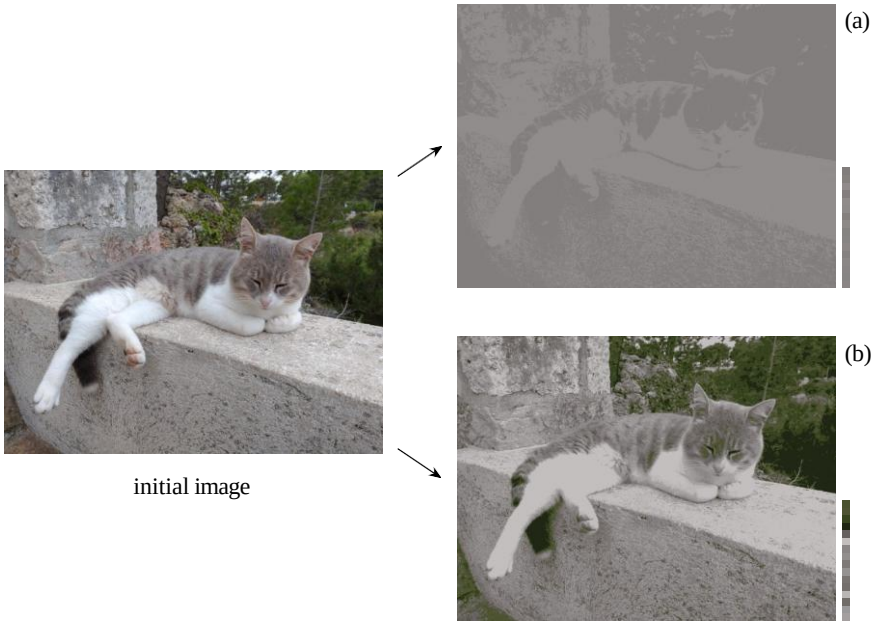

Figure 1: Dynamic palette calculation and application to a sample photograph. The 16-colour palette is shown beside the picture. (a) Na¨ıve dynamic palette calculation. (b) Improved dynamic palette calculation. (Photograph taken by the author.)

the remaining pixels (i.e. the pixels not classified yet). Hence, the worst-case time complexity of this colour enumeration and frequency calculation is $O(nk) = O(n^2)$ with k the number of distinct colours in the image.

On the other hand, the optimised approach for colour enumeration and frequency calculation based on a hash table is faster: since hash table access operations are constant time $O(1)$, one single pass of the n image pixels suffices, thus inducing an $O(n)$ time complexity.

Next, the enumerated colours need to be grouped. So, in either approach, palette calculation requires an additional time complexity of $O(\log(k/p) \times k \log k)$, with k the number of distinct colours in the image and p the maximum number of colours inside the palette. Indeed, since each iteration of the k enumerated image colours eliminates at least one and at most $k/2$ colours, a palette of size p will be obtained after $\log(k/p)$ iterations. So, the k enumerated image colours are sorted no more than $\log(k/p)$ times, which induces the dominant time complexity.

As a result, in the worst case (i.e. $k = n$), the non-optimised approach requires $O(n^2 + n \log n \log(n/p))$ and the optimised one $O(n \log n \log(n/p))$. Moreover, we have empirically confirmed the theoretically established worst-case time complexity with a computer experiment: we have run an implementation of the described dynamic palette calculation algorithm for several image files as follows. We have selected one photograph, so that numerous colours be included and palette calculation be thus meaningful, and we have resized it to produce several other image files of lower resolutions. The different image resolutions have thus enabled us to vary the value of n.

Next, we make a remark regarding the number of colours k. While variations of k in the image files selected for this experiment could impact the measurement of the average time complexity, in the worst case k equals n, a case which has been considered when establishing the worst-case time complexity above. So, variations of k will not prevent experimentally confirming the theoretically established worst-case time complexity. In practice, the original photograph, likely because of physical limitations of the camera sensor or the camera image compression algorithm, may have a lower k/n ratio, and even a lower k, than after applying a first resizing operation. So, in an attempt to stabilise the k/n ratio and thus to estimate the average time complexity by emphasising the variations of n, we consider only images resulting from at least one resizing operation. Hence, the original photograph is not used in this experiment other than to produce the experiment images by resizing.

This experiment has been conducted on a computer running the Debian GNU/Linux 12 (64-bit) OS equipped with a 12th generation Intel Core i5-12400 processor and 16 GB RAM. The experimental results show the difference between the non-optimised dynamic palette calculation method, based on the `partition` function, and the optimised dynamic palette calculation method, based on a hash table. Time measurements were reported by the `time` function of Racket (applied to the dynamic palette calculation function), whose "real time" value was retained. The photograph of Figure 1 before applying a palette has been considered in the following different resolutions (in pixels): 591×443 (i.e. $n = 261813$), 443×332 (i.e. $n = 147076$), 296×222 (i.e. $n = 65712$) and 148×111 (i.e. $n = 16428$). The number of colours k was 29 721, 23 190, 16 326 and 6 854, respectively. The palette size p was fixed to 16. The obtained results are illustrated in Figure 2.

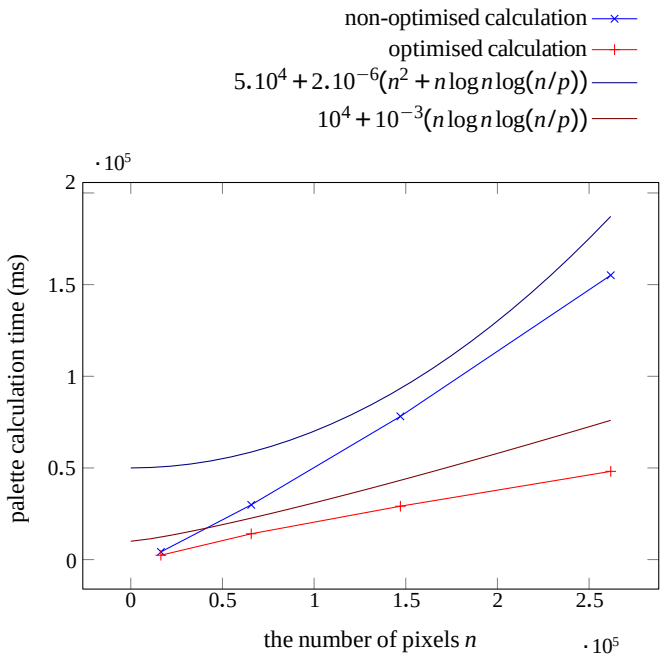

Figure 2: Experimental measurement of the dynamic palette calculation time with and without optimisation, function of n the number of pixels. Theoretical estimations of the worst-case time complexity (with coefficients for visibility) are also plotted for reference.

Experimental evaluation shows that the evolution (slope) of the measured execution times is significantly slower than the theoretical worst-case estimation, which first confirms what has been theoretically established and second is a positive indicator of the performance of the algorithm and of its implementation, and thus of the practicability of the functional paradigm in this case.

4 Application to Dithering

In order to further inspect the practicability of functional programming for image processing, we next consider a dithering (error diffusion) algorithm for images based on a colour palette.

We have selected the well-known Floyd-Steinberg dithering algorithm whose approach is to calculate the difference between the current pixel's colour and the nearest colour inside the palette, and to next diffuse with predefined coefficients this error to neighbours of the current pixel, precisely to the east pixel, south west pixel, south pixel and south east pixel of the current pixel.

This algorithm can be implemented in accordance to the functional paradigm with a function that takes as parameters the image pixels as a list of colours, the palette to apply and the image width and height. The returned value is the new image pixels, as a list of colours. Dithering is applied in one single pass, with thus a worst-case time complexity of $O(n)$.

Application of this error diffusion algorithm implementation to a sample image is illustrated in Figure 3.

We have conducted an empirical evaluation in similar conditions as for the experiment of Section 3.3, this time with the photograph of Figure 3 before applying a palette. The following distinct resolutions (in pixels) were used: 591×787 (i.e. $n = 465117$), 443×590 (i.e. $n = 261370$), 296×394 (i.e. $n = 116624$) and 148×197 (i.e. $n = 29156$). The number of colours k was 42 216, 33 210, 23 636 and 9 629, respectively. The palette size p remained fixed at 16. The time taken by the dithering process is shown in Figure 4.

As in the previous experiment, the empirical results show the efficiency of our implementation as the evolution of the measured dithering times is slower than the theoretical estimation.

5 Going Further: Parallel Processing

We complete this constructive proof of the practicability of functional programming for image processing by considering parallel processing.

5.1 Threads

First, we have relied on Racket threads to conduct dithering in parallel. The idea to enable parallel processing for the Floyd-Steinberg dithering algorithm is to divide the image into several consecutive areas and to process each of those in a separate thread. Each thread applies dithering on its area and returns the result. Results are then merged back into one single image.

We have used two threads, in addition to the control (main) thread, for our experiments, with thus the original image divided into what we call the upper half and the lower half. One can note that the first pixel row of the lower half does not fully aggregate error since the previous pixel row, that is the last pixel row of the upper half, is treated separately in another thread, and without resource sharing. Therefore, error is not diffused from the last pixel row of the upper half to the first pixel row of the lower half. It is however merely a remark since this does not produce artefacts and thus goes unnoticed.

An excerpt of our implementation with threads is given in Listings 2 and 3.

Listing 2: Parallel processing for dithering with threads: thread creation and result reporting to the main thread.

```
1 (define (create-thread parent-thread half-id half-image palette bitmap-width
        half-bitmap-height)
2   (thread (lambda () (let ([half-result (apply-palette-dithering half-image
        palette bitmap-width half-bitmap-height)])
3     ; completed: report the result to the main thread, together with
          the half identifier
4     (thread-send parent-thread (cons half-id half-result))))))
```

(The function apply-palette-dithering applies the dithering algorithm as described in Section 4.)

Listing 3: Parallel processing for dithering with threads: the control (main) thread.

```
1 (define (dith-thread image-colours palette bitmap-width bitmap-height)
2   (let* ([upper-half-height (floor (/ bitmap-height 2))]
```

(a) (b)

Figure 3: 16-colour palette: (a) no dithering; (b) dithering applied. (Photograph taken by the author.)

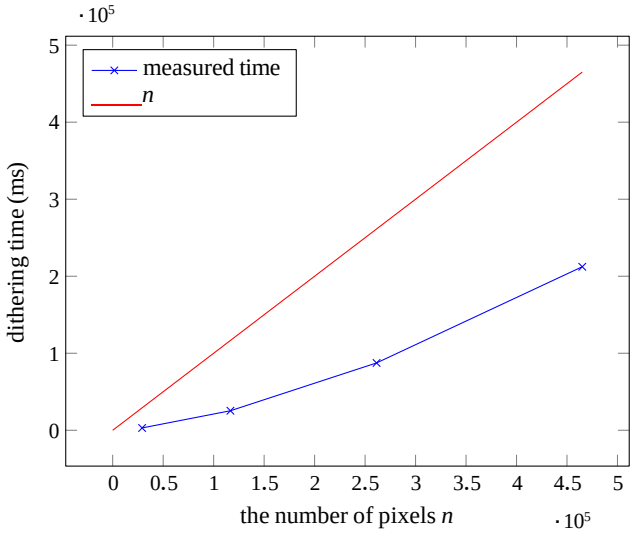

Figure 4: Experimental measurement of the dithering time, function of n the number of pixels. The theoretical estimation of the worst-case time complexity is also plotted for reference.

```
    [lower-half-height (- bitmap-height upper-half-height)])
(let-values ([(upper-half lower-half) (split-at image-colours (*
      bitmap-width upper-half-height))])
  (let* ([thread1 (create-thread (current-thread) 'upper-half upper-half
      palette bitmap-width upper-half-height)]
    [thread2 (create-thread (current-thread) 'lower-half lower-half
      palette bitmap-width lower-half-height)]
    [id-data1 (thread-receive)] ; receive one half of the result
    [id1 (car id-data1)] [data1 (cdr id-data1)]
    [data2 (cdr (thread-receive))]) ; receive the other half
   (thread-wait thread1) ; for safety as 'thread-receive' signals...
   (thread-wait thread2) ; ... that the thread is about to terminate
   (if (eq? id1 'upper-half) ; merge the results based on the half
       identifier
     (append data1 data2) (append data2 data1))))))
```

Moreover, it is recalled that Racket threads run on one single core of the processor, even if several are available. One should note that our implementation is elegant, short and using thread mail boxes (thread-send, thread-receive). There are no shared resources across threads (no concurrent access), which will be even more important for the next section on futures.

We have conducted an empirical evaluation in similar conditions as for the experiment of Section 4, notably with the same four image files, but this time with the multithreaded implementation. As explained, two threads in addition to the control (main) thread were used. The measured dithering times are summarised in Table 1 together with the experimental results of the sequential implementation for comparison.

Table 1: Experimental measurement of the dithering time induced by the multithreaded implementation, function of n the number of pixels. The results in the case of the sequential implementation are included for reference.

Image resolution (pixels)	Sequential implementation (ms)	Multithreaded implementation (ms)	Speed-up factor
148×197	3 032	2 369	1.28
296×394	25 365	18 041	1.41
443×590	87 420	61 370	1.42
591×787	212 302	148 652	1.43

The empirical results show significant speed-up compared to the sequential implementation. Furthermore, the speed-up value is rather stable at approximately 1.4. Which is remarkable in that

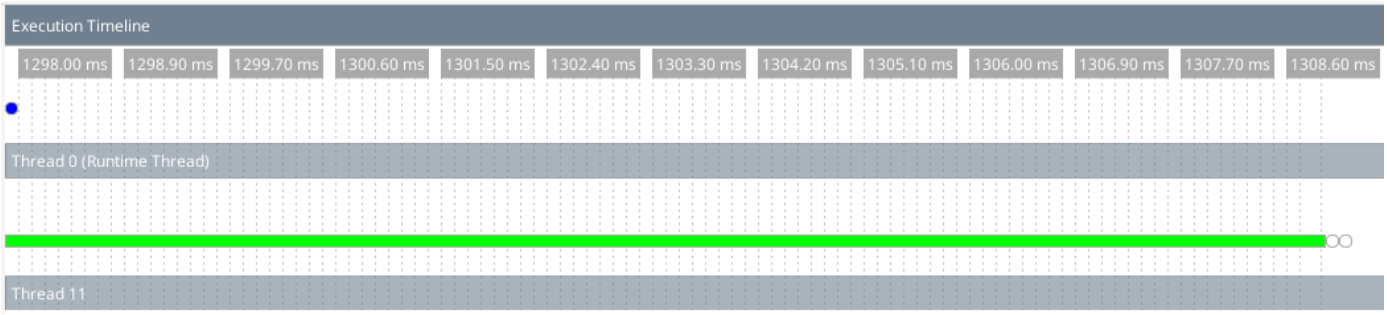

Figure 5: Output of the future visualizer tool: the green bar from start to end of the dithering algorithm execution shows that the main thread and the future's thread successfully run fully in parallel on distinct CPU cores.

as explained, all the Racket threads run on one single processor core.

5.2 Futures

In order to achieve parallel processing, unlike threads, on *several* cores of the processor, Racket provides the "future" mechanism. Parallelism with futures can however become rapidly hampered when information from the main thread is required, thus blocking parallel processing. This is the case, for instance, when I/O operations are conducted.

As with the single core multithreaded approach above, we divide the image into two parts, the upper and lower half. The upper half is treated in parallel by a future, and the lower half in the main thread. Both results are eventually merged and returned. Refer to Listing 4.

Listing 4: Parallel processing on several cores for dithering with futures.

```
1 (define (dith-future image-colours palette bitmap-width bitmap-height)
2   (let* ([upper-half-height (floor (/ bitmap-height 2))]
3          [lower-half-height (- bitmap-height upper-half-height)])
4     (let-values ([(upper-half lower-half) (split-at image-colours (*
           bitmap-width upper-half-height))])
5       (let* ([future1 (future (lambda () (apply-palette-dithering upper-half
                palette bitmap-width upper-half-height)))]
6              [lower-half-result (apply-palette-dithering lower-half palette
                     bitmap-width lower-half-height)]
7              [upper-half-result (touch future1)])
8         (append upper-half-result lower-half-result)))))
```

The results obtained from this multithreaded implementation based on futures show that parallelism on several cores of the CPU has been successfully achieved. When applied to the smallest of the four sample images of the previous experiment, the future visualizer tool output is as shown in Figure 5. The topmost row represents the main thread, for us processing the lower half of the image, and the second row corresponds to the future's thread: it displays a green bar spanning the whole dithering execution time. This uninterrupted green bar means that the corresponding future has been successfully run fully in parallel to the main thread, on a distinct CPU core. This desirable situation is enabled by the absence of shared resources, and communication in general, between the main thread (processing the lower half of the image) and the future's thread (processing the upper half of the image).

We have conducted an empirical evaluation in similar conditions as for the experiment of Section 4, notably with the same four image files, but this time with the multithreaded implementation based on futures. As explained, one future in addition to the control (main) thread was used. The measured dithering times are plotted in Figure 6 together with the experimental results of the sequential and single core multithreaded implementations for comparison.

This empirical evaluation shows that parallel processing on several cores, when successfully achieved by futures as explained (see Figure 5), further significantly reduces the time required to apply the dithering algorithm to the image: as shown in this figure, we measured a 1.87, 1.90, 1.98 and 1.94 speed-up factor for the four images, respectively, compared with the single core multithreaded approach.

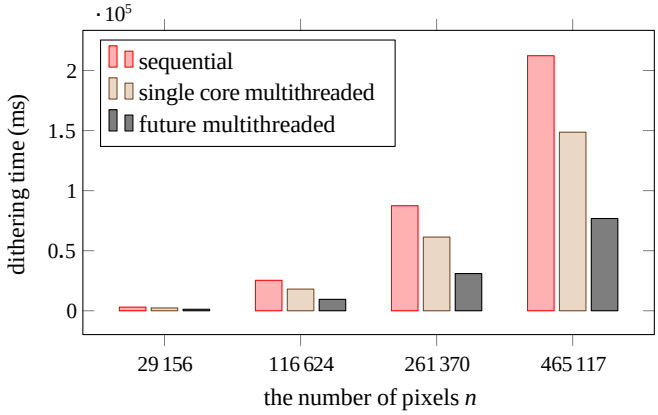

Figure 6: Experimental measurement of the dithering time induced by the implementation based on futures, function of n the number of pixels. The results in the case of the sequential and single core multithreaded implementations are also displayed for reference.

6 Concluding Remarks

Through this work, we have successfully shown that functional programming, with some minor exceptions to the functional paradigm, is capable for image processing. Image processing algorithms, such as dynamic palette calculation and dithering (error diffusion) can be elegantly implemented. Moreover, even parallel processing, with threads on one single core and with futures on distinct physical cores, is feasible and brings significant performance improvements, as quantitatively shown by our experiments.

Besides, parallelization is notoriously challenging for programmers, and often harmful to program robustness. Thanks to the functional paradigm, robustness is retained when implementing parallel computation tasks. Overall, because Racket is a high-level language and tolerates exceptional imperative programming constructs, it features a very high usability, notably confirmed throughout our experiments.

Although we have been able to show to some degree the applicability and practicability of functional programming, and more generally of the functional paradigm, to image processing, the experimentally measured processing times remain relatively high. So, as future work, it would be interesting to next compare the achieved performances and those obtained from an implementation based on an imperative or object-oriented programming language. Such a discussion could also be extended to a pure functional language such as Haskell: it would certainly be more difficult to manipulate, but the existing libraries, like GUI ones for Haskell, could perhaps facilitate implementation. Finally, showing whether graphical animation is possible too, like for interactive content, is yet another interesting subject, with however even less tolerance for long processing times.

References

[1] Antoine Bossard. The SOF programming paradigm: A sequence of pure functions. *International Journal of Software Innovation*, 10(1):1–14, 2022.

[2] Antoine Bossard and Keiichi Kaneko. A new methodology for a functional and logic programming course: On smoothening the transition between the two paradigms. In *Proceedings of the 20th Annual SIG Conference on Information Technology Education (SIGITE; Tacoma, WA, USA, 2–5 October)*, pages 63–68. Association for Computing Machinery, 2019.

[3] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, February 2018.

[4] Jason Gregory. *Game Engine Architecture*. Taylor and Francis, Boca Raton, FL, USA, 2009.

[5] Till Haenisch. A case study on using functional programming for Internet of Things applications. *Athens Journal of Technology & Engineering*, 3(1):29–38, March 2016.

[6] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[7] ISO/IEC JTC 1/SC 22. Programming languages – C++. Technical Report ISO/IEC 14882:2017, 5th edition, International Organization for Standardization, December 2017.

[8] Dimitris Kyriakoudis and Chris Kiefer. uSEQ: A LISPy modular sequencer for Eurorack with a livecodable microcontroller. In *Proceedings of 7th International Conference on Live Coding (ICLC; Utrecht, The Netherlands, 19–23 April)*, pages 1–15. Zenodo, April 2023.

[9] Simon Marlow. *Haskell 2010 Language Report*, April 2010.

[10] Rinus Plasmeijer and Marko van Eekelen. Keep it clean: a unique approach to functional programming. *ACM SIGPLAN Notices*, 34(6):23–31, June 1999.

[11] Franco Raimondi, Giuseppe Primiero, Kelly Androutsopoulos, Nikos Gorogiannis, Martin J. Loomes, Michael Margolis, Puja Varsani, Nick Weldin, and Alex Zivanovic. A Racket-based robot to teach first-year computer science. In Kent M. Pitman, editor, *Proceedings of the 7th European Lisp Symposium (ELS; Paris, France, 5–6 May)*, pages 54–62, 2014.

[12] Jocelyn Se´rot, Georges Que´not, and Bertrand Zavidovique. Functional programming on a dataflow architecture: Applications in real-time image processing. *Machine Vision and Applications*, 7:44–56, December 1993.

[13] Brian Terlson. ECMAScript 2018 language specification. Technical Report ECMA-262, 9th edition, Ecma International, June 2018.

[14] Mikus Vanags and Rudite Cevere. The perfect lambda syntax. *Baltic Journal of Modern Computing*, 6(1):13–30, 2018.

Antoine Bossard is a Professor of the Graduate School of Science of Kanagawa University in Japan. He received the BS and MS degrees from Universite´ de Caen Basse-Normandie, France in 2005 and 2007, respectively, and the Ph.D. degree from Tokyo University of Agriculture and Technology, Japan in 2011. Amongst others, he is in charge of the computer architecture and functional programming lectures for undergraduate students, and of a graph theory lecture for master students. His research activities are focused on interconnection networks (e.g. network topologies, routing problems, fault tolerance) and information representation and processing of Chinese characters (e.g. fingerprinting). He is a Senior Member of ACM and a member of TUG.