Parallel Dithering: How Fast Can We Go ?

Quentin Guilloteau

▶ To cite this version:

Quentin Guilloteau. Parallel Dithering: How Fast Can We Go ?. 2022. hal-03594790

HAL Id: hal-03594790
https://hal.science/hal-03594790v1

Preprint submitted on 2 Mar 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Parallel Dithering: How Fast Can We Go ?

Quentin Guilloteau
ENSIMAG & MoSIG DI

Abstract

From palette reduction in GIF images to printing, dithering techniques are widely used. With the increasing size of the images today, due to better photographic hardware, we can wonder if we can gain in performance by processing an image in parallel. In this paper, we will implement a MPI algorithm to apply the Floyd-Steinberg dithering in parallel. We will then focus on performance and look for optimal values of parameters.

1 Introduction

1.1 Dithering

The action of dithering is used in image processing to reduce the number of colors used in an image. One example is for printers that have only two values: either with ink or without.

The error diffusion in the dithering process consists in spreading to neighbouring pixels the quantification error due to the restriction of a pixel to a reduced palette. This technique reduces phenomena such as banding (i.e. inaccurate color presentation) but introduces some noise.

In this paper, we will focus on the palette reduction of a grey scale image (256 values) to a black and white image (2 values: 0 and 255).

Figure 1 represents a grey scale image, and figure 2 is its dither image.

We can see on figure 2 the dots composing the image. Even with only two pixel values, the contrast of the different parts of the image is repescted.

1.2 Floyd-Steinberg Dithering

There are many different possible ditherings. They all use the same principle, only some numerical constants change. In this paper, we will only focus on

Figure 1: Grey Scale Image

Figure 2: Dithered Image

the **Floyd-Steinberg Dithering**. It uses the error diffusion pattern depicted in figure 3.

The dithering works as follows:

1. We assign a new value to the current pixel
2. We compute the error of this pixel as the difference between the new value and the old value
3. [Error Diffusion] We add a fraction of this error to the neighbouring pixels according to figure 3

For example, if the error for the current pixel is 42, we will add $\frac{7}{16} \times 42$ to the value of the pixels on its right.

Figure 4 shows another way to look at the problem,

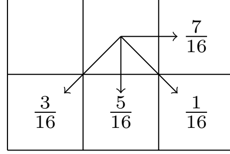

Figure 3: Error Diffusion for the Floyd-Steinberg Dithering

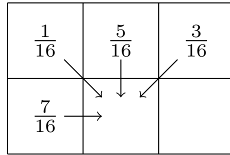

Figure 4: Local Dependencies of the Floyd-Steinberg Dithering

by considering the dependencies for a single pixel.

1.3 Pseudo-Code

We can write the pseudo code of the Floyd-Steinberg Dithering:

```
for (y = 0; y < rows; y++) {
  for (x = 0; x < cols; x++) {
    // Computation of the error
    int old_value = pixels[y * cols + x];
    int new_value = (current_value < 127) ?
                    0 : 255;
    int error = old_value - new_value;
    pixels[y * cols + x] = new_value;

    // Error Propagation
    pixels[(y + 0) * cols + (x + 1)]
        += error * 7 / 16;
    pixels[(y + 1) * cols + (x + 1)]
        += error * 1 / 16;
    pixels[(y + 1) * cols + (x + 0)]
        += error * 5 / 16;
    pixels[(y + 1) * cols + (x - 1)]
        += error * 3 / 16;

  }
}
```

As we can see, this algorithm is *highly sequential*. We have to start from the top left of the image and work ourselves to the right until we reach the end of the line. Then we start again from the next line.

2 Experimental Setup

Let us present the experimental setup used for every experiment presented on this paper.

2.1 Experimental Design

All the experiments presented in this paper have 2 associated R scripts:

- A script generating the design of the experiment.
- A script reading the design of the experiment and running it.

The first one will generate a CSV file composed of all the configurations to benchmark during the experiment. The second one will read the first one and then run the correct MPI commands to run the experiment.

2.2 Hardware

For all the experiements, we used the Grid5000 **dahu** cluster located in Grenoble.

The hardware on this cluster is:

- CPU: 2 x Intel Xeon Gold 6130
- Cores: 16 cores/CPU
- Memory: 192 GiB
- Storage: 240 GB SSD + 480 GB SSD + 4.0 TB HDD
- Network: 10 Gbps + 100 Gbps Omni-Path

3 Important Notions

We will here give some notions and notations that we will use for the rest of this paper.

- w: processing time of a single pixel
- H: height of the image
- W: width of the image
- p: number of processes
- B: bandwidth of the network
- L: latency of the network
- $T(H, W, p)$: execution time of the algorithm depending on the image and the number of processors

1	2	3	4	5
3	4	5		
5				

Figure 5: Potential Parallel Execution

- $S(H, W, p) = \frac{t_{seq}}{t_{par}}$: speedup of the algorithm for an image of size $H \times W$ with p processors (the greater the better)
- $Eff = \frac{S(H,W,p)}{p}$: efficiency (the greater the better)

4 Parallel Dithering

The main idea of the parallel dithering, is that the progression looks more like a triangle than a rectangle:

Figure 5, shows that there are indeed some possible parallelism in the dithering process. The numbers correspond to the order of the execution. When some pixels have the same number, it means that they can be processed in parallel.

We also see that each line needs to be 2 pixels ahead of the line below due to the error diffusion pattern of the dithering algorithm (see figure 3).

4.1 Alternate Processes: Presentation

The first idea to process the image in parallel is to alternate processes and giving them one line at the time to work with.

Figure 6 gives an example of the distribution of data between 2 processes.

Process 0 will start working on the first pixel of its line. As we saw in figure 5, a process must be done processing pixel n of its line for the next process to be able to process the pixel $n - 2$ of its own line. So, we have to make sure to respect this requirement.

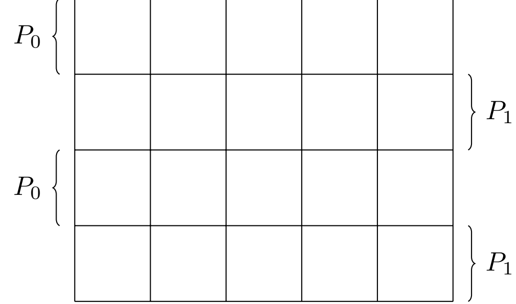

Figure 6: Data Distribution per Process (with 2 processes)

4.2 Implemenation Details

4.2.1 Representing one Pixel

One decision made to simplify the code was to encode pixel values on a `int16_t` integer.

Of course, pixels in a grey scale image only go from 0 to 255, only requiring 8 bits.

However, as we will need to add or substract error values to the value of a pixel, being able to have negative values was crucial.

4.2.2 Distributing the lines to the processes

As one process has non adjacent lines, we have to define our own `MPI_Vector_Type` to properly send the correct lines to each process.

As we want process P_i to have $\frac{H}{p}$ lines with $p - 1$ lines between each line and stating with the ith line.

```
// Definition of the custom type
MPI_Type_vector(h / world_size,
                w, world_size * w,
                MPI_INT16_T, &PixelLine);
MPI_Type_commit(&PixelLine);
```

Unfortunatly, it is not possible to simply use the `MPI_Scatter` function to send the pixels to the processes. Indeed, it will start the next `PixelLine` at the end of the previous one, however, we want it to start at the same position that the previous one with an offset of `w`.

So we decided to simply call the `MPI_Send` function manually to scatter the lines among the processes.

4.2.2.1 Processing a line

Apart for the first line of the first process, every process has to receive the error from the above process to be able to process the pixels of its current line.

```
// Call to recv the error from above
MPI_Recv(&error_from_top, 1, MPI_INT16_T,
    (my_rank + world_size - 1) % world_size,
    0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Once the error from the process above has been received, we need to update the value of the current pixel.

```
// Updating the local value
local_pixels[i + w * line_index]
        += error_from_top;
```

At this point, we can compute the new value for this pixel and its error to propagate.

```
// Processing the current pixel
int16_t current_value =
    local_pixels[i + w * line_index];
int16_t new_value = (current_value < 127) ?
                    0 : 255;
local_pixels[i + w * line_index] = new_value;
int16_t error = current_value - new_value;
```

4.2.3 Propagating the error to the process below

As figure 3 shows, once its value updated, one pixel has to send its error to (at most) 3 pixels to the line below. However, in order for the first error of the line to be send to the process below, we need to wait for the second pixel to be done processing as we want to send the error only once.

We decided to use a circular buffer of size 3 to manage this issue.

This buffer will store the cumulated errors to send to the process below until they are ready to be sent. We need a size of 3 because a pixel propagates its error to (at most) 3 pixels on the line below.

Figure 7 resumes the mechanism used with the buffer.

1. The first pixel of line i is processed on process P_k. We add the proportions of the error to the corresponding cells of the buffer. In this example, there are only 2 cells to update as we are on the far left on the image (see figure ??)

2. The second pixel of line i is processed on process P_k. We add the propotions of the error to the corresponding cells of the buffer.

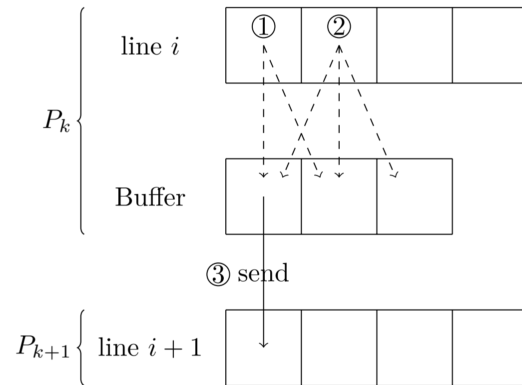

Figure 7: Use of the cicular buffer

3. The first cell of the buffer is ready to be send to process P_{k+1} (no more dependencies). It contains the cumulated error for the first pixel of line $i+1$.

4. Once sent, we set the value of the sent cell back to 0.

As the buffer is circular, the first cell will be used to store the error for the right dependency of the 3rd pixel.

4.3 Performance Analysis

Let p be the number of processors. Let us consider an image of size $H \times W$. Let L be the latency of the network and B its bandwidth. Let w be the processing time of a pixel.

Each processor has $\frac{H \times W}{p}$ pixels to process.

The global time spent "busy waiting" is $2(H-1)w$ (because of the 2 pixels spacing between processes) and we have to wait for the last process to finish.

We thus have:

$$T(H, W, p) = \frac{HW}{p}\left(w + L + \frac{1}{B}\right) + 2(H-1)w \quad (1)$$

The sequential time is: wHW.

We can thus compute the speedup:

$$S(H, W, p) = \frac{wHW}{\frac{HW}{p}\left(w + L + \frac{1}{B}\right) + 2(H-1)w}$$

The limit speedup is thus:

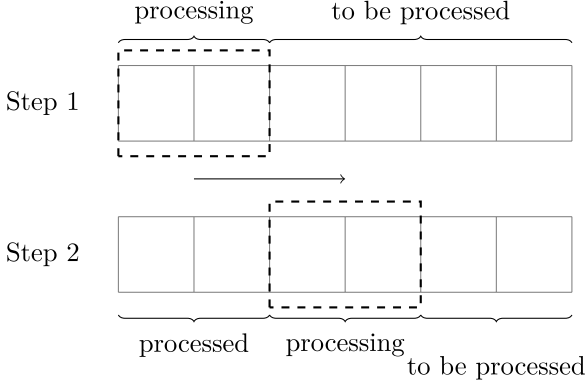

Figure 8: Execution per block with $k = 2$

$$\lim_{(H,W)\to(\infty,\infty)} S(H,W,p) = \frac{1}{\frac{1}{p} + \frac{L}{wp} + \frac{1}{wpB}} \quad (2)$$

The efficiency of this algorithm is:

$$Eff = \lim_{(H,W)\to(\infty,\infty)} \frac{S(H,W,p)}{p} = \frac{1}{1 + \frac{L}{w} + \frac{1}{B\times w}} \quad (3)$$

The efficiency of this algorithm (equation 3) is less than one. It thus means that it is not very efficient.

5 Reducing the Granularity

5.1 Presentation

Sending the error every time a process process a pixel introduce too much loss due to the latency of the network. We will try to reduce the granularity of the algorithm by grouping pixels per block. We will also send the errors per block. We call k the number of pixels in a block. Figure 8 summarise the execution of a line of pixels using blocks.

5.2 Performance Analysis

We note k the size of a block. So a block contains k pixels.

$$T(H,W,p) = \frac{HW}{p\times k}\left(w\times k + L + \frac{k}{B}\right) + 2k(H-1)w \quad (4)$$

We compute next the speedup of this new version:

$$S(H,W,p) = \frac{wHW}{\frac{n^2}{p\times k}\left(w\times k + L + \frac{k}{B}\right) + 2k(H-1)w}$$

The limit speedup is thus:

$$\lim_{(H,W)\to(\infty,\infty)} S(H,W,p) = \frac{1}{\frac{1}{p} + \frac{L}{wpk} + \frac{1}{wpB}} \quad (5)$$

The efficiency of this algorithm is:

$$Eff = \lim_{(H,W)\to(\infty,\infty)} \frac{S(H,W,p)}{p} = \frac{1}{1 + \frac{L}{wk} + \frac{1}{Bw}} \quad (6)$$

We see that we improve the efficiency of the algorithm by limiting the use of the network and reducing the global cost of the latency. So, the higher k, the less we will pay the cost of the latency.

5.3 Upper bound for k

We want k to be the higher possible, but if we increase k, we will at some point have some busy waiting time.

Let us find the upper bound for k that does not generate busy waiting time.

5.3.1 Theoretical Value

Let W be the width of the image and p be the number of processes.

In order to not have any idle time by the processes, we would like the process P_{p-1} to have at least finished processing its first 2 blocks of its line when process P_0 is done processing its line.

Otherwise, process P_0 would have to wait for process P_{p-1} to send the error of the first block, thus creating some busy-waiting time.

Let k_{hi} be the lower bound of k such that there is no busy waiting by the processes.

There are $\frac{W}{k}$ blocks to process on one line.

Once the first line will be done by process P_0, process P_{p-1} would have processed $\frac{W}{k} - 2\times(p-1)$.

We want the last process to have processed at least 2 blocks (so it can send the error of the first block to process P_0).

Thus,

$$\frac{W}{k} - 2 \times (p-1) \geq 2 \implies k \leq \frac{W}{2 \times p} = k_{hi} \quad (7)$$

5.3.2 Experiment

We took an example with an image of width 8192 pixels on 16 processors. We doubled the block size starting from 2 pixels up to the total width of the image.

We can compute the upper bound for k:

$$k_{hi} = \frac{W}{2p} = \frac{8192}{2 \times 16} = 256 \quad (8)$$

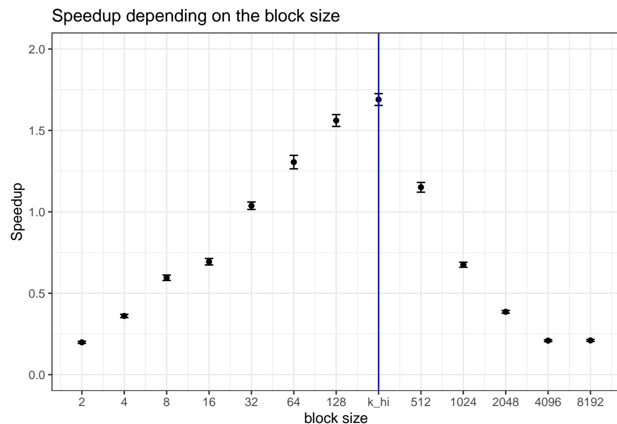

Figure 9: Speedup depending on the block size (k)

In figure 9 we show that the speedup is indeed maximal for $k = k_{hi}$.

5.3.3 Remarks

We can also see a small dip in performance for $k = 16$. We suspect that it must be linked to MPI having different ways to send the data depending on the size of the message compared to a threshold.

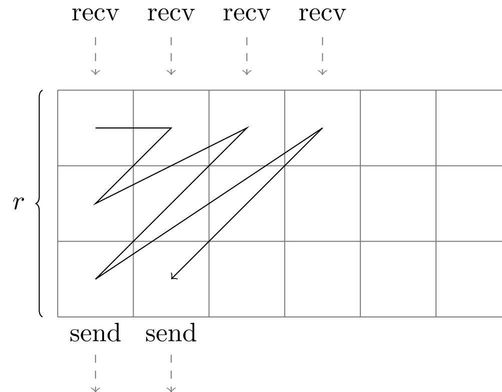

Figure 10: Zig-Zag processing on a block of $r = 3$ lines

6 Limiting the Impact of the Bandwidth

6.1 Presentation

For the moment, we only managed to reduce the cost of the latency of the network. In order to reduce the impact of the bandwith, we must send less messages through the network. In the previous section, we increased the size of the messages by sending pixels per block. In this section, we will create blocks of lines. Each process will have several blocks of lines. Each block of line will contain r **consecutive** lines of pixels. Only the top and bottom lines of the block will require communications. The remaining pixels will be processed sequentially.

We cannot however afford to process the pixels in a block of lines, sequentially line by line from left to right. Indeed, this would result in too much time "busy waiting" for the processes. We thus decided to process the pixels in the block of lines in a zig-zag (or serpentine) way, as depicted in figure 10.

6.2 Performance Anlysis

We pretty much have the same logic than in section 4.3, with blocks instead of pixels.

The waiting time is:

$$Wait = wk\left(\frac{H}{r} - 1\right)\frac{r(r-1)}{2} = wk\frac{(H-r)(r-1)}{2} \quad (9)$$

$$T(H,W,p) = \frac{HW}{krp}\left(wk + L + \frac{k}{B}\right) + wk\frac{(H-r)(r-1)}{2} \quad (10)$$

We compute next the speedup of this new version:

$$S(H,W,p) = \frac{wHW}{\frac{HW}{krp}\left(wk + L + \frac{k}{B}\right) + wk\frac{(H-r)(r-1)}{2}}$$

The limit speedup is thus:

$$\lim_{(H,W)\to(\infty,\infty)} S(H,W,p) = \frac{1}{\frac{1}{pr} + \frac{L}{pkrw} + \frac{1}{pwrB}} \quad (11)$$

The efficiency of this algorithm is:

$$Eff = \lim_{(H,W)\to(\infty,\infty)} \frac{S(H,W,p)}{p} = \frac{1}{\frac{1}{r} + \frac{L}{wkr} + \frac{1}{Brw}} \quad (12)$$

We see that we improve the efficiency of the algorithm by limiting the use of the network and reducing the global cost of the bandwidth. The highest r is, the better the speedup will be.

6.3 Upper bound for r

As for k, increasing r too much will produce some busy waiting time. Let us find the upper bound for r that does not produce busy waiting time.

6.3.1 Theoretical Value

In order to process the ith block of pixel of the last line of the block of line, we need to process the $(i+1)^{th}$ block of the previous line of the block.

We can then compute the number of block to process first to be able to process the ith block of the last line:

$$\sum_{j=0}^{r-1} i + j = r \times i + \frac{(r-1)r}{2} \quad (13)$$

So, to send the first error to the process below, we need to process the first 2 blocks of the last line (i.e. $i = 2$). Thus, we need to process the total of $\frac{r(r+3)}{2}$ blocks of pixels.

The working time is then:

$$Work = wk\frac{r(r+3)}{2} \quad (14)$$

We don't want to have processes being busy waiting between when they finish processing their previous block of lines and when they start their next block of lines.

For processes other than the first one, there are actually unavoidable waiting times. Indeed, for example, to receive the second error, a process has to wait for the process above to send it. However, there is not enough work to do yet to be busy until the error is sent.

The waiting time for such a process is:

$$Wait = wk\sum_{i=1}^{r}(r-i) = wk\frac{r(r-1)}{2} \quad (15)$$

Thus, for processes that are not the first process, the time to send the first error is:

$$Work + Wait = wk\frac{r(r+3)}{2} + wk\frac{r(r-1)}{2} = wkr(r+1) \quad (16)$$

Thus, the total time to get the first error in the second block of line of the first process is:

$$wk\frac{r(r+3)}{2} + wk(p-1)r(r+1) = wkr\left(r\left(p-\frac{1}{2}\right) + \left(p+\frac{1}{2}\right)\right) \quad (17)$$

We want the first process to finish the first block of lines after the first error for its second block of lines is sent:

$$r\left(r\left(p-\frac{1}{2}\right) + \left(p+\frac{1}{2}\right)\right) \leq \frac{W}{k}r \quad (18)$$

Thus, the upper bound for r is:

$$r \leq \frac{\frac{W}{k} - \left(p+\frac{1}{2}\right)}{p-\frac{1}{2}} = r_{hi} \quad (19)$$

We can also express the value of k given r:

$$k = \frac{W}{r\left(p-\frac{1}{2}\right) + \left(p+\frac{1}{2}\right)} \quad (20)$$

6.3.2 Experiment

We can now make an experiement.

We fixed H, W and p. We took a range of values for r and took the k value associated with this r (see equation 20). We plot the speedup for each configuration.

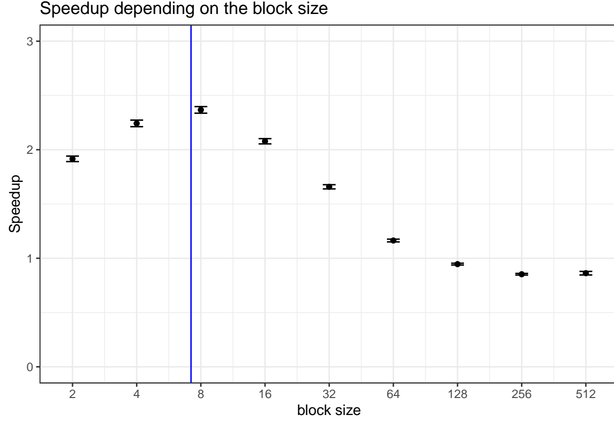

Figure 11: Speedup depending on the block size (r)

We can see the results of the experiment in fugure 11.

We see that for $r = r_{hi}$, we optain the highest speedup.

6.3.3 Remarks

If we plug $k = k_{hi}$ in the value of the upper bound for r, we get $r \leq 1$. Which makes sense as this is an optimal value of k for blocks of one line.

7 Optimal block sizes

7.1 Assumptions

In equation 19, we found a relation linking the parameters of our problem: r and k.

We can thus easily get the optimal value of one given the other. The question now is to get the best couple (r, k).

From 10, we have:

$$T = \frac{HW}{prk}\left(L + \frac{k}{B} + wk\right) + \frac{(H-r)(r-1)}{2}wk$$

Let us focus on the "busy waiting" time and try to minimize it.

$$Wait = \frac{(H-r)(r-1)}{2}k$$

We can plug the expression of k found in equation 20 and derive with respect to r:

$$Wait = \frac{(H-r)(r-1)W}{2\left(r\left(p-\frac{1}{2}\right)+p+\frac{1}{2}\right)}$$

Let us derive with respect to r:

$$\frac{(H+1-2r)(r(p-\frac{1}{2})+p+\frac{1}{2})-(H-r)(r-1)(p-\frac{1}{2})}{(r(p-\frac{1}{2})+p+\frac{1}{2})^2} \quad (21)$$

We are looking at when the upper part equals 0.

After some manipulations, we obtain:

$$r^2(2-P)+r((H+3)P-(H+1))-(2H+1)P = 0 \quad (22)$$

with $P = \frac{p+\frac{1}{2}}{p-\frac{1}{2}}$

We can assume that $P = 1$, which is reasonable for values of $p > 10$.

We thus get the following equation:

$$r^2 + 2r - (2H+1) = 0 \quad (23)$$

The positive solution, which is the r minimizing the waiting time, is:

$$r_+ = -1 + \sqrt{2(H+1)} \quad (24)$$

For large values of H we can simply take $r \simeq \sqrt{2H}$

7.2 Example

Let us take an example with a square image of size 8192×8192 with 16 processes.

```
H <- 2^13
W <- 2^13
p <- 16

r <- sqrt(2 * H)
r
```

```
## [1] 128
```

```r
k <- W / (r * (p - 0.5) + (p + 0.5))
k
```

```
## [1] 4.094976
```

We have to take the r the closest to this optimal value such that $H \equiv 0 \ (mod \ r \times p)$.

In this particular case:

```r
r <- 2^(as.integer(log2(r)))
r
```

```
## [1] 128
```

```r
k <- W / (r * (p - 0.5) + (p + 0.5))
k
```

```
## [1] 4.094976
```

We also need k to be a valid number (integer that divides W).

So, in this example, we would take $r = 128$ and $k = 4$

7.3 Experiment

We can thus make an experiment in order to check if the theorical optimal value for r is the one we found.

In this experiment, we fixed the size of the image and the number of processors. We generated points for some possible values of r and computed the optimal value of k associated with this value of r (see equation 20). We added to the dataset the optimal configuration. We then mesured the speedup for each configuration and plotted the results in figure 12.

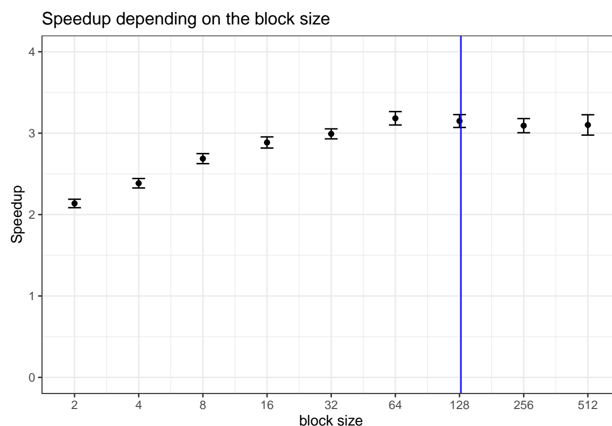

Figure 12: Speedup depending on the block size (r)

We can see that the theorical optimal value of r (in blue on figure 12) is not exactly the experiemental optimal value (around $r = 64$).

However, after $r = 64$, there seems to be a "plateau" where the speedup does not variate much. Thus by keeping the therorical optimal r, we could still get some high performances.

7.4 Summary

Given an image of size $H \times W$, and p processors, we recommend taking:

- $r \simeq \sqrt{2H}$
- $k \simeq \dfrac{W}{p\sqrt{2H}}$

8 Performances

In this experiment, we fix the image size and increase the number of processes.

The values of r and k are computed with respect to the rules given in 7.4.

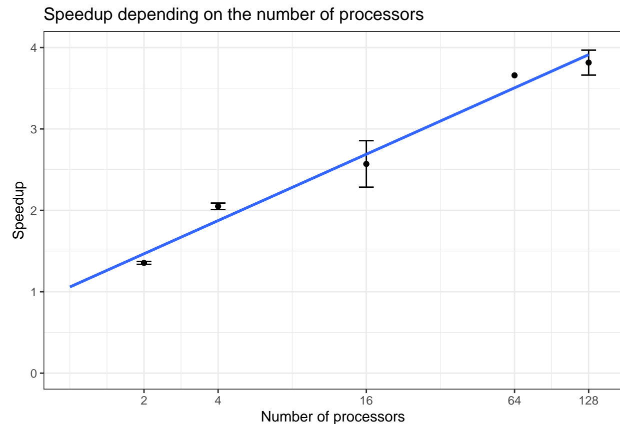

Figure 13: Speedup depending on te number of processors

We can do a linear regression for the speedup, and we find that:

$$S(p) \simeq \log_2(\sqrt{p}) + 1 \tag{25}$$

		X	$\frac{7}{48}$	$\frac{5}{48}$
$\frac{3}{48}$	$\frac{5}{48}$	$\frac{7}{48}$	$\frac{5}{48}$	$\frac{3}{48}$
$\frac{1}{48}$	$\frac{3}{48}$	$\frac{5}{48}$	$\frac{3}{48}$	$\frac{1}{48}$

Figure 14: Error Diffusion for the Jarvis, Judice and Ninke Dithering

9 Conclusion

9.1 To go further

9.1.1 Powers of 2

In this paper, we decided to limit the implementation to a “simple” use. Indeed, our implementation requires $H \equiv 0 \pmod{k}$ and $H \equiv 0\ (mod\ r \times p)$. Those constraints are limiting. This is why all the experiments are based on powers of 2.

But, by only using image with a size that is a power of 2, we limit ourselves to a very small region of the space. We hope however that measuring on those points gave us enough information on the global space.

With more time, implementing a generic program to take any k and any r would allow us to be more precise in our experiments and analysis.

9.1.2 Another Dithering Algorithm

There are several dithering algorithms. We could also compare the impact of the error-diffusion on the performances.

For instance, the **Jarvis, Judice and Ninke Dithering** has the error diffusion pattern depicted in figure 14.

9.1.3 GPU and Shared Memory

It would also be interesting to develop versions of a parallel Floyd-Steinberg Dithering on a GPU with CUDA and on shared memory (with Rust and Rayon or with OpenMP) and look at the difference in performances.

9.2 Performances

Concerning performances, we manage to reduce the execution time by:

- Working on blocks of pixels instead of single pixels: reduced the impact of the latency of the network
- Giving several consecutive lines to each process: reduced the impact of the network (latency + bandwith)
- Computing optimal values of k and r: those values are computed to reduce waiting time