Technical Report

Computer Architecture
2018-2019
University of the Basque Country

Unai Sainz de la Maza Gamboa
Adrián San Segundo Moya

Introduction

We have been commissioned to digitize the library of The Benedictine foundation in Lazkao. This digitalization process needs a lot of computational work and therefore a lot of time to complete it.

There are three main problems to solve. First, we need to filter the image to reduce the noise introduced by the scanner and get a smooth image. Second, to maintain the security of the images, we have to encrypt the images. Third, we need to make the images accessible in the online platform of the foundation.

To solve all these problems, which require a lot of work, we need a high computational power. To get all that power, we use a type of computation called parallel computing. This type of computation executes the processes simultaneously in several processors.

The final purpose of this project is to solve everything mentioned above efficiently, and as quickly as possible.

Problems

Filtering

As the scanner introduces noise into the image, this creates the need to make the image smooth by filtering it. To complete this task, we need to learn about image processing techniques.

Required knowledge

To reduce the noise and to make the image smooth, we apply a low pass filter, also called “smoothing filter”.

This filter is an spatial frequency filter type. The spatial frequency refers to roughness of the variations in DN (digital number) values occurring in an image.

A filter is an 3x3 matrix containing the coefficients. Each pixel of the image is multiplied by the adequate coefficient in the filter matrix. Then, the 9 resulting values are added and the result of that sum replaces the original central pixel.

We use as coefficients 1/N, where N is the number of pixels used by the filter matrix, in this case as we use 3x3 matrix, N is 9, so all the coefficients take the 1/9 value.

This is an illustrated example about the process explained above:

The filter matrix:

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

The original image:

26	48	15	...
35	56	95	...
24	28	41	...
...	...	...	...

We need to multiply every pixel of the original image with the filter matrix, so the new value to replace the central pixel is: V = 1/9 x (26 + 48 + 15 + 35 + 56 + 95 + 24 + 41)

So the output image will be:

			...
	38		...
			...
...	...	...	...

Solution proposed

To apply the low pass filter, we need to implement the algorithm explained, and the function needs to return the average value minus the minimum value.

This is the solution proposed to apply it:

```
for(int i = 1; i < out_page->h-1; i++) {
  for(int j = 1; j < out_page->w-1; j++) {
      int m[] = {
        out_page->im[i-1][j-1],
        out_page->im[i][j-1],
        out_page->im[i+1][j-1],
        out_page->im[i-1][j],
        out_page->im[i][j],
        out_page->im[i+1][j],
        out_page->im[i-1][j+1],
        out_page->im[i][j+1],
        out_page->im[i+1][j+1],
      };
      double sum = 0;
      for(int t = 0; t < 9; t++) sum += m[t];
      double value = sum / 9;
      out_page->im[i][j] = (int) value;
      if(value < min) min = value;
      avg += value;
    }
}
avg /= ((out_page->w-1)*(out_page->h-1));
return avg - min;
```

But the filter needs to be applied until a specific limit is reached, to do this we use that code:

```
double avgMinus = 0;
int i = 0;
do {
  avgMinus = filter_page(in_page, out_page);
  i += 1;
  printf("%d -> %f\n", i, avgMinus);
} while(avgMinus > limit);
```

As the image size increases, this process will be slow and hard to compute with a single processor. This creates the need to parallelize the program and take advantage of the multiprocessor computing.

Parallelization

The first step when we are paralleling a program is to search data dependencies. In this case we have several cases:

1. The *value* variable and the indexes needs to be private.
2. The *min* variable, has to be the minimum of all the thread, so we define it as reduction type with “min” operand. This creates a copy of min on each thread and takes the minimum of them.

3. The *avg* variable is an incremental sum of all the values generated by the loop, so we need to stay coherence. For that purpose, we define the variable as reduction type, with the sum operand (+).

As we already have the dependencies analyzed, now we have to analyze the scheduling strategy. We know that every iteration requires more or less the same time to execute it, so the most adequate strategy is the static scheduling option.

But we also try a dynamic scheduling and the guided scheduling options, and the dynamic scheduling with 4 chunk-size got the best results.

To parallelize the program we add the corresponding directives:

```
#pragma omp parallel for schedule(dynamic, 4) reduction(min:min) reduction(+:avg) private(value)
for(int i = 1; i < out_page->h-1; i++) {
  for(int j = 1; j < out_page->w-1; j++) {
      int m[] = {
        out_page->im[i-1][j-1],
        out_page->im[i][j-1],
        out_page->im[i+1][j-1],
        out_page->im[i-1][j],
        out_page->im[i][j],
        out_page->im[i+1][j],
        out_page->im[i-1][j+1],
        out_page->im[i][j+1],
        out_page->im[i+1][j+1],
      };
      double sum = 0;
      for(int t = 0; t < 9; t++) sum += m[t];
      double value = sum / 9;
      out_page->im[i][j] = (int) value;
      if(value < min) min = value;
      avg += value;
    }
}
avg /= ((out_page->w-1)*(out_page->h-1));
return avg - min;
```

The results obtained:

Results

Book 1 results

Times (ms)	
Filter	
Serial	7.102,70
4 Threads	2.003,62
16 Threads	687,95
48 Threads	393,68

Efficiencies and speed-ups	
Filter	
Prepare Efficiency 4 Threads	0,89
Prepare Speed-up 4 Threads	3,54
Prepare Efficiency 16 Threads	0,65
Prepare Speed-up 16 Threads	10,32
Prepare Efficiency 48 Threads	0,38
Prepare Speed-up 48 Threads	18,04

Book 2 results

Times (ms)	
Filter	
Serial	15677,85
4 Threads	4489,39
16 Threads	2501,54
48 Threads	700,04

Efficiencies and speed-ups	
Filter	
Prepare Efficiency 4 Threads	0,87
Prepare Speed-up 4 Threads	3,49
Prepare Efficiency 16 Threads	0,39
Prepare Speed-up 16 Threads	6,27
Prepare Efficiency 48 Threads	0,47
Prepare Speed-up 48 Threads	22,40

Book 3 results

Times (ms)	
Filter	
Serial	47996,22
4 Threads	15217,45
16 Threads	5808,11

48 Threads	1648,79

Efficiencies and speed-ups	
Filter	
Prepare Efficiency 4 Threads	0,79
Prepare Speed-up 4 Threads	3,15
Prepare Efficiency 16 Threads	0,52
Prepare Speed-up 16 Threads	8,26
Prepare Efficiency 48 Threads	0,61
Prepare Speed-up 48 Threads	29,11

Conclusions

In this section is maybe where we can see the most reasonable times descent. The speed-ups and the efficiencies they are not as good as could be in theory, but the results are reasonable, taking into account that the server is not used by us alone, and that makes the results worse.

We can see that the program gets better results as the image size increases and the number of threads used to perform it is bigger.

Encrypting

To maintain the security of th uploaded images, we need to encrypt those images using encryption techniques.

For that purpose we need to learn about this techniques, picking out an adequate encryption algorithm. As the requirements of the project says, we will use a encryption type called “Hill´s cipher”.

Required knowledge

This type of encryption uses a secret matrix known by the sender and receiver. The picked matrix is multiplied by each pixel of the image, but we get pixels in pairs.

To maintain the coherence of the data, as we know that the color scale goes from 0 to 255, we need to use the modulus (mod 256) operand.

When the receiver goes to decipher the image, it will need the inverse of the matrix, and apply the modulus (mod 256) to it to complete the decipher process.

This is best appreciated with an example:

The secret matrix K:

21	35
18	79

The pair of pixels selected:

50
64

Multiply the matrix´s to get another matrix:

3290
5956

But we need to apply mod 256 to it:

218
68

This new values are the values used to replace the original in the image.

Solution proposed

We know that the image is saved in two-dimensional matrix (variable *iy*), and is saved as one-dimensional array to (variable *dat*). To simplify the code, we use the one-dimensional array option.

The code applies the operations showed before to each pair of pixels of the image. We use two functions, one applies the operations to two pixels, and another to pass each pixel to the previous function.

The code of the function which make operations to two pixels:

```
void encrypt (unsigned char *v1, unsigned char *v2)
{

//////////////////////////////////////////////////////////////////
/* TO COMPLETE: code to calculate the encryption        */
//////////////////////////////////////////////////////////////////
        int enc_m[2][2] = {{21, 35},{18, 79}};
        int n1 = (enc_m[0][0] * *v1 + enc_m[0][1] * *v2) % 256;
        int n2 = (enc_m[1][0] * *v1 + enc_m[1][1] * *v2) % 256;
        *v1 = n1;
        *v2 = n2;
}
```

The code to apply previous function to all pair of pixels:

```
for(int i = 0; i < n-1; i+=2) {
        out_page->dat[i] = in_page.dat[i];
        out_page->dat[i+1] = in_page.dat[i+1];
        encrypt(&out_page->dat[i], &out_page->dat[i+1]);
}
```

Parallelization

There is no data dependencies, and we know that the time required to complete the loop is always more or less the same, so the normal approach would be to do it with static scheduling, but seeing that in the previous problem we get the best results with dynamic scheduling, we test all the options and we get the best results with guided scheduling.

```
int n = (in_page.h*in_page.w);
#pragma omp parallel for schedule(guided)
for(int i = 0; i < n-1; i+=2) {
        out_page->dat[i] = in_page.dat[i];
        out_page->dat[i+1] = in_page.dat[i+1];
        encrypt(&out_page->dat[i], &out_page->dat[i+1]);
}
```

Results

Book 1 results

Times (ms)
Filter

Serial	204,87
4 Threads	61,75
16 Threads	20,39
48 Threads	17,60

Efficiencies and speed-ups	
Filter	
Prepare Efficiency 4 Threads	0,83
Prepare Speed-up 4 Threads	3,32
Prepare Efficiency 16 Threads	0,63
Prepare Speed-up 16 Threads	10,05
Prepare Efficiency 48 Threads	0,24
Prepare Speed-up 48 Threads	11,64

Book 2 results

Times (ms)	
Filter	
Serial	379,62
4 Threads	127,38
16 Threads	96,97
48 Threads	26,54

Efficiencies and speed-ups	
Filter	
Prepare Efficiency 4 Threads	0,75
Prepare Speed-up 4 Threads	2,98
Prepare Efficiency 16 Threads	0,24
Prepare Speed-up 16 Threads	3,91
Prepare Efficiency 48 Threads	0,30
Prepare Speed-up 48 Threads	14,30

Book 3 results

Times (ms)	
Filter	
Serial	1066,85
4 Threads	486,51
16 Threads	214,42
48 Threads	51,05

Efficiencies and speed-ups	
Filter	
Prepare Efficiency 4 Threads	0,55
Prepare Speed-up 4 Threads	2,19
Prepare Efficiency 16 Threads	0,31
Prepare Speed-up 16 Threads	4,98
Prepare Efficiency 48 Threads	0,44
Prepare Speed-up 48 Threads	20,90

Conclusions

We get so much lower times, but the program does not scale well, because the speed-ups are far from the number of threads, and the efficiencies are not close to 1.

Uploading

Due to the need to upload images to the cloud, we need to generate a “special” error-code. This is made by processing the rows of the encrypted file. The necessary code to carry out this task has been provided, but it must be applied.

The function has to be applied to all the pixels of the images, so the logical way to do it, it is through two loops that travel through the 2x2 matrix and apply the function to each pixel of the matrix.

It gives us an important data about the function, the time required to execute it is not always the same, varies depending on the pixels and their positions. This will have to be taken into account when parallelizing the loops.

Solution proposed

As we mentioned earlier, the solution to this problem is simple. The code to apply the function to all the pixels of the image is this:

```
for (i = 0; i < in_page.h; i++)
        for (j = 0; j < in_page.w; j++)
                upload(in_page.im[i][j], i, j, in_page.h, in_page.w);
```

This is the serial solution, but this solution requires a lot of time to execute, we need to parallelize the loops. Before parallelizing the loops, is compulsory to analyze the code to decide what parallelizing strategy is the most appropriate in this case.

First of all, we realize that there is no data dependencies, so in principle, we can parallelize it without problems. To decide the scheduling type used by OpenMP, it is necessary to analyze the execution time of the functions executed in the loop. The function executed in this loop, the "upload" function, requires different execution times for each pixel depending on the pixel itself and in the position of the pixel.

Taking into account this analysis, we know that the indexes needs to be private and we think that the best option for scheduling strategy is to use the dynamic scheduling technique.

Parallelization

The first implementation of the solution is using the dynamic scheduling option in OpenMP, without specifying the chunk-size, so it defaults to chunk-size one. We test also using guided scheduling, but the times were higher, so we decide to use the first option (dynamic scheduling), but with 4 chunk-size. This is the code used:

```
void prepare_up(page in_page) {
        int i, j;
////////////////////////////////////////////////////////////////////////////
//////
/* TO COMPLETE: code to prepare the upload of the page          */
////////////////////////////////////////////////////////////////////////////
//////
        #pragma omp parallel for schedule(dynamic, 4) private(i, j)
        for (i = 0; i < in_page.h; i++)
                for (j = 0; j < in_page.w; j++)
                        upload(in_page.im[i][j], i, j, in_page.h, in_page.w);
}
```

With this approach we get this results:

Results

Book 1 results

Times (ms)	
Prepare	
Serial	709,29
4 Threads	190,57
16 Threads	47,77
48 Threads	26,35

Efficiencies and speed-ups	
Prepare	
Prepare Efficiency 4 Threads	0,93
Prepare Speed-up 4 Threads	3,72
Prepare Efficiency 16 Threads	0,93
Prepare Speed-up 16 Threads	14,85
Prepare Efficiency 48 Threads	0,56
Prepare Speed-up 48 Threads	26,92

Book 2 results

Times (ms)	
Prepare	
Serial	709,29
4 Threads	190,57
16 Threads	47,77
48 Threads	26,35

Efficiencies and speed-ups	
Prepare	
Prepare Efficiency 4 Threads	0,93
Prepare Speed-up 4 Threads	3,72

Prepare Efficiency 16 Threads	0,93
Prepare Speed-up 16 Threads	14,85
Prepare Efficiency 48 Threads	0,56
Prepare Speed-up 48 Threads	26,92

Book 3 results

Times (ms)	
Prepare	
Serial	3559,69
4 Threads	975,58
16 Threads	351,6
48 Threads	86,56

Efficiencies and speed-ups	
Prepare	
Prepare Efficiency 4 Threads	0,91
Prepare Speed-up 4 Threads	3,65
Prepare Efficiency 16 Threads	0,63
Prepare Speed-up 16 Threads	10,12
Prepare Efficiency 48 Threads	0,86
Prepare Speed-up 48 Threads	41,12

Conclusions

The time reduction are remarkable, we can see that in the case of book 1, the time goes from 709ms to 26ms, in the book 2, goes from 1296ms to 37ms, and in the book 3, the time required to prepare image, goes from 3559ms to 86ms.

We can see that the program scales fine with 4 and 16 threads, getting speed-ups close to the number of the threads and efficiencies close to one, but with 48 threads it is not so efficient, although the results obtained are also satisfactory.

Final conclusions

In the technical part of the final conclusions, we can conclude that the theoretical laws are rarely satisfied. But the results obtained are similar to the expected values, so the improvement is remarkable comparing with the initial values. Is difficult to get speed-ups close to number of threads, and efficiencies close to 1.

We learn that the hypothesis done initially are not always satisfied, for example, in the encryption problem initially we think that static scheduling was the best option, but after testing all option, we get the best results with the guided scheduling option. For complete datasheet of the data recovered see Appendix A.

In the personal part of the final conclusions, we learn a lot about image processing, data encryption and parallel computing. Now, we know the power of the parallelization and how a program that requires a lot of time, can be executed in a much less time using multiprocessor programming.

Bibliography

- Image Filtering (Filters.pdf) - https://egela1819.ehu.eus/mod/resource/view.php?id=967161
- Image Filtering (ILWIS.pdf) - https://egela1819.ehu.eus/mod/resource/view.php?id=967162
- Encryption Algorithms (Hill.pdf) - https://egela1819.ehu.eus/mod/resource/view.php?id=967163
- OpenMP Scheduling Techniques - http://jakascorner.com/blog/2016/06/omp-for-scheduling.html
- OpenMP Loop Scheduling - https://software.intel.com/en-us/articles/openmp-loop-scheduling

Appendices

- Appendix A: all the data obtained is in the Results.pdf document.