

Paralelización do cálculo da convolución dunha imaxe de niveis de gris

NICOLÁS VILELA PÉREZ

Fundamentos de Sistemas Paralelos

nicolas.vilela@rai.usc.es

17 de decembro do 2022

Resumo

Realízase a convolución dunha imaxe de niveis de gris de forma paralela no supercomputador Finisterre III do Centro de Supercomputación de Galicia (CESGA). As distintas gráficas permiten ver o rendemento e escalabilidade da paralelización desta convolución, mostrando que non é tan próximo ao ideal debido a que a porcentaxe de paralelización do programa non é do 100 % e a que hai que elixir ben o número de columnas por bloque empregado.

Palabras clave: convolución de imaxes, procesado de imaxes, imaxes de niveis de gris, MPI, supercomputadores, paralelización, master-slave, rendemento, speedups, escalabilidade, lei de Amdahl

I. INTRODUCIÓN

Este informe contempla o cálculo da convolución dunha imaxe de niveis de gris cun kernel 3x3 de forma paralela.

Dado que a convolución dunha imaxe se basea en operacións concretas sobre cada pixel, este é un traballo altamente paralelizable entre múltiples fíos ou procesos, polo cal se empregarán o estándar de paso de mensaxes MPI no supercomputador Finisterrate III do CESGA para a implementación e execución.

Desta forma realizarase unha análise de escalabilidade do problema en termos do número de procesos usados e do tamaño do problema.

O código e scripts empregados están dispoñibles adxuntos xunto a este informe.

II. CONVOLUCIÓN DUNHA IMAXE

Unha convolución dunha imaxe é o proceso de agregar a cada píxel os seus veciños locais, ponderados por un núcleo de convolución ou *kernel*. Os valores dos píxeles da imaxe de saída son o resultado de calcular a convolución de cada píxel da imaxe de entrada. Isto pode ser descrito mediante un algoritmo, tal e como se mostra no seguinte pseudocódigo:

```

for each image row in input image:
    for each pixel in image row:

        set accumulator to zero

        for each kernel row in kernel:
            for each element in kernel row:

                if element position corresponding* to pixel position then
                    multiply element value corresponding* to pixel value
                    add result to accumulator
                endif

        set output image pixel to accumulator

```

*corresponding input image pixels are found relative to the kernel's origin.

Figura 1: Pseudocódigo da convolución dunha imaxe extraído de [1]

Para entender mellor como ten lugar unha convolución dunha imaxe pódese acceder á web interactiva de [2].

Na web interactiva citada pódese ver como nos píxeles que bordean a imaxe o kernel non se usa na súa totalidade, senón que só se usan aquelas posiciones do kernel que se sitúan dentro da imaxe. Esta vai ser a filosofía que se vai seguir na implementación do programa do presente informe. Adxúntase a continuación unha imaxe para clarificar isto último:

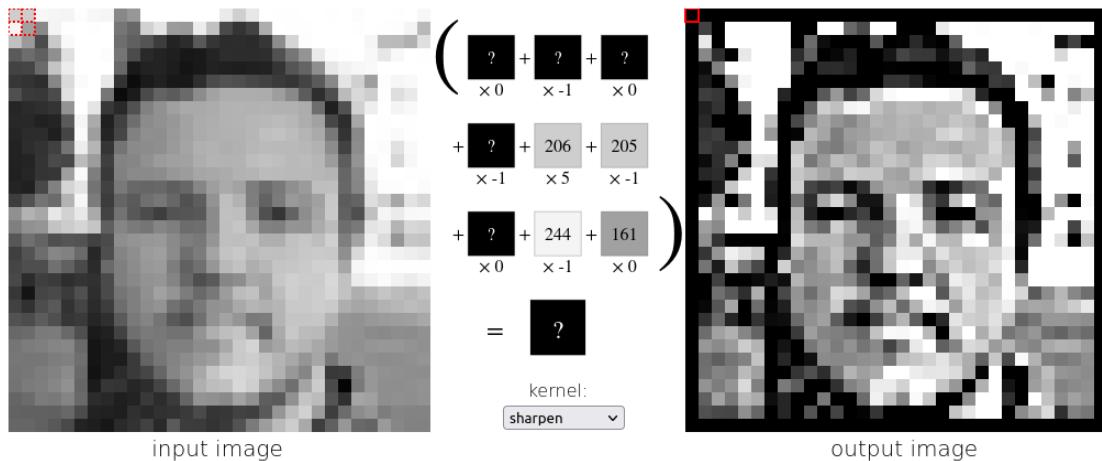


Figura 2: Convolución do píxel da esquina superior esquerda extraída de [2]

Para a elaboración deste informe empregouse un kernel de tipo *sharpen*, cuxo filtro mostra as formas que hai nunha imaxe. O kernel en cuestión é o seguinte:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 6 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Así pois, móstrase a continuación a imaxe de entrada usada e a imaxe de saída correspondente aplicando o kernel mencionado á convolución:



(a) Imaxe de entrada

(b) Imaxe de saída

Figura 3: Imaxes de entrada e de saída do programa

III. ESTUDO PREVIO

A. Arquitectura empregada

O entorno no que se van realizar as probas será un *Thin node* do Finisterrae III, que conta coas seguintes características:

- Procesador: 2x Intel Xeon Ice Lake 8352Y de 32 núcleos de 2,2 GHz
- Memoria RAM: 256 GB
- GPU: non ten

B. Paralelización da convolución

Para paralelizar o cálculo da convolución realizarase unha **distribución cíclica en bloques de C columnas** da matriz de píxeles da imaxe, de forma que o proceso 0 calculará a convolución das primeiras C columnas; o proceso 1, das seguintes C columnas, e así con todos os procesos da comunicación. Unha vez se mandara un bloque de C columnas a cada un dos procesos da comunicación, vólvese empezar polo proceso 0 coas seguintes C columnas que correspondan.

Para calcular a convolución da imaxe seguindo este tipo de distribución hai que ter en conta que os píxeles da primeira e última columna do bloque necesitan información da columna anterior e posterior, respectivamente. Por esta razón decidiuse que en cada envío dun bloque

de C columnas se enviarán dúas columnas extras, que son a anterior á primeira columna do bloque e a posterior á última columna do bloque. Desta forma, as C columnas centrais das $C + 2$ totais enviadas xa teñen toda a información necesaria para realizar a convolución correctamente.

C. Lectura e escritura de imaxes en C

Para a lectura e escritura de imaxes en C empregáronse dúas librarías das dispoñibles en [3], implementadas por Sean T. Barrett.

As librarías que se empregaron máis concretamente son a de cargar e a de escribir imaxes, correspondentes aos arquivos `stb_image.h` e `stb_image_write.h`, respectivamente.

Para ler unha imaxe emprégase a función `stbi_load`, a cal devolve un vector de `unsigned char`, correspondente a todos os píxeles da imaxe, de forma que recorrelos é moi sinxelo.

Para escribir unha imaxe nun ficheiro emprégase a función `stbi_write_EXT`, sendo *EXT* a extensión desexada. Neste caso empregouse `stbi_write_jpg`, polo cal todas as imaxes de saída terán formato JPG.

IV. PREPARACIÓN DO PROGRAMA

A organización das comunicacións será de tipo *master-slave*. O proceso *master* será aquel que teña `myrank=0` e encargarse de:

1. Ler a imaxe de entrada.
2. Enviar os bloques de columnas correspondentes a cada un dos outros procesos da comunicación (*slaves*).
3. Enviar aos procesos *slave* o kernel e a información necesaria para unha correcta execución.
4. Realizar a convolución dos bloques de columnas que lle correspondan segundo a distribución cíclica.
5. Recibir os bloques de columnas dos procesos *slave* tras aplicarles a convolución.
6. Construír a imaxe de saída.

Mentres tanto, os procesos *slave* encargaránse de realizar a convolución dos bloques de columnas que vaian recibindo do proceso *master* e envialos de novo a devandito proceso.

A estratexia empregada para a elaboración do programa é SPMD (*Single Program Multiple Data*), xa que todos os procesos executarán o mesmo código, que é a convolución dos píxeles, sobre datos diferentes, que son os diferentes bloques de columnas enviados polo proceso *master*.

Para o estudo da escalabilidade do cálculo da convolución partírase dun único proceso na comunicación, e incrementarase o número de procesos en cada execución en 1, chegando a un máximo de 32. Para cada número de procesos usaránse 4 valores distintos para o número de columnas C de cada un dos bloques a procesar da distribución exposta no apartado B da páxina 3. Estos valores elixíronse co obxectivo de ter diferentes magnitudes e obter así unha análise dos resultados más completa, e son 1, 15, 55 e 105.

De cada execución realizaranse varias iteracións co obxectivo de ter distintos valores de referencia para así evitar situacíons atípicas que alteren os resultados e a súa posterior análise.

Cabe destacar que para o reparto de bloques de columnas entre os distintos procesos e a posterior unión podería terse empregado o par de funcións MPI_Scatter - MPI_Gather, mais optouse por realizar estas repartición e unión de forma manual, xa que MPI_Scatter envía partes troceadas a todos os procesos da comunicación, incluíndo o propio proceso *master* [4]. Así pois, ao realizar a repartición de forma manual evítanse esas mensaxes extras que son realmente autoenvíos, diminuindo así o número de mensaxes enviadas a través da comunicación e evitando a carga computacional a maiores que estas supoñen. A maiores disto, os bloques de columnas non son continuos realmente, se non que se intersecan debido á primeira e última columnas de cada bloque (que conteñen información estática para que a convolución das C columnas centrais sexa correcta), de forma que estas funcións tampouco funcionarían correctamente.

De cada execución mediranse os seguintes tempos:

- **Tempo dos envíos**, identificado na saída do programa como Sends. Tempo que tarda o proceso *master* en enviar os bloques de columnas correspondentes aos procesos *slave*.
- **Tempo de broadcasts**, identificado na saída do programa como Bcasts. Tempo que se tardan en realizar os 4 broadcasts.
- **Tempo de convolucións**, identificado na saída do programa como Process: X; Convolutions. Tempo que tarda o proceso X en realizar a recepción, convolución e envío de todos bloques que lle correspondan. Cabe destacar que no caso do proceso *master* non hai nin recepcións nin envíos.
- **Tempo de recepcións e asignacións**, identificado na saída do programa como Recvs. Tempo que tarda o proceso *master* en recibir os bloques de columnas dos procesos *slave* e asignar os seus valores á imaxe de saída.

Cabe destacar que para as medicións de tempos considerouse que o *overhead* da función para medir o tempo empregada é nulo, xa que só aproximadamente o 6 % das chamadas tardan 1 ms e as ordes de magnitud de tempo tratadas son superiores, polo cal non se tivo en conta. Este cálculo realiza mediante un pequeno programa que obtén o valor do *overhead* e o imprime por pantalla. Devandito programa pódese repetir tantas veces como se deseñe inseríndoo nun bucle:

```
1 int main(int argc, char *argv[]) {
2     struct timeval t1, t2;
3     gettimeofday(&t1, NULL);
4     gettimeofday(&t2, NULL);
5     printf("tiempo: %lf\n", (t2.tv_sec - t1.tv_sec + (t2.tv_usec - t1.tv_usec) / 1.
e6));
6     exit(EXIT_SUCCESS);
7 }
```

A partir dos tempos medidos explicados anteriormente vaise obter un **tempo de referencia para cada número de procesos**, que será o que se usará para a análise dos resultados. Este tempo será o resultado de coller o peor (máximo) tempo de convolucións de entre todos os procesos para cada iteración, e, de entre todas as iteracións, coller o mellor (mínimo) dos peores tempos anteriores. Unha vez elixido este tempo, sumaráselles os outros 3 tempos asociados á iteración á que corresponde (sempre e cando os procesos implicados sexan máis ca un, se non

non habería paso de mensaxes).

Antes de mostrar as diferentes partes da execución do programa cabe comentar que non se mostrarán as liñas de código correspondentes á convolución¹, senón que no seu lugar mostrarase o comentario /* CONVOLUTION CODE */. Isto é debido a que esa parte é moi extensa como consecuencia das diferentes condicións que existen, xa que non é mesmo convolucionar o primeiro ou último píxel dunha columna que os píxeles centrais, e tampouco o é convolucionar a primeira ou última columna da imaxe que as columnas centrais, tal e como se pode ver reflexado na figura 2, e máis a fondo na web interactiva [2].

A execución do programa consta de varias partes ben diferenciadas:

1. Antes que nada hai que definir os valores das variables globais que se van empregar durante a execución do programa.

```
1 #define MASTER_RANK 0
2
3 #define MAX_FILENAME_LENGTH 50
4 #define MAX_COLUMN_LENGTH 10000
5
6 #define K_ROWS 3
7 #define K_COLUMNS 3
```

2. Na primeira parte da execución inicialízase a comunicación MPI, compróbanse os argumentos pasados por liña de comandos e resérvase memoria para o kernel da convolución, o cal usarán todos os procesos nalgún momento.

```
1 // Initialization of MPI communication
2 MPI_Init(&argc, &argv);
3 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
4 MPI_Comm_size(MPI_COMM_WORLD, &mpi_comm_size);
5
6
7 // Check the arguments
8 f_out = NULL;
9 switch (argc) {
10     case 2:
11         C = atoi(argv[1]);
12         f_in = "images/input.jpg";
13         f_out = (char *) malloc(MAX_FILENAME_LENGTH * sizeof(char));
14         if (f_out == NULL) {
15             printf("ERROR: unable to allocate memory for the output image
filename.\n\n");
16             exit(EXIT_FAILURE);
17         }
18         sprintf(f_out, "images/output_%d_%d.jpg", C, mpi_comm_size);
19         break;
20     case 3:
21         C = atoi(argv[1]);
22         f_in = argv[2];
23         f_out = (char *) malloc(MAX_FILENAME_LENGTH * sizeof(char));
24         if (f_out == NULL) {
25             printf("ERROR: unable to allocate memory for the output image
filename.\n\n");
26             exit(EXIT_FAILURE);
27         }
28         sprintf(f_out, "images/output_%d_%d.jpg", C, mpi_comm_size);
29         break;
30     case 4:
31         C = atoi(argv[1]);
```

¹Se se desexa ver o código íntegro, este está dispoñible no arquivo adxunto a este informe parallel_conv.c.

```

32     f_in = argv[2];
33     f_out = argv[3];
34     break;
35   default:
36     printf("ERROR: incorrect number of parameters.\n\tUse ./program C ["
37     INPUT_IMAGE_FILENAME] [OUTPUT_IMAGE_FILENAME].\n\n");
38     exit(EXIT_FAILURE);
39 }
40
41 // Allocate a contiguous block of memory for the kernel
42 float (*kernel)[K_COLUMNS] = malloc(K_ROWS * sizeof(*kernel));
43 if (kernel == NULL) {
44   printf("ERROR: unable to allocate memory for the kernel.\n\n");
45   exit(EXIT_FAILURE);
46 }
```

3. A continuación o proceso *master* realiza varias tarefas: le a imaxe de entrada, reserva memoria para a imaxe de saída, define o kernel convolucional e o tipo de dato de MPI para o bloque de columnas a través de `MPI_Type_vector` [5], determina o número total de bloques que ten a imaxe e o número de ciclos que ten que facer cada proceso cos seus respectivos restos, e envía os bloques correspondentes aos pertinentes procesos *slave*. Sobre esta última parte de envíos mídese o tempo usado e imprímese.

```

1 if (myrank == MASTER_RANK) {
2   printf("Init\n");
3
4   // Load the input image
5   img_in = stbi_load(f_in, &img_width, &img_height, NULL, 1);
6   if (img_in == NULL) {
7     printf("ERROR: cannot loading the image.\n\n");
8     exit(EXIT_FAILURE);
9   }
10  img_size = img_width * img_height;
11
12  // Allocate memory for the output image
13  img_out = (unsigned char *) malloc(img_size * sizeof(unsigned char));
14  if (img_out == NULL) {
15    printf("ERROR: unable to allocate memory for the output image.\n\n");
16    exit(EXIT_FAILURE);
17  }
18
19
20  // Define the convolution kernel
21  //      [0, -1, 0]
22  //      [-1, 6, -1]
23  //      [0, -1, 0]
24  kernel[0][0] = 0;
25  kernel[0][1] = -1;
26  kernel[0][2] = 0;
27  kernel[1][0] = -1;
28  kernel[1][1] = 6;
29  kernel[1][2] = -1;
30  kernel[2][0] = 0;
31  kernel[2][1] = -1;
32  kernel[2][2] = 0;
33
34
35  // Define the MPI Datatype
36  //      C + 2 because of the convolution of the first and last columns of
37  //      the block
38  MPI_Type_vector(img_height, C + 2, img_width, MPI_UNSIGNED_CHAR, &
block_type);
  MPI_Type_commit(&block_type);
```

```

39 // Determine the number of blocks to scatter the image columns and the
40 // rest
41 total_blocks = img_width / C;
42 rest_columns = img_width % C;
43
44 // Determine the number of cycles that each process have to do and the
45 // rest
46 num_cycles = total_blocks / mpi_comm_size;
47 rest_processes = total_blocks % mpi_comm_size;
48
49 // Send the corresponding blocks to each process
50 if (mpi_comm_size > 1) {
51     memset(&t1, 0, sizeof(struct timeval));
52     memset(&t2, 0, sizeof(struct timeval));
53     gettimeofday(&t1, NULL);
54
55     processed_blocks = 1;
56     num_proc = 1;
57     while (processed_blocks < total_blocks) {
58         // Subtract 1 from the offset to have the information from the
59         // column before the first column of the block
60         MPI_Send(img_in + processed_blocks * C - 1, 1, block_type,
61         num_proc, 0, MPI_COMM_WORLD);
62         processed_blocks++;
63         num_proc++;
64         num_proc %= mpi_comm_size;
65         // If the block is the last one, the num_proc must not be
66         // modified
67         if (num_proc == 0 && processed_blocks < total_blocks) {
68             num_proc++;
69             processed_blocks++;
70         }
71
72         // Extra block
73         // If the master process doesn't have to convolve it, it will be send
74         if (rest_columns > 0 && num_proc != MASTER_RANK) {
75             MPI_Type_free(&block_type);
76             // rest_columns + 1 because of the first column of the last block
77             // of the image
78             MPI_Type_vector(img_height, rest_columns + 1, img_width,
79             MPI_UNSIGNED_CHAR, &block_type);
80             MPI_Type_commit(&block_type);
81             MPI_Send(img_in + total_blocks * C - 1, 1, block_type, num_proc,
82             0, MPI_COMM_WORLD);
83         }
84     }
85
86     MPI_Type_free(&block_type);
87
88     gettimeofday(&t2, NULL);
89     printf("Sends: %lf\n", (t2.tv_sec - t1.tv_sec + (t2.tv_usec - t1.
90     tv_usec) / 1.e6));
91 }

```

4. Unha vez remata o proceso *master* as tarefas do punto anterior, realizanse 4 broadcasts para compartir con todos os procesos *slave* o kernel convolucional, o número de ciclos que ten que facer cada proceso e o resto de devandito número (é dicir, o número de procesos que teñen que facer un ciclo extra), e o resto do número de bloques (é dicir, o número de columnas que terá o bloque extra). Desta parte mídese o tempo empregado e imprímese.

```

1 if (mpi_comm_size > 1) {
2     // Barrier to wait for master process

```

```

3   MPI_Barrier(MPI_COMM_WORLD);
4
5   if (myrank == MASTER_RANK) {
6       memset(&t1, 0, sizeof(struct timeval));
7       memset(&t2, 0, sizeof(struct timeval));
8       gettimeofday(&t1, NULL);
9   }
10
11  MPI_Bcast(&(kernel[0][0]), K_ROWS*K_COLUMNS, MPI_FLOAT, MASTER_RANK,
12  MPI_COMM_WORLD);
13  MPI_Bcast(&num_cycles, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
14  MPI_Bcast(&rest_processes, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
15  MPI_Bcast(&rest_columns, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
16
17  if (myrank == MASTER_RANK) {
18      gettimeofday(&t2, NULL);
19      printf("Bcasts: %lf\n", (t2.tv_sec - t1.tv_sec + (t2.tv_usec - t1.
20      tv_usec) / 1.e6));
21  }

```

5. Agora a execución toma dous camiños diferentes segundo o proceso sexa o *master* ou un *slave*. Como xa se dixo anteriormente, o código correspondente á convolución propiamente dita non se vai mostrar, senón que se mostrará o comentario /* CONVOLUTION CODE */ no seu lugar para simplificar. Móstranse a continuación os códigos de ambos tipos de proceso:

- Proceso *master*: realiza a convolución local (sen recepcións nin envíos MPI), a recepción dos bloques dos procesos *slave* e a asignación dos valores recibidos de devanditos bloques á imaxe de saída. Mídese e imprímese o tempo empleado tanto para a convolución como para a recepción e asignación.

```

1  if (myrank == MASTER_RANK) {
2      memset(&t1, 0, sizeof(struct timeval));
3      memset(&t2, 0, sizeof(struct timeval));
4      gettimeofday(&t1, NULL);
5
6      // CONVOLUTION OF THE MASTER PROCESS
7      // First block of the image
8      // First column of the first block of the image
9      /* CONVOLUTION CODE */
10
11     // Middle blocks
12     processed_blocks = mpi_comm_size;
13     while (processed_blocks < total_blocks) {
14         offset = processed_blocks * C;
15         /* CONVOLUTION CODE */
16         processed_blocks += mpi_comm_size;
17     }
18
19     // Last block of the image
20     if (processed_blocks == total_blocks && rest_columns > 0) {
21         offset = total_blocks * C;
22         /* CONVOLUTION CODE */
23     }
24
25     gettimeofday(&t2, NULL);
26     printf("Process: %d; Convolutions: %lf\n", myrank, (t2.tv_sec - t1.
27     tv_sec + (t2.tv_usec - t1.tv_usec) / 1.e6));
28
29     // Reception of the blocks from the slave processes and construction
      of the output image

```

```

30     if (mpi_comm_size > 1) {
31         memset(&t1, 0, sizeof(struct timeval));
32         memset(&t2, 0, sizeof(struct timeval));
33         gettimeofday(&t1, NULL);
34
35         block_size = (C + 2) * img_height;
36         recv_block = (unsigned char *) malloc(block_size * sizeof(
37             unsigned char));
38
39         for (i = 1; i < mpi_comm_size; i++) {
40             for (j = 0; j < num_cycles; j++) {
41                 offset = (i + j * mpi_comm_size) * C;
42                 MPI_Recv(recv_block, block_size, MPI_UNSIGNED_CHAR, i, j,
43                         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
44                 for (k = 0; k < C; k++) {
45                     for (p_in = recv_block + k + 1, p_out = img_out +
46                         offset + k;
47                         p_in != recv_block + block_size + k + 1;
48                         p_in += C + 2, p_out += img_width) {
49                             *p_out = *p_in;
50                         }
51                     }
52                 }
53
54             // Extra cycle
55             if (rest_processes > 1) {
56                 for (i = 1; i < rest_processes; i++) {
57                     offset = (i + num_cycles * mpi_comm_size) * C;
58                     MPI_Recv(recv_block, block_size, MPI_UNSIGNED_CHAR, i,
59                         num_cycles, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
60                     for (j = 0; j < C; j++) {
61                         for (p_in = recv_block + j + 1, p_out = img_out +
62                             offset + j;
63                             p_in != recv_block + block_size + j + 1;
64                             p_in += C + 2, p_out += img_width) {
65                                 *p_out = *p_in;
66                             }
67                         }
68                     }
69
70                 // Process with the last block of the image
71                 if (rest_columns > 0) {
72                     MPI_Recv(recv_block, block_size, MPI_UNSIGNED_CHAR,
73                         rest_processes, num_cycles, MPI_COMM_WORLD, &status);
74                     MPI_Get_count(&status, MPI_UNSIGNED_CHAR, &block_size);
75                     offset = (rest_processes + num_cycles * mpi_comm_size) *
76                         C;
77                     for (i = 0; i < rest_columns; i++) {
78                         for (p_in = recv_block + i + 1, p_out = img_out +
79                             offset + i;
80                             p_in != recv_block + block_size + i + 1;
81                             p_in += rest_columns + 1, p_out += img_width) {
82                                 *p_out = *p_in;
83                             }
84                         }
85                     }
86                     free(recv_block);
87
88                     gettimeofday(&t2, NULL);
89                     printf("Recvs: %lf\n", (t2.tv_sec - t1.tv_sec + (t2.tv_usec -
90                         t1.tv_usec) / 1.e6));
91                 }
92             }
93         }
94     }

```

```

86     stbi_write_jpg(f_out, img_width, img_height, 1, img_out, 100);
87
88     stbi_image_free(img_in);
89     stbi_image_free(img_out);
90 }
```

- Procesos *slave*: recíbense os bloques de columnas correspondentes, realiza-se a convolución e envíanse os bloques resultantes ao proceso *master* de volta. Desta parte mídese o tempo empregado e imprímese por pantalla.

```

1 else {
2     memset(&t1, 0, sizeof(struct timeval));
3     memset(&t2, 0, sizeof(struct timeval));
4     gettimeofday(&t1, NULL);
5
6     recv_block = (unsigned char *) malloc((C + 2) * MAX_COLUMN_LENGTH *
7         sizeof(unsigned char));
8     if (recv_block == NULL) {
9         printf("ERROR: unable to allocate memory for recv_block.\n\n");
10        exit(EXIT_FAILURE);
11    }
12
13     MPI_Recv(recv_block, (C + 2) * MAX_COLUMN_LENGTH, MPI_UNSIGNED_CHAR,
14             MASTER_RANK, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
15     MPI_Get_count(&status, MPI_UNSIGNED_CHAR, &block_size);
16     block_column_length = block_size / (C + 2);
17     recv_block = (unsigned char *) realloc(recv_block, block_size *
18         sizeof(unsigned char));
19     if (recv_block == NULL) {
20         printf("ERROR: unable to reallocate memory for recv_block.\n\n");
21         ;
22         exit(EXIT_FAILURE);
23    }
24
25     send_block = (unsigned char *) malloc(block_size * sizeof(unsigned
26         char));
27     if (send_block == NULL) {
28         printf("ERROR: unable to allocate memory for send_block.\n\n");
29         exit(EXIT_FAILURE);
30    }
31
32 // Aux columns
33 for (p_in = recv_block, p_out = send_block;
34      p_in != recv_block + block_size;
35      p_in += C + 2, p_out += C + 2) {
36     *p_out = *p_in;
37 }
38
39 /* CONVOLUTION CODE */
40 MPI_Send(send_block, block_size, MPI_UNSIGNED_CHAR, MASTER_RANK, 0,
41 MPI_COMM_WORLD);
42
43 for (i = 1; i < num_cycles; i++) {
44     MPI_Recv(recv_block, block_size, MPI_UNSIGNED_CHAR, MASTER_RANK,
45             MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
46     /* CONVOLUTION CODE */
47     MPI_Send(send_block, block_size, MPI_UNSIGNED_CHAR, MASTER_RANK,
48             i, MPI_COMM_WORLD);
49 }
```

```

48 // If the current process is one of those that have to do an extra
49 cycle
50 if (rest_processes > 1 && myrank < rest_processes) {
51     MPI_Recv(recv_block, block_size, MPI_UNSIGNED_CHAR, MASTER_RANK,
52     MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
53     /* CONVOLUTION CODE */
54     MPI_Send(send_block, block_size, MPI_UNSIGNED_CHAR, MASTER_RANK,
55     num_cycles, MPI_COMM_WORLD);
56 }
57
58 // If the current process is the one that have to convolve the last
59 block of the image
60 if (rest_columns > 0 && myrank == rest_processes) {
61     MPI_Recv(recv_block, block_size, MPI_UNSIGNED_CHAR, MASTER_RANK,
62     MPI_ANY_TAG, MPI_COMM_WORLD, &status);
63     MPI_Get_count(&status, MPI_UNSIGNED_CHAR, &block_size);
64     rest_columns = block_size / block_column_length;
65
66     // Convolution of the last block of the image
67     /* CONVOLUTION CODE */
68
69     MPI_Send(send_block, block_size, MPI_UNSIGNED_CHAR, MASTER_RANK,
70     num_cycles, MPI_COMM_WORLD);
71 }
72 }
```

6. Para finalizar a execución, libéranse as memorias globais empregadas, finalízase a comunicación MPI e sáese.

```

1 free(kernel);
2 if (f_out != NULL) {
3     free(f_out);
4 }
5
6 // Termination of MPI communication
7 MPI_Finalize();
8
9 exit(EXIT_SUCCESS);
```

A. Scripts auxiliares empregados

Para realizar as execucións de forma automatizada escribiríronse dous scripts en Bash:

- **global_execution.sh**

1. Cárganse os módulos MPI.
 module load intel impi
2. Compíllase o código.
 mpicc -Wall -o parallel_conv parallel_conv.c -lm
3. Bórranse os resultados anteriores.
 rm images/output_* slurm_files/* output_files/*
4. Repítense a execución individual de 1 a 32 procesos.
 for ((i = 1; i <= 32; i++))

```

do
    sbatch -n $i -ntasks-per-node $i single_execution.sh $i
done

```

■ **single_execution.sh**

1. Establécense as directivas de execución.

```

#SBATCH -J parallel_convolution (nome do traballo no rexistro do CESGA)
#SBATCH -output=./slurm_files/%x-%j.out (nome do ficheiro Slurm)
#SBATCH -t 10:00 (límite de tempo de 10 minutos)
#SBATCH -mem=32G (límite de memoria RAM de 32 GB)

```

2. Defínense as variables empregadas no script.

```

C_values=(1 15 55 105) (listado dos números de columnas por bloque)
ITER=5 (número de iteracións de cada execución)

```

3. Itérase por un bucle que recorre a lista de todos os valores dos números de columnas por bloque, e executa o programa para cada un deses valores ITER veces. As impresións redirecciónanse a uns ficheiros de saída específicos para postprocesalos despois.

```

for C in $C_values[0]
do
    output=output_files/output_${1}.txt
    echo -e "C: $C\n" > $output
    for ((i = 1; i <= $ITER; i++))
    do
        echo -e "iteration: $i\n" >> $output
        srun parallel_conv $C >> $output
    done
done

```

Para o postprocesado dos datos escribiuse un script en Python, obtendo como resultado diversas gráficas. A execución ten diferentes partes ben diferenciadas:

1. Antes que nada defínense as variables globais usadas no script e compróbase se a carpeta dos gráficos existe, e, se non, créase.

```

1 plots_folder='plots/'
2 num_proc = range(1, 33)
3 C_values = [1, 15, 55, 105]
4 ITER = 5
5 final_times = [[0 for _ in range(len(num_proc))] for _ in range(len(C_values))]
6
7
8 # Check if the plots folder exists; if not, create it
9 if not os.path.exists(plots_folder):
10     os.mkdir(plots_folder)

```

2. A continuación éntrase nun bucle que recorre os 32 ficheiros de saída creados, cuxo nome é da forma output_X.txt, sendo X o número de procesos empregado en dita execución. Para cada ficheiro, éntrase outro bucle que recorre os distintos valores da lista C_values, correspondentes aos números de columnas por bloque. Este bucle interno de execución consta de varias partes:

- a) Primeiramente incialízanse as listas a usar e sáltanse as liñas ata que se atope o valor de C esperado e a primeira iteración de devandito valor.

```

1 # Initialize the lists
2 sends_times = []
3 bcasts_times = []
4 recv_times = []
5 process_convolution_times = []
6 for it in range(ITER):
7     process_convolution_times.append([0]*n)
8
9 # Skip lines until the corresponding value of C is found
10 line = datafile.readline()
11 while line:
12     if line == 'C: ' + str(C_values[i]) + '\n':
13         break
14     line = datafile.readline()
15
16 # Skip lines until the first iteration is found
17 line = datafile.readline()
18 while line:
19     if line == 'iteration: 1\n':
20         break
21     line = datafile.readline()

```

- b) A continuación éntrase nun bucle que recorre cada iteración das iteracóns totais por execución. Neste bucle obtense toda a información necesaria.

```

1 # Read the data of the ITER iterations for more than one process
2 for it in range(ITER):
3
4     # Skip lines until the init of the information
5     line = datafile.readline()
6     while line:
7         if line == 'Init\n':
8             break
9         line = datafile.readline()
10
11
12     # Extract the data
13     if n > 1:
14         # Sends: X
15         line = datafile.readline()
16         sends_times.append(float(line.split(':')[1].strip()))
17
18         # Bcast: X
19         line = datafile.readline()
20         bccasts_times.append(float(line.split(':')[1].strip()))
21
22     # Process X; Convolutions X
23     for j in range(n):
24         line = datafile.readline()
25         info = line.split(';')
26         proc = int(info[0].split(':')[1].strip())
27         process_convolution_times[it][j] = float(info[1].split(':')[1].strip())
28
29     if n > 1:
30         # Recvs: X
31         line = datafile.readline()
32         recv_times.append(float(line.split(':')[1].strip()))
33
34     # Read the line "iteration: X"
35     if it < ITER - 1:
36         datafile.readline()

```

- c) Tras recorrer todas as iteracóns, obtense o mellor tempo de execución de entre todas as iteracóns.

```

1 # Calculate the best time value and add it to the final values
2 max_conv_times = []
3 for j in range(ITER):
4     max_conv_times.append(max(process_convolutions_times[j]))
5 best_time = min(max_conv_times)
6 best_it = max_conv_times.index(best_time)
7 if n > 1:
8     best_time += sends_times[best_it] + bcasts_times[best_it] +
9     recvs_times[best_it]
10
10 final_times[i][n-1] = best_time

```

3. Para finalizar, grafícanse os melhores tempos e os speedups obtidos.

```

1 # Plots
2 # By separate
3 for i in range(len(C_values)):
4     # Time
5     ideal_times = [final_times[i][0]/n for n in num_proc]
6     plt.plot(num_proc, ideal_times, color='b', linestyle='dotted', label='Tempo ideal')
7     plt.plot(num_proc, final_times[i], color='r', label='Tempo real')
8     plt.xlabel('Número de procesos')
9     plt.ylabel('Tempo de ejecución (s)')
10    plt.title('Tempo de ejecución con bloques de tamaño ' + str(C_values[i]))
11    plt.legend()
12    plt.savefig(plots_folder + 'time_' + str(C_values[i]) + '.png')
13    plt.clf()
14
15    # Speedup
16    ideal_speedups = num_proc
17    real_speedups = [final_times[i][0]/j for j in final_times[i]]
18    plt.plot(num_proc, ideal_speedups, color='b', linestyle='dotted', label='Speedup ideal')
19    plt.plot(num_proc, real_speedups, color='r', label='Speedup real')
20    plt.xlabel('Número de procesos')
21    plt.ylabel('Speedup')
22    plt.title('Speedup con bloques de tamaño ' + str(C_values[i]))
23    plt.legend()
24    plt.savefig(plots_folder + 'speedup_' + str(C_values[i]) + '.png')
25    plt.clf()
26
27    # All together
28    # Time
29    for i in range(len(C_values)):
30        plt.plot(num_proc, final_times[i], label='C = ' + str(C_values[i]))
31    plt.xlabel('Número de procesos')
32    plt.ylabel('Tempo de ejecución (s)')
33    plt.title('Tempo de ejecución con bloques de tamaño C')
34    plt.legend()
35    plt.savefig(plots_folder + 'time_together.png')
36    plt.clf()
37
38    # Speedup
39    for i in range(len(C_values)):
40        speedups = [final_times[i][0]/j for j in final_times[i]]
41        plt.plot(num_proc, speedups, label='C = ' + str(C_values[i]))
42    plt.xlabel('Número de procesos')
43    plt.ylabel('Speedup')
44    plt.title('Speedup con bloques de tamaño C')
45    plt.legend()
46    plt.savefig(plots_folder + 'speedup_together.png')
47    plt.clf()

```

V. RESULTADOS

A continuación realizarase un estudo do tempo de execución e *speedup* en función do número de procesos e do valor de C empregados na convolución paralela.

A. Tempo de ejecución

Neste apartado estudarase o desenvolvemento do tempo de execución a medida que aumentan os procesos involucrados e varía o tamaño do bloco de columnas, comparándoo co tempo ideal esperado, onde o tempo debería ser a metade cada vez que se duplica o número de procesos, é dicir, para:

- n = número de procesos.
- s_n = velocidad obtida con n procesos
- s_1 = velocidad obtida na versión secuencial

O comportamento ideal é:

$$s_n = \frac{s_1}{n}$$

Observando os resultados pódese ver que a tendencia é semellante á ideal, mais o tempo real estáncase nun límite maior a devandito tempo ideal:

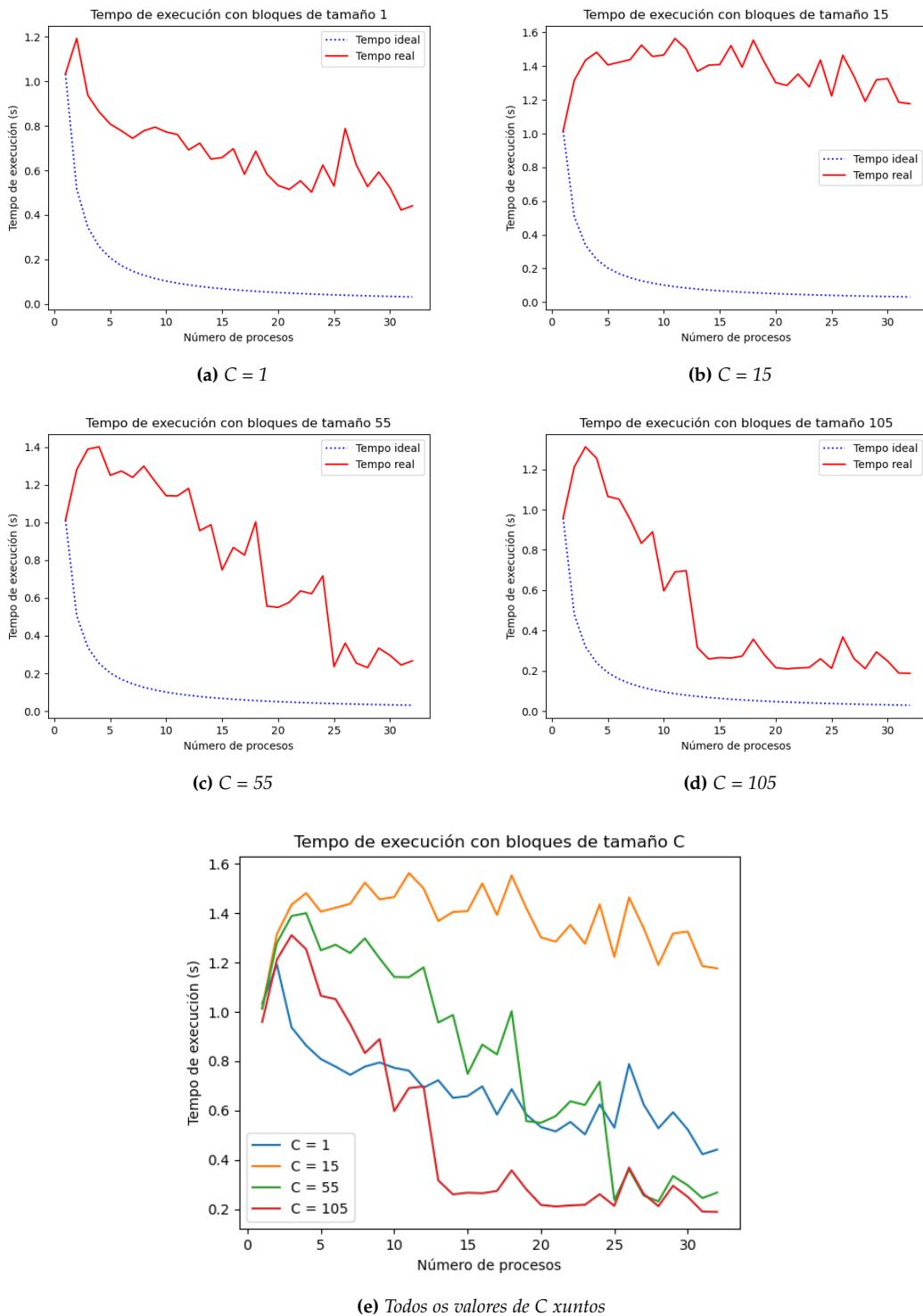


Figura 4: Tempos de execución

Por un lado, pódese observar como o tempo está bastante lonxe do ideal esperado. Desta forma, **o programa non é moi escalable en canto ao número de procesos se refire**, xa que o tempo estabilízase a partir dun número de procesos non moi alto. Isto pódese ver claramente

na figura 4d, onde a última gran diferenza temporal ten lugar entre os 12 e 13 procesos, estabilizándose o tempo despois arredor dos 0,2 segundos a pesar de estar involucrados máis procesos.

A maiores do anterior, tamén se pode observar como para valores de C superiores á decena inicialmente o tempo dispárase, sendo ata un 40 % peor que a versión secuencial (1 proceso). Estes dous comportamentos son debido ao seguinte:

- Ao incrementar o número de procesos involucrados na convolución paralela, o tempo que tarda cada proceso en facer as súas convolucións é menor, xa que a cada proceso tócanlle menos bloques. Non obstante, o proceso *master* segue tardando o mesmo en separar a imaxe, enviar os bloques, facer os broadcasts, recibir os bloques resultantes e construír a imaxe de saída. Desta forma, o programa non é paralelizable ao 100 %, polo cal será imposible lograr unha aproximación ao tempo ideal, estancándose nun tempo superior debido ao coste de execución do proceso *master*.
- Para poucos procesos involucrados, a redución temporal gañada no tempo que tarda cada proceso en facer as súas convolucións vese eclipsada polo tempo que tarda o proceso *master* en realizar as tarefas mencionadas no punto anterior. Desta forma, o cómputo total do mellor tempo é superior ao obtido para a versión secuencial, a cal carece de separación da imaxe, envío de bloques e broadcasts e recepción de bloques.

Por outro lado, pódese observar na figura 4e como segundo se aumenta o tamaño do bloque de columnas, os tempos obtidos son menores. Desta forma, **o programa é escalable en canto ao tamaño do problema se refire**, xa que o tempo vese considerablemente reducido segundo se vai aumentando C.

Cómpre comentar que inicialmente o tempo para $C = 1$ é mellor que para o resto de valores. Isto é debido a que a separación, envío, recepción e construcción son moito más veloces ao tratarse de bloques moi pequenos.

Tamén se pode observar que para $C = 15$, o tempo non se dá reducido o suficiente como para chegar a ser nalgún momento inferior que para $C = 1$. Isto é debido a que o bloque non é nin tan pequeno como para facer que a separación, envío, recepción e construcción sexan moi veloces, nin tan grande como para facer que o tempo que tarda cada proceso en facer as súas convolucións compense. Desta forma, queda un tempo superior debido a unha mala elección do tamaño do bloque de columnas, facendo ver que este valor ten que ser elixido con cautela.

B. Speedup

Neste apartado estudarase o desenvolvemento do speedup a medida que aumentan os procesos involucrados e varía o tamaño do bloque de columnas, comparándoo co speedup ideal esperado, onde este debería aumentar da mesma forma que aumenta o número de procesos (n), sendo ambos valores iguais. O speedup ideal é así soamente se se asume que o programa é paralelizable ao 100 %, quedando a lei de Amdahl da seguinte forma:

$$\text{Speedup} \leq n$$

Non obstante, o proceso *master* ten que realizar tarefas que soamente pode realizar el, polo cal o programa non é paralelizable ao 100 %, así que o speedup sempre será menor que o número de procesos, nunca igual.

Móstranse a continuación os valores de speedup obtidos xunto ao seu valor ideal supoñendo que o 100 % do programa é paralelizable:

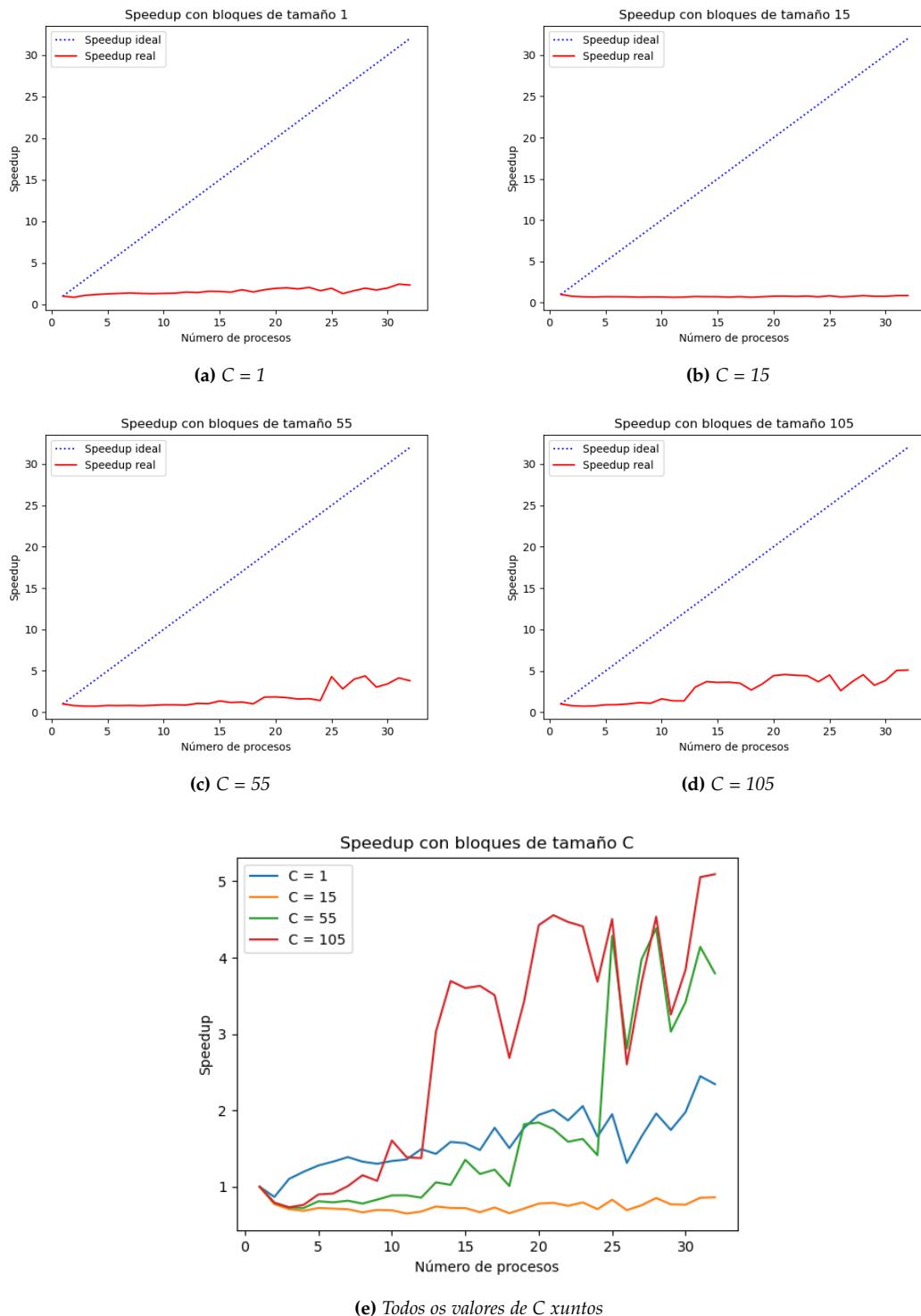


Figura 5: Speedups suponendo que o programa é paralelizable ao 100 %

Por un lado, pódese observar na figura 5e como os speedups seguen a mellora esperada a partir dos tempos observados na figura 4e, sendo a execución con menor speedup a de $C = 15$; e, a de

maior speedup, a de $C = 105$.

Por outro lado, pódese observar nas figuras 5a, 5b, 5c e 5d como o speedup real está moi lonxe do ideal. Como xa se explicou na páxina 18, isto é debido a que o programa non é paralelizable no seu 100 %, senón que a súa porcentaxe de paralelización é menor.

A partir destes valores de speedup pódese facer unha estimación da porcentaxe do programa que é paralelizable: para $C = 105$ (figura 5d), con 31 procesos pódese ver que o speedup está en torno a 5, polo cal se pode empregar a lei de Amdahl para obter a fracción paralelizable. Inicialmente:

$$\text{Speedup} \leq \frac{1}{\frac{p}{n} + (1 - p)}$$

onde:

- n = número de procesos.
- p = fracción paralelizable

Con $\text{Speedup} = 5$ e $n = 31$ e igualando a inecuación tense o seguinte:

$$5 = \frac{1}{\frac{p}{31} + (1 - p)} \rightarrow p \simeq 0,83$$

Desta forma, pódese dicir que **a porcentaxe do programa que é paralelizable é aproximadamente do 84 %**. Cabe destacar que se lle suma un 1 % á porcentaxe obtida para así obter unha marxe debido a posibles erros de precisión do cálculo.

Así pois, pódense obter unhas novas gráficas para os speedups adaptando o valor ideal ao 84 % de fracción paralelizable. Para isto hai que cambiar unha liña do script de Python na zona do código na que se grafican os speedups explicada na páxina 15. Concretamente hai que cambiar a liña:

```
ideal_speedups = num_proc  
por esta outra:  
ideal_speedups = [1/((0.84/n) + 0.16) for n in num_proc]
```

Tras a modificación do script obtéñense as seguintes gráficas:

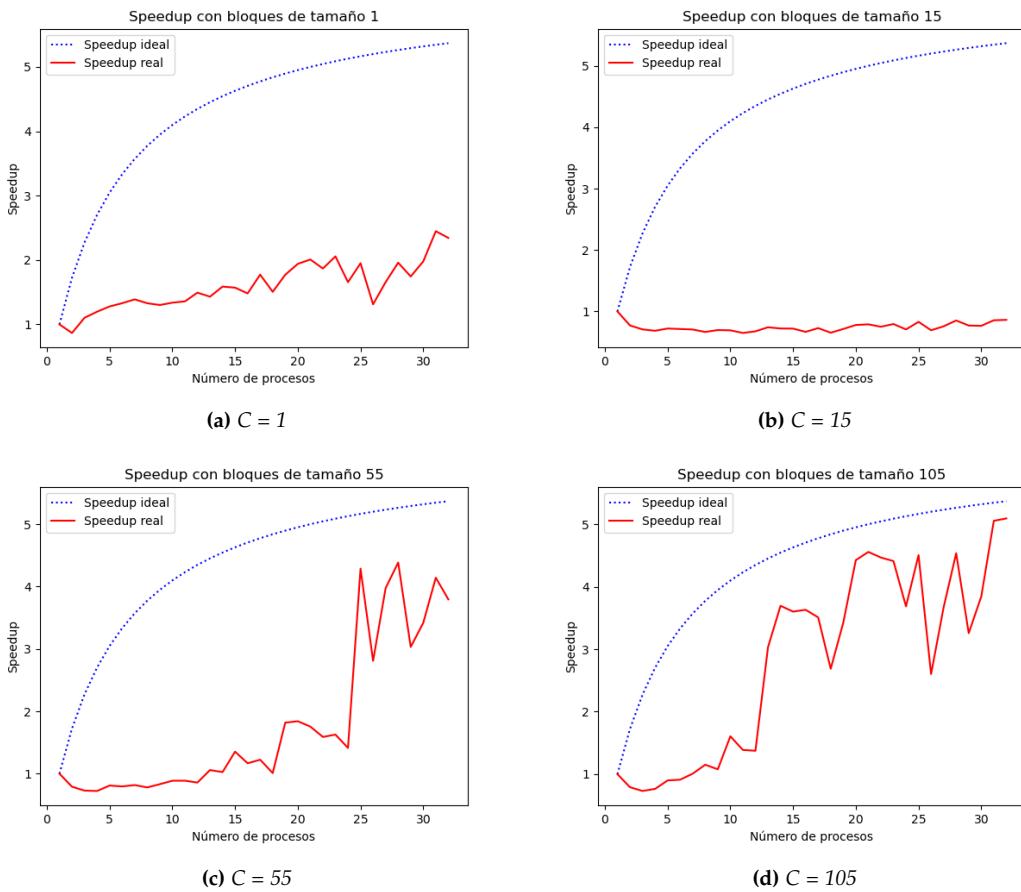


Figura 6: Speedups axustando a porcentaxe paralelizable do programa ao 84 %

Desta forma pódese observar nas figuras 6a, 6b, 6c e 6d que, en contraposición coas figuras 5a, 5b, 5c e 5d, o speedup ideal está máis preto ao valor real. Isto é debido a que nas figuras 5a, 5b, 5c e 5d estábbase supoñendo un speedup ideal erróneo, xa que o programa non era paralelizable ao 100 %, facendo así que o speedup ideal das figuras 6a, 6b, 6c e 6d sexa moito más realista.

VI. CONCLUSIÓNS

A convolución dunha imaxe de niveis de gris é paralelizable, mais non ao 100 %, e así o mostran as gráficas expostas ao longo deste informe, cunha estabilización temporal temperá en canto ao número de procesos se refire. Tamén cómpre remarcar que a eficiencia temporal da paralelización da convolución dunha imaxe depende en boa parte do número de columnas por bloque empregado, pois un tamaño non axeitado pode facer que os tempos se disparen, tal e como pasa con $C = 15$.

En resumo, se se emprega un número de procesos e de columnas por bloque alto, pódense obter bons resultados temporais, mais haberá un momento no que non servirá de nada seguir incrementando o número de procesos, desperdiциando así recursos.

Este estudo podería ser ampliado por diferentes vías:

- Executando o programa noutros sistemas con máis recursos para ver que sucede cun maior número de procesos.
- Empregando imaxes de entrada cun número de columnas moi alto, dando lugar así a analizar novos valores de C a analizar.
- Implementando o programa de diferentes maneiras para estudar o efecto das diferentes implementacións.

REFERENCIAS

- [1] Colaboradores de Wikipedia. Kernel (image processing) [en liña]. *Wikipedia, The Free Encyclopedia*, 2022 [data de consulta: 12 decembro 2022]. Dispoñible en: [https://en.wikipedia.org/wiki/Kernel_\(image_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing))
- [2] Victor Powell. *Image Kernels explained visually* [en liña] [data de consulta: 12 decembro 2022]. Dispoñible en: <https://setosa.io/ev/image-kernels/>
- [3] Sean T. Barrett. *stb libraries* [en liña] [data de consulta: 12 decembro 2022]. Dispoñible en: <https://github.com/nothings/stb>
- [4] Universidad de Granada. MPI_Scatter [en liña]. *Web de Programación Paralela*, 2022 [data de consulta: 13 decembro 2022]. Dispoñible en: https://lsi2.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Scatter
- [5] Universidad de Granada. MPI_Type_vector [en liña]. *Web de Programación Paralela*, 2022 [data de consulta: 14 decembro 2022]. Dispoñible en: https://lsi2.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Type_vector