

# Memoria proyecto

## Paralelización de un programa secuencial

Francisco Abel Cedrón Santaeufemia  
Nuria López Rivas

16 de mayo de 2013

### 1. Descripción del problema

Lo que se va a tratar de desarrollar en este trabajo es una manera de escalar una imagen digital. Dicho proceso se hace mediante la siguiente transformación

$$R(x, y) = A \left( \frac{x}{i_x}, \frac{y}{i_y} \right) \quad (1)$$

donde  $A(x, y)$  es la imagen de entrada mientras que  $R(x, y)$  es la salida y donde el valor de  $i_x$  indica el incremento en el eje horizontal e  $i_y$  el incremento en el eje vertical. El problema que puede ocasionar la ecuación 1 es que puede dar valores de números decimales en los índices con lo que intenta referenciar un valor de un pixel que no existe en una imagen. Para poder solucionar este problema se emplea la interpolación. Los métodos más comunes de interpolar una imagen son los siguientes:

1. Interpolación por el vecino más próximo.
2. Interpolación bilineal.
3. Interpolación bicúbica.

La interpolación por el vecino más próximo resuelve el problema cogiendo el valor del pixel más próximo, siendo una manera sencilla de implementar y con una rápida ejecución pero consigue un efecto pixelado en los resultados. Para poder mejorar los resultados obtenidos se puede emplear la interpolación bilineal que para obtener el valor de un pixel que no existe usa los cuatro reales que están a su alrededor y calcula una media ponderada consiguiendo así un mejor resultado visual en la imagen de salida, pero con un tiempo de ejecución mucho mayor. Si aún así se necesita unos mejores resultados se puede emplear una interpolación bicubica que requiere el uso de 16 pixels. El resultado que se consigue con la interpolación bicubica es eliminar los efectos “cuadrículados” que se obtienen empleando con la interpolación bilineal, efecto que se consigue al emplear 16 pixels en vez de 4 al calcular el valor de un pixel resultante (a pesar del gran coste computacional que eso conlleva). Para poder ver el resultado obtenido con los distintos tipos de interpolación se puede ver el resultado en la figura 1.

El algoritmo que se desarrollará en este trabajo es el de la interpolación bicubica, que como hemos dicho emplea dos interpolaciones cúbicas. En el caso de una sola dimensión la interpolación cúbica consiste en trazar una cúbica entre los 4 puntos más próximos (2 a la izquierda y 2 a la derecha).

$$f(x) = c_0x^3 + c_1x^2 + c_2x + c_3 \quad (2)$$

para aplicar la interpolación bicubica se necesita hacer dos interpolaciones cúbicas:

1. Interpolación cúbica *horizontal*, en las filas existentes (usando 4 puntos).
2. Interpolación cúbica *vertical* en todo el espacio usando 4 puntos (empleando la anterior interpolación).

En este tipo de interpolación un punto  $(p_x, p_y)$  emplea los 16 pixels circundantes. Así tenemos que el valor del punto se puede calcular como una media ponderada de los 4x4 pixels circundantes. Para calcular el punto interpolado hay que realizar

$$A'(p_x, p_y) = \sum_{n=-1,2} \sum_{m=-1,2} A(i+n, j+m)P(n-a)P(b-m) \quad (3)$$

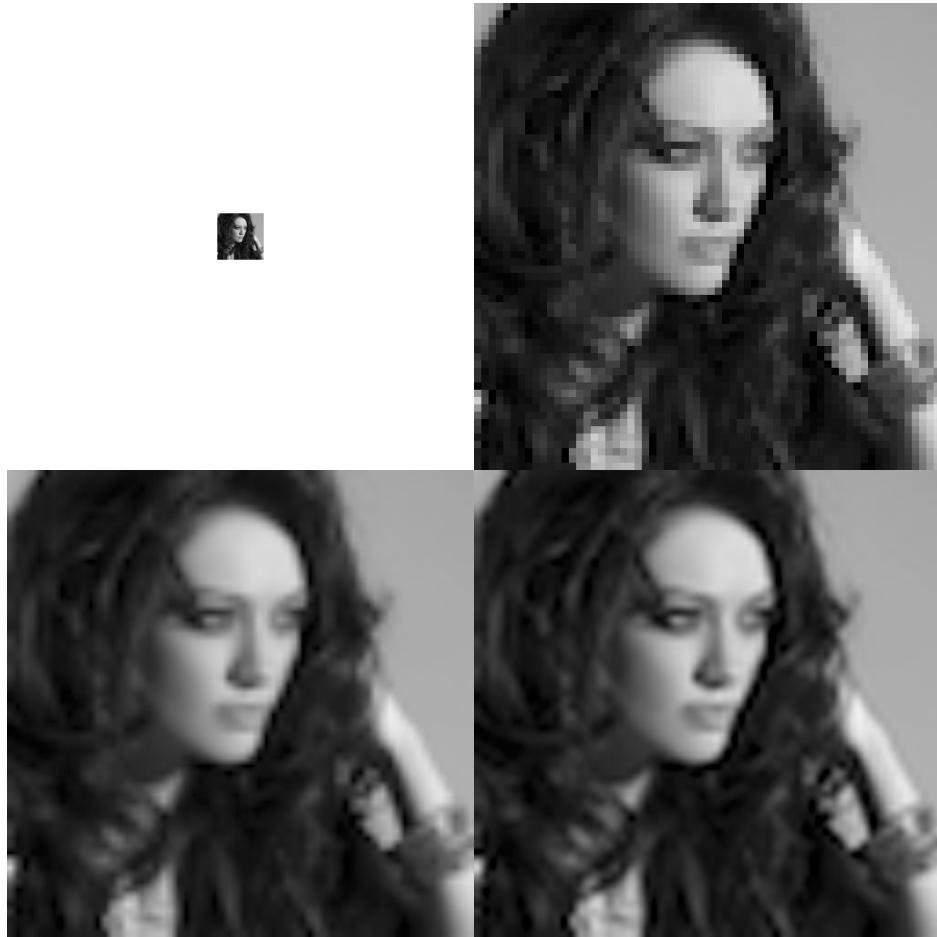


Figura 1: Resultados obtenidos al emplear distintas interpolaciones. De izquierda a derecha y de arriba a abajo: a) imagen original, b) zoom 10x interpolación por vecino más próximo, c) zoom 10x interpolación bilineal, d) zoom 10x interpolación bicúbica

siendo

$$P(k) = \frac{1}{6}(C(k+2)^3 - 4C(k+1)^3 + 6C(k)^3 - 4C(k-1)^3)C(k) = \max\{0, k\}$$

Un problema que hay que abordar en cuanto al uso del tipo de interpolación es saber qué pixels se van emplear en el caso de que se necesite calcular valores que estén próximos al borde, para ello se pueden usar dos métodos

1. Usar ceros como valor siempre que se necesite un valor que este fuera de la matriz de la imagen.
2. Usar un efecto espejo en los bordes de la imagen.

En este trabajo se implementará la segunda opción.

## 2. Ejemplo práctico: Aumento de una imagen

### 2.1. Creación del programa secuencial

Lo primero que se necesita es poder simular una imagen. Para ello se realiza una simulación tal y como se muestra en la figura 3

Al utilizar el método de escalado de una imagen descrito anteriormente nos encontraríamos con un problema al intentar hallar el valor de un pixel en los bordes de la imagen. Para solucionar este problema se llevó a cabo una de las soluciones encontradas en la literatura, se emplea el método de creación de bordes alrededor de la imagen con efecto espejo, añadiendo dos filas en la parte superior e inferior y también dos columnas a la izquierda y derecha de la imagen. En la figura 4 se muestra un ejemplo representativo de la matriz original y la aumentada.

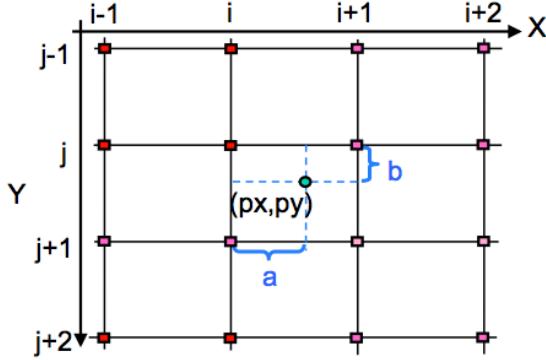


Figura 2: Parámetros empleados para realizar la interpolación bicubica

```

1 void getImage(short *matrix, int height, int width) {
2     int i, j;
3
4     for (i = 0; i < height; i++)
5         for (j = 0; j < width; j++)
6             matrix INDEX(i, j, width) = rand() % 256; //gray level
7 }
```

Figura 3: Creación de la imagen original.

Una vez que tenemos la imagen original y se han creado los “bordes”, se realiza el proceso de escalado con la función que aparece en la figura 5.

## 2.2. Técnicas de paralelización utilizadas

Como se dijo anteriormente, el escalado de una imagen es el tipo de tarea adecuada para realizar con programación paralela. En este caso ésto se llevó a cabo mediante MPI, OpenMP y SSE, técnicas aplicadas en el punto donde se concentra toda la carga computacional del programa, esto es, en el cálculo para generar la imagen resultante. En la Figura 5 se puede ver el segmento de código secuencial perteneciente a este punto.

### 2.2.1. Paso de mensajes con MPI

Esta técnica se aplicó al bucle más externo de los mostrados en la Figura 5. Con ella se pretendía dividir la carga computacional que crea este segmento de código entre varios procesadores, de manera que cada uno de ellos realizase una parte del cálculo, siendo así más rápida la terminación del programa. El motivo de realizar este reparto es dividir las filas de la imagen original para que cada proceso MPI se encargue de escalar su trozo de imagen tal y como se muestra en la figura 6.

Para poder repartir la matriz de la imagen entre los distintos procesos se utilizaron las funciones *MPI\_Scatterv* y *MPI\_Gatherv*. El motivo de su uso fue que los segmentos de matriz enviados a cada proceso estaban solapados, y estas funciones permiten indicar al padre en qué punto empieza la parte de la matriz que debe enviar a un proceso en concreto, así como el tamaño a enviar. Además este motivo fue la razón de que el reparto de trabajo fuese consecutivo, porque en el caso de ser cíclico, se tendría que enviar toda la imagen a todos los procesos. En la figura 7 se puede ver como lo que se envía a cada proceso MPI está solapado con lo que se envía a otros procesos.

El motivo de tener que enviar a cada proceso partes solapadas de la matriz se explica por el mismo motivo de la creación de la matriz espejo, es decir, la parte que debe calcular cada proceso también tiene pixels en los “bordes”, que necesitarán sus pixels circundantes para calcular su nuevo valor.

Para que se pueda repartir a cada proceso la información necesaria se debe poder indicar las variables necesarias para llamar a las funciones *MPI\_Scatterv* y *MPI\_Gatherv* (la figura 8 muestra dichas funciones), con lo que en la versión de programa paralelo se han creado unas funciones que indican a las funciones de envío y recepción de MPI los detalles necesarios. Esas funciones se pueden ver en la figura 9.

1		236	74	74	236	41	205	186	171	171	186
2		198	103	103	198	105	115	81	255	255	81
3	103	198	105	115	81	255	198	103	103	198	105
4	74	236	41	205	186	171	236	74	74	236	41
5	242	251	227	70	124	194	251	242	242	251	227
6	84	248	27	232	231	141	248	84	84	248	27
7	118	90	46	99	51	159	90	118	118	90	46
8							90	118	118	90	46
9							90	118	118	90	46

Figura 4: Izquierda: Imagen original. Derecha: Imagen espejo.

### 2.2.2. Paralelización multihilo con OpenMP

Este método de paralelización se aplicó sobre el mismo bucle que se paralelizó con MPI. La idea es la misma que lo que se pretendía conseguir con la paralelización hecha con MPI, y es que cada thread se encargue de procesar un conjunto de filas, pero como tenemos un programa híbrido paralelizado con MPI y OpenMP las filas se reparten por los distintos procesos MPI y dentro de cada proceso MPI las filas se reparten entre los distintos threads creados por OpenMP. En la figura 10 se muestra como es este reparto para una configuración de 3 procesos MPI con 2 threads OpenMP.

Para llevar a cabo el proceso de paralelización con OpenMP se dividió el trabajo del bucle usando la cláusula *pragma omp for* y se privatizaron algunas variables para asegurar una ejecución correcta, tal y como se muestra en la figura 11<sup>1</sup>.

### 2.2.3. Extensiones multimedia con SSE

Las extensiones multimedia de SSE permiten que se realicen varias operaciones aritmético-lógicas en la misma etapa del procesador. En este trabajo se emplean los registros multimedia SSE para el proceso de interpolación (necesario para calcular los valores de los píxeles de la imagen resultado). Como se puede ver en la figura 2, en la ecuación 2 y en el código secuencial de la figura 5 se emplean dos bucles que realizan cuatro iteraciones cada uno para realizar el proceso. Los registros flotantes de precisión simple disponibles en SSE nos permite deshacernos de los bucles necesarios para realizar la interpolación al realizar el proceso de *unrolling* (como se puede ver en la figura 12).

En la primera parte del proceso de paralelización usando SSE se obtienen los valores de las funciones  $P(k)$  necesarios y se guardan en los registros  $pn\_1$ ,  $pn0$ ,  $pn1$ ,  $pn2$  y  $pm$ , y como se puede ver en la ecuación 2 se multiplican entre ellos. El siguiente paso es almacenar en los registros de SSE los valores de los pixels necesarios para llevar a cabo la interpolación. Como los pixels son valores enteros, y para hallar el valor de interpolación se necesitan valores reales, primero se añaden a los registros de SSE *int* (mediante la función *\_mm\_setr\_epi32*) y después se convierten en *float* (con la función *\_mm\_cvtepi32\_ps*). El siguiente paso es multiplicar los valores de los pixels por su correspondiente ponderación, que está en los registros  $pn\_1$ ,  $pn0$ ,  $pn1$  y  $pn2$ , teniendo así los resultados de las aportaciones de los valores de cada pixel en los registros  $sum\_1$ ,  $sum0$ ,  $sum1$  y  $sum2$  que, para poder aprovechar las instrucciones de SSE, se suman entre sí, quedando así en un solo registro las sumas de las columnas. El siguiente paso sería extraer los valores de ese registro y sumarlos entre sí, pero se decidió realizar una operación de reducción sobre la suma con los registros SSE utilizando sumas horizontales (teniendo así en el registro  $sum\_1$  la suma total).

Como se puede ver en la figura 12 todas las variables que se emplean durante el proceso de paralelización con los registros multimedia que nos proporciona SSE están privatizadas mediante directivas de OpenMP para cada thread impidiendo que se “machaquen” valores y así evitar provocar *race conditions*.

**Precisión en los registros flotantes de SSE** Uno de los problemas que se ha observado con los registros que proporciona la librería de Intrinsics, es que en el momento de pasar los registros de tipo *int* a tipo *float* mete una imprecisión de 10<sub>-4</sub>. Si nos fijamos en el código el valor que se acaba guardando en la matriz sigue siendo un entero, pero esa imprecisión en ocasiones lleva a provocar que el valor de la intensidad se vea aumentado en una unidad. Esto, en el caso de las imágenes, no supondría un gran problema debido a que el cambio de un pixel no se notaría mucho y el proceso que se hace al escalar una imagen es inventar valores y ese cambio no se notaría demasiado visualmente.

<sup>1</sup> Las variables que aparecen privatizadas son para poder evitar *race conditions*

```

1 int ** getScale(short *mirror, short *result, int height, int width, int delay, float ix,
2   float iy) {
3   //Asignacion de memoria para la imagen resultante
4   int **tmp = malloc(sizeof(int *)*(height*iy));
5
6   if (tmp == NULL)
7     exit_msg("getImage: cannot allocate memory (1)");
8
9   int i, j;
10  for (i = 0; i < height*iy; i++) {
11    if ((tmp[i] = malloc(sizeof(int) * (width*ix))) == NULL)
12      exit_msg("getImage: cannot allocate memory (2)");
13  }
14
15  int cnt = 0;
16  int n, m;
17  float sum;
18  float pn, pm;
19  float a, b;
20  //Iniciar el proceso de convolucion
21  for (i = 0; i < height*iy; i++) {//Bucle a paralelizar con MPI y OMP
22    for (j = 0; j < width*ix; j++) {
23      //tmp[i][j] = MAX(0, cnt++);
24      sum = 0.0f;
25      a = ((float) i)/ix - ((int) i/ix);
26      b = ((float) j)/iy - ((int) j/iy);
27      for (n = -1; n < 3; n++) { //Ambos bucles con SSE
28        for (m = -1; m < 3; m++) {
29          pn = Pk(n - a);
30          pm = Pk(b - m);
31          sum += mirror[(int) (i/ix+delay+n)][(int) (j/iy+delay+m)]*pn*pm;
32        }
33      }
34      tmp[i][j] = (int) sum;
35    }
36  }
37  return tmp;
}

```

Figura 5: Código de secuencial para el escalado de una imagen.

### 3. Performance Benchmark

Para comprobar que realmente la parelización es efectiva en el escalado de la imagen se realizaron varias ejecuciones sobre distintos tamaños de imagen, tanto con el programa secuencial como con la versión paralelizada. Las pruebas de *benchmark* se realizaron en el CESGA (CEntro de Supercomputación de Galicia) en las máquinas de SVG que han sido actualizadas durante el inicio del año 2011 y, tras la ampliación del sistema, cuenta con:

- 46 nodos de computación HP ProLiant SL 165z G7. Cada cual cuenta con:
  - 2 x AMD Opteron Processor 6174 2.2GHz, 12Mb Level 3 Cache, 12 cores por procesador
  - 36 GB (17 nodos) y 64 GB (27 nodos) de memoria RAM x 500 GB 3G SATA 7,2K NHP
- 4 nodos de visualización HP DL 385 G7. Cada cual cuenta con:
  - 2 x AMD Opteron Processor 7174 2.2 GHz, 12Mb Level 3 Cache
  - 64 GB PC3-1333R de memoria RAM
  - 1 x 2TB 3G SATA 7.2K NHP

Esta nueva configuración dota al sistema de 1200 cores a 2.2GHz, una memoria total de 2400 GB, una capacidad de almacenamiento de 31 TB y una *performance* de 10,420 GFlops.

El supercomputador SVG será el encargado de albergar las simulaciones necesarias para llevar a cabo el estudio presente de este trabajo.

El código empleado fue compilado con un archivo *Makefile* cuya estructura se muestra en la figura 13. Además para la ejecución de dicho código en las máquinas se ha creado un script que se encarga de

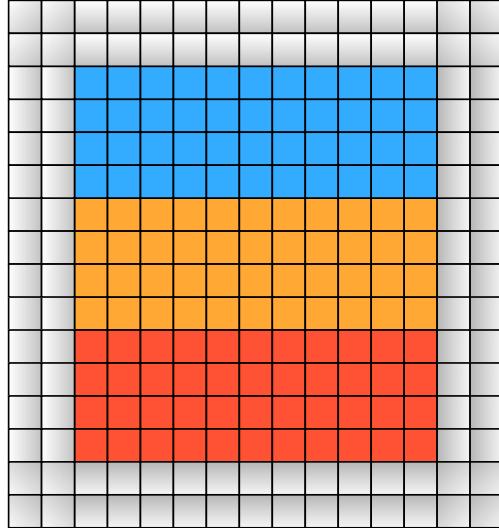


Figura 6: Imagen dividida en filas. La región grisácea representa el espejo que se creó para dicha imagen. La imagen original esta formada por los píxeles de color azul, naranja y rojo. Como se puede visualizar, la imagen está dividida en tres regiones, que son las que cada proceso MPI se encargaría de escalar.

crear los scripts para ejecutar las pruebas y ponerlos en la cola de ejecución. Su estructura puede verse en la figura 14. Para el código secuencial se creó un archivo bash que incluye los comando de compilación y ejecución.

### 3.1. Comparación de resultados

En los siguientes subapartados se mostrarán las configuraciones obtenidas, al emplear una imagen de tamaño  $14000 \times 15000$  como parámetro de entrada y querer obtener una esta misma imagen con un zoom de  $2x$ . Para poder realizar una comparación de tiempos adecuada, cada problema se ejecutó dos veces y se obtuvo la media de los tiempos para utilizarla como dato final.

#### 3.1.1. Aumento de Threads OpenMP o Procesos MPI

En este apartado se mostrarán tablas y resultados que se obtienen al fijar en uno la cantidad de threads OpenMP o procesos MPI para ir variando el otro parámetro. En la tabla 3.1.1 se muestran los resultados obtenidos. En la figura 15 se puede ver una gráfica de como varía el speedup al variar la cantidad de procesos MPI frente a variar la cantidad de threads OpenMP, mientras que en la figura 16 se ve la variación del tiempo necesario de ejecución.

#### 3.1.2. Uso de 24 cores.

El número máximo de cores que permite el CESGA para usar a la vez por cada proceso a un usuario es de 24, es este apartado se verá como se modifica el tiempo y el speedup al usar distintas configuraciones de procesos MPI y threads OpenMP que lleguen a usar los 24 cores. La terminología usada será  $AxB$  donde  $A$  será la cantidad de procesos MPI que se usaron, mientras que  $B$  será la cantidad de threads OpenMP. En la tabla 3.1.2 se puede ver los resultados obtenidos mientras que en las figuras 17 y 18 se puede ver las gráficas de como varía el speedup y el tiempo de ejecución respectivamente.

#### 3.1.3. Pruebas de *benchmark*

En este apartado se mostrarán dos gráficas con las distintas pruebas realizadas en la etapa de comparativa de tiempos. Para poder entender los gráficas es necesario comprender que la fórmula de cores empleados es  $AxB$  donde  $A$  está la cantidad de procesos MPI que se usaron, mientras que  $B$  es la cantidad de threads OpenMP. Para la experimentación se realizaron todas las posibles combinaciones de pruebas con las configuraciones que se pueden realizar en el CESGA empleando desde 1 hasta 24 cores. En las figuras 19 y 20 el eje X indica la cantidad de cores empleados. Como se puede comprobar hay distintas

MPI/OMP	Tiempo		SpeedUp	
	MPI	OpenMP	MPI	OpenMP
1	2112,211	2112,211	1	1
2	1115,826	1182,174	1,89295732488757	1,78671752212449
3	710,983	796,96	2,97083193269037	2,65033502308773
4	532,582	647,445	3,96598270313304	3,26237904377978
5	425,127	578,439	4,96842355343227	3,65157086572655
6	354,616	553,808	5,95633304757822	3,81397704619652
7	314,711	447,934	6,71158936293933	4,71545138346274
8	267,639	428,59	7,89201499034147	4,92827877458643
9	238,114	412,218	8,87058719772882	5,12401447777632
10	213,882	386,608	9,87559027875183	5,46344359144146
11	195,110	369,351	10,8257444518477	5,71870930361634
12	178,421	364,23	11,838354229603	5,79911319770475
13	164,994	363,1	12,8017443058535	5,8171605618287
14	152,912	358,424	13,813245526839	5,89305124656831
15	142,626	357,283	14,8094386717709	5,91187098182673
16	134,227	356,909	15,7361112145842	5,91806594958379
17	125,901	349,614	16,7767611059483	6,04155153969807
18	118,994	353,121	17,7505672554919	5,98155023348937
19	113,37	342,878	18,6311281644174	6,16024066869265
20	107,175	335,332	19,7080569162585	6,29886500542746
21	102,90	333,989	20,5268318756074	6,32419331175578
22	98,103	326,131	21,5305444277953	6,476572297635
23	93,631	314,788	22,5588854118828	6,70994764730549
24	89,829	323,298	23,5136871166327	6,53332529121739

Cuadro 1: Resultados al variar los procesos MPI o los threads OpenMP

MPIxOMP	SEG,MSEG	SpeedUp
1x24	323,298	6,53332529121739
2x12	184,326	11,4591050638543
3x8	142,724	14,799269919565
4x6	118,274	17,858624887972
6x4	103,908	20,3277033529661
8x3	101,955	20,7170908734245
12x2	97,243	21,7209567783799
24x1	89,829	23,5136871166327

Cuadro 2: Resultados con distintas configuraciones que emplean 24 cores.

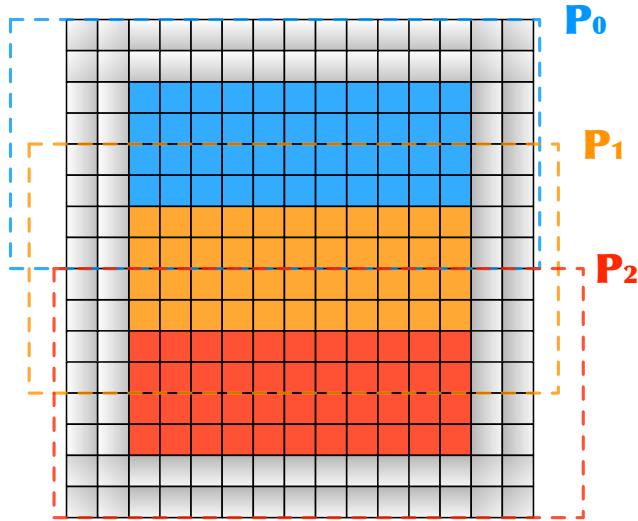


Figura 7: Cada color de la imagen original (azul, naranja o rojo) son las filas que tendrá que escalar cada proceso, pero para realizar el escalado necesita unos “bordes” para poder realizar la interpolación. Lo que se enviará a cada proceso MPI está delimitado con un rectángulo de líneas punteadas (mostrando así que hay trozos de la imagen que puede estar en varios procesos).

```

1 MPI_Scatterv(mirror, size_send_child, aux_split,
2   MPLSHORT, work, size_send_child[myid],
3   MPLSHORT, 0, MPLCOMMWORLD);
4
5 MPI_Gatherv(result, size_send_dad[myid], MPLSHORT,
6   scale, size_send_dad, aux_begin,
7   MPLSHORT, 0, MPLCOMM.WORLD);

```

Figura 8: Funciones *MPIScatterv* y *MPIGatherv*

pruebas para la misma cantidad de cores. Es ahí donde está expresada la cantidad de pruebas que se realizaron para ese número de cores. La manera de saber cuál es cada prueba es mediante la fórmula  $AxB = NumCores$ , donde de izquierda a derecha van aumentando los procesos MPI y disminuyendo los threads OpenMP. Un ejemplo es con 4 cores, donde de izquierda a derecha se usaron las configuraciones:

- *4x1* (4 procesos MPI y un único thread OpenMP),
- *2x2* (2 procesos MPI y 2 threads OpenMP) y
- *1x4* (un único proceso MPI y 4 threads OpenMP).

así podemos ver los distintos resultados viendo la variación del speedup y del tiempo de ejecución en las gráficas que se encuentran en las figuras 19 y 20 respectivamente.

```

1 #define MIN(x,y) ((x<y)?x:y)
2
3 //Función para indicar las filas con las que trabajará cada proceso
4 void fill_aux_rows(int *array, int size, int height, int delay) {
5     int rows = height / size;
6     if (height % size != 0)
7         rows++;
8
9     int i;
10    for (i = 0; i < size; i++)
11        array[i] = MIN(rows, height-i*rows) + 2*delay;
12}
13
14 //Función para conocer el número de elementos a enviar a cada proceso
15 void fill_size_send_child(int *fill, int size, int *rows, int width, int delay) {
16     int i;
17
18     for (i = 0; i < size; i++)
19         fill[i] = rows[i]* (width + 2*delay);
20}
21
22 //Función para conocer desde donde se empieza a enviar a cada proceso
23 void fill_aux_split(int *fill, int size, int stride, int width, int delay) {
24     int i;
25
26     for (i = 0; i < size; i++)
27         fill[i] = i * (stride - (width + delay*2)*4);
28}
29
30 //Función para conocer el tamaño de la respuesta que enviará cada proceso
31 void fill_size_send_dady(int *fill, int size, int *rows, int width, float ix, float iy,
32                         int delay) {
33     int i;
34
35     for (i = 0; i < size; i++)
36         fill[i] = (rows[i] -(2*delay))*iy * (width*ix);
37}
38
39 //Función que indica al proceso que almacena el resultado donde tiene que colocar los
40 // resultados parciales
41 void fill_aux_begin(int *fill, int size, int bytes) {
42     int i;
43
44     for (i = 0; i < size; i++)
45         fill[i] = bytes*i;
46}

```

Figura 9: Funciones para necesarias para llenar las variables que necesitan las variables que se pasan por parámetros a las funciones *MPI-Gatherv* y *MPI-Scatherv*

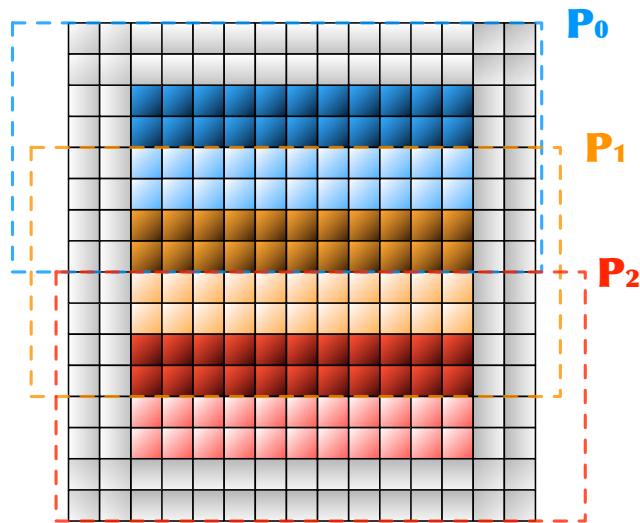


Figura 10: Dentro de cada proceso MPI se reparten consecutivamente las filas. En la imagen podemos ver que dentro de cada grupo de filas por proceso MPI se muestran un grupo oscurecido y otro más claro, indicando así que threads trabajan sobre cada fila.

```

1 #pragma omp parallel private(i, j, sum, a, pn_1, pn0, pn1, pn2, pm, sum_1, sum0, sum1,
2     sum2)
3 {
4     #pragma omp for
5         for (i = 0; i < size; i++) { ... }

```

Figura 11: Cláusulas de OpenMP

```

1 //Get all pn
2 pn_1 = _mm_set1_ps(Pk(-1 - a));
3 pn0 = _mm_set1_ps(Pk(- a));
4 pn1 = _mm_set1_ps(Pk(1 - a));
5 pn2 = _mm_set1_ps(Pk(2 - a));
6 //get all pm
7 pm = _mm_set_ps(Pk(b-2),Pk(b-1),Pk(b),Pk(b+1));
8 //tmp mul pn*pm
9 pn_1 = _mm_mul_ps(pm, pn_1);
10 pn0 = _mm_mul_ps(pm, pn0);
11 pn1 = _mm_mul_ps(pm, pn1);
12 pn2 = _mm_mul_ps(pm, pn2);
13 //get all mirror pos
14 sum_1 = _mm_cvtepi32_ps(_mm_setr_epi32(
15     mirror INDEX ((int) (i/ix+delay-1), (int) (j/iy+delay-1), depth),
16     mirror INDEX ((int) (i/ix+delay-1), (int) (j/iy+delay), depth),
17     mirror INDEX ((int) (i/ix+delay-1), (int) (j/iy+delay+1), depth),
18     mirror INDEX ((int) (i/ix+delay-1), (int) (j/iy+delay+2), depth)));
19 sum0 = _mm_cvtepi32_ps(_mm_setr_epi32(
20     mirror INDEX ((int) (i/ix+delay), (int) (j/iy+delay-1), depth),
21     mirror INDEX ((int) (i/ix+delay), (int) (j/iy+delay), depth),
22     mirror INDEX ((int) (i/ix+delay), (int) (j/iy+delay+1), depth),
23     mirror INDEX ((int) (i/ix+delay), (int) (j/iy+delay+2), depth)));
24 sum1 = _mm_cvtepi32_ps(_mm_setr_epi32(
25     mirror INDEX ((int) (i/ix+delay+1), (int) (j/iy+delay-1), depth),
26     mirror INDEX ((int) (i/ix+delay+1), (int) (j/iy+delay), depth),
27     mirror INDEX ((int) (i/ix+delay+1), (int) (j/iy+delay+1), depth),
28     mirror INDEX ((int) (i/ix+delay+1), (int) (j/iy+delay+2), depth)));
29 sum2 = _mm_cvtepi32_ps(_mm_setr_epi32(
30     mirror INDEX ((int) (i/ix+delay+2), (int) (j/iy+delay-1), depth),
31     mirror INDEX ((int) (i/ix+delay+2), (int) (j/iy+delay), depth),
32     mirror INDEX ((int) (i/ix+delay+2), (int) (j/iy+delay+1), depth),
33     mirror INDEX ((int) (i/ix+delay+2), (int) (j/iy+delay+2), depth)));
34 //get sum for all mirror pos
35 sum_1 = _mm_mul_ps(sum_1, pn_1);
36 sum0 = _mm_mul_ps(sum0, pn0);
37 sum1 = _mm_mul_ps(sum1, pn1);
38 sum2 = _mm_mul_ps(sum2, pn2);
39 //sum all record sse for mirror pos *pn*pm
40 sum_1 = _mm_add_ps(sum_1, sum0);
41 sum1 = _mm_add_ps(sum1, sum2);
42
43 sum_1 = _mm_add_ps(sum_1, sum1);
44 //reduction
45 sum_1 = _mm_hadd_ps(sum_1, sum_1);
46 sum_1 = _mm_hadd_ps(sum_1, sum_1);
47
48 _mm_store_ss(&sum, sum_1);

```

Figura 12: Código paralelizado con SSE

```

1 OPT = -O3 -fopenmp -msse3
2 MPICC      = mpicc
3 CLINKER    = $(CC)
4 CFLAGS     = $(OPT) -Wall -pedantic -std=c99
5
6 MPICXX     = mpicxx
7 CFLAGS     = $(OPT) -Wall -std=c++98 -Wno-long-long
8
9 LFLAGS      = -lm -lx11 -L/usr/X11R6/lib
10
11 ALL = mpi_omp_sse
12 HFILES =
13
14 .PHONY: all
15 all: $(ALL)
16
17 %: %.c $(HFILES)
18   $(MPICC) -o $@ $(CFLAGS) $< $(LFLAGS)
19
20 .PHONY: clean
21 clean:
22   -rm $(ALL)

```

Figura 13: Archivo *Makefile* para la compilación del código.

```

1#!/bin/bash
2MAXPROC=24 #Numero maximo de procesadores que puede usar el CESGA
3MIN_PROC=1 #Numero minimo de procesadores/threads a usar
4MPI="${MIN_PROC} ${MAX_PROC}"
5OMP="${MIN_PROC} ${MAX_PROC}"
6REPEAT=2 #numero de veces que se repite cada problema
7TIME=01:10:00 #tiempo que se solicita para la ejecucion
8MEM=2.5G #memoria que se usara
9ARCH=amd #arquitectura a emplear
10EXEC="./mpi_omp_sse" #programa a ejecutar
11ARGS="14000 15000 2 2" #argumentos del programa
12for i in `seq ${MPI}` ; do
13  for j in `seq ${OMP}` ; do
14    let PROCS=${i}*${j}
15    if `eval test ${PROCS} -le ${MAXPROC}` ; then
16      for r in `seq 1 ${REPEAT}` ; do
17        SH_NAME=ej_it${r}_mpi${i}.omp${j}.sh
18        OUT_NAME=job-it${r}-mpi${i}-omp${j}
19        echo "Make \"${SH_NAME}\"..."
20        echo "#!/bin/bash" > ${SH_NAME}
21        echo "module load mpich2/1.3.2p1-gnu" >> ${SH_NAME}
22        echo "mpirun -np ${i} ${EXEC} ${ARGS}" >> ${SH_NAME}
23        echo "qsub -cwd -l arch=${ARCH},num_proc=${j},s_rt=${TIME},s_vmem=${MEM},h_fsize=4G -pe
24          mpi ${i} -N ${OUT_NAME} ${SH_NAME}"
25        qsub -cwd -l arch=${ARCH},num_proc=${j},s_rt=${TIME},s_vmem=${MEM},
26          h_fsize=4G -pe mpi ${i} -N ${OUT_NAME} ${SH_NAME}
27        rm ${SH_NAME}
28      done
29    fi
30  done
31done

```

Figura 14: Archivo *launch.sh* para la enviar procesos a ejecutar.

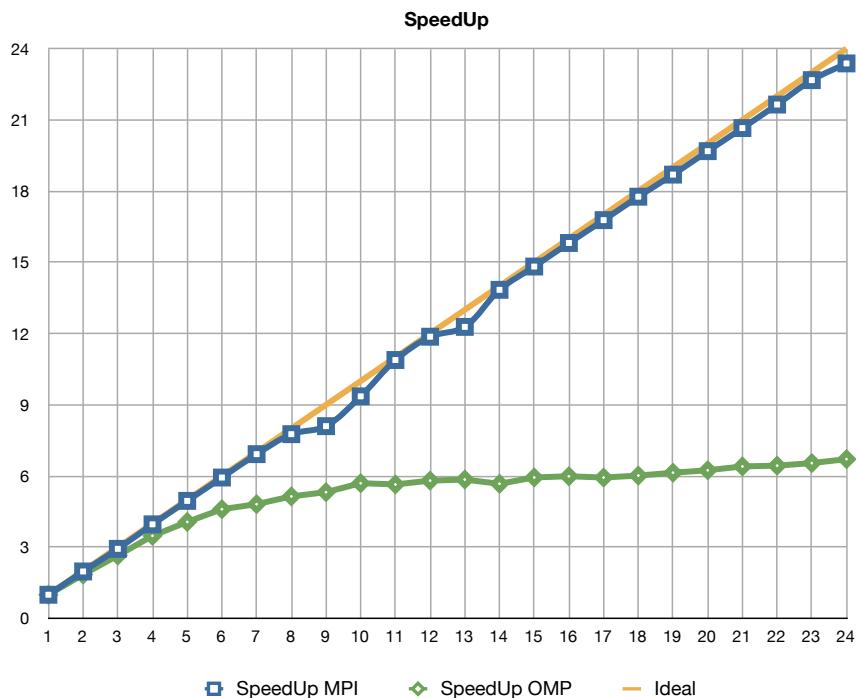


Figura 15: Variación del SpeedUp al variar solo la cantidad de procesos MPI o threads OpenMP.

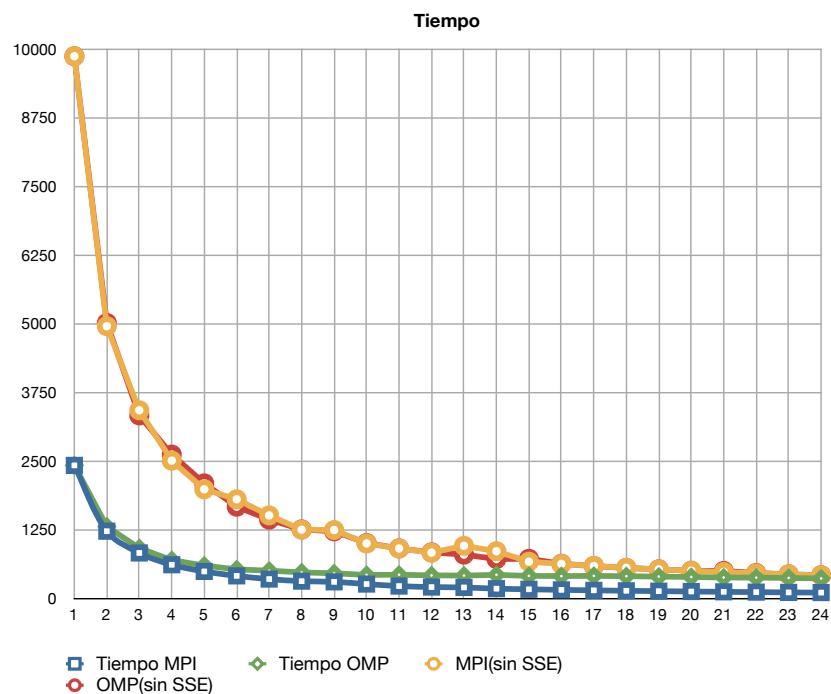


Figura 16: Variación del tiempo de ejecución al variar solo la cantidad de procesos MPI o threads OpenMP.

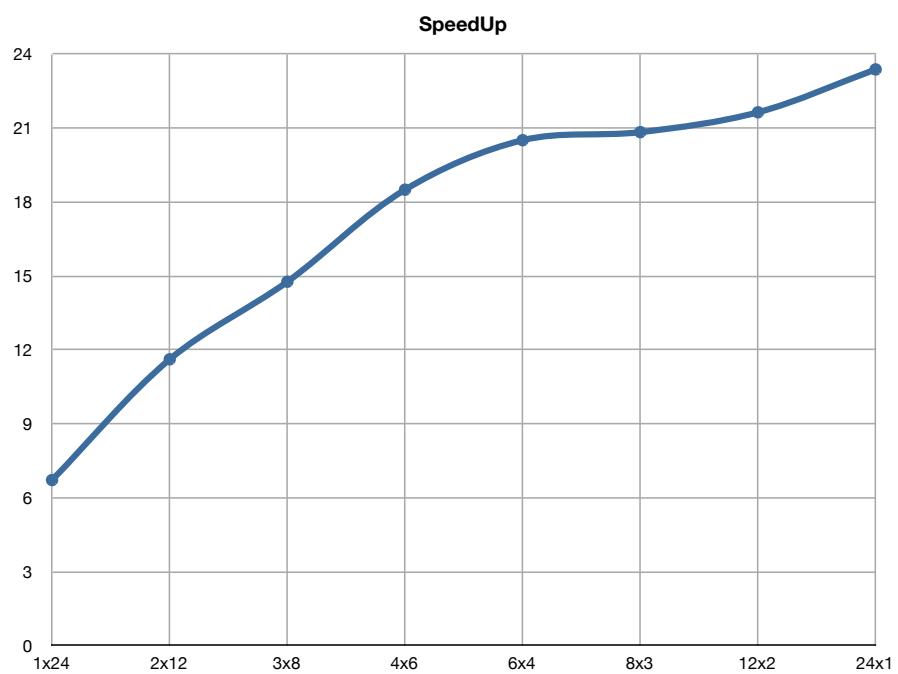


Figura 17: Variación del SpeedUp con distintas configuraciones que emplean 24 cores.

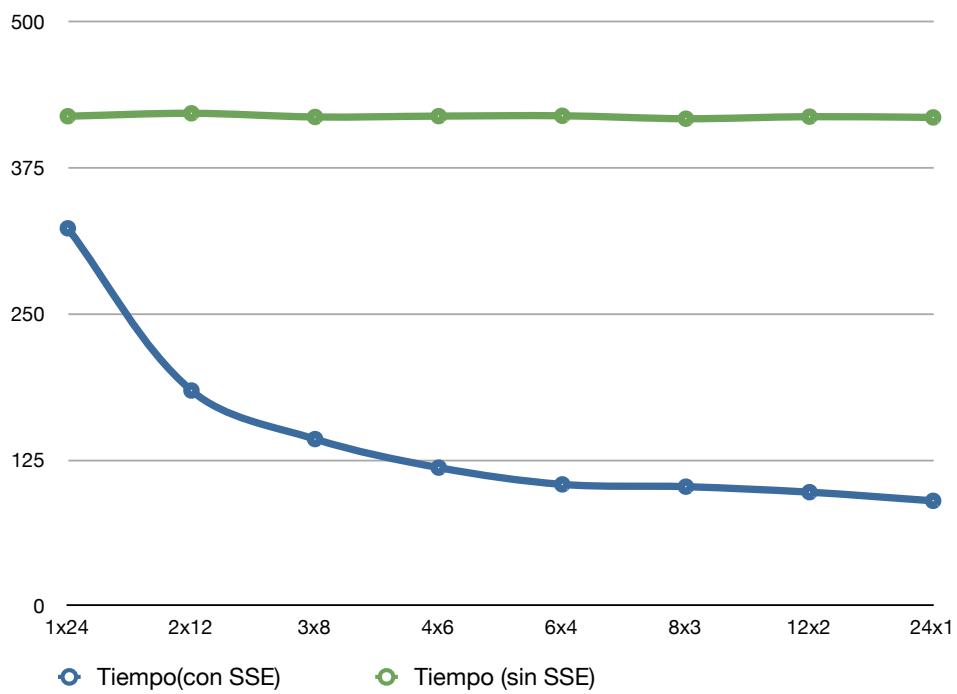


Figura 18: Variación del tiempo de ejecución con distintas configuraciones que emplean 24 cores.

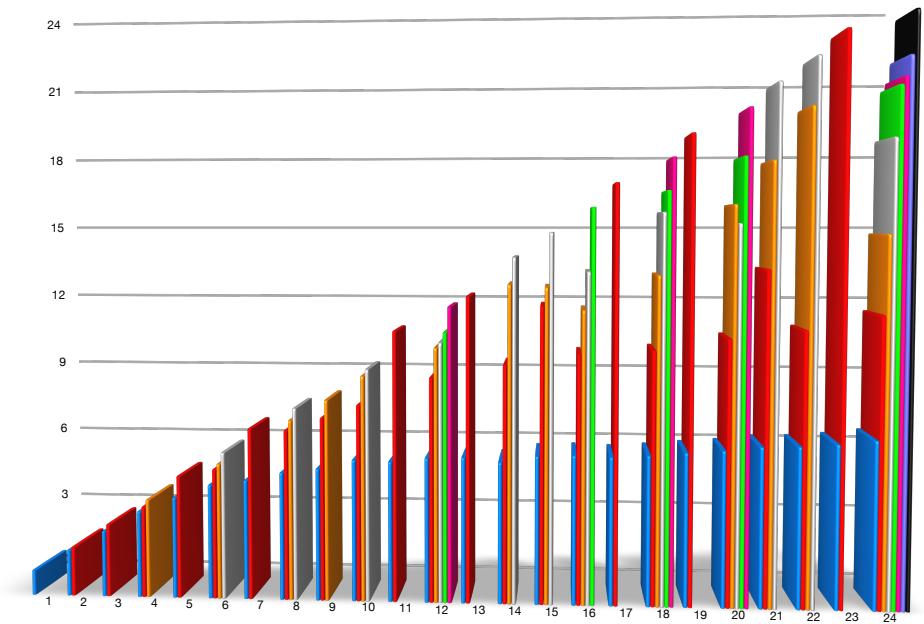


Figura 19: Variación del SpeedUp con distintas configuraciones.

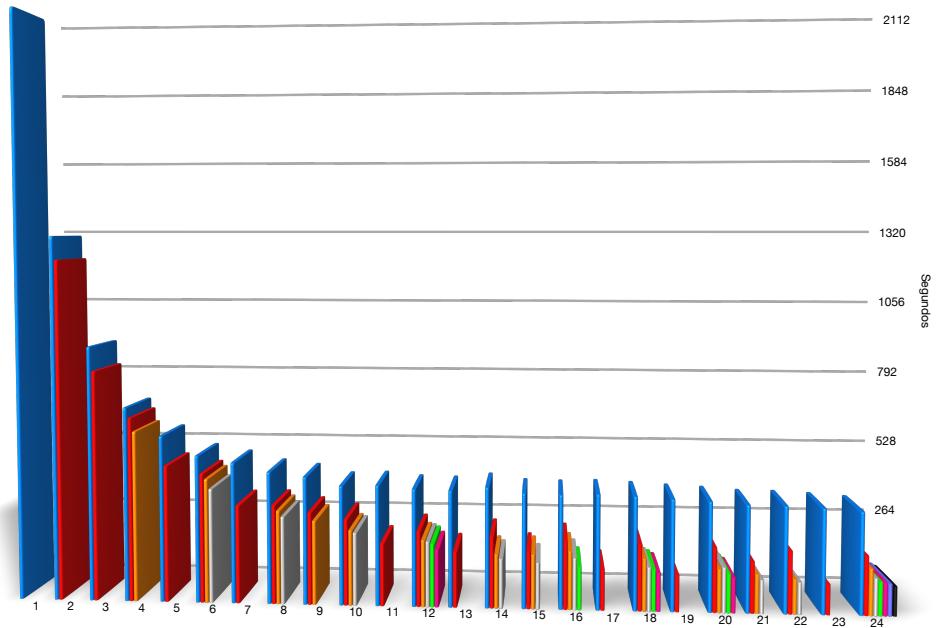


Figura 20: Variación del tiempo de ejecución con distintas configuraciones.

## Referencias

- [1] Rafael C. González and Richard E. Woods *Digital Image Processing* Addison-Wesley, USA 2nd Edition 1993
- [2] Páginas de manual de Apple  
<http://developer.apple.com/library/mac/#documentation/Darwin/Reference/Manpages/index.html>
- [3] Página del supercomputador CESGA  
<http://archivo.cesga.es/content/view/496/1/lang,es/>
- [4] Biblioteca MPI de software libre  
<http://www.mcs.anl.gov/research/projects/mpich2/>
- [5] Biblioteca MPI de software libre  
<http://www.open-mpi.org>
- [6] Intel ®Intrinsic Reference  
<http://software.intel.com/sites/default/files/m/9/4/c/8/e/18072-347603.pdf>