Trabajo Práctico Final

Programación Avanzada

Algoritmos de ordenamiento con OpenMP





|  |  |
| --- | --- |
| **Proyecto:** | Trabajo Práctico Final |
| **Autor:** | Marco Cupo |
| **Versión del documento:** | 1.0 |
| **Fecha de creación:** | 23/05/2016 |

# Control de cambios

|  |  |  |  |
| --- | --- | --- | --- |
| **Fecha** | **Revisión** | **Autor** | **Observaciones** |
| 23/05/2016 | 1.0 | *Marco* | Primera versión del documento |

# Índice

[Control de cambios 2](#_Toc453509404)

[Índice 3](#_Toc453509405)

[Introducción 5](#_Toc453509406)

[Plataforma 6](#_Toc453509407)

[Desarrollo del programa 7](#_Toc453509408)

[Algoritmos de ordenamiento 8](#_Toc453509409)

[Burbujeo 9](#_Toc453509410)

[Versión lineal 9](#_Toc453509411)

[Versión paralela 10](#_Toc453509412)

[Burbujeo bidireccional (cocktail sort) 11](#_Toc453509413)

[Versión lineal 11](#_Toc453509414)

[Versión paralela 12](#_Toc453509415)

[Shell Sort 13](#_Toc453509416)

[Versión lineal 13](#_Toc453509417)

[Versión paralela 14](#_Toc453509418)

[InsertionSort 15](#_Toc453509419)

[Versión lineal 15](#_Toc453509420)

[Versión paralela 16](#_Toc453509421)

[Resultados 17](#_Toc453509422)

[Primera ejecución – 8 núcleos con 8 threads 18](#_Toc453509423)

[Cuadro comparativo 18](#_Toc453509424)

[Gráfico de uso de CPU 18](#_Toc453509425)

[Segunda ejecución – 8 núcleos con 4 threads 19](#_Toc453509426)

[Cuadro comparativo 19](#_Toc453509427)

[Gráfico de uso de CPU 19](#_Toc453509428)

[Tercera ejecución – 8 núcleos con 12 threads 20](#_Toc453509429)

[Cuadro comparativo 20](#_Toc453509430)

[Gráfico de uso de CPU 20](#_Toc453509431)

[Cuarta ejecución – 8 núcleos con 50 threads 21](#_Toc453509432)

[Cuadro comparativo 21](#_Toc453509433)

[Gráfico de uso de CPU 21](#_Toc453509434)

[Quinta ejecución – 2 núcleos con 2 threads 22](#_Toc453509435)

[Cuadro comparativo 22](#_Toc453509436)

[Gráfico de uso de CPU 22](#_Toc453509437)

[Conclusiones 23](#_Toc453509438)

[Puntos de vista 24](#_Toc453509439)

[El programa completo 24](#_Toc453509440)

[Por algoritmo 25](#_Toc453509441)

[Bibliografía 26](#_Toc453509442)

# Introducción

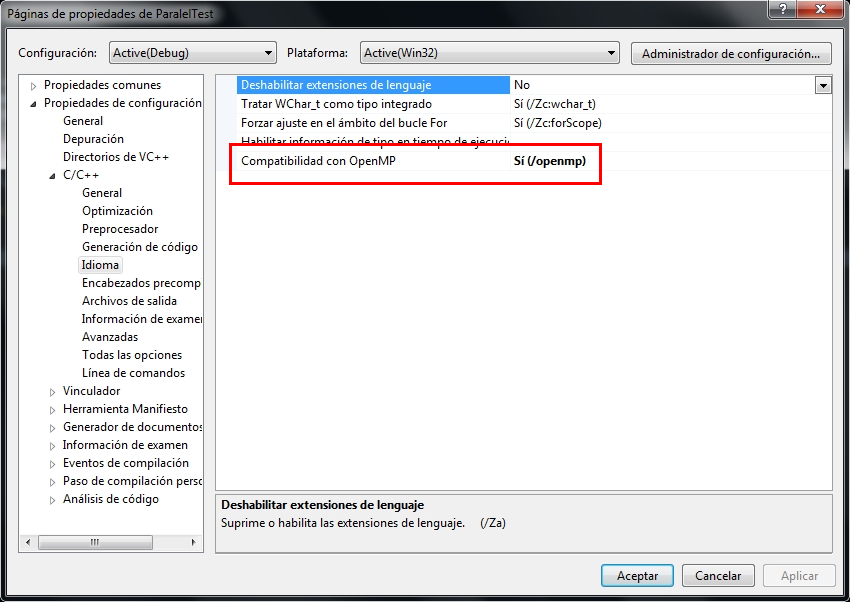
El presente trabajo práctico se propone hacer una revisión formal de los principales algoritmos de ordenamiento, tanto en el ámbito de la programación lineal como el de la programación paralela. El objetivo del trabajo es del de generar un documento que sirva como referencia para el entendimiento de la utilización de la programación paralela en el campo de la Informática.

# Plataforma

Luego de investigar las distintas opciones disponibles para el procesamiento en paralelo, (se investigó CUDA, OpenMP y MPI), finalmente se decidió utilizar OpenMP por su amplia compatibilidad y la cantidad de aportes en línea. Se trabajó sobre un Sistema Operativo Windows 7, utilizando el Visual Studio 2012 SP 4 en el lenguaje C/C++.

Se eligió el Visual Studio ya que pose soporte nativo para OpenMP, incluyendo su librería correspondiente:

#include <omp.h>

Sin embargo, el compilador no comenzará a trabajar con multithreading hasta que no se habilite la opción *Compatibilidad con OpenMP* dentro de *Propiedades de la configuración* ->  *C/C++* -> *Idioma*, en las propiedades del proyecto:

# Desarrollo del programa

Como se mencionó anteriormente, se eligió como lenguaje C/C++. Para esto se creó una solución con un proyecto de Aplicación de consola Win32.

El programa comienza cargando un vector desordenado, utilizando las funciones para la generación de números pseudo-aleatorios de la librería stdlib, inicializando siempre la generación de la semilla con la función:

srand(2);

para que todas las ejecuciones posean el mismo vector, para lograr un informe lo más objetivo posible. Para setear la cantidad de threads deseados, se utilizó la función provista por OpenMP, que recibe por parámetro la cantidad a crear:

omp\_set\_num\_threads(CANTIDAD\_THREADS);

Luego el programa solicitará por pantalla la selección de una de las 4 opciones, que representan a los distintos algoritmos de ordenamiento; una vez seleccionada; el programa ejecutará primero un ordenamiento lineal, y luego en paralelo; indicando los tiempos de cada uno. Para llevar a cabo dicho objetivo, se utilizó la librería time, donde al comienzo de cada ordenamiento se declaró una variable de tipo clock\_t:

clock\_t t;

t = clock();

y al finalizar dicho ordenamiento, se obtuvo la diferencia, para luego mostrarla por pantalla tanto en ticks del reloj como en segundos, usando la constante CLOCKS\_PER\_SEC:

t = clock() - t;

cout<<"Ordenamiento realizado.\n";

cout<<"Tardo "<<t<<" ticks del reloj. ( "<< ((float)t)/CLOCKS\_PER\_SEC<<" segundos)";

Luego de cada set de ordenamientos, es posible ver determinada posición del vector, para corroborar que el mismo fue correctamente realizado. Esto fue corroborado en todos los casos, ya que en varias ocasiones, antes de lograr un ordenamiento paralelo correcto, el vector no quedaba correctamente ordenado. Algoritmos de ordenamiento

Para este trabajo práctico se eligieron 4 algoritmos de ordenamiento:

1. Burbujeo
2. Burbujeo bidireccional
3. Shell Sort
4. Insertion Sort.

A continuación se copiará el código fuente de sus versiones en forma lineal y en forma paralela.

## Burbujeo

Performance para el peor caso: О(*n*2)

Performance para el mejor caso: O(*n*)

Complejidad: O(n²)

### Versión lineal

int i;

int iAux;

int iPaso=1;

bool bOrd;

do

{

bOrd=TRUE;

for(i=0;i<TOPEF-iPaso;i++)

{

if(v2[i]>v2[i+1])

{

iAux = v2[i];

v2[i] = v2[i+1];

v2[i+1] = iAux;

bOrd=FALSE;

}

}

iPaso++;

}while(!bOrd);

### Versión paralela

int i;

int fase;

int tmp;

int iPos;

#pragma omp parallel default(none) shared(v2) private(i, tmp, fase**)//Usando default fuerzo a decidir el scope de TODAS las variables**

for (fase=0; fase<TOPEF;fase++)

{

if(fase % 2 == 0)

**//#pragma omp for //SE COMENTA PORQUE CON 8 NUCLEOS TARDA MAS QUE EL LINEAL**

for (i=1;i<TOPEF;i+=2)

{

if(v2[i-1]>v2[i])

{

tmp = v2[i-1];

v2[i-1] = v2[i];

v2[i] = tmp;

}

}

else

**//#pragma omp for //SE COMENTA PORQUE CON 8 NUCLEOS TARDA MAS QUE EL LINEAL**

for(i=1;i<TOPEF-1;i+=2)

{

if(v2[i]>v2[i+1])

{

tmp=v2[i+1];

v2[i+1] = v2[i];

v2[i]= tmp;

}

}

}

## Burbujeo bidireccional (cocktail sort)

Performance para el peor caso: О(*n*2)

Performance para el mejor caso: O(*n*)

Complejidad: O(n²)

### Versión lineal

register int a;

int exchange;

int t;

do

{

exchange = 0;

for(a=TOPEF-1; a>0; --a)

{

if(v2[a-1] > v2[a])

{

t = v2[a-1];

v2[a-1] = v2[a];

v2[a] = t;

exchange = 1;

}

}

for(a=1; a < TOPEF; ++a)

{

if(v2[a-1] > v2[a])

{

t = v2[a-1];

v2[a-1] = v2[a];

v2[a] = t;

exchange = 1;

}

}

}while(exchange);

### Versión paralela

register int a;

int exchange;

int t=0;

long iPos;

#pragma omp parallel default(none) shared(v2) private(a, t, exchange)

do

{

exchange = 0;

**//#pragma omp for //SE CUELGA CON 8 NUCLEOS, PERO CON 2 FUNCIONA**

for(a=TOPEF-1; a>0; --a)

{

if(v2[a-1] > v2[a])

{

t = v2[a-1];

v2[a-1] = v2[a];

v2[a] = t;

exchange = 1;

}

}

**//#pragma omp for //SE CUELGA CON 8 NUCLEOS, PERO CON 2 FUNCIONA**

for(a=1; a < TOPEF; ++a)

{

if(v2[a-1] > v2[a])

{

t = v2[a-1];

v2[a-1] = v2[a];

v2[a] = t;

exchange = 1;

}

}

}while(exchange);

## Shell Sort

Performance para el peor caso: O(*n*log2 2*n*)

Performance para el mejor caso: O(*n* log *n*)

Complejidad: О(n)

### Versión lineal

int i, j, incrmnt=0, temp;

incrmnt = TOPEF/2;

while (incrmnt > 0)

{

for (i=incrmnt; i < TOPEF; i++)

{

j = i;

temp = v2[i];

while ((j >= incrmnt) && (v2[j-incrmnt] > temp))

{

v2[j] = v2[j - incrmnt];

j = j - incrmnt;

}

v2[j] = temp;

}

incrmnt /= 2;

}

### Versión paralela

int i=0, j=0, incrmnt=0, temp=0;

long iPos;

incrmnt = TOPEF/2;

#pragma omp parallel default(none) firstprivate(incrmnt) shared(v2) private(i, j, temp)**//Con firstprivate mantengo el valor inicializado anteriormente de incrmnt**

{

while (incrmnt > 0)

{

#pragma omp for

for (i=incrmnt; i < TOPEF; i++)

{

j = i;

temp = v2[i];

while ((j >= incrmnt) && (v2[j-incrmnt] > temp))

{

v2[j] = v2[j - incrmnt];

j = j - incrmnt;

}

v2[j] = temp;

}

incrmnt /= 2;

}

}

## InsertionSort

Performance para el peor caso: О(*n*2) comparaciones, swaps

Performance para el mejor caso: O(*n*) comparaciones, O(*1*) swaps

Complejidad: O(n²)

### Versión lineal

int i, j, index;

for (i=1; i<TOPEF; i++)

{

index = v2[i];

j = i;

while ((j>0) && (v2[j-1] > index))

{

v2[j] = v2[j-1];

j = j - 1;

}

v2[j] = index;

}

### Versión paralela

int i, j, index;

long iPos;

#pragma omp parallel for default(none) shared(v2) private(i, j, index)

for (i=1; i<TOPEF; i++)

{

index = v2[i];

j = i;

while ((j>0) && (v2[j-1] > index))

{

v2[j] = v2[j-1];

j = j - 1;

}

v2[j] = index;

}

# Resultados

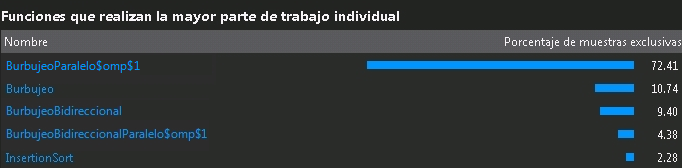
Se realizaron 5 ejecuciones del programa completo, que contiene los 4 algoritmos de ordenamiento con sus 2 variantes. A continuación se detallan los resultados de las mismas, indicando el ámbito donde fueron ejecutadas.

## Primera ejecución – 8 núcleos con 8 threads

### Cuadro comparativo

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | **Ordenamiento** | **Paralelo** | **Tiempo de ejecución(segundos)** | **Diferencia entre sí** |
| **Equipo** | Intel Core i7 |  | Burbujeo | No | 24,892 | 1,76 |
| **Núcleos** | 8 |  | Burbujeo | Sí | 23,132 | -1,76 |
| **Threads** | 8 |  | Burbujeo bidireccional | No | 22,5 | 20,942 |
| **Tamaño array** | 100.000 |  | Burbujeo bidireccional | Sí | 1,558 | -20,942 |
| **Tipo array** | int |  | Shell Sort | No | 0,066 | 0,055 |
|  |  |  | Shell Sort | Sí | 0,011 | -0,055 |
|  |  |  | InsertionSort | No | 5,692 | 5,273 |
|  |  |  | InsertionSort | Sí | 0,419 | -5,273 |
|  |  |  | **TOTAL** | | 78,27 |  |

### Gráfico de uso de CPU

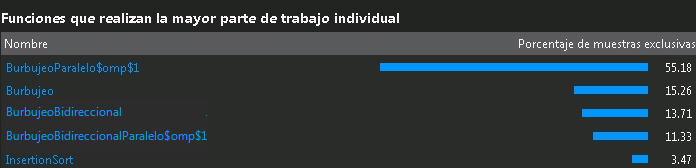


## Segunda ejecución – 8 núcleos con 4 threads

### Cuadro comparativo

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | **Ordenamiento** | **Paralelo** | **Tiempo de ejecución(segundos)** | **Diferencia entre sí** |
| **Equipo** | Intel Core i7 |  | Burbujeo | No | 24,118 | 1,626 |
| **Núcleos** | 8 |  | Burbujeo | Sí | 22,492 | -1,626 |
| **Threads** | 4 |  | Burbujeo bidireccional | No | 21,824 | 17,292 |
| **Tamaño array** | 100.000 |  | Burbujeo bidireccional | Sí | 4,532 | -17,292 |
| **Tipo array** | int |  | Shell Sort | No | 0,068 | 0,059 |
|  |  |  | Shell Sort | Sí | 0,009 | -0,059 |
|  |  |  | InsertionSort | No | 5,484 | 5,029 |
|  |  |  | InsertionSort | Sí | 0,455 | -5,029 |
|  |  |  | **TOTAL** | | 78,982 |  |

### Gráfico de uso de CPU

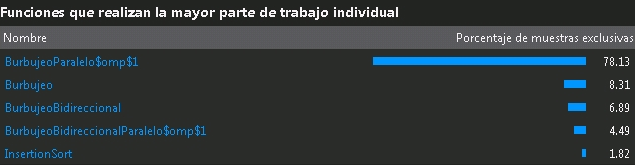


## Tercera ejecución – 8 núcleos con 12 threads

### Cuadro comparativo

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | **Ordenamiento** | **Paralelo** | **Tiempo de ejecución(segundos)** | **Diferencia entre sí** |
| **Equipo** | Intel Core i7 |  | Burbujeo | No | 24,096 | -7.39 |
| **Núcleos** | 8 |  | Burbujeo | Sí | 31,486 | -1,626 |
| **Threads** | 12 |  | Burbujeo bidireccional | No | 19,88 | 18,116 |
| **Tamaño array** | 100.000 |  | Burbujeo bidireccional | Sí | 1,764 | -18,116 |
| **Tipo array** | int |  | Shell Sort | No | 0,066 | 0,056 |
|  |  |  | Shell Sort | Sí | 0,01 | -0,056 |
|  |  |  | InsertionSort | No | 5,492 | 5,2 |
|  |  |  | InsertionSort | Sí | 0,292 | -5,2 |
|  |  |  | **TOTAL** | | 83,086 |  |

### Gráfico de uso de CPU

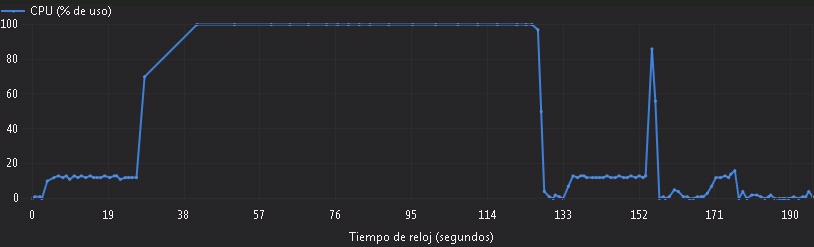


## Cuarta ejecución – 8 núcleos con 50 threads

### Cuadro comparativo

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | **Ordenamiento** | **Paralelo** | **Tiempo de ejecución(segundos)** | **Diferencia entre sí** |
| **Equipo** | Intel Core i7 |  | Burbujeo | No | 24,043 | -76,542 |
| **Núcleos** | 8 |  | Burbujeo | Sí | 100,585 | 76,542 |
| **Threads** | 50 |  | Burbujeo bidireccional | No | 19,89 | 18,114 |
| **Tamaño array** | 100.000 |  | Burbujeo bidireccional | Sí | 1,776 | -18,114 |
| **Tipo array** | int |  | Shell Sort | No | 0,065 | 0,048 |
|  |  |  | Shell Sort | Sí | 0,017 | -0,048 |
|  |  |  | InsertionSort | No | 5,477 | 5,335 |
|  |  |  | InsertionSort | Sí | 0,142 | -5,335 |
|  |  |  | **TOTAL** | | 151,995 |  |

### Gráfico de uso de CPU

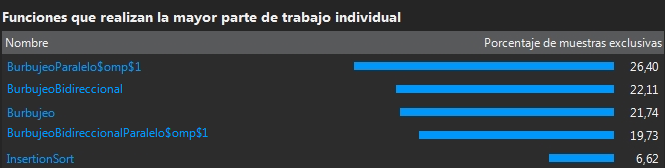


## Quinta ejecución – 2 núcleos con 2 threads

### Cuadro comparativo

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | **Ordenamiento** | **Paralelo** | **Tiempo de ejecución(segundos)** | **Diferencia entre sí** |
| **Equipo** | Intel Core 2 DUO |  | Burbujeo | No | 33,639 | 10,288 |
| **Núcleos** | 2 |  | Burbujeo | Sí | 23,351 | -10,288 |
| **Threads** | 2 |  | Burbujeo bidireccional | No | 33,368 | 17,116 |
| **Tamaño array** | 100.000 |  | Burbujeo bidireccional | Sí | 16,252 | -17,116 |
| **Tipo array** | int |  | Shell Sort | No | 0,035 | 0,013 |
|  |  |  | Shell Sort | Sí | 0,022 | -0,013 |
|  |  |  | InsertionSort | No | 10,151 | 7,354 |
|  |  |  | InsertionSort | Sí | 2,797 | -7,354 |
|  |  |  | **TOTAL** | | 119,615 |  |

### Gráfico de uso de CPU



# Conclusiones

Dependiendo de los objetivos deseados, cada ejecución tiene sus PROS y CONTRAS. A continuación se detallan los diferentes puntos de vista en base a las distintas consideraciones posibles de los resultados obtenidos.

## Puntos de vista

### El programa completo

#### Cuadro comparativo

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mejor performance** | | | **Peor performance** | | | **Diferencia** | **% de mejora** |
| **Tiempo de ejecución(segundos)** | **Núcleos** | **Threads** | **Tiempo de ejecución(segundos)** | **Núcleos** | **Threads** |
| 78,27 | 8 | 8 | 151,995 | 8 | 50 | 73,725 | 48,5 |

Si debemos considerar el rendimiento del programa como un todo, independientemente de cada algoritmo de ordenamiento por separado, la ejecución más rápida fue la primera (8 núcleos e igual cantidad de threads), logrando 78,27 segundos. Ninguna otra ejecución pudo bajar ese tiempo (aunque la segunda ejecución quedó apenas por debajo con 78,982 segundos), ya que el Burbujeo Paralelo comenzó a volverse más lento, afectando al resultado general. Por otro lado, quizás sorpresivamente, la ejecución más lenta fue la cuarta (8 núcleos y 50 threads), ya que el burbujeo paralelo tomó 76,542 segundos, totalizando 151,995 segundos. Esto permite concluir que al superar la misma cantidad de núcleos que de threads, el comportamiento del burbujeo paralelo comienza a tardar más tiempo, en vez de mejorar.

### Por algoritmo

#### Cuadro comparativo

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Ordenamiento** | **Mejor performance** | | | **Peor performance** | | | **Diferencia** | **% de mejora** |
| **Tiempo de ejecución(segundos)** | **Núcleos** | **Threads** | **Tiempo de ejecución(segundos)** | **Núcleos** | **Threads** |
| Burbujeo | 22,492 | 8 | 4 | 100,585 | 8 | 50 | 78,093 | 77,63 |
| Burbujeo bidireccional | 1,558 | 8 | 8 | 16,252 | 2 | 2 | 14,694 | 90,41 |
| Shell Sort | 0,009 | 8 | 4 | 0,022 | 2 | 2 | 0,013 | 59,03 |
| InsertionSort | 0,142 | 8 | 50 | 2,797 | 2 | 2 | 2,655 | 94,92 |

Como se mencionó anteriormente, el burbujeo fue el único algoritmo que obtuvo resultados negativos en algunas ejecuciones en paralelo (tercera y cuarta), mientras que el resto de los algoritmos siempre tuvieron una mejor performance en ejecución en paralelo.

Al visualizar el cuadro comparativo podemos notar que la variedad de combinaciones “ganadoras” nos indica que no existe una única receta a la hora de elegir la mejor cantidad de threads; de 4 algoritmos, se obtuvieron 3 combinaciones ganadoras distintas, hecho que afirma lo mencionado. Si bien se puede notar cierta tendencia a que los algoritmos fueron más beneficiados con la mitad de threads que de núcleos (Burbujeo y Shell Sort), y apenas duplicando los threads nos encontramos con la mejor versión del Burbujeo Bidireccional; en el otro extremo tenemos al InsertionSort, que contrariamente a los demás algoritmos, sí se beneficio del gran número de threads (50) y obtuvo su mejor performance. Además fue el que tuvo mejor porcentaje de mejora, pasando de un tiempo de ejecución en 2 núcleos con 2 threads de 2,797 segundos a 8 núcleos con 50 threads de 0,142 segundos; logrando un porcentaje de progreso del 94,92%.

Finalmente podemos concluir que dependiendo de la estrategia a utilizar, tendremos que seleccionar de manera distinta la cantidad de threads adecuados para lograr la mejor performance, seguramente si se hubiesen probado más algoritmos de ordenamiento, nos hubiésemos encontrado con más combinaciones distintas; ya que la mejor performance dependerá de cómo este programado el algoritmo, su cantidad de comparaciones y su complejidad.

# Bibliografía

<http://openmp.org/wp/>

<https://datsi.fi.upm.es/docencia/Arquitectura_09/Multis/AC_MP_OpenMP.pdf>

<http://www.ditec.um.es/~javiercm/curso_psba/sesion_03_openmp/PSBA_OpenMP.pdf>

<https://en.wikipedia.org/wiki/Sorting_algorithm>

<https://en.wikipedia.org/wiki/Bubble_sort>

<https://en.wikipedia.org/wiki/Cocktail_shaker_sort>

<https://en.wikipedia.org/wiki/Shellsort>

<https://en.wikipedia.org/wiki/Insertion_sort>