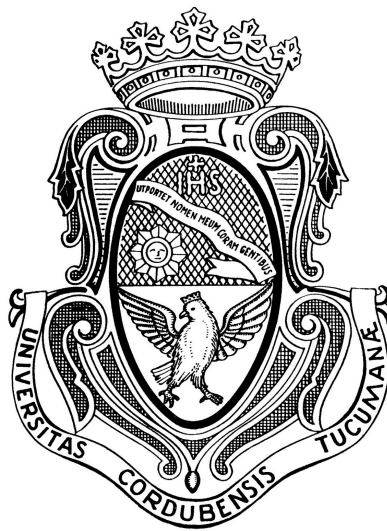


# UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE CIENCIAS EXACTAS  
FÍSICAS Y NATURALES



## SISTEMAS OPERATIVOS II

TRABAJO PRÁCTICO N°2: Programación paralela  
con OpenMP

Integrantes:

- Cazajous Miguel A. - 34980294

Córdoba - Argentina

13 de mayo de 2018

# Índice

<b>1. Introducción:</b>	<b>1</b>
1.1. Propósito: . . . . .	1
1.2. Ambito del sistema: . . . . .	1
1.3. Definiciones, acrónimos y abreviaturas: . . . . .	1
1.4. Referencias: . . . . .	3
1.5. Descripción general del documento: . . . . .	3
<b>2. Descripción general:</b>	<b>4</b>
2.1. Perspectiva del producto: . . . . .	4
2.2. Funciones del producto: . . . . .	4
2.3. Características de los usuarios: . . . . .	4
2.4. Restricciones: . . . . .	4
2.5. Suposiciones y dependencias: . . . . .	4
2.6. Requisitos futuros: . . . . .	5
<b>3. Requisitos específicos:</b>	<b>5</b>
3.1. Interfaces externas: . . . . .	5
3.2. Funciones: . . . . .	5
3.3. Requisitos de rendimiento: . . . . .	5
3.4. Restricciones de diseño: . . . . .	6
3.5. Atributos del sistema: . . . . .	6
<b>4. Otros requisitos:</b>	<b>6</b>
<b>5. Diseño de solución:</b>	<b>7</b>
<b>6. Implementación y resultados:</b>	<b>7</b>
6.1. Procedural: . . . . .	7
6.1.1. Ejecución PC: . . . . .	10
6.1.2. Ejecución Clúster - Franky (32) . . . . .	11
6.2. Paralelo: . . . . .	13
6.2.1. Ejecución PC . . . . .	13
6.2.2. Ejecución Clúster - Franky (32) . . . . .	14
6.3. Formato archivo de salida: . . . . .	16
<b>7. Conclusiones:</b>	<b>16</b>

<b>8. Apéndices:</b>	<b>17</b>
8.1. OpenMP: . . . . .	17
8.1.1. Sintaxis básica: . . . . .	17
8.1.2. Modelo de ejecución: . . . . .	17
8.1.3. Cláusulas de planificación: . . . . .	17
8.1.4. Funciones: . . . . .	18

# 1. Introducción:

## 1.1. Propósito:

El propósito del trabajo es el procesamiento de pulsos recibidos desde un conversor A/D que se encuentra ubicado a la entrada de un receptor en un radar, con el fin de implementar un procesador doppler.

## 1.2. Ambito del sistema:

El programa será realizado usando un desarrollo procedural y luego modificarlo para que utilice el paralelismo, por lo que se espera obtener información acerca de los beneficios de la programación paralela en tareas en las que sea posible su aplicación.

Se busca además aprender sobre el uso de herramientas de profiling.

## 1.3. Definiciones, acrónimos y abreviaturas:

1. Procesador Doppler: Procesador usado en procesamiento de señales “pulse-doppler”
2. Pulse-Doppler signal procesing: Es una estrategia de mejora de rendimiento de un radar que permite la detección de objetos pequeños que se mueven a gran velocidad en cercanía con objetos más grandes que también están en movimiento.
3. OpenMP: Es una API para la programación multiproceso de memoria compartida.
4. API: Interfaz de programación de aplicaciones, conjunto de funciones y subrutinas que ofrece una biblioteca.
5. Biblioteca: Conjunto de implementaciones funcionales.
6. Memoria compartida: Se refiere a un conjunto de procesadores que comparten memoria principal, como en el caso de las computadoras multinúcleos.
7. Hardware: Referido a productos tangibles donde corre un software. Se refiere a una computadora, un clúster, etc.
8. Software: Todo lo referente a programas, código fuente, etc.



9. Conversor A/D: Convierte una señal analógica en un valor digital que consta de palabras binarias.
10. Radar: Acrónimo de RAdio Detection And Ranging, emite ondas electromagnéticas para la medición de distancias y velocidades entre otras cosas.
11. Procedural: Referido a un programa que posee solo un hilo de ejecución, todas las tareas se realizan secuencialmente.
12. Paralelo: Las tareas se pueden ejecutar en paralelo siempre y cuando no haya una dependencia fuerte entre una y otra.
13. Pulsos: Emisión de energía electromagnética de alta intensidad y corta duración.
14. Clúster: Conjunto de computadoras autónomas que operan juntas para proporcionar mayor potencia que cada computadora individualmente.
15. Gate: Es una medida de la resolución de un radar.
16. Autocorrelación: Herramienta estadística para el procesamiento de señales.
17. Archivo binario: Archivo que posee palabras binarias, (valor 0 ó 1), no puede ser interpretado como un archivo de texto, sino que debe ser abierto con un programa dedicado.
18. Estructura: Se refiere a un bloque de código que puede representar como un nuevo tipo de datos que es definido fuera de los tipos de un lenguaje.
19. C90: Es un estándar del lenguaje C, existe además C89, C95, C99 y C11. La idea de estos estándares es mejorar la portabilidad de código.
20. UML: Unified Modeling Language, lenguaje de modelado unificado, es un estándar para el diseño de productos, que cuenta con diferentes tipos de diagramas.
21. Profiling: Análisis dinámico de programas.

#### 1.4. Referencias:

- [1] Mendez Gonzalo. Especificacion de Requisitos segun el estandar de IEEE 830. URL: <https://www.fdi.ucm.es/profesor/gmendez/docs/is0809/ieee830.pdf>.
- [2] Lucid Software Inc. LucidChart. URL: <https://www.lucidchart.com>.
- [3] Wikipedia. OpenMP. URL: <https://es.wikipedia.org/wiki/OpenMP>.
- [4] Wikipedia. Autocorrelacion. URL: <https://es.wikipedia.org/wiki/Autocorrelaci%C3%83%C2%B3n>.
- [5] Wikipedia. Pulso electromagnetico. URL: [https://es.wikipedia.org/wiki/Pulso\\_electromagn%C3%83%C2%A9tico](https://es.wikipedia.org/wiki/Pulso_electromagn%C3%83%C2%A9tico).
- [6] Wikipedia. Radar. URL: <https://es.wikipedia.org/wiki/Radar>.
- [7] Wikipedia. Pulse-Doppler signal processing. URL: [https://en.wikipedia.org/wiki/Pulse-Doppler\\_signal\\_processing](https://en.wikipedia.org/wiki/Pulse-Doppler_signal_processing).

#### 1.5. Descripción general del documento:

Este documento se compone de 7 secciones principales, siendo la primera la introducción, donde se explica brevemente el fin del proyecto como así también sus requerimientos de forma muy general. En la segunda sección la descripción general del sistema con el fin de conocer las principales funciones que debe ser capaz de realizar, conocer a quién va dirigido, además de restricciones y supuestos que puedan afectar el desarrollo del mismo. En la sección 3 se definen de manera detallada los requerimientos del sistema a desarrollar, los que definen el comportamiento del sistema como así también otros requerimientos que puedan ser deseados considerando el uso que el sistema va a tener. En la cuarta sección se presenta el diseño del sistema que dará solución a los requerimientos antes enumerados. En la quinta sección se discuten los resultados de implementación del diseño elaborado en la etapa anterior donde se muestra como el sistema opera. En la sección 6 se elabora una breve conclusión acerca de la experiencia en la elaboración del proyecto. Finalmente en la última sección se agrega información alternativa que pueda ser de interés.

## **2. Descripción general:**

### **2.1. Perspectiva del producto:**

La aplicación deberá poner en evidencia los beneficios de la programación paralela aprovechando de la mejor manera los recursos de hardware para la mejora de tiempos de procesamiento. En proyectos de gran envergadura el tiempo es un recurso preciado.

### **2.2. Funciones del producto:**

De manera general el programa debe ser capaz de:

1. Abrir, procesar y cerrar un archivo binario que contiene información de los pulsos.
2. Hacer uso del paralelismo mediante OpenMP.

### **2.3. Características de los usuarios:**

Personas que estén a cargo de la implementación del procesador Doppler serán los principales interesados en los resultados que ofrece el programa. El programa no es interactivo por lo que el desarrollador será el principal usuario, los demás solo están interesados en la información que este procesa y obtiene.

### **2.4. Restricciones:**

El programa debe ser enteramente desarrollado en C y debe hacer uso de la librería OpenMP, la creación del archivo donde se guarda el procesamiento debe ser binario al igual que el de origen.

El sistema operativo a implementarse no está definido, ya que la información que se obtiene del programa es la parte final del producto.

El sistema operativo de desarrollo es GNU/Linux, por ese lado sería conveniente, si se desea ejecutarlo, usar una distribución de esa plataforma.

### **2.5. Suposiciones y dependencias:**

Si se presenta información sobre pulsos con otra estructura que difiera por mucho del archivo binario usado en este proyecto puede ser necesario realizar una revisión del programa que se adapte a ese tipo de estructura.

## 2.6. Requisitos futuros:

No se tienen al momento de confeccionar este informe requisitos futuros que puedan ser de interés.

## 3. Requisitos específicos:

### 3.1. Interfaces externas:

#### Interfaz de hardware:

El programa será corrido en una computadora personal y en un clúster. En ambos la potencia del hardware no es de mayor importancia. Cualquier computadora personal debería ser capaz de realizar la tarea sin inconvenientes.

### 3.2. Funciones:

El sistema deberá ser capaz de:

1. Abrir un archivo binario conociendo previamente su estructura.
2. Contabilizar la cantidad de estructuras que este posee (pulsos) y la información de cada uno (muestras) para cada canal.
3. Obtener, con el dato anterior, el valor complejo de las muestras de cada pulso para cada canal.
4. Obtener gates por pulso para cada canal.
5. Armar una estructura (gate, pulso) por canal.
6. Aplicar fórmula de autocorrelación para cada canal.
7. Guardar lo obtenido en un archivo binario.
8. Cada una de las funcionalidades anteriores debe ser implementada, siempre y cuando sea posible, mediante el uso de la librería OpenMP.

### 3.3. Requisitos de rendimiento:

Los requisitos de rendimiento están ligados al tiempo de ejecución del programa para el procesamiento de datos, una vez más, no es un programa interactivo, por lo que no se mencionan conceptos como el tiempo de respuesta.



### **3.4. Restricciones de diseño:**

La mayor restricción de diseño que se tiene es que el proyecto debe ser implementado en su totalidad en el lenguaje de programación C en su estándar C90.

### **3.5. Atributos del sistema:**

El sistema es fácilmente portable a cualquier computadora y plataforma de desarrollo que cumpla con los requisitos.

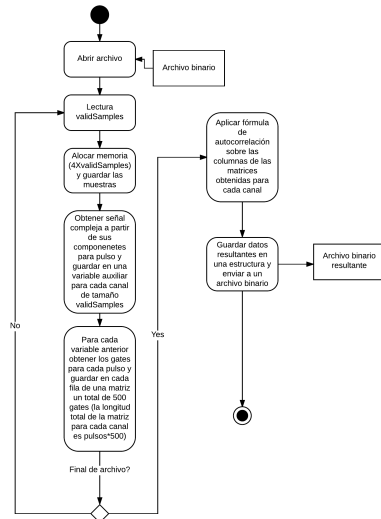
La mantenibilidad del software es sencilla ya que el mismo lleva consigo un control de versiones implementado mediante la herramienta Git.

## **4. Otros requisitos:**

No posee.

## 5. Diseño de solución:

El diseño de solución se realizará usando un diagrama de actividades del lenguaje de modela UML. (Click en la imagen para abrir)



## 6. Implementación y resultados:

### 6.1. Procedural:

A continuación se muestran las mediciones de tiempo del programa procedural para lo cual se hizo uso de diferentes herramientas de profiling.

1. Gprof: Primero se intentó con esta herramienta pero se tuvieron inconvenientes ya que el tiempo que demoraba la ejecución del programa en varias ocasiones era menor que la resolución de esta herramienta y no se podía obtener información relevante del tiempo que el programa demoraba, aunque se pudo observar que función era la que consumía más tiempo, dato que sirvió para saber que paralelizar.

Para hacer uso de Gprof se deben agregar las a GCC las siguientes banderas.

#### Código

```
gcc -g -pg ...
```

Luego ejecutar el programa (make proc) para que se cree un archivo llamado gmon.out, y después mediante.

#### Comando

```
gprof ./programa
```

Se obtiene la información solicitada.

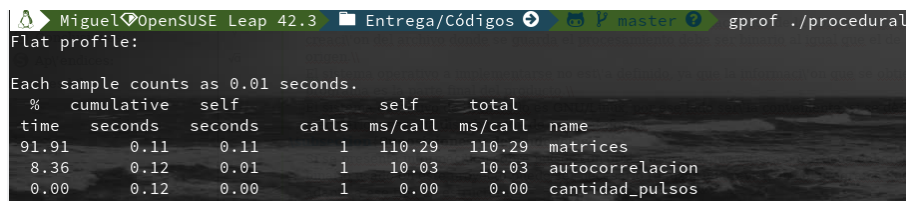


Figura 1: Salida del programa “gprof”

La función matrices como puede verse es la que más tiempo demora en ejecutar. Los datos que expone difieren de otras herramientas por lo que de esta solo se considera que función es la que más demora en ejecutarse.

2. Time: Una función muy simple que puede ejecutarse mediante.

#### Comando

```
time ./programa
```

Nos da una salida como la siguiente:

```
./procedural 0.31s user 0.09s system 99% cpu 0.398 total
```

Figura 2: Salida del programa “time”

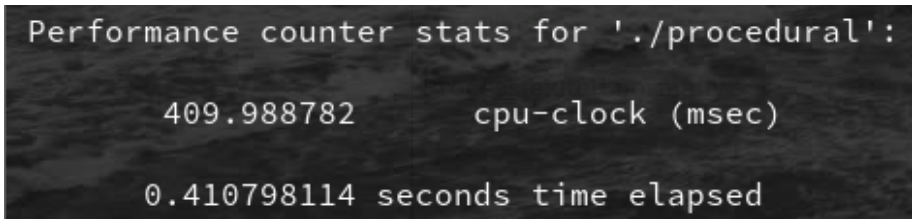
Podemos estimar el tiempo total en 40ms.

3. Perf: Otra herramienta útil que se puede encontrar ya instalada en GNU/Linux es perf y se ejecuta mediante:

Comando

```
perf stat -e cpu-clock ./programa
```

La salida es:



```
Performance counter stats for './procedural':  
  
409.988782      cpu-clock (msec)  
  
0.410798114 seconds time elapsed
```

Figura 3: Salida del programa “perf”

Como vemos el tiempo es muy similar al obtenido mediante time

4. Openmp (omp\_get\_wtime()): Aunque no se está paralelizando, omp.h cuenta con una función muy útil de la que se hizo uso para medir el tiempo en esta ejecución procedural. Se debe compilar el programa con:

Código

```
gcc -fopenmp ...
```

Obteniendo el tiempo antes y después de la ejecución de la función de la cual queremos medir el tiempo y luego haciendo la diferencia obtenemos los siguientes resultados.

Código

```
valor_inicial = omp_get_wtime();  
funcion();  
valor_final = omp_get_wtime();  
tiempo = valor_final - valor_inicial;
```



```
Miguel OpenSUSE Leap 42.3 Entrega/Códigos master make proc
./procedural
Cálculo de complejos: 0.34361
Cálculo de matrices: 0.03267
Tiempo autocorrelación: 0.00747
Tiempo total: 0.40412
Tiempo de la función matrices: 0.39316
```

Figura 4: Salida del programa usando “openmp”

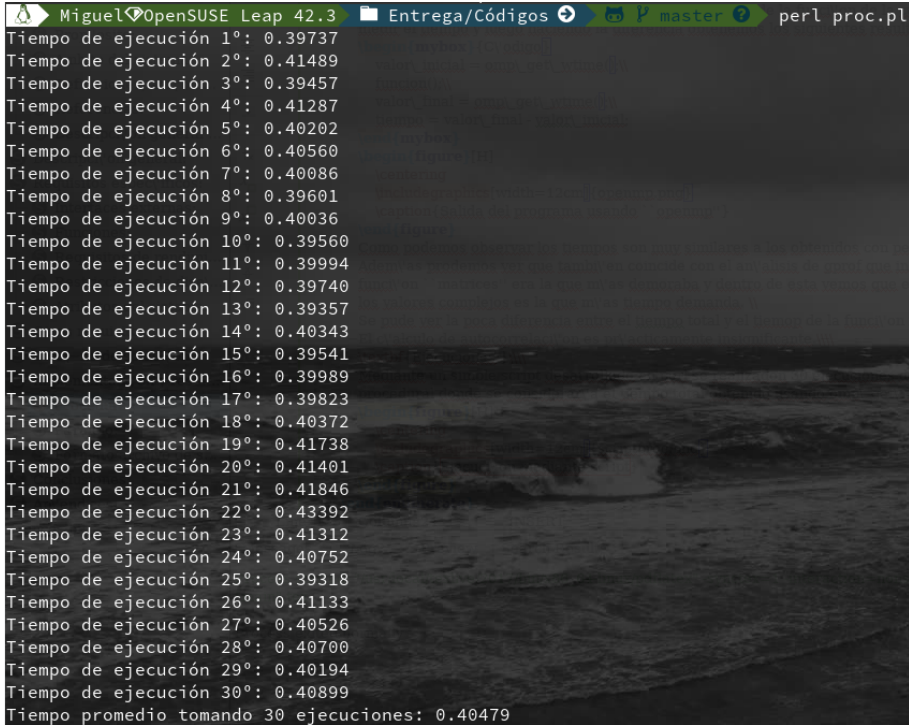
Como podemos observar los tiempos son muy similares a los obtenidos con `perf` y `time`. Además podemos ver que también coincide con el análisis de `gprof` que indicaba que la función “matrices” era la que más demoraba y dentro de esta vemos que el cálculo de los valores complejos es la que más tiempo demanda.

Se puede ver la poca diferencia entre el tiempo total y el tiempo de la función matrices.

El cálculo de autocorrelación es prácticamente insignificante.

#### 6.1.1. Ejecución PC:

Mediante un simple script desarrollado en perl se realizaron 30 ejecuciones del programa procedural donde se considerará el tiempo total obtenido usando `omp_get_wtime()`



```
Miguel@OpenSUSE Leap 42.3 Entrega/Códigos master perl proc.pl
Tiempo de ejecución 1º: 0.39737
Tiempo de ejecución 2º: 0.41489
Tiempo de ejecución 3º: 0.39457
Tiempo de ejecución 4º: 0.41287
Tiempo de ejecución 5º: 0.40202
Tiempo de ejecución 6º: 0.40560
Tiempo de ejecución 7º: 0.40086
Tiempo de ejecución 8º: 0.39601
Tiempo de ejecución 9º: 0.40036
Tiempo de ejecución 10º: 0.39560
Tiempo de ejecución 11º: 0.39994
Tiempo de ejecución 12º: 0.39740
Tiempo de ejecución 13º: 0.39357
Tiempo de ejecución 14º: 0.40343
Tiempo de ejecución 15º: 0.39541
Tiempo de ejecución 16º: 0.39989
Tiempo de ejecución 17º: 0.39823
Tiempo de ejecución 18º: 0.40372
Tiempo de ejecución 19º: 0.41738
Tiempo de ejecución 20º: 0.41401
Tiempo de ejecución 21º: 0.41846
Tiempo de ejecución 22º: 0.43392
Tiempo de ejecución 23º: 0.41312
Tiempo de ejecución 24º: 0.40752
Tiempo de ejecución 25º: 0.39318
Tiempo de ejecución 26º: 0.41133
Tiempo de ejecución 27º: 0.40526
Tiempo de ejecución 28º: 0.40700
Tiempo de ejecución 29º: 0.40194
Tiempo de ejecución 30º: 0.40899
Tiempo promedio tomando 30 ejecuciones: 0.40479
```

Figura 5: Ejecución mediante script

### 6.1.2. Ejecución Clúster - Franky (32)

Se realizaron también 30 ejecuciones del programa.

```
[Alumno2@franky Códigos]$ perl proc.pl
Tiempo de ejecución 1º: 0.40757
Tiempo de ejecución 2º: 0.40625
Tiempo de ejecución 3º: 0.40843
Tiempo de ejecución 4º: 0.40930
Tiempo de ejecución 5º: 0.41057
Tiempo de ejecución 6º: 0.40861
Tiempo de ejecución 7º: 0.43996
Tiempo de ejecución 8º: 0.43996
Tiempo de ejecución 9º: 0.41096
Tiempo de ejecución 10º: 0.41047
Tiempo de ejecución 11º: 0.40903
Tiempo de ejecución 12º: 0.41024
Tiempo de ejecución 13º: 0.41036
Tiempo de ejecución 14º: 0.41012
Tiempo de ejecución 15º: 0.40790
Tiempo de ejecución 16º: 0.41021
Tiempo de ejecución 17º: 0.43932
Tiempo de ejecución 18º: 0.41184
Tiempo de ejecución 19º: 0.40979
Tiempo de ejecución 20º: 0.43941
Tiempo de ejecución 21º: 0.40981
Tiempo de ejecución 22º: 0.40783
Tiempo de ejecución 23º: 0.43917
Tiempo de ejecución 24º: 0.41168
Tiempo de ejecución 25º: 0.43712
Tiempo de ejecución 26º: 0.39626
Tiempo de ejecución 27º: 0.42774
Tiempo de ejecución 28º: 0.40937
Tiempo de ejecución 29º: 0.42566
Tiempo de ejecución 30º: 0.43844
Tiempo promedio tomando 30 ejecuciones: 0.41711
```

Figura 6: Ejecución mediante script en el nodo Franky

Se puede ver que los valores no difieren demasiado con los obtenidos en la ejecución en la PC.

## 6.2. Paralelo:

### 6.2.1. Ejecución PC

Para el programa paralelo se hizo uso de otro script en perl que, mediante su módulo “gnuplot” grafica el promedio del resultado de 100 veces de ejecución para 1,2,3, ..., 10 hilos.

La computadora es una FX - 6300 de 6 núcleos de 1.4Ghz

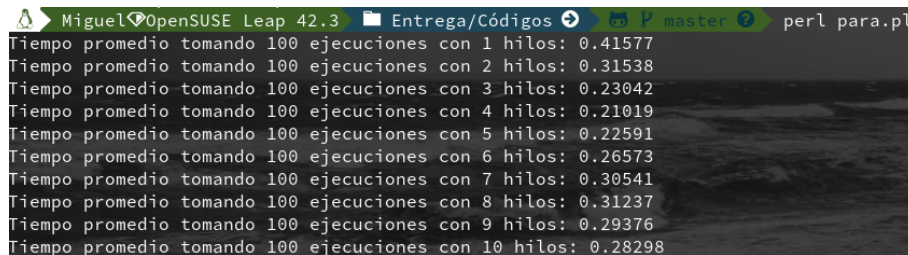
Para usar este script es necesario instalar el módulo para perl mediante CPAN desde consola.

Comando

```
sudo cpan install Chart:Gnuplot
```

*Nota: Verificar también se tenga instalado el programa “gnuplot”.*

Se tomaron 100 ejecuciones por lo que los resultados son muy variables, de modo que así se espera disminuya la desviación en torno a la media.



Hilos	Tiempo promedio (segundos)
1	0.41577
2	0.31538
3	0.23042
4	0.21019
5	0.22591
6	0.26573
7	0.30541
8	0.31237
9	0.29376
10	0.28298

Figura 7: Ejecución mediante script

Se puede ver en la tabla que el óptimo está en torno a los 4 hilos y se observa como se empeora al ir agregando más. En el siguiente gráfico puede apreciarse mejor.



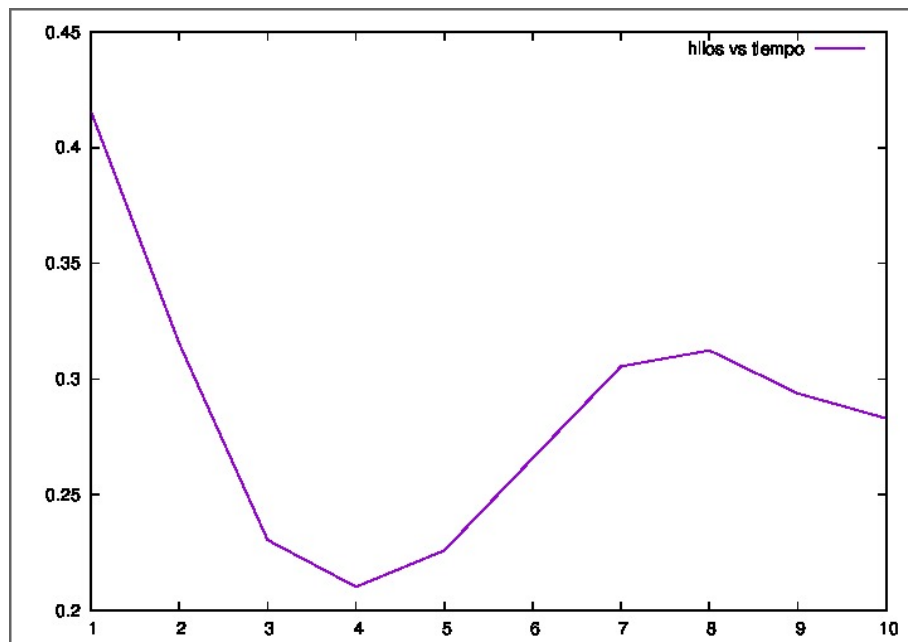


Figura 8: Gráfica mediante “gnuplot”

### 6.2.2. Ejecución Clúster - Franky (32)

Como Franky dispone de 32 procesadores, se hicieron 100 ejecuciones por 32, que es la cantidad de procesadores que dispone, o sea en este caso se realizaron 100 ejecuciones para 1, ... , 32 hilos.

El nodo Franky es de 32 núcleos de 2Ghz.

```
[Alumno2@franky Códigos]$ perl para.pl
Tiempo promedio tomando 100 ejecuciones con 1 hilos: 0.42067
Tiempo promedio tomando 100 ejecuciones con 2 hilos: 0.33879
Tiempo promedio tomando 100 ejecuciones con 3 hilos: 0.31404
Tiempo promedio tomando 100 ejecuciones con 4 hilos: 0.29886
Tiempo promedio tomando 100 ejecuciones con 5 hilos: 0.29244
Tiempo promedio tomando 100 ejecuciones con 6 hilos: 0.28903
Tiempo promedio tomando 100 ejecuciones con 7 hilos: 0.28439
Tiempo promedio tomando 100 ejecuciones con 8 hilos: 0.35027
Tiempo promedio tomando 100 ejecuciones con 9 hilos: 0.36685
Tiempo promedio tomando 100 ejecuciones con 10 hilos: 0.36067
Tiempo promedio tomando 100 ejecuciones con 11 hilos: 0.38070
Tiempo promedio tomando 100 ejecuciones con 12 hilos: 0.37052
Tiempo promedio tomando 100 ejecuciones con 13 hilos: 0.39173
Tiempo promedio tomando 100 ejecuciones con 14 hilos: 0.37887
Tiempo promedio tomando 100 ejecuciones con 15 hilos: 0.37715
Tiempo promedio tomando 100 ejecuciones con 16 hilos: 0.38773
Tiempo promedio tomando 100 ejecuciones con 17 hilos: 0.37586
Tiempo promedio tomando 100 ejecuciones con 18 hilos: 0.38091
Tiempo promedio tomando 100 ejecuciones con 19 hilos: 0.39459
Tiempo promedio tomando 100 ejecuciones con 20 hilos: 0.39944
Tiempo promedio tomando 100 ejecuciones con 21 hilos: 0.42824
Tiempo promedio tomando 100 ejecuciones con 22 hilos: 0.41535
Tiempo promedio tomando 100 ejecuciones con 23 hilos: 0.41898
Tiempo promedio tomando 100 ejecuciones con 24 hilos: 0.45241
Tiempo promedio tomando 100 ejecuciones con 25 hilos: 0.45292
Tiempo promedio tomando 100 ejecuciones con 26 hilos: 0.45424
Tiempo promedio tomando 100 ejecuciones con 27 hilos: 0.50901
Tiempo promedio tomando 100 ejecuciones con 28 hilos: 0.53862
Tiempo promedio tomando 100 ejecuciones con 29 hilos: 0.56869
Tiempo promedio tomando 100 ejecuciones con 30 hilos: 0.63909
```

Figura 9: Ejecución mediante script en el nodo Franky

Se observa que el óptimo es para una cantidad de 7 procesadores.

Se esperaba que el valor óptimo esté en los 32 núcleos.

*Nota1: Se canceló la ejecución ya que para 31 núcleos se estaba demorando demasiado y no se veía ninguna mejora en los tiempos.*

*Nota2: No se pudo realizar una gráfica, ya que debido a que no se tienen permisos de root, no se pudo instalar el módulo que el script de perl necesita.*

### 6.3. Formato archivo de salida:

Se dispone de un programa destinado a abrir el archivo binario resultante.

De todas formas se muestra a continuación la estructura del mismo.

Muestras	int
Autocorrelacion_canalV[0]	float
...	float
Autocorrelacion_canalV[Muestras-1]	float
Autocorrelacion_canalH[0]	float
...	float
Autocorrelacion_canalH[Muestras-1]	float

## 7. Conclusiones:

Con este trabajo se mejoró el uso y la implementación de arreglos dinámicos, sobre todo de dos dimensiones además, también, del manejo de punteros con los que se implementaron estos arreglos para ser tratados entre diferentes funciones.

Se conocieron los beneficios del paralelismo y se encontró a perl un lenguaje muy versátil y de fácil uso que permite hacer diversas tareas, y que además permite ampliar su campo de acción mediante la incorporación de diferentes módulos.

La mayor dificultad estuvo en el momento de realizar la paralelización al notar que el programa procedural que se había implementado no era fácilmente paralelizable, lo que llevó a que se tenga que rehacer, una vez corregido, la implementación con OpenMP para conseguir mejoras en el tiempo de ejecución también trajo algunas dificultades pero pudieron resolverse.

Luego de correr los programas, tanto en la PC como en el nodo Franky, se observaron tiempos no muy satisfactorios, ya que se esperaba que la ejecución en paralelo en Franky fuera notablemente mejor que en la PC, lo cual fue totalmente al revés. Quizás debido a que el programa en paralelo no estuvo implementado optimamente y no se aprovecharon los recursos del nodo donde se ejecutó.

## 8. Apéndices:

### 8.1. OpenMP:

Es una API para la programación multiproceso de memoria compartida. Permite añadir concurrencia a programas C, C++ y Fortran.

Se basa en el modelo fork-join donde una tarea muy pesada se divide en K hilos con menor peso y luego “recolecta” sus resultados al final en uno solo.

#### 8.1.1. Sintaxis básica:

Para C y C++.

##### Código

```
#pragma omp <directiva >[cláusula[, ...]...]
```

#### 8.1.2. Modelo de ejecución:

1. parallel: Directiva que indica que parte del código puede ser ejecutada por varios hilos.
2. for: versión de parallel optimizada para ciclos for.
3. single: Solo se puede ejecutar por un único hilo.
4. master: La parte de código solo puede ejecutarse por el hilo padre.
5. critical: Solo un hilo puede estar en esta sección a la vez.
6. atomic: Se utiliza cuando la operación involucre solo a una posición de memoria, y tiene que ser actualizada por un hilo simultáneamente.

#### 8.1.3. Cláusulas de planificación:

Schedule (tipo, chunk). Es útil si la carga procesal es un bucle do o un bucle for. Las iteraciones son asignadas a los hilos basándose en el método definido en la cláusula.

### Tipos

1. **static:** Aquí todas las iteraciones se reparten entre los hilos antes de que estos ejecuten el bucle. Se reparten iteraciones contiguas equitativamente entre todos los hilos.
2. **dynamic:** Se reparten todas las iteraciones entre los hilos. Cuando un hilo en concreto acaba las iteraciones asignadas, ejecuta una de las iteraciones que estaban por ejecutar. El parámetro “chunk” define el número de iteraciones contiguas que cada hilo ejecutará a la vez.
3. **guided:** Un gran número de iteraciones contiguas son asignadas a un hilo.  
Dicha cantidad de iteraciones decrece exponencialmente con cada nueva asignación hasta un mínimo especificado por el parámetro chunk.

#### **8.1.4. Funciones:**

1. **omp\_set\_num\_threads:** Fija el número de hilos simultáneos.
2. **omp\_get\_num\_threads:** Devuelve el número de hilos en ejecución.
3. **omp\_get\_max\_threads:** Devuelve el número máximo de hilos que lanzará nuestro programa en las zonas paralelas. Es muy útil para reservar memoria para cada hilo.
4. **omp\_get\_thread\_num:** Devuelve el número de hilo dentro del equipo.
5. **omp\_get\_num\_procs:** Devuelve el número de procesadores de nuestro ordenador o los que estén disponibles.
6. **omp\_set\_dynamic:** Valor booleano que nos permite especificar si queremos que el número de hilos crezca y decrezca dinámicamente.