CÁTEDRA DE SISTEMAS OPERATIVOS II

Departamento de Computación

FCEFyN - UNC

Trabajo Práctico 2 – Programación Distribuída

Maero Facundo
Mat: 38479441

Córdoba, Mayo de 2017

Índice

1. Introducción
   1.1. Propósito
   1.2. Ambito del Sistema
   1.3. Definiciones, Acrónimos y Abreviaturas
   1.4. Referencias
   1.5. Descripción General del Documento
2. Descripción General
   2.1. Perspectiva del Producto
   2.2. Funciones del Producto
   2.3. Características de los Usuarios
   2.4. Suposiciones y Dependencias
3. Requisitos Específicos
   3.1. Interfaces Externas
   3.2. Funciones
   3.3. Requisitos de Rendimiento
   3.4. Restricciones de Diseño
4. Diseño de solución
5. Implementación y Resultados
6. Conclusiones
7. Apéndices

1. Introducción

En el presente documento se mostrará el proceso de diseño, codificación y testing de un programa que realiza operaciones matemáticas sobre un conjunto dado de datos, en forma procedural, y utilizando recursos de programación distribuída, con el objetivo de comprobar la ganancia en performance alcanzada.

1.1 Propósito.

El propósito de este trabajo práctico es comprender como funciona la programación distribuída, más específicamente OpenMP, para resolver un problema en particular. Para ello se debe diseñar una solución al problema planteado, primero proceduralmente, y luego explotando el paralelismo de la mejor manera posible. Además se solicita investigar sobre diferentes herramientas de profiling para poder cuantificar la ganancia en performance obtenida.

1.2 Ámbito del Sistema.

El sistema se conforma de una única PC, que ejecuta el programa en cuestión. Esta PC puede ser la que se utilizó para desarrollar el software, o un nodo en el clúster de la Facultad. El uso de esta última se justifica para realizar una comparación de los programas de todos los alumnos sobre un mismo hardware.

1.3 Definiciones, Acrónimos y Abreviaturas

- Pulso: señal que emite el radar cada cierto tiempo, con polarización horizontal y vertical. La señal que regresa al radar luego de emitir un pulso es digitalizada y almacenada para su posterior procesamiento.
- Gate: el radar tiene un alcance de hasta 250 Km. En este rango, puede discriminar distancias de hasta 500 m. El espacio entre cada uno de estos puntos se denomina gate. Por lo que el radar puede discriminar en la distancia dada, 500 gates.

1.4 Descripción General del Documento

El presente documento muestra una visión del proceso desarrollo del trabajo práctico, su diseño, implementación, esquemas a seguir y conclusiones sacadas del mismo.

2. Descripción General

2.1 Perspectiva del Producto

El producto solicitado es un software que lea un archivo binario con información sobre las mediciones que realizó un radar, interpretar los resultados, realizar ciertas operaciones aritméticas con las mediciones, y guardar los resultados en otro archivo binario, con estructura a detallar. La ejecución del programa debe hacerse primero secuencialmente, y luego en forma paralela, aprovechando la naturaleza del problema para disminuir los tiempos de ejecución en la medida de lo posible.

2.2. Funciones del Producto

Las funciones que proporciona el producto son:
Lectura de un archivo binario con información sobre los pulsos, el cálculo de determinadas operaciones matemáticas sobre los datos, la asociación de los mismos con sectores en el rango de lectura del radar, y el almacenamiento de los resultados en otro archivo binario para su posterior análisis.

2.3. Características de los Usuarios

El usuario necesita solo el archivo de pulsos para la ejecución del programa, y un mínimo conocimiento de uso de la terminal de Linux. Luego los resultados de la ejecución pueden ser observados por la misma terminal, o en un archivo de texto plano.

2.4. Suposiciones y Dependencias

- Se supone que la PC que va a ejecutar el programa posee una distribución Linux estándar.
- Para la ejecución del código en paralelo es necesario tener instaladas las librerías de OpenMP.
- El archivo con la información del radar debe estar en la misma carpeta que el binario compilado, tal como se suministra en el archivo del proyecto.

3. Requisitos Específicos

3.1. Interfaces Externas

Para interactuar con el programa, se utiliza la interfaz estándar (terminal). Por este medio se muestra el progreso de ejecución del programa, y las mediciones de tiempo, si son solicitadas.

3.2. Funciones

Las funciones que proporciona el producto son:

- Lectura de un archivo binario de longitud arbitraria, que contiene mediciones de un pulso emitido por el radar, con una estructura predeterminada.
- Asignación de estas mediciones a diferentes rangos en el alcance del radar.
- Cálculo del valor absoluto de las mediciones, y valor promedio de los mismos.
- Cálculo del vector de autocorrelación de estas mediciones promedio, por cada rango que discrimina el radar, o gate.
- Almacenamiento de la información calculada en otro archivo binario, con estructura también definida.

3.3. Requisitos de Rendimiento

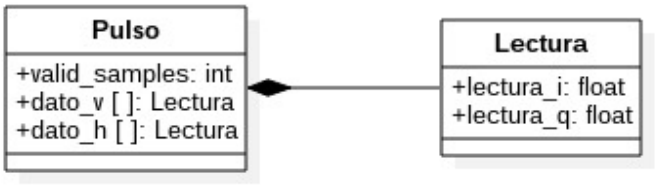
El proyecto debe funcionar en cualquier computadora con una distribución de Linux que tenga instaladas las librerias OpenMP para su ejecución en paralelo.
El tiempo de procesamiento en sí no es relevante, sino la ganancia de tiempo comparando la ejecución secuencial y paralela. Para ello debe medirse el tiempo de ejecución con herramientas de profiling.

3.4. Restricciones de Diseño

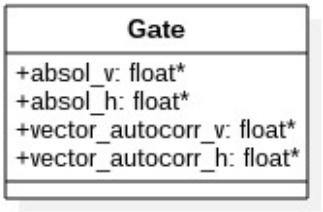
- El primer desarrollo del programa debe realizarse de manera procedural, resolviendo una parte del problema luego de la siguiente. Esto permite comprender mejor lo que se solicita.
- Luego debe realizarse un desarrollo explotando el paralelismo del problema, luego de haber identificado las partes que pueden realizarse concurrentemente sin acarrear errores, como corrupción de los datos o race conditions.
- El archivo con los resultados finales debe ser binario, y el formato de los datos guardados debe especificarse de antemano, para a futuro poder leerlo y analizar su contenido.
- Para la compilación es recomendable utilizar flags como -Werror, -Wall y -pedantic.

4. Diseño de solución

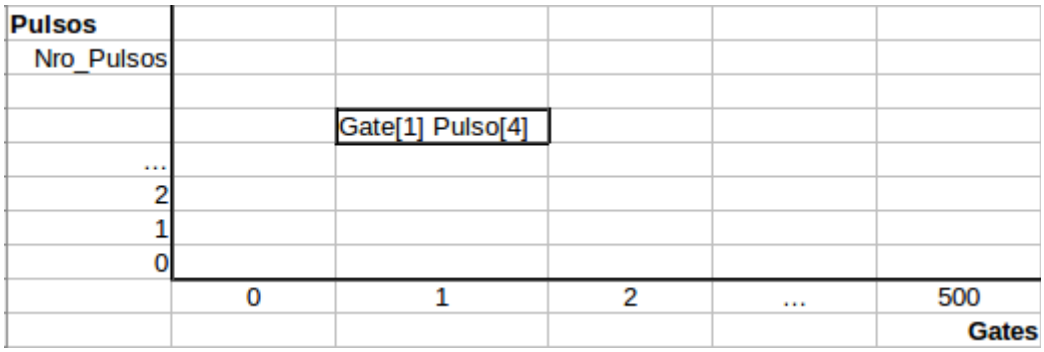
Para diseñar el software que cumpla estos requisitos, se comenzó con una estructura simple que pueda alojar los datos del archivo de pulsos. Esta estructura aloja la información de cada pulso, que consiste en mediciones de la polarización vertical, y horizontal. Además, cada medición es un número complejo, por lo que se almacena su valor real e imaginario.



Luego se pasa esta información a una nueva estructura, que discrimina los datos en función del gate al que corresponden, y no dependiendo el pulso. En otras palabras, en un primer momento se tenían los datos de cada pulso emitido, a lo largo de todos los gates. Después se invierte la información, para tener almacenado, por cada gate, la información que le corresponde, de cada uno de los pulsos emitidos.



En este momento se tiene entonces una matriz de pulsos y gates, y se debe diseñar un procedimiento para procesar la información.



En la figura se observa la matriz mencionada previamente. En el eje horizontal se sitúan los gates, y en el vertical los pulsos. Luego de reacomodar los datos, se tiene una estructura de gates, que alojan información de cada pulso. Cada celda contiene lecturas del radar que se asocian a un cierto gate, y corresponden a un cierto pulso.

Luego, para los cálculos siguientes, se computa el valor absoluto de cada conjunto de lecturas de los pulsos, para aprovechar la estructura por gate (por columnas). Estos valores absolutos son promediados por gate, con lo que cada celda del gráfico ahora contiene un solo valor. Cabe destacar que los valores de las lecturas de diferente polarización son tratados de manera independiente.

El paso siguiente es la autocorrelación de los datos por gate. Esto se realiza siguiendo la siguiente fórmula:

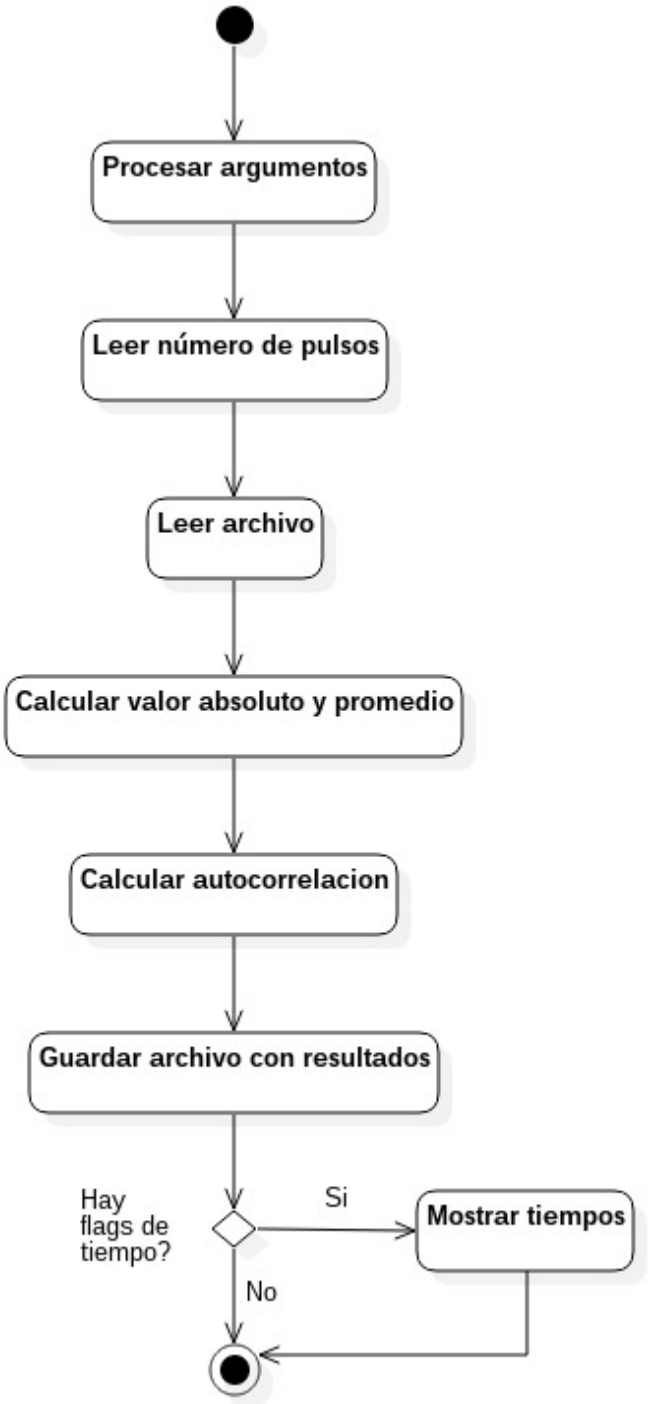
$$y(k) = \frac{1}{N} \sum_{n=1}^{N-k} x(n) * x(n+k)$$

De modo que el vector resultado tiene la misma longitud que el vector a correlacionar. Este resultado es almacenado también en las estructuras de tipo Gate, y nuevamente los valores con diferente polarización se toman como independientes.

Como último paso, se almacenan los resultados en un archivo binario. La estructura diseñada para tal fin es la siguiente, para un número de pulsos dado, y 500 gates:

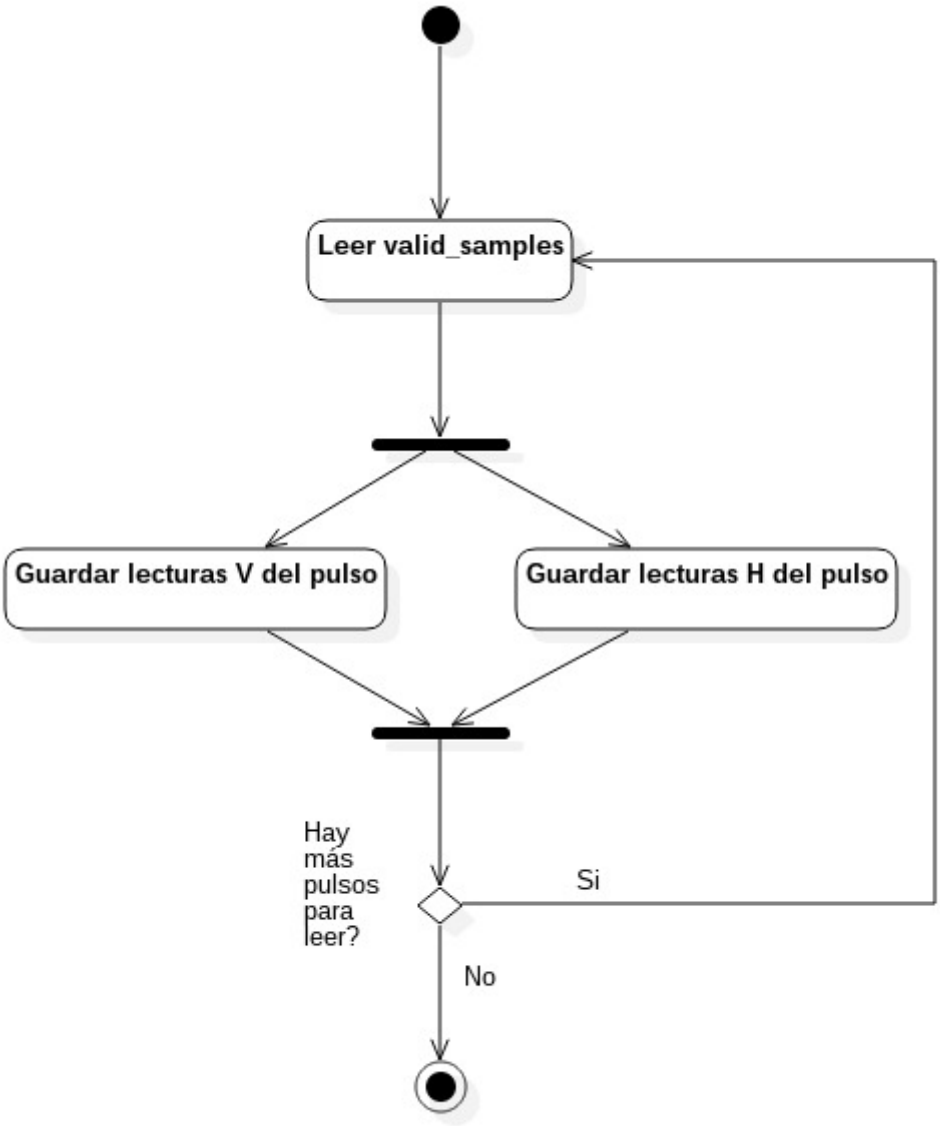
Nombre	**Tipo**	**Descripción**
Número de pulsos	uint16_t	Número de pulsos en el total del archivo
Número de gate (0)	uint16_t	Numero del gate actual
Correlacion_v[0]	float	Valor de correlacion para el pulso 0, polarización vertical
Correlacion_v[1]	float	Valor de correlacion para el pulso 1, polarización vertical
…		
Correlacion_v[Numero de pulsos]	float	Valor de correlacion para el último pulso, polarización horizontal
Correlacion_h[0]	float	Valor de correlacion para el pulso 0, polarización horizontal
Correlacion_h[1]	float	Valor de correlacion para el pulso 1, polarización horizontal
…		
Correlacion_h[Numero de pulsos]	float	Valor de correlacion para el último pulso, polarización horizontal
Número de gate (1)	uint16_t	Numero del gate actual
Correlacion_v[0]	float	Valor de correlacion para el pulso 0, polarización vertical
Correlacion_v[1]	float	Valor de correlacion para el pulso 1, polarización vertical
…		
Correlacion_v[Numero de pulsos]	float	Valor de correlacion para el último pulso, polarización horizontal
Correlacion_h[0]	float	Valor de correlacion para el pulso 0, polarización horizontal
Correlacion_h[1]	float	Valor de correlacion para el pulso 1, polarización horizontal
…		
Correlacion_h[Numero de pulsos]	float	Valor de correlacion para el último pulso, polarización horizontal
…		
Número de gate (500)	uint16_t	Numero del gate actual
Correlacion_v[0]	float	Valor de correlacion para el pulso 0, polarización vertical
Correlacion_v[1]	float	Valor de correlacion para el pulso 1, polarización vertical
…		
Correlacion_v[Numero de pulsos]	float	Valor de correlacion para el último pulso, polarización horizontal
Correlacion_h[0]	float	Valor de correlacion para el pulso 0, polarización horizontal
Correlacion_h[1]	float	Valor de correlacion para el pulso 1, polarización horizontal
…		
Correlacion_h[Numero de pulsos]	float	Valor de correlacion para el último pulso, polarización horizontal

El siguiente diagrama de actividades pretende dejar más claro el proceso que se diseñó para solucionar el problema planteado.
Consiste en una secuencia de funciones que leen datos de archivo, manipulan esos datos, y guardan un resultado en otro archivo.
Se anexó al final de la ejecución la opción de ver los resultados de la medición del tiempo de ejecución, mediante el paso de argumentos al ejecutar el programa.
Los flags de tiempo son seteados al procesar los argumentos del programa, al inicio de la ejecución.



Para la segunda parte del práctico, que consiste en analizar el paralelismo del problema y explotarlo para mejorar la performance, se realizó el siguiente procedimiento. Se analizaron las distintas funciones diseñadas, y se evaluó la capacidad de ejecución paralela de cada una. Esto tiene mucha relación con el hecho de que las iteraciones en los bucles, y otras subrutinas pueden ser ejecutadas en un orden arbitrario sin alterar el resultado final.

La rutina de lectura del archivo no fue paralelizada de por sí, en cambio se aprovechó la estructura de tipo pulso, y el hecho de almacenar datos de distintas polarizaciones por separado, para acelerar el proceso de iterar sobre la lista de floats leída, y asignarla a la estructura dada. El siguiente diagrama explica lo realizado:

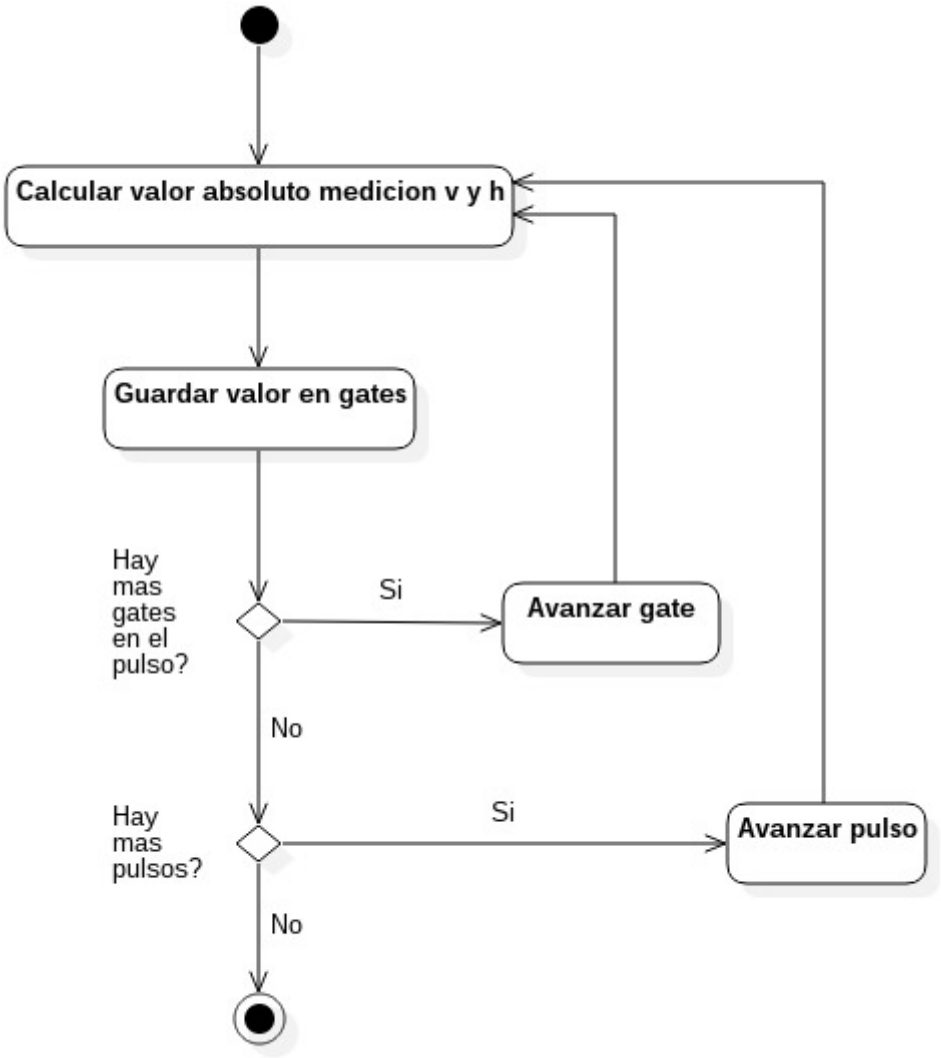


Se leen de una sola vez los valid_samples datos, y se divide el procesamiento y discriminación de los mismos para guardarlos en la estructura pulso en 2 secciones paralelas.

El cálculo de promedios y módulos también se paralelizó. Como se tienen datos de “n” pulsos, y se asignan a 500 gates, es posible procesar esos “n” pulsos

independientemente, y asignar las mediciones de cada uno al gate que corresponda sin importar el orden. En este paso se realiza el cálculo de los valores absolutos, y su promedio, para directamente guardar en las estructuras gate solo este valor, y economizar el uso de memoria. El siguiente diagrama muestra lo mencionado.

Cabe destacar que la completa ejecución de este diagrama puede hacerse en paralelo, por lo que en el mismo no se observará ninguna operación fork y join.



Luego, el cálculo de la autocorrelación fue mejorado de la misma manera. Como los gates son independientes entre sí, su vector de correlación puede calcularse en orden arbitrario, siguiendo la fórmula detallada anteriormente.

El guardado de los datos es un proceso netamente serial, por lo que no pueden aplicarse las mismas herramientas que en los casos anteriores. Un único hilo estará escribiendo posiciones de memoria correspondientes al archivo binario con los resultados.

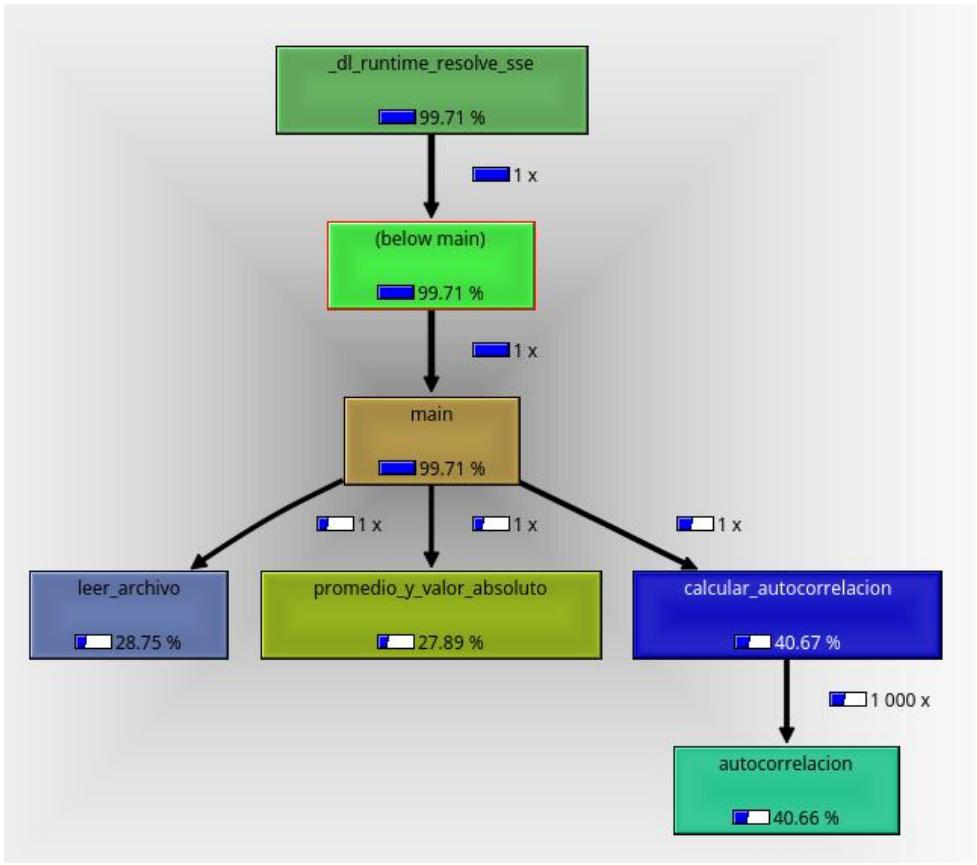
5. Implementación y Resultados

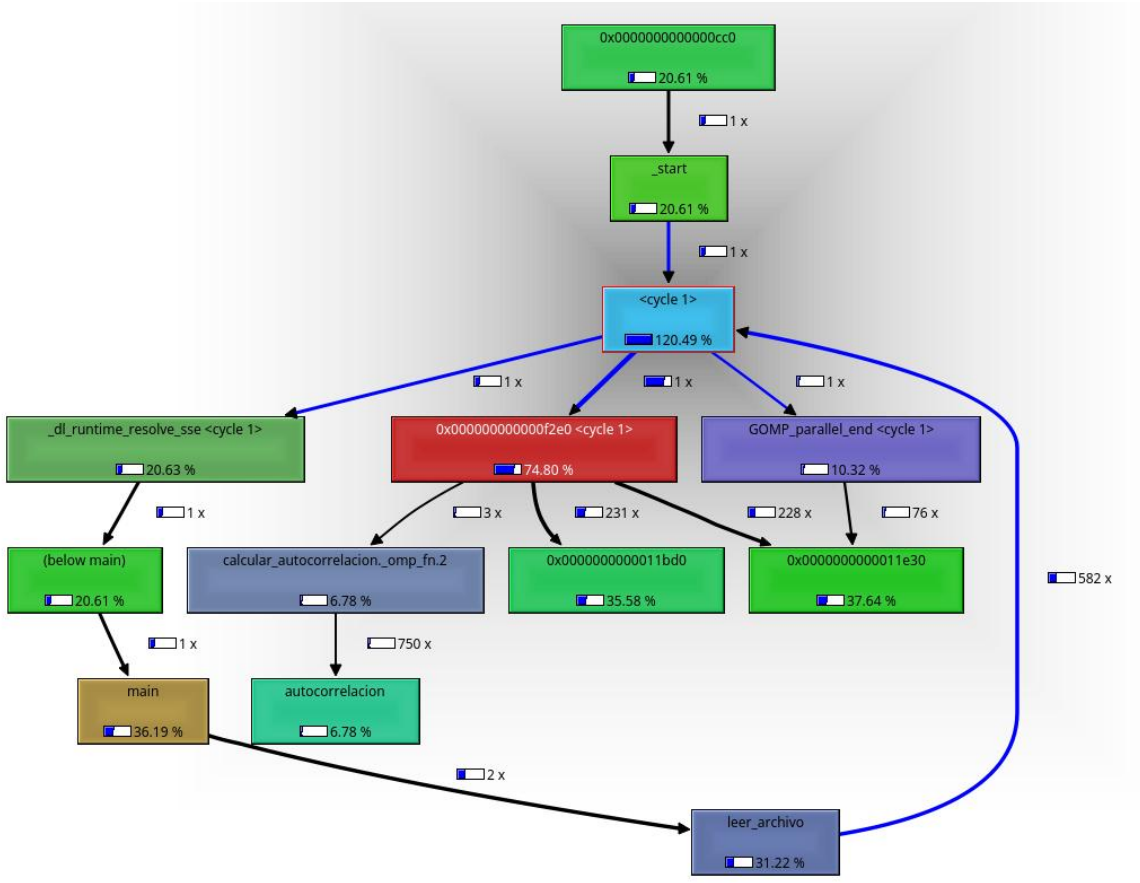
Para la implementación y desarrollo se utilizó el software Sublime Text. Se programaron las funciones de acceso y lectura al archivo binario, se comprobó que se estaba leyendo correctamente, y que el guardado de los datos en las estructuras explicadas era correcto.
Se estructuró el proyecto en dos mitades. La primera se centra en la ejecución monothread, con su main, sus funciones y archivos de headers. La otra mitad utiliza como base los códigos anteriores, y agrega la funcionalidad multihilo.

La ejecución del programa sigue los lineamientos generales descritos en el apartado de Diseño. Primero se lee el archivo buscando solo la cantidad de pulsos y el tamaño total del mismo, para poder acelerar la lectura del mismo luego.
Al leer efectivamente los datos de las tablas en el archivo, ya se sabe la información necesaria del mismo, y se itera llenando las estructuras de tipo Pulso. El tamaño del arreglo de pulsos es inicializado dependiendo de la cantidad de pulsos encontrada en el archivo en la primer lectura rápida.
Luego se inicializan las estructuras de tipo Gate, que contienen punteros a datos tipo float. El tamaño de este arreglo dinámico es creado en función también de la cantidad de pulsos leída en el archivo.
Una vez inicializado todo, se comienzan a calcular los valores absolutos y promedios, y se guardan en los gates, como se comenta en el apartado anterior.
Para el cálculo de la autocorrelación, se procede de a un gate por vez, computando los vectores V y H del mismo. Como los datos a guardar en el archivo a la salida son los de la correlación, la memoria reservada en los Gate para los promedios de valores absolutos no es necesaria, y se libera con free().
El guardado del archivo de resultados se lleva a cabo a continuación, siguiendo el formato descrito previamente.
Por último, si se ingresó algún flag, se ejecuta la acción correspondiente (guardar el tiempo de ejecución en un archivo, mostrarlo en terminal, o ambos).

Para la medición de tiempos de ejecución se utilizó la función clock() en el programa monothread, que devuelve el número de ticks de clock que ocurrieron desde que el programa inició. Con esta función, se utiliza un esquema de tipo “tick-tock” para calcular el tiempo que demora en ejecutarse.
En el programa multithread se usa un enfoque similar, pero con la función omp_get_wtime(), que devuelve el tiempo que transcurrió desde un momento dado. De nuevo es usado el mecanismo “tick-tock”, restando el valor de tiempo al final de la ejecución al valor al inicio.

Para el uso de profilers se utilizó Valgrind, junto con la utilidad Kcachegrind, para poder ver el call graph del programa, donde se muestra el porcentaje de tiempo que se invierte en cada subrutina, y la cantidad de veces que fue llamada cada una. Si bien esta herramienta no muestra tiempos, es una manera visual de entender donde se está perdiendo más tiempo en la ejecución. Se muestran a continuación capturas de pantalla del call graph, llamados sobre el proyecto single thread y multithread ejecutado con 4 hilos:


_dl_runtime_resolve_sse
99.71 %
1 x
(below main)
99.71 %
1 x
main
99.71 %
1 x
1 x
1 x
leer_archivo
28.75 %
promedio_y_valor_absoluto
27.89 %
calcular_autocorrelacion
40.67 %
1 000 x
autocorrelacion
40.66 %

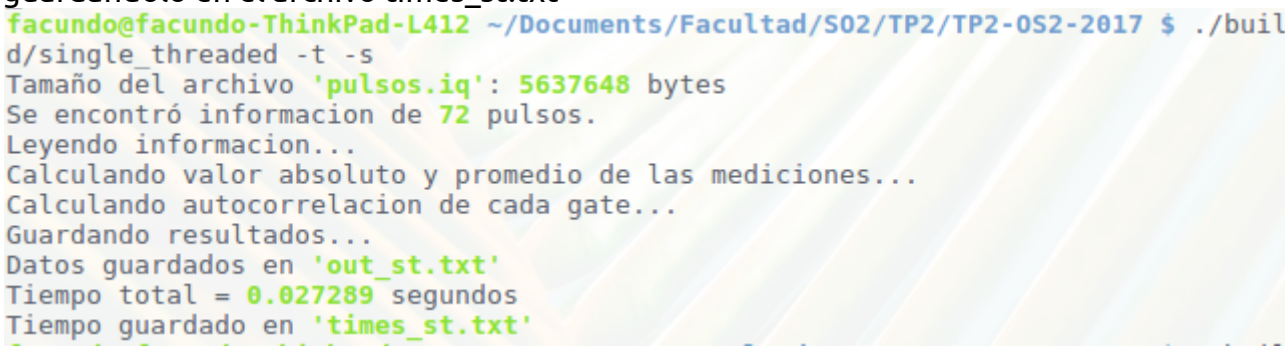

0x0000000000000cc0
20.61 %
1 x
_start
20.61 %
1 x
<cycle 1>
120.49 %
1 x
1 x
1 x
_dl_runtime_resolve_sse <cycle 1>
20.63 %
0x000000000000f2e0 <cycle 1>
74.80 %
GOMP_parallel_end <cycle 1>
10.32 %
1 x
3 x
231 x
228 x
76 x
582 x
(below main)
20.61 %
calcular_autocorrelacion._omp_fn.2
6.78 %
0x0000000000011bd0
35.58 %
0x0000000000011e30
37.64 %
1 x
750 x
main
36.19 %
autocorrelacion
6.78 %
2 x
leer_archivo
31.22 %

En la etapa de paralelización se utilizaron pragmas de OpenMP, tales como #pragma omp parallel for, y #pragma omp parallel sections para intentar acelerar la ejecución del programa. El número de hilos a utilizar es configurable, variando desde 1 a un tope máximo de 200 (también configurable desde los headers).

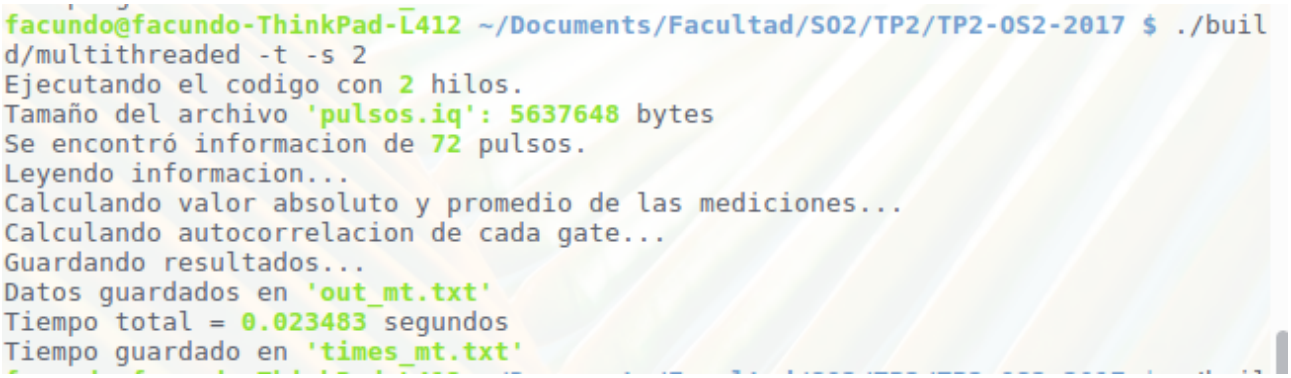
El código se encuentra documentado utilizando notación de Doxygen. La documentación se generó y se encuentra en la carpeta doc/.

Con respecto a la estructura general del proyecto, se hizo un Makefile que compile y realice el linkeo de los archivos, guardando los .o en la carpeta obj, y los binarios finales en la carpeta build. Se incluyó un target para ejecutar CppCheck sobre el proyecto, guardando los resultados en el archivo err.txt.
Luego de un make clean se eliminan todos los archivos objeto, binarios compilados y logs de CppCheck, dejando solamente archivos .c y .h, además del archivo pulsos.iq.
A continuación se adjuntan capturas de pantalla del programa en ejecución:

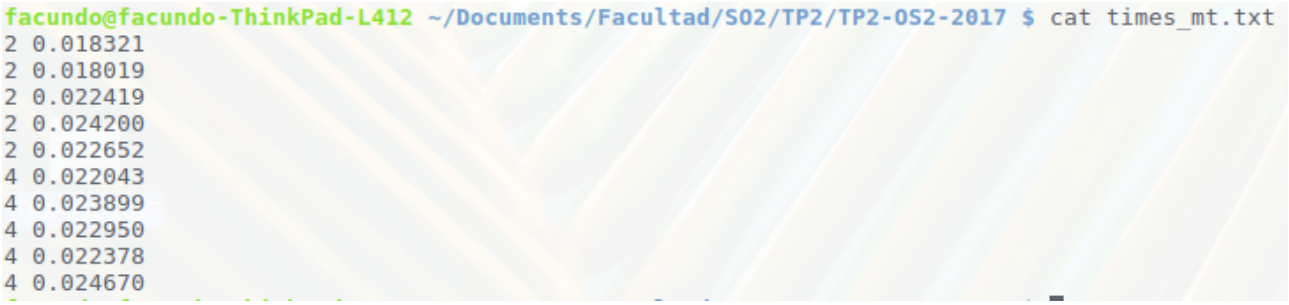
1) Ejecución del programa single threaded, mostrando el tiempo en terminal y guardándolo en el archivo times_st.txt



2) Ejecución del programa multithreaded con 2 hilos, mostrando el tiempo en terminal y guardándolo en el archivo times_st.txt



3) Cat del archivo con tiempos de ejecución del programa multithreaded. El primer valor es el número de hilos con los que se corrió, y el siguiente es el tiempo en segundos.

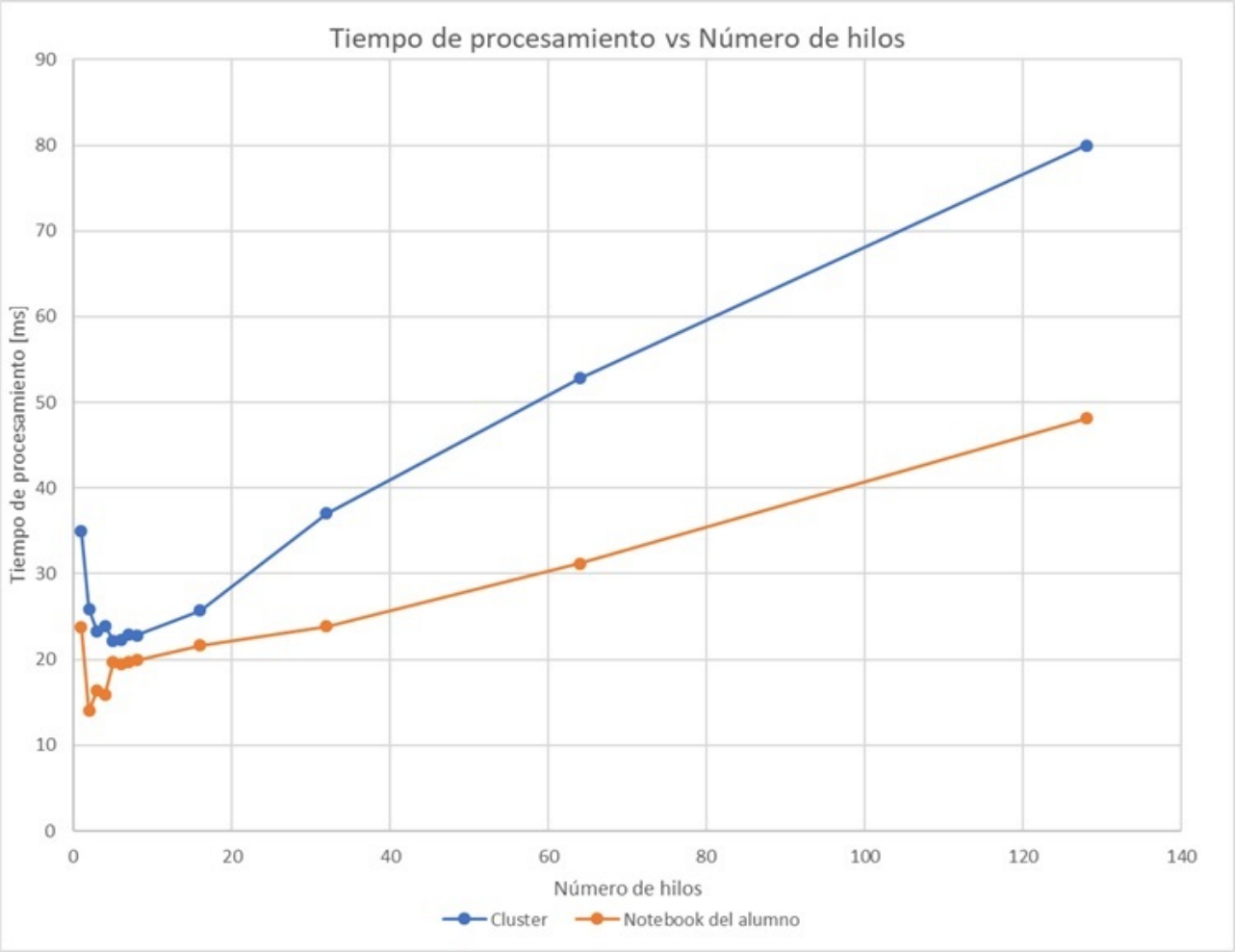


Los resultados de la ejecución en la notebook personal del alumno (Intel Core i5 560M, 4gb RAM DDR3, Linux Kernel 4.8.0) y en el cluster de la facultad, arrojaron los siguientes resultados:

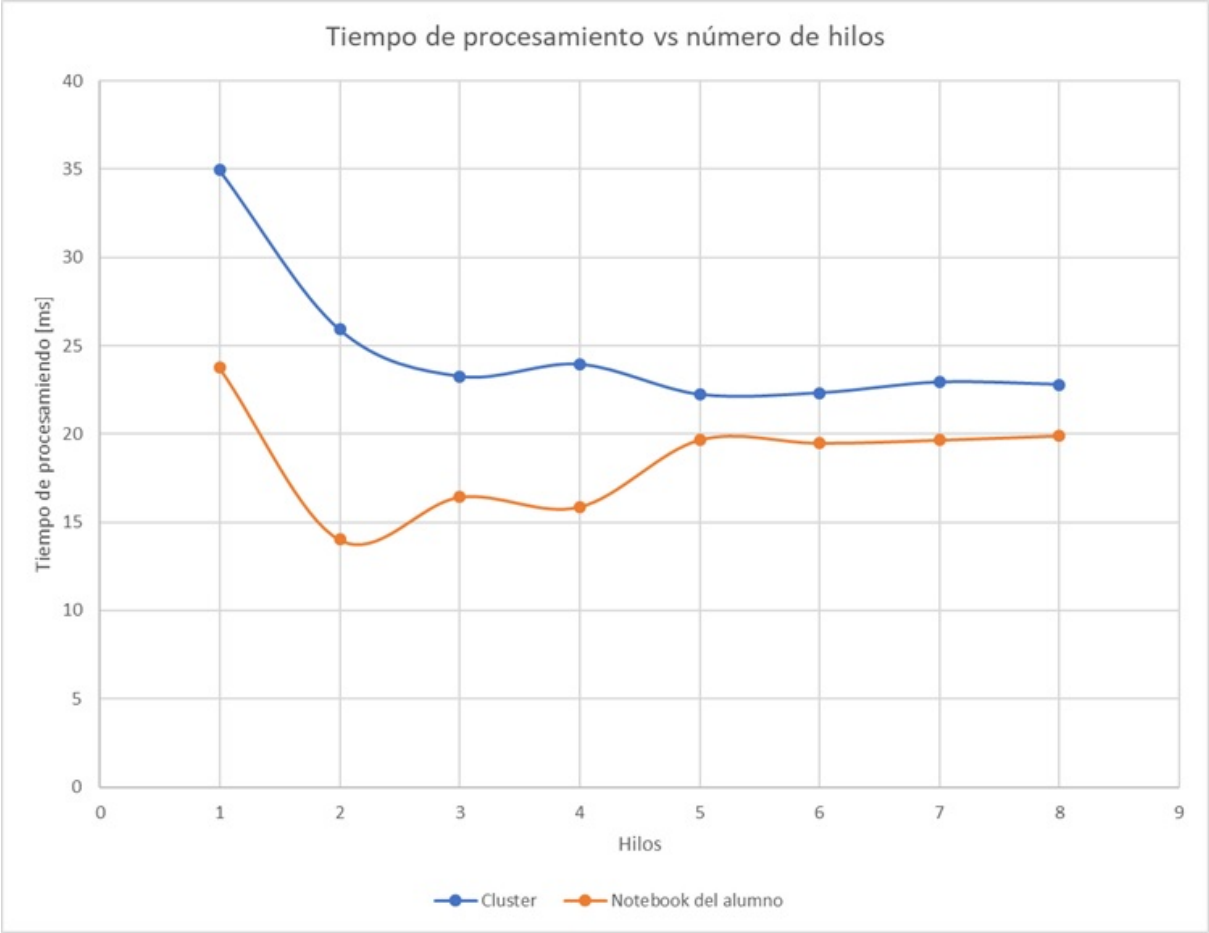
Para agilizar el proceso se programó un Bash Script que ejecute el código 100 veces en su modalidad monothread, y 100 veces multithread, variando el valor de los hilos usados. A continuación se muestran algunos gráficos y valores representativos.

A partir de las 100 muestras obtenidas por número de hilos, se calculó su media. Esto se realizó para las mediciones en el clúster y en la notebook personal del alumno. El número de hilos utilizado en las pruebas multithread fue: 2,3,4,5,6,7,8,16,32,64,128.

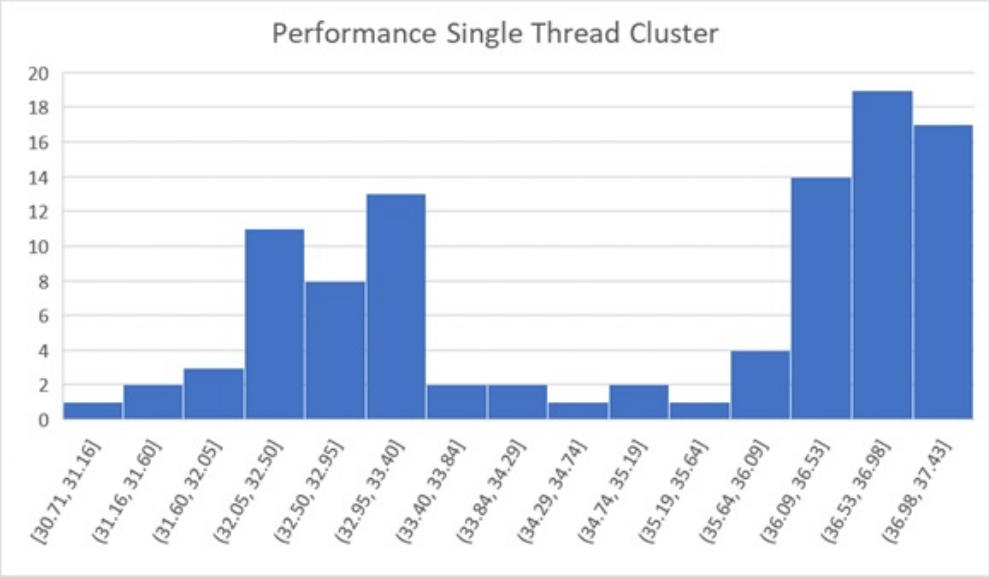
1) Este gráfico muestra el tiempo de procesamiento medio en función del número de hilos empleados, en el clúster de la facultad y la notebook del alumno. Puede verse que pasado el valle de máxima performance, los tiempos crecen junto con el número de hilos.
También puede observarse que los tiempos en la computadora personal son menores que en el clúster. Esto puede deberse a que el conjunto de procesadores del nodo está orientado al paralelismo a gran escala, y su poder de procesamiento por core no es tan elevado como en una cpu de propósito general, de consumo masivo.

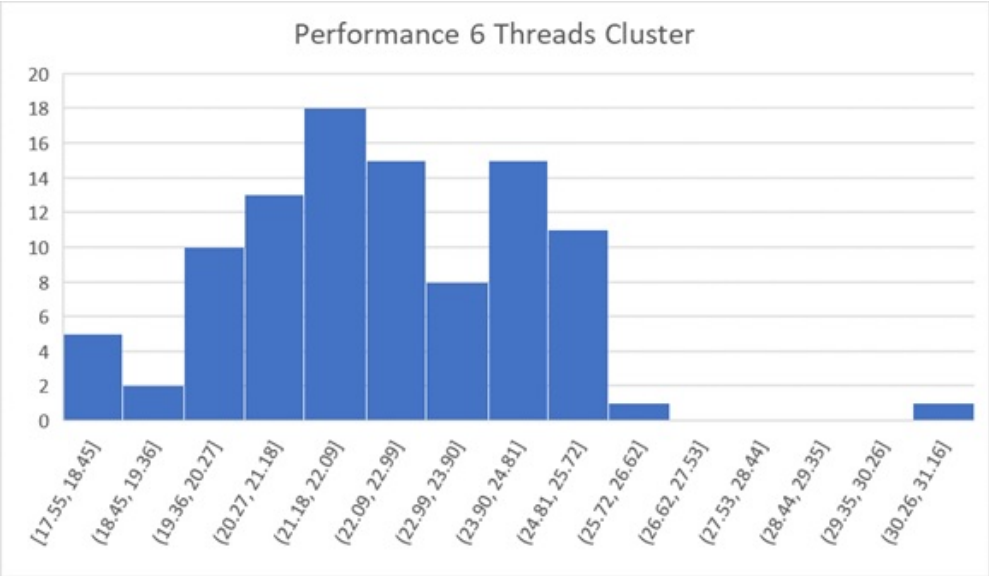


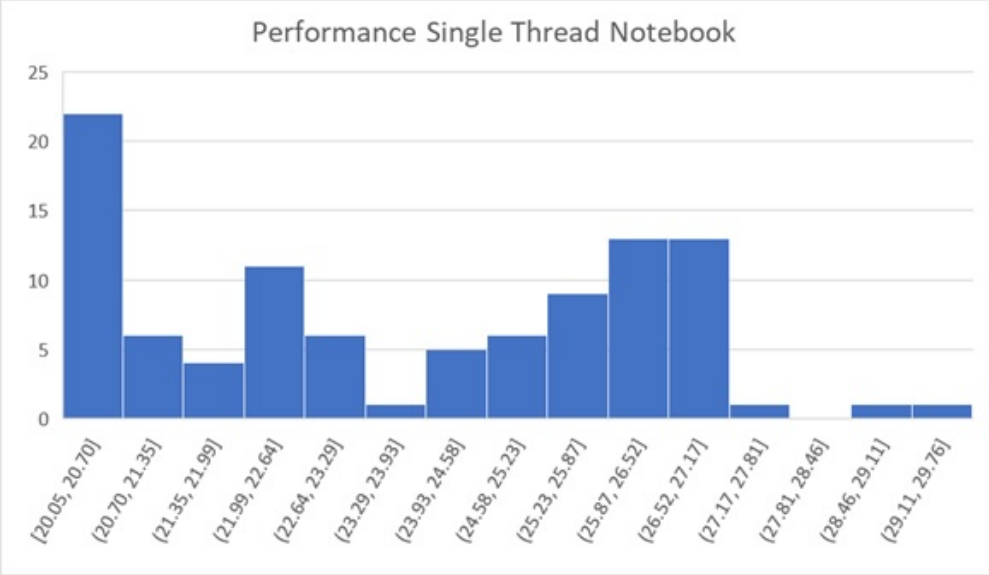
2) Este gráfico muestra los mismos valores que el anterior, pero haciendo énfasis en los tiempos que demora el programa con un número de hilos razonable: de 1 a 8. Aquí puede verse mucho más acentuado en la línea naranja (notebook) que al ejecutarse con un número de hilos múltiplo de la cantidad de cores de la máquina, se obtienen mejores resultados.
En el clúster este comportamiento no se deja ver tan claramente ya que se cuenta con 48 cores, versus 2 en la notebook del alumno.

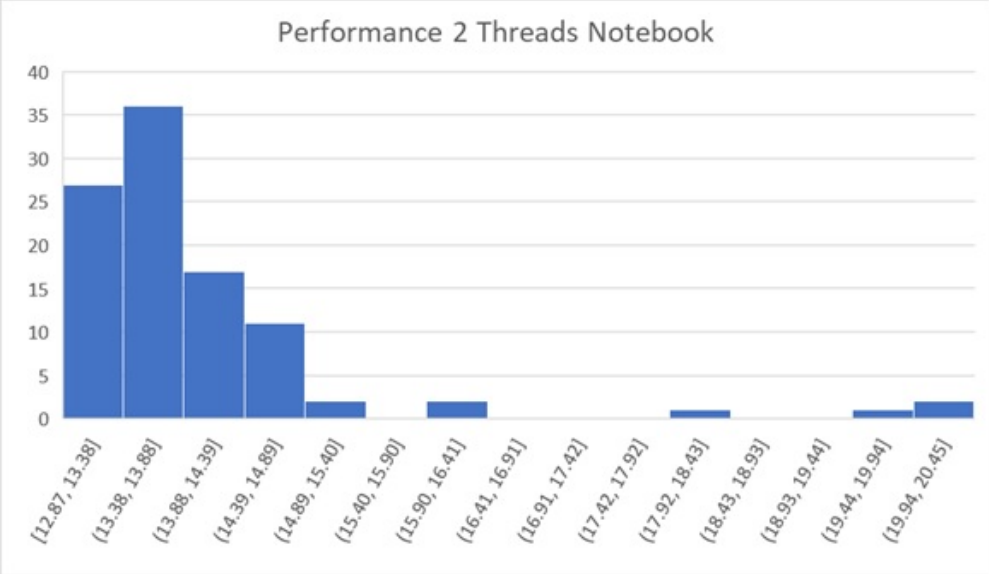


3) Los siguientes gráficos son histogramas que muestran como se distribuyen los tiempos observados en la ejecución del programa, en el nodo de la Facultad (single thread y 6 hilos), y en la notebook (single thread y 2 hilos).




Performance 6 Threads Cluster
20
18
16
14
12
10
8
6
4
2
0
[17.55, 18.45]
(18.45, 19.36]
(19.36, 20.27]
(20.27, 21.18]
(21.18, 22.09]
(22.09, 22.99]
(22.99, 23.90]
(23.90, 24.81]
(24.81, 25.72]
(25.72, 26.62]
(26.62, 27.53]
(27.53, 28.44]
(28.44, 29.35]
(29.35, 30.26]
(30.26, 31.16]


Performance Single Thread Notebook
25
20
15
10
5
0
[20.05, 20.70]
(20.70, 21.35]
(21.35, 21.99]
(21.99, 22.64]
(22.64, 23.29]
(23.29, 23.93]
(23.93, 24.58]
(24.58, 25.23]
(25.23, 25.87]
(25.87, 26.52]
(26.52, 27.17]
(27.17, 27.81]
(27.81, 28.46]
(28.46, 29.11]
(29.11, 29.76]


Performance 2 Threads Notebook
40
35
30
25
20
15
10
5
0
[12.87, 13.38]
(13.38, 13.88]
(13.88, 14.39]
(14.39, 14.89]
(14.89, 15.40]
(15.40, 15.90]
(15.90, 16.41]
(16.41, 16.91]
(16.91, 17.42]
(17.42, 17.92]
(17.92, 18.43]
(18.43, 18.93]
(18.93, 19.44]
(19.44, 19.94]
(19.94, 20.45]

Finalmente se muestran los valores medios obtenidos de las mediciones.

Hilos	Cluster	Notebook del alumno
1	34.9596	23.74391
2	25.9094	14.0242
3	23.2578	16.4316
4	23.9413	15.88012
5	22.232	19.6647
6	22.3216	19.48172
7	22.933	19.65828
8	22.7988	19.90021
16	25.6893	21.66765
32	37.0433	23.87132
64	52.8307	31.21168
128	79.9535	48.14678

6. Conclusiones

Como conclusión del proyecto, se aprendió a pensar un problema y diseñar una solución pensando en el paralelismo. Es muy importante esta etapa, ya que facilita la obtención de mejores resultados a la hora de la implementación.
Fue muy útil la lectura de blogs y sitios con información útil sobre OpenMP, que brindan ejemplos y tips para programar con la API.
Mejorar los tiempos fue un desafío, y si bien los resultados no son sobresalientes, pudo comprobarse que es posible, mediante un diseño correcto, la ejecución más rápida de un programa que explote el paralelismo del problema.
Con respecto a los resultados obtenidos, puede concluirse que una ejecución en paralelo es mucho más rápida si se diseña desde un principio la aplicación con ello en mente. Esto depende mucho del tipo de problema a resolver, y del hardware donde se quiera ejecutar, ya que un programa diseñado para explotar al máximo el paralelismo en una cpu para servers no estará diseñada de la misma manera que una pensada para aprovechar los 4 u 8 cores de una cpu de propósito general (un juego, por ejemplo).

7. Apéndices

A la hora de realizar este trabajo se utilizó contenido de las siguientes URL:

- https://www.dsprelated.com/showthread/comp.dsp/59527-1.php
- http://stackoverflow.com/questions/35026910/malloc-error-checking-methods
- http://jakascorner.com/blog/