CÁTEDRA DE SISTEMAS OPERATIVOS II

Departamento de Computación
FCEFyN - UNC

TP2- Programacion Distribuida

Gustavo Gonzalez

Índice

Introducción

Propósito del proyecto: El propósito es diseñar e implementar una aplicación que explotando el paralelismo y utilizando la librería ***openmp*** lea los datos del radar de un archivo binario(pulsos.iq), calcule la correlación de los mismos y escriba los resultados en otro archivo binario(correlacion.iq).

Ámbito del sistema: Se posee un radar Hidro-meteorológico que opera en Banda C (5,625 Ghz), al cual se le desea introducir un procesador Doppler que permite calcular la componente
radial del campo velocidad. La palabra radar, es un acrónimo de radio detection and ranging.
El funcionamiento de un radar, de forma muy resumida y simplificada, consiste en la emisión de una onda electromagnética, que recorre el espacio hasta encontrarse con un blanco. Este blanco, al ser inducido por la onda, se comporta como un espejo, que hace retornar la señal a la antena que emitió la onda. Esta señal se procesa y es
permite extraer información del blanco.
La antena gira sobre su eje horizontal en 360o (una vuelta completa) y la posición de la antena sobre este eje de denomina acimut. A su vez, puede moverse en su eje vertical, más comúnmente entre 0 y 90o, que se denomina elevación.
El radar con que se desea trabajar del tipo pulsado, posee un transmisor que genera pulsos electromagnéticos en doble polarización, es decir, transmite un pulso con polarización horizontal (0o) y otro con polarización vertical (90o), en forma
conjunta. Cada canal se denomina H y V respectivamente. A su vez, recordemos que una señal electromagnética es compleja, y esta se puede descomponer en sus componentes en fase y en cuadratura (I + jQ).
El radar posee una resolución de 0,5 km en rango, y un rango máximo de cobertura de 250km de radio. A cada componente de rango se le denomina gate.

Definiciones y abreviaturas:

-

Descripción del documento: El propósito de este documento es proporcionar a quien lo lea un entendimiento básico de los requisitos y

alcance de este proyecto así como los pasos que se tomaron para el diseño e implementación al proyecto propuesto.

Descripción General

Funciones del Producto:
La aplicación es un programa a correrse en líneas de comando y no provee flexibilidad en cuanto a opciones de ejecución ni interacción con el usuario.
Una vez compilado el programa se utiliza el comando *./radar* para ejecutarlo. La salida es un archivo binario llamado correlacion.iq con los resultados del calculo. En resumen el programa provee las siguientes funcionalidades:

- Cálculo de la correlación de los pulsos(en ambos ejes).
- Cálculo de la cantidad de pulsos en el archivo pulsos.iq.
- Escritura a un archivo binario para el almacenamiento y posterior utilización.
- Medición del tiempo de ejecución del programa.

Características de los Usuarios: El usuario es cualquier persona o empresa que posea el programa radar.exe y una computadora con alguna distribución de Linux(Las pruebas se hicieron en Ubuntu) y posea un conocimiento básico para manejarse con la consola ya que no se cuenta con una interfaz gráfica.

Restricciones: La versión actual cumple con todas las especificaciones, si bien optimizaciones de tiempo pueden ser implementadas dando como resultado un programa más rapido.

Suposiciones y Dependencias:
Se debe contar con el código fuente y el Makefile para poder compilar el programa. Además en la carpeta que contiene a source se debe tener el archivo *pulsos.iq* con los datos del radar para poder realizar los cálculos. El archivo *correlacion.iq* (donde se escriben los resultados) en caso de no existir en el sistema de archivos se crea en la misma carpeta que *pulsos.iq*.

Requisitos futuros:
Los principales puntos a cumplir son referentes a la optimización y mayor paralelización del código. Luego de codificar la versión serial del programa se descubrió que leyendo los datos siempre del archivo no se puede lograr el nivel de paralelización deseada. Para ello sería óptimo leer los datos en un buffer y luego acceder a ellos o utilizar la función mmap(de cualquiera de las formas se podría utilizar varios threads que trabajen al mismo tiempo en la lectura y cálculos de promedios sin una sobrecarga de I/o scheduling)

Requisitos Específicos

Requisitos de rendimiento: No se tienen requisitos específicos más que el de alcanzar el mayor nivel de paralelismo posible y el menor tiempo de ejecución.
Restricciones de diseño: Ninguna.

Diseño de solución
El programa cuenta con un solo modulo y sin interacciones complejas, por lo que no se complejizo el diseño y no se opto por ningún patrón de arquitectura especifico.

Implementación y Resultados
Para la implementación del programa se siguió la siguiente estructura:

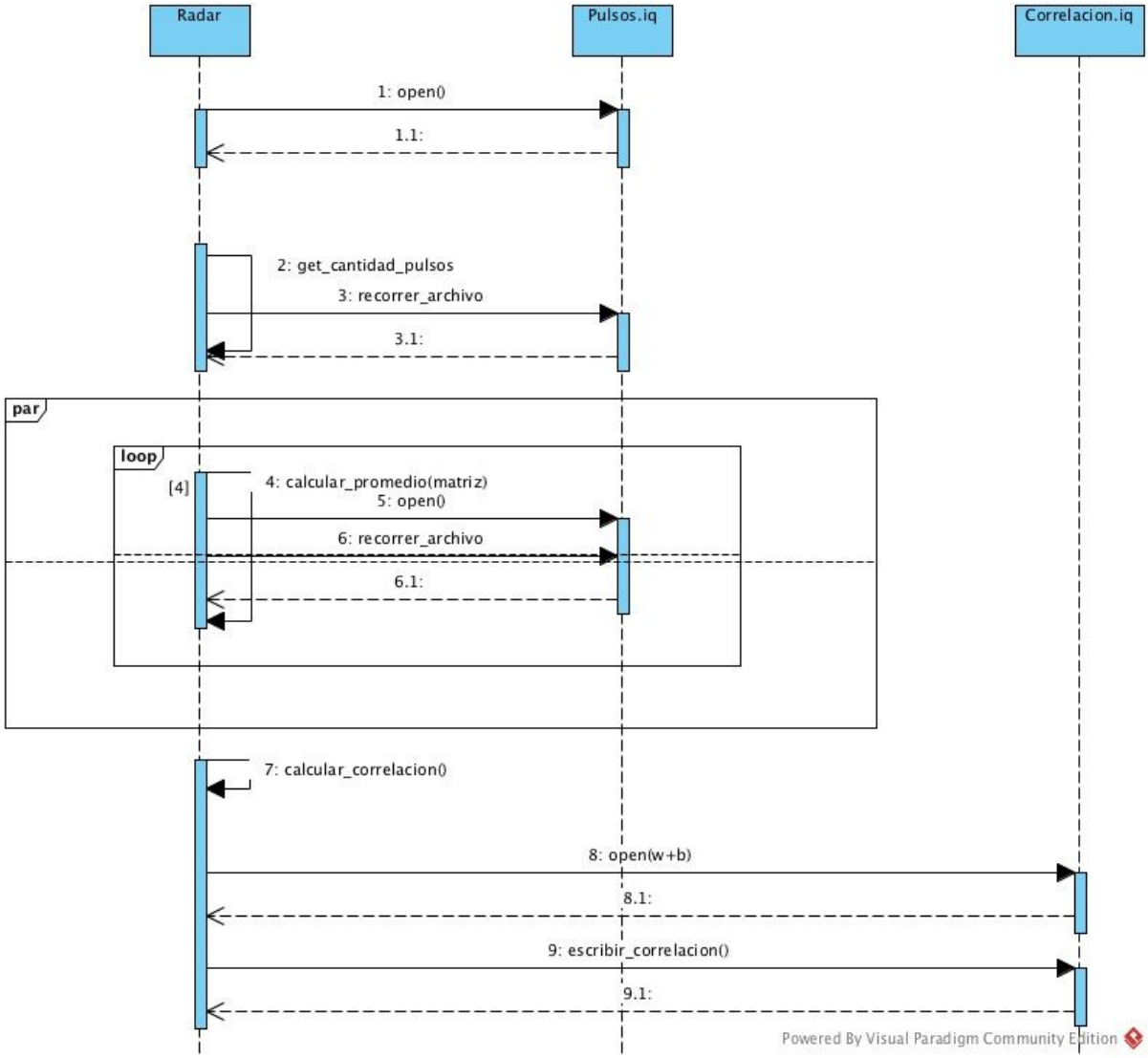
Primero se realizó la solucion serial del problema, dividiendo las tareas en funciones las cuales fueron implementadas en el archivo *radarHelpers.c* para luego ser llamadas desde el main.
Al iniciar el programa se llama a la función encargada de contar la cantidad de pulsos, luego se calculan los promedios de las matrices H_real, H_imaginaria, V_real, V_imaginaria mediante la función calcular promedio y en base a los resultados obtenidos se calcula la correlación para los pulsos horizontales y verticales respectivamente.
Como paso final se escribe los resultados en el archivo *correlacion.iq*

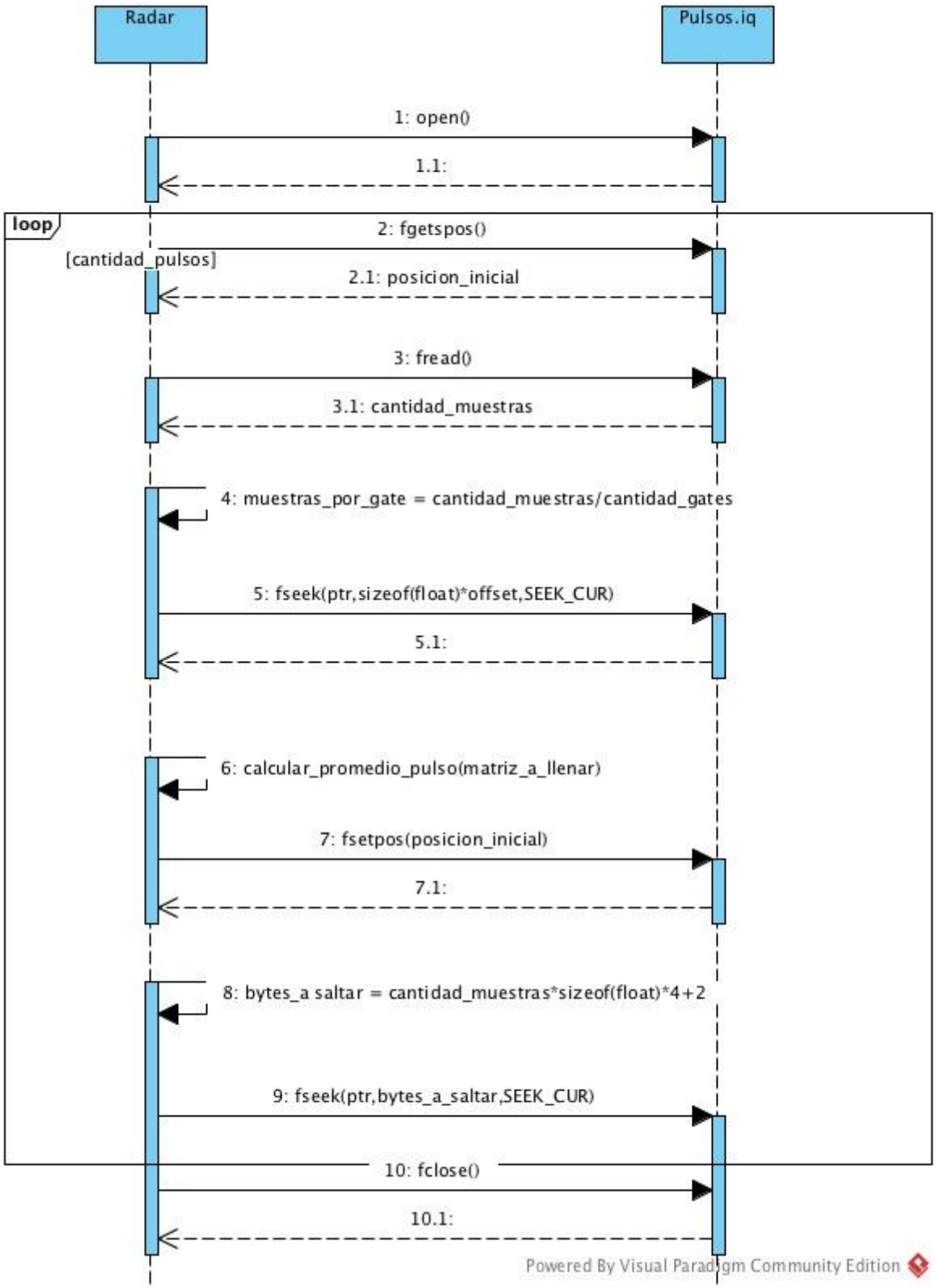
Primera aproximación para paralelizar: Como primera aproximación se observaron dos partes del codigo principal donde se podria explotar la concurrencia. Estas dos partes son el cálculo de promedios donde se llena cada una de las cuatro matrices mencionadas anteriormente y el calculo de los vectores de correlación para su posterior escritura. A continuación se muestra la implementacion:

```
    #pragma omp parallel sections
{
    #pragma omp section
    {
            calcular_promedio(path_pulsos,GATES,cantidad_pulsos,V_real,0);
    }

    #pragma omp section
    {
            calcular_promedio(path_pulsos,GATES,cantidad_pulsos,V_imaginario,1);
    }
    #pragma omp section
    {
            calcular_promedio(path_pulsos,GATES,cantidad_pulsos,H_real,2);
    }
    #pragma omp section
    {
            calcular_promedio(path_pulsos,GATES,cantidad_pulsos,H_imaginario,3);
    }
}

    #pragma omp parallel sections
{

     #pragma omp section
     {

calcular_correlacion(GATES,cantidad_pulsos,H_real,H_imaginario,H_correlacion);
```

```
        }
        #pragma omp section
        {

calcular_correlacion(GATES,cantidad_pulsos,V_real,V_imaginario,V_correlacion);
        }
    }
```

A continuación se muestra el diagrama de secuencias a alto nivel del programa para dar una idea general de su implementacion seguido de un diagrama de secuencia más especifico sobre una de las funciones principales(calcular_promedio)




Radar
Pulsos.iq
1: open()
1.1:
loop
[cantidad_pulsos]
2: fgetspos()
2.1: posicion_inicial
3: fread()
3.1: cantidad_muestras
4: muestras_por_gate = cantidad_muestras/cantidad_gates
5: fseek(ptr,sizeof(float)*offset,SEEK_CUR)
5.1:
6: calcular_promedio_pulso(matriz_a_llenar)
7: fsetpos(posicion_inicial)
7.1:
8: bytes_a saltar = cantidad_muestras*sizeof(float)*4+2
9: fseek(ptr,bytes_a_saltar,SEEK_CUR)
10: fclose()
10.1:
Powered By Visual Paradigm Community Edition

Conclusiones

El momento de las conclusiones llegó cuando se intentó paralelizar y se utilizó profilers para medir el tiempo y el consumo de memoria(encontrar memory errors y memory leaks).
La primera y más importante conclusión fue que en la primera región que se utilizó omp para paralelizar el cálculo de las cuatro matrices promedio se vio un deterioro en la performance. Al principio esto fue desconcertante, pero luego de varias pruebas con distinto número de hilos y de búsquedas en internet se llegó a la conclusión de que esto se debe a que los datos son leídos en cada ocasión (cada llamada a la función calcular_promedio) del archivo el cual está en el disco. Por lo tanto al aumentar el número de threads el overhead de hacer scheduling de I/o es mucho mayor a los beneficios obtenidos.
En el segundo caso (cálculo de los vectores de correlación) la mejora existe pero no es muy significativa, ya que la ganancia en tiempo se ve opacada por el overhead de crear un nuevo hilo.

Consideraciones de memoria: Mediante la herramienta se pudo encontrar un memory error el cual era ocasionado por una variable no inicializada, mediante esta herramienta también se pueden detectar los memory leaks(el programa no presentaba ninguno grave cuando se realizó el profiling).

Mediciones de tiempo

Para la comparación se ejecuto el codigo con dos hilos en el calculo de la correlacion 30 veces en la PC local y 30 en el cluster. La tabla de abajo muestra los resultados.
El tiempo se calculó usando la función *clock()* del sistema al inicio y final del programa.
Se consideraron otras opciones como el uso de profilers los cuales devuelven resultados más precisos y más detallados, pero valgrind no tiene una herramienta óptima para medir tiempo cuando se utilizan muchas operaciones de Entrada/Salida y el profiler de gnuprof arrojo siempre tiempos de ejecución nulos en todas las funciones(no se logró

detectar a qué se debe este error). Debido a lo mencionado anteriormente se optó por la opción más sencilla y usar la función clock().

Numero Ejecucion	***PC[ms]***	***Cluster[ms]***
1	242	170
2	241	180
3	240	180
4	243	180
5	244	180
6	247	180
7	250	180
8	248	180
9	246	180
10	248	190
11	245	180
12	250	190
13	243	170
14	250	170
15	241	190
16	241	180
17	244	180
18	243	170
19	244	180
20	245	180
21	243	180
22	244	180
23	242	180
24	244	190
25	242	180
26	248	180
27	242	180
28	253	170
29	244	180

30	247	180
Promedio	**244.8**	**179.6666667**

Apendices (Fuentes)

http://www.scadacore.com/field-tools/programming-calculators/online-hex-converter/
Convertidor hexa a binario,float,uint,etc
http://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf
Tutorial openmp
http://stackoverflow.com/questions/13421767/multithread-read-from-disk
Paralelismo en funciones E/S