CÁTEDRA DE SISTEMAS OPERATIVOS II

Departamento de Computación

FCEFyN - UNC

Trabajo Practico Nro2

Agustin Colazo

Cordoba, 2 de abril de 2017

Índice

Introduccion

A un radar Hidro-meteorológico que opera en Banda C (5,625 Ghz), al cual se le desea introducir un procesador Doppler que permita calcular la componente radial del campo velocidad.

Descripcion general

El software a construir tiene, como objetivo principal, leer los datos recolectados de los pulsos leidos por el ADC, los cuales se encuentran en formato binario en el archivo pulsos.iq. Luego, a partir de estos datos debera calcular varios gates y finalmente la autocorrelacion de la matriz (gates, pulsos) de las componentes vertical y horizontal de los pulsos. Finalmente, se debe generar un archivo de salida binario que contenga los resultados de los calculos.

Requerimentos

- Calcular distintos gates para cada pulso. Los cantidad de gates sera rango_maximo/resolucion.
- Los gates para cada pulso se obtendran tomando la media aritmetica de las muestras validas.
- Calcular la autocorrelacion de cada gate.
- Los calculos mencionados anteriormente se deben hacer para dos canales. No es necesaria la capacidad de reaccionar a variaciones en la cantidad de canales.
- La cantidad de memoria alocada para almacenar los datos no debe tener limites inferiores o superiores.
- La cantidad de memoria a alocar no debe tener un valor fijo, esta dependera de la cantidad de datos entrantes. El programa debe ser capaz de responder a cantidades de datos variables.
- No debe alocarse mas memoria de la necesaria. Toda memoria alocada debe ser adecuadamente liberada.
- El programa debe ser capaz de responder a distintas cantidades de pulsos en ejecuciones distintas.
- El programa debe ser capaz de analizar pulsos con distintas cantidades de muestras validas en una misma ejecucion.
- El tiempo de ejecucion del programa procedural para el conjunto de prueba debe ser menor a un segundo en la maquina pulqui.
- La paralelizacion debe otorgar una mejora de al menos 150% en el conjunto de prueba en el caso de 2 threads.
- Presentar un archivo de salida con los resultados de la autocorrelacion. Presentando todos los gates de un canal, y luego del otro. Ordenado por numero de gate.

- El formato del archivo de salida debe ser binario.
- En el programa procedural, el archivo de salida debe llamarse “resultados.iq”
- En el programa paralelo, el archivo de salida debe llamar “resultados_paralelo.iq”
- La salida del programa procedural y del programa paralelo debe ser la misma.
- El archivo de entrada debe llamarse “pulsos.iq” y debe ser un archivo binario.
- Capacidad de modificar la cantidad de threads que se utilizan con facilidad. Esto es, que la cantidad de threads este definida al principio del codigo fuente y que modificando su valor, se pueda compilar y utilizar esa cantidad de threads.
- Se debe llevar una cuenta de la cantidad de pulsos, aunque no es necesario mostrar este valor en la consola.

Diseño de la solucion

Se va a desarrollar una aplicación que hara uso de paralelismo para la resolucion del problema. Se descompone en dos el problema: la parte secuencial y la parte paralela.

La parte secuencial se hara con un solo thread en la cual se leera y escribira sobre los archivos correspondientes.

Para resolver la parte paralela se hara uso de la librería OpenMP y se paralelizara las operaciones. En la primera instancia de calculos se paralelizara sobre los pulsos, y en la segunda instancia se paralelizara sobre los gates.

En el primer caso se paraleliza sobre los pulsos por el modo en que son leidos los datos del archivo pulsos.iq. Como leemos por pulso y estos se los guardara en nodos. Habra un nodo por pulso, la mejor manera de procesar estos datos es haciendo la division por los distintos pulsos. En esta primera instancia se construiran matrices (gate, pulso), donde cada thread construira todos los gates de los pulsos que le corresponden. Esto es hacer el calculo de modulo y media aritmetica que corresponde a cada gate.

En el segundo caso, se dividira por gates. En C los datos contiguos corresponden a la misma fila en una matriz, mientras que en otros lenguajes como Fortran los datos contiguos son las columnas. Como las filas corresponden a los gates, para el segundo conjunto de operación se trabajara diviendo en gates. Cada thread se encargara de calcular la autocorrelacion de determinados gates. Cada thread se encargara de numero_de_gates/cantidad_de_threads de gates.

Se organizaran los datos de modo de que esten organizados de la mejor manera para evitar las fallas de cache. Esto es, datos que seran procesados por el mismo core deben ser contiguos.

Tambien se utilizara el flag -O3 para la optimizacion de la parte paralela. Un ejemplo de optimizacion es el alineamiento de la memoria.

La cantidad de threads que se utilizan dependera de la cantidad de datos a procesar, y de la cantidad de cores fisicos que haya disponibles. No tiene sentido utilizar mas threads que la cantidad de cores fisicos disponibles.

Implementacion

Se implementaron dos soluciones. Una secuencial y una paralela.

La primera implementacion fue secuencial. Normalmente al hacer un programa paralelo para procesar datos se hace una primera implementacion secuencial para poder comparar que la salida de ambos programas sea identica, y tambien para analizar si hay ganancia en rendimiento.

El objetivo de usar paralelismo es tener un mejor tiempo de respuesto, pero hay veces que programas paralelos tendran peor tiempo de respuesta que programas secuenciales. Esto se debe principalmente a dos cosas: una mala implementacion del programa paralelo y un conjunto de datos pequeño.

En el caso de un conjunto de datos pequeño esto se debe a que el paralelismo pierde eficiencia al suponer mayor fallas de cache. Las fallas de cache tienen un importante impacto en el tiempo de respuesta de los programas.

Tambien, al hacer una implementacion paralela, es importante que la salida del programa paralelo sea la misma que la del programa secuencial. Sino, el resultado es erroneo. Hay que tener cuidado con esto ya que es comun que las variables se corrompan en programas paralelos debido a un mal manejo de las variables o mala sincronizacion.

En la segunda implementacion se paralelizo algunas partes del codigo como se menciono en el diseño. Estas partes corresponden al procesamiento de datos. La lectura y escritura en archivos sigue siendo secuencial (ya que se trabaja con flujos de bytes).

Para paralelizar se trabajo con la librería OpenMP, y para la medicion del tiempo se uso una funcion de esta librería que no tiene gran impacto en el rendimiento del programa. Esta funcion es omp_get_wtime().

Para la paralelizacion se usaron 2 y 4 threads en la maquina local. Como esta tiene 2 cores fisicos no tiene sentido usar mas de 2 threads, pero se utilizo 4 threads para mostrar que al tener mas threads que cores fisicos se pierde rendimiento mas que ganarlo.

En el cluster, que cuenta con 12 cores fisicos, se implementaron 2, 4, 6 y 8 threads y se tomaron las medidas correspondientes.

Se tomaron 30 medidas de cada programa y se tomo promedios para compararlos graficamente. Tambien se calculo la ganancia de rendimiento que habia al usar las distintas cantidades de threads y se dibujo un grafico con estos datos.

En la parte paralela del programa se utilizaron las directivas

#pragma omp parallel private(variable 1, variable 2, ...) shared(variable a, variable b)

#pragma omp for

#pragma omp parallel: Establece la parte paralela del codigo.

Private(...): define las variables privadas de cada thread.

Shared(): define las variables compartidas entre los thread.

#pragma omp for: Las iteraciones del bucle siguiente a esta directiva se ejecutaran en paralelo. Se dividira las iteraciones entre los distintos threads.

Al utilizar paralelismo hay que tener cuidado con los datos compartidos y la dependencia de datos.

En las iteraciones que se dividen no hay dependencia de datos entre una iteracion y otra, por lo tanto no habra corrupcion de los datos, ni perdida de rendimiento por sincronizaciones. Pero estas iteraciones tienen otras anidadas dentro que si tienen dependecia de datos, pero estas iteraciones son hechas por un mismo thread.

Explicacion del codigo

La primera parte del codigo es secuencial, esta es la lectura de archivo de entrada. Los datos de cada pulso se guardan en nodos. Estos nodos tienen una variable y dos arreglos. La variable es para guardar el valor de muestras validas, un arreglo guarda todos los valores (reales e imaginarios) de un canal, y el otro arreglo guarda los valores del otro canal. Los nodos forman parte de una lista enlazada.

Como no se sabe cuantos pulsos hay en total, hay que hacer un primer recorrido y generar la lista enlazada. La memoria necesaria se va alocando en el proceso.

Una vez generada la lista, se crea un arreglo de punteros y se recorre la lista una vez, guardando la direccion de cada nodo en el arreglo instanciado.

Luego comienza la seccion paralela en el codigo, todos los calculos se hacen en esta seccion. En la seccion paralela hay dos bucles for “principales” (que pueden tener otros bucles anidados) que son divididos entre los distintos threads con la sentencia “#pragma omp for”. El primer bucle es el encargado de generar las matrices (gates, pulsos) y el segundo bucle calcula las autocorrelaciones de los gates.

Una vez realizado todos los calculos necesarios, termine la seccion paralela y se pasa a la ultima parte secuencial en que se escriben los resultados de las autocorrelaciones en un archivo de salida.

El programa secuencia es igual, excepto que no tiene las directivas correspondientes a la parte paralela. Por lo tanto, los bucles que se dividen entre distintos threads, en el caso secuencial, los hace un solo thread.

Resultados

A continuacion se mostrara un analisis del tiempo de respuesta de los programas en la maquina local y el cluster pulqui. Para esto se tomaron datos de la ejecucion de los programas en las distintas maquinas, se usaron distintas cantidades de threads en cada maquina y se recolectaron 30 muestras. Tambien se calculo la media aritmetica y el speedup de estos programas, con estos resultados se hicieron distintos graficos.

Cabe destacar que en el caso de la maquina local se tomaron muestras hasta para 4 threads. Ya que esta solo cuenta con dos cores fisicos, no iba a haber mejora en el rendimiento al tener mas que 2 threads. Se tomaron muestras para 4 threads (aunque se sabia que no iba a haber mejora) para mostrar que, el usar mas threads que la cantidad de cores disponibles, puede suponer un empeoramiento en el rendimiento del programa.

Tambien, mencionamos, que la programacion paralela aprovecha los cores fisicos, no los cores virtuales. Es importante tener esto en cuenta al diseñar un programa que haga uso del paralelismo.

En la programacion paralela el tamaño de las cache es importante. Si los datos entran en la cache es mejor utilizar programacion secuencial que programacion paralela, ya que la velocidad en que un core procesa todos esos datos, es mejor que dos o mas cores procesando pero con fallas de cache. Esto es asi porque la cache es mas lenta que los cores.

Por esto se considera el tamaño de las caches y el tamaño de los bloques de cache se tiene en consideracion por el alineamiento de la memoria (aunque esto se hace con el flag -O3).

Tiempo de ejecucion en maquina local

- Nombre del modelo: Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz
- Cores virtuales: 4
- Cores fisicos: 2
- L2 cache: 256K
- L3 cache: 3072K
- Tamaño de bloque de cache: 64 bytes

	1 Threads	2 Threads	4 Threads
1	0.036633	0.014497	0.019495
2	0.036595	0.014654	0.021005
3	0.041156	0.01451	0.021296
4	0.020182	0.013069	0.018073
5	0.041603	0.010806	0.01922
6	0.030165	0.014316	0.02142
7	0.039653	0.010484	0.021508
8	0.042037	0.014215	0.021277
9	0.041433	0.015263	0.016782
10	0.043322	0.01333	0.019316
11	0.042521	0.014387	0.017225
12	0.040599	0.014762	0.017248
13	0.039638	0.014118	0.019731
14	0.037245	0.016094	0.01663
15	0.043161	0.015671	0.018713
16	0.04206	0.014556	0.020568
17	0.046642	0.013201	0.020253
18	0.041734	0.014398	0.017813
19	0.042661	0.013345	0.015653
20	0.040828	0.012778	0.017733
21	0.039787	0.01445	0.018088
22	0.040165	0.015408	0.016012
23	0.043377	0.015434	0.017977
24	0.037209	0.015235	0.019117
25	0.040501	0.01626	0.019386
26	0.03412	0.016065	0.017089
27	0.04116	0.015741	0.019218
28	0.041251	0.01461	0.023307
29	0.037951	0.013358	0.018301
30	0.041154	0.014109	0.018982
Promedio	0.0395514333	0.0143041333	0.0189478667

Tiempo de ejecucion en cluster pulqui

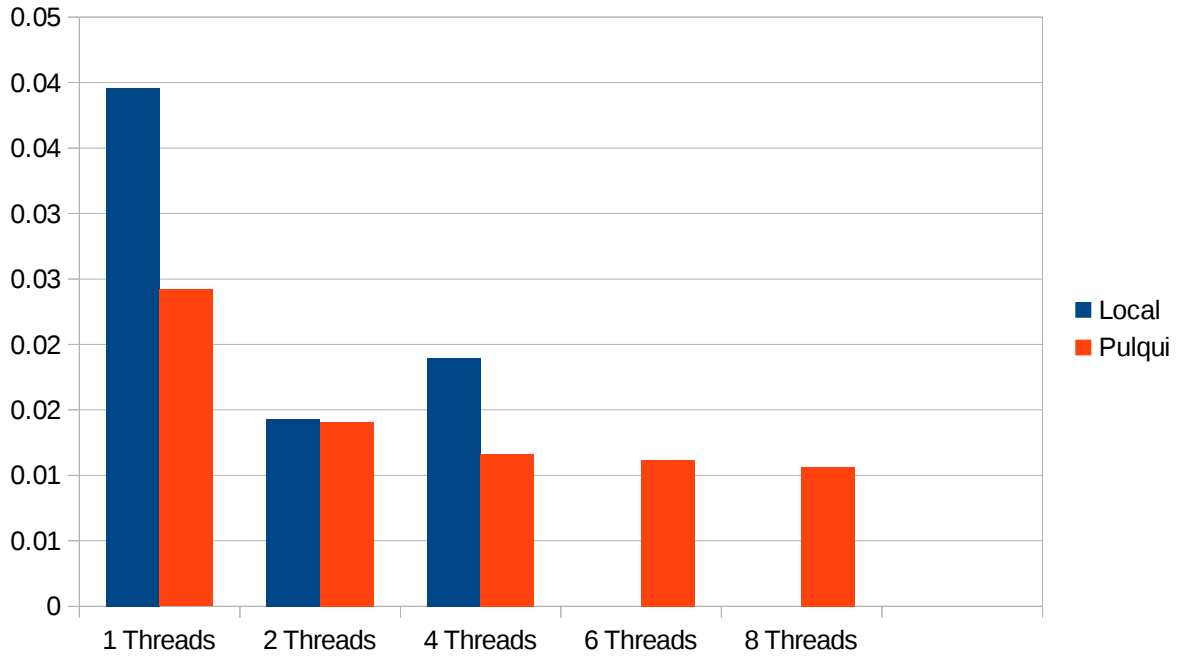
- Nombre del modelo: Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30GHz
- Cores virtuales: 24
- Cores fisicos: 12
- L2 cache: 256K
- L3 cache: 15360K
- Tamaño de bloque de cache: 64 bytes

	1 Threads	2 Threads	4 Treads	6 Threads	8 Threads
1	0.026435	0.01465	0.011064	0.011015	0.01161
2	0.023679	0.01448	0.012705	0.011623	0.009717
3	0.025389	0.012511	0.012429	0.011943	0.011414
4	0.022091	0.014706	0.012595	0.010534	0.009445
5	0.023131	0.013704	0.010784	0.011913	0.011116
6	0.023033	0.014267	0.008483	0.011678	0.010051
7	0.024984	0.013999	0.011988	0.011178	0.012026
8	0.023695	0.014414	0.011442	0.011061	0.009563
9	0.023163	0.014486	0.012294	0.011658	0.009401
10	0.022749	0.013376	0.008915	0.011494	0.01112
11	0.022286	0.014617	0.012391	0.01052	0.011514
12	0.024594	0.014536	0.012136	0.010952	0.01091
13	0.025513	0.013341	0.011599	0.011864	0.010232
14	0.022673	0.014511	0.012356	0.011873	0.009205
15	0.022321	0.014675	0.012277	0.011555	0.011821
16	0.024995	0.014507	0.012372	0.009094	0.008019
17	0.023816	0.015174	0.010417	0.011154	0.011688
18	0.025588	0.0145	0.010634	0.011642	0.010458
19	0.025053	0.013619	0.012734	0.010876	0.01135
20	0.022808	0.014437	0.011463	0.010492	0.010028
21	0.024993	0.014137	0.012027	0.011703	0.011651
22	0.024585	0.013996	0.01214	0.011566	0.01136
23	0.023603	0.014529	0.012163	0.011565	0.011066
24	0.022647	0.014484	0.012062	0.01044	0.009637
25	0.024077	0.013548	0.011254	0.010176	0.011701
26	0.027731	0.012844	0.011876	0.01172	0.011535
27	0.025984	0.014385	0.009984	0.010935	0.009957
28	0.024479	0.014258	0.012484	0.01162	0.010818
29	0.027867	0.014491	0.012009	0.011603	0.011442
30	0.021883	0.011057	0.012217	0.009944	0.009501
Promedio	0.0241948333	0.0140746333	0.0116431333	0.0111797	0.0106452

Promedios

A continuacion se muestra una tabla con los promedios de los tiempos de ejecucion en la maquina local y en el cluster pulqui, para los distintos threads.

	1 Threads	2 Threads	4 Threads	6 Threads	8 Threads
Local	0.0395514333	0.0143041333	0.0189478667		
Pulqui	0.0241948333	0.0140746333	0.0116431333	0.0111797	0.0106452



Se observan varias cosas.

Primero, observamos que en la maquina local al usar 4 threads empeora el rendimiento respecto de 2 threads. Esto se debe a que esta dispone de dos cores fisicos.

Segundo, podemos notar que el tiempo de respuesta cuando se usa 1 thread en la maquina local es considerablemente mas lento que en pulqui, pero al usar 2 threads las velocidades son similares. Esto puede deberse al tamaño de las cache en la maquina local.

Finalmente, en el tiempo de respuesta del programa paralelo en el cluster pulqui, al utlizar mas que 2 threads, no se observan mejoras considerables. Esto puede deberse al tamaño del conjunto de datos. Normalmente, al tener mas datos sobre los que operar, el uso de mas threads tiene mayor impacto.

En caso de que sea posible que el conjunto de datos a procesar sea mayor que el de prueba, hay que considerar hacer otro analisis estadistico con un conjunto de prueba mayor para observar si el uso de mas threads mejora el rendimiento del programa. Y si se justifica el uso de estos.

Mejora en rendimiento (%)

El speedup potencial según la ley de amdahl se calcula:

Speedup = 1/(P/N + S)

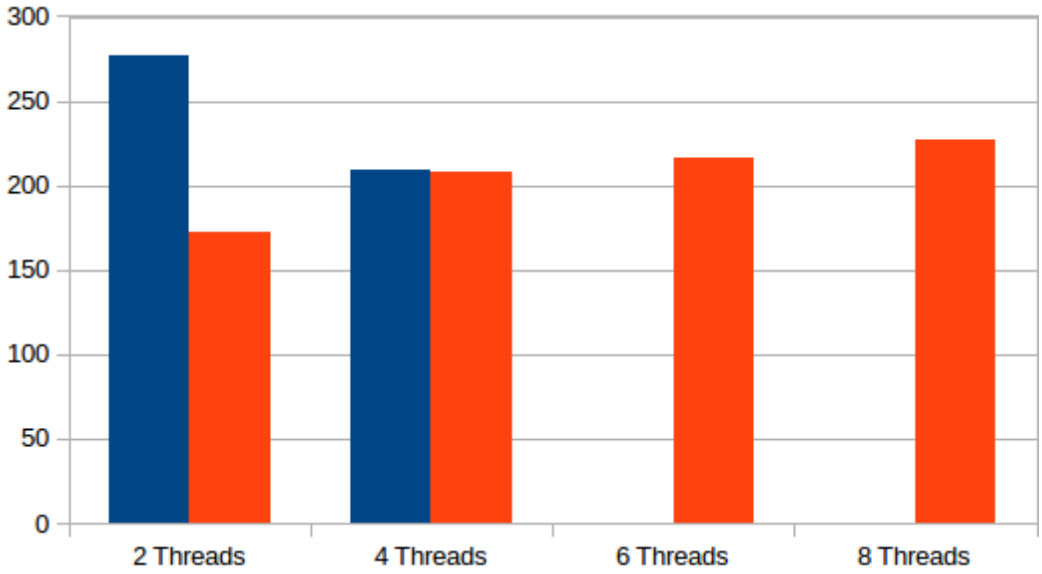
P: Fraccion paralela del codigo

S: Fraccion secuencial del codigo

N: Numero de cores fisicos.

Supongamos el caso en que usamos 2 cores. El speedup potencial seria de 2 en caso de que todo el codigo sea paralelizable (cosa que cabe mencionar que no es asi). Esto supondria que el programa seria el doble de rapido.

	2 Threads	4 Threads	6 Threads	8 Threads
Local	276.5035281177	208.7381868847		
Pulqui	171.9038269795	207.8034549692	216.417554436	227.2839714926



Podemos notar que la mejora en la maquina local al usar 2 threads es mayor al 200% (de 277%). Esto puede deberse a que la ejecucion procedural tenga muchas fallas de cache, lo que provoca que el tiempo de ejecucion sea lento. En la maquina pulqui, la mejora en el rendimiento al usar 2 threads es de 172%, la cual es considerablemente menor, pero aun es buena.

Lo anterior se debe a que la ejecucion procedural es considerablemente mejor en la maquina pulqui que en la local. Esto se puede deber a que la maquina pulqui tiene cache de mayor tamaño. Por lo tanto, la ejecucion procedural es mas eficiente en pulqui. Lo que supone que al utilizar mas threads la mejora potencial es menor.

Como se menciono en la seccion anterior, hay perdida de rendimiento al utililizar 4 threads en la maquina local, ya que se excede la cantidad de cores fisicos.

Tambien podemos notar que la mejora en el rendimiento al usar mas que 2 threads en la maquina pulqui no es muy significativa, esto se puede deber al tamaño del conjunto de datos.

Comparacion de herramientas de profiling

Se evaluan tres herramientas de profiling: gprof, valgrind y gperftools.

Gprof usa un enfoque hibrido de instrumentacion asistida por el compilador y de muestreo.

La instrumentacion es usada para obtener informacion sobre las llamadas a funciones: para poder generar graficos de llamadas a funciones y contar las llamadas. Para obtener informacion de perfilado en tiempo de ejecucion, se usa un proceso de muestreo. Un contador de programa se aumenta en intervalos periodicos interrumpiendo el programa en ejecucion con interrupciones del sistema operativo. Al ser un proceso estadistico, los datos perfilados no son exactos, sino que son una aproximacion.

Para poder crear un perfil de CPU de la aplicación con gprof hay que compilar el codigo fuente con los flags correspondientes.

Gprof muestra un perfil plano y un grafico de llamdas. El perfil muestra el tiempo de ejecucion total en cada funcion y su porcentaje del tiempo total de ejecucion. Tambien muestra contadores para las llamadas de funcion. El grafico muestra quien llamo a cada funcion, y quien llama a cada funcion.

El overhead de Gprof puede ser bastante grande. Ademas Gprof no tiene soporte para aplicaciones multi-hilo, ni para perfilar librerias compartidas.

Valgrind es un framework de instrumentacion para construir herramientas de analisis dinamico. Para el perfilado de rendimiento se usa una herramienta que se llama callgrind. Esta registra las llamadas de funcion y lo representa en un grafico. Para perfilar con valgrind no hace falta compilar el programa con flags especiales. Se puede ejecutar cualquier ejecutable con valgrind. Valgrind ralentiza mucho la ejecucion del programa, lo que no lo hace una buena opcion cuando hay que usar aplicaciones grandes o hacer mediciones de tiempo.

Gperftools es un conjunto de herramientas provisto por google para analizan y mejorar el rendimiento de aplicaciones multi-hilo. Ofrecen un perfilado de CPU, un perfilado de la pila, un detector de fuga de memoria y una implementacion de malloc de alto rendimiento para multi-hilos.

En el caso de querer perfilar secciones del codigp con gperftools hay que compilar con ciertos flags. Ademas hay que seleccionar las partes del codigo que se desea perfilar con las sentencias ProfileStart(“nombre.log”) y ProfileStop().

Por otro lado, se puede perfilar toda la aplicación sin hacer cambios, ni compilar con flags adicionales.

Este perfilador puede perfilar aplicaciones multi-hilo. Ademas el overhead en tiempo de ejecucion es muy bajo, lo cual lo hace mas adecuado para mediciones de tiempo.

Gprof es una herramienta antigua, con sus bases en los '80. Tiene soporte limitado para aplicaciones multi-hilo y para librerias compartidas. Ademas genera un overhead importante en tiempo de ejecucion. Esto lo hace inadecuado para las aplicaciones actuales.

Valgrind da los resultados mas precisos y soporta aplicaciones multi-hilo. Es facil de usar y se puede usar KCachegrind para visualizar y analizar los resultados del perfilado. El problema es que ralentiza la ejecucion de los programas, lo cual lo hace inadecuado para aplicaciones grandes, aplicaciones con tiempos estrictos o mediciones de tiempo.

Gperftools tiene poco overhead en tiempo de ejecucion. Ademas, se pueden seleccionar areas de interes del codigo a ser perfiladas, y tiene soporte con aplicaciones multi-hilo. Tambien se puede usar Kcachegrind para analizar los datos de perfilado. Este perfilador esta basado en muestreo, y al ser un proceso estadistico genera resultados con cierta impresicion. No es tan preciso como Valgrind. Pero si es necesario mejorar la precision del perfilado, se puede aumentar la frecuencia de muestreo cuando sea necesario.

Por ultimo, se mencionaran las funciones omp_get_wtime() y omp_get_wtick() de OpenMP. Las cuales son recomendadas para hacer mediciones de tiempo al usar esta librería, ya que son implementadas por la misma y tienen poco overhead (ignorando el overhead que provoca la creacion de hilos, este overhead se tiene siempre que se usa paralelismo). Omp_get_wtime() se utiliza para obtener una medida de tiempo(en segundos) relativa a un valor pasado. Para calcular el tiempo transcurrido con omp_get_wtime() se toma la diferencia del valor devuelto por esta funcion en dos momentos distintos. Omp_get_wtick() sirve para obtener la precision del temporizador ya que devuelve la cantidad de tiempo que pasa entre dos ticks del reloj.

Conclusion

En este trabajo se aprendio sobre las distintas herramientas de perfilado que hay disponibles y las distintas funcionalidades que ofrece cada una. El uso de estas herramientas es importante en el desarrollo de software para analizar la fuga de memoria, el uso de la pila, entre otras cosas.

Tambien se hizo un analisis estadistico sobre el tiempo de ejecucion de programas secuenciales y con varios hilos. Pudiendo observar las mejoras que supone usar varios hilos y hasta que punto. Ademas de las dificultades que presenta el uso de paralelismo. Esto dejo en claro la importancia de analizar los programas que uno desarrolla, para comprobar si cumplen con los requisitos planteados.

Se observo la importancia de la fallas de cache en los programas paralelos, la independencia de los datos y el tamaño del conjunto de datos. Tambien, no siempre usar mas hilos va a suponer una mejora en el tiempo de ejecucion del programa. El uso de mas hilos puede incluso suponer un empeoramiento si el conjunto de datos no es suficientemente grande, como se pudo observar en este trabajo.

Hay que tener cuidado al seleccionar la cantidad de hilos que se usara, tanto por el tamaño del conjunto de datos a procesar, como por la cantidad de cores fisicos disponibles. Si usamos mas threads que la cantidad de cores disponibles, puede suponer una baja en el rendimiento del programa.

Ademas por la ley de amhdal podemos ver que la mejora potencial de un programa esta acotada por la fraccion del programa que es paralelizable.

Fuentes

http://gernotklingler.com/blog/gprof-valgrind-gperftools-evaluation-tools-application-level-cpu-profiling-linux/

https://github.com/gperftools/gperftools

https://stackoverflow.com/questions/1168766/what-is-a-good-easy-to-use-profiler-for-c-on-linux

https://computing.llnl.gov/tutorials/parallel_comp/

https://computing.llnl.gov/tutorials/openMP/