Informe práctica 3

Paralelización función SAXPY iterativa

Autor: Santiago Escobar

1. Introducción

“Single-Precision A*X Plus Y” o SAXPY en un función básica del Álgebra lineal, la cual es usada frecuentemente para el desarrollo de métodos numéricos en el procesamiento de señales, etc.
En esta función se multiplica un escalar “a” por un vector “X” y esto es sumado a otro vector “Y”.El resultado de la operación se almacena nuevamente en el vector “Y”, así:
Y = a * **X** + **Y**
Esta operación no es computacionalmente costosa, pero en ocasiones se pueden tener una cantidad muy grande de datos y debido al planteamiento y las característica de la función, esto es susceptible a paralizarse. Por tales razones se presenta la función SAXPY de manera secuencial implementada en lenguaje de programación C con el objetivo de desarrollar una versión mejorada con paralelismo.
Adicionalmente, cabe aclarar que el código presentado inicialmente no solamente realiza la función SAXPY, sino que, la realiza en sucesivas iteraciones, y en cada una calcula el promedio de Y. De esta manera se generan mayores oportunidades para explorar conceptos asociados a la paralelización y los hilos.

Para el desarrollo de esta práctica se contará principalmente con:

- El código propuesto inicialmente (Función SAXPY modificada)
- Editor de texto para la programación.
- API de “POSIX Threads” para llevar a cabo la paralelización.
- Entorno LINUX para compilar y ejecutar el código escrito en C.

2. Método de Paralelización

La estrategia de paralelización utilizada fue: tomar la porción del código que realiza las iteraciones (con las operaciones del saxpy dentro) y llevarlo a una función aparte; esta función es la que se le pasa a los hilos cuando son inicializados, así, solo se crean hilos una vez, los cuales individualmente manejan las iteraciones (max_iters) y el cálculo de SAXPY.

Para abordar la sección crítica del código - la cual es la actualización del vector Y_Avgs - se calculó el promedio de una manera diferente, creando una variable local por cada hilo la cual se iba incrementando y al finalizar la porción asignada de la operación SAXPY se llevaba este valor a la variable compartida Y_avgs -la cual tenía exclusión mutua gracias a que fue

protegida mediante un semáforo binario- disminuyendo así la cantidad de accesos a esta sección critica (se pasó de **max_iters * p** accesos a **max_iters* n_threads** accesos) y aumentando el desempeño.

Creación y "join" de hilos.

```
pthread_t threads[n_threads]; //Vector para almacenar n hilos
param_t  params[n_threads];  //Vector  para  almacenar  los  argumentos  a
pasar a n hilos
sem_init(&mutex,0,1); //Inicialización de semáforo binario
    //Creación de n hilos
for (t = 0; t < n_threads; t++){
//Pasar valores para operar los vectores
      //Cálculo de porciones asignadas a cada hilo
      params[t].ini = (p / n_threads) * t;
      params[t].end = (p / n_threads) * (t + 1);
      //Manejo de caso en el que p no es divisible entre n_threads
      if(t == n_threads-1){
         params[t].end = p;
      }
      params[t].X = X;
      …
      params[t].max_iters = max_iters;
//Creación del hilo: Se toma aquel que está en la posición t del vector
que contiene los hilos
//No se requieren parámetros adicionales -> NULL
//Ejecutarán la función "compute"
//Se pasa los parámetros correspondientes al hilo "params[t]"
      pthread_create(&threads[t], NULL, &compute, &params[t]);
      ...
}

for (t = 0; t < n_threads; t++){
      pthread_join(threads[t],  &status);   //Espera  a  que  los  n  hilos
finalicen
      ...
}
```

Función paralelizada

```
void* compute (void *arg){ //Función que es ejecutada por los hilos
    param_t* par =  (param_t *) arg;
    …//Se reciben los argumentos
    tipo nombre_var = par->nombre_var
    …
```

```
    double acc;
    int i;
    //SAXPY iterative SAXPY function
    for(it = 0; it < max_iters; it++){
        acc = 0;//Variable local para optimizar el cálculo del promedio
        for(i = ini; i < end; i++){
            Y[i] = Y[i] + a * X[i];
            acc += Y[i];
        }
      //Sección crítica protegida con semáforo binario
        sem_wait(&mutex);
        Y_avgs[it] += acc/p;
        sem_post(&mutex);
    }
}
```

3. Resultados

A continuación se presentan algunas de las características principales obetenidas con el comando “lscpu”, del computador con SO *lubuntu* en el que se ejecutó el código.

- Nombre del modelo: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
- Socket(s): 1
- Núcleo(s) por socket: 2
- Hilo(s) de procesamiento por núcleo: 2
- CPU(s): 4

El código para realizar las pruebas fue compilado sin optimización usando la instrucción *gcc -Wall nombre.c -o nombre.out -lpthread*, tanto para la versión secuencial como para la paralela.
La ejecución se realizó con el equipo recién encendido y solo teniendo abierto el editor de texto y dos terminales -aquella en la que se ejecutaron las pruebas(*test_long.sh* y *test_short.sh*) y otra en la que se supervisó el uso de cpu utilizando el comando *top* .

Se realizaron dos conjuntos de pruebas variando los parámetros de ejecución, cada combinación de parámetros se repitió 10 veces para obtener el promedio y minimizar el error de los resultados.
En el primer conjunto se fue incrementado el número de iteraciones de la función, para la versión secuencial y la paralela con 1,2,4 y 8 hilos. Se trabajaron 30 configuraciones diferentes, de la siguiente manera:
-p: por defecto(10`000.000)
-n: [versión sin hilos, 1, 2, 4, 8]
-i : [1, 10, 50, 100, 500, 1000]

Para el segundo conjunto, se probó una carga de trabajo muy liviana para dar muestra de los casos en los que la paralelización no está justificada.Así:
-p: 10
-n: [versión sin hilos, 1, 2, 4, 8]
-i : 1

Con el fin de determinar el aumento (o en casos específicos la disminución) del rendimiento de la función ejecutada, con respecto a su versión secuencial, se calculará el *Speed up* asi: $S_p = T_1 / T_p$,donde S_p se refiere al speedup usando p hilos, T_1 se refiere al tiempo de ejecución del programa secuencial y T_p se refiere al tiempo de ejecución usando p hilos.

A continuación se presentan las gráficas con los resultados del Speed up obtenido (el resumen completo de los datos se puede encontrar en el archivo *results_summary.md*)

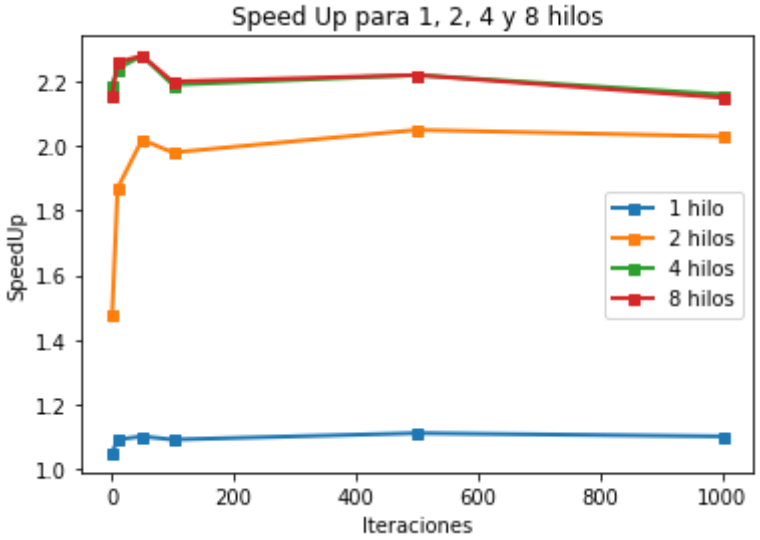

Figura 1. Speed up para diferentes hilos variando el número de iteraciones

- Para el caso de 1 hilo es posible que aunque se tenga que dividir el trabajo, crear los hilos y manejar un semáforo binario se haya dado la pequeña mejoría(speed up ~= 1,1) debido al cambio del cálculo de promedio en la versión paralela ya que no se debe acceder a diferentes posiciones de un vector constantemente (instruccion Y_avgs[it] += Y[i]).
- Para el caso de 2 hilos se obtienen speed up que varían entre aproximadamente 1,5 y 2.Un resultado esperado ya que se tienen el doble de unidades de ejecución y en cargas de trabajo suficientemente intensivas (en términos de cpu) esto equivale aproximadamente a una reducción del tiempo a la mitad del de la versión secuencial
- Para el caso de 4 y 8 hilos se tiene resultados prácticamente iguales, esto debido a que el número de hilos que puede ejecutar el procesador es 4, por lo que un aumento más allá de este número no representaría un aumento en desempeño. En cuanto a speed up se obtuvieron valores de aproximadamente 2,2 en lugar de mayores a 3 como cabría de esperar. Esto podría deberse a que el procesador solo tiene 2 núcleos físicos, pero gracias a la “Tecnología Intel® Hyper-Threading” cada núcleo

físico puede ofrecer dos cadenas de procesamiento, sin embargo, esto no da resultados tan buenos como tener 4 unidades de ejecución **físicas**.

- Es posible que la pequeña disminución de rendimiento en el caso de 1000 iteraciones se deba a que el planificador disminuya la prioridad de estos procesos al considerarlo “cpu intensive”, ya que aunque no se ejecutarán demasiadas aplicación adicionales se sigue teniendo un entorno de escritorio, el cual en ciertos momentos puede estar en niveles de prioridad más altos.

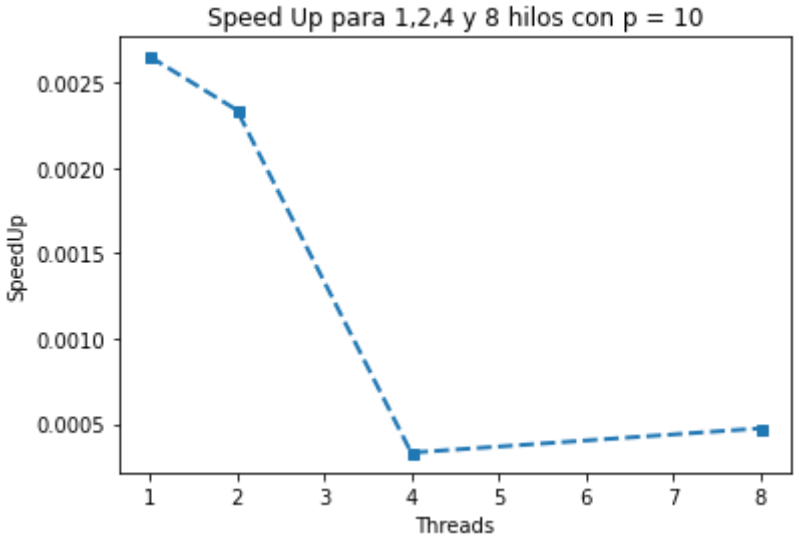

Figura 2. Speed up para diferentes hilos con una iteración y un vector de 10 posiciones

- Para este caso se tienen valores de Speedup menores a 1, lo que indica una disminución del rendimiento.Esta situación es totalmente razonable, ya que la división de trabajo, creación y finalización de hilos o en general, la preparación previa para un programa paralelo solo se amerita cuando se tienen tareas relativamente(dependiendo del problema) largas o intensivas.

4. Conclusiones

Finalmente, a partir de las actividades desarrolladas(paralelización del código original, generación de pruebas y resumen de speed up) se puede constatar cómo se logró satisfactoriamente la paralelización de la función SAXPY modificada(SAXPY y cálculo de promedio), siendo los resultados de la versión paralela consistentes a través de las repetición con ella misma y también con los resultados de la versión secuencial. Así mismo, la paralelización realizada mejoró la eficiencia para los casos en los que la carga del trabajo realizado era lo suficientemente intensiva como para amortizar la preparación del trabajo y la creación de los hilos.

Estos resultados obtenidos demuestran que paralelizar puede llegar a ser de mucha ayuda, permitiendo mejorar la eficiencia y el desempeño en los casos donde se pueda, se deba (si la carga de trabajo lo amerita) y se realiza correctamente. Sin embargo, al paralelizar se debe tener especial cuidado con cómo se hace, ya que si esta se desarrolla de una manera

incorrecta puede generar muchos problemas como: bugs, deadlock, livelock y resultados no determinísticos(cuando deberían serlo).

De igual manera, cabe resaltar que el estándar POSIX también permitió realizar de manera amigable y sencilla la paralelización, gracias a su API *pthreads*; no obstante, este estándar se debe usar de manera correcta, so pena de incurrir en los problemas que se comentaron anteriormente.

Lo desarrollado a lo largo de este informe se puede encontrar en el archivo adjunto o en el repositorio de github: https://github.com/sescobar99/SO-Lab3-20201

5. Referencias bibliográficas

Arpaci-Dusseau, R., & Arpaci-Dusseau, A. (2015). Operating Systems: Three Easy Pieces (English Edition) (1.a ed.). Arpaci-Dusseau Books. Recuperado de: http://pages.cs.wisc.edu/~remzi/OSTEP/

Barney, B. (s. f.). POSIX Threads Programming [Libro electrónico]. Lawrence Livermore National Laboratory. Recuperado de: https://computing.llnl.gov/tutorials/pthreads/

Procesador Intel® CoreTM i5-5200U (caché de 3 M, hasta 2,70 GHz) Especificaciones de productos. (s. f.). Intel. Recuperado de: https://ark.intel.com/content/www/es/es/ark/products/85212/intel-core-i5-5200u-processor-3m-cache-up-to-2-70-ghz.html

Lopez Azaña, D. (12 de febrero de 2015). Diferencias entre CPU física, CPU lógica, Core/Núcleo, Thread/Hilo y Socket. [Entrada en Blog]. Recuperado de: https://www.daniloaz.com/es/diferencias-entre-cpu-fisica-cpu-logica-core-nucleo-thread-hilo-y-socket/