T1 - Computación de Alto Rendimiento (IIC3533)

Christian Klempau - Cristian Carrasco

Introducción

El código a analizar se trata de una simulación de un escenario de la evolución de un incendio con múltiples focos y una cantidad de equipos de bomberos. En concreto, el código extinguishing.c es propiedad de *Computación Paralela, Grado en Informática (Universidad de Valladolid)*, de la simulación llamada *Simplified simulation of fire extinguishing*.

El código se encuentra en **kraken.ing.puc.cl**:**/home/cristian.carrasco/IIC3533-T1**

Descripción de paralelización

Se paralelizaron 4 secciones del código. Sin embargo se probó con más parelelizaciones. Se mantuvieron las que por un lado no causaban errores de cálculo en la ejecución del programa y además tenían una mejora de la velocidad más evidente.

En general, algunas paralelizaciones generaban inconsistencias de resultados, ya que el hecho de introducir paralelismo, por temas de consistencia de datos, causaba una simulación incorrecta. Por lo mismo, sólo marcamos como código paralelo aquellas secciones que sí permitían ser paralelizadas, ya que por ejemplo accedían a secciones de datos adyacentes, o directamente distintas.

También se probó colapsar (argumento “collapse” en directiva omp parallel) los ciclos for anidados hasta 2 niveles. No hubo ninguna ganancia significativa en tiempo de ejecución, incluso con peor *performance* en algunos casos.

En la sección 4.1, se utilizó:

```
#pragma omp parallel for reduction(+:num_deactivated)
 for( i=0; i<num_focal; i++ ) {
        ...
```

En la sección anterior, el código original no paralelo itera sobre los puntos focales, y cuenta la cantidad de puntos desactivados.
Dado que no hay modificaciones cruzadas de datos, es una sección completamente paralelizable entre núcleos de CPU. Por lo mismo, se utiliza la directiva “reduction”, que como el nombre indica reduce **num_deactivated** al total final, entre todos los núcleos. En esencia, cada proceso paralelo calcula para su sección de datos asignada (dividida automáticamente por OMP), y el resultado se agrega a una variable compartida global entre procesos.
Por lo mismo, el resultado final es consistente al código secuencial, sólo que a un menor tiempo de ejecución.

En la sección 4.2.2, se utilizó:

```
#pragma omp parallel for schedule(static)
 for( i=1; i<rows-1; i++ )
```

```
        for( j=1; j<columns-1; j++ )
            accessMat( surfaceCopy, i, j ) = accessMat( surface, i, j );
```

En este caso, los “for” anidados no hacen más que copiar los datos desde el arreglo **surface** a **surfaceCopy**. Por lo tanto, la sección es completamente paralelizable, ya que no hay dependencias de datos entre threads, dado que se escribe en secciones distintas de **surfaceCopy** mientras que **surface** solo se lee.

En la sección 4.2.3, se utilizó:

```
    #pragma omp parallel for schedule(static)
    for( i=1; i<rows-1; i++ )
        for( j=1; j<columns-1; j++ )
            accessMat( surface, i, j ) = (
                accessMat( surfaceCopy, i-1, j ) +
                ...
```

La explicación es análoga a la sección 4.2.2. En este caso, se copia desde la matriz **surfaceCopy** hacia la matriz **surface**. Nuevamente, como se paraleliza, omp entrega secciones de datos (en este caso, filas y columnas distintas) a los distintos threads, entonces no hay duplicidad de escritura en los mismos datos, ni problemas de consistencia.

En la sección 4.2.4, se utilizó:

```
    #pragma omp parallel for reduction(max : global_residual)
    for( i=1; i<rows-1; i++ )
        for( j=1; j<columns-1; j++ )
            // global residual
```

Acá, en el caso secuencial, para cada elemento de la matriz se calcula el residual (diferencia en valor absoluto) entre las dos matrices mencionadas anteriormente. Luego, si en ese punto, el desigual es mayor al residual global, se actualiza el global al nuevo valor.
El argumento de **reduction** para la directiva, en este caso, reduce el valor entre todos los threads al máximo obtenido. Es decir, cada thread calcula su propio **global_residual**, y luego se obtiene el máximo valor final entre todos ellos.
Ya que cada valor residual no depende de otros valores residuales, ni ninguna sección de código dinámica, entonces la operación es perfectamente paralelizable sin problemas de validez de datos.

En la sección 4.3 y 4.4 no se paralelizó, ya que el rendimiento era peor en todos los casos y *testcases*. En algunos casos, incluso el resultado final o *output* obtenido era inválido, teniendo como base el código sin paralelizaciones.

Instrucciones de ejecución

Basta con ejecutar desde la raíz de la carpeta:

sbatch execute.sh

Lo anterior ejecuta el binario compilado con configuración desde 1 hasta 32 threads, con el test “test3”.

Resultados

A continuación, se muestran los resultados de la ejecución de la simulación paralelizada, desde 1 a 32 threads, para el *testcase* 3.
La primera tabla contiene los resultados finales de ejecución, para tiempo, speedup, y eficiencia paralela. Luego, se grafican en gráficos de línea dichos resultados.
En las tablas se observan los resultados para los tests 2 y 3.
En el caso de los gráficos, todos ellos muestran los resultados para el test 3.

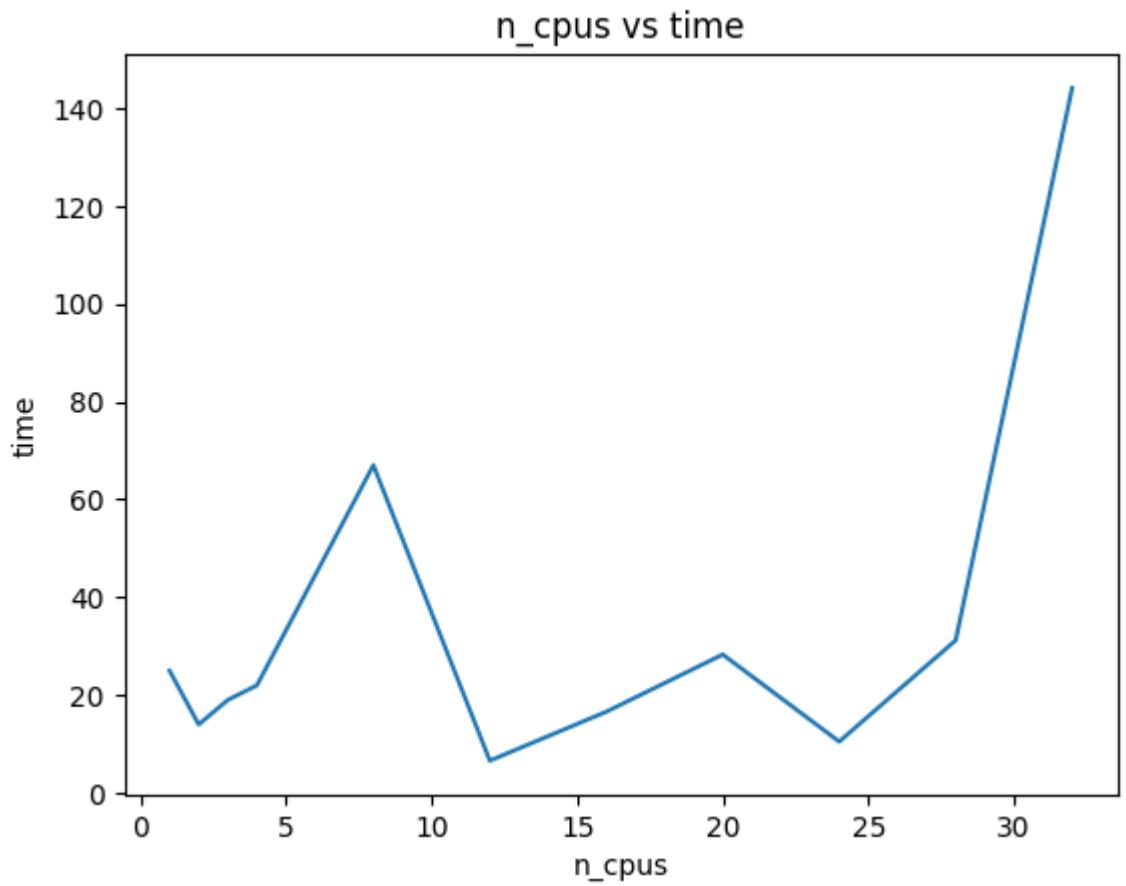
Tabla de resultados para test4

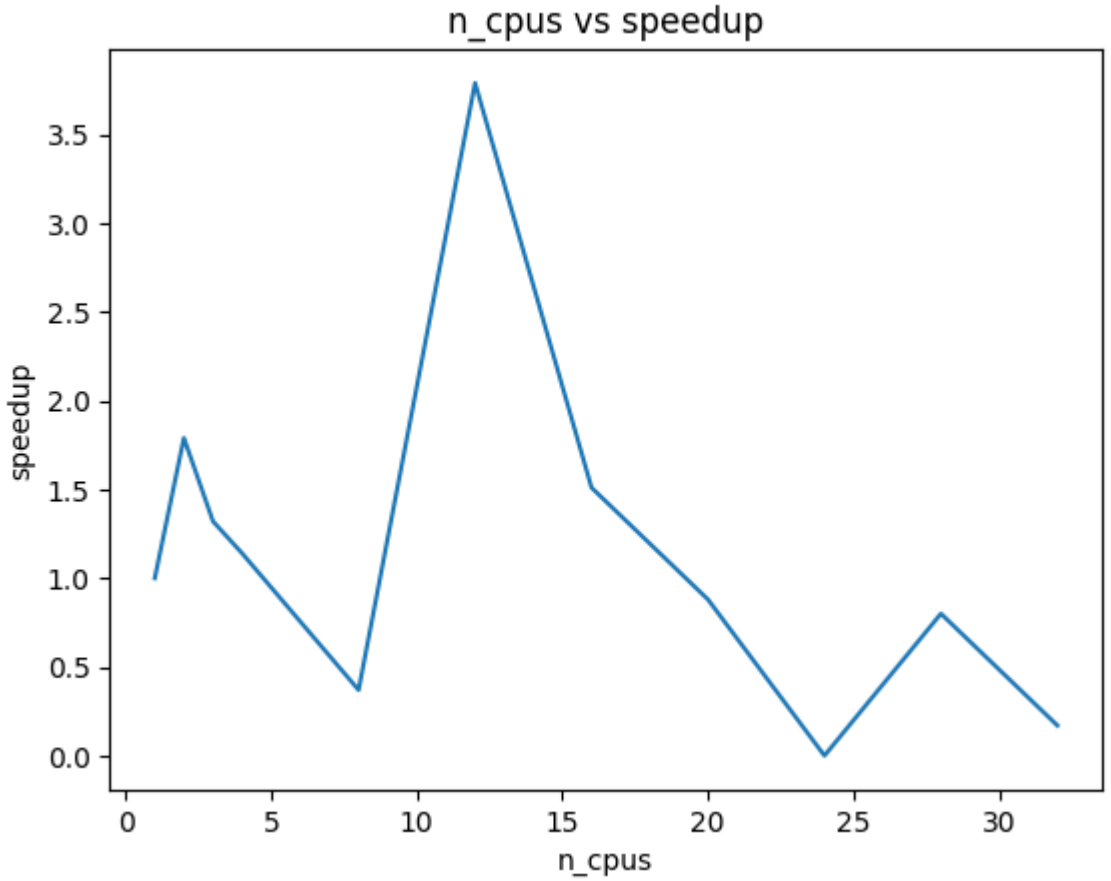
n_cpus	time	speed up	ef paralela
1	29,2	1,00	1,00
2	26,7	1,09	0,55
3	27,5	1,06	0,35
4	57,7	0,51	0,13
8	22	1,33	0,17
12	22,8	1,28	0,11
16	74,7	0,39	0,02
20	33,6	0,87	0,04
24	84,6	0,35	0,01
28	34,8	0,84	0,03
32	24,3	1,20	0,04

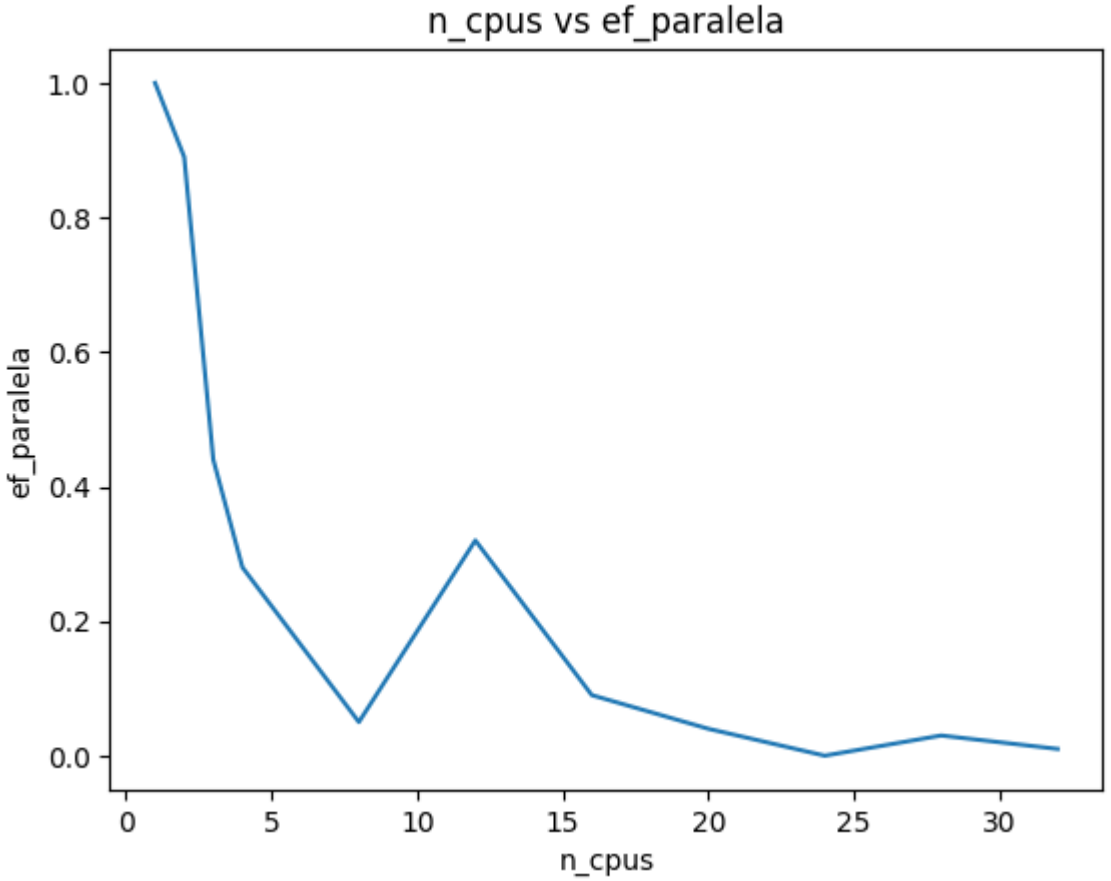
Tabla de resultados para test3

n_cpus	time	speed up	ef paralela
1	25	1,00	1,00
2	14	1,79	0,89
3	19	1,32	0,44
4	22	1,14	0,28
8	67	0,37	0,05
12	6,6	3,79	0,32
16	16,6	1,51	0,09
20	28,3	0,88	0,04
24	10.5	0,00055	0,00

28	31,2	0,80	0,03
32	144,1	0,17	0,01




n_cpus vs speedup
speedup
n_cpus
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
5
10
15
20
25
30


n_cpus vs ef_paralela
ef_paralela
n_cpus
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20
25
30

Discusión

Los resultados obtenidos no representan el trasfondo teórico que se ha estudiado en el curso. Según las distintas leyes comentadas y los análisis realizados se esperaría que al aumentar la cantidad de threads y núcleos a la ejecución el tiempo de ejecución disminuye hasta detenerse y luego incluso volver a aumentar.

Lo que se observó en esta tarea es que en general si existe una disminución del tiempo al aumentar el número de threads de una ejecución. Por lo general disminuye el tiempo, sin embargo no siempre fue el caso, incluso notamos casos donde el tiempo empeoró. Se observaron disminuciones en el tiempo, pero también cambios erráticos donde finalmente el tiempo terminó por aumentar.

Lo anterior puede explicarse por la aleatoriedad que tiene cada ejecución y la presencia de distintos procesos ejecutándose en el mismo cluster de ejecución (en nuestro caso, IALAB-HIGH), lo que genera una competencia por los recursos del cluster. Además, la misma ejecución del programa puede ocurrir de diversas formas y por lo mismo las asignaciones de recursos y memoria serán diferentes también. incluso entre ejecuciones iguales se observó cambios considerables en el tiempo de ejecución.

Se podría haber obtenido un mejor resultado en la simulación, por ejemplo considerando la naturaleza del input de la simulación en la distribución de recursos y threads del programa. También, con un análisis más exhaustivo de cada directiva y sus posibles opciones, como el schedule y el collapse, incluso sincronización entre procesos.

En conclusión, se observó una tendencia general de disminución en el tiempo de ejecución de este programa a medida que se utiliza un mayor número de threads, sin embargo, esta tendencia no fue tan consistente como cabría esperar, pudiendo ser causado por factores de aleatoriedad de disponibilidad de recursos (CPU especialmente, y en menor grado RAM y disco) del mismo cluster. Es posible que un ambiente mucho más controlado hubiera mostrado resultados que se parezcan más a la teoría. Esto no quita que el hecho de que el paralelismo pueda ser muy beneficioso para nuestros programas (en los debidos casos) y es muy recomendable saber utilizarlo.