Departamento de [Ingeniería Civil Informática]
Facultad de Ingeniería
Universidad de Santiago de Chile

[Sistemas Distribuidos y Paralelos]

LAB 2: Paralelismo orientado al objetos uCPP

Nícolas Farfán Cheneaux

Resumen

En el presente informe de rendimiento computacional, se estudia el desempeño de una solución en el dialecto de c++, **ucpp** o **uc++**, el cual está implementado usando objetos de este, tales como **Tareas, Corrutinas, Mutex, Monitores**. mediante la lectura de un archivo que contiene millones de líneas de visibilidades, las cuales deben ser leiadas por multiples tareas grideadoras de forma concurrente utilizando el **multikernel** de **uc++**.

Estrategia de paralización

Se realizan dos métodos de paralelización y concurrencia, en primer lugar el de las matrices propias/privadas donde cada tarea tiene sus propias matrices y al finalizar deben ser acumuladas en una absoluta en el main o tarea principal, la otra solución consiste en que exista solo una matríz absoluta y sea escrita por exclusión mútua usando un Mutex, para ello se crean **t** tareas para que lean el archivo y se repartan concurrentemente las líneas a leer, dado que cada visibilidad es independiente a otra, las tareas se crean en la tarea principal mediante un ciclo for, en un arreglo de tareas. Cada una de estas accederá a la corrutina con **Exclusión Mútua (EM)** que será la encargada de leer **-c** líneas del archivo (el número de chunk), estas líneas son devueltas en forma de vector a la tarea y son grideadas en las matrices, finalmente cuando las tareas hayan terminado serán eliminadas y se obtendran los resultados del grideo mediante sus matrices locales o globales.

Pruebas realizadas

Las primeras pruebas se realizaron en el **Cluster xi.diinf.usach.cl**, en el que el presente programa fue ejecutado mediante el script:

```
#!/bin/bash
#SBATCH --job-name=gridding-job
#SBATCH --partition=batch
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --output=job-gridding.out
#SBATCH --error=job-gridding.err
#SBATCH --cpus-per-task=64

./gridding -i data.csv -o out.raw -d 0.003
-N 2048 -c 3 -t 10
```

Obteniendo un archivo **job-gridding.out** con el output del problema. Sin embargo se obtuvo un peor tiempo de ejecución al ejecutar el script que solamente en la línea de comando el código, por lo tanto todas las pruebas fueron ejecutadas en el nodo principal o por defecto del cluster.

Resultados

Todas las pruebas fueron usando el cluster del diinf, en el nodo principal de login **sbtach** con las especificaciones:

- CPU: 2x 32-Core/64-Thread 3rd Gen AMD EPYC 7513
- Memory: 256GB
- OS: Ubuntu 22.04 lts

¿Existe una diferencia entre ambos tipos métodos?

Se toman 2 muestras pareadas de 20 observaciones cada una, independientes de cada método, como se ve en la siguiente tabla y se utiliza la prueba **t student** de muestras pareadas, se calcula la diferencia de tiempos entre

ambos métodos, tiene fijado 5 tareas y 10000 chunk.

Shared Matrix	Private Matrix
4,25316	4,01180
4,08298	3,98429
4,33247	4,19808
4,43508	3,93601
4,31961	3,91274
4,36475	4,17609
4,41134	3,99839
4,40525	4,19969
4,24102	4,19143
4,38621	3,98246
4,28619	4,17385
4,41644	4,22297
4,26755	4,23089
4,4156	4,18258
4,34266	4,17078
4,32459	4,20009
4,29833	4,20117
4,30637	3,96725
4,37074	3,96108
4,29446	4,07614

```
Shapiro-Wilk normality test

data:  diff
W = 0.92212, p-value = 0.1088
```

Se usa shapiro para la normalidad y se concluye que los datos siguen una distribución normal con un **p-value** de 0.10 con 95 % de confianza. Se realiza la prueba t

```
data:  diff
t = 7.4588, df = 19, p-value = 4.668e-07
```

Con 95 % de confianza se rechaza la hipótesis nula, es decir se concluye que ambos algoritmos tienen tiempos de ejecución distintos, es decir una solución difere de la otra.

¿El tamaño del chunk influye en el rendimiento?

Para esta prueba se toman 60 muestras, 20 con un tamaño de chunk de 10, 1000 y 100000.

chunk 10	chunk 1000	chunk 100000
4,32000	4,19717	4,21791
4,22000	4,22174	4,31700
4,27117	4,28654	4,21325
4,34203	4,24367	4,24704
4,33079	4,19843	4,38931
4,26585	4,19495	4,22924
4,28742	4,17968	4,2418
4,32123	4,12144	4,2342
4,29019	4,0708	4,26846
4,30624	4,23781	4,38267
4,38115	4,22217	4,26174
4,29657	4,12284	4,26799
4,24633	4,08128	4,29617
4,30114	4,2243	4,20238
4,54894	4,27612	4,22473
4,3482	4,19013	4,38614
4,43427	4,23085	4,24121
4,3048	4,09935	4,26369
4,33578	4,22439	4,28081
4,18644	4,16547	4,27665

Se utiliza ANOVA para muestras correlacionadas dado que se trata de un solo algoritmo (matrices compartidas), también se obtiene que siguen una distribución normal.

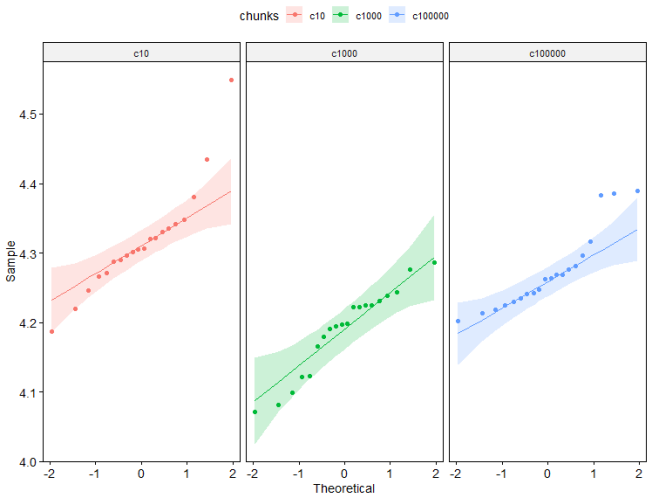

Figura 1: N ormalidad

Se realiza el test de ANOVA

```
prueba2 <- ezANOVA ( data = datos , dv = tiempo ,
within = chunks ,wid = instancia , return_aov = TRUE
prueba2
```

Se obtiene un p-value muy pequeño de **7.136871e-07** en consecuencia, concluye con 99 % de confianza que

el tiempo de ejecución promedio es significativamente diferente para al menos uno de los chunks.

Se grafica el tamaño del efecto

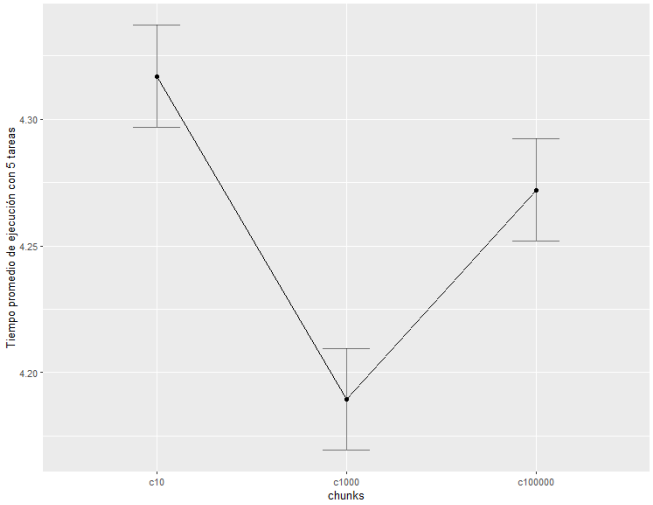

Figura 2: Tamaño del efecto

Se aplica la prueba post hoc de Bbonferroni para ver donde se encuentran las diferencias.

```
bonferroni <- pairwise.t.test(datos$tiempo,
datos$chunks,p.adj = "bonferroni",
paired = TRUE )
print ( bonferroni )
```

```
        c10     c1000
c1000   1.4e-06 -
c100000 0.1911  0.0014
```

Donde se observa diferencias significativas entre el chunk de tamaño 10 y 100000, por lo que existe una diferencia en los tiempos de ejecución dependiente del tamaño del chunk, sin embargo entre 10 y 1000 chunk no se observa una diferencia significativa, dado que al tener que acceder mediante exclusión mutua las tareas si se realiza repetitivamente con un número pequeño de chunk el tiempo será desfavorable.

Speedup

Usando la formula

$$\frac{T_s}{T_p(n)}$$

En donde **Ts** representa el valor obtenido en la ejecución secuencial (1 tarea) en este caso se obtuvo un tiempo de **2.92978**. Por otro lado, Tp(n) representa el valor de tiempo de ejecución obtenido utilizando n tareas, desde 2 tareas hasta 15 tareas. Se utilizó los siguientes parámetros

```
./gridding -d 0.003 -N 2048 -c 10000
```

Tareas	Tiempo
1 (Ts)	2.92978
2	3.14326
3	4.15092
4	4.1311
5	4.25886
6	4.29552
7	4.46537
8	4.45734
9	4.42189
10	4.34559
11	4.41952
12	4.34371
13	4.5731
14	4.48062
15	4.51469

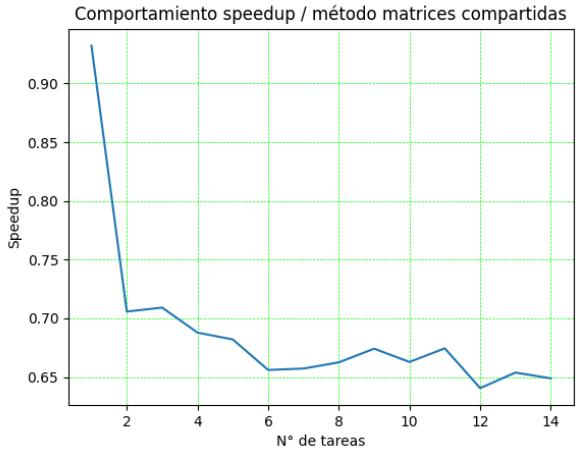

Figura 3: Speedup

Se puede observar un speedup negativo, es decir que el tiempo secuencial ejecutado por una sola tarea es mejor incluso que agregar solo 1 tarea más, incluso se ha probado con **distintos tamaños de chunk**, dado que en la sección anterior se concluyó que el tamaño del chunk influye en el tiempo de ejecución, sin embargo no se logra batir la marca del tiempo de ejecución de una sola tarea (No se muestra el otro método dado que el resultado es el mismo). además se utiliza el comando **htop** para ver el

trabajo de la CPU y efectivamente se observa una carga mayor al tener más tareas.

Intentos de mejorar/optimizar la solución

En cuanto al código se utilizó arreglos dinámico y se entregó una referencia a cada tarea y no vectores, además los valores de deltaU fueron calculados antes de cada iteración de la tarea, además se utilizó float en vez de double. En terminos de optimización todo lo que está dentro de los posibles se ha realizado.

Se intentó también implementar un cluster/uCluster de uCPP con distintos procesadores/uProcessor, mediante un código provisto por el profesor, donde cada tarea sea asignada a uno de estos pero se obtuvo un peor resultado, llegando a 114[s] incluso x10 al tiempo normal, a pesar de estar compilando el programa mediante las flags.

```
u++ -multi -debug gridding.cc
```

Pero no se obtuvo ninguna mejora.

Posibles motivos del fallo

Si bien es cierto se implementó la solución tal cual se pedía, se sospecha que esta solución no funciona debido a un overhead/cuello de botella de las tareas, dado que el archivo de las visibilidades debe ser accedido mediante exclusión mutua, solo una tarea/hebra puede acceder a este, lo ideal es que mientras una tarea esté leyendo las siguientes estén discretizando las visibilidades y grideando en la matriz, pero por algún motivo este proceso se entorpece y no otorga un mejor tiempo que de manera secuencial, por lo tanto se cree que es debido al excesivo I/O que debe realizar cada tarea.

Imágenes resultantes

Mediante el siguiente script creado por mi, se realiza la visibilización de los resultados

```
% Leer los datos de los archivos
s1_s = fread(fopen("datosgrideados_sharedr.raw", "r"), "float");
s2_s = fread(fopen("datosgrideados_sharedi.raw", "r"), "float");

s1_p = fread(fopen("datosgrideados_privater.raw", "r"), "float");
s2_p = fread(fopen("datosgrideados_privatei.raw", "r"), "float");

% Reshape de los datos
re_s = reshape(s1_s, 2048, 2048);
re_p = reshape(s1_p, 2048, 2048);
im_s = reshape(s2_s, 2048, 2048);
im_p = reshape(s2_p, 2048, 2048);

% Crear n meros complejos
v_s = complex(re_s, im_s);
v_p = complex(re_p, im_p);

% Calcular transformada de Fourier e inversa
I_s = fftshift(ifft2(v_s));
I_p = fftshift(ifft2(v_p));

% Crear una figura con dos im genes en una fila
figure;

% Primera imagen
subplot(1,2,1);
imagesc(abs(I_s));
colormap('hot');
title('Shared matrix result image');

% Segunda imagen
subplot(1,2,2);
imagesc(abs(I_p));
colormap('hot');
title('Private matrix result image');
```

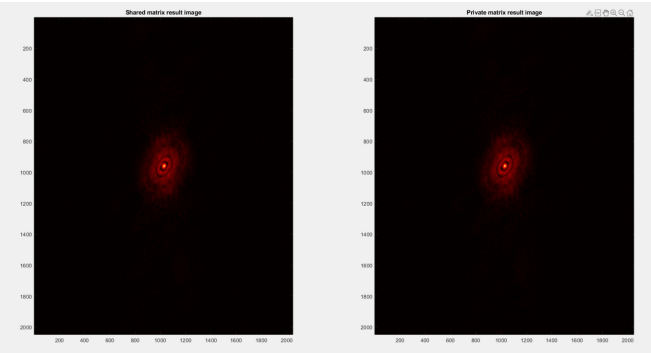

Figura 4: Resultados

Cabe señalar que se logró el objetivo del código en un principio, si bien la solución no fue la esperada, se consiguió gridear las visibilidades del archivo y visualizarlas, donde se obtiene resultados idénticos en ambas soluciones.

Conclusión

Si bien no se logró paralelizar la solución, aún se sigue buscando la solución a este problema, se sospecha que

se trate de la lectura al archivo dado que este debe realizarse con exclusión mutua está limitado a que las tareas esperen que una salga para leer, de la misma forma con la escritura. El paralelismo debería ocurrir cuando una está leyendo las otras podrían ir calculando, aquí si se obtendría una mejora, sin embargo no parece que se esté logrando, dado que lo que se observa es una colisión entre tareas. El otro motivo se podría deber a alguna falla en uCPP o alguna configuración extra faltante, pero debido a la escaza información en internet acerca de este dialecto no se puede llegar a un consenso.

Referencias

CS 343 - Concurrent and Parallel Programming By Container: Uwaterloo.ca Year: 2023 URL: https://student.cs.uwaterloo.ca/ cs343/common/codeExamples.shtml

uCPP repo https://github.com/pabuhr/uCPP/tree/master