Contents

1 Introduction

La ley de Amdahl es un modelo matemático que permite estimar y medir los alcances de una mejora en un proceso.
Mediante esta ley buscamos ver el desempeño de las implementaciones a partir de conocer los tiempos de ejecución de dichas tareas.

Se define como:

$$SU_g = \frac{1}{1 - fm + \frac{fm}{SU_l}} \tag{1}$$

También se puede pensar como:

$$SU_g = \frac{T0}{Tn} \tag{2}$$

Donde:

- SU_g: speedup global
- SU_l: speedup local
- fm: fracción del tiempo en la que se puede aplicar la mejora.
- T0: tiempo que tarda en ejecutarse la implementación sin la mejora.
- Tn: tiempo que tarda en ejecutarse la implementación con la mejora

Un speedup cercano a 1 indicaría que ambas implementaciones se comportan de forma similar en cuanto a desempeño.

Mediante la Ley de Amdahl se estudiará la performance de las implementaciones dadas, para analizar si la introducción de una posible mejora en un proceso aportaba un cambio significativo en el rendimiento.

2 Objetivos

Proveer una versión del programa en la que las funciones new orientation y move forward estén codificadas en assembly MIPS32, y comparar su desempeño con el de esas mismas funciones compiladas con diversos grados de optimización.

3 Consideraciones previas

Se utilizan dos implementaciones base de la hormiga artista y dos implementaciones con las funciones move forward y new orentation codificadas en assembly:

- a) tp1_if implementación sin mejora
- b) tp1_tables implementación mejorada (usando USE TABLES).
- c) tp1_if_asm implementación sin mejora codificada por nosotros en assembly
- d) tp1_tables_asm implementación mejorada codificada por nosotros en assembly.

Los tiempos de ejecución de las implementaciones se miden como un promedio, para esto se realiza la medición tres veces mediante el uso del comando time. Para la obtención del promedio se utiliza la media geométrica.

Con time obtenemos tres tipos de tiempo de ejecución: real, user y sys. Nos interesa usar la medición indicada por user (no se incluyen tareas de kernel), ya que si usaramos real se podría incluir el tiempo de otros procesos y afectar los resultados.

4 Procedimiento

Se empleó como herramienta QEMU para emular un procesador. Como sistema operativo guest se utilizó Debian MIPS.
Como sistema operativo host se utilizó Ubuntu 18.04.2 LTS.

Una vez configurado el entorno, se accede a la máquina guest desde el host mediante ssh. Se copian los archivos alojados en la máquina guest a la otra, donde se los compila con las opciones necesarias.

4.1 Mediciones con time

Se mide el tiempo que las implementaciones en assembly tardan en ejecutar diez mil operaciones en la menor grilla posible (1x1), y se repete escalando la cantidad de operaciones.
Luego se repite con una grilla más grande (1024x1024)

Se compila la versión en C del programa con los grados de optimización -O0,-O1,-O2 y -O3, y se realizan las mediciones para una matriz de 1024x1024 con 10 a la n iteraciones, para n entre 4 y 9.

Para optimizar las ejecuciones de las pruebas se realizaron los siguientes

scripts:

```
#!/bin/bash
#Script to run 3 times

PROGRAM=$1
GRID=$2
MUL=1
TIMEFORMAT=%U
for i in {4..9}
do
        ((MUL = MUL * 10))
        echo iteraciones : $((MUL * 1000))
        for j in {1..3}
        do
                time ./$PROGRAM -g $GRID -p RGBW -r LLLL -t $((MUL
    * 1000)) > /tmp/TP1
        done
done
```

Las ejecuciones realizadas (medidas en segundos) fueron los siguientes:

Parámetros	tp1_if_asm	tp1_tables_asm	Speedup Global
1x1 10*1000	0,019	0,027	0,684
1x1 100*1000	0,048	0,045	1,053
1x1 1000*1000	0,367	0,259	1,418
1024x1024 10*1000	2,846	3,120	0,912
1024x1024 100*1000	2,835	3,196	0,887
1024x1024 1000*1000	3,328	3,210	1,037

Table 1: tp1_if_asm vs. tp1_tables_asm (en segundos)

Grilla 1024x1024	-O0	-O1	-O2	-O3	tp1_if_asm
Iter: 10*1000	3,023	3,052	3,117	3,018	2,846
Iter: 100*1000	3,186	2,933	2,701	2,916	2,835
Iter: 1000*1000	3,186	2,957	3,145	3,248	3,328
Iter: 10000*1000	5,511	4,825	4,541	5,082	6,621
Iter: 100000*1000	29,080	17,328	21,852	16,962	38,350
Iter: 1000000*1000	259,974	269,642	261,369	262,827	343,966

Table 2: Optimizaciones vs. tp1_if_asm (en segundos)

Grilla 1024x1024	-O0	-O1	-O2	-O3	tp1_tables_asm
Iter: 10*1000	3,077	3,058	2,887	3,192	3,120
Iter: 100*1000	3,105	3,024	2,779	3,047	3,196
Iter: 1000*1000	3,517	3,205	3,132	3,135	3,210
Iter: 10000*1000	7,864	3,205	5,370	4,936	5,460
Iter: 100000*1000	39,672	21,986	23,772	23,065	30,818
Iter: 1000000*1000	389,353	198,522	197,986	264,995	258,114

Table 3: Optimizaciones vs. tp1_tables_asm (en segundos)

4.2 Análisis de resultados

En la tabla 1, analizando el speedup global, se observa que al aumentar la cantidad de iteraciones realizadas la implementación tp1_tables_asm presenta una mejora en la performance con respecto a la implementación tp1_if_asm (speedup mayor a uno).

Si tomamos un tamaño fijo de grilla (1024x1024), a medida que aumentamos la cantidad de iteraciones, los tiempos para tp1_tables son mayores. Por lo que obtenemos speedups menores a 1. Esto representa que la implementación no introduce una mejora con respecto a tp1_if_asm, sino que desmejora el tiempo de ejecución.

Para un tamaño de grilla de 1024x1024 con 1000*1000 iteraciones, ambas implementaciones en assembly se comportan de forma similar (speedup cercano a uno).

Con respecto a la performance al compilar el código en C con optimizaciones, tomando un tamaño de grilla fijo (1024x1024), el tiempo de ejecución de tp1_if_asm fue mejor que el obtenido con las optimizaciones.
Para una cantidad de iteraciones grande, el tiempo de ejecución de tp1_if_asm empeora considerablemente con respecto a las optimizaciones.

En el Anexo se puede observar la tabla completa de mediciones.

Como el tiempo obtenido para una cantidad de iteraciones con n=9 no permite apreciar las diferencias para las iteraciones pequeñas, se muestra uno de los gráficos hasta n=8.

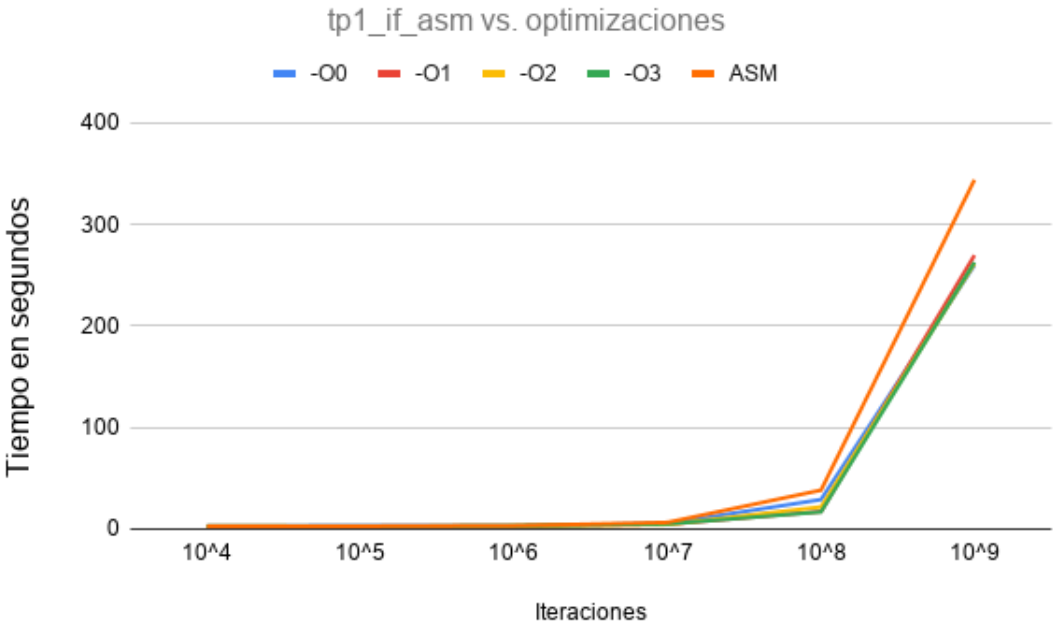

Figure 1: Optimizaciones vs. tp1_if_asm

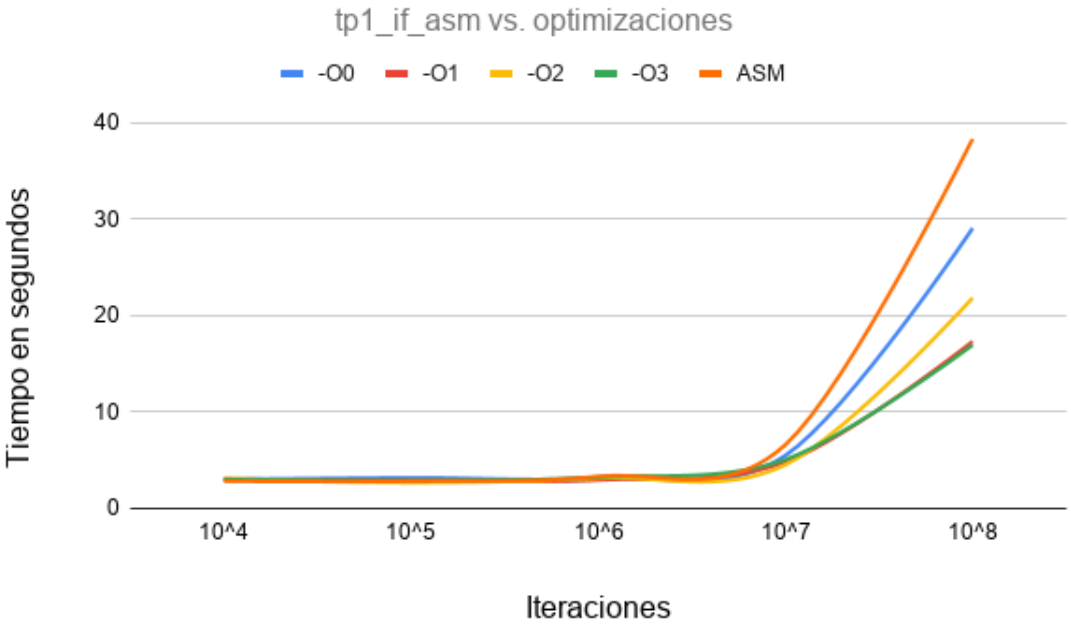

Figure 2: Optimizaciones vs. tp1_if_asm

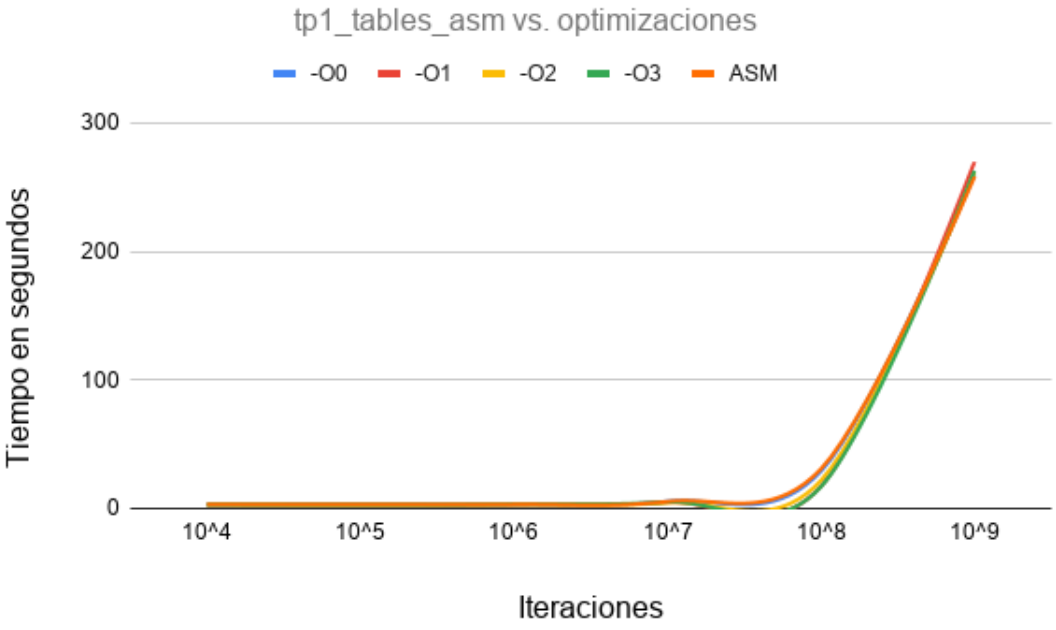

Figure 3: Optimizaciones vs. tp1_tables_asm

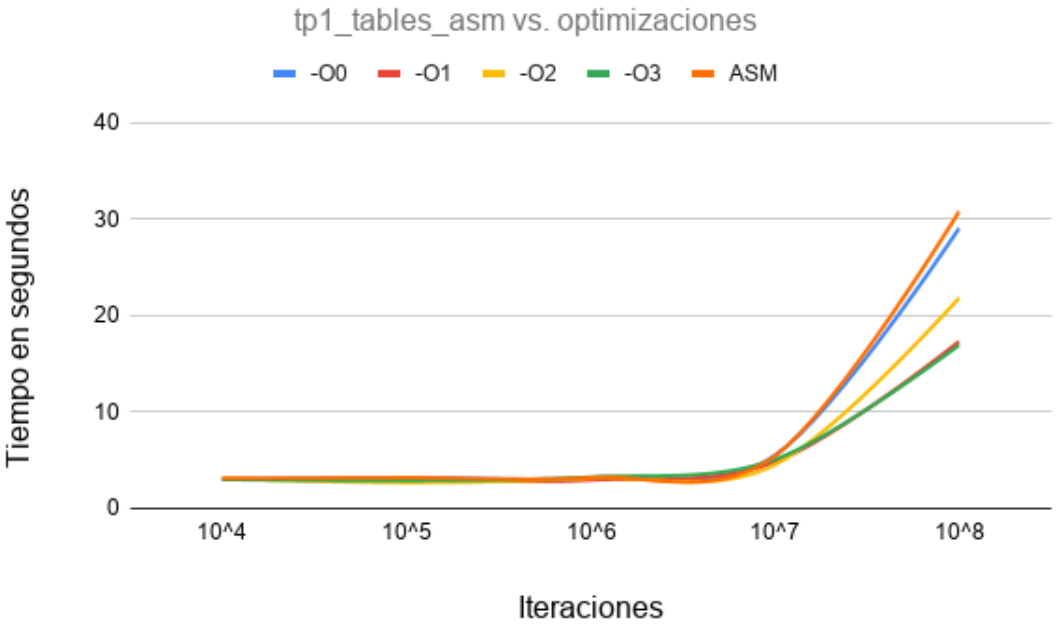

Figure 4: Optimizaciones vs. tp1_tables_asm

5 Conclusión

Assembly nos da más libertad para manejar la memoria de una forma optima. Por ejemplo podemos setear el tamaño del stack requerido. Pero al haber más libertad, el programador tiene mayor responsabilidad, dando lugar a un incremento de la cantidad de errores si no se manejan los de accesos a memoria de forma correcta.

La implementación tp1_tables_asm se ve beneficiada con tamaños de grilla pequeños y al aumentar la cantidad de iteraciones. En esos casos, la implementación es más rápida que tp1_if_asm (speedup mayor a uno).

Ventajas de las optimizaciones: se puede ver que la performance de las implementaciones aumenta al aumentar el número de optimización, siendo -O3 la más performante.
Debido a la cantidad de optimizaciones que realiza el sistema operativo, se obtiene un aumento en el tiempo de compilación. Esto junto con la imposibilidad de usar un debuger, ya que el código assembly se encuentra muy optimizado, son las desventajas que observamos al compilar con optimizaciones. Y es por esto que se desaconseja el uso de -O3.

Notamos que se aconseja usar -O2 ya que a pesar de tener un rendimiento menor al obtenido con -03, no posee las desventajas de este y mejora notablemente con respecto al código sin optimizar.

Con las optimizaciones se encontró que la memoria utilizada es mayor al aumentar el número de las mismas. Por lo tanto no se recomienda este tipo de optimizaciones cuando lo que se desea es optimizar el espacio.

Otra de las desventajas de utilizar assembly en lugar de un lenguaje de alto nivel es que, al ser un lenguaje de bajo nivel se dificulta el seguimiento del código a la hora de desarrollarlo.

6 Anexo - Ejecuciones