

```
C Programa1_1.c > main()
1  #include <stdio.h>
2  #include <time.h>
3  #define N 1000000
4
5  int main() {
6      int sum = 0;
7      int array[N];
8
9      // Initialize the array
10     for (int i = 0; i < N; i++) {
11         array[i] = i;
12     }
13     struct timespec ti, tf;
14     double elapsed;
15     clock_gettime(CLOCK_REALTIME, &ti);
16     // Calculate the sum of the array
17     for (int i = 0; i < N; i++) {
18         sum += array[i];
19     }
20     clock_gettime(CLOCK_REALTIME, &tf);
21     elapsed = (tf.tv_sec - ti.tv_sec)*1e9 + (tf.tv_nsec - ti.tv_nsec)*1e-9;
22     printf("El tiempo que toma el programa es : %.2lf \n", elapsed);
23     // Print the sum
24     printf("Sum: %d\n", sum);
25
26     return 0;
27 }
28
```

PROBLEMAS 1

SALIDA

TERMINAL ...

bash

+

⌵

⌶

🗑

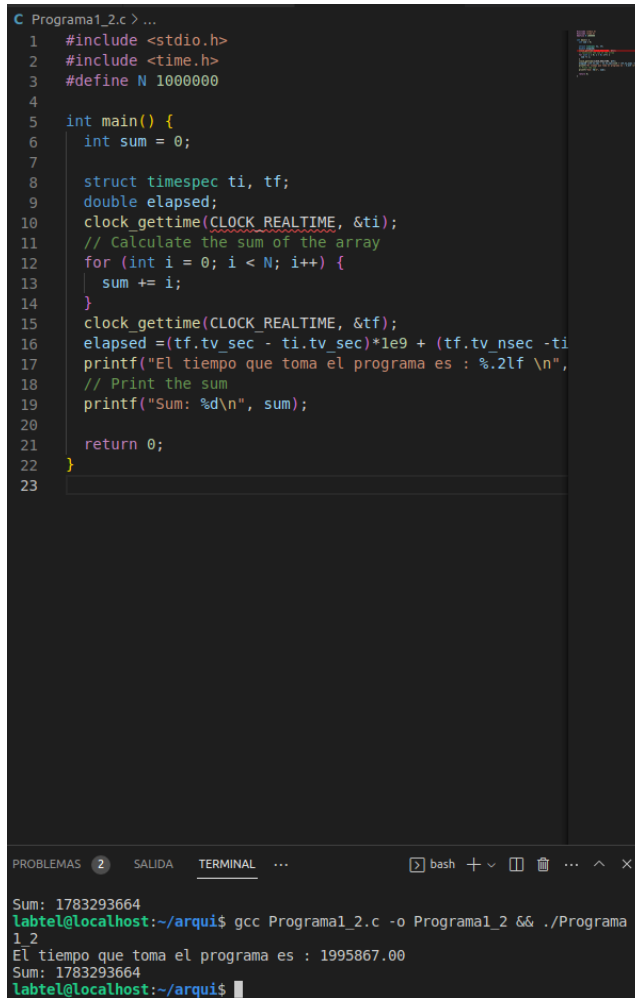
...

^

✖

```
labtel@localhost:~/arqui$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es : 1544578.00
Sum: 1783293664
labtel@localhost:~/arqui$
```

Para Programa1_2.c:



```
C Programa1_2.c > ...
1  #include <stdio.h>
2  #include <time.h>
3  #define N 1000000
4
5  int main() {
6      int sum = 0;
7
8      struct timespec ti, tf;
9      double elapsed;
10     clock_gettime(CLOCK_REALTIME, &ti);
11     // Calculate the sum of the array
12     for (int i = 0; i < N; i++) {
13         sum += i;
14     }
15     clock_gettime(CLOCK_REALTIME, &tf);
16     elapsed = (tf.tv_sec - ti.tv_sec)*1e9 + (tf.tv_nsec - ti.tv_nsec)*1e-6;
17     printf("El tiempo que toma el programa es : %.2lf \n", elapsed);
18     // Print the sum
19     printf("Sum: %d\n", sum);
20
21     return 0;
22 }
23
```

PROBLEMAS 2 SALIDA TERMINAL ...

```
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es : 1995867.00
Sum: 1783293664
labtel@localhost:~/arqui$
```

2. (1.0 punto) Realizar 15 ejecuciones desde el terminal para ambos archivos ejecutables. Realizar una captura de pantalla de su ejecución. Los tiempos en su imagen deben ser visibles.

Para Programa1_1.c:

```
Bienvenido C Programa1_1.c 1 X C Programa1_2.c 1
C Programa1_1.c > ...
1 #include <stdio.h>

PROBLEMAS 2 SALIDA TERMINAL ... bash + v ... ^ x

labtel@localhost:~/arqui$ gcc Programa1_1.c -o Programa1_1 && ./Programa
1_1
El tiempo que toma el programa es : 1755490.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Programa1_1.c -o Programa1_1 && ./Programa
1_1
El tiempo que toma el programa es : 1565592.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Programa1_1.c -o Programa1_1 && ./Programa
1_1
El tiempo que toma el programa es : 1659620.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Programa1_1.c -o Programa1_1 && ./Programa
1_1
El tiempo que toma el programa es : 1632298.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Programa1_1.c -o Programa1_1 && ./Programa
1_1
El tiempo que toma el programa es : 1852306.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Programa1_1.c -o Programa1_1 && ./Programa
1_1
El tiempo que toma el programa es : 1803719.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Programa1_1.c -o Programa1_1 && ./Programa
1_1
El tiempo que toma el programa es : 1974189.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Programa1_1.c -o Programa1_1 && ./Programa
1_1
El tiempo que toma el programa es : 1831030.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Programa1_1.c -o Programa1_1 && ./Programa
1_1
El tiempo que toma el programa es : 1812763.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Programa1_1.c -o Programa1_1 && ./Programa
1_1
El tiempo que toma el programa es : 1809479.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Programa1_1.c -o Programa1_1 && ./Programa
1_1
El tiempo que toma el programa es : 1831415.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Programa1_1.c -o Programa1_1 && ./Programa
1_1
El tiempo que toma el programa es : 1894664.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Programa1_1.c -o Programa1_1 && ./Programa
1_1
El tiempo que toma el programa es : 1781020.00
Sum: 1783293664
labtel@localhost:~/arqui$
```

```

labtel@localhost:~/arqui$ gcc Program1_1.c -o Program1_1 && ./Program1_1
El tiempo que toma el programa es : 1855967.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Program1_1.c -o Program1_1 && ./Program1_1
El tiempo que toma el programa es : 1936257.00
Sum: 1783293664
labtel@localhost:~/arqui$

```

Para el Programa1_2.c:

```

labtel@localhost:~/arqui$ gcc Program1_2.c -o Program1_2 && ./Program1_2
El tiempo que toma el programa es : 1840709.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Program1_2.c -o Program1_2 && ./Program1_2
El tiempo que toma el programa es : 1904375.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Program1_2.c -o Program1_2 && ./Program1_2
El tiempo que toma el programa es : 2056660.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Program1_2.c -o Program1_2 && ./Program1_2
El tiempo que toma el programa es : 1828476.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Program1_2.c -o Program1_2 && ./Program1_2
El tiempo que toma el programa es : 1945253.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Program1_2.c -o Program1_2 && ./Program1_2
El tiempo que toma el programa es : 1906654.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Program1_2.c -o Program1_2 && ./Program1_2
El tiempo que toma el programa es : 1948214.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Program1_2.c -o Program1_2 && ./Program1_2
El tiempo que toma el programa es : 2028449.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Program1_2.c -o Program1_2 && ./Program1_2
El tiempo que toma el programa es : 1646180.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Program1_2.c -o Program1_2 && ./Program1_2
El tiempo que toma el programa es : 2130388.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Program1_2.c -o Program1_2 && ./Program1_2
El tiempo que toma el programa es : 1732430.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Program1_2.c -o Program1_2 && ./Program1_2
El tiempo que toma el programa es : 1936443.00
Sum: 1783293664

```

```

labtel@localhost:~/arqui$ gcc Programa1_2.c -o Programa1_2 && ./Programa
1_2
El tiempo que toma el programa es : 2039238.00
Sum: 1783293664
labtel@localhost:~/arqui$ gcc Programa1_2.c -o Programa1_2 && ./Programa
1_2
El tiempo que toma el programa es : 2008881.00
Sum: 1783293664
labtel@localhost:~/arqui$
labtel@localhost:~/arqui$ gcc Programa1_2.c -o Programa1_2 && ./Programa
1_2
^[[AEl tiempo que toma el programa es : 2061175.00
Sum: 1783293664

```

3. (1.0 punto) Adjuntar una tabla con todas las mediciones.

Tiempos de ejecución del Programa1_1.c (en nanosegundos)	Tiempos de ejecución del Programa1_2.c (en nanosegundos)
1755490.00	1840709.00
1565592.00	1904375.00
1659620.00	2056660.00
1632298.00	1828476.00
1852306.00	1945253.00
1803719.00	1906654.00
1974189.00	1948214.00
1831030.00	2028449.00
1812763.00	1646180.00
1809479.00	2130388.00
1831415.00	1732430.00
1894664.00	1936443.00
1781020.00	2039238.00
1855967.00	2008881.00
1936257.00	2061175.00

4. (1.0 punto) Explicar cómo se relaciona el Programa1_1.c con los conceptos de localidad temporal y localidad espacial.

```
C Programa1_1 (2).c > ...
1  #include <stdio.h>
2
3  #define N 1000000
4
5  int main() {
6      int sum = 0;
7      int array[N];
8
9      // Initialize the array
10     for (int i = 0; i < N; i++) {
11         array[i] = i;
12     }
13
14     // Calculate the sum of the array
15     for (int i = 0; i < N; i++) {
16         sum += array[i];
17     }
18
19     // Print the sum
20     printf("Sum: %d\n", sum);
21
22     return 0;
23 }
```

En el Programa1_1.c , primero , se crea un array con elementos del 0 a N-1 (datos int) .Cada elemento del array se le asigna un valor y es guardado en la memoria principal y en la memoria caché. En esta primera parte del código, no se aprovechó la localidad espacial y temporal porque solo se asignaron y guardaron los valores.

Después, se actualizó el valor de sum sumándole los elementos del array con ayuda del for. En estos procesos, sí se aprovechó la localidad temporal y espacial. El uso de la localidad temporal se nota en la actualización constante del valor de sum en cada iteración porque después de la primera iteración la variable sum ya tenía un espacio en la memoria caché y este espacio permite reducir el tiempo de ejecución al estar ubicado más cerca del procesador. Por otro lado, los elementos del array que se le suman a la variable sum demuestran el aprovechamiento de la localidad espacial, porque cuando se invoca un elemento del array , se traslada a la memoria caché todo el bloque al cual pertenece (el resto del bloque está lleno con los siguientes elementos del arreglo). Luego, cuando se llama al siguiente elemento del array, ya no es necesario ir a la MP porque este elemento ya se encuentra en la memoria caché y esto demuestra un aprovechamiento de la localidad espacial.

En conclusión, el código aprovecha la localidad espacial y temporal de las variables.

5. (1.0 punto) Explicar cómo se relaciona el Programa1_2.c con los conceptos de localidad temporal y localidad espacial.

```
C Programa1_2 (2).c > ...
1  #include <stdio.h>
2
3  #define N 1000000
4
5  int main() {
6      int sum = 0;
7
8      // Calculate the sum of the array
9      for (int i = 0; i < N; i++) {
10         sum += i;
11     }
12
13     // Print the sum
14     printf("Sum: %d\n", sum);
15
16     return 0;
17 }
18
```

En el programa Programa1_2.c, se actualiza constantemente el valor de sum con ayuda del for. Esto demuestra un aprovechamiento de la localidad temporal porque en cada iteración se llama a la misma variable, así que después de la primera iteración esta variable ya tiene un espacio en la caché y para el resto de las iteraciones ya no se tendrá que buscar en la memoria principal sino en la caché ; por lo que esto nos ahorrará tiempo de ejecución. Por lo antes expuesto , el código se relaciona con la localidad temporal pero no con la espacial.

6. (1.0 punto) Utilizar el comando `getconf -a | grep CACHE` en el terminal y verificar el tamaño de bloque de su computador. Recordar que el tamaño de bloque en el nivel 1 se ve en la línea `LEVEL1_DCACHE_LINESIZE` (en bytes). Adjuntar captura de pantalla.

```

labtel@localhost:~/arqui$ getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE          32768
LEVEL1_ICACHE_ASSOC
LEVEL1_ICACHE_LINESIZE      64
LEVEL1_DCACHE_SIZE          32768
LEVEL1_DCACHE_ASSOC         8
LEVEL1_DCACHE_LINESIZE      64
LEVEL2_CACHE_SIZE           262144
LEVEL2_CACHE_ASSOC          4
LEVEL2_CACHE_LINESIZE       64
LEVEL3_CACHE_SIZE           16777216
LEVEL3_CACHE_ASSOC          16
LEVEL3_CACHE_LINESIZE       64
LEVEL4_CACHE_SIZE           0
LEVEL4_CACHE_ASSOC
LEVEL4_CACHE_LINESIZE
labtel@localhost:~/arqui$

```

Con lo obtenido con el comando `getconf -a | grep CACHE`, podemos determinar que el tamaño de los bloques de la caché del nivel 1 son de 64 bytes, el tamaño de la memoria caché del nivel 1 es 32768 bytes , la cantidad de bloques es 512 y conjuntos de 8 bloques .

7. (2.0 punto) Conociendo los datos previos, ¿es posible saber en qué momento se produce hit o miss? De ser posible, comentar el ejemplo, de no serlo, explicar porqué.

Sí ,es posible determinar cuando se produce hit o miss con los datos anteriores, ya que tenemos todas las características de la caché y con eso podemos determinar correctamente el mapeo asociativo (en este caso por 8 vías). Con el tamaño del bloque , podemos calcular la cantidad de bits del offset; con la cantidad de bloques y cantidad de bloques por conjunto , tenemos la cantidad de bits del set ; y con el tamaño tendríamos , el tag. Entonces ,el offset tiene 6 bits, el set tiene 6 bits y el tag saldría con el tamaño de la MP - offset - set .

Por ejemplo, para el programa1_2.c, en la primera iteración, se produce un miss porque el valor sum no está en la memoria caché . Luego , lleva el valor a la memoria caché (junto con todo el bloque) y en la siguiente iteración se produce un hit porque el valor de sum está almacenado en la caché gracias al miss de la anterior iteración. De tal manera, en las siguiente iteraciones , se producirán solo hits ya que en cada iteración se actualiza el valor de sum que está en la memoria caché.

Para el programa1_1.c , primero, se crea el array y el valor inicial de sum (estos están guardados en la MP). Luego, al asignarle valores a los elementos del array , se almacenan en la memoria caché y en la MP . Después ,se usa un for para actualizar el valor de sum. En la primera iteración se produce un miss y se lleva un bloque de la MP que contiene a sum y array[0: tamaño offset-2] para la siguientes “tamaño de offset - 1” iteraciones habrá hits ; para la siguiente iteración se produce un miss porque se perdió al elemento del array que continuaba al llevar el bloque anterior por ello se lleva el siguiente bloque y tendremos más hits hasta que nos falté un valor del array en la caché que se haya borrado en el anterior reemplazo. De cualquier manera, con las características que tenemos de la caché y su mapeo , se podrá determinar en cualquier caso cuando se produzca miss o hit para ambos programas.

8. (4.0 puntos) Basándose en todas las respuestas de los incisos anteriores, ¿qué implementación genera mejores tiempos de ejecución? Justificar su respuesta.

El programa1_1.c tiene un tiempo de ejecución promedio de 1 799 720,6 nseg y el programa1_2.c tiene un tiempo de ejecución promedio de 1 934235 nseg , podemos determinar a través de los tiempos de ejecución que el programa1_1.c es más rápido y genera mejores tiempo de ejecución además que el programa1_1.c aprovecha la localidad temporal y espacial a la vez para calcular la suma por lo cual tiene sentido que tenga tiempos de ejecución más pequeños y sea más rápido a comparación del programa1_2.c que solo aprovecha la localidad temporal.En conclusión , el programa1_1.c genera mejores tiempos de ejecución debido a su manejo de la localidad espacial y temporal

9. (3.0 puntos) ¿Cómo influye el tipo de datos del arreglo en este ejercicio? Esperaría resultados similares para un tipo char, short, long? Comentar acerca de su respuesta.

Los tipos de datos ocupan un espacio en la memoria principal y en la memoria caché , dependiendo del tamaño de los tipos de datos podremos almacenar más o menos variables en un solo bloque. Al almacenar más variables en un bloque, se podría aprovechar de mejor manera la localidad espacial de los datos porque cada que se produzca un fallo y se llevé el bloque a la memoria caché , se estarían llevando más datos y al llamarlos , se podría obtenerlos más rápido de la caché (se mejoran los tiempos de ejecución). Por otro lado, ocurre lo contrario cuando el tipo de dato es más grande ; se almacenan menos datos en un bloque y al ocurrir un fallo , llevar el bloque a la caché , ya tendríamos tantas variables en la caché como para utilizarlos y ahorrar tiempos de ejecución. Por tal motivo , en programación, el manejo correcto del tipo de datos permite una mejor performance del programa (mejores tiempos de ejecución). Entonces por lo antes expuesto, al usar datos tipo char (1 byte) y short (2 bytes) en el programa, se obtendrían menores tiempos de ejecución (el programa sería mucho más rápido) porque son tipos de datos más pequeños que los int (4 bytes). En cambio , los datos tipo long (8 bytes) presentan mayores tiempos de ejecución (el programa sería más lento) porque son datos de mayor tamaño comparado a los int .